

Pattern Matching with Typed Holes

Yongwei Yuan

University of Michigan

What is Pattern Matching

```
match (3::[ ]) {
| [ ] -> "empty"
| 1::xs -> "1,..."
| 2::xs -> "2,..."
}
```

Listing 1: Not Exhaustive

```
match (2::[ ]) {
| [ ] -> "empty"
| 1::xs -> "1,..."
| 1::2::xs -> "1,2,..."
}
```

Listing 2: Redundant Branch

Why Adding Holes to Pattern Matching

- Chapter *Pattern Matching* in PFPL [Harper, 2012] introduces a match constraint language to check exhaustiveness of a match expression and redundancy of a single rule. We **extend the constraint language with *Unknown constraint*** and adapt the checking algorithm to our setting.
- Hazel is a programming environment featuring typed holes [Omar et al., 2017, Omar et al., 2019], but it only supports simple case analysis on binary sum types. We want to **formalize the full-fledged pattern matching with Typed Holes**.
- Agda allows the programmer to automatically generate code through "case splitting", our work is focused on **giving live feedbacks and guidance** as the programmer enters a match expression.

How Pattern Matching with Typed Holes Works

```
match (inr((1, (⊖u))) {
| inl(()) => "empty"
| inr((⊖w, xs)) => "1,..."
| inr((1, inr((2, xs)))) => "1,2,..."
}
```

Listing 3: A match expression in our internal language

```
match (inr((1, (⊖u))) {
| inl(()) => "empty"
| inr((⊖w, xs)) => "1,..."
| inr((1, inr((2, xs)))) => "1,2,..."
}
```

Listing 4: Does Not Match

```
match (inr((1, (⊖u))) {
| inl(()) => "empty"
| inr((⊖w, xs)) => "1,..."
| inr((1, inr((2, xs)))) => "1,2,..."
}
```

Listing 5: May Match

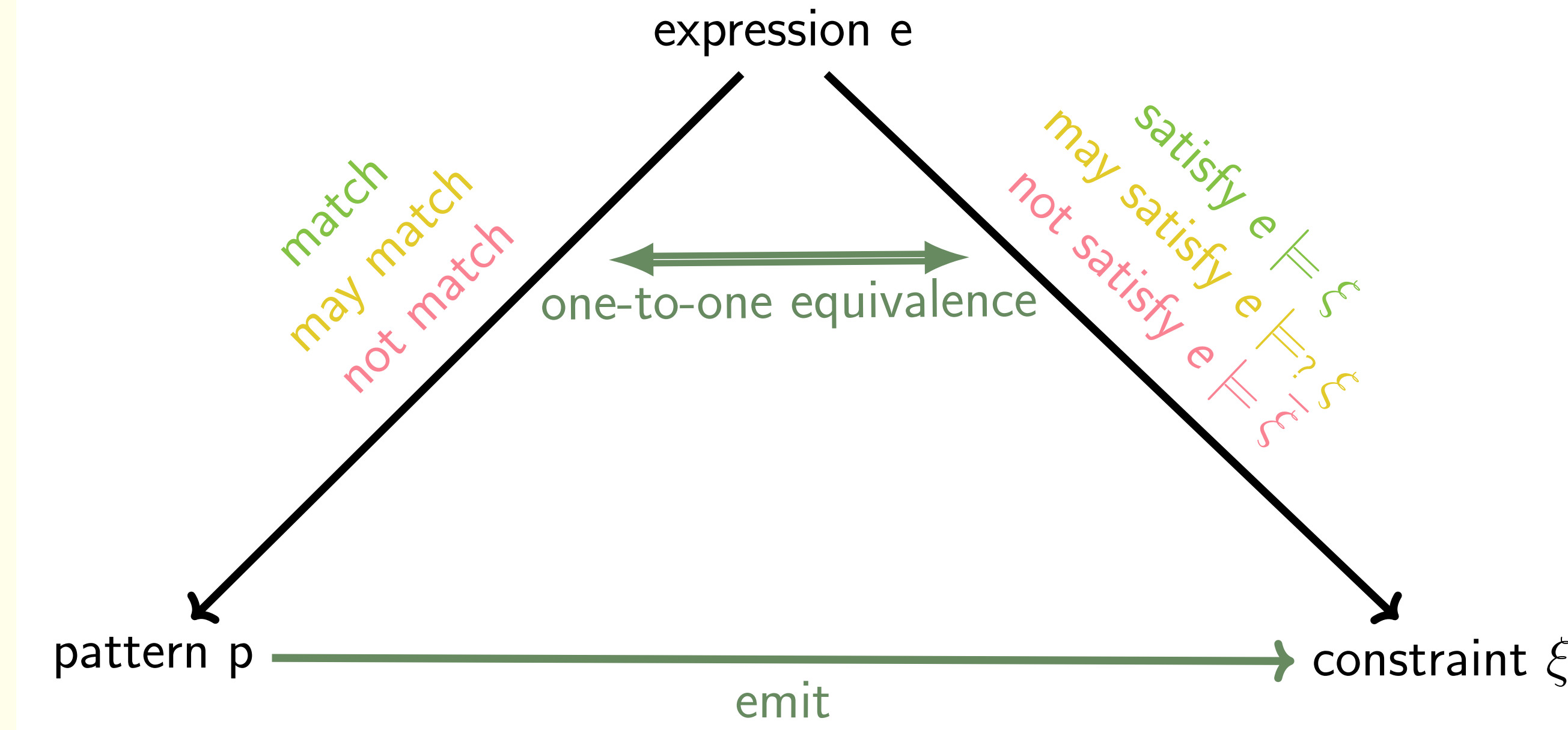
We know that an expression that cannot be further evaluated is a **value**. When the value of an expression is indeterminate due to unfilled holes, such an expression is **indeterminate**. For example,

- any expression that contains an expression hole $(\ominus)^u$ or $(\ominus e)^u$
- a match expression in which the scrutinee may match the branch under the pointer

And an expression is **final** when it is either a value or indeterminate.

Expressions, Patterns and Constraints

$e ::= x \mid () \mid \underline{n} \mid (e_1, e_2) \mid \text{inl}_\tau(e) \mid \text{inr}_\tau(e) \mid (\ominus)^u \mid (\ominus e)^u$
 $\mid (\lambda x : \tau. e) \mid e_1(e_2) \mid \text{match}(e) \{ \hat{r}\hat{s} \}$
 $p ::= x \mid - \mid () \mid \underline{n} \mid (p_1, p_2) \mid \text{inl}(p) \mid \text{inr}(p) \mid (\ominus)^w \mid (\ominus p)^w$
 $\xi ::= \top \mid \perp \mid () \mid \underline{n} \mid \underline{n} \mid (\xi_1, \xi_2) \mid \text{inl}(\xi) \mid \text{inr}(\xi) \mid ? \mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2$



For example,

pattern hole $(\ominus)^w$ or $(\ominus p)^w$ $\xrightarrow{\text{emit}}$ unknown constraint ?
 branch(rule) $\text{inr}((\ominus^w, xs)) \Rightarrow \text{"empty"}$ $\xrightarrow{\text{emit}}$ constraint $\text{inr}((?, \top))$
 branches(rules) $r_1 | r_2 | r_3 | \dots$ $\xrightarrow{\text{emit}}$ constraint $\xi_1 \vee \xi_2 \vee \dots$

Typing Judgment of Branches (Rules rs) and Redundancy

Branch	Constraint
$\text{inl}() \Rightarrow \text{"empty"}$	$\rightarrow \text{inl}()$
$\text{inr}((\ominus^w, xs)) \Rightarrow \text{"empty"}$	$\rightarrow \text{inr}((?, \top))$
$\text{inr}((1, \text{inr}((2, xs)))) \Rightarrow \text{"empty"}$	$\rightarrow \text{inr}((1, \text{inr}((2, \top))))$

A branch is redundant *iff* all expressions that match or may match that branch, must match one of its preceding branches.

We use emitted constraints to check redundancy of every single branch

- the second branch is not redundant, $\text{inr}((?, \top)) \not\models \text{inl}()$
- the third branch is not redundant, $\text{inr}((1, \text{inr}((2, \top)))) \not\models \text{inl}() \vee \text{inr}((?, \top))$

Typing Judgment of Match Expression and Exhaustiveness

Branches must or may be exhaustive *iff* all expressions either matches one of the branches or may match one of the branches.

We use the constraint emitted from the three branches to enforce that the match expression must or may be exhaustive

$\top \models_? \text{inl}() \vee \text{inr}((?, \top)) \vee \text{inr}((1, \text{inr}((2, \top))))$

Definition ("Must" or "May" Entailment)

$\xi_1 \models_?^{\dagger} \xi_2$ *iff* $\xi_1 : \tau$ and $\xi_2 : \tau$ and for all e such that $\cdot ; \Gamma \vdash e : \tau$ and e final we have $e \models \xi_1$ or $e \models_? \xi_1$ implies $e \models \xi_2$ or $e \models_? \xi_2$.

Definition ("Must" Entailment)

$\xi_1 \models \xi_2$ *iff* $\xi_1 : \tau$ and $\xi_2 : \tau$ and for all e such that $\cdot ; \Gamma \vdash e : \tau$ and e val we have $e \models \xi_1$ or $e \models_? \xi_1$ implies $e \models \xi_2$.

De-unknown

Exhaustiveness or Maybe Exhaustiveness

$\top \models_?^{\dagger} \text{inl}() \vee \text{inr}((?, \top)) \vee \text{inr}((1, \text{inr}((2, \top))))$
 $\top \models \text{inl}() \vee \text{inr}((\top, \top)) \vee \text{inr}((1, \text{inr}((2, \top))))$

Redundancy of Second Branch

$\text{inr}((?, \top)) \models \text{inl}()$
 $\text{inr}((\top, \top)) \models \text{inl}()$

Redundancy of Second Branch

$\text{inr}((1, \text{inr}((2, \top)))) \models \text{inl}() \vee \text{inr}((?, \top))$
 $\text{inr}((1, \text{inr}((2, \top)))) \models \text{inl}() \vee \text{inr}((1, \top))$

Then, we can apply similar checking algorithm as described in Chapter *Pattern Matching* of PFPL [Harper, 2012].

Conclusion

We have formalized the type system and are still working on the proof of the correctness of exhaustiveness checking and redundancy checking. The idea has been implemented in a toy programming language (<https://github.com/fplab/pattern-paper/tree/master/src>).

References

- Harper, R. (2012). *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge.
- Omar, C., Voysey, I., Chugh, R., and Hammer, M. A. (2019). Live functional programming with typed holes. *Proc. ACM Program. Lang.*, 3(POPL):14:1–14:32.
- Omar, C., Voysey, I., Hilton, M., Aldrich, J., and Hammer, M. A. (2017). Hazelnut: a bidirectionally typed structure editor calculus. In Castagna, G. and Gordon, A. D., editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 86–99. ACM.