Pattern Matching with Typed Holes

YONGWEI YUAN*, University of Michigan

1 INTRODUCTION

Most contemporary programming environments either provide meaningful feedback only for complete programs, or fall back to limited heuristic approaches when the program is incomplete. Recent work on modeling incomplete programs as typed expressions with *holes* has focused on tackling this problem. Omar et al. [2017] describes a static semantics for incomplete functional programs. Based on that, Omar et al. [2019] develops a dynamic semantics to evaluate such incomplete programs. These foundational treatments are being implemented and extended in the Hazel programming environment (hazel.org).

$$\texttt{match}((\texttt{inl}_{\texttt{num}}(\text{⦶}^{u}), 2))\{ \\ \mid (\texttt{inr}(x), \_) \Rightarrow x \\ \mid (\_, x) \Rightarrow x \\ \} \quad (1)$$

$$\texttt{match}((\texttt{inl}_{\texttt{num}}(1), 2))\{ \\ \mid (\texttt{inl}(x), \_) \Rightarrow x \\ \mid (\_, \text{⦶}^{w}) \Rightarrow \text{⦶}^{u} \\ \} \quad (2)$$

$$\texttt{match}((\texttt{inl}_{\texttt{num}}(1), 2))\{ \\ \mid (\texttt{inl}(x), \_) \Rightarrow x \\ \mid (\texttt{inl}(\text{⦶}^{w_1}), \text{⦶}^{w_2}) \Rightarrow \text{⦶}^{u} \\ \} \quad (3)$$

Pattern matching is a cornerstone of functional programming languages in the ML family. However, Omar et al. [2019] only supports simple case analysis on binary sum types and does not support nested patterns nor pattern holes. This paper addresses this problem, focusing on adding full ML-style pattern matching with support for pattern holes to the Hazelnut core calculus and implementing this system at full scale into Hazel. Consider the examples below, which contain expression and pattern holes, denoted ⦶^{u} (where each hole has a unique name, u).

For the match expression shown in Eq. 1, the scrutinee contains a hole. By considering the match two rules in order, we observe that no matter how the hole u is filled, $(\texttt{inl}_{\texttt{num}}(\text{⦶}^{u}), 2)$ doesn't match $(\texttt{inr}(x), \_)$, and always matches $(\_, x)$. And thus the value of the match expression is 2, despite the fact that the program is incomplete. If, however, we replaced the pattern in the first rule with $(\texttt{inl}(3), \_)$, it would be impossible to decide whether the pattern matches or not, and so the match expression would be *indeterminate*, i.e. awaiting hole filling.

One of the benefits of pattern matching is that it allows for static reasoning about exhaustiveness and redundancy. However, reasoning about these matters is subtle and complicated in the presence of holes. For example, consider the problem of checking if the rules in Eq. 2 cover all the possibilities, i.e. *exhaustiveness*, or if the second rule in Eq. 3 can never be reached no matter how the holes are filled, i.e. *redundancy*.

This abstract focuses on the formalism of full-fledged pattern matching with typed holes so that the programming environment could give this sort of feedback even when considering incomplete match expressions.

2 STATIC SEMANTICS

Fig. 1 defines part of the internal language. Different from the examples in Sec. 1, the rules in a match expression have a pointer to indicate which rule is being considered. The pointer would move

*Research advisor: Cyrus Omar; ACM student member number: 9899292; Category: undergraduate

Author's address: Yongwei Yuan, slark@umich.edu, University of Michigan.

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

TMatchZPre

$$\frac{\Gamma\,;\Delta \vdash e : \tau_1 \qquad \Gamma\,;\Delta \vdash [\bot]r \mid rs : \tau_1[\xi] \Rightarrow \tau_2 \qquad \boxed{\top \models^{\dagger}_{?} \xi}}{\Gamma\,;\Delta \vdash \texttt{match}(e)\{\cdot \mid r \mid rs\} : \tau_2}$$

TMatchNZPre

$$\frac{\Gamma\,;\Delta \vdash e : \tau_1 \quad e\ \texttt{final} \quad \Gamma\,;\Delta \vdash [\bot]rs_{pre} : \tau_1[\xi_{pre}] \Rightarrow \tau_2 \qquad \Gamma\,;\Delta \vdash [\xi_{pre}]r \mid rs_{post} : \tau_1[\xi_{rest}] \Rightarrow \tau_2 \quad \boxed{e \not\models^{\dagger}_{?} \xi_{pre}} \quad \boxed{\top \models^{\dagger}_{?} \xi_{pre} \vee \xi_{rest}}}{\Gamma\,;\Delta \vdash \texttt{match}(e)\{rs_{pre} \mid r \mid rs_{post}\} : \tau_2}$$

Fig. 1. Typing of Internal Expressions

as we evaluate the match expression and is particularly useful when returning an indeterminate match expression to the user, because it shows which rules have been checked.

$\boxed{\Gamma\,;\Delta \vdash [\xi_{pre}]rs : \tau_1[\xi_{rs}] \Rightarrow \tau_2}$ rs is of type $\tau_1 \Rightarrow \tau_2$, following rules constrained by ξ_{pre} and constrained by ξ_{rs}

TRules

$$\frac{\Gamma\,;\Delta \vdash r : \tau_1[\xi_r] \Rightarrow \tau_2 \qquad \Gamma\,;\Delta \vdash [\xi_{pre} \vee \xi_r]rs : \tau_1[\xi_{rs}] \Rightarrow \tau_2 \qquad \boxed{\xi_r \not\models \xi_{pre}}}{\Gamma\,;\Delta \vdash [\xi_{pre}]r \mid rs : \tau_1[\xi_r \vee \xi_{rs}] \Rightarrow \tau_2}$$

Fig. 2. Typing of rules

Fig. 2 describes the inductive construction of rules.

Constraints ξ is emitted from patterns. The syntax of them is given in Fig. 3. A constraint predict the set of expressions that match the corresponding pattern. For example, variable patterns or wild card patterns emit *Truth* constraint $\top$; a pattern hole emits *Unknown* constraint ?; patterns in multiple rules altogether emit disconjunction of constraints $\xi_1 \vee \xi_2 \vee \cdots \vee \xi_n$.

The dual of a constraint, $\overline{\xi}$, predicts the complement of the set of expressions that match the corresponding pattern of ξ. e.g. $\overline{\underline{n}} = \underline{\not{n}}$ means all numbers other than n; $\overline{\xi_1 \vee \xi_2} = \overline{\xi_1} \wedge \overline{\xi_2}$ means the complement of the union of two constraints.

$$\begin{array}{lcl} p & ::= & x \mid \underline{n} \mid \_ \mid (p_1, p_2) \mid \texttt{inl}(p) \mid \texttt{inr}(p) \mid (\!|)^w \mid (\!|p|\!)^w \\ \xi & ::= & \top \mid \bot \mid \underline{n} \mid \underline{\not{n}} \mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2 \mid (\xi_1, \xi_2) \mid \texttt{inl}(\xi) \mid \texttt{inr}(\xi) \mid ? \end{array}$$

Fig. 3. Syntax of patterns and constraints

3 EXHAUSTIVENESS AND REDUNDANCY

We define a satisfaction judgment between an expression and a constraint, $e \models \xi$, to represent if the expression match the associated pattern(s), e.g. any expression satisfies Truth constraint, $e \models \top$.

Similarly, we define a maybe satisfaction judgment $e \models_? \xi$ with the meaning that e may match the associated pattern(s). For example, an expression hole can be filled by any expression and thus may satisfy any constraint, i.e. $(\!|)^u \models_? \xi$ and $(\!|e|\!)^u \models_? \xi$.

Note that matching process only makes sense for *final* expressions, i.e. either values or indeterminate. The same goes for two sorts of satisfaction judgments.

Definition 3.1 gives the formal definition of exhaustiveness or maybe exhaustiveness. Rule TMatchZPre uses it to enforce the rules associated with ξ cover or may cover all the possibilities of the scrutinee. In Eq. 2, we notice that for any final expression e, either $e \models (\texttt{inl}(\top), \top)$ or $e \models_? (\top, ?)$. Therefore, Eq. 2 may be exhaustive and we shouldn't generate warning about non-exhaustiveness.

Definition 3.1 (Exhaustiveness or Maybe Exhaustiveness)**.** $\top \models^{\dagger}_{?} \xi$ *iff* $\xi : \tau$ *and for all* e *such that* $\cdot ; \Gamma \vdash e : \tau$ *and* e `final` *we have* $e \models \xi$ *or* $e \models_? \xi$.

Definition 3.2 gives the formal definition of redundancy. When rules are constructed, Rule TRules uses it to ensures that no single rule is redundant with respect to its previous rules. In Eq. 3, no matter how we fill the hole w_1, all final expressions that match $\texttt{inl}(⦇⦈^{w_1})$ also match $\texttt{inl}(x)$ and the same goes for the hole w_2. That means for all final expressions e, $e \models_? (\texttt{inl}(?), ?)$ implies $e \models (\texttt{inl}(\top), \top)$. Even if the second rule is not complete, we know it is redundant and thus we can suggest the programmer to either rewrite or eliminate it.

Definition 3.2 (Redundancy)**.** $\xi_r \models \xi_{pre}$ *iff* $\xi_r : \tau$ *and* $\xi_{pre} : \tau$ *and for all* e *such that* $\cdot ; \Gamma \vdash e : \tau$ *and* e `val` *we have* $e \models \xi_r$ *or* $e \models_? \xi_r$ *implies* $e \models \xi_{pre}$.

However, to determine if the redundancy judgement is true, we have to apply *material entailment of constraints* and $\xi_r \models \xi_{pre}$ is equivalent to $\top \models \overline{\xi_r} \vee \xi_{pre}$. Then, we only need to determine if $\overline{\xi_r} \vee \xi_{pre}$ can be satisfied by all expressions.

While Definition 3.1 ensures that the scrutinee at least matches or may match one of the constinuent rules, redundancy doesn't have anything to do with the scrutinee, it is simply the property of a single rule with respect to previous rules. Therefore, we only consider the expression that is already a value and recognize the redundant rule with confidence.

4 DYNAMIC SEMANTICS

As for pattern matching, either the scrutinee matches or may match or doesn't match the pattern. When there are only one remaining rule, the exhaustiveness or maybe exhaustiveness checking ensures that the pattern of the rule either must be matched or may be matched by the scrutinee.

Theorem 4.1 (Preservation)**.** *If* $\cdot ; \Delta \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot ; \Delta \vdash e' : \tau$

Theorem 4.2 (Progress)**.** *If* $\cdot ; \Delta \vdash e : \tau$ *then either* e `final` *or* $e \mapsto e'$ *for some* e'.

5 DISCUSSION

The abstract explores how constraint helps reasoning pattern matching with typed holes. One main contribution of the work is extending the match constraint language [Harper 2012] with Unknown constraint and introduce a three-way logic, including the concept of "maybe". The other contribution is the formalism of pattern matching with typed holes based on the development of the extended match constraint language.

Next, type holes and dynamic type casting would be added and our type system would be turned into a gradual type system. Recently, pattern matching statements have also been proposed to be added to Python, whose *Type Hints* is taken from the idea of gradual typing [Bucher et al. 2020; Siek and Taha 2006; van Rossum et al. 2014].

As a result, the work lays a foundation for integrating full-fledged pattern matching into the Hazel programming environment described in Omar et al. [2019].

REFERENCES

Brandt Bucher, Tobias Kohn, Ivan Levkivskyi, Guido van Rossum, and Talin. 2020. *Structural Pattern Matching*. PEP 622. https://www.python.org/dev/peps/pep-0622/

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

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. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. http://dl.acm.org/citation.cfm?id=3009900

Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*.

Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. *Type Hints*. PEP 484. https://www.python.org/dev/peps/pep-0484/