Pattern Matching with Typed Holes

Computer Science Honors Thesis

SCOTT GUEST, University of Michigan
CYRUS OMAR, University of Michigan

To support reasoning about incomplete programs in a principled way, various programming systems have introduced *typed-holes* - placeholder terms which indicate missing syntactic pieces or semantic inconsistencies [Brady 2013; Norell 2013; Omar et al. 2017a; Peyton Jones et al. 2020]. Ideally, these holes allow every intermediate edit state of a program to be given static or even dynamic meaning, with the aim of enabling simpler and more powerful development tools [Omar et al. 2019, 2017b]. However, current systems are limited in that they only support holes in expressions or types, presenting difficulty when editing binding constructs such as patterns. To resolve this, we have developed Peanut: a calculus for pattern matching with typed pattern holes, including support for exhaustiveness and redundancy checking in this setting. Additionally, we also provide a mechanization of Peanut's semantics and metatheory in the Agda proof-assistant [Norell 2007].

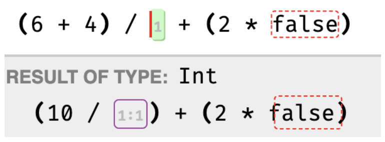
1 INTRODUCTION

For the modern programmer, development has progressed beyond the unassisted writing of code in a text editor, and it now increasingly involves conversing with a wide-variety of programming tools. As incremental changes are made, type checkers, debuggers, interpreters, program synthesizers and so forth all provide feedback, seeking to increase programmer productivity or ensure program correctness. Unfortunately, however, programming languages typically only assign meaning to programs which are already fully-formed and well-typed. As a result, these tools struggle to deal with what is perhaps one of the most common states of a program during development: an unfinished piece of work with ongoing, syntax-breaking editing or as-yet-unresolved semantic errors. Throughout the editing process then, feedback from these tools is only available sporadically, limiting their usefulness when they are needed most.

This troublesome issue, where programming tools have gaps in their services throughout the editing process, is aptly known as the *gap problem*. Previously, most attempts to close such gaps have relied on *ad hoc* or fragile heuristics, for example, making assumptions about missing tokens, or ignoring large swathes of invalid code all together [Kats et al. 2009; Omar et al. 2017b]. Recently though, the work of [Omar et al. 2017b] has outlined a more principled approach: rather than treating intermediate edit states as meaningless, one can extend a language's syntax and semantics to explicitly represent all such states as well-formed terms. Every editor state can then be assigned static or even dynamic meaning, allowing programming tools to handle them in a uniform way and removing the need for such *ad hoc* heuristics.

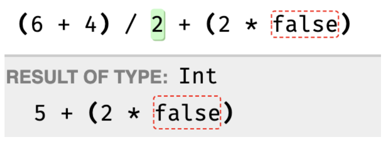
To varying degree, systems including Haskell [Peyton Jones et al. 2020], Idris [Brady 2013], Agda [Norell 2013], and Hazel [Omar et al. 2017a] implement this approach through a feature known as *typed-holes*. When a program has a missing syntactic piece, rather than treating the entire program as meaningless, we localize the issue by inserting an *empty hole* expression at the unfinished location. Likewise, semantically inconsistent expressions, e.g. those that are ill-typed or reference undefined identifiers, may be wrapped with a *non-empty hole* expression, isolating the inconsistency from the program at large. Tools can then the aid the developer by providing static information about these holes - displaying the variables and types in scope, inferring the expected type, or even synthesizing possible hole-fillings [Gissurarson 2018; Lubin et al. 2020].

University of Michigan, April, 2022, Ann Arbor, MI, USA
2022.



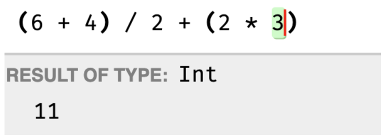
```
(6 + 4) / 1 + (2 * false)
RESULT OF TYPE: Int
  (10 / 1:1) + (2 * false)
```

(a) Empty and non-empty holes



```
(6 + 4) / 2 + (2 * false)
RESULT OF TYPE: Int
  5 + (2 * false)
```

(b) Result after filling empty hole



```
(6 + 4) / 2 + (2 * 3)
RESULT OF TYPE: Int
  11
```

(c) Result after filling non-empty hole

Fig. 1. Live Evaluation with Expression Holes

Most notable among the aforementioned systems is Hazel - a structure editor which was designed from logical and type-theoretic first principles to seamlessly and fully-automatically insert holes throughout the editing process. Starting from an empty hole, every possible program can be built up using a set of edit actions which directly manipulate the syntax tree. Along the way, Hazel maintains a *maximal liveness* invariant so that every intermediate step is a valid term with well-defined static and dynamic semantics, completely eliminating the gap problem [Omar et al. 2019]. Fig. 1 displays screenshots of a small part of the Hazel editor, with the user in the process of editing a simple arithmetic expression. Considering only Fig. 1a for now, both variants of typed-holes are present: an empty hole labeled with id `1` indicates a missing piece of the expression, and a non-empty hole, shown as a dashed red box, serves as a membrane around the type inconsistency related to the term `false`.

One of Hazel's main contributions is its comprehensive dynamic semantics which allows evaluation to continue "around holes", thereby preventing gaps in services which rely on actual program execution [Omar et al. 2019]. This stands in contrast to systems such as Haskell, which, when supplied with appropriate compiler flags, treat holes as panics - reminiscent of a common pattern where developers raise an exception at yet-to-be-implemented code locations [Peyton Jones et al. 2020]. With Hazel, however, upon evaluation encountering a hole, rather than panicking or otherwise terminating the program, Hazel defers the evaluation of that particular expression and those that rely on. It then continues to evaluate all other complete (i.e. hole-less) parts of the program until no such evaluation is possible. Finally, evaluation pauses, yielding an *indeterminate* value if there are still holes present. At a later time, the remaining holes in this indeterminate result can be filled, and evaluation resumes without having to restart the program. As a result, Hazel provides continuous live feedback throughout the entire editing process.

The rest of Fig. 1 demonstrates this live evaluation. Initially, the expression shown in Fig. 1a contains two holes. Hazel partially evaluates the expression, reducing the complete term `6 + 4` to `10`, but can perform no further reductions. An indeterminate result is displayed, still containing both holes present in the initial expression. In Fig. 1b, we then fill the empty hole with id `1` using the value `2`. This allows evaluation to resume, further reducing `(6 + 4) / 2` to `5`, but again pausing due to the presence of the non-empty hole. Finally, in Fig. 1c, we resolve the typing inconsistency by replacing the term `false` with the integer `3`. The non-empty hole is automatically removed, and Hazel now produces a concrete value `11`. At no point during this process is the editor state meaningless, nor is the program ever restarted.

While Hazel itself is still under development, and is currently little more than a lambda calculus with primitive types, sums, and products, it seeks to soon reach feature-parity with production-level languages such as Elm [Czaplicki 2018; Czaplicki and Chong 2013; Omar et al. 2019]. Indeed, the machinery behind Hazel is not limited to its specific features, but rather, it provides a systematic, near-mechanical way to build similar structure editors for any language. Its semantics are built up using common techniques from gradual typing [Siek et al. 2015]: holes are considered to have the unknown type, and terms are elaborated into a cast calculus to handle such typing at runtime. Additional machinery from contextual modal type theory [Nanevski et al. 2008] is then used to track substitutions which occur around holes, allowing holes to be filled in any order while producing the same final result. Given this principled theoretical basis, future work may even allow us to algorithmically generate Hazel-like structure editors for numerous languages in a fully automatic way, developing something akin to the Gradualizer [Cimini and Siek 2016]. However, for any of this development be possible, and indeed to apply Hazel's techniques to any full-fledged functional programming language at all, one commonplace feature still needs more careful consideration: *structural pattern matching*.

Throughout the rest of this paper, we tackle the challenge of integrating pattern matching into a Hazel-like system while seeking to maintain the maximal liveness invariant. We begin in Sec. 2 by reviewing the required background and terminology for pattern matching in general. In Sec. 3, we then introduce the concept of a *pattern hole*, and provide a high-level discussion of our desired semantics for pattern matching with these holes, including redundancy and exhaustiveness in this setting. Finally, in Sec. 4, we formalize this high-level discussion, presenting our work as a type theoretic calculus Peanut which extends the Hazelnut Live internal language of [Omar et al. 2019]. To round off the development, we also outline algorithmic implementations of redundancy and exhaustiveness checking in Sec. 4.6, then discuss an Agda mechanization of Peanut's semantics and metatheory in Sec. 5.

2 BACKGROUND

Let us begin by reviewing pattern matching in the context of languages without holes. Such a feature should be familiar to many modern developers, as it has seen widespread adoption in typed functional languages such as OCaml [Leroy et al. 2021], Haskell [Marlow 2010], and Elm [Czaplicki 2018], and more recently, has also been implemented to varying degree in mainstream imperative languages including Rust [Klabnik and Nichols 2018] and Python [Bucher and van Rossum 2020], and has also been proposed for a future version of C++ [Murzin et al. 2019].

At a high-level, *structural pattern matching* allows one to conditionally branch based on the "shape" of a piece of data, while simultaneously binding sub-terms of the data to specified variable names. Concretely, this accomplished through a **match** expression consisting of a *scrutinee*, i.e. the expression whose shape we inspect, and a series of *rules*. Each rule contains a *pattern* describing the desired form of the data for that particular case, as well as a *branch expression* indicating the result of evaluation in that case. The branch expression may reference the variables bound in the pattern.

Fig. 2a and Fig. 2b present pseudocode showcasing this feature. The scrutinee is a variable `tree` containing some value of an algebraic datatype. Each pattern is an expression of nested constructors that could possibly match those constructors of the value in `tree`, with variable names in a pattern acting similarly to a wildcard. Semantically, the match expression compares the value of `tree` against each of these patterns, beginning from the top down. The first pattern accurately describing the constructors in `tree` is selected, then variables in the pattern are bound to corresponding sub-terms of `tree`, and finally, evaluation continues at the selected branch expression.

```
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. 2. Two examples demonstrating structural pattern matching and common pitfalls.

Explicitly, in Fig. 2a, assume `tree` contains the value `Node([Foo(4), Bar(5)])`, where the syntax `[x , y]` is sugar for the nested cons cells `x::y::[]`. Although the first two patterns indeed have an outermost `Node` constructor, they are not matched due to the inner list constructors differing from our value. However, the third pattern `Node([x, y])` is successfully matched, thereby binding the variable `x` to `Foo(4)` and the variable `y` to `Bar(5)`. Correspondingly, the whole term evaluates to the branch expression `Node([f(Foo(4)), f(Bar(5))])`. Note that our scrutinee would also match the fourth pattern `Node(x::y::tl)` - binding `x` to `Foo(4)`, `y` to `Bar(5)`, and `tl` to `[]`. However, execution does not actually reach this point, as the third pattern is matched earlier in the rule sequence.

More than just a useful control flow structure, pattern matching also helps to ensure program correctness. Consider if the user modifies Fig. 2a to the program shown in Fig. 2b. For our previously discussed value, the new ordering of rules results in the pattern `Node(x::y::tl)` being matched before the pattern `Node([x, y])` is ever considered, changing the overall behavior of the program. In fact, any scrutinee which matches the second pattern `Node([x, y])` will also match the first pattern `Node(x::y::tl)`. Resultingly, as evaluation considers rules top to bottom, the second pattern here is unreachable and entirely *redundant*. Such a bug caused by reordering can be quite subtle, and it would be difficult to detect if our program had instead been formulated using, say, a series of **if-then-else** expressions. However, with pattern matching, we can easily prevent such issues by performing *redundancy analysis* - an algorithm which allows the language to statically detect if any rule in a **match** is impossible to reach due to it being entirely subsumed by earlier rules in the sequence. The user can then be notified of a likely bug, eliminating large classes of errors related to code reordering or unused code paths.

Additionally, note that Fig. 2b also fails to include the `Leaf x` pattern, meaning that certain scrutinee values will no longer match any of the given cases. By performing *exhaustiveness analysis*, we can prevent this issue as well, statically checking that every expression of a given type matches at least one of the patterns in a **match** expression. As a result, the programmer is forced to handle every possible form of an input, and if they fail to do so, the language can often assist them by generating expressions which are not yet handled [Harper 2012]. Again, this enables large classes of bugs to be prevented statically. For example, from a certain viewpoint, common security issues such as null-pointer exceptions can be viewed as a failure to check exhaustiveness. Exhaustiveness also aids refactoring: if a datatype is extended with additional constructors, then the language can statically identify every location where these additional constructors must be handled.

Anecdotally, the author's found these analyses immensely useful during the development of the Agda mechanization discussed in Sec. 5. In the context of Agda and other dependently typed theorem provers, exhaustiveness is used to ensure totality, in effect, guaranteeing that every proof handles all necessary cases. When, for example, we added a unit type to our development after an initial mechanization had already been completed, this removed the need to manually track down and identify all proofs that needed to be updated. Instead, all that was necessary was to one-by-one handle every inexhaustiveness error reported by the compiler.

3 PATTERN MATCHING WITH TYPED HOLES IN HAZEL

Let us now return to the problem of extending Hazel with pattern matching, placing particular emphasis on maintaining Hazel's maximal liveness invariant. Presently, all current systems with typed-holes only support holes in expressions or types, but notably, do not permit holes in binding constructs such as patterns. As a result, when editing a pattern - a necessarily incremental process - a user still faces meaningless editor states and thus gaps in editor services. To resolve this, we again turn to the approach outlined in [Omar et al. 2017b], extending our language to represent such intermediate pattern edit states as well-formed terms.

Note that, for our purposes, patterns and expressions are quite similar: both are built up compositionally and are subject to typing restrictions. Correspondingly, we proceed by introducing two variants of *pattern holes*. We include *empty pattern holes* to indicate a missing sub-term of a pattern, and we include *non-empty pattern holes* to act as a membrane around patterns that are ill-typed with regards to their location in a larger pattern. Syntactically, pattern holes indeed enable us to represent any pattern edit state without much difficulty, and they are fairly trivial to implement into our extension of Hazel. Semantically, however, the situation is much more subtle.

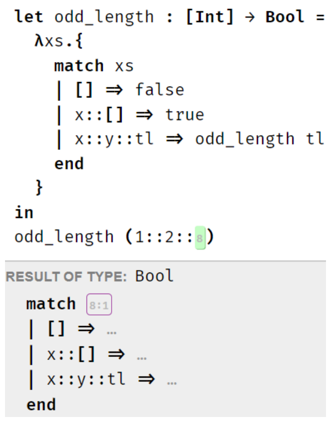
At a high level, expression holes indicate *unknown expressions* and pattern holes indicate *unknown patterns*. Thus, whatever semantics we choose to give terms with holes, it must be sound with respect to all possible hole-fillings, and resultingly, we must reason conservatively about the contents of any particular hole. At the same time, we must strike a balance, limiting this conservativeness as much as possible in order to still provide viable static analysis and dynamic evaluation in cases where we can soundly do so without knowledge about the contents of any hole. For pattern matching, this balance manifests as stating that a match succeeds only if it succeeds in all possible (expression and pattern) hole-fillings. Likewise, a match fails only if it fails in all possible hole-fillings. However, as the astute reader will note, this leaves open a third possibility: some hole-fillings may result in a match succeeding while other hole-fillings result in the match failing - an expression may *indeterminately* match a pattern.

When we allow expression and pattern holes, we can then no longer determinately say whether an expression matches a given pattern. What was once a binary decision - either e matches p or e does not match p - is now a ternary decision: either e must match p, e must not match p, or e indeterminately matches p depending on how the various holes are filled. In turn, redundancy and exhaustiveness also become ternary decisions: a **`match`** expression may either be necessarily exhaustive, necessarily inexhaustive, or indeterminate, and likewise for redundancy.

To more concretely present these subtleties, throughout the rest of this section we discuss a running example where the user edits an `odd_length` function, returning whether an input of type `[Int]` has an odd number of elements. We explore the full semantics of the **`match`** expression with live evaluation in Sec. 3.1, then further explore exhaustiveness in Sec. 3.2 and redundancy in Sec. 3.3.

3.1 Live Evaluation with Pattern Holes

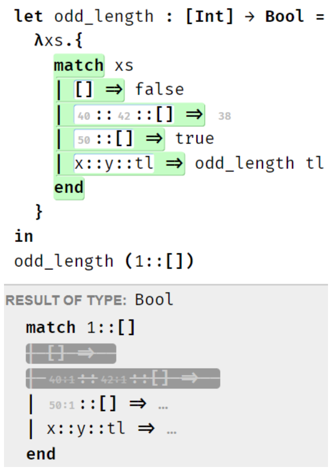
As discussed, in order to support live evaluation with pattern holes, we must distinguish whether a given expression and pattern *must* match, *must not* match, or *indeterminately* match. Let us consider how these different possibilities play out in Fig. 3, working through examples of a **`match`** expression with expression holes in the scrutinee and with pattern holes in the listed rules. The provided screenshots are from the Hazel implementation of our work.



```
let odd_length : [Int] → Bool =
  λxs.{
    match xs
    | [] ⇒ false
    | x::[] ⇒ true
    | x::y::tl ⇒ odd_length tl
    end
  }
in
odd_length (1::2::8)
```

```
RESULT OF TYPE: Bool
  match 8:1
  | [] ⇒ …
  | x::[] ⇒ …
  | x::y::tl ⇒ …
  end
```

(a) Pattern matching with expression holes



```
let odd_length : [Int] → Bool =
  λxs.{
    match xs
    | [] ⇒ false
    | 40 :: 42 ::[] ⇒  38
    | 50 ::[] ⇒ true
    | x::y::tl ⇒ odd_length tl
    end
  }
in
odd_length (1::[])
```

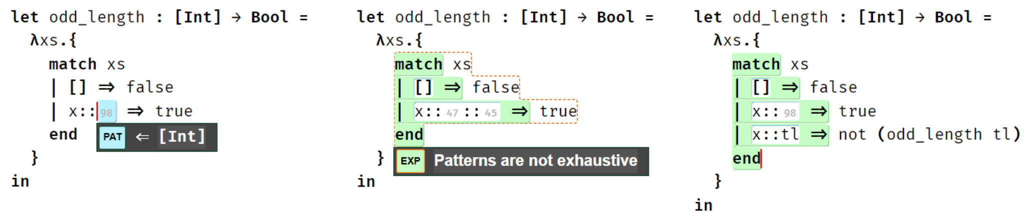
```
RESULT OF TYPE: Bool
  match 1::[]
  ~~| [] ⇒~~
  ~~| 40:1 :: 42:1 ::[] ⇒~~
  | 50:1 ::[] ⇒ …
  | x::y::tl ⇒ …
  end
```

(b) Pattern matching with pattern holes

Fig. 3. Live Evaluation with Expression and Pattern Holes

In Fig. 3a, we apply the function `odd_length` to an argument that contains an expression hole with id 8. Although the argument is indeterminate and cannot be fully-evaluated, per Hazel's live dynamic semantics, we still substitute it into the function body and continue as far as possible. Our argument then becomes the scrutinee of the **match** expression, and we proceed with pattern matching. Beginning with the first pattern, regardless of hole-filling, the outermost constructor of our scrutinee is the cons operator `::` rather than the empty list `[]`. Thus, our scrutinee must not match the first pattern `[]`, and we continue to the next rule. For the second pattern, regardless of hole-filling, our scrutinee always contains at least two elements while the pattern specifies only one element. Again then, our scrutinee must not match the second pattern, so evaluation moves to consider the third rule. Finally, as the scrutineee indeed always contains at least two elements, the third match succeeds in every hole-filling. Correspondingly, `x` is bound to `1`, `y` is bound to `2`, and `tl` is bound to the hole with id 8. Evaluation then proceeds to the recursive call, and in this frame, our scrutinee becomes just an expression hole. Because we do not know whether the hole will be filled with an empty list or some other contents, we cannot determinately say whether it matches the first pattern `[]`, hence evaluation must pause. We cannot proceed without further hole-filling, so the entire **match** expression becomes indeterminate, displayed as the final result at the bottom of the image. Note that the scrutinee in the final result is indeed just the expression hole with id 8.

In Fig. 3b, our argument no longer contains any expression holes, but indeterminacy still arises due to the pattern holes in the second and third rules of the **match** expression. Specifically, evaluation begins by analyzing the scrutinee against the first pattern, and as `1 :: []` is not the empty list, we determine it must not match the first pattern. Likewise, despite the presence of pattern holes, the second pattern may only be matched by a two element list, so our scrutinee must not match it, and we continue to the third pattern. The third pattern gives the most interesting case. It indeed specifies a list with a single element, but the head of the pattern is a pattern hole with id `50`, indicating that the first element of the scrutinee must match some yet-unknown pattern. If hole `50` were filled with the integer pattern `1` or a variable `x`, then `1::[]` would match. However, for many other hole-fillings, e.g. the integer `2`, the match clearly fails. As a result, we can only state that `1::[]` indeterminately



```
let odd_length : [Int] → Bool =
  λxs.{
    match xs
    | [] ⇒ false
    | x::98 ⇒ true
    end   PAT ⇐ [Int]
  }
in
```

```
let odd_length : [Int] → Bool =
  λxs.{
    match xs
    | [] ⇒ false
    | x:: 47 :: 45 ⇒ true
    end
  }  EXP Patterns are not exhaustive
in
```

```
let odd_length : [Int] → Bool =
  λxs.{
    match xs
    | [] ⇒ false
    | x:: 98 ⇒ true
    | x::tl ⇒ not (odd_length tl)
    end
  }
in
```

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

Fig. 4. Exhaustiveness Checking with Pattern Holes

matches the third pattern, and evaluation must pause pending further hole-filling. Correspondingly, the entire **match** expression becomes indeterminate and is shown as the final result. By striking out the first two rules and displaying them in gray, we indicate to the user that evaluation has already proceeded past these cases.

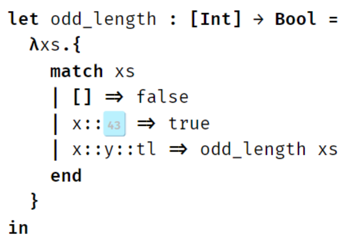
3.2 Exhaustiveness Checking with Pattern Holes

With the dynamic semantics for live evaluation now clear, let us consider how to statically reason about this dynamic behavior through checks such as *exhaustiveness analysis*. Recall that, in a language without holes, exhaustiveness requires that every expression of appropriate type matches at least one of the patterns in a **match** expression. When we introduce holes, however, exhaustiveness becomes more subtle, and we again must reason conservatively about all possible hole-fillings and thereby handle cases of indeterminacy.

Explicitly, assume the editor state is as in Fig. 4a. The **match** expression contains two rules - the first with an empty list pattern `[]`, and the second with a cons cell pattern `::` containing a variable `x` at the head and a pattern hole at the tail. The cursor is placed over the pattern hole, and Hazel is able to infer the type of the pattern as `[Int]` from the surrounding context, presenting this information to the user. Should we consider such an expression to be exhaustive? If the user fills the pattern hole with a variable `xs`, then the **match** is indeed exhaustive: any empty list matches the first pattern, and any non-empty list matches the second pattern. However, if the user instead fills the hole with, say, `[]` or `y::tl`, then a list with three elements would fail to match any pattern. Thus, without further hole-filling, we can only soundly state that the **match** is *indeterminately exhaustive*.

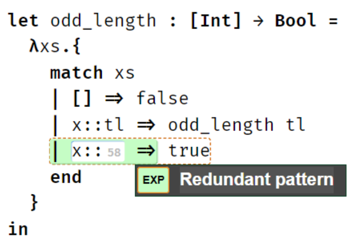
While we cannot guarantee exhaustiveness in Fig. 4a, such indeterminacy does not necessarily reflect an error on the user's part. Instead, they may simply be in the middle of on-going editing, working towards what will soon become an exhaustive expression. To avoid distracting the user with unnecessary information in these cases, for Hazel, we choose not to alert the user of indeterminate exhaustiveness. Instead, we only report an error when, regardless of hole-fillings, the expression is always inexhaustive. We say such a **match** is *necessarily inexhaustive*, and Fig. 4b provides an illustrative example. The first given rule given only matches the empty list, and the second rule only possibly matches lists with at least two elements, with this holding regardless of the choice of hole-fillings. Thus, in any hole-filling, a one element list such as `1 :: []` will not match any of the given patterns, and correspondingly, we display an error to the user. Note that the entire **match** expression is also placed within a non-empty hole. This is necessary to prevent evaluation from getting "stuck" with no applicable rule when the scrutinee witnesses the inexhaustiveness.

Finally, even with the presence of pattern holes, there are still cases where we can in fact guarantee exhaustiveness. Considering Fig. 4c, the first and third patterns together already cover all appropriately typed expressions. Thus, regardless of how the pattern hole in the second rule



```
let odd_length : [Int] → Bool =
  λxs.{
    match xs
    | [] ⇒ false
    | x:: 43 ⇒ true
    | x::y::tl ⇒ odd_length xs
    end
  }
in
```

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



```
let odd_length : [Int] → Bool =
  λxs.{
    match xs
    | [] ⇒ false
    | x::tl ⇒ odd_length tl
    | x:: 58 ⇒ true
    end     EXP Redundant pattern
  }
in
```

(b) Necessarily Redundant (third pattern)

Fig. 5. Redundancy Checking with Pattern Holes

is filled, the **match** expression as a whole will always be exhaustive, i.e. it is *necessarily exhaustive*. Note that we do not make a visual distinction between necessarily exhaustive and indeterminately exhaustive expressions, again because such information is more likely to be distracting than useful to the user. However, the semantic distinction is interesting to note, and it may be of use to future editor services designed around such holes.

3.3 Redundancy Checking with Pattern Holes

From a certain point of view, exhaustiveness checking guarantees a notion of *sufficiency* - that the patterns in a **match** expression are sufficient to cover all possible values of the scrutinee's type. Redundancy checking, on the other hand, guarantees a notion of *necessity* - that all the patterns in a **match** expressions are in fact required in that none is fully subsumed by other patterns earlier in the sequence. Explicitly, we say that a pattern p in a **match** expression is redundant if, for every value e of the scrutinee's type matching p, necessarily e also matches some pattern p' preceding p in the sequence of rules. Because a **match** expression considers rules one-by-one from the top down, a rule with a redundant pattern will be an unreachable code path.

As the reader should now anticipate, including pattern holes again requires us to reason conservatively about all possible hole-fillings, thereby introducing indeterminacy into our analysis. That is, a pattern may be either *necessarily redundant*, *necessarily irredundant*, or *indeterminately irredundant* due to the presence of expression or pattern holes. Similarly to how we handled exhaustiveness checking, we again only alert the user in cases where an error is guaranteed, i.e. when a rule is necessarily redundant.

Fig. 5 concretely demonstrates all of these possibilities, continuing with our running example of an `odd_length` function. In the editor state in Fig. 5a, there is a pattern hole in the second pattern. Depending on how this pattern hole is filled, the third pattern can be made either redundant or irredundant. Explicitly, if we filled the hole with the empty list `[]`, then the third pattern is the only one matching lists with more than one element, so it is irredundant. If instead we filled the hole with `y::tl`, then the second pattern is in fact the same as the third pattern, and obviously subsumes it. Thus, we can only soundly state that the third pattern is indeterminately irredundant.

For the second pattern itself, despite containing a pattern hole, we can still deem it necessarily irredundant. The only pattern preceding it is an empty list `[]`, while the second pattern only matches non-empty lists regardless of hole-filling. Thus, no hole-filling allows the second pattern to be made redundant by the first. Note that, vacuously, the first pattern is also necessarily irredundant - it cannot be subsumed by previous rules because there are no previous rules. Similarly, the first pattern of any **match** expression is necessarily irredundant, except in the case when the scrutinee's type has no values at all (e.g. is of `void` type). In this case, all rules are necessarily redundant.

Finally, the editor state in Fig. 5b showcases a necessarily redundant pattern. Explicitly, regardless of the choice of hole-filling, the third pattern can only be matched by a non-empty list. However, the second pattern already matches all such non-empty lists. Thus, the third pattern is necessarily redundant, and correspondingly, we report an error to the user. Note that, in this particular case, the first two rules are already exhaustive, so any pattern we could possibly place in the third rule will be necessarily redundant. More generally, stating that a pattern p is redundant is equivalent to stating that the preceding patterns exhaustively cover all values matching p. As we will explore in more detail in Sec. 4.5, this enables us to reduce the problem of redundancy checking to that of exhaustiveness checking.

4 PEANUT: A TYPED PATTERN HOLE CALCULUS

We now formalize the high-level discussion described in Sec. 3. While all of our work has been implemented into the full Hazel system, for ease of presentation, we distill our contribution into a bare-bones typed lambda calculus called Peanut. The core of our calculus is based upon the Hazelnut Live internal language [Omar et al. 2019], but extended to include pattern holes. We develop a dynamic semantics which allows live evaluation as was discussed in Sec. 3.1, and additionally, present a static semantics guaranteeing (indeterminate or necessary) exhaustiveness and irredundancy as was discussed in Sec. 3.2 and Sec. 3.3.

We begin in Sec. 4.1 by presenting the syntax of our calculus. In Sec. 4.2, we then provide a small-step dynamic semantics with support for evaluating incomplete programs. In Sec. 4.3, Sec. 4.4, and Sec. 4.5, we give the corresponding static semantics as a system of type assignment, making use a constraint language to reason about exhaustiveness and irredundancy. We defer discussion of decidability and implementation until Sec. 4.6.

4.1 Syntax

$$
\begin{array}{rcl}
e & ::= & x \mid \underline{n} \\
 & \mid & \lambda x : \tau . e \mid e_1(e_2) \\
 & \mid & (e_1, e_2) \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \\
 & \mid & \mathtt{inl}_\tau(e) \mid \mathtt{inr}_\tau(e) \\
 & \mid & \mathtt{match}(e)\{\hat{rs}\} \\
 & \mid & \llparenthesis\rrparenthesis^u \mid \llparenthesis e \rrparenthesis^u
\end{array}
\qquad
\begin{array}{rcl}
\tau & ::= & \mathtt{num} \mid (\tau_1 \to \tau_2) \mid (\tau_1 \times \tau_2) \mid (\tau_1 + \tau_2) \\
\hat{rs} & ::= & (rs \mid r \mid rs) \\
rs & ::= & \cdot \mid (r \mid rs') \\
r & ::= & p \Rightarrow e \\
p & ::= & x \mid \_ \mid \underline{n} \mid (p_1, p_2) \\
 & \mid & \mathtt{inl}(p) \mid \mathtt{inr}(p) \mid \llparenthesis\rrparenthesis^w \mid \llparenthesis p \rrparenthesis^w_\tau
\end{array}
$$

Fig. 6. Syntax

$\boxed{(\hat{rs})^\diamond = rs}$ 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. 7. Rule Pointer Erasure

Figure 6 presents the syntax of Peanut. Peanut closely mirrors the internal language of Hazelnut Live [Omar et al. 2019], a typed lambda calculus with expression holes which provides the base of the Hazel system. To ensure the generality of our approach, we attempt to include only those forms necessary to have a rich discussion of pattern matching. As a result, we remove most of the machinery related to gradual typing [Siek et al. 2015], but such an extension is fairly straightforward to implement. Most of the forms are standard, following a formulation outlined in [Harper 2012].

Unsurprisingly, we include lambda functions and function application. We choose natural numbers as our base type. In order to create interesting expressions to pattern match on, we also allow the formation of binary sum and binary product types. Correspondingly, we include pairs, projection operators, and left and right injections. (Note that pattern matching generally subsumes the need for explicit projections operators, but we include such forms here for reasons discussed later.) To simplify our type assignment system, we also require injections and functions to have explicit type annotations. Such annotations are fairly innocuous, as Peanut represents an internal language, so if desired, annotations can always be inserted during elaboration. Finally, we include the main forms of interest: holes, patterns, and match expressions.

As discussed in Sec. 1, holes come in two variants. Empty expression holes are written $(\!|\,|\!)^u$ and indicate missing syntactic pieces, while non-empty holes expression holes are written $(\!|e|\!)^u$, acting as a membrane around a type inconsistency at the expression e. Analogously, empty pattern holes are written $(\!|\,|\!)^w$ and indicate missing sub-terms of a pattern, while non-empty pattern holes are written $(\!|p|\!)^w_\tau$ and surround a type-inconsistency, with τ recording the type of the wrapped expression p. Here, the labels u and w are identifiers for the corresponding holes. As Peanut represents an internal language, distinct hole identifiers should correspond to distinct holes in the original program source code. However, as evaluation can lead to holes being replicated during substitution, we do not enforce a uniqueness constraint on hole identifiers within Peanut itself.

Outside of holes, patterns also include variables, a wildcard (_), and forms corresponding to each constructor for values. Semantically, the wildcard is matched by any expression, allowing the user to indicate a default case. Match expressions $\texttt{match}(e)\{\hat{rs}\}$ then consist of a scrutinee e and a list of rules $\hat{rs}$. As we wish to support live evaluation, we must be able to represent intermediate execution states of the match expression. Correspondingly, the list $\hat{rs}$ follows Huet's zipper construction [Huet 1997], effectively containing a pointer to the current rule under consideration. Syntactically, this is given by a triple, $rs_{pre} \mid r \mid rs_{post}$, with a prefix list of rules already considered, rs_{pre}, the current rule, r, and a suffix list of rules yet to be considered, rs_{post} . As defined in Fig. 7, We can erase this pointer through the pointer erasure operator, $(\hat{rs})^\diamond$, yielding an unzippered list. Each rule is of the form $p \Rightarrow e$, where p is a pattern and e is the corresponding branch expression.

4.2 Dynamic Semantics

Dynamically, Peanut seeks to extend Hazelnut Live [Omar et al. 2019] while maintaining the ability to evaluate "around" expressions with holes. That is, upon encountering a hole, Peanut should delay its evaluation as long as possible, then proceed to take all other evaluation steps which do not rely on the eventual contents of the hole. For all non-pattern matching forms, our rules correspond exactly to the rules of Hazelnut Live, albeit, formulated as a small-step operational semantics rather than a contextual one. Fig. 8 displays this stepping judgement $e \mapsto e'$, indicating that evaluating an expression e one step yields the expression e'.

To begin, consider how one typically evaluates, say, a function application $e(e')$ in a strict language without holes. Initially, the function e is stepped as far as possible, continuing until it is reduced to a value. Next, evaluation steps the argument e', again continuing until it is reduced to a value. Once all these reductions have occurred, only then is the argument actually substituted into the function body. Notably, the key ingredient to this process is the ability to detect when an expression has been reduced "as far as possible", or equivalently, when an expression has been reduced to a value.

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

$$\frac{e \mapsto e'}{⦇e⦈^u \mapsto ⦇e'⦈^u}\text{ITHole} \qquad \frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)}\text{ITApFun} \qquad \frac{e_1\ \mathtt{final} \qquad e_2 \mapsto e_2'}{e_1(e_2) \mapsto e_1(e_2')}\text{ITApArg}$$

$$\frac{e_2\ \mathtt{final}}{\lambda x : \tau.e_1(e_2) \mapsto [e_2/x]e_1}\text{ITAp} \qquad \frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)}\text{ITPairL}$$

$$\frac{e_1\ \mathtt{final} \qquad e_2 \mapsto e_2'}{(e_1, e_2) \mapsto (e_1, e_2')}\text{ITPairR} \qquad \frac{e_1 \mapsto e_1'}{\mathtt{fst}(e_1) \mapsto \mathtt{fst}(e_1')}\text{ITFst} \qquad \frac{e_1 \mapsto e_1'}{\mathtt{snd}(e_1) \mapsto \mathtt{snd}(e_1')}\text{ITSnd}$$

$$\frac{(e_1, e_2)\ \mathtt{final}}{\mathtt{fst}((e_1, e_2)) \mapsto e_1}\text{ITFstPair} \qquad \frac{(e_1, e_2)\ \mathtt{final}}{\mathtt{snd}((e_1, e_2)) \mapsto e_2}\text{ITSndPair} \qquad \frac{e \mapsto e'}{\mathtt{inl}_\tau(e) \mapsto \mathtt{inl}_\tau(e')}\text{ITInl}$$

$$\frac{e \mapsto e'}{\mathtt{inr}_\tau(e) \mapsto \mathtt{inr}_\tau(e')}\text{ITInr} \qquad \frac{e \mapsto e'}{\mathtt{match}(e)\{\hat{rs}\} \mapsto \mathtt{match}(e')\{\hat{rs}\}}\text{ITExpMatch}$$

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

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

Fig. 8. Stepping

In the presence of pattern and expression holes, evaluation proceeds in much the same way. Fig. 9 defines a judgement e `final` indicating when an expression e is *final*, i.e when no further evaluation steps can occur. The rules ITApFun, ITApArg, and ITAp are then just as we described in our example. Crucially, however, holes extend the notion of finality to include not just values, but *indeterminate* forms as well. That is, there are terms which have indeed been reduced as far as possible, but they still contain holes in the end result, and they will require further evaluation if such holes are filled at a later point in time. We differentiate between these cases with the judgement e `val`, characterizing values, and the judgement e `indet`, characterizing indeterminate expressions. Correspondingly, the e `final` judgement is given as a disjunction of these two cases.

For Peanut, the main task is to describe the behavior of a match expression $\mathtt{match}(e)\{\hat{rs}\}$. First, as described by ITExpMatch, we step the scrutinee e, eventually yielding either a value or indeterminate form. Once e is final, we then proceed to pattern match, comparing e to each pattern sequentially from the top down. If e indeed matches a pattern, then we step to the corresponding branch expression, applying appropriate substitutions for the bound variables as in the conclusion of ITSuccMatch. Instead, if e does not match the pattern as in ITFailMatch, then we move to consider the next rule, using the pointer erasure operator of Fig. 7 to increment the zipper. Note that, per our previous discussion, an expression may also indeterminately match a pattern. The IMatch rule in the e `indet` judgement covers this case, indicating that we cannot safely proceed past an indeterminate pattern match, leading to the entire match expression being indeterminate without further hole-filling.

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

$$\frac{}{\underline{n}\ \texttt{val}}\ \text{VNum} \qquad \frac{}{\lambda x : \tau.e\ \texttt{val}}\ \text{VLam} \qquad \frac{e_1\ \texttt{val} \qquad e_2\ \texttt{val}}{(e_1, e_2)\ \texttt{val}}\ \text{VPair} \qquad \frac{e\ \texttt{val}}{\texttt{inl}_\tau(e)\ \texttt{val}}\ \text{VInl}$$

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

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

$$\frac{}{\llparenthesis\rrparenthesis^u\ \texttt{indet}}\ \text{IEHole} \qquad \frac{e\ \texttt{final}}{\llparenthesis e\rrparenthesis^u\ \texttt{indet}}\ \text{IHole} \qquad \frac{e_1\ \texttt{indet} \qquad e_2\ \texttt{final}}{e_1(e_2)\ \texttt{indet}}\ \text{IAp}$$

$$\frac{e_1\ \texttt{indet} \qquad e_2\ \texttt{val}}{(e_1, e_2)\ \texttt{indet}}\ \text{IPairL} \qquad \frac{e_1\ \texttt{val} \qquad e_2\ \texttt{indet}}{(e_1, e_2)\ \texttt{indet}}\ \text{IPairR} \qquad \frac{e_1\ \texttt{indet} \qquad e_2\ \texttt{indet}}{(e_1, e_2)\ \texttt{indet}}\ \text{IPair}$$

$$\frac{e\ \texttt{indet} \qquad e \neq (e_1, e_2)}{\texttt{fst}(e)\ \texttt{indet}}\ \text{IFst} \qquad \frac{e\ \texttt{indet} \qquad e \neq (e_1, e_2)}{\texttt{snd}(e)\ \texttt{indet}}\ \text{ISnd} \qquad \frac{e\ \texttt{indet}}{\texttt{inl}_\tau(e)\ \texttt{indet}}\ \text{IInl}$$

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

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

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

Fig. 9. Final expressions

Our stepping and finality judgements indeed cover all possible cases. The following theorem states this, combining progress and determinism. Here, the judgement $\cdot\,;\Delta \vdash e : \tau$ indicates that e is a closed term of type τ, as discussed further in Sec. 4.4 when presenting Peanut's static semantics.

Theorem 4.1 (Deterministic Progress). *If $\cdot\,;\Delta \vdash e : \tau$ then exactly one of the following holds*

(1) $e\ \texttt{val}$
(2) $e\ \texttt{indet}$
(3) $e \mapsto e'$ *for some unique* e'

Essential to the aforementioned rules are the judgements determining whether an expression e must match, must not match, or indeterminately matches a pattern p. Fig. 10 presents these three cases. First, the judgement $e \rhd p \dashv\!\mid \theta$ indicates that e successfully matches p, emitting a series of corresponding substitutions θ for the variables bound in p. Next, the judgement $e \perp p$ indicates that e does not match p. Finally, the judgement $e\ ?\ p$ indicates that e indeterminately matches p depending on the eventual contents of holes in e or p. These judgements correspond respectively to the rules ITSuccMatch, ITFailMatch, and IMatch discussed above, appearing as a premise in each.

$\boxed{e \triangleright p \dashV \theta}$ e matches p, emitting θ

$$\frac{}{e \triangleright x \dashV e/x}\text{MVar} \qquad \frac{}{e \triangleright \_ \dashV \cdot}\text{MWild} \qquad \frac{}{\underline{n} \triangleright \underline{n} \dashV \cdot}\text{MNum}$$

$$\frac{e_1 \triangleright p_1 \dashV \theta_1 \qquad e_2 \triangleright p_2 \dashV \theta_2}{(e_1, e_2) \triangleright (p_1, p_2) \dashV \theta_1 \uplus \theta_2}\text{MPair} \qquad \frac{e \triangleright p \dashV \theta}{\texttt{inl}_\tau(e) \triangleright \texttt{inl}(p) \dashV \theta}\text{MInl}$$

$$\frac{e \triangleright p \dashV \theta}{\texttt{inr}_\tau(e) \triangleright \texttt{inr}(p) \dashV \theta}\text{MInr}$$

$$\frac{e\ \texttt{notintro} \qquad \texttt{fst}(e) \triangleright p_1 \dashV \theta_1 \qquad \texttt{snd}(e) \triangleright p_2 \dashV \theta_2}{e \triangleright (p_1, p_2) \dashV \theta_1 \uplus \theta_2}\text{MNotIntroPair}$$

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

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

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

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

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

$$\frac{}{e \,?\, \llparenthesis\rrparenthesis^w}\text{MMEHole} \qquad \frac{}{e \,?\, \llparenthesis p \rrparenthesis^w_\tau}\text{MMHole}$$

$$\frac{e\ \texttt{notintro} \qquad p\ \texttt{refutable}_?}{e \,?\, p}\text{MMNotIntro} \qquad \frac{e_1 \,?\, p_1 \qquad e_2 \triangleright p_2 \dashV \theta_2}{(e_1, e_2) \,?\, (p_1, p_2)}\text{MMPairL}$$

$$\frac{e_1 \triangleright p_1 \dashV \theta_1 \qquad e_2 \,?\, p_2}{(e_1, e_2) \,?\, (p_1, p_2)}\text{MMPairR} \qquad \frac{e_1 \,?\, p_1 \qquad e_2 \,?\, p_2}{(e_1, e_2) \,?\, (p_1, p_2)}\text{MMPair} \qquad \frac{e \,?\, p}{\texttt{inl}_\tau(e) \,?\, \texttt{inl}(p)}\text{MMInl}$$

$$\frac{e \,?\, p}{\texttt{inr}_\tau(e) \,?\, \texttt{inr}(p)}\text{MMInr}$$

Fig. 10. Three possible outcomes of pattern matching

Lemma 4.2 states that, as desired, these three matching judgements are exclusive and cover all possible cases. Here, the judgement again $\cdot\, ; \Delta \vdash e : \tau$ specifies that e is a closed term of type τ, while the judgement $\Delta \vdash p : \tau \dashV \Gamma$ indicates that we can also assign the type τ to our pattern p. Note that because of the aforementioned correspondence between these matching judgements and the semantics of the match expression, Lemma 4.2 encapsulates the majority of the work required to prove Theorem 4.1.

Lemma 4.2 (Matching Determinism). *If e $\texttt{final}$ and $\cdot\,;\Delta \vdash e : \tau$ and $\Delta \vdash p : \tau \dashv\!\mid \Gamma$ then exactly one of the following holds*

(1) $e \rhd p \dashv\!\mid \theta$ *for some* θ
(2) $e\,?\,p$
(3) $e \perp p$

Let us now explore these pattern matching judgements in more depth. Note that semantics of our expression guarantees that we only perform pattern matching on well-typed and final scrutinees (per the inclusion of e $\texttt{final}$ as a premise to ITSuccMatch and ITFailMatch). Throughout this discussion, we then consider matching an expression e against a pattern p, assuming e is final and that e and p are of the same type.

If we ignore cases of indeterminacy, most of the rules are fairly straightforward. For the judgement $e \rhd p \dashv\!\mid \theta$, each rule specifies that e and p have the same outermost constructor, and that their corresponding subterms match inductively. Conversely, the rules for $e \perp p$ identify cases where either the outermost constructors of e and p mismatch (e.g. NMConfL and NMConfR), or where their outermost constructors agree, but some corresponding subterm inductively fails to match (e.g. NMPairL and NMPairR). To reason about indeterminacy, however, we require a few auxillary judgements which are defined in Fig. 11 and Fig. 12.

$\boxed{e\ \texttt{notintro}}$ 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. 11. Expressions that are syntactically not values

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

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

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

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

Fig. 12. Refutable Patterns

As we consider only final scrutinees, in the context of our matching judgements, an expression e is indeterminate if and only if it is not a value. Fig. 11 defines a judgement e `notintro` which characterizes this case, syntactically analyzing the outermost constructor of e to determine that it cannot be a value. Note that an indeterminate e will be further evaluated after hole filling, so correspondingly, our rules should treat such an e as opaque, never inspecting any of its subterms for matching purposes. However, even with this restriction, there are still cases where an indeterminate e must match a pattern p. In particular, a pattern p could be irrefutable in the sense that it must be matched by all expressions of the same type, say, if it is a variable x or wildcard _. Correspondingly, Fig. 12 defines a judgement p $\texttt{refutable}_?$ indicating that p is not necessarily irrefutable, i.e. there is some hole filling which allows at least one expression to fail to match p.

Together, the combination of these judgements allows us to handle matching with indeterminacy. The rule MMNotIntro gives that a match is indeterminate whenever e is indeterminate, so long as p is not necessarily irrefutable. The rules MMEHole and MMHole state that directly matching against a hole will always be indeterminate. All other indeterminate matches arise inductively from some subterms indeterminately matching. Conversely, even in the case when e is indeterminate, we wish to have e match an irrefutable pattern p. The rules MVar and MWild give two explicit cases of this. In MNotIntroPair, the irrefutability premise is implicit - a term `fst`(e) or `snd`(e) can only match an irrefutable pattern, but we explicitly derive the matching judgements in order to emit appropriate substitutions. This also motivates the inclusion of explicit projection operators, despite the fact that pattern matching also allows deconstructing of pairs.

4.3 Match Constraint Language

With the dynamic semantics of Peanut defined, we now turn to the problem of statically reasoning about the runtime behavior of our programs. We extend an idea outlined in [Harper 2012] by introducing a *match constraint language*. Intuitively, constraints encode the logic of patterns, with each constraint corresponding to the restriction that a pattern puts on those expression matching it. Further, we allow taking boolean combinations of constraints using negation, logical and ($\wedge$), and logical or ($\vee$). In Sec. 4.4, we describe how constraints are generated by patterns. In Sec. 4.5, we then describe how constraints enable static reasoning about exhaustiveness and irredundancy.

The syntax of the constraint language is displayed in Fig. 13. Note that each pattern indeed has a corresponding constraint of the same syntactic form. The only exception here is that variables and wildcards both map to constraint ($\top$), as they are both matched by any expression, and both forms of pattern holes map to an unknown constraint (?). We specify that a constraint ξ restricts expressions of type τ using the constraint type assignment judgement $\xi : \tau$.

Ignoring indeterminacy and ? for now, one can explicitly model constraints of type τ as the Boolean algebra of sets of final expressions of type τ under intersection, union, and complement. That is, a constraint $\xi : \tau$ can be interpreted as the set of all final expressions e of type τ such that e *satisfies* the restriction represented by ξ. The truth constraint $\top$ and falsity constraint $\bot$ are then interpreted as the whole set and the empty set respectively. Likewise, Fig. 13 defines a negation operator called the *dual* of a constraint. Written $\overline{\xi}$, the dual of ξ is satisfied by all appropriately-typed final expressions not satisfying ξ. In our model, $\overline{\xi}$ identifies the complement of the set identified by ξ. Up to equivalence, the usual laws of Boolean logic hold, e.g. the dual of $\overline{\xi}$ is equivalent to ξ.

Just as we introduced pattern holes into patterns, we include an unknown constraint (?) in our constraint language to represent indeterminacy. Correspondingly, we have analogs of the three pattern matching judgements - for a final expression e and constraint ξ of the same type as e, either e *must satisfiy* ξ, e *must not satisfy* ξ, or e *indeterminately satisfies* ξ due to the presence of holes or the unknown constraint. Fig. 14 explicitly defines these judgements. The judgement $e \models \xi$ specifies

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

$\boxed{\xi : \tau}$ ξ is of type τ

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

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

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

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

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

$\boxed{\overline{\xi_1} = \xi_2}$ dual of ξ_1 is ξ_2

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

Fig. 13. Match Constraints

that e satisfies ξ, the judgement $e \models_? \xi$ specifies that e may satisfy ξ, and the judgement $e \models^{\dagger}_{?} \xi$ provides the disjunction of these two cases. Note that we do not include a corresponding "e does not satisfy ξ" judgement, but rather reason about the non-derivability of $e \models^{\dagger}_{?} \xi$.

The rules for each of these satistfaction judgements almost exactly mirror the corresponding rules for the pattern matching judgements. The only notable difference is between the rules MMNotIntro and CMSNotIntro. Recall that MMNotIntro captures the notion that an indeterminate expression e should indeterminately match any pattern p, excluding the edge case where p is matched by all expressions in all hole-fillings. While CMSNotIntro captures this same notion, it additionally must consider the edge case where a constraint ξ is impossible to match at all. With patterns, this is not a concern, as every pattern is matchable by at least one expression. To accomplish this, the ξ `possible` judgement defined in Fig. 15 specifies that ξ is possibly satisfied by at least one expression, i.e. it is not equivalent to $\bot$. We also define a ξ $\texttt{refutable}_?$ judgement in Fig. 16, which exactly corresponds to the p $\texttt{refutable}_?$ judgement for patterns.

Expectedly, one can prove that there is indeed a correspondence between a pattern and its emitted constraint. The judgement $\Delta \vdash p : \tau[\xi] \dashv\mid \Gamma$ indicates that a pattern p emits the constraint ξ, but we defer a more in depth discussion until Sec. 4.4.

Lemma 4.3 (Matching Coherence of Constraint). *Suppose that* $\cdot\, ; \Delta_e \vdash e : \tau$ *and* e `final` *and* $\Delta \vdash p : \tau[\xi] \dashv\mid \Gamma$. *Then we have*

(1) $e \models \xi$ *iff* $e \rhd p \dashv\mid \theta$
(2) $e \models_? \xi$ *iff* $e \,?\, p$
(3) $e \not\models^{\dagger}_{?} \xi$ *iff* $e \perp p$

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

$$\frac{}{e \models \top}\text{CSTruth} \qquad \frac{}{\underline{n} \models \underline{n}}\text{CSNum} \qquad \frac{n_1 \neq n_2}{\underline{n_1} \models \underline{\not{n_2}}}\text{CSNotNum} \qquad \frac{e \models \xi_1 \qquad e \models \xi_2}{e \models \xi_1 \wedge \xi_2}\text{CSAnd}$$

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

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

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

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

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

Fig. 14. Satisfaction

Combining Lemma 4.3 with Lemma 4.2, we may also verify that the various constraint satisfaction judgements are mutually exclusive and cover all possibilities.

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

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

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

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

Fig. 15. Possible Constraints

$\boxed{\xi\ \texttt{refutable}_?}$ ξ is refutable

$$\frac{}{\bot\ \texttt{refutable}_?}\ \text{RXFalsity} \qquad \frac{}{?\ \texttt{refutable}_?}\ \text{RXUnknown} \qquad \frac{}{\underline{n}\ \texttt{refutable}_?}\ \text{RXNum}$$

$$\frac{}{\underline{\not{n}}\ \texttt{refutable}_?}\ \text{RXNotNum} \qquad \frac{}{\texttt{inl}(\xi)\ \texttt{refutable}_?}\ \text{RXInl} \qquad \frac{}{\texttt{inr}(\xi)\ \texttt{refutable}_?}\ \text{RXInr}$$

$$\frac{\xi_1\ \texttt{refutable}_?}{(\xi_1, \xi_2)\ \texttt{refutable}_?}\ \text{RXPairL} \qquad \frac{\xi_2\ \texttt{refutable}_?}{(\xi_1, \xi_2)\ \texttt{refutable}_?}\ \text{RXPairR}$$

Fig. 16. Refutable Constraints

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) $e \models \xi$
(2) $e \models_? \xi$
(3) $e \not\models^{\dagger}_{?} \xi$

With the setup of our constraint language clear, let us return to the issue of redundancy and exhaustiveness checking. For now, we focus only on defining these notions in terms of constraints, deferring discussion of their actual implementation until Sec. 4.6.

Consider if we have a zippered list of rules $rs_{pre} \mid r \mid rs_{post}$ with the currently considered rule r given by $p \Rightarrow er$. Recalling the discussion in Sec. 3.3, we state that r is redundant if it is unreachable in any hole filling, i.e. if any expression which could possibly reach r will instead match against one of the rules in rs_{pre}. Stated formally, r will be redundant if for all appropriately-typed final expressions e, if either $e \rhd p \dashv\!\!| \theta$ or $e\ ?\ p$ is derivable then so is $e \rhd p' \dashv\!\!| \theta'$ for some pattern p' in a previous rule. Using Theorem 4.3, we can translate this into a statement about constraints. That is, a constraint ξ is redundant if any appropriately-typed expression e with $e \models^{\dagger}_{?} \xi$ necessarily has $e \models \xi_{pre}$ for some constraint ξ_{pre} emitted by a pattern earlier in the sequence. In fact, as we may take the logical or of constraints, we can consider just a single ξ_{pre} taken as the disjunction of all previously emitted constraints. This inspires the following definition of *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 $e\ \mathtt{val}$ we have $e \models^{\dagger}_{?} \xi_1$ implies $e \models \xi_2$

Correspondingly, if ξ_{pre} is the disjunction of all constraints emitted by the previously considered rules in a match, and if ξ is the constraint emitted by the current rule, then the current rule is necessarily redundant if and only if $\xi \models \xi_{pre}$. Conversely, to ensure a rule is either necessarily or indeterminately irredundant, we must check that $\xi \not\models \xi_{pre}$. In our yet-to-be-discussed static system, such checks are added as a premise in the typing rules TOneRules and TRules.

It is worth emphasizing that the above definition identifies *necessary* redundancy rather than *indeterminate* redundancy. Indeed, indeterminate redundancy may be resolved by further hole-filling, so we do not yet desire to report an error to the user. Likewise with exhaustiveness checking, we only wish to report an error in cases of *necessary* inexhaustiveness. To that end, conversely, we must be able to identify when a constraint is either *necessarily* exhaustive or *indeterminately* exhaustive. This is captured by a slightly weaker notion of entailment.

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 $e\ \mathtt{final}$ we have $e \models^{\dagger}_{?} \xi_1$ implies $e \models^{\dagger}_{?} \xi_2$

We can then state that a constraint ξ is either necessarily or indeterminately exhaustive exactly when $\top \models^{\dagger}_{?} \xi$, relying on the fact that every expression possibly satisfies the truth constraint $\top$. That is, ξ is not necessarily inexhaustive if every expression possibly satisfies ξ.

$\boxed{e' \in \mathtt{values}[\Delta](e)}$ e' is one of the possible values of e

$$\frac{e\ \mathtt{val} \qquad \cdot ; \Delta \vdash e : \tau}{e \in \mathtt{values}[\Delta](e)}\ \text{IVVal} \qquad \frac{e\ \mathtt{notintro} \qquad \cdot ; \Delta \vdash e : \tau \qquad e'\ \mathtt{val} \qquad \cdot ; \Delta \vdash e' : \tau}{e' \in \mathtt{values}[\Delta](e)}\ \text{IVIndet}$$

$$\frac{e_1' \in \mathtt{values}[\Delta](e_1)}{\mathtt{inl}_{\tau_2}(e_1') \in \mathtt{values}[\Delta](\mathtt{inl}_{\tau_2}(e_1))}\ \text{IVInl} \qquad \frac{e_2' \in \mathtt{values}[\Delta](e_2)}{\mathtt{inr}_{\tau_1}(e_2') \in \mathtt{values}[\Delta](\mathtt{inr}_{\tau_1}(e_2))}\ \text{IVInr}$$

$$\frac{e_1' \in \mathtt{values}[\Delta](e_1) \qquad e_2' \in \mathtt{values}[\Delta](e_2)}{(e_1', e_2') \in \mathtt{values}[\Delta]((e_1, e_2))}\ \text{IVPair}$$

Fig. 17. Values

Note that the definition of potential entailment quantifies over all final expressions rather than just those e with $e\ \mathtt{val}$. Resultingly, so long as the constraint ξ emitted by a list of rules has $\top \models^{\dagger}_{?} \xi$, any scrutinee, either a value or indeterminate form, will possibly match at least one of the patterns in the rule, preventing evaluation from getting stuck. Such a definition simplifies the proof of progress in Theorem 4.1. However, as the following lemma states, exhaustiveness over values also ends up being sufficient.

Lemma 4.7. *Suppose* $\dot{\xi} : \tau$. *Then* $e \models^{\dagger}_{?} \dot{\xi}$ *for all* e *such that* $\cdot ; \Delta \vdash e : \tau$ *and* $e\ \mathtt{final}$ *if and only if* $e \models^{\dagger}_{?} \dot{\xi}$ *for all* e *such that* $\cdot ; \Delta \vdash e : \tau$ *and* $e\ \mathtt{val}$.

To prove this, we reason about the possible values that result when filling holes in an indeterminate expression. Formally, the judgement $e' \in \mathtt{values}[\Delta](e)$ defined in Fig. 17 indicates that e' is one of the possible values of e after hole-filling. Here, Δ is a hole context used for typing information, which we discuss further in Sec. 4.4. Using such a judgment, Lemma 4.7 follows straightforwardly from the ensuing lemma.

Lemma 4.8. *Assume* e $\texttt{final}$ *and* $\cdot\,;\Delta \vdash e : \tau$ *and* $\dot{\xi} : \tau$. *If* $e \not\vDash {}^{\dagger}_{?}\dot{\xi}$, *then for any* e' *with* $e' \in \texttt{values}[\Delta](e)$ *we also have* $e' \not\vDash {}^{\dagger}_{?}\dot{\xi}$.

4.4 Static Semantics

We are now ready to present the static semantics of Peanut, using the discussed match constraint language to enforce exhaustiveness and irredundancy of well-typed terms. Again, our type system is based on the internal language of Hazelnut Live [Omar et al. 2019], but extended to include typing of both patterns and expressions. Throughout, we use expression contexts Γ to map variable names to types, and we use hole contexts Δ to map both expression and pattern hole names to their corresponding types.

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

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

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

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

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

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

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

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

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

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

Fig. 18. Typing of Patterns, Single Rules, and Series of Rules

4.4.1 Typing of Rules and Emitted Constraints. We begin by formalizing pattern typing and the relationship between patterns and constraints. Fig. 18 presents three judgements specifying typing for patterns p, a single rule r, and a series of rules rs. All of the judgements here record the emitted constraints, and additionally, are used to enforce irredundancy.

For patterns, the judgement $\Delta \vdash p : \tau[\xi] \dashv\!\mid \Gamma$ specifies that, in the hole context Δ, the pattern p is assigned the type τ, emits the constraint ξ, and makes typing assumptions about bound variables as recorded in the expression context Γ. Note that, unlike in any other of our typing judgements, the expression context Γ is morally an output for pattern typing rather than an input. Rules PTVar and PTWild specify that wildcards and variables may be assigned any time, and as they may be matched by any expression, they emit only a truth constraint $\top$. Rules PTEHole and PTHole specify that pattern holes emit the unknown constraint ?, inductively checking that the content of a non-empty hole are also well-typed. All of these rules should be unsurprising.

Once we have specified pattern typing, we can then check the type of a rule. Intuitively, a rule $p \Rightarrow e$ if both p and e are well-typed, with e possibly referencing the variables bound in p. Correspondingly, when we specify the type of e, we use the context $\Gamma \uplus \Gamma_p$, where Γ_p records assumptions about the types of bound variables in p as emitted by the judgement $\Delta \vdash p : \tau[\xi] \dashv\!\mid \Gamma_p$. Note that we need never extend the context Δ, as we assume it to already include all expression and hole types in the original program source code. In summary then, the rule typing judgement $\Gamma ; \Delta \vdash (p \Rightarrow e) : \tau[\xi] \Rightarrow \tau'$ specifies that r matches expressions of type τ and evaluates to a corresponding branch expression of type τ', emitting constraint ξ.

Finally, for later use in type checking match expression, we define typing for an entire sequence of rules rs. With the full rule sequence now available, we also are able to enforce irredundancy. Intuitively, the judgement $\Gamma ; \Delta \vdash [\xi_{pre}]rs : \tau[\xi_{rs}] \Rightarrow \tau'$ states that all of the rules in rs are matched by expressions of the same type τ, and they all have corresponding branch expressions of the same type τ'. Moreover, it states that the disjunction of the all rules together emits the constraint ξ_{rs}, and crucially, that this emitted constraint does not entail another constraint ξ_{pre}. Morally, ξ_{pre} is an input to the judgement which records the constraint emitted by all previously considered rules. Thus, checking that $\xi_{rs} \not\models \xi_{pre}$ ensures irredundancy.

To accomplish this, the rule TOneRules delegates to the single rule typing judgement, while also adding a premise $\xi_r \not\models \xi_{pre}$ to ensure irredundancy. For the inductive case of a list $r \mid rs$, we consider the rules one-by-one, analogously to how a match expression considers rules one-by-one from the top down. Correspondingly, the first premise of TRules specifies that the head of the list r is well-typed and irredundant with respect to ξ_{pre}. In turn, the second premise takes the constraint ξ_r emitted by r, appends it the constraint ξ_{pre} emitted by all previously considered rules, then inductively uses this as input when checking the tail of the list rs. At the same time, we record the constraint emitted by $r \mid rs$ as $\xi_r \vee \xi_{rs}$, the disjunction of the constraint emitted by r and that emitted by rs.

4.4.2 Typing of Expressions. We are now able to present the full typing judgement for expressions. For all non-pattern matching forms, the rules are standard, with our inclusion of type annotations on functions and injections enabling a simple system of type assignment. Fig. 19 displays the definitions for the judgement $\Gamma ; \Delta \vdash e : \tau$ indicating that an expression e has type τ, where the types of free variables are recorded in a context Γ, and the types of expression and pattern holes from the original source code are recorded in a context Δ (recall that Peanut is an internal language). The only rules of interest to us are those for the match expression, TMatchZPre and TMatchNZPre.

Considering the zippered rule list, TMatchZPre represent the case where have not yet started pattern matching, and there are no previously considered rules. The first premise checks that the scrutinee is well-typed. The second premise ensures that the rules are all matchable by expressions

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

$$\frac{}{\Gamma, x : \tau\,; \Delta \vdash x : \tau}\text{TVar} \qquad \frac{}{\Gamma\,; \Delta, u :: \tau \vdash (\!|\!)^u : \tau}\text{TEHole} \qquad \frac{\Gamma\,; \Delta, u :: \tau \vdash e : \tau'}{\Gamma\,; \Delta, u :: \tau \vdash (\!|e|\!)^u : \tau}\text{THole}$$

$$\frac{}{\Gamma\,; \Delta \vdash \underline{n} : \mathsf{num}}\text{TNum} \qquad \frac{\Gamma, x : \tau_1\,; \Delta \vdash e : \tau_2}{\Gamma\,; \Delta \vdash \lambda x : \tau_1.e : (\tau_1 \to \tau_2)}\text{TLam}$$

$$\frac{\Gamma\,; \Delta \vdash e_1 : (\tau_2 \to \tau) \qquad \Gamma\,; \Delta \vdash e_2 : \tau_2}{\Gamma\,; \Delta \vdash e_1(e_2) : \tau}\text{TAp} \qquad \frac{\Gamma\,; \Delta \vdash e_1 : \tau_1 \qquad \Gamma\,; \Delta \vdash e_2 : \tau_2}{\Gamma\,; \Delta \vdash (e_1, e_2) : (\tau_1 \times \tau_2)}\text{TPair}$$

$$\frac{\Gamma\,; \Delta \vdash e : (\tau_1 \times \tau_2)}{\Gamma\,; \Delta \vdash \mathtt{fst}(e) : \tau_1}\text{TFst} \qquad \frac{\Gamma\,; \Delta \vdash e : (\tau_1 \times \tau_2)}{\Gamma\,; \Delta \vdash \mathtt{snd}(e) : \tau_2}\text{TSnd} \qquad \frac{\Gamma\,; \Delta \vdash e : \tau_1}{\Gamma\,; \Delta \vdash \mathtt{inl}_{\tau_2}(e) : (\tau_1 + \tau_2)}\text{TInl}$$

$$\frac{\Gamma\,; \Delta \vdash e : \tau_2}{\Gamma\,; \Delta \vdash \mathtt{inr}_{\tau_1}(e) : (\tau_1 + \tau_2)}\text{TInr}$$

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

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

Fig. 19. Expression Typing

of the same type τ, and that all have branch expressions of type τ'. Correspondingly, as a the match expression will ultimately evaluate to one of these branch expressions, the type of the entire match is also τ'. Note that, because there are no previously considered rules, the second premise need only check irredundancy with respect to the constraint $\bot$, which should never be entailed by any constraint ξ with ξ `possible` (and in particular, any constraint emitted by a pattern).

For TMatchNZPre, we are now considering the case where we are in the midst of pattern matching, having already considered the rules rs_{pre} and currently considering the rule r. As pattern matching should only proceed once a scrutinee is fully evaluated, we add a premise requiring e `final`. Additionally, to ensure that it was valid to reach this point in the rule list, the scrutinee should not have matched any previously considered rules, hence we add a premise $e \not\models^{\dagger}_{?} \xi_{pre}$. Indeed, Theorem 4.3 ensures this is the case. As well, we again require that the rules are well-typed, checking this piecewise - first with the previously considered rules rs_{pre}, then with the remaining rules $r \mid rs_{post}$. Finally, we ensure exhaustiveness on the total emitted constraint as $\top \models^{\dagger}_{?} \xi_{pre} \vee \xi_{rest}$.

Thus, our system indeed ensures a term is well-typed only if it also both exhaustive and irredundant, either necessarily or indeterminately. Alternatively, one could also extract these checks as their own judgements separate from the type system, although exhaustiveness is still required to ensure progress in Theorem 4.1. We take this alternative in the Agda mechanization of Sec. 5.

4.4.3 Type Safety. The type safety of Peanut is established by Theorem 4.1 and Theorem 4.9.

Theorem 4.9 (Preservation). *If* $\cdot\,; \Delta \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot\,; \Delta \vdash e' : \tau$

While we do not enumerate them here, the proof of preservation relies on a number of small lemmas showing that our various typing and finality judgements are well-behaved with respect to substitution (in particular, when applying a substitution list θ emitted by the judgement $e \rhd p \dashv\!\mid \theta$). The details can be found in our mechanization.

4.5 Eliminating Indeterminacy

Thus far, we have managed to describe a type system encoding exhaustiveness and irredundancy checking using constraint entailment. However, it is quite unclear how to actually check when such entailment holds - our definitions require quantifying over all final expressions of a given type, yielding no obvious decision procedure. To make matters worse, our definitions also involve indeterminate satisfaction, requiring us to reason conservatively about all possible hole-fillings through the unknown constraint (?). In this section, as a first step towards resolving these complications, we discuss how to remove all such indeterminacy, eliminating any use of ? from the constraints involved in an entailment. We forgo discussion of the full decision procedure until Sec. 4.6.

Recall that, for exhaustiveness, we only report an error when a match expression is *necessarily inexhaustive*, or equivalently, when it is inexhaustive regardless of how the programmer chooses to fill any pattern holes. In particular, such a match expression should still be inexhaustive even in the most permissive of hole-fillings, i.e. when all pattern holes are treated as wildcards. Conversely, if a match expression is inexhaustive with this most permissive hole-filling, then it will also certainly be inexhaustive given any more restrictive hole-filling. As a result of this, we can state that a match expression is necessarily inexhaustive if and only if it is inexhaustive when all pattern holes are filled with wildcards.

As discussed, we use constraints to encode exhaustiveness. Considering the definitions in Fig. 18, a pattern hole emits an unknown constraint ?, while a wildcard emits the truth constraint $\top$. Correspondingly, filling pattern holes with wildcards replaces all instances of ? with $\top$ in the emitted constraint. To that end, Fig. 20 defines a function $\dagger(\xi)$ performing such an operation. We refer to this as *truifying* the constraint.

With all indeterminacy eliminated, we can now define a simpler notion of exhaustiveness known as *constraint validity.* Here, the premise $\dagger(\xi) = \xi$ specifies that ξ no longer contains any instances of the unknown constraint. We call such a ξ a *fully known* constraint.

Definition 4.10 (Validity of Constraints). Suppose that $\dagger(\xi) = \xi$ and $\xi : \tau$. Then $\models \xi$ if and only if for all e such that $\cdot\,; \Delta \vdash e : \tau$ and e `val` we have $e \models \xi$

The following theorem formalizes the above discussion, stating that a constraint is exhaustive if and only if truifying it produces a valid constraint. A proof is provided in our Agda mechanization.

Theorem 4.11. $\top \models^{\dagger}_{?} \xi$ *if and only if* $\models \dagger(\xi)$.

We now make a similar argument regarding redundancy. Recall that a rule r is necessarily redundant with respect to the previously considered rules rs if, considering any possible hole filling, those expressions e which match the pattern in r also necessarily match some pattern in rs. In the worst case then, any expression e which matches the most permissive hole filling of r should still match some pattern in rs, even if we attempt to adversarially fill the pattern holes in rs as to prevent any subterm of e from matching them. That is, we should consider the most permissive hole filling of r but the most restrictive hole filling of rs. In terms of constraints, if r emits ξ_r and rs emits ξ_{rs}, then r is necessarily redundant with repsect to rs when $\xi_r \models \xi_{rs}$. To that end, Fig. 20

$$\boxed{\dot{\top}(\xi_1) = \xi_2}$$

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

$$\boxed{\dot{\bot}(\xi_1) = \xi_2}$$

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

Fig. 20. Truify and Falsify Constraints

also defines a *falsify* function $\dot{\bot}(\xi)$ which replaces every instance of ? in ξ with $\bot$, corresponding to the most restrictive hole filling. The following theorem formalizes our discussion.

THEOREM 4.12. *$\xi_1 \models \xi_2$ if and only if $\dot{\top}(\xi_1) \models \dot{\bot}(\xi_2)$.*

Further, considering dual as negation and entailment as implication, one can prove something akin to the material implication of propositional logic, transforming constraint entailment into constraint validity. In effect, this demonstrates a "fully known" equivalence between redundancy and exhaustiveness.

LEMMA 4.13 (MATERIAL ENTAILMENT). *Suppose ξ_1 and ξ_2 are fully known constraints. Then $\xi_1 \models \xi_2$ if and only if $\models \overline{\xi_1} \vee \xi_2$.*

Combined with Theorem 4.12, we can then state necessary redundancy in terms of validity.

THEOREM 4.14. *$\xi_r \models \xi_{rs}$ if and only if $\models \overline{\dot{\top}(\xi_r)} \vee \dot{\bot}(\xi_{rs})$.*

4.6 Implementation and Decidability

Considering Theorem 4.11 and Theorem 4.14, we have now reduced both exhaustiveness and irredundancy checking to the problem of constraint validity. Thus, to provide actual implementations of these checks, all that remains to provide an algorithm deciding whether a given constraint ξ is valid. The ensuing theorem provides the key ingredient needed to implement such a procedure.

THEOREM 4.15 (EXCLUSIVENESS OF SATISFACTION JUDGMENT). *If $\xi : \tau$ and $\cdot\,; \Delta \vdash e : \tau$ and e $\texttt{val}$ then exactly one of the following holds*

(1) $e \models \xi$
(2) $e \models \overline{\xi}$

We say that a fully known constraint $\xi : \tau$ is *inconsistent* if no value of type τ satisfies ξ. From the above theorem, it immediately follows that a fully known constraint ξ is valid if and only if its dual $\bar{\xi}$ is inconsistent. Thus, to decide validity, it suffices to provide a decision procedure for inconsistency, and fortunately, inconsistency is syntactically quite obvious.

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

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

$$\frac{}{\Xi, \underline{n}, \underline{\cancel{n}}\ \texttt{incon}}\ \text{CINCNotNum} \qquad \frac{\Xi, \xi_1, \xi_2\ \texttt{incon}}{\Xi, \xi_1 \wedge \xi_2\ \texttt{incon}}\ \text{CINCAnd}$$

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

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

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

Fig. 21. Inconsistency of Constraints

To this end, Fig. 21 defines a judgement $\Xi\ \texttt{incon}$ specifying that the conjunction over a set of constraints Ξ is inconsistent. Here, we choose to use a set of constraints rather than a single constraint, as conceptually, a single constraint may really represent a boolean combination of many smaller constraints, and our rules require reasoning simultaneously about all these constituent parts. By abuse of notation, for a single constraint ξ, we write $\xi\ \texttt{incon}$ to denote $\{\xi\}\ \texttt{incon}$ for the singleton set $\{\xi\}$.

Theorem 4.16 states that our judgement behaves as desired. Note that, unlike all other work we have presented thus far, we do not provide an Agda mechanization for results involving the $\Xi\ \texttt{incon}$ judgement. While intuitively straightforward, the algorithms and proofs here require manipulations of finite sets which are not obviously structurally recursive, making them inordinately cumbersome to implement in Agda.

Theorem 4.16. *If ξ is fully known, then $\models \xi$ if and only if $\overline{\xi}\ \texttt{incon}$.*

Let us discuss the $\Xi\ \texttt{incon}$ judgement in more detail. Throughout, we assume all constraints present in Ξ are of the same type. To begin, logical connectives are handled in a straightforward way. Rule CINCAnd specifies that a set including the constraint $\xi_1 \wedge \xi_2$ is inconsistent exactly when it is inconsistent including both ξ_1 and ξ_2 together. Likewise, rule CINCOr specifies that a set including $\xi_1 \vee \xi_2$ is inconsistent only if it is inconsistent both when including only ξ_1 and when including only ξ_2. As we will soon discuss, algorithmically, these judgements allow us to "flatten" any constraints in our inconsistency decision procedure, removing all instances of $\wedge$ and $\vee$.

Outside of logical connectives, we need only consider how inconsistency can arise for each type. For example, a set of constraints on numbers contains only constraints of the form $\underline{n}$ and $\underline{\cancel{n}}$. If $\underline{n}$ and $\underline{\cancel{n}}$ are both included for a single n, this is clearly inconsistent, and likewise if $\underline{n_1}$ and $\underline{n_2}$ are both included when $n_1 \neq n_2$. CINCNum and CINCNotNum handles these cases. For binary sums, inconsistency only occurs if either different constructors are specified as in CINCInj, or if the same constructors are specified, but the constraints contained within said constructors are inductively inconsistent as in CINCInl and CINCInr. Binary products proceed similarly, inductively considering inconsistency on each coordinate as in CINCPairL and CINCPairR.

Considering each of the rules for Ξ `incon`, all premises only involve constraints that are proper components of the constraints in the corresponding conclusion. Consequently, Ξ `incon` is decidable by simply inverting each applicable rule until there are no such rules remaining. Eventually, we will either hit a case requiring no induction in its premise - one of CINCFalsity, CINCNum, CINCNotNum, and CINCInj - implying inconsistency. Otherwise, we reach the point where no rules are applicable, implying consistency.

A full decision procedure $incon(\Xi)$ is shown below. Following our actual OCaml implementation, we choose to represent Ξ as an ordered list of constraints, and thus proceed from the head onward.

(1) If Ξ is empty, return `false`. Otherwise, let $\Xi = \xi, \Xi'$.
(2) If $\xi = \bot$, return `true`.
(3) If $\xi = \top$, return $incon(\Xi')$.
(4) If $\xi = \xi_1 \wedge \xi_2$, return $incon(\xi_1, \xi_2, \Xi')$.
(5) If $\xi = \xi_1 \vee \xi_2$, return $incon(\xi_1, \Xi') \wedge incon(\xi_2, \Xi')$.
(6) Partition Ξ, letting Ξ_{log} contain all constraints of the form $\xi_1 \vee \xi_2$ or $\xi_1 \wedge \xi_2$ and letting Ξ_{other} contains all other constraints. If Ξ_{log} is non-empty, return $incon(\Xi_{log} {+}{+}\ \Xi_{other})$.
(7) If $\xi = \mathtt{inl}(\xi')$ or $\xi = \mathtt{inr}(\xi')$, return `true` if Ξ' contains any mismatching injection constructor. Otherwise, strip the outermost injection constructor from each element of Ξ' to produce a list Ξ'_{str}, then return $incon(\xi', \Xi'_{str})$.
(8) If $\xi = (\xi_1, \xi_2,)$, let Ξ'_1 be the result of mapping $(\xi_1, \xi_2) \mapsto \xi_1$ onto Ξ', and similarly, let Ξ'_2 be the result of mapping $(\xi_1, \xi_2) \mapsto \xi_2$ onto Ξ'. Return $incon(\xi_1, \Xi'_1) \vee incon(\xi_2, \Xi'_2)$.
(9) If $\xi = \underline{n}$ or $\xi = \cancel{\underline{n}}$, then partition Ξ into two lists, letting Ξ_{yes} contain all constraints $\underline{m}$ and Ξ_{no} contain all constraints $\cancel{\underline{m}}$. If Ξ_{yes} contains $\underline{m}$ and Ξ_{no} contains $\cancel{\underline{m}}$ for the same m, return `true`. If Ξ_{yes} contains $\underline{m_1}$ and $\underline{m_2}$ with $m_1 \neq m_2$, return `true`. Otherwise, return `false`.

Note that line (6) moves all constraints built up using $\wedge$ and $\vee$ to the front of the list, causing them to be expanded on lines (4) and (5) in the recursive call. Resultingly, after line (6), we can assume all constraints are "flattened" to an outermost non-$\wedge$ non-$\vee$ constructor. Correspondingly, the following lines each handle a different possibility for these outermost constructors, making use of the fact that all constraints in Ξ are of the same type. On line (7), all constraints must be of the form $\mathtt{inl}(\xi)$ or $\mathtt{inr}(\xi)$, and we proceed according to CINCInj, CINCInl, and CINCInr. Similarly, on line (8), we handle the case where all constraints are of the form $\xi_1 \wedge \xi_2$, checking inconsistency on each projection as in CINCPairL and CINCPairR. Finally, on line (9) we reach our base case, only having to handle constraints on our base number type, which is easily decidable. If Peanut is extended to include other base types, similar cases would have to be added following line (9).

With such a decision procedure, using the results of Theorem 4.11, Theorem 4.14, and Theorem 4.16, we now have decision procedures for exhaustiveness and redundancy. Note that the latter here theorem uses the fact that a constraint is equivalent to the dual of its dual.

Theorem 4.17. *$\top \models^{\dagger}_{?} \xi$ if and only if $incon(\overline{\dot{\top}(\xi)}) = true$.*

Theorem 4.18. *$\xi_r \models \xi_{rs}$ if and only if $incon(\dot{\top}(\xi_r) \wedge \overline{\dot{\bot}(\xi_{rs})}) = true$.*

4.6.1 Witness Generation. We have now shown how to decide both exhaustiveness and redundancy in the presence of holes. As discussed in Sec. 2 and displayed in Fig. 4 and Fig. 5, such checks enable us to provide helpful feedback, statically eliminating large classes of bugs. However, note that simply reporting inexhaustivity with no further guidance may be quite frustrating for the user - we only opaquely state that at least one expression is unhandled, without ever giving a concrete example of such an expression. Particularly in the presence of holes, it may not be immediately clear why the inexhaustiveness arises.

To remedy such opaqueness, if our necessarily inexhaustive rules emit a constraint ξ, we must construct a value e such that $e \not\models_{?}^{\dagger} \xi$. Equivalently, considering Theorem 4.11, we must construct a value e such that $e \not\models \dot{\top}(\xi)$, which, by Theorem 4.15, is in turn equivalent to having $e \models \overline{\dot{\top}(\xi)}$. That is, constructing a witness to inexhaustiveness can be reduced to the problem of constructing a witness for a fully known constraint.

Fortunately, constructing a witness only requires a slight modification of the *incon*(Ξ) procedure we defined earlier. Indeed, the Ξ `incon` judgement exactly pins down when it is impossible to construct a witness to a set of constraints Ξ, so it unsurprising that these algorithms are related. Formally, we give a procedure *witness*(Ξ) where Ξ is a set of constraints, either returning *None* if Ξ is inconsistent, or otherwise returning *Some*(e) for a value e with $e \models \xi$ for all $\xi \in \Xi$.

(1) If Ξ is empty, return $\mathsf{Some}(e)$ for any arbitrary expression e. Otherwise, let $\Xi = \xi, \Xi'$.
(2) If $\xi = \bot$, return None.
(3) If $\xi = \top$, return $witness(\Xi')$.
(4) If $\xi = \xi_1 \wedge \xi_2$, return $witness(\xi_1, \xi_2, \Xi')$.
(5) If $\xi = \xi_1 \vee \xi_2$, then compute $w_1 = witness(\xi_1, \Xi')$ and $w_2 = witness(\xi_2, \Xi')$. If both w_1 and w_2 are None, then return None. Otherwise, arbitrarily return whichever of w_1 and w_2 is not None.
(6) Partition Ξ, letting Ξ_{log} contain all constraints of the form $\xi_1 \vee \xi_2$ or $\xi_1 \wedge \xi_2$ and letting Ξ_{other} contains all other constraints. If Ξ_{log} is non-empty, return $witness(\Xi_{log}{+}{+}\ \Xi_{other})$.
(7) If $\xi = \mathtt{inl}(\xi')$ or $\xi = \mathtt{inr}(\xi')$, return None if Ξ' contains any mismatching injection constructor. Otherwise, strip the outermost injection constructor from each element of Ξ' to produce a list Ξ'_{str}. Compute $w = witness(\xi', \Xi'_{str})$. If w is None then return None. Otherwise, if w is $\mathsf{Some}(e)$, return $\mathsf{Some}(\mathtt{inl}(e))$ if $\xi = \mathtt{inl}(\xi')$ or $\mathsf{Some}(\mathtt{inr}(e))$ if $\xi = \mathtt{inr}(\xi')$.
(8) If $\xi = (\xi_1, \xi_2,)$, let Ξ'_1 be the result of mapping $(\xi_1, \xi_2) \mapsto \xi_1$ onto Ξ', and similarly, let Ξ'_2 be the result of mapping $(\xi_1, \xi_2) \mapsto \xi_2$ onto Ξ'. Compute $w_1 = incon(\xi_1, \Xi'_1)$ and $w_2 = incon(\xi_2, \Xi'_2)$. If w_1 is $\mathsf{Some}(e_1)$ and w_2 is $\mathsf{Some}(e_2)$, then return $\mathsf{Some}((e_1, e_2))$. Otherwise, return None.
(9) If $\xi = \underline{n}$ or $\xi = \underline{\cancel{n}}$, then partition Ξ into two lists, letting Ξ_{yes} contain all constraints $\underline{m}$ and Ξ_{no} contain all constraints $\underline{\cancel{m}}$. If Ξ_{yes} contains $\underline{m}$ and Ξ_{no} contains $\underline{\cancel{m}}$ for the same m, return None. If Ξ_{yes} contains $\underline{m_1}$ and $\underline{m_2}$ with $m_1 \neq m_2$, return None. Otherwise, if Ξ_{yes} is non-empty return $\mathsf{Some}(m)$ for any $\underline{m} \in \Xi_{yes}$, else return $\mathsf{Some}(m)$ for any m with $\underline{\cancel{m}} \notin \Xi_{no}$.

The above almost exactly follows the decision procedure *incon*(Ξ), except that we actually construct a witness for the case of the base type on line (9), and at each recursive call, we wrap the result with an appropriate constructor to propagate such base witnesses upward, reporting inconsistency whenever this fails. In fact, then, our *witness*(Ξ) procedure supersedes the need for the *incon*(Ξ) procedure, with $witness(\Xi) = \mathsf{None}$ exactly when $incon(\Xi) = true$.

On a final note, we can use a similar decision procedure to generate witnesses to irredundancy. That is, if $\xi_r \not\models \xi_{rs}$, then in light of Theorem 4.14, $witness(\dot{\top}(\xi_r) \wedge \overline{\dot{\bot}(\xi_{rs})})$ will be some value e such that $e \models_{?}^{\dagger} \xi_r$ and $e \not\models \xi_{rs}$. As irredundancy is not an error, this is unlikely to be useful to the user, but it is interesting to note nonetheless.

5 AGDA MECHANIZATION

Thus far, we have presented a system for pattern matching with typed holes, and we have discussed the reasoning and motivation for all of its aspects. Hopefully, the reader should find our system fairly intuitive, and should be reasonably convinced of its correctness. Indeed, we have provided various metatheoretical and semantic theorems which indicate that all of our judgements behave as expected, and that they enforce the constraints we desire.

However, even with such theorems stated and proved on-paper, we must consider that humans have quite a propensity for error. (After all, much of our work here focuses on analyses which seek to prevent common human errors!) We have seen that pattern matching with holes in both expressions and patterns can be very intricate, requiring reasoning about a three-valued logic of must, must not, and indeterminate judgements. Likewise, the proofs of our theorems involve large amounts of casework, having to consider numerous combinations of expressions, patterns, and constraints. Anecdotally, our initial work required nearly 150 pages of tediously typed out proofs. Thus, while on-paper proofs *supposedly* show that our system is correct, it is highly unlikely that such voluminous work is entirely error-free.

To address this, and to provide a stronger guarantee of our system's correctness, we then must utilize something much more reliable than human intuition. Namely, we turn to the Agda proof-assistant [Norell 2007], providing a computer-checked mechanization of nearly all theorems stated here. The only results we do not mechanize are those related to the Ξ `incon` procedure discussed in Sec. 4.6. As previously mentioned, such proofs involve algorithms which use finite sets sets in a non-structurally recursive way, making them cumbersome to implement in Agda.

Throughout the rest of this section, we present an in depth discussion of our mechanization. In Sec. 5.1, we review the general background for dependently type theorem provers. Next, in Sec. 5.2, we discuss how we encode our judgements, giving particular attentions to choices regarding binders, contexts, and other aspects that are often glossed over on paper. Finally, in Sec. 5.3, we reflect on the benefit of our mechanization, enumerating the various subtle errors it revealed.

Our mechanization is available for viewing at https://github.com/hazelgrove/patterns-agda, and it is well-commented with documentation explaining the contents of each file. All deviations from the paper are explicitly laid out and motivated in the README of our repository. The development of this nearly 15,000 LOC mechanization represents the bulk of the first author's contribution to the Peanut project.

5.1 Theorem Proving with Dependent Types

To begin, we review the required background on the Agda proof-assistant, discussing how its type system enables users to mechanically verify mathematical propositions. All of the material here is standard and should be familiar to the academic programming language researcher.

Briefly, Agda and other dependently-typed proof assistants are designed around the following key observation: a type T may be regarded as representing a proposition "T is inhabited", and constructing a term $t : T$ correspondingly provides a proof of this proposition. If our type system is sufficiently rich, we also have the converse: for any proposition P, we may construct a type T_P such that "T_P is inhabited" is logically equivalent to P, and the terms of T_P are then exactly the proofs of P. With such a type system, there is a one-to-one correspondence between propositions and types and between proofs and terms, commonly known as the *Curry-Howard Correspondence* [Howard 1980].

Considering Agda in particular, it is a typed functional programming language based on a variant of Martin-Löf type theory [Martin-Löf 1984] known as the Unified Theory of Dependent Types [Luo 1994; Norell 2007]. With dependent types, the division between types and values becomes blurry - types may be parameterized by arbitrary values, and types may be given as arguments and results of functions. Such a formulation is quite powerful, allowing Agda's type system to correspond with a variant of intuitionistic logic.

Let us show a small taste of the details of this correspondence. For a simple example, take binary product types $A \times B$, whose terms $(a, b) : A \times B$ require $a : A$ and $b : B$. In our correspondence, such a term (a, b) then consists of a proof a for the proposition given by A and a proof b for the proposition given by B. Thus, $A \times B$ corresponds to the logical "and" of propositions $A \wedge B$. Indeed, Fig. 22 shows that the type checking rules for $A \times B$ syntactically mirror the intuitionistic natural deduction rules for $A \wedge B$.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \text{AndIntro} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \text{AndElimL} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \text{AndElimR}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}\ \text{TAPair} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathtt{fst}(p) : A}\ \text{TAFst} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{snd}(p) : B}\ \text{TAFst}$$

Fig. 22. Correspondence between $A \times B$ and $A \wedge B$

Analogously, binary sums $A + B$ correspond to the logical "or" of propositions $A \vee B$. For a more interesting case, consider propositions of the form $\forall x.B$, stating that for any x, the proposition B holds, where B may depend on x. In our correspondence, we can consider B to be a type parameterized by a value x, say $x : A$ for some other type A. That is, B is a *dependent type*. To prove $\forall x.B$, we then must be able to produce a proof $f(a) : [a/x]B$ for any $a : A$. That is, we should have a function f which takes inputs $a : A$ and produces outputs $f(a) : [a/x]B$. We refer to f as a *dependent function*, noting that its return type may depend on the particular value of its input. Formally, we write $f : \Pi_{x:A}B$ or $f : (x : A) \to B$. Again, Fig. 23 shows a syntactic correspondence between type checking for dependent functions and natural deduction for universal quantification.

$$\frac{\Gamma \vdash B \qquad (x \text{ not free in } \Gamma)}{\Gamma \vdash \forall x.B}\ \text{UniversalIntro} \qquad \frac{\Gamma \vdash \forall x.B}{\Gamma \vdash [a/x]B}\ \text{UniversalElim}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.b : \Pi_{x:A}B}\ \text{TALam} \qquad \frac{\Gamma \vdash f : \Pi_{x:A}B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : [a/x]B}\ \text{TAAp}$$

Fig. 23. Correspondence between $\Pi_{x:A}B$ and $\forall x.B$

With the general idea of this correspondence now clear, let us present another feature of Agda we use pervasively in our development: *inductive type families*. Briefly, an inductive type T is specified by a finite list of functions with return type T, called *constructors*, and the terms of T consist of any syntactic form built up freely by repeated application of these constructors. In particular, the constructors themselves may inductively take arguments from T, allowing terms of T to be built up from other "smaller" terms of T.

For example, recall the Peano construction of the natural numbers, where a natural number is either 0 or inductively the successor of another natural number $suc(n)$. In Agda, we can represent this as an inductive type `Nat` as defined in Fig. 24. Here, there are two constructors: `zero` which takes no arguments, and `suc` which inductively takes a single argument of type Nat. The terms of Nat are then of the form `zero`, `suc zero`, `suc (suc zero)`, and so forth. Formally, the constructors yield the introduction rules shown in Fig. 24, and while not enumerated, appropriate elimination rules are also added, allowing functions to be defined by handling the case of each different constructor.

Inductive type families behave similarly, but rather than defining a single type T, we define a family of types T parameterized by values or other types. With such inductive families, we can easily encode almost any mathematical judgement, letting T represent the judgement and the

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

four : Nat
four = suc (suc (suc zero))
```

(a) Inductive Type Definition

$$\frac{}{\Gamma \vdash zero : Nat}\ \text{Zero}$$

$$\frac{\Gamma \vdash n : Nat}{\Gamma \vdash suc\ n : Nat}\ \text{Suc}$$

(b) Introduction Rules

Fig. 24. Natural numbers in Agda

constructors of T represent the defining inference rules. For example, we can define a type $x < y$ parameterized by values $x\ y : Nat$, specifying that x is less than y. For any such x, y, one can prove $x < y$ either by the fact that $0 < n$ for any n, or by inductively using the fact that $m < n$ to get that $suc\ m < suc\ n$. Correspondingly, Fig. 25 defines an inductive type family $\_{<}\_ : Nat \longrightarrow Nat \longrightarrow Set$ with two constructors. The relevant inference rules are also shown, and intuitively, a term of type $x < y$ will be a derivation using these rules . Here, the syntax $\_{<}\_$ is a *mixfix operator* where each _ denotes an argument position and $x < y$ desugars to $\_{<}\_\ x\ y$. Additionally, arguments written with curly braces in the form $\{x : Nat\}$ are implicit, with Agda inferring their value at each call site.

```
data _<_ : Nat → Nat → Set where
  z<s : {n : Nat} →
        zero < suc n

  s<s : {m n : Nat} →
        m < n →
        suc m < suc n
```

(a) Inductive Type Definition

$$\frac{\Gamma \vdash n : Nat}{\Gamma \vdash z{<}s : zero < suc\ n}\ \text{Z<S}$$

$$\frac{\begin{array}{c}\Gamma \vdash m : Nat \\ \Gamma \vdash n : Nat \qquad \Gamma \vdash p : m < n\end{array}}{\Gamma \vdash s{<}s\ p : suc\ m < suc\ n}\ \text{S<S}$$

(b) Introduction Rules

Fig. 25. Ordering of natural numbers

Considering the discussed Curry-Howard correspondence, we can prove a theorem such as $\forall n \in \mathbb{N}, n < suc(n)$ by constructing a term of type $\Pi_{n:Nat}\ (n < suc\ n)$ as displayed in Fig. 26. Note that Agda uses a more lightweight syntax $(n : Nat) \longrightarrow n < suc\ n$ to denote the dependent function type. The corresponding term / proof / function is defined by pattern matching, handling the case of each constructor and using recursion analogously to an induction on derivations. Exhaustiveness checking guarantees that all cases are handled.

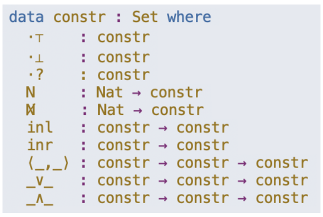
```
two<three : suc (suc zero) < suc (suc (suc zero))
two<three = s<s (s<s z<s)

n<sucn : (n : Nat) → n < suc n
n<sucn zero = z<s
n<sucn (suc n) = s<s (n<sucn n)
```

Fig. 26. Example proofs using the ordering relation

5.2 Mechanization of Peanut

With the relevant background material discussed, we are now prepared to present the specifics of how we encode Peanut into Agda. Beginning with syntax, we can straightforwardly translate our grammars into inductive data types. We utilize mixfix operators and Agda's unicode support in order to closely mirror the on-paper notation. For example, Fig. 27 directly translates the syntax for constraints given in Fig. 13 into an inductive data type.



```
data constr : Set where
  ·⊤    : constr
  ·⊥    : constr
  ·?    : constr
  N     : Nat → constr
  N̸     : Nat → constr
  inl   : constr → constr
  inr   : constr → constr
  ⟨_,_⟩ : constr → constr → constr
  _∨_   : constr → constr → constr
  _∧_   : constr → constr → constr
```

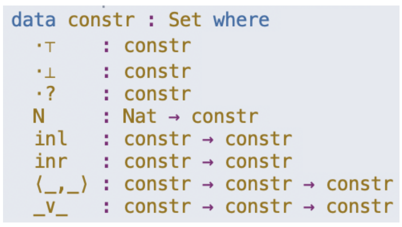
Fig. 27. Constraint Language

However, for usability reasons, our actual mechanization deviates slightly from the above definition of constraints. Explicitly, we instead define two separate constraint languages as shown in Fig. 28. This formalizes the fact that we work with constraints in two distinct stages: first, the constraints directly emitted by lists of rules, then, for use in redundancy and exhaustiveness checking, the "fully-known" constraints which are in the image of the truify and falsify functions and their duals. The truify and falsify functions are then maps between these constraint languages. If we did not have such separation in our mechanization, we would continually have to pass around and reason about proofs that a constraint is fully know, and, morally, such changes are irrelevant.

The rest of the mechanization proceeds straightforwardly, with only a few other minor deviations from the paper which we will discuss shortly. Each judgement is encoded as an inductive type family, with its rules given by dependently typed constructors there of. Mimicking the induction on derivations used for on-paper proofs about such judgements, all of our mechanized proofs proceed by pattern matching, handling the case of each constructor.

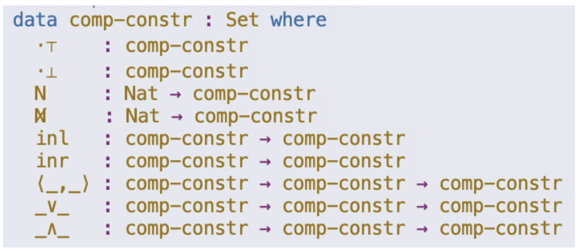
However, it is worth noting that many details which are considered irrelevant on paper and frequently glossed over must be explicitly handled in Agda. For example, consider Fig. 29, which shows the Agda data type for our typing judgement $\Gamma ; \Delta \vdash e : \tau$ and the constructor corresponding to the rule TLam. For clarity, all other constructors of the data type are not displayed.

The first two lines state that we are defining an inductive type family of the form $\Gamma, \Delta, \Delta_p \vdash e :: \tau$, where Γ is a typing context, Δ is an expression hole context, Δ_p is a pattern hole context, e is an



```
data constr : Set where
  ·⊤    : constr
  ·⊥    : constr
  ·?    : constr
  N     : Nat → constr
  inl   : constr → constr
  inr   : constr → constr
  ⟨_,_⟩ : constr → constr → constr
  _∨_   : constr → constr → constr
```

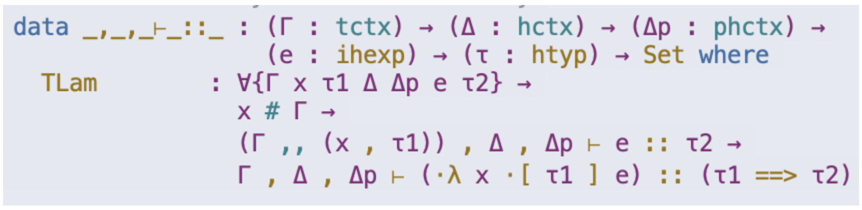
(a) Incomplete Constraints



```
data comp-constr : Set where
  ·⊤    : comp-constr
  ·⊥    : comp-constr
  N     : Nat → comp-constr
  N̸     : Nat → comp-constr
  inl   : comp-constr → comp-constr
  inr   : comp-constr → comp-constr
  ⟨_,_⟩ : comp-constr → comp-constr → comp-constr
  _∨_   : comp-constr → comp-constr → comp-constr
  _∧_   : comp-constr → comp-constr → comp-constr
```

(b) Complete "Fully Known" Constraints

Fig. 28. Separated Constraint Languages



```
data _,_,_⊢_::_ : (Γ : tctx) → (Δ : hctx) → (Δp : phctx) →
                  (e : ihexp) → (τ : htyp) → Set where
  TLam        : ∀{Γ x τ1 Δ Δp e τ2} →
                 x # Γ →
                 (Γ ,, (x , τ1)) , Δ , Δp ⊢ e :: τ2 →
                 Γ , Δ , Δp ⊢ (·λ x ·[ τ1 ] e) :: (τ1 ==> τ2)
```

Fig. 29. Lambda Type Assignment

expression, and τ is a type. Here, we see another small deviation from the paper, namely that our mechanization uses separate hole contexts for expression holes and pattern holes. Such a separation is irrelevant to the theory, but it is implemented with future developments in mind, e.g. if we later implement a fill-and-resume operation à la Hazelnut Live [Omar et al. 2019], requiring us to store different sorts of hole closures for pattern and expression holes.

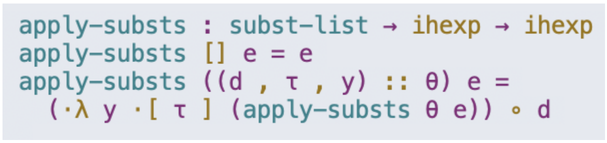
In our mechanization, we implement contexts as metafunctions from variable names to optional contents, and for simplicity, we assume all variables are natural numbers. Correspondingly, the typing context Γ above is a map Nat $\longrightarrow$ Maybe htyp. Note that while we do not explicitly require it, in practice, these mappings are always finitely supported. Additionally, in order to simplify working with such contexts, we also postulate function extensionality, which is known to be independent of Agda's axioms [Awodey et al. 2012]. We assume no other postulates in our development

Continuing, the next line defines a constructor TLam. The type of this constructor begins with $\forall\{\Gamma\ \Delta\ \Delta_p\ e\ \tau\}$, where the syntax here specifies a number of implicit arguments which we instruct Agda to infer the type of. The following lines then correspond directly with the inference rules for TLam, except notably, the inclusion of the premise $x\#\Gamma$. Such a premise specifies that x does not occur in Γ, which in terms of our implementation of contexts, specifies that $\Gamma\ x == \text{None}$. Why must we include this premise in our mechanization, while we do not on paper?

The core of the issue here is that on-paper presentations frequently ignore issues related to variable shadowing and conflict between free and bound variables. Instead, we implicitly assume that whenever required, α-conversion can be performed as to rename any variables in the case of a conflict. However, for our mechanization, we must make an explicit choice of how to handle such conflicts. There are number of standard options. For example, De Bruijn indices replace variables with a number indicating how many lambda abstraction there are between the variable use and where it bound, modifying these indices whenever required, and thereby removing any shadowing. Other approaches such as Abstract Binding Trees and Higher Order Abstract Syntax annotate the base syntax tree, explicitly tying binders to their scope and automatically performing rewriting when needed [Harper 2012]. However, in our experience, all of these approaches yield a pervasive overhead, quickly obfuscating the actual points of interest in our mechanization. As little of our theory depends on any specifics about binders, we choose to take a more lightweight approach known as Barendregt's convention [Barendregt 1985; Urban et al. 2007].

Briefly, Barendregt's convention simply assumes that all binders and free variables in an expressions are given globally unique names from the start. In our development, we then define judgements specifying such uniqueness, and wherever required, insert these as premises to our theorems and definitions. Returning to Fig. 29, the premise $x\#\Gamma$ is an example of this, specifying that a lambda expressions can ony be well-typed if its binder is unique with respect to the free-variables recorded in the typing context. Again, we note that all these premises are benign and easily discharged by α-variation.

On a final note, our choice to follow Barendregt's convention did introduce a subtle issue in exactly one place in our mechanization: the rule ITSuccMatch. In the case of a successful a pattern match $e \rhd p \dashv\!\mid \theta$, this rule applies the substitution θ to the corresponding branch expression for the rule. In our proof of Theorem 4.9, following Barendreg'ts convention, we initially assume that the expression under consideration has globally unique binder names. Thus, applying the emitted subtitutions θ should ideally present no issues related to variable conflicts. However, in one specific case, namely, when the match succeeds due to the MNotIntroPair rule, we may emit a substitution such as $[\mathtt{fst}(e)/x,\ \mathtt{snd}(e)/y]$. When these substitutions are applied one by one, the latter substitution $[\mathtt{fst}(e)/x]$ now replaces x in a term that already has a copy of e, and thus has conflicting binders. Resultingly, the final expression may not be well-typed due to disjointness premises such as the one discussed in Fig. 29.



```
apply-substs : subst-list → ihexp → ihexp
apply-substs [] e = e
apply-substs ((d , τ , y) :: θ) e =
  (·λ y ·[ τ ] (apply-substs θ e)) ∘ d
```

Fig. 30. Expanding substitutions into lambdas

Luckily, we can resolve this issue with only a slight modification. When we apply $[\theta]e_r$ for our branch expression e_r, rather than actually performing the substitution, we wrap e_r with appropriate lambda abstractions and applications as shown in Fig. 30. (The syntax $e_1 \circ e_2$ denotes function application in our mechanization.) Effectively, this expansion defers the substitutions until later evaluation steps, and as a result, we can indeed re-apply Barendregt's convention between each substitution, resolving our issue. Unfortunately, however, as we only support half-annotated lambdas, this also requires θ to record typing information, which we ultimately achieve by adding a type annotation to the $e \rhd p \dashv\!\mid \theta$ judgement. Such annotations are benign, and are only required to handle this specific case.

5.3 Issues Revealed by Mechanization

Finally, we reflect on the benefit that our mechanization provided, enumerating some of the various small issues that it revealed in the initial draft of our system. All of the issues here were revealed during mechanization, and were not discovered in the on-paper proofs of the same theorems.

As mechanization leaves no stone unturned, it is quite adept at revealing small subtle errors in our inference rules and judgements which are easy to overlook by a human. For example, consider the rules IFst and ISnd. Initially, these rules stated that for any expression e with e `indet`, we always have both $\mathtt{fst}(e)$ `indet` and $\mathtt{snd}(e)$ `indet` - a seemingly intuitive statement. However, when mechanizing the fact that final expressions do not have applicable evaluation steps in Theorem 4.1, it quickly became apparent that this conflicts with the stepping rules ITFstPair and ITSndPair, which allow taking projections of indeterminate pairs. Correspondingly, we updated IFst and ISnd to add the premise $e \neq (e_1, e_2)$ as in their current version. From the mechanization of Theorem 4.1 alone, we also discovered that we had forgotten to allow evaluation steps under projections, requiring us to add the rules ITFst and ITSnd. Further, we discovered that the rules ITApArg, ITAp, and ITPairR incorrectly specified various e `val` premises where there is now e `final`, again breaking progress.

Mechanizing the other half of type-safety, Theorem 4.9, also proved useful. Initially, our judgement for pattern typing was syntactically written $p : \tau[\xi] \dashv\!\mid \Gamma; \Delta$, rather than the current $\Delta \vdash p : \tau[\xi] \dashv\!\mid \Gamma$ defined in Fig. 18. While the rules themselves were the same, the syntax here indicated that we conceptually considered both Γ and Δ to be outputs, with Γ recording variables bound in

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

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

Fig. 31. Old and New Single Rule Typing

the pattern, and Δ recording hole names. Correspondingly, our rule typing judgement was initially stated as BadTRule shown at the top of Fig. 31, extending both Γ and Δ to $\Gamma \uplus \Gamma_p$ and $\Delta \uplus \Delta_p$ when type checking the branch expression e. However, this is subtly incorrect, as it allows e to include hole names which are also bound in p but with conflicting typing information, and as our mechanization revealed, this results in the ITSuccMatch rule stepping to an ill-typed expression. We corrected the rule correspondingly, as repeated in GoodTRule in Fig. 31, and updated the pattern matching syntax to conceptually reflect this change.

For one final error, mechanization also revealed the need for the ξ `possible` judgement discussed in Sec. 4.3. Without it, we would judge that an indeterminate expression e indeterminately satisfies the $\perp$ constraint, despite the fact that no hole-filling would ever result in satisfaction.

While often fixable, our discussion makes it clear that subtle issues can still permeate even with on-paper proofs of correctness, and we hope the reader is motivated to take up the mechanization of their own work. Anecdotally, we also found the mechanization extremely useful as we incrementally defined and extended our system. When changes are made, a robust mechanization makes it easy to determine which theorems must be updated, and if the theorems cannot be updated, it is often obvious which parts of a definition are at fault. For one example, we extended our system with a unit type. While conceptually a very simple change, without mechanization, it would have required scanning through and updating all of our 150 pages of on-paper proofs. With the mechanization, however, the compiler immediately identified all proofs which were non-exhaustive, and with Agda's proof search capabilities, almost all the missing cases could be filled in automatically.

6 CONCLUSION

As the authors can attest, programming is an oft painful process, full of failures, subtle flaws, breaking edits, and frequently fragile tools. Yet, scattered between these frustrating moments are periods of utter joy, when the stars align, code compiles, and programming tools help the despairing developer with clarifying insights, guiding them to their goal.

In this paper, we seek to make these joyful moments all the more frequent. We continue the line of research of [Omar et al. 2017a,b], extending the concept of typed-holes to enable pattern matching while allowing all intermediate edit states to maintain full static and dynamic meaning. By implementing our work into Hazel, we also inch closer to the development of the first fully-featured functional programming language which completely eliminates the *gap problem*. As pattern matching is increasingly a central feature of modern programming languages, we believe the contributions of this paper represent significant progress towards this goal.

In all, we hope for a bright future, with robust support for typed-holes as commonplace as any other editor service, and with meaningless editor states and gaps in editor services entirely abolished. We hope too, that others share this ambition, continuing to bridge the gap between the beautiful purity of type-theoretic semantic foundations and the utmost messiness of real-world program editors.

ACKNOWLEDGMENTS

The completion of this honors thesis would not have been possible without my advisor, Professor Cyrus Omar, and I thank him for his continual support, encouragement, instruction, and utmost kindness. Additionally, I would like to thank Yongwei Yuan for carrying out much of the work towards developing the Peanut calculus, and for his frequent discussions with me throughout the process of mechanization.

REFERENCES

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.

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

Brandt Bucher and Guido van Rossum. 2020. *Structural Pattern Matching: Specification*. PEP 634. https://peps.python.org/pep-0634/

Matteo Cimini and Jeremy G. Siek. 2016. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 443–455. https://doi.org/10.1145/2837614.2837632

Evan Czaplicki. 2018. An Introduction to Elm. (2018). https://guide.elm-lang.org/. Retrieved Mar. 15, 2022.

Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 411–422. https://doi.org/10.1145/2462156.2462161

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

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

William A Howard. 1980. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), 479–490.

Gérard P. Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (1997), 549–554. http://journals.cambridge.org/action/displayAbstract?aid=44121

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

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2021. *The OCaml system release 4.13: Documentation and user's manual*. Intern report. Inria. 1–876 pages. https://hal.inria.fr/hal-00930213

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

Zhaohui Luo. 1994. *Computation and reasoning - a type theory for computer science*. International series of monographs on computer science, Vol. 11. Oxford University Press.

Simon Marlow. 2010. Haskell 2010 language report. (2010).

Per Martin-Löf. 1984. *Intuitionistic type theory*. Studies in proof theory, Vol. 1. Bibliopolis.

Sergei Murzin, Michael Park, David Sankel, Dan Sarginson, and Bjarne Stroustrup. 2019. Pattern Matching.

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.

Ulf Norell. 2013. Interactive programming with dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 1–2. https://doi.org/10.1145/2500365.2500610

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. https://doi.org/10.1145/3009837.3009900

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 *2nd Summit on Advances in Programming Languages, SNAPL*

2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71), Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:12. 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.

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

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