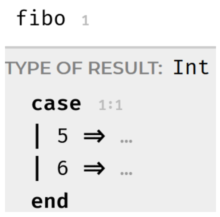
Interactive Stepper for Expression with Holes

YANJUN CHEN*, University of Michigan, USA

1 INTRODUCTION

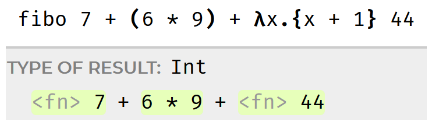
Stepper is a programming tool that can display all immediate states of a program, which is useful for debugging and educating. It simplifies expression step by step so that people can track the evaluation procedure to find error easily. Hazel, a programming language environment developed by [citation Cyrus Omar], 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:



```
fibo 1

TYPE OF RESULT: Int
  case 1:1
  | 5 ⇒ …
  | 6 ⇒ …
  end
```

(a) Example of paused environment



```
fibo 7 + (6 * 9) + λx.{x + 1} 44

TYPE OF RESULT: Int
  <fn> 7 + 6 * 9 + <fn> 44
```

(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 empty 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.

Figure 1b shows an expression with three parts. The left part is heavy since it takes many steps to evaluate while user might just want to debug the right lambda part. According to the regular dynamics, there should contain 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. In the Figure 1b, the green boxes contain subexpressions that can be evaluated. So, our solution is just allowing the multiple evaluation contexts so that user can choose the subexpression they need.

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

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

2 PAUSING ENVIRONMENT

We formally define the syntax for discussion. The formal syntax is defined below in Figure 2. $\llparenthesis\rrparenthesis$ is the empty hole used to fill 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.

Normally, we have judgement like Final, BoxedValue and since Hazel contains incomplete programs, it means there exists some indeterminate programs [citation]. Pausing environment is

*Research advisor: Cyrus Omar; ACM student member number: ; Category: undergraduate

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

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

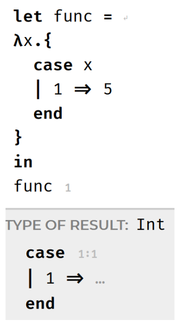
Fig. 2. Syntax of H-types, H-expressions, and Type Constraint Set

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

$$\frac{e_0\ \mathsf{indet}}{\mathsf{case}(e_0, x.e_1, y.e_2)\ \mathsf{paused}}$$

Fig. 3. Paused Forms

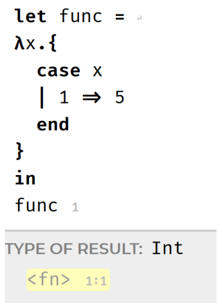
introduced by extra judgement, Paused. For example, when $(\!|\,|\!)$ appears at case expression so that the match algorithm gives indeterminate result, it will be a paused expression so that the stepper can stop immediately. Figure 4a gives a expression for that. Also, Figure 4b shows an example that when augment is only a hole, the stepper will not evaluate but stop with yellow box.



```
let func =
λx.{
  case x
  | 1 ⇒ 5
  end
}
in
func 1
```

```
TYPE OF RESULT: Int
  case 1:1
  | 1 ⇒ …
  end
```

(a)



```
let func =
λx.{
  case x
  | 1 ⇒ 5
  end
}
in
func 1
```

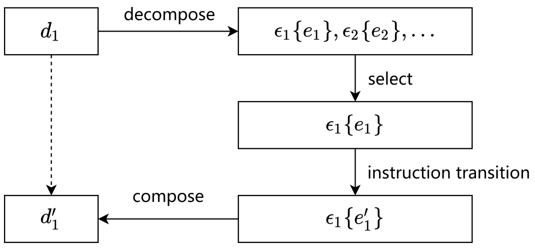
```
TYPE OF RESULT: Int
  <fn> 1:1
```

(b)

Fig. 4. Example of stepper with paused environment

3 CONTEXTUAL DYNAMICS

The structure of the stepper is shown in Figure 5. The expression is first decomposed into many independent evaluation contexts. Then, user may choose one of them to progress. And the stepper will compose them into the 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.



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

Fig. 5. Structure of Stepper

We now formally define the instruction transition judgement. We use $e_1 \mapsto e_2$ as instruction transition judgement. Figure 6 shows some of the transition judgement.

As what Section **??** discussed, in Hazel, we have four types of expressions: boxed value, indeterminate, step, and paused. The instruction transition can be implemented to provide the type of the expressions. Therefore, we can easily know whether an expression is final, paused or steppable.

Then, we have the evaluation context. Unlike the regular evaluation context, the decompose function will return a list of contexts. The recursive definition of the decompose function is shown in Figure 6.

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

$$\frac{e_2\ \texttt{final}}{(\lambda x.d_1)(d_2) \mapsto [d_2/x]d_1}$$

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

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

$$\texttt{DFinal}\ \frac{e\ \texttt{final}}{e \ \Rightarrow\ []} \qquad \texttt{DBoxedVal}\ \frac{e\ \texttt{boxedval}}{e \ \Rightarrow\ [\circ\{e\}]} \qquad \texttt{DApFinal}\ \frac{e_1\ \texttt{final}\ e_2\ \texttt{final}}{e_1(e_2) \ \Rightarrow\ [\circ\{e_1(e_2)\}]}$$

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

$$\texttt{DHoleFinal}\ \frac{e\ \texttt{final}}{\llparenthesis e \rrparenthesis \ \Rightarrow\ [\circ\{\llparenthesis e \rrparenthesis\}]} \qquad \texttt{DAdd}\ \frac{e_1 \ \Rightarrow\ [\epsilon_1\{e_1'\}, \cdots] \qquad e_2 \ \Rightarrow\ [\epsilon_2\{e_2'\}, \cdots]}{(e_1 + e_2) \ \Rightarrow\ [(\epsilon_1 + e_2)\{e_1'\}, \cdots, (e_1 + \epsilon_2)\{e_2'\}, \cdots]}$$

Fig. 6. 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 DAdd rule, 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.

In conclusion, we first decompose the expression into many evaluation contexts for user to choose. Then, we run the instruction transition and compose the result to user.

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.