Pattern Matching with Typed Holes

Yongwei Yuan

University of Michigan

September 19, 2020

Supervised by Cyrus Omar at Future of Programming Lab

Pattern Matching

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

The scrutinee does not match the pattern in the first branch.

Pattern Matching

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

The scrutinee matches the pattern in the second branch.

Exhaustiveness and Redundancy

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

Not Exhaustive

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

Redundant Branch

Chapter *Pattern Matching* in PFPL [**?**] introduces a match constraint language and the algorithm to check exhaustiveness of branches and redundancy of a single branch with respect to its preceding branches.

What if Match Expression is Incomplete?

- Agda and Haskell allow programmers to use typed holes to represent missing parts in the program.
- Hazel is a live programming environment featuring typed holes [**?**, **?**], but it only supports simple case analysis on binary sum types.

What if Match Expression is Incomplete?

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

Expression hole

What if Match Expression is Incomplete?

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

Expression hole

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

Pattern hole

What if Match Expression is Incomplete?

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

Expression hole

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

Pattern hole

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

Exhaustive?

What if Match Expression is Incomplete?

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

Expression hole

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

Pattern hole

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

Exhaustive?

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

Redundant?

The key point behind reasoning about incomplete programs is to give feedback that is always correct no matter how programmers fill these holes at the end.

Early Evaluation

We can evaluate the expression even if there are holes
as long as the evaluation is correct regardless of how holes are filled.

Expression Hole doesn't have to Stop Evaluation

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

The scrutinee does not match the pattern in the first branch.

Expression Hole doesn't have to Stop Evaluation

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

The scrutinee matches the pattern in the second branch.
The value of the match expression is a string "1,...".

Pattern Hole May be Matched

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

The scrutinee does not match the pattern in the first branch.

Pattern Hole May be Matched

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

The scrutinee **may match** the pattern in the second branch.
Because 1 may match the hole ? depending on what is filled.
We say that the match expression is **indeterminate**.

Evaluate as further as possible

We keep evaluating the expression until it is either a **value** or **indeterminate**.
We regard such an expression as **final**.

Best-Case Error Reporting

Report error only when it can't be avoided.

No Error Messsage when the Branches May be Exhaustive

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

Any pattern of Num type can be filled in the hole.

No Error Messsage when the Branches May be Exhaustive

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

By filling the hole with some variable, the branches can be exhaustive.

Prompt Error Messsage only when the Branches Mustn't be Exhaustive

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

Any pattern of List of Num type can be filled in the hole.

Prompt Error Messsage only when the Branches Mustn't be Exhaustive

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

No matter what pattern we fill in the hole,
the branches can't be exhaustive.

No Error Messsage when the Branch may be Redundant

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

Any pattern of Num type can be filled in the hole.

No Error Messsage when the Branch may be Redundant

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

By filling the hole with number 2, the third branch can be redundant.

No Error Messsage when the Branch may be Redundant

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

By filling the hole with a variable, the third branch is also redundant.

No Error Messsage when the Branch may be Redundant

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

By filling the hole with a number other than 2,
the third branch is not redundant.

Prompt Error Messsage when the Branch must be Redundant

match (3::[ ]) {
| [ ] −> "empty"
| x::xs −> "..."
| ?::xs −> "?,..."
}

Any pattern of Num type can be filled in the hole.

Prompt Error Messsage when the Branch must be Redundant

match (3::[ ]) {
| [ ] −> "empty"
| x::xs −> "..."
| 1::xs −> "?,..."
}

No matter what pattern we fill in the hole,
the third branch must be exhaustive.

Prompt Error Messsage when the Branch must be Redundant

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

No matter what pattern we fill in the hole,
the third branch must be exhaustive.

Prompt Error Messsage when the Branch must be Redundant

match (3::[ ]) {
| [ ] −> "empty"
| x::xs −> "..."
| y::xs −> "?,..."
}

No matter what pattern we fill in the hole,
the third branch must be exhaustive.

Prompt Error Message or Not

Yes

- Branches mustn't be exhaustive
- Some branch must be redundant

No

- Branches must be exhaustive
- Branches may be exhaustive
- Every branch either
  - mustn't be redundant
  - may be redundant

A Match Expression in Our Internal Language

match (inr((1, $(\!|\,|\!)^u$))) {
| inl(()) −> "empty"
| inr(($(\!|\,|\!)^w$, xs)) −> "1,..."
| inr((1, inr((2, xs)))) −> "1,2,..."
}

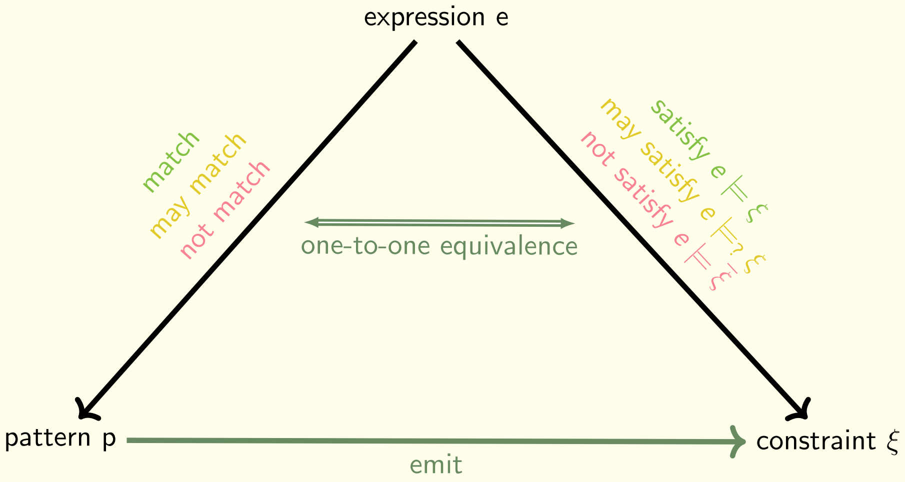
1::?

[]

?::xs

1::2::xs

Constraint Emitting and Relationship Triangle



Branches and $\vee$ constraint

Branch r	Constraint ξ
$\mid \texttt{inl}(()) \Rightarrow$ "empty"	$\texttt{inl}(())$
$\mid \texttt{inr}((\llparenthesis\rrparenthesis^{w}, xs)) \Rightarrow$ "empty"	$\texttt{inr}((?, \top))$
$\mid \texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, xs)))) \Rightarrow$ "empty"	$\texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, \top))))$

And multiple branches correspond to their constraints connected by $\vee$.

Entailment of Constraints

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* $e \models \xi_2$.

Redundancy(must) Checking in Statics

Branch r	Constraint ξ
$\mid \mathtt{inl}(()) \Rightarrow$ "empty"	$\mathtt{inl}(())$
$\mid \mathtt{inr}((\llparenthesis\rrparenthesis^{w}, xs)) \Rightarrow$ "empty"	$\mathtt{inr}((?, \top))$
$\mid \mathtt{inr}((\underline{1}, \mathtt{inr}((\underline{2}, xs)))) \Rightarrow$ "empty"	$\mathtt{inr}((\underline{1}, \mathtt{inr}((\underline{2}, \top))))$

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

Redundancy(must) Checking in Statics

Branch r	Constraint ξ
$\mid \texttt{inl}(()) \Rightarrow$ "empty"	$\texttt{inl}(())$
$\mid \texttt{inr}((\llparenthesis\rrparenthesis^w, xs)) \Rightarrow$ "empty"	$\texttt{inr}((?, \top))$
$\mid \texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, xs)))) \Rightarrow$ "empty"	$\texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, \top))))$

TRules

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

Exhaustiveness(must or maybe) Checking in Statics

Branch r	Constraint ξ
$\mid \texttt{inl}(()) \Rightarrow$ "empty"	$\texttt{inl}(())$
$\mid \texttt{inr}((\llparenthesis\rrparenthesis^{w}, xs)) \Rightarrow$ "empty"	$\texttt{inr}((?, \top))$
$\mid \texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, xs)))) \Rightarrow$ "empty"	$\texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, \top))))$

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

Exhaustiveness(must or maybe) Checking in Statics

Branch r	Constraint ξ
$\mid \texttt{inl}(()) \Rightarrow$ "empty"	$\texttt{inl}(())$
$\mid \texttt{inr}((\llparenthesis\rrparenthesis^w, xs)) \Rightarrow$ "empty"	$\texttt{inr}((?, \top))$
$\mid \texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, xs)))) \Rightarrow$ "empty"	$\texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, \top))))$

TMatchZPre

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

De-unknown in Exhaustiveness Checking

$$\top \models^{\dagger}_{?} \texttt{inl}(()) \vee \texttt{inr}((?, \top)) \vee \texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, \top))))$$

$$? :: xs \longrightarrow x :: xs$$

$$\top \models \texttt{inl}(()) \vee \texttt{inr}((\top, \top)) \vee \texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, \top))))$$

De-unknown in Redundancy Checking

Second branch:

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

$$? :: xs \longrightarrow x :: xs$$

$$\texttt{inr}((\top, \top)) \models \texttt{inl}(())$$

Third Branch:

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

$$? :: xs \longrightarrow 2 :: xs, \text{ or } 3 :: xs, \text{ or } \ldots$$

$$\texttt{inr}((\underline{1}, \texttt{inr}((\underline{2}, \top)))) \models \texttt{inl}(()) \vee \texttt{inr}((\bot, \top))$$

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

Conclusion

- Formalize pattern matching with typed holes in a type system
- Develop a theoretical foundation for constant feedback on match expressions
- Implement the type system in a toy programming language (`https://github.com/fplab/pattern-paper/tree/master/src`)

Conclusion

- Formalize pattern matching with typed holes in a type system
- Develop a theoretical foundation for constant feedback on match expressions
- Implement the type system in a toy programming language (`https://github.com/fplab/pattern-paper/tree/master/src`)

What is next?

- Prove the correctness of checking algorithm
- Integrate it into Hazel

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.