Fully-Annotated Extraction from Hazel to OCaml

Siyuan He
hesy@umich.edu

April 25, 2020

Abstract

We proposed a fully annotated extraction from Hazel to OCaml in this project, where "fully annotated" means we will explicitly show the types of all variables in the extracted OCaml code. The work involves recording types of variables in Hazel and translating to OCaml. For some cases that types are not explicitly given in Hazel, our extraction will do best-effort type inference. We design an intermediate type and a series of inference rule to help type inference. The main idea is to first map Hazel code into our type and expression, do inference and expression translation, and finally translate to OCaml code. Most parts of the extraction program are formally defined.

1 Introduction

The source programming language of our extraction, Hazel, is a live functional programming environment with typed holes, and allowing incomplete programs (Omar, Voysey, Chugh, & Hammer, 2019). Hazel spent a lot of time working on the incomplete programs by introducing and formalizing typed holes, and Hazel enables type check and even run incomplete programs.

One important feature for Hazel is the editor based nature, meaning that every edition in Hazel is viewed as an action to change the "editor state". The rule of editor states are formally implemented, hence there're no meaningless editor states (Omar et al., 2019). From the view of user, editing code in Hazel is like filling boxes in the editor. For example, typing `"let" + space` in keyboard will introduce a `let` expression `let` $\boxed{0}$ `=` $\boxed{1}$ `in` $\boxed{2}$. The editor will create three holes for you, and the user are only allowed to edit inside holes.

There is a very nice introduction and tutorial on the Hazel website `https://hazel.org/`. It would be better for understanding if you could first have a quick view on the website before reading the following parts of this report.

The programming syntax of Hazel is quite similar to typed lambda calculus. Namely, the `let...in` expression and λ expression makes up the main structure of Hazel programs, along with other common elements in functional programming language such as `case` and `injection`. However, there's a big difference between Hazel and typed lambda calculus in that user doesn't always need to explicitly provide types for variables. For example, the expression $\lambda\ \boxed{0}:\boxed{?}.\{\boxed{1}\}$ in Hazel has a $\boxed{?}$ hole, which doesn't necessarily need to be filled. The editor will assert an error if there doesn't exist any type satisfied. You can refer to an example code (figure 2) for a intuitive understanding.

However, Hazel isn't a complete editor, and there are a lot of functionalities waiting to be implemented. One current problem for Hazel is that Hazel is fully rely on the editor,

hence we can't run hazel outside its editor. One solution is to extract Hazel code to some popular languages. In this project we choose OCaml to be our target language because OCaml is also a functional programming language, and the syntax of OCaml is similar to Hazel to some extent.

2 Instruction for Usage

2.1 How to Use

Since I am a member of FPlab, the group developing and maintaining Hazel, I publish my work on the *extraction* branch of the Hazel github. You can find every code on the page `https://github.com/hazelgrove/hazel/tree/extraction`.

Note that we're making progress on Hazel continuously, hence you may found the code updated and become different from the report. Meanwhile, the version of Hazel base code on the extraction branch is a little (or quite a lot) slower than the master or dev branch, because I want a relatively stable environment. Indeed, the Hazel AST had a big update two months ago, hence I need to rewrite a lot of functions.

Here lists the instruction for you to try extraction. You can read the `readme` file for more instructions on how to compile and run it.

1. Prepare a system or shell supporting Linux-like commands with environment *Opam 4.08* or higher, while *4.08.1* is highly suggested. It's tested ok to use *WSL2* on *Windows*.

2. Clone the git `https://github.com/hazelgrove/hazel.git` and move to the `hazel` folder.

3. **Switch to `extraction` branch**.

4. Run `make deps` to install dependency. If you encounter some error, please read the error message and install missing libraries.

5. Run `make` to build.

6. Run `make echo-html` . It will display a filename, then you could use browser to open the file.

   For Windows you could directly run `make win-chrome` . For MacOS you can run `open $(make echo-html)` to open the file by browser as a sugar.

7. Write code in the Hazel Editor.

8. Open the console of the browser and click "extraction to OCaml" button at the bottom of the bar at right side, and you will found the OCaml code is displayed in the console. The button usually locates at the right-bottom of the whole website.

2.2 Example

If you follow the instructions, you should get a result similar to this example. The example shows the result of a typical extraction result. You can easily reproduce the result by typing the hazel code and press the "extraction to ocaml" button (if the code is updated, please rollback to the last version before the date of this paper).

```
let x = 1 in
let f = λx:Num.{x + 1} in
let y =
  case (f x)
  | 1 ⇒ true
  | _ ⇒ false
  end : ?
in
y
```

RESULT OF TYPE: ?

Figure 2: Hazel Code

```
let (x:int) = 1
 in
let (f:int->int) = (fun (x:int) -> (x:int)+1
)
 in
let (y:bool) = ((match ((f:int->int) (x:int)
)
 with
 | 1 -> true

 | _ -> false

) : bool)
 in
(y:bool)
```

Figure 3: Extraction Result

In the example, we show the fully-annotated extraction including `let` expression, λ expression and `case` expression in Hazel. You can find that every variable in our extracted code is annotated with a type.

The example also shows the type inference of the extraction. You can see in each `let` expression, we don't explicitly assign a type to the variable we declare, but the type is explicitly annotated in the OCaml code. Our extraction provide best effort automatic detection on the type of the variables. The inference also happens in the `case` expression, where we leave a ? type hole in Hazel code but automatically fill the type in extracted OCaml code.

The format of our code is not beautiful as you see. We will add a auto-formater in the future, or make it look nicer.

3 Technical Design of Extraction

3.1 Type Translation

Our idea is to declare a intermediate type, naming `pass_t`. We map Hazel types to `pass_t` and map `pass_t` to OCaml types, leaving every inference operation only on `pass_t`. In other word, Hazel types and OCaml types don't appear in our extraction program except for the start and exit. This design is to keep a relative stable coding environment on the situation that Hazel structure may continuously change.

Here we give formal definition of `pass_t`:

```
pass_t ::=HOLE|Bool|Number|Unit|List(pass_t)|
        ARROW(pass_t, pass_t)|SUM(pass_t, pass_t)|PROD(pass_t, pass_t)|
        EMPTY|UNK|CONFLICT
```

We don't give formal definition on the type translation map here, because the map is quite simple and mostly a one-to-one map. We think it doesn't help understanding our design to formalize the type map here, but if you really want, you can simply think them as:

$$\frac{\Sigma_{\text{Hazel}}, \Gamma_{\text{Hazel}} \vdash x : \text{Num} \quad \Sigma_{\text{Hazel}}, \Gamma_{\text{Hazel}}, \Sigma, \Gamma \vdash x \longrightarrow_e x'}{\Sigma, \Gamma \vdash x' : \texttt{Number}}$$

where the symbol $\longrightarrow_e$ means the pattern x in Hazel is translated to x' in intermediate expression (will be discussed later).

The figure below shows the type translation map in our extraction, note that the `pass_t` type can be changed to other `pass_t` type during the extraction.

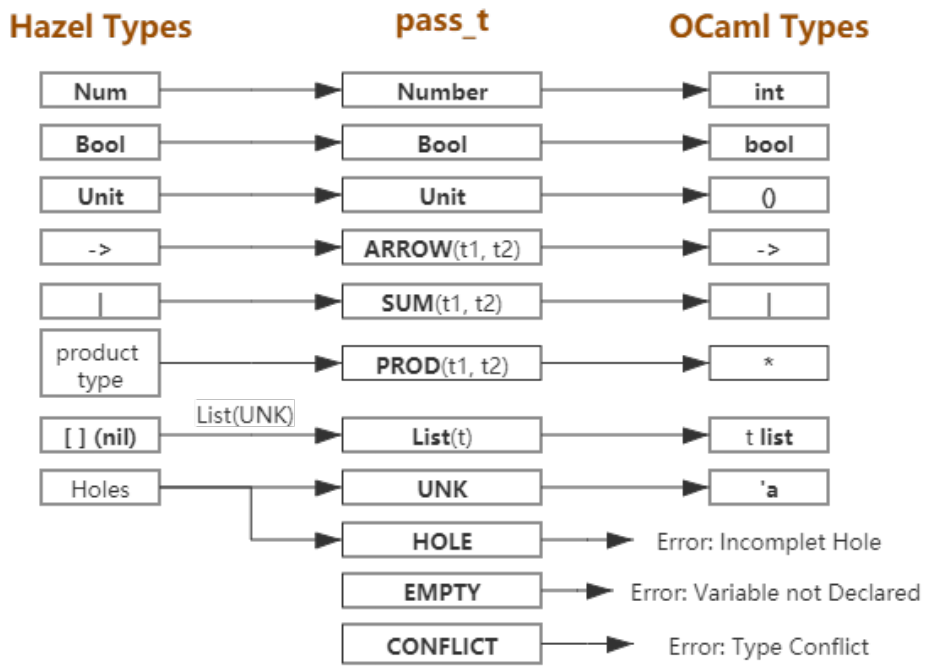

Figure 3: Type Translation Map

The meaning of each item in `pass_t` is quite straight-forward from its meaning. The `HOLE`, `CONFLICT`, `UNK`, `EMPTY` are four types internally used during the extraction process.

1. `HOLE` is a type indicating we find a incomplete hole. The $\boxed{?}$ option hole will never trigger a `HOLE`.

2. `UNK` is like a "placeholder" of a type. It appears whenever we lack information to determine (infer) a type, and hoping to infer it later. Here is a (badly designed) trick that we give all no-type terms the type `UNK` because it won't affect other type, such as string.

3. `EMPTY` is a type indicating that we can't find a variable in the environment, so it's related to "Not defined variable" error.

4. Types will be changed to `CONFLICT` whenever a type conflict is found. We'll always pass `CONFLICT` to top level, meaning that if we have a `CONFLICT`, we'll stop translating.

3.2 Syntax Translation

Since all operations in Hazel are quite common in all functional programming language, we can easily find a corresponding syntax in OCaml for every Hazel syntax. We don't show all translations in this part because the limit of space, and we only show an example for you.

Notation: "Hazel: e" means the e is a Hazel term.

Definition: Σ, Γ are the environment we're using through the whole system. Σ records the type of type variables, and Γ records the map from a variable to its type. For example, adding x : int will become firstly create a type variable t_x to Γ, and assign $t_x :=$ int in Σ, i.e. $\Sigma[t_x :=$ int$], \Gamma[x : t_x]$.

$$\frac{\text{Hazel: "}\lambda\underline{x} : \underline{t}.\underline{e}\text{"} \quad \Sigma, \Gamma \vdash t, e \longrightarrow_{t,e} t', e'}{\text{OCaml:"fun } \underline{x} : \underline{t'} \rightarrow \underline{e'}\text{"}} (\textbf{INCOMPLETE})$$

where the underline patterns mean that they're variables and can be arbitrarily changed into any valid pattern, and symbol $\longrightarrow_{t,e}$ means applying type pattern, or expression translation. The rule says that if we can translate the types, patterns and expressions (will discuss later) from Hazel to OCaml, we can translate the string of Hazel code to the string of OCaml code.

Note that our extraction is fully-annotated, hence we need to annotate the types in the extracted OCaml code. We rewrite the rule to be:

$$\frac{\text{Hazel: "}\lambda\underline{x} : \underline{t}.\underline{e}\text{"} \quad \Sigma, \Gamma \vdash t \longrightarrow_{t,e} t' \quad \Sigma[t := t'_x], \Gamma[x : t] \vdash e \longrightarrow_e e' : t_e}{\text{Some("fun } \underline{x} : \underline{t'} \rightarrow \underline{e'} : \underline{t_e}\text{")}} \quad (1)$$

Other syntax translation are similar to this example. You just need to find the corresponding syntax in OCaml and translate the string. However, there's a problem that the string translations will appear in both pattern translation and expression translation. We create a notation to call these rules and save space (or the page is not enough to explicitly write every string).

Notation: $\rightsquigarrow$ translates a Hazel syntax expression to an option string following the

syntax translation rules. $\Sigma, \Gamma \vdash c \rightsquigarrow s$ means hazel code c can be translated to a option string s if the translation of all variables inside c can be translated by rules whose premises can be derived from Σ, Γ.

Using the notation, the syntax translation rule in the example 1 can be directly rewrite to

$$\text{Hazel: "}\lambda \underline{x} : \underline{t}.\underline{e}\text{"} \rightsquigarrow \text{Some("fun } \underline{x'} : \underline{t'} \to \underline{e'} : \underline{t_e}\text{")}$$

, given that the derivation of x', e', t' can be found in premises.

Note that the translation will also combine the sub-terms. For example, assume Hazel: $e_1 \rightsquigarrow$ Some(s_1) and Hazel: $e_2 \rightsquigarrow$ Some(s_2), then Hazel: $e_1 * e_2 \rightsquigarrow$ Some(s_1 ++ "$*$" ++ s_2).

3.3 Type Check

Type check happens in the situation where we fill a expression with type t_1 into where expects a type t_2. We type check t_1 and t_2 to get a new type t : `pass_t`, meaning the inferred type in this position from the information in both t_1 and t_2.

Notation: $t = t_1 \sqcap_{\text{tc}} t_2$ means that t is the result type of type check between t_1 and t_2.

Since t_1 and t_2 might contains different information, so we can say whether we're matching t_1 to t_2 or matching t_2 to t_1. In our design, we view t_1 and t_2 in equivalent position, hence our type check rules are symmetric.

Here are three conventions to avoid writing redundant rules before we show the inference rules. Firstly, if we have `failwith`, then the program is actually stopped right there. However in the rules we still assign a result (though meaningless) to make the rules complete. Secondly, since all rules are symmetric, for each rule presented, **the symmetric case (if exists) is also defined before the next rule**. Finally, all types shown below are all `pass_t` except for explicit declaration.

Notation: "_" means all possible cases except for previously appeared ones, i.e. $\_ = \{t | \text{all possible } x : \texttt{pass\_t} \text{ doesn't appeared in other rules}\}$.

Rule Definition: For $t = t_1 \sqcap_{\text{tc}} t_2$:

1. t_1 or t_2 is `HOLE`.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{HOLE} \quad \Sigma, \Gamma \vdash t_2 = \_}{\Sigma, \Gamma \vdash t = \texttt{HOLE}, \text{ failwith: incomplete holes}}$$

   `HOLE` is the top rule because incomplete hole is the most serious problem.

2. `HOLE` as an argument of constructor.

$$\frac{\Sigma, \Gamma \vdash t = \texttt{HOLE}}{\Sigma, \Gamma \vdash \texttt{List}\,(t) = \texttt{HOLE}, \text{ failwith: incomplete holes}}$$

$$\frac{\Sigma, \Gamma \vdash t = \texttt{HOLE}}{\Sigma, \Gamma \vdash \texttt{ARROW}, \texttt{PROD}, \texttt{SUM}\,(t, \_) = \texttt{HOLE}, \text{ failwith: incomplete holes}}$$

3. t_1 or t_2 is `CONFLICT`.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{CONFLICT} \quad \Sigma, \Gamma \vdash t_2 = \_}{\Sigma, \Gamma \vdash t = \texttt{CONFLICT}, \ \text{failwith: type conflict}}$$

If there's type conflict, we stop further translating because the work is meaningless.

4. `CONFLICT` as an argument of constructor.

$$\frac{\Sigma, \Gamma \vdash t = \texttt{CONFLICT}}{\Sigma, \Gamma \vdash \texttt{List}\,(t) = \texttt{CONFLICT}, \ \text{failwith: type conflict}}$$

$$\frac{\Sigma, \Gamma \vdash t = \texttt{CONFLICT}}{\Sigma, \Gamma \vdash \texttt{ARROW,PROD,SUM}\,(t, \_) = \texttt{CONFLICT}, \ \text{failwith: type conflict}}$$

5. t_1 or t_2 is `EMPTY`.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{EMPTY} \quad \Sigma, \Gamma \vdash t_2 = \_}{\Sigma, \Gamma \vdash t = \texttt{CONFLICT}}$$

If we call a undeclared variable, it may only harm local structure, so we still want to translate. For example, if we call an undeclared variable in a lambda expression and never use the lambda expression, then the code is only broken locally. Hence we assert an error if the expression containing undeclared variable is called instead of directly assert an error when finding the problem.

6. `Number` case.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{Number} \quad \Sigma, \Gamma \vdash t_2 = \texttt{Number}}{\Sigma, \Gamma \vdash t = \texttt{Number}}$$

7. `Bool` case.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{Bool} \quad \Sigma, \Gamma \vdash t_2 = \texttt{Bool}}{\Sigma, \Gamma \vdash t = \texttt{Bool}}$$

8. `List()` case.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{List}\,(t_a) \quad \Sigma, \Gamma \vdash t_2 = \texttt{List}\,(t_b) \quad \Sigma, \Gamma \vdash t_r = t_a \sqcap_{\text{tc}} t_b}{\Sigma, \Gamma \vdash t = \texttt{List}\,(t_r)}$$

Only list type can be matched with list type.

9. `ARROW(., .),SUM(., .),PROD(., .)` cases.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{ARROW}\,(t_{1a}, \ t_{1b}), t_2 = \texttt{ARROW}\,(t_{2a}, \ t_{2b}) \quad \Sigma, \Gamma \vdash t_a = t_{1a} \sqcap_{\text{tc}} t_{2a}, t_b = t_{1b} \sqcap_{\text{tc}} t_{2b}}{\Sigma, \Gamma \vdash t = \texttt{ARROW}\,(t_a, \ t_b)}$$

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{SUM}\,(t_{1a},\ t_{1b})\,, t_2 = \texttt{SUM}\,(t_{2a},\ t_{2b}) \quad \Sigma, \Gamma \vdash t_a = t_{1a} \sqcap_{\text{tc}} t_{2a}, t_b = t_{1b} \sqcap_{\text{tc}} t_{2b}}{\Sigma, \Gamma \vdash t = \texttt{SUM}\,(t_a,\ t_b)}$$

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{PROD}\,(t_{1a},\ t_{1b})\,, t_2 = \texttt{PROD}\,(t_{2a},\ t_{2b}) \quad \Sigma, \Gamma \vdash t_a = t_{1a} \sqcap_{\text{tc}} t_{2a}, t_b = t_{1b} \sqcap_{\text{tc}} t_{2b}}{\Sigma, \Gamma \vdash t = \texttt{PROD}\,(t_a,\ t_b)}$$

Similarly, we have to ensure the constructor is same.

10. t_1 or t_2 is `UNK`.

$$\frac{\Sigma, \Gamma \vdash t_1 = \texttt{UNK} \quad \Sigma, \Gamma \vdash t_2 = \_}{\Sigma[t_1 := t_2], \Gamma \vdash t = t_2}$$

We meanwhile re-assign the type variable t_1 to the value of t_2. Since no variable is changed, so we don't modify Γ.

Whenever the `UNK` (placeholder) encounters another type, then it should be transformed to other types. Note that we put it last because we don't want `UNK` to be inferred as dependency types such as `List(t)`. If both t_1 and t_2 are `UNK`, the rule also works.

11. Other cases.

$$\frac{\text{Other Cases}}{\Sigma, \Gamma \vdash t = \texttt{CONFLICT},\ \text{failwith: type conflict}}$$

For example, we can't match a `Number` to where needs `ARROW(Number, Number)`.

You can find the implementation of these rules in the code `Extraction_tool.pass_check`. Some implementation is different from the formalized definition here, which is a problem we'll going to fix soon.

3.4 Pattern Translation

Patterns in Hazel are the terms to fill in a hold. In other word, it's the basic element in the program including variable "x", integer "1" and boolean expression "$true$".

We have shown in the syntax translation section 3.2 that how we translate a Hazel code string into OCaml code string. However, the Hazel programs are not directly translated to OCaml, and they're stored in an intermediate structure called `extract_t`. (The syntax translation rules are still applied, you will see them in expression translation section 3.5.) We give the formal definition of `extract_t` as:

$$\texttt{extract\_t} ::= \langle \text{option(string)}, \texttt{pass\_t} \rangle$$

where the first element of the pair is a option string translated by syntax translation rules or pattern translation rules discussing here, and the second element is the type of this term. A `extract_t` can stores a expression like in syntax translation rule, or only save a pattern such as a variable "x".

As an example to help understanding, in the example in syntax translation 3.2, you can try to expand the rule to a expression translation rule by using `extract_t` instead of only string. We will show these expression translation rules in next section 3.5, here please don't think deeply into it.

$$\frac{\text{Hazel:"}\lambda \underline{x} : \underline{t}.\underline{e}\text{"} \quad \Sigma, \Gamma \vdash t \longrightarrow_t t' \quad \Sigma[t := t'_x], \Gamma[x : t] \vdash e \longrightarrow_e e' : t_e}{\langle \text{Some("fun } \underline{x} : \underline{t'} \rightarrow \underline{e'} : \underline{t_e}\text{")}, \texttt{ ARROW(}t'\texttt{, } t_e\texttt{)} \rangle}$$

Let's back to the pattern translation rules.

Rule Definition: For pattern p in Hazel to translate to p' of `extract_t`

1. Empty Hole:

$$\frac{p = \text{Hazel: Empty Hole}}{\Sigma, \Gamma \vdash p' = \langle \text{None}, \texttt{ HOLE} \rangle}$$

2. Integer:

$$\frac{p = \text{Hazel: n} \quad \Sigma, \Gamma \vdash p \rightsquigarrow s}{\Sigma, \Gamma \vdash p' = \langle s, \texttt{ Number} \rangle}$$

3. Boolean:

$$\frac{p = \text{Hazel: true, false} \quad \Sigma, \Gamma \vdash p \rightsquigarrow s}{\Sigma, \Gamma \vdash p' = \langle s, \texttt{ Bool} \rangle}$$

4. List nil:

$$\frac{p = \text{Hazel: []} \quad \Sigma, \Gamma \vdash p \rightsquigarrow s}{\Sigma, \Gamma \vdash p' = \langle s, \texttt{ List(UNK)} \rangle}$$

5. Wild:

$$\frac{p = \text{Hazel: \_} \quad \Sigma, \Gamma \vdash p \rightsquigarrow s}{\Sigma, \Gamma \vdash p' = \langle s, \texttt{ UNK} \rangle}$$

6. Variable:

   If variable can be found in the environment,

$$\frac{p = \text{Hazel: x} \quad \Sigma, \Gamma \vdash p \rightsquigarrow s \quad \Sigma, \Gamma \vdash x : t}{\Sigma, \Gamma \vdash p' = \langle s, t \rangle}$$

If can't find the declaration,

$$\frac{p = \text{Hazel: x} \quad x \notin \Sigma}{\Sigma, \Gamma \vdash p' = \langle \text{None}, \ \texttt{EMPTY} \rangle}$$

7. Parenthesize:

$$\frac{p = \text{Hazel: } (p_1) \quad \Sigma, \Gamma \vdash p \rightsquigarrow s \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : t}{\Sigma, \Gamma \vdash p' = \langle s, \ t \rangle}$$

8. Injection:

   Note that in OCaml we don't explicitly have injection, so the string can simply store subterm.

$$\frac{p = \text{Hazel: inj[L]}(p_1) \quad \Sigma, \Gamma \vdash p_1 \rightsquigarrow s \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : \texttt{SUM}\,(t_l, \ t_r)}{\Sigma, \Gamma \vdash p' = \langle s, \ t_l \rangle}$$

$$\frac{p = \text{Hazel: inj[R]}(p_1) \quad \Sigma, \Gamma \vdash p_1 \rightsquigarrow s \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : \texttt{SUM}\,(t_l, \ t_r)}{\Sigma, \Gamma \vdash p' = \langle s, \ t_r \rangle}$$

$$\frac{p = \text{Hazel: inj[side]}(p_1) \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : \text{otherwise}}{\Sigma, \Gamma \vdash p' = \langle \text{None}, \ \texttt{CONFLICT} \rangle}$$

9. Comma operation:

$$\frac{p = \text{Hazel: } p_1, p_2 \quad \Sigma, \Gamma \vdash p \rightsquigarrow s \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : t_1, \ p_2 \longrightarrow_p p_2' : t_2}{\Sigma, \Gamma \vdash p' = \langle s, \ \texttt{PROD}\,(t_1, \ t_2) \rangle}$$

10. Space operation (apply):

$$\frac{p = \text{Hazel: } p_1 \ p_2 \quad \Sigma, \Gamma \vdash p \rightsquigarrow s \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : \texttt{ARROW}\,(t_2, \ t_1), \ p_2 \longrightarrow_p p_2' : t_2}{\Sigma, \Gamma \vdash p' = \langle s, \ t_1 \rangle}$$

$$\frac{p = \text{Hazel: } p_1 \ p_2 \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : \text{otherwise}}{\Sigma, \Gamma \vdash p' = \langle \text{None}, \ \texttt{CONFLICT} \rangle}$$

11. Cons operation (list):

$$\frac{p = \text{Hazel: } p_1 :: p_2 \quad \Sigma, \Gamma \vdash p \rightsquigarrow s \quad \Sigma, \Gamma \vdash p_1 \longrightarrow_p p_1' : t_1, \ p_2 \longrightarrow_p p_2' : \texttt{List}\,(t_2)}{\Sigma, \Gamma \vdash p' = \langle s, \ \texttt{List}\,(t_1 \sqcap_{\text{tc}} t_2) \rangle}$$

$$\frac{p = \text{Hazel: } p_1 :: p_2 \quad \Sigma, \Gamma \vdash \ p_2 \longrightarrow_p p_2' : \text{otherwise}}{\Sigma, \Gamma \vdash p' = \langle \text{None}, \ \texttt{CONFLICT} \rangle}$$

Since either t_1 or t_2 might be `UNK`, so we use $\sqcap_{\text{tc}}$ to combine them.

Remark that there are many subterms such as p_1 in some cases, the subterms are also patterns but not expressions. The expression conditions will be introduced later. Also remark that the type t in the rules are `pass_t` without specific notation.

3.5 Inference Rules and Expression Translation

Once we finished the type translation and pattern translation, we can start to work on the rules of type inference dealing with syntax translation.

It might confuse you that the difference of pattern and expression since similar subcases also included in expression such as boolean and number. Expression in definition is a closed term, and can appear independently. Patterns are items inside an expression, so they should only appear inside a expression. For example in `let x=1 in 2`, `x` and `1` are patterns, and `2` is an expression.

Rule Definition: For expression e in Hazel to translate to e' of type `extract_t` (the first several rules are similar to pattern ones):

1. Empty Hole:

$$\frac{e = \text{Hazel: Empty Hole}}{\Sigma, \Gamma \vdash e' = \langle \text{None}, \texttt{HOLE} \rangle}$$

2. Integer:

$$\frac{e = \text{Hazel: n} \quad \Sigma, \Gamma \vdash e \rightsquigarrow s}{\Sigma, \Gamma \vdash e' = \langle s, \texttt{Number} \rangle}$$

3. Boolean:

$$\frac{e = \text{Hazel: true, false} \quad \Sigma, \Gamma \vdash e \rightsquigarrow s}{\Sigma, \Gamma \vdash e' = \langle s, \texttt{Bool} \rangle}$$

4. List nil:

$$\frac{e = \text{Hazel: []} \quad \Sigma, \Gamma \vdash e \rightsquigarrow s}{\Sigma, \Gamma \vdash e' = \langle s, \texttt{List(UNK)} \rangle}$$

5. Variable:

   If variable can be found in the environment,

$$\frac{e = \text{Hazel: x} \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash x : t}{\Sigma, \Gamma \vdash e' = \langle s, t \rangle}$$

   If can't find the declaration,

$$\frac{e = \text{Hazel: x} \quad x \notin \Sigma}{\Sigma, \Gamma \vdash e' = \langle \text{None}, \texttt{EMPTY} \rangle}$$

6. Injection:

$$\frac{e = \text{Hazel: inj[L]}(e_1) \quad \Sigma, \Gamma \vdash e_1 \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : \texttt{SUM}(t_l, \ t_r)}{\Sigma, \Gamma \vdash e' = \langle s, \ t_l \rangle}$$

$$\frac{e = \text{Hazel: inj[R]}(e_1) \quad \Sigma, \Gamma \vdash e_1 \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : \texttt{SUM}(t_l, \ t_r)}{\Sigma, \Gamma \vdash e' = \langle s, \ t_r \rangle}$$

$$\frac{e = \text{Hazel: inj[side]}(e_1) \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : \text{otherwise}}{\Sigma, \Gamma \vdash e' = \langle \text{None}, \ \texttt{CONFLICT} \rangle}$$

7. Parenthesize:

$$\frac{e = \text{Hazel: } (e_1) \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : t}{\Sigma, \Gamma \vdash e' = \langle s, \ t \rangle}$$

8. Case:

   Since the case selection is a very long item, we assume that it's in such a format in Hazel,

$$\begin{aligned} &\text{case } x \\ &\quad p_1 \Rightarrow e_1 \\ &\quad p_2 \Rightarrow e_2 \\ &\quad \dots \\ &\text{end } : \boxed{?} \end{aligned}$$

   We further move a series of premises here to save space, notated as P_{case}:

$$\begin{aligned} &\Sigma, \Gamma \vdash p_1, p_2 \cdots \longrightarrow_p p_1' : t_{p_1}, p_2' : t_{p_2} \dots \\ &\Sigma, \Gamma \vdash e_1, e_2 \cdots \longrightarrow_e e_1' : t_{e_1}, e_2' : t_{e_2} \dots \\ &\Sigma, \Gamma \vdash x \rightarrow_p x' : t_x \\ &\Sigma, \Gamma \vdash e \rightsquigarrow s \end{aligned}$$

   There are two cases, the first case is $\boxed{?}$ is assigned with t_c. We apply type check to all case rules, and ensures it is the same as the type provided. The "=" operation here means "assert equal".

$$\frac{P_{\text{case}} \quad \Sigma, \Gamma \vdash t_c \longrightarrow_t t_c' \quad t_{e_1} \sqcap_{\text{tc}} t_{e_2} \sqcap_{\text{tc}} \dots \sqcap_{\text{tc}} t_c' = t_c' \quad t_{p_1} \sqcap_{\text{tc}} t_{p_2} \sqcap_{\text{tc}} \dots \sqcap_{\text{tc}} t_x = t_x}{\Sigma, \Gamma \vdash e' = \langle s, \ t_c' \rangle}$$

$$\frac{P_{\text{case}} \quad \Sigma, \Gamma \vdash t_c \longrightarrow_t t_c' \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None}, \ \texttt{CONFLICT} \rangle}$$

If $\boxed{?}$ is left empty. We need to infer the given type, hence we infer the type from all case rules. We still need to check the patterns.

$$\frac{P_{\text{case}} \quad t_{p_1} \sqcap_{\text{tc}} t_{p_2} \sqcap_{\text{tc}} \ldots \sqcap_{\text{tc}} t_x = t_x}{\Sigma, \Gamma \vdash e' = \langle s,\ t_{e_1} \sqcap_{\text{tc}} t_{e_2} \sqcap_{\text{tc}} \ldots \rangle}$$

$$\frac{P_{\text{case}} \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None},\ \texttt{CONFLICT} \rangle}$$

9. Lambda expression: Assuming P_λ as $e =$ Hazel: $\lambda x : t_x.e_1 \quad \Sigma, \Gamma \vdash e \rightsquigarrow s$ in the premise

   If the lambda expression explicitly show the type,

   $$\frac{P_\lambda \quad \Sigma, \Gamma \vdash t_x \longrightarrow_t t'_x \quad \Sigma[t := t'_x], \Gamma[x : t] \vdash e_1 \longrightarrow_e e'_1 : t_e}{\Sigma, \Gamma \vdash e' = \langle s,\ \texttt{ARROW(}t'_x\texttt{,}\ \ t_e\texttt{)} \rangle}$$

   If the lambda expression doesn't show the type, we need to infer the type of x by our best effort. It means we first evaluate e, and since `UNK` will be modified in the environment, hence we can read the type of x inferred from the environment after the evaluation of e.

   $$\frac{P_\lambda \quad \Sigma[t_x := \texttt{UNK}], \Gamma[x : t_x] \vdash e_1 \longrightarrow_e e'_1 : t_e \text{ with } x : t'_x}{\Sigma, \Gamma \vdash e' = \langle s,\ \texttt{ARROW(}t'_x\texttt{,}\ \ t_e\texttt{)} \rangle}$$

10. Let expression:

    The "let" expression in hazel is another way to introduce a variable like lambda expression. We can similarly define the rules like lambda expression.

    In first case, assume the premise contains P_{let} as $e =$ Hazel: let $x = e_1$ in $e_2 \quad \Sigma, \Gamma \vdash e \rightsquigarrow s$.

    $$\frac{P_{\text{let}} \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e'_1 : t_{e_1} \quad \Sigma[t_x := t_{e_1}], \Gamma[x : t_x] \vdash e_2 \longrightarrow_e e'_2 : t_{e_2}}{\Sigma, \Gamma \vdash e' = \langle s,\ t_{e_2} \rangle}$$

    In the other case, we have explicitly declare the type of x. P_{let2} as $e =$ Hazel: let $x : t = e_1$ in $e_2 \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad t \longrightarrow_t t'$.

    $$\frac{P_{\text{let2}} \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e'_1 : t_{e_1} \quad t_{e_1} \sqcap_{\text{tc}} t' = t' \quad \Sigma[t_x := t'], \Gamma[x : t_x] \vdash e_2 \longrightarrow_e e'_2 : t_{e_2}}{\Sigma, \Gamma \vdash e' = \langle s,\ t_{e_2} \rangle}$$

    $$\frac{P_{\text{let2}} \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e'_1 : t_{e_1} \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None},\ \texttt{CONFLICT} \rangle}$$

    Remark: In the real structure of Hazel, "Let" expression is an individual "line" as `let x=e in` . The expression actually contains multiple lines, but it is a reasonable

change to this more intuitive and simple structure in formal definition.

11. Space (apply):

    P_{apply}: $e = \text{Hazel: } e_1\ e_2 \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : t_1 \quad \Sigma, \Gamma \vdash e_2 \longrightarrow_e e_2' : t_2$

    The argument should have a concrete type, so it can't be `UNK`.

$$\frac{P_{\text{apply}} \quad \Sigma, \Gamma \vdash t_1 = \texttt{ARROW(}t_{1a}\texttt{,}\ \ t\texttt{)} \quad \Sigma, \Gamma \vdash t_{1a} \sqcap_{\text{tc}} t_2 = t_2}{\Sigma, \Gamma \vdash e' = \langle s,\ t \rangle}$$

$$\frac{P_{\text{apply}} \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None},\ \texttt{CONFLICT} \rangle}$$

12. Cons (list):

    P_{list}: $e = \text{Hazel: } e_1 :: e_2 \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : t_1 \quad \Sigma, \Gamma \vdash e_2 \longrightarrow_e e_2' : t_2$

    The new element of list should have a concrete type, so it can't be `UNK`.

$$\frac{P_{\text{list}} \quad \Sigma, \Gamma \vdash t_2 = \texttt{List(}t\texttt{)} \quad \Sigma, \Gamma \vdash t \sqcap_{\text{tc}} t_1 = t_1}{\Sigma, \Gamma \vdash e' = \langle s,\ \texttt{List(}t_1\texttt{)} \rangle}$$

$$\frac{P_{\text{list}} \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None},\ \texttt{CONFLICT} \rangle}$$

13. Plus, Minus, Times:

    P_{arth}: $e = \text{Hazel: } e_1 \oplus e_2 \text{ where } \oplus = \{+, -, *\} \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : t_1 \quad \Sigma, \Gamma \vdash e_2 \longrightarrow_e e_2' : t_2$

$$\frac{P_{\text{arth}} \quad \Sigma, \Gamma \vdash t_1 \sqcap_{\text{tc}} t_2 \sqcap_{\text{tc}} \texttt{Number} = \texttt{Number}}{\Sigma, \Gamma \vdash e' = \langle s,\ \texttt{Number} \rangle}$$

$$\frac{P_{\text{arth}} \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None}, \texttt{CONFLICT} \rangle}$$

14. Less than, Greater than, Equals:

    P_{comp}: $e = \text{Hazel: } e_1 \odot e_2 \text{ where } \odot = \{<, >, =\} \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : t_1 \quad \Sigma, \Gamma \vdash e_2 \longrightarrow_e e_2' : t_2$

$$\frac{P_{\text{comp}} \quad \Sigma, \Gamma \vdash t_1 \sqcap_{\text{tc}} t_2 \sqcap_{\text{tc}} \texttt{Number} = \texttt{Number}}{\Sigma, \Gamma \vdash e' = \langle s,\ \texttt{Bool} \rangle}$$

$$\frac{P_{\text{comp}} \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None}, \texttt{CONFLICT} \rangle}$$

15. And, Or:

    P_{bool}: $e = \text{Hazel: } e_1 \oplus e_2 \text{ where } \oplus = \{\&\&, ||\} \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1 \longrightarrow_e e_1' : t_1 \quad \Sigma, \Gamma \vdash e_2 \longrightarrow_e e_2' : t_2$

$$\frac{P_{\text{bool}} \quad \Sigma, \Gamma \vdash t_1 \sqcap_{\text{tc}} t_2 \sqcap_{\text{tc}} \texttt{Bool} = \texttt{Bool}}{\Sigma, \Gamma \vdash e' = \langle s,\ \texttt{Bool} \rangle}$$

$$\frac{P_{\text{bool}} \quad \text{otherwise not assert equal}}{\Sigma, \Gamma \vdash e' = \langle \text{None}, \texttt{CONFLICT} \rangle}$$

16. Comma:

$$\frac{e = \text{Hazel: } e_1, e_2 \quad \Sigma, \Gamma \vdash e \rightsquigarrow s \quad \Sigma, \Gamma \vdash e_1, e_2 \longrightarrow_e e_1' : t_1, e_2' : t_2}{\Sigma, \Gamma \vdash e' = \langle s,\ \texttt{PROD(}t_1\texttt{,}\ \ t_2\texttt{)} \rangle}$$

Hereby we finish all the cases and all terms in Hazel.

Important Remark: The sum type in hazel of expression should be achieved by case, and Hazel allows the sub-expressions in rules having different types, hence the result is a sum type. However, OCaml enforces all sub-expressions in pattern matching to have a same type. For this reason, we can't do the translation and assert an error.

3.6 To OCaml

After we execute our program of extraction and the program doesn't throw an error, then we'll have a `extract_t` type result. Then the first element of the result (option string) records all the code. We just need to de-option the string and print.

4 Existing Problems and Future Work

Just as Hazel, our extraction is still improving, and there are many problems in the current version. Here lists some known bugs in current version.

1. In the lambda expression, $\lambda\boxed{x}\texttt{:}\boxed{?}\texttt{.}\{\texttt{e}\}$, if you leave the $\boxed{?}$ hole blank, the inference may fail and assign a `'a` type (arbitrary type) in OCaml code. It won't cause the OCaml program fail to run, but is should be inferred in our extraction.

2. The error message might be wrong for some cases. For example an incomplete hole in very deep level will, we might assert "Type Conflict" instead of "Incomplete Holes".

3. If the code involves application of multiple arguments, then the code will be extracted as expected, but we'll have a error message.

```
let f =
  λx:Num.{
    λy:Bool.{
      case y
      | true ⇒ x + 1
      | false ⇒ x + 2
      end : Num
    }
  }
in
let x = 1 in
f x true
```

Figure 5: Bug Example

```
let (f:int->bool->int) = (fun (x:int) -> (fun (y:bool) ->
((match (y:bool)
 with
 | true -> (x:int)+1

 | false -> (x:int)+2

) : int)
)
)
 in
let (x:int) = 1
 in
(f:int->bool->int) (x:int) true

Conflict Here
```

Figure 6: Extraction Result of Bug

You can see that the code is correctly translated, but in the last line we have a "Conflict Here" message. It means the extraction finds a type conflict, but there should be no. We think it's because the application code is moved from the old version, so the type check for application may do extra compares and thus get an error.

We guess these known bugs result from the migration from old version. As mentioned, the AST structure of Hazel a big update so I had to rewrite some functions. The type checking functions are still from old version and not fully tested on the new structure, so there might have potential issues. Fixing these things are our current focus, so please wait for update.

Besides bugs, there are something we want to improve in the future.

1. Introduce a `STR` constructor for strings and `ERR` constructor for error messages. It's formally correct but is really awful to use `UNK` for strings.
2. Introduce a auto-formater or indent control functions to make the code looks nicer.
3. Assert error messages indicating the location where we detect the error.

5 Conclusion

In this project, we formally design and implement the extraction from Hazel to OCaml. We managed to design the formal rules for most of the program. Unluckily, it's still a prototype and there are many known and unknown issues in the current version. We're keeping working on it to make it better.

References

Omar, C., Voysey, I., Chugh, R., & Hammer, M. A. (2019). Live functional programming with typed holes. *PACMPL*, *3*(POPL). Retrieved from `http://doi.acm.org/10.1145/3290327` doi: 10.1145/3290327