Pattern Matching with Holes

YONGWEI YUAN, University of Michigan

1 INTRODUCTION

Pattern matching is a cornerstone of functional programming. An expression of product and sum types often rely on pattern matching to be eliminated. The expression that corresponds to the concept of pattern matching is called *match* expression. The first argument to match expression is called *scrutinee*, and the second argument consists of a finite sequence of *rule*. Each rule consists of a *pattern* that may introduce variables and an expression that may involve those variables. The meaning of match expression is given by considering each rule in order until the first rule whose pattern matches the scrutinee is found. With the context extended during pattern matching, the expression of the matched rule is computed [Harper 2012].

Pattern matching is widely used in the ML family. However, the way how match expression is structured determine that it requires programmer to deal with many components at the same time, which is impossible. Even worse, throughout the process of entering a match expression, there are substantial lengths of time when the expression is not well-typed by nature.

There have been works that address the problem of such "gap" caused by incomplete program. 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. However, the pattern matching described in the appendix of Omar et al. [2019] only work with sum types. In other words, it doesn't allow nested patterns, which is the more interesting case and is how pattern matching looks like in modern functional programming languages.

This abstract focuses on the formalism of full-fledged pattern matching with holes such that programming environment or programmer could understand incomplete match expression.

The main contribution of the work is about reasoning the behavior that are specific to pattern matching instead of simply introducing holes into the semantics of pattern matching. In particular, the static semantics is defined in a way to enforce *exhaustivenss* and *redundancy* of rules. In a match expression, there is at least one of the rules that the scrutinee matches or may match. Hence, any well-typed match expression is guaranteed to have no exception. Besides, no rule is definitely redundant with respect to preceding rules, *i.e.*, no rule could be safely eliminated without affecting the execution of pattern matching.

2 STATIC SEMANTICS

Fig. 1 only includes typing of match expression due to page limits. The match expression $\texttt{match}(e)\{\hat{rs}\}$ takes two arguments, the expression to match, e, called scrutinee, and a series of rules with pointer, $\hat{rs}$. When a match expression hasn't been evaluated, the pointer is on the first rule, which is the case of Rule TMatchZPre. As 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 from the first rule.

Since our focus is more on the side of pattern matching, we doesn't inherit the bidirectional static semantics from Omar et al. [2017]. Instead, the static semantics aims to predict the behavior of match expression in runtime. For example, the last premise of Rule TMatchZPre is used to check the exhaustivenss, which ensures that the match expression satisfy Thm. 3.2

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}

TZeroRule

$$\frac{}{\Gamma\,;\Delta \vdash [\xi]\cdot : \tau_1[\bot] \Rightarrow \tau_2}$$

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 defines the typing judgement for rules. When rules is constructed inductively in Rule TRules, the constraints on previous rules is accumulated and the redundancy of each rule is checked in order by comparing the constraint on the current rule, ξ_r, with the constraint on previous rules, ξ_{pre}. Meanwhile, the constraint on rules $r \mid rs$, $\xi_r \vee \xi_{rs}$ is accumulated as well.

The constraint on a single rule is derived from its pattern. For example, a variable pattern corresponds to *Truth* constraint, $\top$, while a pattern hole, $(\!|)^w$ or $(\!|p|\!)^w$, corresponds to *Unknown* constraint, ?. Therefore, the typing judgement of a pattern is of the form $p : \tau[\xi] \dashv\Gamma\,;\Delta$.

Basically, constraint predict the set of expressions that match the corresponding rule, or rules if the constraint is a accumulated one.

3 DYNAMIC SEMANTICS

The stepping judgement of match expressions are composed of four rules. The first rule states that if the scrutinee can take a step, the match expression can take a step. The other three rules are closely related to pattern matching and describe three possible matching result:

(1) The scrutinee matches the pattern of the current rule, next step is to evaluate the corresponding expression under the rule.
(2) The scrutinee may match the pattern of the current rule, we say that the match expression is *indeterminate*, the concept of which originates from Omar et al. [2019].
(3) The scrutinee doesn't match the pattern of the current rule, the pointer will be moved to the next rule. Theorem 3.2 ensures that there are remaining rules for the pointer to move on.

The following theorem establishes that for a match expresion, no matter the scrutinee matches the current rule or not, the stepping preserves its typing.

Theorem 3.1 (Preservation). *If* $\cdot\,;\Delta \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot\,;\Delta \vdash e' : \tau$

Since Theorem 3.1 ensures that no matter where the rule pointer is at, the match expression is still well-typed. For Rule TMatchZPre, we simply have $e \models^{\dagger}_{?} \xi$. Since ξ constrains the set of expressions that match the rules, the judgement means that the scrutinee either matches or may match one of the rules. For 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 or is indeterminate, which constructs the proof of the following theorem.

Theorem 3.2 (Progress). *If* $\cdot\,;\Delta \vdash e : \tau$ *then either* e `final` *or* $e \mapsto e'$ *for some* e'.

Theorem 3.2 states that for all well-typed expression, either it is final, i.e., a value or is indeterminate, or it can take a step. The proof relies on the one-to-one correspondance between pattern matching and constraint satisfaction.

4 EXHAUSTIVENESS AND REDUNDANCY

With the intuition of what constraint is for as discussed in Sec. 2 and Sec. 3, we will talk about how constraint is used to check exhaustivenss of rules and redundancy of any single rule and give formal definition.

$e \models \xi$ is inductively defined by a set of rules, with meaning that expression e satisfies constraint ξ. For example, any expression satisfy Truth constraint, $e \models \top$.

$e \models_? \xi$ is inductively defined by a set of rules, with the meaning that expression e may satisfy constraint ξ. 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 expression hole may match arbitrary patterns.

Assume the constraint is associated with all the rules in the match expression. The definition of the constraint's exhaustivenss is that for all final expression of the same type as the constraint, it either satisfy or may satisfy the constraint. Since the constraint is a disjunction of constrains that are associated with each rule in the match expression respectively, the definition means either e matches or may match one of the rules. The definition requires that the expression is final because exhaustiveness is reasoning the runtime behavior of the match expression. Consequently, even if the scrutinee is a hole, the progress will still hold.

Note that only when the match expression can't be exhaustive, should the programming environment warn the programmer. Therefore, the exhaustiveness we define here is actually "exhaustiveness or maybe exhaustiveness".

Definition 4.1 (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* $e \models_? \xi$.

Assume the constraint ξ_r is associated with the current rule, r, and the constraint ξ_{pre} is associated with the previous rules, rs_{pre}. The idea that rule r is redundant with respect to rs_{pre} is described by $\xi_r \models \xi_{pre}$. The definition requires that the expression is a value because redundancy checking is helping programmer eliminate redundant code and the definition is based on the fact that the scrutinee is complete (without holes).

Similar to the exhaustiveness, only when some rule must be redundant, should we warn the programmer. Therefore, for those expression that may match the current rule, they should match one of the previous rules so that we can say with confidence that the current rule can't be reached.

Definition 4.2 (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* $e \models \xi_{pre}$.

REFERENCES

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