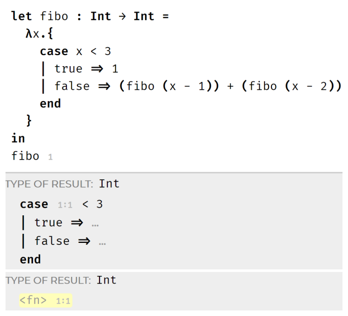
Interactive Stepper for Expression with Holes

YANJUN CHEN[*], University of Michigan, USA

1 INTRODUCTION

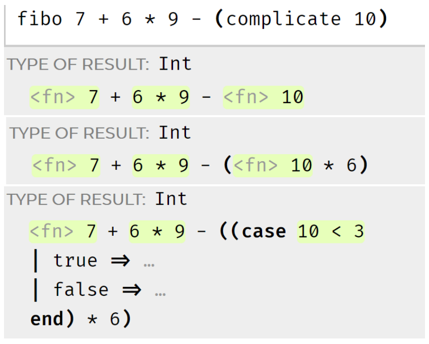
Stepper is a programming tool that can display all immediate states of a program, which is useful for debugging and educating. It simplifies expression step by step so that people can track the evaluation procedure to find error easily. Hazel, a pure functional programming language environment developed by Omar et al. [2017], assigns semantic meaning to incomplete program. It contains a special variable called hole indicating the place that is empty and needs to fill. In this paper, we want to develop an interactive stepper for Hazel. Take the following programs as examples:



```
let fibo : Int → Int =
  λx.{
    case x < 3
    | true ⇒ 1
    | false ⇒ (fibo (x - 1)) + (fibo (x - 2))
    end
  }
in
fibo 1
```

```
TYPE OF RESULT: Int
 case 1:1 < 3
 | true ⇒ …
 | false ⇒ …
 end
TYPE OF RESULT: Int
 <fn> 1:1
```

(a) Example of paused judgement



```
fibo 7 + 6 * 9 - (complicate 10)
TYPE OF RESULT: Int
 <fn> 7 + 6 * 9 - <fn> 10
TYPE OF RESULT: Int
 <fn> 7 + 6 * 9 - (<fn> 10 * 6)
TYPE OF RESULT: Int
 <fn> 7 + 6 * 9 - ((case 10 < 3
 | true ⇒ …
 | false ⇒ …
 end) * 6)
```

(b) Example of multiple environment

Fig. 1. Two programs in Hazel

Figure 1a is an example of expression with a hole. The evaluation result is a case expression with a hole as parameter. However, the origin expression is already simple enough and it is unnecessary to expand like that especially when there are many cases in the function. Therefore, we want to develop a stepper that can detect this situation so that it will pause until user requires it to step further. The result with yellow box is calculated by our stepper with pause judgement.

Figure 1b shows an expression with three parts. The left part is heavy since it takes many steps to evaluate. However, user might just want to debug the right complicated part. According to the regular dynamics, there should be only one way to step for any non-final programs. So, user has to click many times to reduce the left one in order to debug the right one. Our solution is to provide options for user to choose where to step. In the Figure 1b, the green boxes contain subexpressions that can be evaluated. In these three results, we always click the right box to evaluate. We can see how the multiple contexts work.

Our interactive stepper develops pausing judgement and a simple algorithm to decompose multiple evaluation contexts. Furthermore, it also has a webview so that user can click boxes to step expressions.

Contributions. The contributions of this paper are: (1) a pausing judgement in section 2; (2) an algorithm to decompose multiple evaluation contexts in section 3.

[*]Research advisor: Cyrus Omar; Category: undergraduate

Author's address: Yanjun Chen, University of Michigan, USA, yanjunc@umich.edu.

2 PAUSING ENVIRONMENT

We first formally define the simplified syntax of Hazel shown in Figure 2 [Omar et al. 2019]. The symbol $\llparenthesis\rrparenthesis$ represents the empty hole used to indicate the place that need to fill in the incomplete program. And for the expression that doesn't have clear meaning, we use $\llparenthesis e \rrparenthesis$ to indicate that it doesn't have semantic meanings. We only consider binary sums so for $\mathtt{inj}_i(e)$, $i \in \{L, R\}$.

$$
\begin{array}{lcl}
\mathit{HTyp}\ \tau & ::= & \mathsf{num} \mid \tau \rightarrow \tau \mid (\tau + \tau) \mid \llparenthesis\rrparenthesis^{A} \\
\mathit{HExp}\ e & ::= & x \mid (\lambda x.e) \mid (\lambda x : \tau.e) \mid e(e) \mid \underline{n} \mid (e + e) \mid e : \tau \mid \mathtt{inj}_i(e) \mid \mathsf{case}(e, x.e, y.e) \mid \llparenthesis\rrparenthesis \mid \llparenthesis e \rrparenthesis
\end{array}
$$

Fig. 2. Syntax of H-types, H-expressions

$\boxed{e\ \mathsf{paused}}$ e is paused

$$
\texttt{PCaseHole}\ \frac{}{(\lambda z.\mathsf{case}(z, x.e_1, y.e_2))(\llparenthesis\rrparenthesis)\ \mathsf{paused}}
\qquad
\texttt{PCase}\ \frac{e_0\ \mathsf{paused}}{\mathsf{case}(e_0, x.e_1, y.e_2)\ \mathsf{paused}}
$$

$$
\texttt{PAp1}\ \frac{e_1\ \mathsf{paused}}{e_1(e_2)\ \mathsf{paused}}
\qquad
\texttt{PAp2}\ \frac{e_2\ \mathsf{paused}}{e_1(e_2)\ \mathsf{paused}}
$$

$$
\texttt{PAdd1}\ \frac{e_1\ \mathsf{paused}}{e_1 + e_2\ \mathsf{paused}}
\qquad
\texttt{PAdd2}\ \frac{e_2\ \mathsf{paused}}{e_1 + e_2\ \mathsf{paused}}
\qquad
\texttt{PHole}\ \frac{e\ \mathsf{paused}}{\llparenthesis e \rrparenthesis\ \mathsf{paused}}
$$

Fig. 3. Paused forms

Normally, we have judgement like `final`, `boxedvalue` for a program. Furthermore, since Hazel contains incomplete programs, there exists some indeterminate programs which induces a judgement called `indet` [Omar et al. 2017]. Pausing judgement is noted as paused. For example, when $\llparenthesis\rrparenthesis$ appears at case expression so that the match algorithm gives indeterminate result, it will be a paused expression so that the stepper can stop immediately. Figure 1a gives a expression as example. It defines a function func. But we don't provide any argument for it. The evaluator expand the case expression directly. However, the result in bottom shows an example that when augment is only a hole, the stepper will not evaluate but stop with yellow box indicating it is steppable but paused.

3 CONTEXTUAL DYNAMICS

The structure of the stepper is shown in Figure 4. The expression is first decomposed into many independent evaluation contexts. Then, user may choose one of them to step. And the stepper will compose them into the final result then. For example, in Figure 1b, we have 3 evaluation contexts highlighted as green boxes. And when we click the second one, only the expression in second box is evaluated and composed into the whole expression.

We now formally define the instruction transition judgement. We use $e_1 \mapsto e_2$ as instruction transition judgement. Figure 5 shows some of the transition judgements.

Then, we formally define the evaluation context. Unlike the regular evaluation context, the decompose function will return a list of contexts. The decompose function is defined recursively. Figure 5 provides some of inference rules for decompose function.

For example, we have an expression $e_0 = 4 + 1 + (5 + 6)$. First, we know that $4 + 1 \implies [\circ\{4 + 1\}]$ and $5 + 6 \implies [\circ\{5 + 6\}]$. So, According to DAdd rule, we have,

$$4 + 1 + (5 + 6) \implies [(\circ + (5 + 6))\{4 + 1\}, (4 + 1 + \circ)\{5 + 6\}].$$

User may choose second one so that the stepper will first evaluate $5 + 6$ and put result into mark. Hence, we get $4 + 1 + 11$ finally.

$$\begin{tikzcd}
\boxed{d_1} \arrow[r, "\text{decompose}"] \arrow[dd, dashed] & \boxed{\epsilon_1\{e_1\}, \epsilon_2\{e_2\}, \ldots} \arrow[d, "\text{select}"] \\
& \boxed{\epsilon_1\{e_1\}} \arrow[d, "\text{instruction transition}"] \\
\boxed{d_1'} & \boxed{\epsilon_1\{e_1'\}} \arrow[l, "\text{compose}"']
\end{tikzcd}$$

Fig. 4. Structure of stepper

$\boxed{e \mapsto e'}$ e takes an instruction transition to e'

$$\frac{e_2 \ \mathtt{final}}{(\lambda x.e_1)(e_2) \mapsto [e_2/x]e_1}$$

$$\text{EvalCtx } \epsilon \quad ::= \quad \circ \mid \epsilon(e) \mid e(\epsilon) \mid (\!|\epsilon|\!) \mid \epsilon + e \mid e + \epsilon$$

$\boxed{e \ \Rightarrow \ [\epsilon_1\{e_1\}, \epsilon_2\{e_2\}, \cdots]}$ e is decomposed into contexts $\epsilon_1\{e_1\}$, $\epsilon_2\{e_2\}$, $\cdots$

DFinal
$$\frac{e \ \mathtt{final}}{e \ \Rightarrow \ []}$$

DApFinal
$$\frac{e_1 \ \mathtt{final} \quad e_2 \ \mathtt{final}}{e_1(e_2) \ \Rightarrow \ [\circ\{e_1(e_2)\}]}$$

DAp
$$\frac{e_1 \ \Rightarrow \ [\epsilon_1\{e_1'\}, \cdots] \qquad e_2 \ \Rightarrow \ [\epsilon_2\{e_2'\}, \cdots]}{e_1(e_2) \ \Rightarrow \ [\epsilon_1(e_2)\{e_1'\}, \cdots, e_1(\epsilon_2)\{e_2'\}, \cdots]}$$

DHole
$$\frac{e \ \Rightarrow \ [\epsilon\{e'\}, \cdots]}{(\!|e|\!) \ \Rightarrow \ [(\!|\epsilon|\!)\{e'\}]}$$

DAdd
$$\frac{e_1 \ \Rightarrow \ [\epsilon_1\{e_1'\}, \cdots] \qquad e_2 \ \Rightarrow \ [\epsilon_2\{e_2'\}, \cdots]}{(e_1 + e_2) \ \Rightarrow \ [(\epsilon_1 + e_2)\{e_1'\}, \cdots, (e_1 + \epsilon_2)\{e_2'\}, \cdots]}$$

Fig. 5. Insturction transition and decomposition

4 CONCLUSION

This paper develops an interactive stepper for Hazel, hence, language environment with holes. It uses the paused judgement to detect the unnecessary situation so that the stepper will stop over the paused expression. Also, we develop a simple algorithm for decomposition that can find all subexpressions which need to evaluate. We use the contextual dynamics to progress the expression for stepper. This result is useful for debugging or educational purpose.

REFERENCES

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. 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 *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.