

# Running Incomplete Programs

Ian Voysey

Carnegie Mellon University

Cyrus Omar

Carnegie Mellon University

Matthew A. Hammer

University of Colorado Boulder



Hello everyone! Thank you all for coming to what I think is the last talk in the last session of any track at POPL instead of having enough time before your flight to get something to eat.

My name is Ian Voysey. Today I'm going to talk about what it might mean to run incomplete programs. This is joint work between myself, Cyrus Omar at CMU, and Matt Hammer University of Colorado Boulder.

You should think of this talk as a kind of the best version of one of the bullet points in the "future work" section for the talk Cyrus gave in the main POPL session on Wednesday about Hazelnut --- which is why I'm entitled to use the cute hazel logo on my slides.

In that work, we present a calculus of structure editing that gives static meaning to every edit action and all the intermediate states, by incorporating several notions of holes in a program. We say nothing whatsoever about **what you might do with those intermediate programs that still have holes in them**. That's what I'm going to talk about today.

I should warn you that most of what I'm talking about today is just our dream for the future, it's very much work in progress--so to speak. We have some sketches of how these things might work but no tooling or real proofs, even just on paper. So if things seem fishy: they may well be!

# Hazelnut Syntax



2

I'm going to start by going over the relevant parts of the syntax offered in the Hazelnut work -- so it's OK if you didn't see Cyrus's talk.

This isn't going to be everything from that work, because so much of the effort in that paper is focused on giving static meaning to edit actions themselves and transitions between edit states.

This is more of the next step -- "now that i have some incomplete programs that i wrote in this nice way, what can i do with them?" -- so much of the machinery isn't relevant.

$$\begin{aligned}\dot{\tau} &::= \dot{\tau} \rightarrow \dot{\tau} \mid \text{num} \mid \textcolor{violet}{\parallel}_u \\ \dot{e} &::= x \mid \lambda x. \dot{e} \mid (\dot{e})\dot{e} \mid \underline{n} \mid \dot{e} + \dot{e} \mid (\dot{e} : \dot{\tau}) \mid \textcolor{violet}{\parallel}_u \mid \textcolor{violet}{(\dot{e})}_u\end{aligned}$$

$$\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau} \qquad \dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}$$



one of the main contributions was a language for intermediate edit states, with holes and a bidirectional type system to give static meaning to each state. holes themselves aren't new -- agda, ghc, etc.

we hope to avoid some of the problems with non-local interactions between holes, elisp nonsense, etc. that you're familiar with if you've used agda, by taking a more principled approach to the whole ecosystem.

(say what a bidirectional system is, algorithmic version of declarative rules with local inference, etc)

all this was work developed in lock-step with mechanization rather than having be post hoc due diligence, as evidenced by our artifact evaluation for POPL. that's a style we intend to continue as we develop a dynamic semantics because the payoffs are huge.

## Towards a Dynamics



4

One key feature of any dynamic semantics is that it must have theorems that are recognizably "progress" and "preservation".

to begin to state those, the first thing you need is to know when to stop, so to begin thinking about how we want a small step relation to act ...

$\dot{e}$  **value**

$(\lambda x.9 + \text{hole}_u)$  **value**

$\dot{e}$  **indet**

$(9 + \text{hole}_u)$  **indet**



5

Usually these rely on a judgement that describes which expressions are values. So obviously we have that judgement as well, and it's what you'd expect. For example, lambdas without arguments are still a value form, even if they have holes in them---they have no argument, so how could you hope to say what they step to?

but what to do about terms that really rely on the holes in them? where a hole appears in an elimination position? they can't properly be called values because they are different -- they have holes in them, values shouldn't, etc.

To get the right sense for P&P, we then also have another judgment for "indeterminate forms" -- expressions which are not yet values, but upon which no more work can be done because there's a hole in the way.

(example)

we sometimes call a term that is *either* a value or indeterminate "final", and use this notion in most of the places you'd expect to see just "value" in a more standard dynamics

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

- i) there exists  $e'$  such that  $e \mapsto e'$ , or*
- ii)  $e$  **value**, or*
- iii)  $e$  **indet***

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



with those judgements, we can state progress and preservation in familiar forms.

note that the **judgemental form of the typing rules has changed**. It's no longer bidirectional, because doing so means you have to duplicate the metatheory for no reason; therefore we have to relate our bidirectional system to this declarative one in the appropriate way. this is more or less standard.

We also have another context to the right of the ending turnstile -- this is a context that contains all the names of the holes. We're borrowing this idea almost entirely from Nanevski, Pfenning, and Pientka, but i'll talk about that later.

note that these are slightly curious -- they're closed terms in  $\Gamma$  but open in  $\Delta$ . so it's an interesting hybrid of open and closed term evaluation that we don't really understand yet.

note that holes can appear and disappear during eval traces, hence the side condition on names for preservation. we're not really sure if  $\subseteq$  is the right relation for  $\Delta$  and  $\Delta'$ , but there's something there for sure, so that symbol should be thought of as metasyntax

$$(\lambda x.x + \llbracket \cdot \rrbracket_u) \llbracket \cdot \rrbracket_{u'} \longmapsto \llbracket \cdot \rrbracket_{u'} + \llbracket \cdot \rrbracket_u$$

$$(\lambda x.x + \llbracket f\ x \rrbracket_u) 5 \longmapsto 5 + \llbracket f\ 5 \rrbracket_u$$



Keeping these ideas in the back of your head, we can think about a few examples of how we might want terms to step in a small-step dynamic semantics and how they interact with P&P. why do they have the same type? why does the typedness guarantee that they can step?

note that we're **not treating these holes like exceptions**, which languages like Agda do if you try to normalize a term with a hole in it and the execution path actually touches that hole.

having a final form is enough to substitute -- even though these things aren't values. this is why it's important for holes to have unique names; you could imagine if  $x$  appears more than once the hole seems to be "duplicated", or if it doesn't appear it goes away in this particular subterm. unique names give us some hope of knowing the provenance of the hole in the resultant expression.

we have to substitute into variables where ever they appear, even in non-empty holes, because preservation obviously breaks with free variables.

these examples do not really reveal if i'm talking about an eager or lazy evaluation semantics. some of this discussion seemingly works for both -- there is one place that's not quite independent that i'll get to later when i have a better example.

## Pie-In-The-Sky Example



8

So those are all ideas that we can present inside the very austere syntax of Hazelnut.

Now I'm going to step out of reality: imagine that Hazelnut exists in some ideal future form that's like a mini-ML -- no modules maybe, but ADTs, recursion, polymorphism, pattern matching, the works.



**map** :  $(\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$

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



In this hypothetical syntax, imagine a programmer implementing map on lists for the first time in an interactive system a little bit like Agda's emacs mode that allows them to really work with the holes in their term, but with the structure editor features from hazelnut that banning them from ever producing nonsense incomplete programs.

At some point in that process, you may know what pattern of recursion you're going to follow -- that map is structurally recursive, basically -- but have no idea what to do at the inductive step, of what to do at each specific element.

How can the system help them figure out what to do next?

$$\text{map } (\lambda x.x+1) \ [1, 2]$$

$$\mapsto^* \text{hole}_u :: (\text{map } (\lambda x.x+1) \ [2])$$

$$\mapsto^* \text{hole}_u :: \text{hole}_u :: (\text{map } (\lambda x.x+1) \ [])$$

$$\mapsto^* \text{hole}_u :: \text{hole}_u :: []$$


x?

(a) hole type:  $\beta = \text{int}$

var	$\dot{\tau}$	val
f	$\text{int} \rightarrow \text{int}$	$\lambda x.x+1$
x	$\text{int}$	1
xs	$\text{int list}$	[2]
$\alpha$	—	int

(b) hole type:  $\beta = \text{int}$

var	$\dot{\tau}$	val
f	$\text{int} \rightarrow \text{int}$	$\lambda x.x+1$
x	$\text{int}$	2
xs	$\text{int list}$	[]
$\alpha$	—	int



10

Well, you run the program!

And you get a trace in the small step semantics that looks something like this.. ( $\mapsto^*$  is reflexive transitive closure, so there's some question of what to elide in these traces)

Note that a hole can appear multiple times in the trace, in effect getting duplicated. This is why holes need to have unique names. You want to be able to look at a trace of a program and see which holes came from which part of your candidate code so you can figure out what to write there.

Indeed, you can go one step further and let the programmer inspect the holes. what they get is the environment around that hole when it was passed by in the relationship. This comes right out of CMTT.

It's worth pointing out that this is a task almost all of us are familiar with doing in an ad hoc way with students if you've TAed a functional programming course.  
**Reasoning about incomplete programs isn't bolted onto the side of an editor --- it's a fundamental activity that is how people figure out what to write next.**

so let's say the programmer still doesn't know what to do but they know that it has something to do with x -- note that x isn't type correct so we couldn't replace the hole with x right away.

(it works because beta happens to be the same as alpha in this instance, but the polymorphism restricts you. and the editor would need to know this sort of thing: not just what the type variables are bound to in a particular example, but also the unification constraints at the same time.)

$$\begin{aligned}
& \text{map } (\lambda x.x+1) \ [1, 2] \\
\mapsto^* & \ (\text{hole})_u :: (\text{map } (\lambda x.x+1) \ [2]) \\
\mapsto^* & \ (\text{hole})_u :: (\text{hole})_u :: (\text{map } (\lambda x.x+1) \ []) \\
\mapsto^* & \ (x)_u :: (\text{hole})_u :: []
\end{aligned}$$


so the user can then make that change, right in the trace!

ok, now i know this looks weird, so don't get excited just yet.

why does it look weird? well: the holes are all from the same place in the source program, and we know that because they all have the same name. so it doesn't make sense to change just one of them, and the system knows that, so..

$$\begin{aligned}
& \text{map } (\lambda x.x+1) \ [1, 2] \\
\mapsto^* & \ (\lambda x.x+1)_u :: (\text{map } (\lambda x.x+1) \ [2]) \\
\mapsto^* & \ (\lambda x.x+1)_u :: (\lambda x.x+1)_u :: (\text{map } (\lambda x.x+1) \ []) \\
\mapsto^* & \ (\lambda x.x+1)_u :: (\lambda x.x+1)_u :: []
\end{aligned}$$


the change actually takes place everywhere that hole appears in the trace! the variable name will always be in scope because it's "in the same place" in the source program.

$$\begin{aligned}
& \text{map } (\lambda x.x+1) \ [1, 2] \\
\mapsto^* & \ (1)_u :: (\text{map } (\lambda x.x+1) \ [2]) \\
\mapsto^* & \ (1)_u :: (2)_u :: (\text{map } (\lambda x.x+1) \ []) \\
\mapsto^* & \ (1)_u :: (2)_u :: []
\end{aligned}$$


and! since we know the closure for each hole, we can make the appropriate variable substitutions. this now also looks odd, but you can imagine dancing between the two views -- variables or what they are under the hole-specific substitutions.

**map** :  $(\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$

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



...and we can go one step further than that, and back propagate the change into source of the program! this really smooths out the difference between working on the trace and working on the original source. the programmer could have made the change here too, first, but maybe we don't want to force them to.

this is "edit and resume" or, if you'll forgive me, "larger than live programming"

$$\mathbf{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$$

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



15

So let's say our erstwhile programmer decides to fill the hole in their program with `f x`. it happens to be type-correct in that it's now a beta and therefore a candidate for a `finish` action.

but!

that doesn't mean you have to. one thing that holes can do is delay type checking for sub expressions that are type correct, but not type correct wrt their surroundings. **another thing** holes can do is serve to tag subexpressions that are still under consideration. **well-typed programs don't go wrong, but not all type-correct programs are correct programs.** the programmer should be able to see how a change acts before they commit to it.

so, specifically, you leave the expression and re-run the tracer

```

      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 :: []

```



this is what i meant when i said that there's something that kind of inherently eager. the semantics of function application can be either eager or lazy, but i really think you want to see how the things in the holes evaluate, so you do want to eagerly reach inside and evaluate non-empty holes. or maybe i'm just biased because i'm from CMU and an SML programmer? this is an interesting design question.

this really demonstrates how this kind of environment encourages the programmer to **explore**, but with a unique amount of tool support as they do. i like to think of this as kind of "the best possible debugger" because you don't ever really make the bug.

i should also point out that there's connections here to incremental computation, following Matt Hammer's work -- these traces don't need to get recreated every single time, because there's a ton of shared structure as the program evolves in many cases. that's also something we're thinking about, as an implementation concern more than a metatheoretical one.



# Related Work

- Hazelnut at POPL, submitted to SNAPL
- Contextual Modal Type Theory
  - Nanevski, Pfenning, and Pientka, TCL 2008
- Gradual Typing
  - Siek and Taha, SFPW 2006, and many others
- Evaluating open terms



17

alright! that's what i'd like to show you today, but i want to talk about some things that we're related to. this is not an exhaustive list by any means: people have been thinking about a lot of the issues around holes, structure editors, and dynamics for unusual calculi for a long time.

these dynamics really sort of are the next step of what hazel started -- exploring the behavior of your program is what you do when you want to figure out what edit action to do next. we let you do that in a way that tags the sub expressions you're working on to see how they interact, even if they aren't type correct yet.

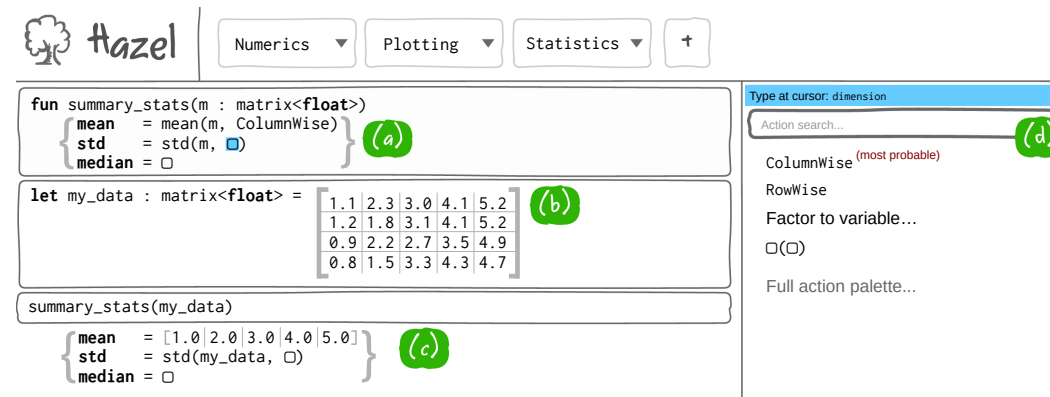
cmmt gives us the notion of Delta, closures for holes, but no type holes. and no non-empty holes -- very similar proof theory but not the right way to have an interactive tool. cmmt also comes from contextual modal logic -- so we know we're coming from a strong position in tradition, this isn't ad hoc, there are mathematical reasons that these things act the way that they do.

gradual typing has a kind of holes in types but no expression holes. (and we accidentally rediscovered some of their meta theory structure in doing the Hazel work, so there is some deeper connection here to be understood.)

so we bridge the gap in some sense -- it seems that CMTT and gradual typing are kind of two points on a spectrum and that trying to unify them into something like "gradual contextual modal type theory" is more interesting than it might seem. we can write down a small step relation for expressions with just term holes and it's not that surprising, but making it work with type holes at the same time is hard.

open terms: P&P are about terms that are closed in gamma but open in delta, so there is something more to be said about the relationship to equational theories, etc.

# What Next?



Cyrus Omar, Ian Voysey, Matthew Hammer, Michael Hilton,  
Jonathan Aldrich, Claire Le Goues



18

this has all been future work in some sense, and only a little bit of it. There are other directions we want to consider with derived actions and meta-programming and other things.

holistic approach to programming environment; really kind of smoothing the transition between modes of thinking about it. how about test driven hole and program repair, providing good suggestions for how to fill holes from program synthesis? many other things.

this "screen shot" is the mock-up of the tool that we're pointed towards, conceptually. from our POPL paper, due to M. Hilton's efforts, we have a small implementation of the core calculus of edit actions from hazelnut and you can play with that on the web, but really we're looking larger.

once we work out the theory to get a dynamic semantics that acts how we feel like it should, an obvious the next step is to start adding language features and making this something people might be able to actually use for small programs. in theory you should be able to teach people to program with a tool like this if you wanted to.

it's also **not just for teaching**. if you've ever done non-trivial Agda development: you know that this is not trivial. the map example seems easy because it's what fits into the scope of a talk: if you do this right and develop a robust theory of how this works, it'll work for any program.

# Questions?

[hazeltrove.org](http://hazeltrove.org)



19

Alright, that's all I have to say!

You can find more information, including links to all the goodies for the main POPL paper and the slides and abstract for this talk, and our vision paper that's in submission to SNAPL, on our website which is [hazeltrove.org](http://hazeltrove.org).

Thank you all very much for your attention; I am excited to answer any questions.