Running Incomplete Programs

Ian Voysey[1] Cyrus Omar[1] Matthew A. Hammer[2]

[1]Carnegie Mellon University (USA)
{iev, comar}@cs.cmu.edu

[2]University of Colorado Boulder (USA)
matthew.hammer@colorado.edu

Abstract

We typically only consider running programs that are completely written. Programmers end up inserting ad hoc dummy values into their incomplete programs to receive feedback about dynamic behavior. In this work we suggest an evaluation mechanism for incomplete programs, represented as terms with holes. Rather than immediately failing when a hole is encountered, evaluation propagates holes as far as possible. The result is a substantially tighter development loop.

1. Introduction

Programming language definitions assign meaning to *complete* programs. Programmers, however, spend a substantial amount of time interacting with *incomplete* programs, using tools like program editors or live programming environments that interleave editing and evaluation. Semanticists have paid comparatively little attention to these interactions, so the designers of tools like these lack foundational principles comparable to those available to language designers. For example, Agda does provide a notion of incomplete programs and a mechanism to evaluate terms with holes in them, but it fails as soon as it actually encounters a hole (Norell 2007).

In our paper describing the Hazelnut structure editor, appearing at POPL 2017, we consider the static semantics of incomplete programs, i.e. programs with holes in expressions and types (Omar et al. 2017). We also introduce a notion of non-empty hole, which serves to safely encapsulate putative subexpressions whose types may not yet be consistent with the type that is expected. The "beaten track" strategy would be to define a dynamics only for programs without holes, or to have holes act like exceptions and end evaluation. Instead, our "off the beaten track" suggestion is to continue evaluation past a hole, revealing more about the dynamic behavior of the incomplete term. This could benefit programmers by tightening the feedback loop between editing and evaluation, even beyond what is available in current live programming tools (Burckhardt et al. 2013).

The main technical challenge is to recover a meaningful notion of type safety—we seek recognizable statements of progress and preservation for terms with holes in them, with respect to a dynamic semantics that matches our intuition.

In the ideal system, a Hazelnut programmer writes their program with confidence that her edits are all safe. When she arrives at a candidate, possibly incomplete, program that she wants to explore more, she can run it on specific values and observe how the holes in her program are actually used in evaluation. She can then return to editing the program, working towards a complete program with better insight into how the concrete values of variables are used. This insight can help her develop intuition about how to fill each hole in the incomplete program.

2. Examples

In our Hazelnut paper, incomplete expressions and types are given by the grammar:

$$\dot{\tau} ::= \dot{\tau} \to \dot{\tau} \mid \texttt{num} \mid ⦇⦈_u$$

$$\dot{e} ::= x \mid \lambda x.\dot{e} \mid (\dot{e})\dot{e} \mid \underline{n} \mid \dot{e} + \dot{e} \mid (\dot{e} : \dot{\tau}) \mid ⦇⦈_u \mid ⦇\dot{e}⦈_u$$

where $⦇⦈_u$ and $⦇\dot{e}⦈_u$ are holes with unique names u.

We write $\mapsto$ for the small-step relation. The judgement $\dot{e}$ **value** is derivable when a $\dot{e}$ is a value, in the standard sense. For example:

$$(\lambda x.9 + ⦇⦈_u) \textbf{ value}$$

The judgement $\dot{e}$ **indet** is derivable when $\dot{e}$ is *indeterminate*—when its evaluation cannot proceed because of a hole in a critical place (roughly, in elimination position.) For example, the following cannot continue to step, nor is it a value:

$$(9 + ⦇⦈_u) \textbf{ indet}$$

Evaluation continues until the term in question is either a value or indeterminate. Collectively, we call these *final* evaluation states.

A function applied to a hole, or any other final term, steps as usual:

$$(\lambda x.x + ⦇⦈_u)⦇⦈_{u'} \longmapsto ⦇⦈_{u'} + ⦇⦈_u$$

Substitutions occur inside non-empty holes, even if the type of the contents is not yet consistent with its surroundings. Otherwise, free variables would appear in the result, breaking any familiar notion of preservation.

For instance, if f : num → num → num,

$$(\lambda x.x + (\!|f\ x|\!)_u)5 \longmapsto 5 + (\!|f\ 5|\!)_u$$

Extended Hazel To consider a more realistic example, we consider a hypothetical extension of Hazelnut with features such as polymorphism, pattern matching, ADTs, and recursive functions in ML-like syntax.

Imagine a programmer implementing

$$\textbf{map} : (\alpha \to \beta) \to \alpha\ list \to \beta\ list$$

from the informal specification that the function be applied to every list element uniformly. She might arrive at the incomplete term

```
fun map f [] = []
  | map f (x::xs) = (||)u::(map f xs)
```

if she knows the pattern of recursion, but is not sure what to do with each element. Evaluating this incomplete program on a small example produces a trace:

```
    map (λx.x+1) [1, 2]
↦* (||)u::(map (λx.x+1) [2])
↦* (||)u::(||)u::(map (λx.x+1) [])
↦* (||)u::(||)u::[]
```

Notice that a hole can become duplicated during evaluation. Each hole in the trace has an associated environment with the overall goal type for the hole and a mapping from in-scope variables names and their types to run-time values. The environments for the two occurrences of the hole above are in Table 1.

(a) hole type: β = int

var	$\dot{\tau}$	val
f	*int* → *int*	λx.x+1
l	*int list*	[1,2]
x	*int*	1
xs	*int list*	[2]

(b) hole type: β = int

var	$\dot{\tau}$	val
f	*int* → *int*	λx.x+1
l	*int list*	[2]
x	*int*	2
xs	*int list*	[]

Table 1: environment around each occurrence of $(\!|\,|\!)_u$

This trace might lead the programmer to have more confidence about their solution so far. An editor might display the associated environments during the edit process, making it easier for the programmer to explore how to fill in the hole. Here, following the types, the programmer might fill the hole with $(\!|\texttt{f x}|\!)_u$. If she makes this change and then reruns her example, she will see that she has indeed satisfied the specification, and should now replace the hole with its contents:

```
    map (λx.x+1) [1, 2]
↦* (|((λx.x+1))1|)u::(map (λx.x+1) [2])
↦* (|1+1|)u::(map (λx.x+1) [2])
↦* (|2|)u::(map (λx.x+1) [2])
↦* (|2|)u::(|(λx.x+1)2|)u::(map (λx.x+1) [])
↦* (|2|)u::(|2+1|)u::(map (λx.x+1) [])
↦* (|2|)u::(|3|)u::(map (λx.x+1) [])
↦* (|2|)u::(|3|)u::[]
```

3. Metatheory

As terms change through evaluation, we rely on a notion of holes being associated with substitutions, closures and unique names borrowed directly from work on contextual modal type theory (CMTT) (Nanevski et al. 2008). This allows a programmer to view the trace as above and identify where each hole is used. CMTT is the Curry-Howard interpretation of contextual modal logic, which gives us confidence that our approach is rooted in the logical tradition. CMTT does not provide a way to reason about the dynamics of terms with free metavariables, nor does it include the notions of non-empty holes or holes in types which appear prominently in Hazelnut, so there is work to be done to extend it to our use case. In the judgemental form below, Δ maps each unique hole name to a typing context and type.

Progress states that a well-typed incomplete program will either step, is a value, or is indeterminate.

Conjecture 1 (Progress). *If* $\emptyset \vdash \dot{e} : \dot{\tau} \dashv \Delta$ *then either*

i) there exists $\dot{e}'$ *such that* $\dot{e} \longmapsto \dot{e}'$, *or*
ii) $\dot{e}$ **value**, *or*
iii) $\dot{e}$ **indet**

Preservation states that stepping preserves the type of an incomplete program and that no new hole names are created by evaluation.

Conjecture 2 (Preservation). *If* $\emptyset \vdash \dot{e} : \dot{\tau} \dashv \Delta$ *and* $\dot{e} \longmapsto \dot{e}'$ *then* $\emptyset \vdash \dot{e}' : \dot{\tau} \dashv \Delta'$ *and* $\Delta' \subseteq \Delta$.

Here, we need to explicitly consider the possibility of duplication of holes, but ensure that they continue to appear at the same type and with compatible closures.

Since complete terms coincide exactly with the simply typed lambda calculus, it is important that any dynamic semantics and associated metatheory for possibly incomplete terms agree with the standard treatment. For example, complete programs should not become indeterminate during evaluation, and standard type saftey should be provable for complete programs without changing the small-step relation.

References

S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in UI programming. In *PLDI*, 2013.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. In *POPL*, 2017. URL http://arxiv.org/abs/1607.04180.