Live Pattern Matching with Typed Holes

YONGWEI YUAN, Purdue University, USA
SCOTT GUEST, University of Michigan, USA
ERIC GRIFFIS, University of Michigan, USA
HANNAH POTTER, University of Washington, USA
DAVID MOON, University of Michigan, USA
CYRUS OMAR, University of Michigan, USA

Several modern programming systems, including GHC Haskell, Agda, Idris, and Hazel, support *typed holes*. Assigning static and, to varying degree, dynamic meaning to programs with holes allows program editors and other tools to offer meaningful feedback and assistance throughout editing, i.e. in a *live* manner. Prior work, however, has considered only holes appearing in expressions and types. This paper considers, from type theoretic and logical first principles, the problem of typed pattern holes. We confront two main difficulties, (1) statically reasoning about exhaustiveness and irredundancy when patterns are not fully known, and (2) live evaluation of expressions containing both pattern and expression holes. In both cases, this requires reasoning conservatively about all possible hole fillings. We develop a typed lambda calculus, Peanut, where reasoning about exhaustiveness and redundancy is mapped to the problem of deriving first order entailments. We equip Peanut with an operational semantics in the style of Hazelnut Live that allows us to evaluate around holes in both expressions and patterns. We mechanize the metatheory of Peanut in Agda and formalize a procedure capable of deciding the necessary entailments. Finally, we scale up and implement these mechanisms within Hazel, a programming environment for a dialect of Elm that automatically inserts holes during editing to provide static and dynamic feedback to the programmer in a maximally live manner, i.e. for every possible editor state. Hazel is the first maximally live environment for a general-purpose functional language.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Patterns**.

Additional Key Words and Phrases: pattern matching, typed holes

ACM Reference Format:
Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live Pattern Matching with Typed Holes. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 96 (April 2023), 27 pages. https://doi.org/10.1145/3586048

1 INTRODUCTION

Programming language definitions typically assign meaning to programs only once they are fully-formed and fully-typed. However, programming tools—type checkers, language-aware editors, interpreters, program synthesizers, and so on—are frequently asked to reason about and manipulate programs that are incomplete or erroneous. This can occur when the programmer has made a mistake, or when the programmer is simply in the midst of an editing task. These meaningless states are sometimes transient but they can also persist, e.g. through long refactoring tasks, causing

Authors' addresses: Yongwei Yuan, Purdue University, USA, yuan311@purdue.edu; Scott Guest, University of Michigan, USA, sguest@umich.edu; Eric Griffis, University of Michigan, USA, egriffis@umich.edu>; Hannah Potter, University of Washington, USA, hkpotter@cs.washington.edu; David Moon, University of Michigan, USA, dmoo@umich.edu; Cyrus Omar, University of Michigan, USA, comar@umich.edu.

2475-1421/2023/4-ART96
https://doi.org/10.1145/3586048

programming tools to flicker out of service or to turn to *ad hoc* heuristics, e.g. arbitrary token insertion, or deletion of problematic lines of code, to offer best-effort feedback and assistance [Bour et al. 2018; Kats et al. 2009; Omar et al. 2017b]. In brief, definitional gaps lead to gaps in service.

In recognition of this pernicious *gap problem*, several programming systems, including GHC Haskell [Peyton Jones et al. 2020], Agda [Norell 2007], Idris [Brady 2013], and Hazel [Omar et al. 2019, 2017a], have introduced *typed holes*. Typed holes come in two basic forms: *empty holes* stand for terms that have yet to be constructed, and *non-empty holes* operate as membranes around erroneous terms, e.g. as-yet-type-inconsistent expressions or as-yet-unbound variables, isolating them from the rest of the program. By incorporating holes into the syntax and semantics, it is possible to assign meaning to a greater number of notionally incomplete programs. Language services can thereby avoid gaps without needing to rely on *ad hoc* heuristics. Services can also be developed specifically for working with holes. For example, all of the systems mentioned above report the expected type and the variables in scope at each hole, and they are also able to synthesize hole fillings in various ways [Gissurarson 2018; Lubin et al. 2020].

In most of these systems the programmer manually inserts holes where necessary. Luckily, holes are syntactically lightweight: in GHC Haskell, for example, an unnamed empty expression hole is simply `_`, a named hole is `_name`, and non-empty holes can be inserted implicitly around static errors with an appropriate compiler flag. In Agda, programmers can express non-empty holes explicitly as `{e}n` where `e` is an expression and `n` is an identifying hole number.

The Hazel structure editor is distinct in that it inserts both empty and non-empty holes fully automatically during editing. For example, Fig. 2(a), discussed further below, shows an automatically inserted empty hole to the right of the `::` operator, numbered `98`. Hazel goes on to eliminate the gap problem entirely, maintaining a *maximal liveness invariant*: Hazel assigns both static and dynamic meaning to *every* possible editor state [Omar et al. 2019, 2017a]. This allows Hazel language services to remain fully functional at all times. This includes services that require program evaluation, because Hazel is capable of evaluating "around" expression holes, producing *indeterminate results* that retain holes [Omar et al. 2019]. Hazel also supports holes that appear in type annotations, including those where type inference is unable to find a solution, because Hazel is gradually typed [Siek et al. 2015]. Dynamic type errors (and other dynamic errors) are reformulated as run-time holes to localize their effect on evaluation [Omar et al. 2019].[1]

In all of the systems just described, holes can appear in expressions and types. None of these systems have previously supported holes in patterns. Pattern holes would, however, be useful for much the same reason as expression holes are useful: patterns are compositional in structure and are governed by a type discipline. In Hazel, our focus in this paper, pattern holes are in fact critical to scale up beyond the language in the prior work, which included only binary products and sums with primitive eliminators. Programmers will necessarily construct patterns incrementally[2] and Hazel must be able to assign meaning to each step to maintain its maximal liveness invariant.

While expressions and types are central to functional programming, patterns are also ubiquitous and pattern holes are far from trivial. Pattern matching can involve both a large number of patterns and individually large and complex patterns. For example, central to the Hazel editor implementation is a single match expression with 68 rules because it matches simultaneously on a pair of values (an action and a state). Several of these patterns have a compositional depth of 5 and span multiple lines of code. Entering individually well-typed patterns and collectively irredundant and exhaustive

[1] GHC Haskell can also run programs with expression holes, but the program crashes when a hole is reached. It also supports holes in type annotations, but the type inference system must be able to uniquely solve for these holes.

[2] In Agda, Hazel, and various other systems, the user can automatically generate an exhaustive set of patterns, but these match only on the outermost constructor and involve automatically generated variable names. Programmers often rearrange these incrementally into more deeply nested patterns.

```
match tree
| Node([]) -> Empty
| Node([x]) -> Node([f x, Empty])
| Node([x, y]) -> Node([f x, f y])
| Node(x::y::tl) -> Node(
  [f x, f (Node (y::tl))])
| Leaf x -> Leaf x
| Empty -> Empty
end
```

(a) Exhaustive + Irredundant

```
match tree
| Node(x::y::tl) -> Node(
  [f x, f (Node (y::tl))])
| Node([x, y]) -> Node([f x, f y])
| Node([x]) -> Node([f x, Empty])
| Node([]) -> Empty
| Empty -> Empty
end
```

(b) Inexhaustive + Redundant (Second Pattern)

Fig. 1. Two examples demonstrating structural pattern matching and common pitfalls.

sequences of patterns in non-trivial programs like this is not always straightforward, and it is easy to make mistakes, e.g. during a non-trivial refactoring. It is also useful to be able to test branches as they become complete, even when many remaining branches remain incomplete. In short, live feedback serves to surface problems as soon as they are certain to occur, rather than in an infrequently batched manner, which helps programmers diagnose problems early and maintain confidence in their mental model of program behavior [Tanimoto 2013].

This paper describes our integration of full-scale pattern matching (reviewed in Sec. 2) with support for pattern holes and live evaluation into Hazel (Sec. 3). We then distill out the essential ideas with a type-theoretic calculus equipped with a system of logical pattern constraints called Peanut, which extends the Hazelnut Live calculus of Omar et al. [2019] with pattern matching and pattern holes (Sec. 4). We have mechanized the metatheory of Peanut using Agda (Sec. 4.6 and Supplemental Material). We develop decision procedures in Sec. 4.7, describing (1) a simple decision procedure for the necessary logical entailments, and (2) how to express the entailments as an SMT problem for integration into more sophisticated static analyses. To go from Peanut to Hazel, we generalize it to support finite labeled sums, including sums with *datatype constructor holes* (Sec. 5). The result, which we integrate into Hazel, produces the first general-purpose functional programming environment that maintains maximal liveness.

2 BACKGROUND

Before discussing pattern holes, let us briefly review the necessary background and terminology, which will be familiar to users of functional languages. Briefly, *structural pattern matching* combines structural case analysis with destructuring binding. Patterns are compositional, so pattern matching can dramatically collapse what would otherwise need to be a deeply nested sequence of case analyses and destructurings. The central construct is the **match** expression, examples of which appear in Fig. 1. A match expression consists of a *scrutinee* and an ordered sequence of |-separated *rules*. Each rule consists of a *pattern* and a *branch expression* separated by ->. The value of the scrutinee is matched against each pattern in order, and if there is a match, the corresponding branch is taken, with each variable in the pattern bound to the corresponding matched value. The examples in Fig. 1 case analyze on the outer constructor of the value of the scrutinee, `tree`. In cases where the scrutinee was constructed by the application of the `Node` constructor, they simultaneously match on the structure of the list argument. Pattern variables and wildcard patterns, _, match any value, but the latter induces no binding.

Although superficially similar, Fig. 1a and Fig. 1b behave quite differently. In particular, the **match** expression in Fig. 1b is *inexhaustive*: there are values of the scrutinee's type, namely values of the form `Leaf n`, for which none of the patterns will match, leading to a run-time error or undefined behavior. Moreover, the second pattern in Fig. 1b is *redundant*: there are no values that

match `Node([x, y])` that do not also match some previous pattern in the rule sequence, here only `Node(x::y::tl)`, because `[x, y]` is syntactic sugar for `x::y::[]`. By contrast, Fig. 1a has no redundant patterns because `Node([x, y])` appears first. Subtleties like these are easy to miss, particularly for novices but even for experienced programmers when working with complex datatypes.

Fortunately, modern typed functional languages perform static analyses to detect inexhaustive rule sequences and redundant patterns within a rule sequence. Exhaustiveness checking compels programmers to consider all possible inputs, including rare cases that may lead to undesirable or undefined behavior. Indeed, many major security issues can be understood as a failure to exhaustively case analyze (e.g. null pointer exceptions). In the setting of a dependently-typed theorem prover, exhaustiveness checking is necessary to ensure totality and thus logical soundness. Exhaustiveness checking also supports program evolution: when extending datatype definitions with new constructors, exhaustiveness errors serve to alert the programmer of every **`match`** expression that needs to be updated to handle the new case, excepting those that use catch-all wildcard patterns, `_` (which for this reason are discouraged in functional programming practice). Redundancy checking similarly improves software quality by helping programmers avoid subtle order-related bugs and duplicated or unnecessary code paths.

3 LIVE PATTERN MATCHING IN HAZEL

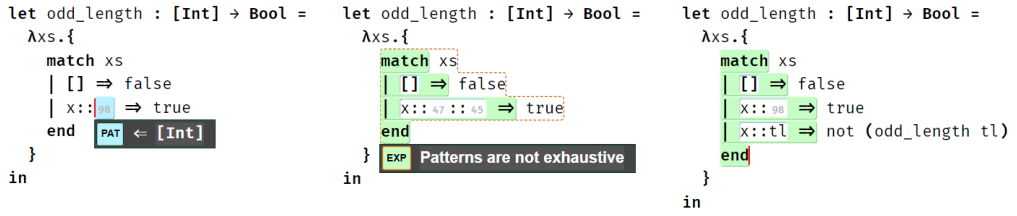
Adding holes to patterns is syntactically straightforward. In (our extension of) Hazel, pattern and expression holes look identical, represented by a gray, automatically generated numeric identifier (cf. pattern hole 40 and expression hole 38 in Fig. 4). In other systems with typed holes, we would need to take care to distinguish the syntax of pattern holes, perhaps `??`, from wildcard patterns, `_`. As we will see, wildcard patterns are semantically quite distinct from pattern holes.

The subtleties arise when we introduce pattern holes into (1) the static semantics and, in particular, into exhaustiveness and redundancy checking, and (2) the dynamic semantics of a system with support for live programming with holes *a la* Hazel. In both regards, the key concept is that a pattern hole represents an *unknown pattern*, so previously binary distinctions—between exhaustiveness and inexhaustiveness, redundancy and irredundancy, or run-time match and mismatch—become ternary distinctions: we need to consider *indeterminate* situations, i.e. where the determination cannot be made without filling one or more holes, including perhaps expression holes in the scrutinee. However, we also seek to avoid becoming unnecessarily conservative in situations where a determination *can* be made no matter how the holes are filled.

Let us consider several characteristic examples of each of these distinctions in turn. As a simple running example, suppose a programmer is writing a function `odd_length` which determines, by returning a `Boolean` value, whether an input list, of type `[Int]`, has an odd number of elements. Let us consider various intermediate editor states that the programmer may produce, and the live feedback that Hazel offers in each of these [Potter and Omar 2020].

3.1 Exhaustiveness Checking with Pattern Holes

We begin with the editor state in Fig. 2a, where there is a hole in the tail position of the cons (`::`) pattern. In determining whether this **`match`** expression is exhaustive, we must reason over all potential hole fillings. In this case, the **`match`** expression is *indeterminately exhaustive*, because there are hole fillings, such as a pattern variable `tl`, that would result in a determination of exhaustiveness, while there are other hole fillings, such as `[]` or `y::tl`, where the determination would instead be inexhaustiveness. In the Hazel user interface, we choose to alert the programmer with an error indicator only when the **`match`** expression is necessarily inexhaustive, so no error appears for this editor state. The motivation for this choice is that we do not want to draw the programmer's attention for a problem that may or may not actually persist once the user has filled the holes.



(a) Indeterminately Exhaustive (b) Necessarily Inexhaustive (c) Necessarily Exhaustive

Fig. 2. Exhaustiveness Checking with Pattern Holes

The cursor inspector does, however, provide typing information for the pattern hole when the programmer places their cursor there, as seen in Fig. 2a. This and other pattern holes can be typed using information about how they are used. In this example, Hazel can determine that the pattern hole must have type `[Int]` because that is the only valid type for the right-hand side of the cons pattern when we are matching on `xs`, which is of type `[Int]`.

While pattern holes can make it impossible to conclusively determine the exhaustiveness of a **`match`** expression, the mere presence of a pattern hole does not mean that exhaustiveness errors never arise. Consider now the editor state in Fig. 2b. In this case, although there are again pattern holes in the second pattern, it can be determined that this **`match`** expression is *necessarily inexhaustive* because no matter how those holes are filled, this **`match`** expression will fail to match lists with exactly one element (singleton lists). Since this **`match`** expression is inexhaustive no matter how the holes are filled, we alert the user with an error indicator and an error message when the cursor is on the **`match`** expression.

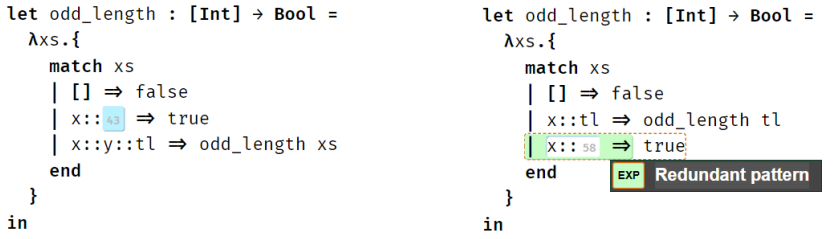
It is also possible for a **`match`** expression with pattern holes to be *necessarily exhaustive*, as seen in Fig. 2c. Here, the first and third pattern are exhaustive, regardless of how the hole in the second pattern is filled (e.g. with `[]` to special case singleton lists). We choose not to visually distinguish between necessarily and indeterminately exhaustive **`match`** expressions, again to avoid drawing the programmer's attention to information that is unlikely to be actionable, but the underlying semantic distinction is interesting to observe.

3.2 Redundancy Checking with Pattern Holes

A pattern is redundant if, for every value of the scrutinee's type that match that pattern, one of the preceding patterns in the **`match`** expression will necessarily match it. In the presence of pattern holes, we again have to consider all possible hole fillings in this analysis, leading to three possibilities: *necessarily irredundant*, *indeterminately irredundant*, and *necessarily redundant* patterns.

In the editor state in Fig. 3a, there is a pattern hole in the *second* pattern of the **`match`** expression. It is possible to fill this hole in a way that would make the *third* pattern irredundant, such as the empty list pattern, `[]`. However, it is also possible that the hole could be filled in a way that causes the third pattern to become redundant, such as the cons pattern `y::tl`. Consequently, the third pattern in this **`match`** expression is indeterminately irredundant (we chose "irredundant" rather than "redundant" in this phrase because, like exhaustiveness, irredundancy is the programmer's goal).

The second pattern itself, although it contains a pattern hole, is necessarily irredundant because no matter how the hole is filled, the first pattern, `[]`, does not overlap with it. The first pattern is always necessarily irredundant, even when it is a hole, because there are no previous patterns (except if the scrutinee has the nullary sum, i.e. `void`, type, which has no values; see Supplementary Material).



(a) Necessarily Irredundant (first two patterns) + Indeterminately Irredundant (third pattern)

(b) Necessarily Redundant (third pattern)

Fig. 3. Redundancy Checking with Pattern Holes

The editor state in Fig. 3b shows that a pattern with a hole can nevertheless be determined to be necessarily redundant. In this case, the previous pattern handles every non-empty list, so no matter how the pattern hole is filled, it will be redundant. Indeed, even an empty hole pattern in the third pattern would be redundant because the two preceding patterns are exhaustive. As with exhaustiveness, we only alert the user to necessarily redundant patterns, so this is the only pattern in Fig. 3 where an error is reported.

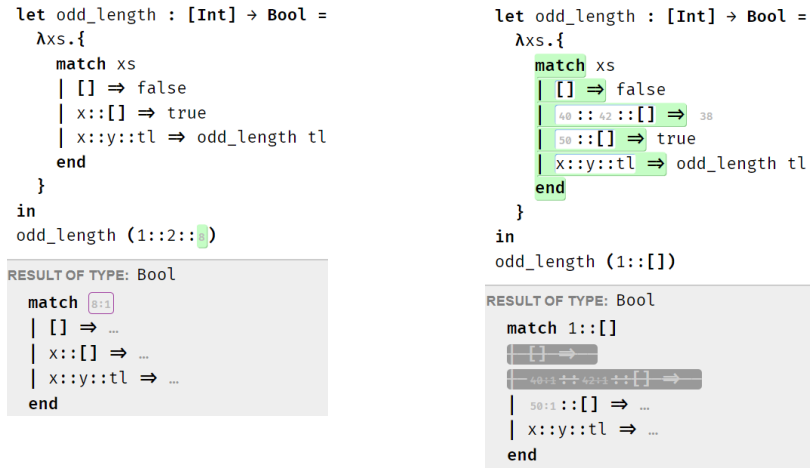
3.3 Live Evaluation with Expression and Pattern Holes

In the examples above, we considered only static analysis of programs with pattern holes. However, Hazel also supports live evaluation with holes, proceeding "around" holes as necessary to produce a result with a possibly *indeterminate value*, i.e. an expression that might retain holes in critical elimination positions [Omar et al. 2019].

In a language without holes, the value of the scrutinee can be determined to either *match* or *mismatch* every pattern of the same type. In order to support live evaluation with both expression and pattern holes, we have to consider now a third possiblilty: an *indeterminate match* between the scrutinee, which may itself have an indeterminate value, and a pattern. Let us consider the possibilities in Fig. 4.

We first look in Fig. 4a at the situation where there are expression holes but no pattern holes. We see in Fig. 4a that the argument to `odd_length` has a hole, so evaluation cannot determine a unique value. Instead, the argument has an *indeterminate value*. When we proceed through pattern matching, we can only make decisions that would be valid no matter how the hole in the tail of the list would be filled. We can determine that the first pattern, `[]`, necessarily mismatches since the argument is not empty. Similarly, we can also determine that the second pattern necessarily mismatches because regardless of how the hole is filled, the list is at least of length two. When checking the third pattern, we see that this matches any list of at least two elements. We know that the scrutinee necessarily matches this pattern, binding `tl` to whatever the hole is filled with, so we can take this branch. We next proceed to recurse, now with the hole itself as the argument. In the recursive call, we cannot determine whether the first pattern, `[]`, matches, since the empty expression hole could be filled with `[]` or with any other list. This indeterminate match leaves the entire **`match`** expression with an indeterminate value, and we cannot proceed any further as shown in the evaluation result at the bottom of Fig. 4a.

Next we consider in Fig. 4b the situation where there are (several) pattern holes encountered during evaluation. We know that the first pattern, `[]`, necessarily mismatches the scrutinee, `1::[]`, for the standard reasons. The second pattern, which contains two holes but which matches only lists of length two, also necessarily mismatches the scrutinee. The third pattern matches singleton lists, like the scrutinee. However, a pattern hole appears at the pattern's head, so it cannot be determined whether there is necessarily a match or a mismatch with `1::[]`. For example, if the pattern hole



(a) Pattern matching with expression holes (b) Pattern matching with pattern holes

Fig. 4. Live Evaluation with Expression and Pattern Holes

$$
\begin{array}{llll}
\text{Exp} & e & ::= & x \mid \underline{n} \\
& & \mid & \lambda x : \tau.e \mid e_1(e_2) \\
& & \mid & (e_1, e_2) \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \\
& & \mid & \mathtt{inl}_\tau(e) \mid \mathtt{inr}_\tau(e) \\
& & \mid & \mathtt{match}(e)\ \{\hat{rs}\} \\
& & \mid & (\!|\,|\!)^u \mid (\!|e|\!)^u
\end{array}
\qquad
\begin{array}{llll}
\text{Typ} & \tau & ::= & \mathsf{num} \mid (\tau_1 \to \tau_2) \mid (\tau_1 \times \tau_2) \mid (\tau_1 + \tau_2) \\
\text{ZRules} & \hat{rs} & ::= & (rs \mid r \mid rs) \\
\text{Rules} & rs & ::= & \cdot \mid (r \mid rs') \\
\text{Rule} & r & ::= & p \Rightarrow e \\
\text{Pat} & p & ::= & x \mid \_ \mid \underline{n} \mid (p_1, p_2) \\
& & \mid & \mathtt{inl}(p) \mid \mathtt{inr}(p) \mid (\!|\,|\!)^w \mid (\!|p|\!)^w_\tau
\end{array}
$$

Fig. 5. Syntax of Peanut. Here, n ranges over mathematical numbers (of any suitable sort), x over variables, u over expression hole names, and w over pattern hole names.

is filled with the pattern 2, then there would be a mismatch. Alternatively, if the pattern hole is filled with the pattern x, then there would be a match. This indeterminate match causes the **match** expression itself to have an indeterminate value. However, by graying out mismatched rules in the result, we can report to the programmer how far match evaluation was able to proceed, as shown at the bottom of Fig. 4b.

4 PEANUT: A TYPED PATTERN HOLE CALCULUS

We now formalize the mechanisms described in use in Sec. 3 with a core calculus, called Peanut, that supports pattern matching with typed holes in both expression and pattern position. We develop both a static semantics that enforces exhaustiveness and irredundancy as described in Sec. 3.1 and 3.2, and a dynamic semantics, based on the Hazelnut Live internal language [Omar et al. 2019], that allows us to engage in live programming with expression and pattern holes as described in Sec. 3.3.

Sec. 4.1 defines the syntax of Peanut and Sec. 4.2 defines its small-step operational semantics. Sec. 4.3, 4.4 and 4.5 define a corresponding static semantics as a type assignment system equipped with a match constraint language that we use in reasoning about exhaustiveness and redundancy in the presence of holes. Sec. 4.6 describes the Agda mechanization of Peanut's metatheory. Finally, Sec. 4.7 describes algorithmic procedures for exhaustiveness and redundancy checking.

$\boxed{(\hat{rs})^\diamond = rs}$ rs can be obtained by erasing the pointer from $\hat{rs}$

$$(\cdot \mid r \mid rs)^\diamond = r \mid rs$$
$$((r' \mid rs') \mid r \mid rs)^\diamond = r' \mid (rs' \mid r \mid rs)^\diamond$$

Fig. 6. Rule Pointer Erasure

4.1 Syntax

Fig. 5 presents the syntax of Peanut. Peanut is based on the internal language of Hazelnut Live, a typed lambda calculus featuring holes in expression position [Omar et al. 2019]. We choose numbers as the base type and add binary sums and binary products so that we have interesting patterns to consider. We also remove the machinery related to gradual typing (casts and failed casts) and expression hole closures to focus our attention on pattern matching in particular, though these features are retained in our Hazel implementation. Most forms are standard (we base our formulation of the basic constructs on Harper [2012]). We include functions, function application, pairs, explicit projection operators (for reasons we will consider below), and left and right injections, which are the introduction forms for sum types. Functions and injections include type annotations so that we can define a simple type assignment system. In our Hazel implementation, there is a corresponding bidirectionally typed external language where type annotations are not always necessary, given meaning as in Hazelnut Live by an elaboration process [Omar et al. 2019], but the details follow straightforwardly from the prior work, so we favor a minimal presentation here.

Empty expression holes are written $(\!|)^u$ and non-empty expression holes, which serve as membranes around type inconsistencies, are written $(\!|e|\!)^u$. Similarly, empty pattern holes are written $(\!|)^w$ and non-empty pattern holes, which are analogous, are written $(\!|p|\!)^w_\tau$. Here, u denotes the name of an expression hole, w denotes the name of a pattern hole, and τ is the type of the enclosed pattern p (which serves to ensure that the typing judgment in Sec. 4.4 is straightforwardly algorithmic). We assume that hole names correspond to unique holes in the external language, which we do not model here. We do not impose a uniqueness constraint, however, because we are defining an internal language so substitution can cause a hole can appear multiple times during evaluation.

A match expression, $\texttt{match}(e)\ \{\hat{rs}\}$, consists of a scrutinee, e, and a zipper of rules, $\hat{rs}$, which takes the form of a triple, $rs_{pre} \mid r \mid rs_{post}$, consisting of a prefix rule sequence, rs_{pre}, a current rule, r, and a suffix rule sequence, rs_{post}. In other words, it is a sequence of one or more rules with a pointer marking the rule currently being considered during evaluation (starting on the first rule). This zippered representation, which is unique to Peanut, is necessary to provide an intermediate result when the evaluation is stuck due to indeterminate pattern matching as shown in Fig. 4b. Specifically, rs_{pre} corresponds to the first two rules that are crossed out, r corresponds to the rule where pattern matching was indeterminate, and rs_{post} corresponds to the last rule that is yet to be considered. We define a pointer erasure operator $(\hat{rs})^\diamond$ in Fig. 6. Each rule, r, consists of a pattern, p, and a branch expression, e.

4.2 Live Operational Semantics

The dynamic semantics of Peanut extends Hazelnut Live [Omar et al. 2019] with support for pattern matching with holes. Hazelnut Live defines its dynamic semantics as a contextual operational semantics [Pientka and Dunfield 2008] capable of evaluating expressions with holes, whereas we choose a structural operational semantics [Plotkin 2004] for Peanut because it slightly simplifies the presentation of stepping through individual rules in a match expression. In this section, we

$\boxed{e\ \texttt{val}}$ e is a value

$$\frac{}{\underline{n}\ \texttt{val}}\ \text{VNum} \qquad \frac{}{\lambda x : \tau.e\ \texttt{val}}\ \text{VLam}$$

$$\frac{e_1\ \texttt{val} \qquad e_2\ \texttt{val}}{(e_1, e_2)\ \texttt{val}}\ \text{VPair}$$

$$\frac{e\ \texttt{val}}{\texttt{inl}_\tau(e)\ \texttt{val}}\ \text{VInl} \qquad \frac{e\ \texttt{val}}{\texttt{inr}_\tau(e)\ \texttt{val}}\ \text{VInr}$$

$\boxed{e\ \texttt{indet}}$ e is indeterminate

$$\frac{}{(\!|\,|\!)^u\ \texttt{indet}}\ \text{IEHole} \qquad \frac{e\ \texttt{final}}{(\!|e|\!)^u\ \texttt{indet}}\ \text{IHole} \qquad \cdots$$

$$\frac{e\ \texttt{final} \qquad e\ ?\ p_r}{\texttt{match}(e)\ \{rs_{pre} \mid (p_r \Rightarrow e_r) \mid rs_{post}\}\ \texttt{indet}}\ \text{IMatch}$$

$\boxed{e\ \texttt{final}}$ e is final

$$\frac{e\ \texttt{val}}{e\ \texttt{final}}\ \text{FVal} \qquad \frac{e\ \texttt{indet}}{e\ \texttt{final}}\ \text{FIndet}$$

Fig. 7. Final Expressions

$\boxed{e \mapsto e'}$ e takes a step to e'

$$\frac{e \mapsto e'}{\texttt{match}(e)\ \{\hat{rs}\} \mapsto \texttt{match}(e')\ \{\hat{rs}\}}\ \text{ITExpMatch}$$

$$\frac{e\ \texttt{final} \qquad e \rhd p_r \dashv\!\!| \theta}{\texttt{match}(e)\ \{rs_{pre} \mid (p_r \Rightarrow e_r) \mid rs_{post}\} \mapsto [\theta](e_r)}\ \text{ITSuccMatch}$$

$$\frac{e\ \texttt{final} \qquad e \perp p_r}{\texttt{match}(e)\ \{rs \mid (p_r \Rightarrow e_r) \mid (r' \mid rs')\} \mapsto \texttt{match}(e)\ \{(rs \mid (p_r \Rightarrow e_r) \mid \cdot)^\diamond \mid r' \mid rs'\}}\ \text{ITMisMatch}$$

Fig. 8. Stepping Match Expressions

present primarily the rules related to pattern matching (see the Supplemental Material for the full definition), as the rules for the other forms in Peanut follow directly from Hazelnut Live.

We say an expression is *final* if it cannot take any more steps. Fig. 7 presents the finality judgment, $e\ \texttt{final}$. A final expression can be either (1) a value, specified by $e\ \texttt{val}$, which is standard, or (2) an *indeterminate* expression, $e\ \texttt{indet}$, where the next step of evaluation would require filling one or more holes. For example, expression holes are indeterminate (Rules IEHole and IHole), as are function applications with indeterminate expressions in function position (omitted).

As with Hazelnut Live, evaluation proceeds around holes, taking those steps that do not depend on how the holes are filled. Fig. 8 presents the novel Rules of the stepping judgment, $e \mapsto e'$. ITExpMatch steps the scrutinee of a match expression and is applied until the scrutinee is final. Once the scrutinee is final, the match expression's rules are considered in turn by stepping through the zipper. At each step, one of three situations can arise, corresponding to Rules ITSuccMatch (a successful match), ITMisMatch (a mismatch), and IMatch (an indeterminate match). The second premise of each of these Rules invokes a corresponding judgment, presented in Fig. 9, relating the scrutinee with the pattern being considered. Exactly one of these three judgments holds, given a well-typed expression and a pattern of the same type (see Sec. 4.4).

Lemma 4.1 (Matching Determinism). *If $e\ \texttt{final}$ and $\cdot ; \Delta \vdash e : \tau$ and $\Delta \vdash p : \tau \dashv\!\!| \Gamma$ then exactly one of the following holds: (1) $e \rhd p \dashv\!\!| \theta$ for some θ; (2) $e\ ?\ p$; or (3) $e \perp p$.*

$\boxed{e \rhd p \dashv\!\mid \theta}$ e matches p, emitting θ

$$\frac{}{e \rhd x \dashv\!\mid e/x}\ \text{MVar} \qquad \frac{}{e \rhd \_ \dashv\!\mid \cdot}\ \text{MWild} \qquad \frac{}{\underline{n} \rhd \underline{n} \dashv\!\mid \cdot}\ \text{MNum} \qquad \frac{e \rhd p \dashv\!\mid \theta}{\mathtt{inl}_\tau(e) \rhd \mathtt{inl}(p) \dashv\!\mid \theta}\ \text{MInl}$$

$$\frac{e \rhd p \dashv\!\mid \theta}{\mathtt{inr}_\tau(e) \rhd \mathtt{inr}(p) \dashv\!\mid \theta}\ \text{MInr} \qquad \frac{e_1 \rhd p_1 \dashv\!\mid \theta_1 \qquad e_2 \rhd p_2 \dashv\!\mid \theta_2}{(e_1, e_2) \rhd (p_1, p_2) \dashv\!\mid \theta_1 \uplus \theta_2}\ \text{MPair}$$

$$\frac{e\ \mathtt{notintro} \qquad \mathtt{fst}(e) \rhd p_1 \dashv\!\mid \theta_1 \qquad \mathtt{snd}(e) \rhd p_2 \dashv\!\mid \theta_2}{e \rhd (p_1, p_2) \dashv\!\mid \theta_1 \uplus \theta_2}\ \text{MNotIntroPair}$$

$\boxed{e \perp p}$ e does not match p

$$\frac{n_1 \neq n_2}{\underline{n_1} \perp \underline{n_2}}\ \text{NMNum} \qquad \frac{e_1 \perp p_1}{(e_1, e_2) \perp (p_1, p_2)}\ \text{NMPairL} \qquad \frac{e_2 \perp p_2}{(e_1, e_2) \perp (p_1, p_2)}\ \text{NMPairR}$$

$$\frac{}{\mathtt{inr}_\tau(e) \perp \mathtt{inl}(p)}\ \text{NMConfL} \qquad \frac{}{\mathtt{inl}_\tau(e) \perp \mathtt{inr}(p)}\ \text{NMConfR} \qquad \frac{e \perp p}{\mathtt{inl}_\tau(e) \perp \mathtt{inl}(p)}\ \text{NMInl}$$

$$\frac{e \perp p}{\mathtt{inr}_\tau(e) \perp \mathtt{inr}(p)}\ \text{NMInr}$$

$\boxed{e\ ?\ p}$ e indeterminately matches p

$$\frac{}{e\ ?\ (\!|\,|\!)^w}\ \text{MMEHole} \qquad \frac{}{e\ ?\ (\!|p|\!)^w_\tau}\ \text{MMHole} \qquad \frac{e_1\ ?\ p_1 \qquad e_2 \rhd p_2 \dashv\!\mid \theta_2}{(e_1, e_2)\ ?\ (p_1, p_2)}\ \text{MMPairL}$$

$$\frac{e_1 \rhd p_1 \dashv\!\mid \theta_1 \qquad e_2\ ?\ p_2}{(e_1, e_2)\ ?\ (p_1, p_2)}\ \text{MMPairR} \qquad \frac{e_1\ ?\ p_1 \qquad e_2\ ?\ p_2}{(e_1, e_2)\ ?\ (p_1, p_2)}\ \text{MMPair} \qquad \frac{e\ ?\ p}{\mathtt{inl}_\tau(e)\ ?\ \mathtt{inl}(p)}\ \text{MMInl}$$

$$\frac{e\ ?\ p}{\mathtt{inr}_\tau(e)\ ?\ \mathtt{inr}(p)}\ \text{MMInr} \qquad \frac{e\ \mathtt{notintro} \qquad p\ \mathtt{refutable?}}{e\ ?\ p}\ \text{MMNotIntro}$$

Fig. 9. The three possible outcomes of pattern matching

From this lemma follow the determinism and progress theorems governing the stepping judgment, which are combined below.

Theorem 4.2 (Deterministic Progress). *If $\cdot; \Delta \vdash e : \tau$ then exactly one of the following holds: (1) e $\mathtt{val}$; (2) e $\mathtt{indet}$; or (3) $e \mapsto e'$ for some unique e'.*

Let us now consider each of the three possible outcomes in more detail.

The judgment $e \rhd p \dashv\!\mid \theta$ denotes a successful match as witnessed by the substitution θ over the variables bound in p. This substitution is applied, written $[\theta](e_r)$, when the corresponding branch is taken in the conclusion of ITSuccMatch. Notice that for an irrefutable pattern like x and _, all scrutinees match, including indeterminate scrutinees. Pair patterns are an interesting case. If the scrutinee is a pair, then MPair matches the corresponding expressions and patterns. However, the scrutinee may also be an indeterminate expression of product type that is not syntactically a pair.

$\boxed{e\ \texttt{notintro}}$ e cannot be a value syntactically

$$\frac{}{(\!|\,|\!)^u\ \texttt{notintro}}\ \text{NVEHole} \qquad \frac{}{(\!|e|\!)^u\ \texttt{notintro}}\ \text{NVHole} \qquad \frac{}{e_1(e_2)\ \texttt{notintro}}\ \text{NVAp}$$

$$\frac{}{\texttt{match}(e)\ \{\hat{rs}\}\ \texttt{notintro}}\ \text{NVMatch} \qquad \frac{}{\texttt{fst}(e)\ \texttt{notintro}}\ \text{NVFst} \qquad \frac{}{\texttt{snd}(e)\ \texttt{notintro}}\ \text{NVSnd}$$

Fig. 10. Expressions that are syntactically not values

Because pairs are the only introduction form of product type, we know that no matter how the scrutinee is eventually completed, it will eventually become a pair, so we can successfully match even in this situation. The MNotIntroPair rule handles this case by taking the first and second projections of the scrutinee in any case where the scrutinee is not an introduction form of the language, defined by the judgment $e\ \texttt{notintro}$ in Fig. 10.

The judgment $e \perp p$ denotes that e does not match p. In the case of such a mismatch, ITMisMatch moves the focus of the zipper to the next rule. Note that if there are no remaining rules, the operational semantics are undefined (i.e. we do not define run-time match failure judgmentally). This does not violate Theorem 4.2 because in our static semantics in Sec. 4.4, exhaustiveness will be a requirement. In settings where exhaustiveness checking only generates a warning, it would be straightforward (but tedious) to add match failure errors and their corresponding propagation rules to the semantics.

$\boxed{p\ \texttt{refutable}_?}$ p is indeterminately refutable

$$\frac{}{(\!|\,|\!)^w\ \texttt{refutable}_?}\ \text{REHole} \qquad \frac{}{(\!|p|\!)^w_\tau\ \texttt{refutable}_?}\ \text{RHole} \qquad \frac{}{\underline{n}\ \texttt{refutable}_?}\ \text{RNum}$$

$$\frac{}{\texttt{inl}(p)\ \texttt{refutable}_?}\ \text{RInl} \qquad \frac{}{\texttt{inr}(p)\ \texttt{refutable}_?}\ \text{RInr} \qquad \frac{p_1\ \texttt{refutable}_?}{(p_1, p_2)\ \texttt{refutable}_?}\ \text{RPairL}$$

$$\frac{p_2\ \texttt{refutable}_?}{(p_1, p_2)\ \texttt{refutable}_?}\ \text{RPairR}$$

Fig. 11. Indeterminately Refutable Patterns

Finally, the judgment $e\ ?\ p$ indicates an *indeterminate match* due to the presence of holes in e or p. IMatch designates the entire match expression indeterminate once it has stepped to a rule where there is such an indeterminate match. MMEHole and MMHole specify that pattern holes always indeterminate match any scrutinee, because the hole has not been filled. MMPairL, MMPairR, and MMPair specify that pairs indeterminately match as long as at least one of the corresponding projections of the scrutinee indeterminately match and neither mismatches. MMInl and MMInr specify that matching injections indeterminately match if the arguments indeterminately match. Finally, MMNotIntro handles all situations where the scrutinee is not an introduction form (e.g. a hole or indeterminate function application), as specified in Fig. 10, and p is indeterminately refutable, i.e. it is not a variable, wildcard, or pair with irrefutable patterns on both sides, as specified by the

$$\text{Constraint} \quad \xi \quad ::= \quad \top \mid \bot \mid ? \mid \underline{n} \mid \underline{\not n} \mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2 \mid \texttt{inl}(\xi) \mid \texttt{inr}(\xi) \mid (\xi_1, \xi_2)$$

$\boxed{\xi : \tau}$ ξ is a constraint on final expressions of type τ $\qquad$ $\boxed{\overline{\xi_1} = \xi_2}$ dual of ξ_1 is ξ_2

$$\frac{}{\top : \tau}\ \text{CTTruth} \qquad \frac{}{\bot : \tau}\ \text{CTFalsity} \qquad \frac{}{? : \tau}\ \text{CTUnknown}$$

$$\frac{}{\underline{n} : \texttt{num}}\ \text{CTNum} \qquad \frac{}{\underline{\not n} : \texttt{num}}\ \text{CTNotNum}$$

$$\frac{\xi_1 : \tau \qquad \xi_2 : \tau}{\xi_1 \wedge \xi_2 : \tau}\ \text{CTAnd} \qquad \frac{\xi_1 : \tau \qquad \xi_2 : \tau}{\xi_1 \vee \xi_2 : \tau}\ \text{CTOr}$$

$$\frac{\xi_1 : \tau_1}{\texttt{inl}(\xi_1) : (\tau_1 + \tau_2)}\ \text{CTInl} \qquad \frac{\xi_2 : \tau_2}{\texttt{inr}(\xi_2) : (\tau_1 + \tau_2)}\ \text{CTInr}$$

$$\frac{\xi_1 : \tau_1 \qquad \xi_2 : \tau_2}{(\xi_1, \xi_2) : (\tau_1 \times \tau_2)}\ \text{CTPair}$$

$$\begin{aligned}
\overline{\top} &= \bot \\
\overline{\bot} &= \top \\
\overline{\underline{n}} &= \underline{\not n} \\
\overline{\underline{\not n}} &= \underline{n} \\
\overline{\xi_1 \wedge \xi_2} &= \overline{\xi_1} \vee \overline{\xi_2} \\
\overline{\xi_1 \vee \xi_2} &= \overline{\xi_1} \wedge \overline{\xi_2} \\
\overline{\texttt{inl}(\xi_1)} &= \texttt{inl}(\overline{\xi_1}) \vee \texttt{inr}(\top) \\
\overline{\texttt{inr}(\xi_2)} &= \texttt{inr}(\overline{\xi_2}) \vee \texttt{inl}(\top) \\
\overline{(\xi_1, \xi_2)} &= (\xi_1, \overline{\xi_2}) \vee (\overline{\xi_1}, \xi_2) \vee (\overline{\xi_1}, \overline{\xi_2})
\end{aligned}$$

Fig. 12. Match Constraints

judgment p $\texttt{refutable}_?$ defined in Fig. 11. (Irrefutable patterns lead to matches, as described above, so failing to exclude these would violate Lemma 4.1.)

4.3 Match Constraint Language

We can now move on in this and the remaining subsections to the static semantics of Peanut. Before continuing to describe the typing judgments in Sec. 4.4, we must introduce *match constraints*, ξ, which generalize patterns to form a simple first-order logic. Constraints are generated from patterns and rules during typechecking and are the structures on which we will perform exhaustiveness and redundancy checking.

Fig. 12 defines the syntax of the match constraint language. Excluding the unknown constraint ? for a moment, a constraint ξ identifies a satisfying subset of final expressions of some type τ as specified by the constraint typing judgment, $\xi : \tau$, in Fig. 12. Assuming $\xi : \tau$, then the dual of ξ, denoted by $\overline{\xi}$ and defined also in Fig. 12, represents the complement of the subset identified by ξ under the set of final expressions of type τ. We must also model indeterminacy in our constraint language. So just as there are three possible outcomes of matching a final expression against a pattern, there are three possible relationships between a final expression e and a constraint ξ, assuming they can be checked against the same type—e necessarily satisfies ξ, e indeterminately satisfies ξ, and e necessarily does not satisfy ξ. Fig. 13 defines the satisfaction judgments that encode such relationships. Assuming final expression e and constraint ξ are of the same type, the judgment $e \models \xi$ specifies that e satisfies ξ while the judgment $e \models_? \xi$ specifies that e indeterminately satisfies ξ. Both judgments closely follow the definition of pattern matching judgments $e \rhd p \dashv\!\mid \theta$ and $e \; ? \; p$, adding disjunction and conjunction. The main differences of note are in Rule CSNotIntroPair and Rule CMSNotIntro. The refutability premise follows the one in the dynamics (the definition of ξ $\texttt{refutable}_?$ is in the Supplementary Material). The CMSNotIntro rule includes a third premise, ξ $\texttt{possible}$ (defined in Fig. 14). This is due to the presence of a bottom constraint, $\bot$, which does

$\boxed{e \models \xi}$ e necessarily satisfies ξ

$$\frac{}{e \models \top}\ \text{CSTruth} \qquad \frac{}{\underline{n} \models \underline{n}}\ \text{CSNum} \qquad \frac{n_1 \neq n_2}{\underline{n_1} \models \cancel{\underline{n_2}}}\ \text{CSNotNum} \qquad \frac{e \models \xi_1 \qquad e \models \xi_2}{e \models \xi_1 \wedge \xi_2}\ \text{CSAnd}$$

$$\frac{e \models \xi_1}{e \models \xi_1 \vee \xi_2}\ \text{CSOrL} \qquad \frac{e \models \xi_2}{e \models \xi_1 \vee \xi_2}\ \text{CSOrR} \qquad \frac{e_1 \models \xi_1}{\texttt{inl}_{\tau_2}(e_1) \models \texttt{inl}(\xi_1)}\ \text{CSInl}$$

$$\frac{e_2 \models \xi_2}{\texttt{inr}_{\tau_1}(e_2) \models \texttt{inr}(\xi_2)}\ \text{CSInr} \qquad \frac{e_1 \models \xi_1 \qquad e_2 \models \xi_2}{(e_1, e_2) \models (\xi_1, \xi_2)}\ \text{CSPair}$$

$$\frac{e\ \texttt{notintro} \qquad \texttt{fst}(e) \models \xi_1 \qquad \texttt{snd}(e) \models \xi_2}{e \models (\xi_1, \xi_2)}\ \text{CSNotIntroPair}$$

$\boxed{e \models_? \xi}$ e indeterminately satisfies ξ

$$\frac{}{e \models_? ?}\ \text{CMSUnknown} \qquad \frac{e\ \texttt{notintro} \qquad \xi\ \texttt{refutable}_? \qquad \xi\ \texttt{possible}}{e \models_? \xi}\ \text{CMSNotIntro}$$

$$\frac{e \models_? \xi_1 \qquad e \models \xi_2}{e \models_? \xi_1 \wedge \xi_2}\ \text{CMSAndL} \qquad \frac{e \models \xi_1 \qquad e \models_? \xi_2}{e \models_? \xi_1 \wedge \xi_2}\ \text{CMSAndR}$$

$$\frac{e \models_? \xi_1 \qquad e \models_? \xi_2}{e \models_? \xi_1 \wedge \xi_2}\ \text{CMSAnd} \qquad \frac{e \models_? \xi_1 \qquad e \not\models \xi_2}{e \models_? \xi_1 \vee \xi_2}\ \text{CMSOrL} \qquad \frac{e \not\models \xi_1 \qquad e \models_? \xi_2}{e \models_? \xi_1 \vee \xi_2}\ \text{CMSOrR}$$

$$\frac{e_1 \models_? \xi_1}{\texttt{inl}_{\tau_2}(e_1) \models_? \texttt{inl}(\xi_1)}\ \text{CMSInl} \qquad \frac{e_2 \models_? \xi_2}{\texttt{inr}_{\tau_1}(e_2) \models_? \texttt{inr}(\xi_2)}\ \text{CMSInr}$$

$$\frac{e_1 \models_? \xi_1 \qquad e_2 \models \xi_2}{(e_1, e_2) \models_? (\xi_1, \xi_2)}\ \text{CMSPairL} \qquad \frac{e_1 \models \xi_1 \qquad e_2 \models_? \xi_2}{(e_1, e_2) \models_? (\xi_1, \xi_2)}\ \text{CMSPairR}$$

$$\frac{e_1 \models_? \xi_1 \qquad e_2 \models_? \xi_2}{(e_1, e_2) \models_? (\xi_1, \xi_2)}\ \text{CMSPair}$$

$\boxed{e \models^\dagger_? \xi}$ e necessarily or indeterminately satisfies ξ

$$\frac{e \models_? \xi}{e \models^\dagger_? \xi}\ \text{CSMSMay} \qquad \frac{e \models \xi}{e \models^\dagger_? \xi}\ \text{CSMSSat}$$

Fig. 13. Satisfaction

not correspond to a pattern but rather is the dual of the top constraint, $\top$, which corresponds to variable and wildcard patterns. ξ `possible` rules out the cases where ξ contains $\bot$. ξ $\texttt{refutable}_?$ and ξ `possible` together says that constraint ξ is neither necessarily irrefutable nor necessarily

$\boxed{\xi\ \texttt{possible}}$ ξ is possible

$$\frac{}{\top\ \texttt{possible}}\ \text{PTruth} \qquad \frac{}{?\ \texttt{possible}}\ \text{PUnknown} \qquad \frac{}{\underline{n}\ \texttt{possible}}\ \text{PNum}$$

$$\frac{}{\not{\underline{n}}\ \texttt{possible}}\ \text{PNotNum} \qquad \frac{\xi\ \texttt{possible}}{\texttt{inl}(\xi)\ \texttt{possible}}\ \text{PInl} \qquad \frac{\xi\ \texttt{possible}}{\texttt{inr}(\xi)\ \texttt{possible}}\ \text{PInr}$$

$$\frac{\xi_1\ \texttt{possible} \qquad \xi_2\ \texttt{possible}}{(\xi_1, \xi_2)\ \texttt{possible}}\ \text{PPair} \qquad \frac{\xi_1\ \texttt{possible}}{\xi_1 \vee \xi_2\ \texttt{possible}}\ \text{POrL} \qquad \frac{\xi_2\ \texttt{possible}}{\xi_1 \vee \xi_2\ \texttt{possible}}\ \text{POrR}$$

Fig. 14. Possible Constraints

refuted. The judgment $e \models^{\dagger}_{?} \xi$ is simply the disjunction of necessary and indeterminate satisfaction. We say e does not satisfy ξ when $e \models^{\dagger}_{?} \xi$ is not derivable, written $e \not\models^{\dagger}_{?} \xi$.

Lemma 4.3 establishes a correspondence between pattern matching results and satisfaction judgments, which makes reasoning about pattern matching in the type system possible and helps prove Preservation (Theorem 4.7). Here, $\Delta \vdash p : \tau[\xi] \dashv\Gamma$ not only tells us that the pattern p is of the same type as the scrutinee, but also emits to a constraint ξ, and the judgment will be properly defined in Sec. 4.4.

Lemma 4.3 (Matching Coherence of Constraint)**.** *If* $\cdot\,;\Delta_e \vdash e : \tau$ *and* e `final` *and* $\Delta \vdash p : \tau[\xi] \dashv\Gamma$ *then (1)* $e \models \xi$ *iff* $e \rhd p \dashv\theta$ *for some* θ*; and (2)* $e \models_? \xi$ *iff* $e\ ?\ p$*; and (3)* $e \not\models^{\dagger}_{?} \xi$ *iff* $e \perp p$*.*

And the exclusiveness of satisfaction judgments directly follows from the exclusiveness of pattern matching (Lemma 4.1).

Theorem 4.4 (Exclusiveness of Constraint Satisfaction)**.** *If* $\xi : \tau$ *and* $\cdot\,;\Delta \vdash e : \tau$ *and* e `final` *then exactly one of the following holds: (1)* $e \models \xi$*; or (2)* $e \models_? \xi$*; or (3)* $e \not\models^{\dagger}_{?} \xi$*.*

Exhaustiveness and redundancy checking can be reduced to the problem of deciding *entailments* between two constraints. We need to define two kinds of entailment: indeterminate entailment and potential entailment.

Definition 4.5 (Indeterminate Entailment of Constraints)**.** *Suppose that* $\xi_1 : \tau$ *and* $\xi_2 : \tau$*. Then* $\xi_1 \models \xi_2$ *iff for all* e *such that* $\cdot\,;\Delta \vdash e : \tau$ *and* e `val` *we have* $e \models^{\dagger}_{?} \xi_1$ *implies* $e \models \xi_2$*.*

Indeterminate entailment (Defn. 4.5) allows us to reason about redundancy, namely we need $\xi_r \not\models \xi_{pre}$ to ensure the constraint for the branch r of interest is not redundant with respect to its preceding branches, conjoined in rs_{pre} (see Rules TOneRules and TRules in the next section). The branch r would never be taken only when for all values that determinately or indeterminately match the pattern in r, they determinately match the pattern in one of the previous branches.

Definition 4.6 (Potential Entailment of Constraints)**.** *Suppose that* $\xi_1 : \tau$ *and* $\xi_2 : \tau$*. Then* $\xi_1 \models^{\dagger}_{?} \xi_2$ *iff for all* e *such that* $\cdot\,;\Delta \vdash e : \tau$ *and* e `final` *we have* $e \models^{\dagger}_{?} \xi_1$ *implies* $e \models^{\dagger}_{?} \xi_2$*.*

Potential entailment (Defn. 4.6) will be used for exhaustiveness checking. In particular, if we choose ξ_1 to be $\top$, then the entailment $\top \models^{\dagger}_{?} \xi$ ensures that ξ is either necessarily or indeterminately exhaustive (we will see this appear in Rules TMatchZPre and TMatchNZPre). The type checker should give an error/warning about the inexhaustiveness only when a case must have been missed,

i.e. when there is a value that does not match any rule's pattern, no matter how the programmer may fill the holes in the program. That is, a match expression is indeterminately exhaustive if all values indeterminately match one of its constituent rules' patterns.

We consider final expressions in the definition of potential entailment, rather than values as with indeterminate entailment. In this way, even when the scrutinee e is indeterminate, we can still be confident that either one of the branches will be taken or evaluation will stop on one branch due to the indeterminacy of pattern matching with holes. As we will later show in Sec. 4.5, this is equivalent to quantifying over all potential values of a final expression.

4.4 Typing, Exhaustiveness, and Irredundancy

$\boxed{\Gamma\,;\Delta \vdash e : \tau}$ e is of type τ

$$\frac{\Gamma\,;\Delta \vdash e : \tau \qquad \Gamma\,;\Delta \vdash [\bot]r \mid rs : \tau[\xi] \Rightarrow \tau' \qquad \top \models^{\dagger}_{?} \xi}{\Gamma\,;\Delta \vdash \texttt{match}(e)\ \{\cdot \mid r \mid rs\} : \tau'}\ \text{TMatchZPre}$$

$$\frac{\begin{array}{c}\Gamma\,;\Delta \vdash e : \tau \qquad e\ \texttt{final} \qquad \Gamma\,;\Delta \vdash [\bot]rs_{pre} : \tau[\xi_{pre}] \Rightarrow \tau' \\ \Gamma\,;\Delta \vdash [\xi_{pre}]r \mid rs_{post} : \tau[\xi_{rest}] \Rightarrow \tau' \qquad e \not\models^{\dagger}_{?} \xi_{pre} \qquad \top \models^{\dagger}_{?} \xi_{pre} \vee \xi_{rest}\end{array}}{\Gamma\,;\Delta \vdash \texttt{match}(e)\ \{rs_{pre} \mid r \mid rs_{post}\} : \tau'}\ \text{TMatchNZPre}$$

Fig. 15. Typing Rules for Match Expressions

We can now describe the type system of Peanut, which utilizes entailments between constraints to enforce exhaustiveness and irredundancy. Specifically, we build a similar type system to Omar et al. [2019] by defining typing judgments for both expressions and patterns. The expression typing judgment, $\Gamma\,;\Delta \vdash e : \tau$, is defined in Fig. 15. It takes both a variable context Γ and a hole context Δ as input, and produces a type as output. The pattern typing judgment, $\Delta \vdash p : \tau[\xi] \dashv\Vdash \Gamma$, is defined in Fig. 16. It takes a hole context Δ and a type as input and produces a variable context Γ and a constraint, ξ. Fig. 16 also defines typing judgments for rules, which we consider below.

4.4.1 Typing of Expressions and Exhaustiveness Checking. We start by specifying the typing of expressions in Fig. 15, giving particular attention to match expressions. We will see that our definitions enforce exhaustiveness of the constituent branches of any match expression. Alternatively, one could include exhaustiveness as a separate judgment, modifying the statement of type safety appropriately to include the requirement that all match expressions in an expression are exhaustive, as is done in our Agda mechanization, but we integrate them for simplicity in our presentation.

Rule TMatchZPre corresponds to the case when we are yet to start pattern matching, i.e. the rules zipper has no prefix. The first premise specifies that the scrutinee e is of type τ, and the second premise specifies that the constituent rules $r \mid rs$ are not only well-typed but also transform a final expression of the same type as the scrutinee, into a final expression of type τ'. Typing of a sequence of rules takes an initial constraint as input and produces a final constraint for the rules, which is the disjunction of the constituent constraints (see Fig. 16). As the inductive base case, we supply $\bot$ as the initial constraint (it can be dropped from any disjunction without changing its meaning). The third premise $\top \models^{\dagger}_{?} \xi$ checks for necessary or indeterminate exhaustiveness. In other words, for a well-typed match expression, it is impossible that the scrutinee would determinately fail to match all the patterns as we consider branches rs in order. Intuitively, constraint ξ encodes the set of possible expressions that determinately or indeterminately match any of the branches and the entailment from $\top$ ensures that all expressions are considered.

Rule TMatchNZPre corresponds to the case that we have already started pattern matching and have already considered preceding rules rs_{pre}. First of all, the scrutinee should not only be well-typed but also be final. Next, in addition to ensuring the exhaustiveness of the constituent rules, we want to make sure that at least one of the remaining rules r or rs_{post} would be taken. Note that only when the final scrutinee e determinately mismatches the pattern p, i.e., $e \perp p$, can we move the rule pointer. By Lemma 4.1, for any pattern p in the preceding branches, neither $e \rhd p \dashv\!\mid \theta$ nor $e \,?\, p$ is derivable. Then by Lemma 4.3, we can derive the premise in Rule TMatchNZPre, $e \not\models^{\dagger}_{?} \xi_{pre}$. And thus, the type of the match expression is preserved (Theorem 4.7) as we consider rules in order.

$\boxed{\Delta \vdash p : \tau[\xi] \dashv\!\mid \Gamma}$ p is assigned type τ and emits constraint ξ

$$\frac{}{\cdot \vdash x : \tau[\top] \dashv\!\mid x : \tau}\ \text{PTVar} \qquad \frac{}{\cdot \vdash \_ : \tau[\top] \dashv\!\mid \cdot}\ \text{PTWild} \qquad \frac{}{\Delta, w :: \tau \vdash (\!|\,|\!)^{w} : \tau[?] \dashv\!\mid \cdot}\ \text{PTEHole}$$

$$\frac{\Delta, w :: \tau' \vdash p : \tau[\xi] \dashv\!\mid \Gamma}{\Delta, w :: \tau' \vdash (\!|p|\!)^{w}_{\tau} : \tau'[?] \dashv\!\mid \Gamma}\ \text{PTHole} \qquad \frac{}{\Delta \vdash \underline{n} : \mathsf{num}[\underline{n}] \dashv\!\mid \cdot}\ \text{PTNum}$$

$$\frac{\Delta \vdash p : \tau_1[\xi] \dashv\!\mid \Gamma}{\Delta \vdash \mathtt{inl}(p) : (\tau_1 + \tau_2)[\mathtt{inl}(\xi)] \dashv\!\mid \Gamma}\ \text{PTInl} \qquad \frac{\Delta \vdash p : \tau_2[\xi] \dashv\!\mid \Gamma}{\Delta \vdash \mathtt{inr}(p) : (\tau_1 + \tau_2)[\mathtt{inr}(\xi)] \dashv\!\mid \Gamma}\ \text{PTInr}$$

$$\frac{\Delta \vdash p_1 : \tau_1[\xi_1] \dashv\!\mid \Gamma_1 \qquad \Delta \vdash p_2 : \tau_2[\xi_2] \dashv\!\mid \Gamma_2}{\Delta \vdash (p_1, p_2) : (\tau_1 \times \tau_2)[(\xi_1, \xi_2)] \dashv\!\mid \Gamma_1 \uplus \Gamma_2}\ \text{PTPair}$$

$\boxed{\Gamma\,;\Delta \vdash r : \tau[\xi] \Rightarrow \tau'}$ r transforms a final expression of type τ to a final expression of type τ'

$$\frac{\Delta_p \vdash p : \tau[\xi] \dashv\!\mid \Gamma_p \qquad \Gamma \uplus \Gamma_p\,;\Delta \uplus \Delta_p \vdash e : \tau'}{\Gamma\,;\Delta \vdash (p \Rightarrow e) : \tau[\xi] \Rightarrow \tau'}\ \text{TRule}$$

$\boxed{\Gamma\,;\Delta \vdash [\xi_{pre}]rs : \tau[\xi_{rs}] \Rightarrow \tau'}$ rs transforms a final expression of type τ to a final expression of type τ'

$$\frac{\Gamma\,;\Delta \vdash r : \tau[\xi_r] \Rightarrow \tau' \qquad \xi_r \not\models \xi_{pre}}{\Gamma\,;\Delta \vdash [\xi_{pre}](r \mid \cdot) : \tau[\xi_r] \Rightarrow \tau'}\ \text{TOneRules}$$

$$\frac{\Gamma\,;\Delta \vdash r : \tau[\xi_r] \Rightarrow \tau' \qquad \Gamma\,;\Delta \vdash [\xi_{pre} \vee \xi_r]rs : \tau[\xi_{rs}] \Rightarrow \tau' \qquad \xi_r \not\models \xi_{pre}}{\Gamma\,;\Delta \vdash [\xi_{pre}]r \mid rs : \tau[\xi_r \vee \xi_{rs}] \Rightarrow \tau'}\ \text{TRules}$$

Fig. 16. Typing of Patterns, Single Rules, and Series of Rules

4.4.2 Typing of Patterns and Rules, and Redundancy Checking. Fig. 16 defines the typing judgment for patterns p, a single rule r, and a series of rules rs. We will describe the emission of constraints from and their usage in checking redundancy of a rule r with respect to its preceding ones.

The typing judgment of series of rules rs is of the form $\Gamma\,;\Delta \vdash [\xi_{pre}]rs : \tau_1[\xi_{rs}] \Rightarrow \tau_2$. As shown in Rules TMatchZPre and TMatchNZPre, the constituent rules inherit the variable context Γ and hole context Δ from the outer match expression. When we check the type of a series of rules, we consider each rule in order, just as how we do pattern matching in Sec. 4.2.

Rule TRules corresponds to the inductive case. The first premise is to check the type of the initial rule r. It specifies that each rule takes a final expression of type τ and returns a final expression of type τ'. It also emits a constraint ξ_r, which is actually emitted from the pattern of rule r as we will see later. In order to determine if the initial rule r of the rules $r \mid rs$ is redundant with respect to its preceding rules, we use ξ_{pre} to keep track of the pattern matching information of preceding rules. To accomplish that, as we drop the initial rule r, we append the constraint ξ_r emitted from the pattern of r, to the constraint ξ_{pre}, and use $\xi_{pre} \vee \xi_r$ as the new input to inductively check the type of the trailing rules rs in the second premise. Now that we have shown how to maintain the constraint ξ_{pre} associated with the preceding rules, we can compare it with the constraint of the current rule, ξ_r. As we type check rules, we consider each rule in order and use $\xi_r \not\models \xi_{pre}$ to ensure that the current rule r is not necessarily redundant with respect to its preceding rules. At the same time, the judgment also outputs the accumulated constraint collected from rules $r \mid rs$, which helps exhaustiveness checking , as we have shown in Sec. 4.4.1

Rule TOneRules corresponds to the single rule case. The premises are similar to that of Rule TRules except that there is no trailing rules to check the type of. The reason why we regard one rule as the base case instead of empty rules, is that since our match expression consists of a zipper of rules, we will never need to check the type of an empty set of rules. The only case that it makes sense to allow a match expression with no rules is when we match on a final expression of *Void* type and thus do not need to worry about exhaustiveness checking.

As mentioned above, Rule TRule specifies that rule $p \Rightarrow e$ transforms final expressions of type τ_1 to final expressions of type τ_2. The first premise is the typing judgment of patterns—by assigning pattern p with type τ_1, we collect the typing for all the variables and holes involved in the pattern p and generate variable context Γ_p and hole context Δ_p. Additionally, it emits constraint ξ, which is closely associated with the pattern itself. For example, a pattern hole emits an unknown constraint ?, to indicate that the set of final expressions matching p is yet to be determined (Rules PTEHole and PTHole). The second premise strictly extends two contexts of rule r with bindings from pattern p, and checks the type of sub-expression e.

4.4.3 Type Safety. The type safety of the language is established by Theorem 4.2 and Theorem 4.7.

Theorem 4.7 (Preservation). *If* $\cdot ; \Delta \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot ; \Delta \vdash e' : \tau$.

To prove Theorem 4.7, we need standard substitution lemmas, for both single substitutions and simultaneous substitutions θ (see the appendix or mechanization).

4.5 Elimination of Indeterminacy

Sec. 4.4 has already described exhaustiveness checking and redundancy checking using constraint entailments — Defn. 4.5 and 4.6 — defined in Sec. 4.3. In order for the type system to be decidable, we need to show that the constraint entailments, where both expressions and constraints may involve holes, are decidable. This is nontrivial. But the validity (Defn. 4.8) of a complete constraint (denoted $\dagger(\xi) = \xi$, where $\dagger(\xi)$ replaces all occurrences of ? in ξ with $\top$) that only concerns values is decidable, as we will show shortly in Sec. 4.7. In this section, we eliminate the indeterminacy in constraint entailments by showing that they can be reduced to Defn. 4.8.

Definition 4.8 (Validity of Complete Constraints). *Suppose that* $\dagger(\xi) = \xi$ *and* $\xi : \tau$. *Then* $\models \xi$ *iff for all* e *such that* $\cdot ; \Delta \vdash e : \tau$ *and* e `val` *we have* $e \models \xi$.

As discussed in previous sections, only when a match expression is necessarily inexhaustive, no matter how the programmer fills it, shall the type checking fail. As a human being, such a process can be naturally performed by thinking of all the pattern holes as wildcards and checking if the transformed patterns cover all the possible cases.

Similarly, if we are going to reason about the redundancy of one branch with respect to its previous ones in the presence of pattern holes, the programmer should only get an error/warning if that branch will never be matched no matter how the pattern holes are filled. It is natural to imagine all the previous branches with pattern holes would be refuted during run-time as the programmer may fill them with any refutable patterns. On the other hand, it makes sense to have the branch of our interest to cover, by thinking of its pattern holes as wildcards, as many cases as possible.

As discussed in previous sections, we use a constraint language to capture the semantics of patterns, and then reason about the exhaustiveness and redundancy by reasoning about the constraint language. So we implement the aforementioned intuition through transformation on constraints. Specifically, as a pattern hole emits an unknown constraint ?, replacing the pattern hole with a wildcard corresponds to "truify" ?, i.e., turning ? into $\top$. Similarly, we turn ? into $\bot$ – dubbed "falsify" – to encode the idea of replacing the pattern hole with some pattern that a specific set of expressions won't match. Basically, "truify" operation $\dagger(\xi)$ and "falsify" operation $\perp\!\!\!\downarrow(\xi)$ replace any ? in the input constraint with $\top$ and $\bot$ respectively in a recursive way, the definitions of which are omitted here as they are straightforward (see the Supplementary Material).

With the above observation in mind, Theorem 4.9 and 4.10 establish the "fully-known" equivalence of the entail judgments for exhaustiveness checking $\top \models^{\dagger}_{?} \xi$ and redundancy checking $\xi_r \models \xi_{rs}$.

Theorem 4.9. $\top \models^{\dagger}_{?} \xi$ *iff* $\models \dagger(\xi)$.

Theorem 4.10. $\xi_r \models \xi_{rs}$ *iff* $\models \overline{\dagger(\xi_r)} \vee \perp\!\!\!\downarrow(\xi_{rs})$.

4.6 Agda Mechanization

Our described system is fairly intricate, with many of the proofs requiring extensive case analysis. For instance, while Theorem 4.9 and Theorem 4.10 are intuitively correct, it took us several attempts before ending up with separating the match constraint language presented in the paper into a two-level system. Thus, to ensure no details have been overlooked, the Supplementary Material contains a mechanization of the semantics and metatheory of Peanut using the Agda proof assistant [Norell 2007]. This includes all of the theorems and lemmas stated above and in the appendix.

In general, our mechanization takes an approach similar to that used in the mechanization of Hazelnut Live [Omar et al. 2019]. As is typical in Agda, judgments are encoded as inductive datatypes, and their rules are encoded as dependently typed constructors thereof. For variable and hole names, note that on paper, we ignore issues related to shadowing, and implicitly assume that we may α-convert terms as needed. In our mechanization, rather than explicitly performing such renaming, we insert appropriate uniqueness constraints as a premise to each theorem. Additionally, we make assumptions about disjointedness between bound variables and typing contexts, following a slightly generalized version of Barendregt's convention [Barendregt 1985; Urban et al. 2007]. All such premises are fairly benign, as they can always be satisfied by some α-equivalent term.

Furthermore, again following the mechanization of Hazelnut Live, we restrict variable names to natural numbers, and we encode typing contexts and hole contexts as metafunctions $\mathbb{N} \to \text{Maybe } T$ for appropriate types T. While not strictly necessary, we postulate function extensionality to improve the ergonomics of such an encoding. This postulate is fairly innocuous, as function extensionality is known to be independent of Agda's axioms [Awodey et al. 2012]. We assume no other postulates.

In a few places, the mechanization also differs slightly from the definitions laid out on paper. Most notably, the mechanization removes exhaustiveness and redundancy checking from the typing judgment, and instead includes them separately as their own judgments. This has the advantage of making these checks optional, while also avoiding a morally irrelevant but technically

inconvenient positivity issue with the encoding of the typing judgment in Agda. As a consequence, we also prove separate preservation theorems for each of these judgments, and take exhaustiveness as an additional assumption in the progress theorem. The documentation provided with the mechanization has more details, enumerating all deviations from the paper and their motivation.

4.7 Decidability

In this subsection, we show that the validity of a "fully known" constraint (Defn. 4.8) is decidable.

4.7.1 SAT Encoding. One approach is to reduce the validity checking to a boolean satisfiability problem (SAT). If we revisit the analogy between constraint and set of expressions discussed in Sec. 4.3, we can think of constraints as subsets of values of type τ. Then $\models \xi$ basically says that ξ exactly represents the set of all values of a type τ. However, such a set may be infinite (e.g. top constraint $\top$), and thus defining operations on such infinite sets is nontrivial.

Nevertheless, we may use logical predicates to encode the subset of values corresponding to a constraint. For example, $\xi = \underline{2}$ represents a set with one value $\underline{2}$, and thus can be encoded as a predicate $x = 2$. If there are any connectives ($\wedge$ and $\vee$) in ξ, we can use the connectives of the same form in logical formula. It is tricky to encode *inl*, *inr*, and *pair* constraint as predicate. If we think of a constraint as a set again, a value $e = (e_1, e_2)$ belongs to $\xi = (\xi_1, \xi_2)$ iff e_1 belongs to ξ_1 and e_2 belongs to ξ_2. Therefore, the logical encoding of (ξ_1, ξ_2) would be a conjunction of encoding of both side of the pair, with a variable for each. As for injections, we can use a boolean value b to denote whether a constraint is *inl* or *inr*, and conjoin it with the encoding of ξ_1. In the following example, we use $b = \texttt{true}$ for *inl* and $b = \texttt{false}$ for *inr*.

One last thing to notice here is that we need to make sure when transforming constraints on the same set of values into a predicate, the same variable would be used. To demonstrate how that might work, let's consider a more involved example:

$$(\texttt{inl}(\underline{1}), \texttt{inl}(\underline{3})) \vee (\texttt{inl}(\underline{2}), \texttt{inr}(\underline{1}))$$

$\boxed{\Xi\ \texttt{incon}}$

$$\frac{\Xi\ \texttt{incon}}{\Xi, \top\ \texttt{incon}}\ \text{CINCTruth} \qquad \frac{}{\Xi, \bot\ \texttt{incon}}\ \text{CINCFalsity} \qquad \frac{n_1 \neq n_2}{\Xi, \underline{n_1}, \underline{n_2}\ \texttt{incon}}\ \text{CINCNum}$$

$$\frac{}{\Xi, \underline{n}, \underline{\not n}\ \texttt{incon}}\ \text{CINCNotNum} \qquad \frac{\Xi, \xi_1, \xi_2\ \texttt{incon}}{\Xi, \xi_1 \wedge \xi_2\ \texttt{incon}}\ \text{CINCAnd}$$

$$\frac{\Xi, \xi_1\ \texttt{incon} \qquad \Xi, \xi_2\ \texttt{incon}}{\Xi, \xi_1 \vee \xi_2\ \texttt{incon}}\ \text{CINCOr} \qquad \frac{}{\Xi, \texttt{inl}(\xi_1), \texttt{inr}(\xi_2)\ \texttt{incon}}\ \text{CINCInj}$$

$$\frac{\{\xi' | \texttt{inl}(\xi') \in \Xi\}, \xi\ \texttt{incon}}{\Xi, \texttt{inl}(\xi)\ \texttt{incon}}\ \text{CINCInl} \qquad \frac{\{\xi' | \texttt{inr}(\xi') \in \Xi\}, \xi\ \texttt{incon}}{\Xi, \texttt{inr}(\xi)\ \texttt{incon}}\ \text{CINCInr}$$

$$\frac{\{\xi_1' | (\xi_1', \xi_2') \in \Xi\}, \xi_1\ \texttt{incon}}{\Xi, (\xi_1, \xi_2)\ \texttt{incon}}\ \text{CINCPairL} \qquad \frac{\{\xi_2' | (\xi_1', \xi_2') \in \Xi\}, \xi_2\ \texttt{incon}}{\Xi, (\xi_1, \xi_2)\ \texttt{incon}}\ \text{CINCPairR}$$

Fig. 17. Inconsistency of Constraints

Both $(\texttt{inl}(\underline{1}), \texttt{inl}(\underline{3}))$ and $(\texttt{inl}(\underline{2}), \texttt{inr}(\underline{1}))$ place constraints on the same variable x. Therefore, their left/right side also place constraints on the same variable, though unnecessary to be introduced explicitly. If we let b_{x_l} (b_{x_r}) correspond to the left (right) side of the pairs, and x'_l (x'_r) encode the number constraints under injections, then we encode $\texttt{inl}(\underline{1})$ as $b_{x_l} \wedge (x'_l = 1)$, $\texttt{inl}(\underline{3})$ as $b_{x_r} \wedge (x'_r = 3)$, $\texttt{inl}(\underline{2})$ as $b_{x_l} \wedge (x'_l = 2)$, $\texttt{inr}(\underline{1})$ as $\neg b_{x_r} \wedge (x'_r = 1)$. Put them together and we get the logical encoding of the entire constraint,

$$(b_{x_l} \wedge (x'_l = 1) \wedge b_{x_r} \wedge (x'_r = 3)) \vee (b_{x_l} \wedge (x'_l = 2) \wedge \neg b_{x_r} \wedge (x'_r = 1))$$

As a result, the validity $\models \xi$ of a constraint ξ is equivalent to the validity of its logical encoding. Exhaustiveness and redundancy checking are reduced to the boolean satisfiability, which is NP-complete but decidable, and several tools exist for doing so. For handling numeric patterns we only need linear arithmetic theory in SAT.

4.7.2 Constraint Inconsistency Checking. Using an SMT solver to decide constraint entailments is, however, an overkill. Moreover, it may incur run-time and space overhead in a development environment. When incorporating Peanut into Hazel, we use a different but more lightweight decision procedure. Fig. 17 describes such a procedure by defining a new judgment Ξ $\texttt{incon}$, where Ξ represents a list of constraint ξ. Assuming constraint ξ is fully known and is of type τ, ξ $\texttt{incon}$ means constraint ξ is inconsistent in the sense that no values of type τ satisfy ξ, which corresponds to the unsatisfiability of ξ's logical encoding. Therefore, a constraint is valid, written as $\models \xi$, iff its dual is inconsistent, written as $\overline{\xi}$ $\texttt{incon}$. Note that this is not fully mechanized in Agda. Such proofs require reasoning about finite sets in a non-structurally recursive way, making them inordinately difficult to verify in Agda, but the integration into Hazel uses this algorithm.

5 LABELED SUMS

In this section, we conservatively extend Peanut with finite labeled sums, which is a practical necessity in general-purpose functional languages. Besides adding labeled sums to the syntax, we will only cover pattern matching for labeled sums as this is the most interesting bit of the extension, and leave the other details to the Supplementary Material.

Labeled sums introduce a new sort C for labels, a.k.a. datatype constructors, and a new type-level connective $C_1(\tau_1)+\cdots+C_n(\tau_n)$ for gathering and labeling types. Since we are not usually concerned with the length of any particular sum, we adopt a slightly more general and compact notation $+\{C_i(\tau_i)\}_{C_i \in C}$ for the sum consisting of labels $C = \{C_i\}_{i \leq n}$ with respective argument types $\{\tau_i\}_{i \leq n}$. The introduction form is the *labeled injection expression* $\texttt{inj}^{\tau}_{C}(e)$ for injecting expression e into sum τ at label C. The elimination form is the *labeled injection pattern* $\texttt{inj}^{\tau}_{C}(p)$ for expressions matching pattern p that have been injected into sum τ at label C.

To ensure maximal liveness, we distinguish *concrete labels* $\underline{\texttt{C}}$ from *label holes* which are either empty $(\!|\,|\!)^u$ or not empty $(\!|\underline{\texttt{C}}|\!)^u$. Empty holes arise where labels have yet to be constructed, for example during incremental construction of a sum or injection. Non-empty holes operate as membranes around labels that are syntactically malformed or that violate a semantic constraint. Non-empty holes around syntactically valid labels indicate duplication for sum type declarations and non-membership for injections.

Extending Fig. 9, the rules in Fig. 18 define pattern matching on injections with label holes, subject to the matching determinism conditions imposed by Lemma 4.1. Rule MInj is a straightforward rule for matching labeled injections. Rules MMInjTag and MMInjArg allow indeterminate matching against a pattern whose label or argument indeterminately match, respectively. Rule NMInj forbids matching of concrete but unequal labels, and Rules NMInjTag and NMInjArg forbid matching when the arguments do not match. Rules TMMSym, TMMHole, and TMMEHole define indeterminate

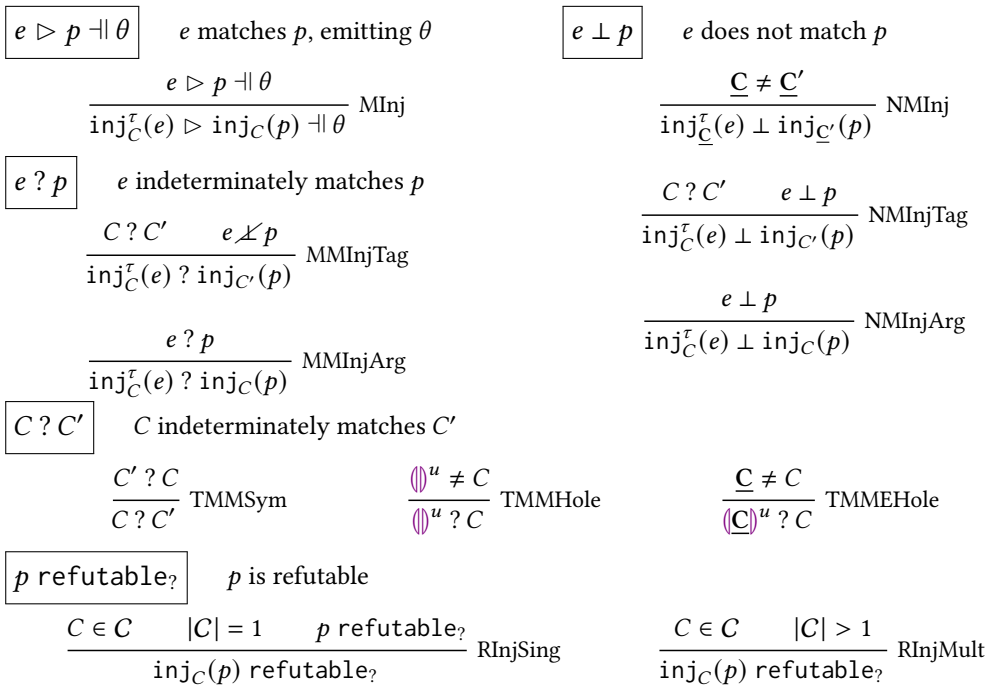


Fig. 18. Pattern matching rules for labeled sums

matching of labels by establishing a partial equivalence among distinct labels when at least one of them is a hole. Rule MMNotIntro works for refutable patterns, and rules RInjMult and RInjSing define refutable patterns of labeled sum type. Rule RInjMult establishes that patterns for nontrivial sums are refutable, i.e., that the set of dual constraints is not empty. The premises of Rule RInjSing express that there are no alternatives for singleton sums, and no choices at all for void sums.

6 RELATED AND FUTURE WORK

6.1 Typed Holes

Hazel, prior to the effort presented in this paper, implemented a minor extension of the Hazelnut Live calculus [Omar et al. 2019]. Peanut is based on the Hazelnut Live internal language, a typed lambda calculus that includes expressions and type holes. Peanut retains expression holes and introduces structural pattern matching, pattern holes, and exhaustiveness and redundancy checking. Like Hazelnut Live, evaluation is able to proceed around holes, including pattern holes. An evaluation step is taken only if it would be justified for all possible hole fillings.

We did not formalize the external language, which is bidirectionally typed, because a simply typed lambda calculus like Peanut can be turned into a bidirectional one in a well-understood way [Dunfield and Krishnaswami 2021]. Our implementation includes a bidirectionally typed external language, whose elaboration to Peanut, the internal language, follows Omar et al. [2019] closely.

Peanut omits type holes, i.e. the unknown types from gradual type theory [Siek and Taha 2006; Siek et al. 2015]. Type holes obscure type information and a scrutinee of unknown type allows patterns of various types to be mixed in a **`match`** expression, so it is difficult to reason statically about exhaustiveness and redundancy except in trivial cases, e.g. variables and wildcards remain

exhaustive.[3] Allowing scrutinees of unknown types would also require performing run-time checks, e.g. in the form of casts inserted on terms matched by variables of unknown type. Building upon the sum type extension described in the appendix of Hazelnut Live paper [Omar et al. 2019], which itself follows the principles of cast insertion in Siek et al. [2015], we have implemented this cast insertion machinery into Hazel. This machinery is orthogonal to the machinery that we contribute in this paper, and which we intend to be relevant to systems that are not gradually typed like GHC Haskell, so we did not include type holes and casts here. It would be reasonably scoped future work to extend Peanut with gradual typing directly, based on the approach implemented in Hazel.

Hazelnut Live augments each expression hole instance with a closure, which serves to record deferred substitutions (it is therefore a contextual type theory [Nanevski et al. 2008]). This information can be presented to the user and it enables soundly resuming from the current evaluation state when the programmer fills a hole (as long as there are no non-commutative side effects in the language). The addition of pattern holes does not interfere with this mechanism. Pattern holes do not themselves need closures because patterns bind, rather than consume, variables. In a language where patterns contain expressions, e.g. when guards are integrated into patterns [Reppy and Zahir 2019], closures on pattern holes would be necessary to support resumption. In languages where datatype constructors are contextually tracked, it would also be possible to track pattern holes as pattern metavariables. It would not be possible to resume evaluation after filling a pattern hole because doing so can, in general, change the binding structure of the program by introducing shadowing or by binding a previously unbound variable. We leave to future work consideration of techniques such as conditional resumption when a pattern hole filling happens not to cause shadowing (which could be considered a *co-contextual* system in that the context around a pattern would limit rather than provide the set of variables that one can use to fill the hole).

In Hazel, holes are inserted automatically during editing. Formally, Hazel is a type-aware structure editor governed by an edit action semantics derived from Hazelnut [Omar et al. 2017a], a type-aware structure editor calculus for the same language as Hazelnut Live. Hazel, by combining machinery from Hazelnut and Hazelnut Live, maintains a powerful liveness invariant: every edit state has a well-defined type and a well-defined result, both possibly containing holes. Our extension of Hazel maintains the same invariant, now with the addition of algebraic datatypes with match expressions as described in this paper. Extending the edit action semantics to allow us to enter patterns presented no challenges relative to prior work, so we omit formal consideration of editing from this paper. The work in this paper has enabled Hazel to be the first general-purpose language (i.e. one that goes beyond the minimal calculus of Hazelnut) with a maximal liveness invariant.

Our contributions do not require a structure editor: they are also relevant to languages where typed holes are inserted manually by programmers, rather than automatically by an editor. For example, GHC Haskell, Agda, and Idris, all feature manually inserted typed holes, both empty and non-empty, in expression position. None of these systems support pattern holes as of this writing, however. Haskell does support unbound data constructors in patterns, but these bring compilation to a halt and do not interact with the exhaustiveness and redundancy checker, much less the run-time system. With the appropriate flags set, Haskell will attempt to evaluate programs with expression holes, but it stops with an exception whenever a hole is reached [Vytiniotis et al. 2012]. In contrast, our system supports evaluating around holes of all sorts, as described in Sec. 3 and formalized in Sec. 4.2. We hope that this paper will prompt other languages to consider pattern

[3]Constraint-based type inference could be deployed to discover a type for the scrutinee in some cases, at which point it would be possible to use our mechanisms as described. Whether inference is deployed for this purpose is an orthogonal consideration.

holes. Our work on Peanut captures the essential character of pattern matching with holes and our implementation in Hazel demonstrates a practical realization, complete with editor integration.

We did not consider the problem of synthesizing pattern hole fillings in this paper, but type-driven techniques used for expression holes, e.g. in Haskell [Gissurarson 2018], could likely be adapted. The exhaustiveness and redundancy analyses could help guide these techniques to search for hole fillings that increase the exhaustiveness of the `match` expression.

There has also been recent progress in bringing graduality to dependent types as an attempt to reason about unknown terms/types [Eremondi et al. 2022, 2019; Lennon-Bertrand et al. 2022; Maillard et al. 2022]. Their goal is to make dependent type system more usable in general, while we want to enable reasoning at *all* program editing states with a focus on patterns. The prior work in this direction has not considered pattern holes. While in this paper, we focused on simple types, the logical character of the work makes it suitable for direct extension to support dependent pattern matching. We leave problems of integration with higher-order unification as is used to make pattern matching more convenient in languages like Agda as future work.

6.2 Pattern Matching

Pattern matching has a long history, first described by Burstall [1969] and implemented as an extension to LISP by McBride [1970]. Early functional languages, such as Hope [Burstall et al. 1980], SASL [Turner 1979], and ML [Milner 1978], started to adopt pattern matching in the 1970s. All prior work on pattern matching assumes that the patterns and expressions being matched are complete, i.e., do not contain any holes. This paper contributes techniques to perform exhaustiveness and redundancy checks and run programs with pattern holes, by reasoning over all future edit states, following the modal interpretation of holes [Nanevski et al. 2008; Omar et al. 2019].

Much of the early work on pattern matching focus on compilation. The earliest work targets the end-to-end compilation of pattern matching into machine code [Augustsson 1984, 1985; Cardelli 1984], while later methods focus on efficient compilation. Some methods construct decision trees encoding the matching procedure, the goal being to minimize the sizes of the constructed trees, obtaining exhaustiveness and redundancy checks as easy by-products [Aitken 1992; Baudinet and MacQueen 1985; Sestoft 1996]. Other methods compile pattern matching to backtracking automata [Maranget 1994, 2007]. While these methods avoid potential exponential behavior exhibited by decision trees, they require separate methods for reporting inexhaustive and redundant patterns [Maranget 2007].

In our setting of incomplete programs, the question of compilation is not (yet) relevant. Our goal in this paper, which was inspired by a sketch of a similar constraint language in the textbook by Harper [2012] (which was omitted in the second edition of the book), is to provide a complete and, notably, mathematically elegant declarative semantics of pattern matching with holes that extends the incomplete program evaluation of Hazelnut Live, then consider decidability separately via the truthification, falsification and material entailment techniques described in this paper. Efficient compilation of programs with pattern holes is, however, an interesting avenue for further work (alongside the more general question of efficient evaluation or compilation of programs with expression holes). One potential direction is to adapt the pattern matrices approach introduced in Maranget [2007] to support unknown patterns.

Krishnaswami [2009] also establishes a logical interpretation of pattern matching, in their case rooted in the technique of focusing, without considering pattern holes. Their exhaustiveness and redundancy checking algorithms operate directly on a syntax of patterns, which includes logical connectives equipped with a dualizing operation, which is similar in character to our approach. However, our work separates out the logical component into a separate pattern constraint language, which arguably simplifies and highlights the essential logical character of the problem, following

the approach suggested by Harper [2012]. Even excluding the pattern holes, then, this paper's approach might be useful for future work in need of a logical account of pattern matching.

More contemporary work focuses on extending pattern matching to settings with more sophisticated types and pattern matching features [Abel et al. 2013; Cockx and Abel 2018; Convent et al. 2020; Graf et al. 2020; Karachalias et al. 2015; Sozeau 2010; Vazou et al. 2014]. This work concerns itself with analyzing increasingly sophisticated predicates delineating a match from a failure (or divergence in the lazy setting), our work introduces new pattern matching *outcomes*—indeterminacy. To that end, we narrow our focus to the novel aspects of our system and leave integration with existing, richer pattern matching features and checkers to future work. Again, the logical character of our system is likely to simplify integration with systems that rely on SMT solving or other logical techniques, e.g. the work of Graf et al. [2020] on exhaustiveness checking in the presence of arbitrary boolean guards. Indeed, we observed that deciding the entailments of interest in our system can be reduced to an SMT problem in Sec. 4.7 (inspired directly by this recent work).

Gradually structured data [Malewski et al. 2021] is the only work we are aware of that also concerns the static reasoning of pattern matching during program construction. It applies general principles from gradual typing to lift exhaustiveness checking given datatypes with unknown constructors. However, unknown patterns or unknown constructors in patterns are beyond their scope, and are the central contribution of this paper. Therefore, it is a complementary approach that can be combined with ours.

7 CONCLUSION

> *As we have struggled through the ages to fathom this strange and wondrous cosmos in which we find ourselves, few ideas have been richer than the concept of nothingness. For to understand anything, as Aristotle argued, we must understand what it is not.*
>
> — Alan Lightman

Programming can perhaps be understood as a progression, full of fits and starts, from an initial nothingness, an empty hole, toward a completely meaningful computational world. This paper continues a promising line of research into applying principled logical methods to understand and support this process [Omar et al. 2017a,b]. In the future, we hope that robust support for typed holes, whether inserted manually or automatically, will be as ubiquitous as other editor services, such as code completion and type inspection. Pattern matching is a central feature of modern typed functional programming languages (and increasingly many other modern programming languages), so we believe that the semantics for typed pattern holes contributed by this paper represents a significant step toward realizing the goal of *maximally live* programming environments.

8 CODE AVAILABILITY

The Agda mechanization and the SAT-based implementation of the constraint validity checking algorithm is available in the accompanying artifact [Yuan et al. 2023]. The Hazel implementation is an open source project. More information, including source code, is available at https://hazel.org/.

ACKNOWLEDGMENTS

We thank the anonymous referees for their insights and feedback over several iterations of this work. This material is based upon work supported by the National Science Foundation under Grant No. 2238744. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 27–38. https://doi.org/10.1145/2429069.2429075

William Aitken. 1992. *SML/NJ Match Compiler Notes*. Technical Report. https://www.smlnj.org/compiler-notes/matchcomp.ps

Lennart Augustsson. 1984. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr. (Eds.). ACM, 218–227. https://doi.org/10.1145/800055.802038

Lennart Augustsson. 1985. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 201)*, Jean-Pierre Jouannaud (Ed.). Springer, 368–381. https://doi.org/10.1007/3-540-15975-4_48

Steven Awodey, Nicola Gambino, and Kristina Sojakova. 2012. Inductive Types in Homotopy Type Theory. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 95–104. https://doi.org/10.1109/LICS.2012.21

Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

Marianne Baudinet and David MacQueen. 1985. *Tree Pattern Matching for ML (Extended Abstract)*. Technical Report. https://smlfamily.github.io/history/Baudinet-DM-tree-pat-match-12-85.pdf

Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *Proc. ACM Program. Lang.* 2, ICFP (2018), 103:1–103:15. https://doi.org/10.1145/3236798

Edwin Brady. 2013. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Rod M. Burstall. 1969. Proving Properties of Programs by Structural Induction. *Comput. J.* 12, 1 (1969), 41–48. https://doi.org/10.1093/comjnl/12.1.41

Rod M. Burstall, David B. MacQueen, and Donald Sannella. 1980. HOPE: An Experimental Applicative Language. In *Proceedings of the 1980 LISP Conference, Stanford, California, USA, August 25-27, 1980*. ACM, 136–143. https://doi.org/10.1145/800087.802799

Luca Cardelli. 1984. Compiling a Functional Language. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr. (Eds.). ACM, 208–217. https://doi.org/10.1145/800055.802037

Jesper Cockx and Andreas Abel. 2018. Elaborating dependent (co)pattern matching. *Proc. ACM Program. Lang.* 2, ICFP (2018), 75:1–75:30. https://doi.org/10.1145/3236770

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2021), 98:1–98:38. https://doi.org/10.1145/3450952

Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional equality for gradual dependently typed programming. *Proc. ACM Program. Lang.* 6, ICFP (2022), 165–193. https://doi.org/10.1145/3547627

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate normalization for gradual dependent types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 88:1–88:30. https://doi.org/10.1145/3341692

Matthías Páll Gissurarson. 2018. Suggesting valid hole fits for typed-holes (experience report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 179–185. https://doi.org/10.1145/3242744.3242760

Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. Lower your guards: a compositional pattern-match coverage checker. *Proc. ACM Program. Lang.* 4, ICFP (2020), 107:1–107:30. https://doi.org/10.1145/3408989

Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press. https://doi.org/10.1017/CBO9781139342131

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2015. GADTs meet their match: pattern-matching warnings that account for GADTs, guards, and laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 424–436. https://doi.org/10.1145/2784731.2784748

Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. 2009. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 445–464. https://doi.org/10.

1145/1640089.1640122

Neelakantan R. Krishnaswami. 2009. Focusing on pattern matching. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 366–378. https://doi.org/10.1145/1480881.1480927

Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the Calculus of Inductive Constructions. *ACM Trans. Program. Lang. Syst.* 44, 2 (2022), 7:1–7:82. https://doi.org/10.1145/3495528

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 109:1–109:29. https://doi.org/10.1145/3408991

Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A reasonably gradual type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 931–959. https://doi.org/10.1145/3547655

Stefan Malewski, Michael Greenberg, and Éric Tanter. 2021. Gradually structured data. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485503

Luc Maranget. 1994. *Two Techniques for Compiling Lazy Pattern Matching.* Technical Report. HAL-Inria. https://hal.inria.fr/inria-00074292/document

Luc Maranget. 2007. Warnings for pattern matching. *J. Funct. Program.* 17, 3 (2007), 387–421. https://doi.org/10.1017/S0956796807006223

Frederick Valentine McBride. 1970. *Computer aided manipulation of symbols.* Ph. D. Dissertation. Queen's University Belfast, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.463849

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

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. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL (2019), 14:1–14:32. 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 *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. http://dl.acm.org/citation.cfm?id=3009900

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).* https://doi.org/10.4230/LIPIcs.SNAPL.2017.11

Simon Peyton Jones, Sean Leather, and Thijs Alkemade. 2020. Glasgow Haskell Compiler 9.2.1 User's Guide (Typed Holes). https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/typed_holes.html. Retrieved Mar 3, 2022.

Brigitte Pientka and Jana Dunfield. 2008. Programming with proofs and explicit contexts. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, Sergio Antoy and Elvira Albert (Eds.). ACM, 163–173. https://doi.org/10.1145/1389449.1389469

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

Hannah Potter and Cyrus Omar. 2020. Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies. https://hazel.org/hazeltutor-hatra2020.pdf

John Reppy and Mona Zahir. 2019. Compiling Successor ML Pattern Guards. https://github.com/JohnReppy/compiling-pattern-guards/blob/master/ml19-paper.pdf

Peter Sestoft. 1996. ML pattern match compilation and partial evaluation. In *Partial Evaluation*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 446–464.

Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming.* http://scheme2006.cs.uchicago.edu/13-siek.pdf

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, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Matthieu Sozeau. 2010. Equations: A Dependent Pattern-Matching Compiler. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 419–434. https://doi.org/10.1007/978-3-642-14052-5_29

Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. https://doi.org/10.1109/LIVE.2013.6617346

D. A. Turner. 1979. A New Implementation Technique for Applicative Languages. *Softw. Pract. Exp.* 9, 1 (1979), 31–49. https://doi.org/10.1002/spe.4380090105

Christian Urban, Stefan Berghofer, and Michael Norrish. 2007. Barendregt's Variable Convention in Rule Inductions. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4603)*, Frank Pfenning (Ed.). Springer, 35–50. https://doi.org/10.1007/978-3-540-73595-3_4

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. 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, 269–282. https://doi.org/10.1145/2628136.2628161

Dimitrios Vytiniotis, Simon L. Peyton Jones, and José Pedro Magalhães. 2012. Equality proofs and deferred type errors: a compiler pearl. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 341–352. https://doi.org/10.1145/2364527.2364554

Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. *Artifact for "Live Pattern Matching with Typed Holes"*. https://doi.org/10.5281/zenodo.7713722

Received 2022-10-28; accepted 2023-02-25