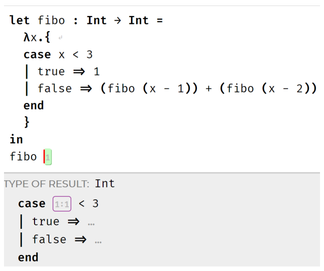
An Interactive Stepper for Expression with Holes

YANJUN CHEN[*], University of Michigan, USA

1 INTRODUCTION

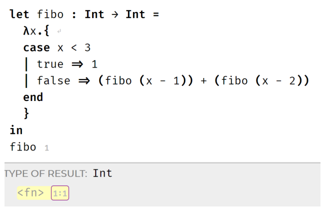
A stepper is a programming tool that can display the intermediate states that arises when evaluating an expression.Steppers promise to ease debugging and learning. FINDLER et al. [2002] developed a stepper with error detection. However, it can only provide feedback for complete programs that don't have missing pieces. Hazel, a pure functional programming language environment developed by Omar et al. [2017], assigns semantic meaning to incomplete programs. It contains a special form called hole indicating missing expressions. 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
```

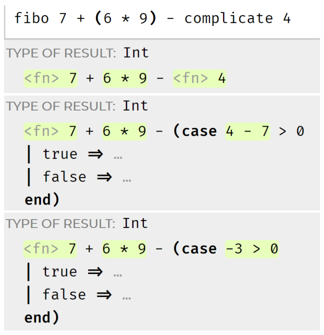
(a) Example of a program with hole as argument



```
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
  <fn> 1:1
```

(b) Example of paused evaluations



```
fibo 7 + (6 * 9) - complicate 4
```

```
TYPE OF RESULT: Int
  <fn> 7 + 6 * 9 - <fn> 4
TYPE OF RESULT: Int
  <fn> 7 + 6 * 9 - (case 4 - 7 > 0
  | true ⇒ …
  | false ⇒ …
  end)
TYPE OF RESULT: Int
  <fn> 7 + 6 * 9 - (case -3 > 0
  | true ⇒ …
  | false ⇒ …
  end)
```

(c) Example of choosing which active expression to evaluate

Fig. 1. Three 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. In Hazel, evaluation is live. So, this result might arise momentarily, just before its size may distract the user. It also reveals internal details about how the function is implemented. 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 shown in Figure 1b with yellow box is calculated by our stepper with pause judgement.

Figure 1c 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 evaluation order for Hazel, 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 1c, 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 is specified by a 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.

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

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

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.

2 PAUSING ENVIRONMENT

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

$$\begin{array}{lll} \mathit{HTyp}\ \tau & ::= & \mathsf{num} \mid \tau \to \tau \mid (\tau + \tau) \mid (\!|\,|\!) \\ \mathit{HExp}\ e & ::= & x \mid (\lambda x.e) \mid (\lambda x : \tau.e) \mid e(e) \mid \underline{n} \mid (e + e) \mid e : \tau \mid \texttt{inj}_i(e) \mid \texttt{case}(e, x.e, y.e) \mid (\!|\,|\!) \mid (\!|e|\!) \end{array}$$

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

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

$$\texttt{PCase}\ \frac{e_0\ \texttt{indet}}{(\lambda z.\texttt{case}(z, x.e_1, y.e_2))(e_0)\ \texttt{paused}} \qquad \texttt{PAp1}\ \frac{e_1\ \texttt{paused}}{e_1(e_2)\ \texttt{paused}} \qquad \texttt{PAp2}\ \frac{e_2\ \texttt{paused}}{e_1(e_2)\ \texttt{paused}} \qquad \cdots$$

Fig. 3. Paused forms

Hazelnut Live, an editor for Hazel, defines judgement like e $\texttt{final}$, e $\texttt{boxedval}$ for expression e [Omar et al. 2019]. Furthermore, since Hazel contains incomplete programs, there exist some indeterminate programs, which induces a judgement denoted as e $\texttt{indet}$.

Pausing judgement is denoted as e $\texttt{paused}$. This paused judgement is designed to simplify the output of the program since Hazel has some complicated programs that can not be evaluated further. Therefore, the paused expression is steppable but this needs user's confirmation. For example, we want to avoid the long cases as output. Therefore, when $(\!|\,|\!)$ appears at case expression so that the match algorithm gives indeterminate result, it will be a paused expression stopping the stepper immediately. The judgement is shown in Figure 3. PCase is the case that the output will be long and useless. Other rules make paused judgement propagate up. Besides, this judgement can also be extended in the future. For example, function with multiple arguments should pause before it goes into beta rule. Otherwise, it is wired to see a partial function when there is only one argument given.

3 CONTEXTUAL DYNAMICS

The structure of our stepper is shown in Figure 4. We use the similar structure developed by Cong and Asai [2016]. 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 final result. For example, in Figure 1c, we have 3 evaluation contexts highlighted as green boxes. And when we click the third one, only the expression in third box is evaluated and then 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 have the formal definition of evaluation context. We show some of rules in Figure 5. For any expression, we define the decompose function $\texttt{decompose}(e)$ which returns all evaluation contexts of e. The formal definition is given below:

$$\texttt{decompose}(e) = \{\epsilon\{e'\} | e = \epsilon\{e'\} \text{ and not } e'\texttt{final}\}.$$

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

Fig. 4. Structure of stepper

Unlike the regular evaluation context, this decompose function decompose will return a list of contexts. In general, we can see that if e $\texttt{final}$, $\texttt{decompose}(e) = \emptyset$. We use addition as an example of non-final expressions. For expression $e_1 + e_2$, if both of them are final, we have $\texttt{decompose}(e_1 + e_2) = \{\circ\{e_1 + e_2\}\}$. Otherwise,

$$\begin{aligned}\texttt{decompose}(e_1 + e_2) = &\{(\epsilon_1 + e_2)\{e_1'\} | \epsilon_1\{e_1'\} \in \texttt{decompose}(e_1)\} \\ &\cup \{(e_1 + \epsilon_2)\{e_2'\} | \epsilon_2\{e_2'\} \in \texttt{decompose}(e_2)\}.\end{aligned}$$

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

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

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

$\boxed{e = \epsilon\{e'\}}$ e is obtained by replacing the mark in ϵ with e'

$$\texttt{FHOuter}\ \frac{}{e = \circ\{e\}} \qquad \texttt{FHAp1}\ \frac{e_1 = \epsilon_1\{e_1'\}}{e_1(e_2) = \epsilon_1(e_2)\{e_1'\}} \qquad \texttt{FHAp2}\ \frac{e_2 = \epsilon_2\{e_2'\}}{e_1(e_2) = e_1(\epsilon_2)\{e_2'\}}$$

Fig. 5. Insturction transition and decomposition

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

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

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.

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 may be useful for debugging or educational purpose. We plan to evaluate these tools in an educational setting in the future.

REFERENCES

Youyou Cong and Kenichi Asai. 2016. Implementing a stepper using delimited continuations. In *7th International Symposium on Symbolic Computation in Software Science, SCSS 2016, Tokyo, Japan, March 28-31, 2016 (EPiC Series in Computing, Vol. 39)*, James H. Davenport and Fadoua Ghourabi (Eds.). EasyChair, 42–54. https://easychair.org/publications/paper/7qlb

ROBERT BRUCE FINDLER, JOHN CLEMENTS, CORMAC FLANAGAN, MATTHEW FLATT, SHRIRAM KRISHNAMURTHI, PAUL STECKLER, and MATTHIAS FELLEISEN. 2002. DrScheme: a programming environment for Scheme. *Journal of Functional Programming* 12, 2 (2002), 159–182. https://doi.org/10.1017/S0956796801004208

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)*.