Pattern Matching with Typed Holes

YONGWEI YUAN*, University of Michigan

1 INTRODUCTION

Today's programming environment either only provides meaningful feedback for complete programs or simply takes heuristic approach to achieve similar result. By modeling incomplete programs as typed expression with *holes*, 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 without exceptions.

Pattern matching is a cornerstone of functional programming languages including ML family. An expression of product and sum types often rely on pattern matching to be eliminated. However, Omar et al. [2019] only introduces a rudimentary form of pattern matching that only allows case analysis on binary sum type and doesn't support nested pattern.

$$\begin{array}{l}\texttt{match}((\texttt{inl}_{\texttt{num}}(\text{⦶}^{u}), 2))\{\\ \mid (\texttt{inr}(x), \_) \Rightarrow x\\ \mid (\_, x) \Rightarrow x\\ \}\end{array} \quad (1) \qquad \begin{array}{l}\texttt{match}((\texttt{inl}_{\texttt{num}}(1), 2))\{\\ \mid (\texttt{inl}(x), \_) \Rightarrow x\\ \mid (\_, \text{⦶}^{w}) \Rightarrow \text{⦶}^{u}\\ \}\end{array} \quad (2) \qquad \begin{array}{l}\texttt{match}((\texttt{inl}_{\texttt{num}}(1), 2))\{\\ \mid (\texttt{inl}(x), \_) \Rightarrow x\\ \mid (\texttt{inl}(\text{⦶}^{w_1}), \text{⦶}^{w_2}) \Rightarrow \text{⦶}^{u}\\ \}\end{array} \quad (3)$$

Holes in match expression doesn't have to stop the match expression being evaluated. For a match expression shown in Eq. 1, the scrutinee contains a hole.By considering 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 $(\texttt{inr}(x), \_) \Rightarrow x$, and always matches $(\_, x)$. And thus the value of the match expression is 2.

Besides the computation result, the real time feedback as we enter a match expression can help programmers write down correct rules. However, it is subtle and complicated to check 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, i.e. *redundancy*.

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

2 STATIC SEMANTICS

Part of the definition of the internal language is shown in Fig. 1. Note that the match expression takes a sequence of rules with pointer, which is different from the examples in Sec. 1. Initially, the pointer is on the first rule (Rule TMatchZPre). As the match expression is evaluated, we consider those rules in order, if the scrutinee doesn't match the pointed rule, the pointer would move to the next rule. Rule TMatchNZPre gives the typing judgement for match expressions whose rule pointer has been moved away from the first rule.

Note that the premise of Rule TMatchZPre and Rule TMatchNZPre involves judgement of the form $\top \models^{\dagger}_{?} \xi$. That means, the rules in match expression is either exhaustive or maybe exhaustive. Take Eq. 2 as an example, the pattern hole ⦶^{w} may be matched by any expression. Therefore, by filling the holes, the rules may cover all the possibilities of scrutinee and we shouldn't give warning on non-exhaustiveness.

*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

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

TMatchNZPre

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

Fig. 1. Typing of Internal Expressions

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

TRules

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

Fig. 2. Typing of rules

Fig. 2 describes the inductive construction of rules. The premise $\xi_r \not\models \xi_{pre}$ in Rule TRules ensures that rule r is not redundant with respect to rules rs_{pre}. Take Eq. 3 as an example, all sub-expressions that match or may match $\texttt{inl}(\langle\!|\,|\!\rangle^{w_1})$ also matches $\texttt{inl}(x)$. 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.

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

Fig. 3. Syntax of patterns and constraints

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, *Truth* constraint $\top$ corresponds to variable pattern or wild card pattern that must be matched by all expressions; *Unknown* constraint ? corresponds to pattern hole $\langle\!|\,|\!\rangle^{w}$ or $\langle\!|p|\!\rangle^{w}$ that may be matched by all expressions; disconjunction of constraints $\xi_1 \vee \xi_2$ corresponds to multiple patterns.

TODO: dual 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 a union.

To give an idea of how rules is related to constraints. In Eq. 2, the corresponding constraint of two rules is $(\texttt{inl}(\top), \top) \vee (\top, ?)$. In Eq. 3, the constraint of the second rule is $(\texttt{inl}(?), ?)$ while the constraint of the first rule is $(\texttt{inl}(\top), \top)$. We will see in Sec. 3 how these constraints are used to check exhaustiveness and redundancy.

3 EXHAUSTIVENESS AND REDUNDANCY

Since constraint correspond to pattern, the relationship between expression and constraint should be defined in a way to represent that the expression match the associated pattern of the constraint.

We write $e \models \xi$ to denote that expression e satisfies constraint ξ, which is inductively defined by a set of rules. For example, any expression satisfy Truth constraint, $e \models \top$.

Similarly, we inductively define a maybe satisfaction judgement $e \models_? \xi$ to denote that expression e may satisfy constraint ξ, i.e. e may match the associated pattern. For example, expression hole, no matter it is empty or not, may satisfy any constraint, i.e. $(\!|\,|\!)^u \models_? \xi$ and $(\!|e|\!)^u \models_? \xi$, since we can always fill the hole with an expression that matches a given pattern.

Note that the judgements mentioned above only makes sense for final expressions. We know that an expression is a *value* when it is completely computed. We furthur say that an expression is indeterminate when it can't be evaluated due to the existence of holes [Omar et al. 2019]. And a final expression is either a value or indeterminate.

For Eq. 2, we notice that for all final expression e, either $e \models (\texttt{inl}(\top), \top)$ or $e \models_? (\top, ?)$. Therefore, we can say Eq. 2 may be exhaustive. Definition 3.1 gives the formal definition of exhaustiveness or maybe 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* $e \models_? \xi$.

By comparing two constraints $(\texttt{inl}(\top), \top)$ and $(\texttt{inl}(?), ?)$ that corresponds to two rules in Eq. 3, we notice that $e \models_? (\texttt{inl}(?), ?)$ implies $e \models (\texttt{inl}(\top), \top)$, which means that no matter how we fill the holes in $(\texttt{inl}((\!|\,|\!)^{w_1}), (\!|\,|\!)^{w_2})$, all expressions that matches the completed pattern also matches $(\texttt{inl}(x), \_)$. The second rule must be redundant. Definition 3.2 gives the formal definition of redundancy.

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* $e \models \xi_{pre}$.

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 match 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'.

Theorem 4.1 and Theorem 4.2 together establishes the type safety of our system. Theorem 4.1 ensures that no matter where the rule pointer is at, the match expression is still well-typed. If the pointer is at the first rule (Rule TMatchZPre), we simply have $e \models^\dagger_? \xi$. Since ξ predicts the set of expressions that match any one of the rules, the scrutinee either matches or may match one of the rules; If the pointer is not at the first rule (Rule TMatchNZPre), the premises gives us $e \models^\dagger_? \xi_{rest}$. Besides, ξ_{rest} is the corresponding constraint of the remaining rules, and thus the scrutinee either matches or may match the current rule, or matches or may match one of the rest rules.

Therefore, we conclude that any well-typed match expression can take a step (match) or is indeterminate (may match), which constructs the proof of Theorem 4.2.

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.

Pattern matching statements have been proposed to be added to Python, whose *Type Hints* is taken from the idea of gradual typing [Bucher et al. 2020; van Rossum et al. 2014]. Next, we will also incorporate the idea of gradually typing by adding type holes and dynamic type casting to the current calculus ([Siek and Taha 2006]).

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/