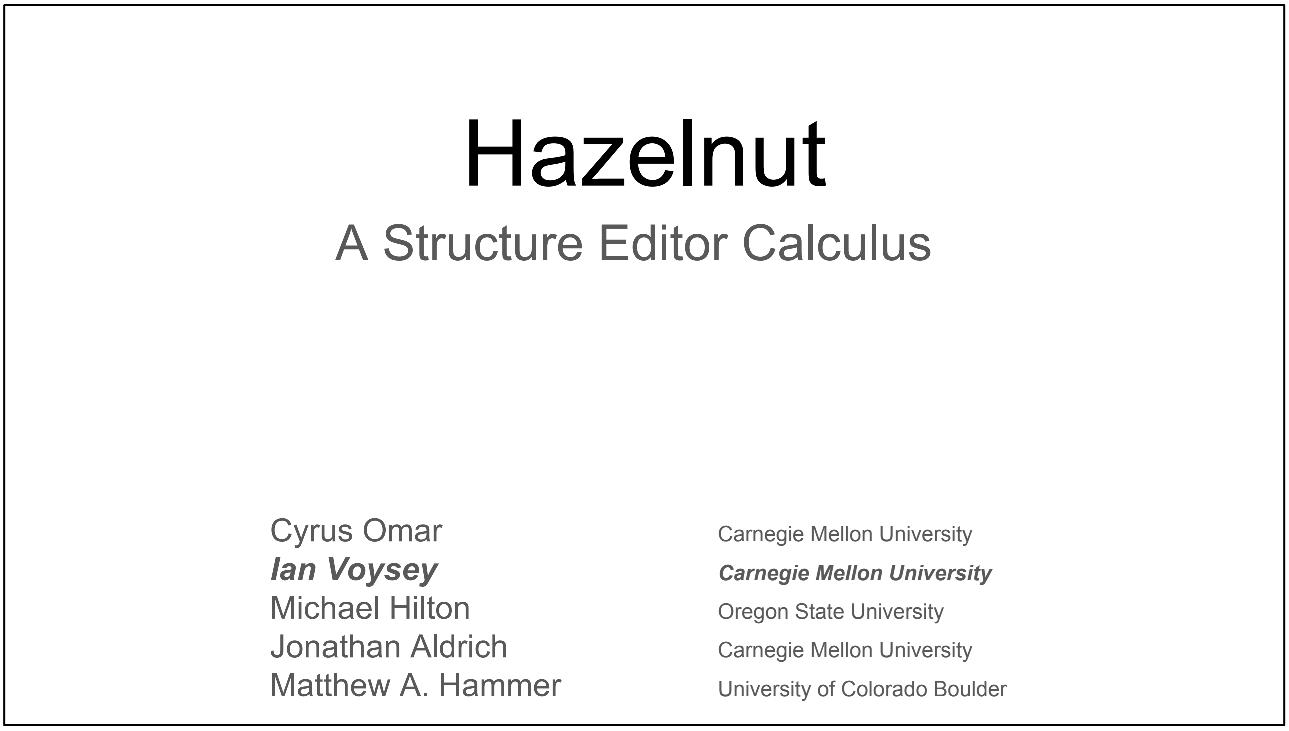
FAIR WARNING ABOUT THIS FILE

The slides in this PDF are what I used to present Hazelnut at PoP Seminar on 13 October 2016.

Because there is not much text on each slide to describe what’s going on, I have also included my speaker’s notes.

As writing, these notes are a little bit rough. There are almost certainly small errors, bad grammar, over- and understatement---and I know for sure that there’s a fair amount of sarcasm. They were written for me to skim from across the room mid-talk in case I forgot something I wanted to say. They are not a transcript of what I said, nor should they be regarded as the serious last word on Hazelnut.

The text of the POPL17 paper is the best source for definitive answers to offline questions about Hazelnut. If you want to know more or have questions, though, the actual best option is just to email me and ask.



Hazelnut

A Structure Editor Calculus

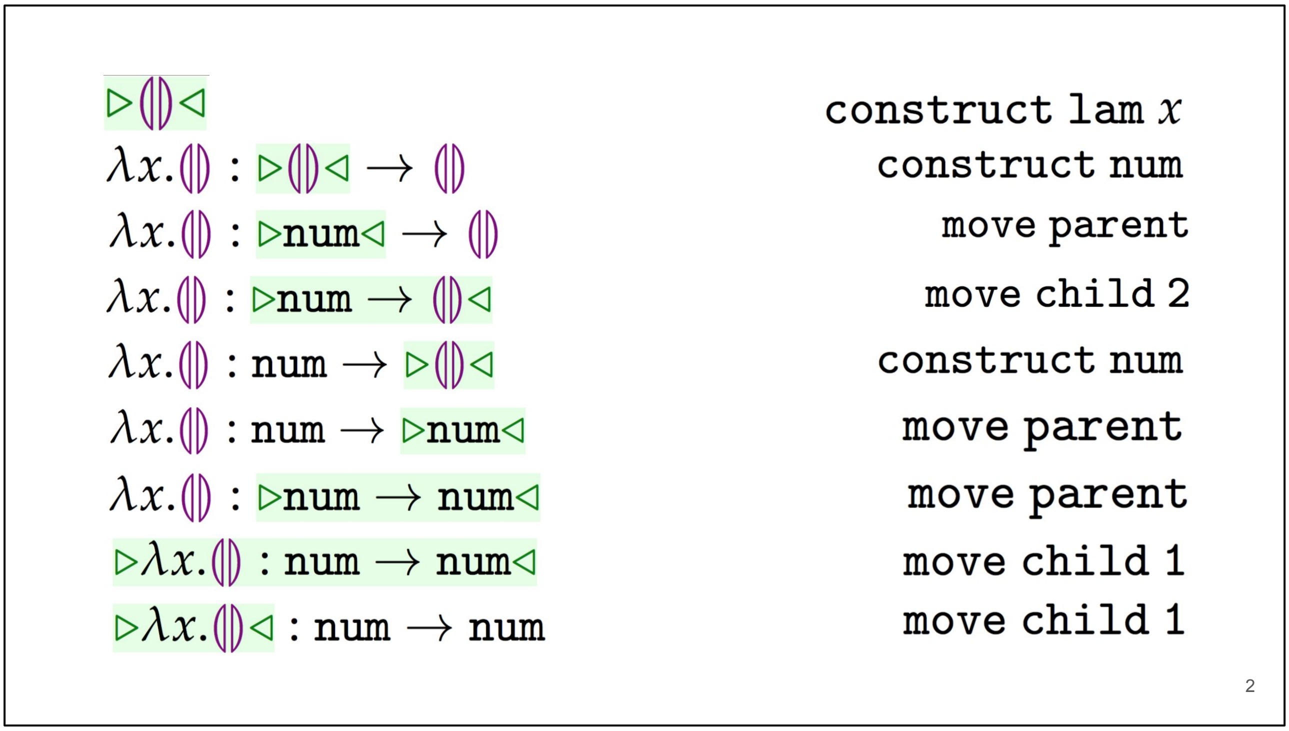
Cyrus Omar — Carnegie Mellon University
Ian Voysey — ***Carnegie Mellon University***
Michael Hilton — Oregon State University
Jonathan Aldrich — Carnegie Mellon University
Matthew A. Hammer — University of Colorado Boulder

Good afternoon, everyone! Thank you all for coming to my talk. For those of you who don't know me, my name is Ian Voysey. Even though POP Seminar is traditionally a venue for folks *not* from CMU and is scheduled opposite President Obama, I'm very glad to be speaking here today and grateful that we've collectively decided to, uh, stoop this low.

I'm going to talk to you today about a project called Hazelnut, which I spent most of my spring and summer formalizing in Agda. Hazelnut is a structure editor calculus.

This project is joint work between myself, Cyrus Omar and his advisor Jonathan Aldrich here at CMU, Michael Hilton at Oregon State University, and Matt Hammer at University of Colorado Boulder.

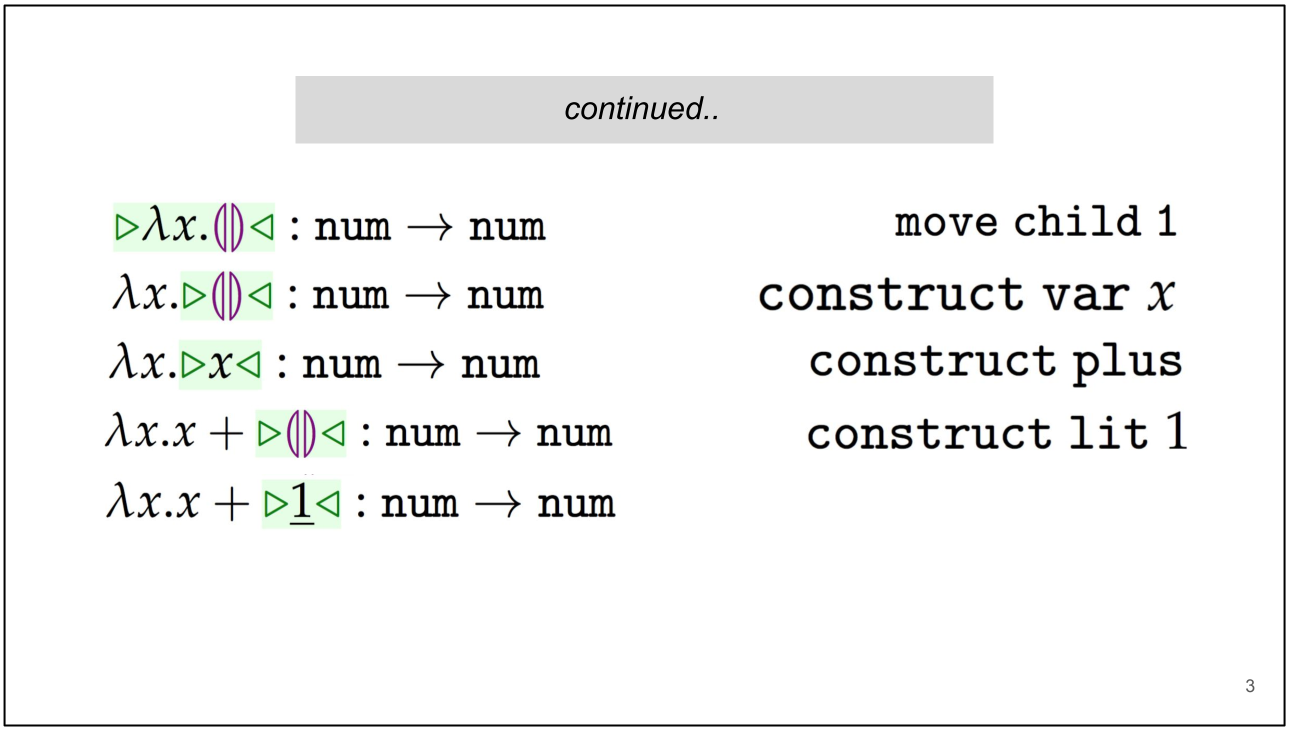
We presented an earlier draft of this work at TFP in the late spring. We just heard back, last week, that our extended paper based on that work was accepted for POPL. So we're very much looking forward to presenting this work there in the winter, and I'm excited to give you a preview now.



$\triangleright\llparenthesis\rrparenthesis\triangleleft$	$\texttt{construct lam}\ x$
$\lambda x.\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft \to \llparenthesis\rrparenthesis$	$\texttt{construct num}$
$\lambda x.\llparenthesis\rrparenthesis : \triangleright\texttt{num}\triangleleft \to \llparenthesis\rrparenthesis$	$\texttt{move parent}$
$\lambda x.\llparenthesis\rrparenthesis : \triangleright\texttt{num} \to \llparenthesis\rrparenthesis\triangleleft$	$\texttt{move child 2}$
$\lambda x.\llparenthesis\rrparenthesis : \texttt{num} \to \triangleright\llparenthesis\rrparenthesis\triangleleft$	$\texttt{construct num}$
$\lambda x.\llparenthesis\rrparenthesis : \texttt{num} \to \triangleright\texttt{num}\triangleleft$	$\texttt{move parent}$
$\lambda x.\llparenthesis\rrparenthesis : \triangleright\texttt{num} \to \texttt{num}\triangleleft$	$\texttt{move parent}$
$\triangleright\lambda x.\llparenthesis\rrparenthesis : \texttt{num} \to \texttt{num}\triangleleft$	$\texttt{move child 1}$
$\triangleright\lambda x.\llparenthesis\rrparenthesis\triangleleft : \texttt{num} \to \texttt{num}$	$\texttt{move child 1}$

Let's start by stepping through a **simple** example: writing the increment function. Right now this is just a bunch of syntax, but I promise that everything i'm about to show you is formally defined (including the new lines). My goal for this talk is basically to just explain everything on this slide. In doing so, I'll tell you almost everything about Hazelnut.

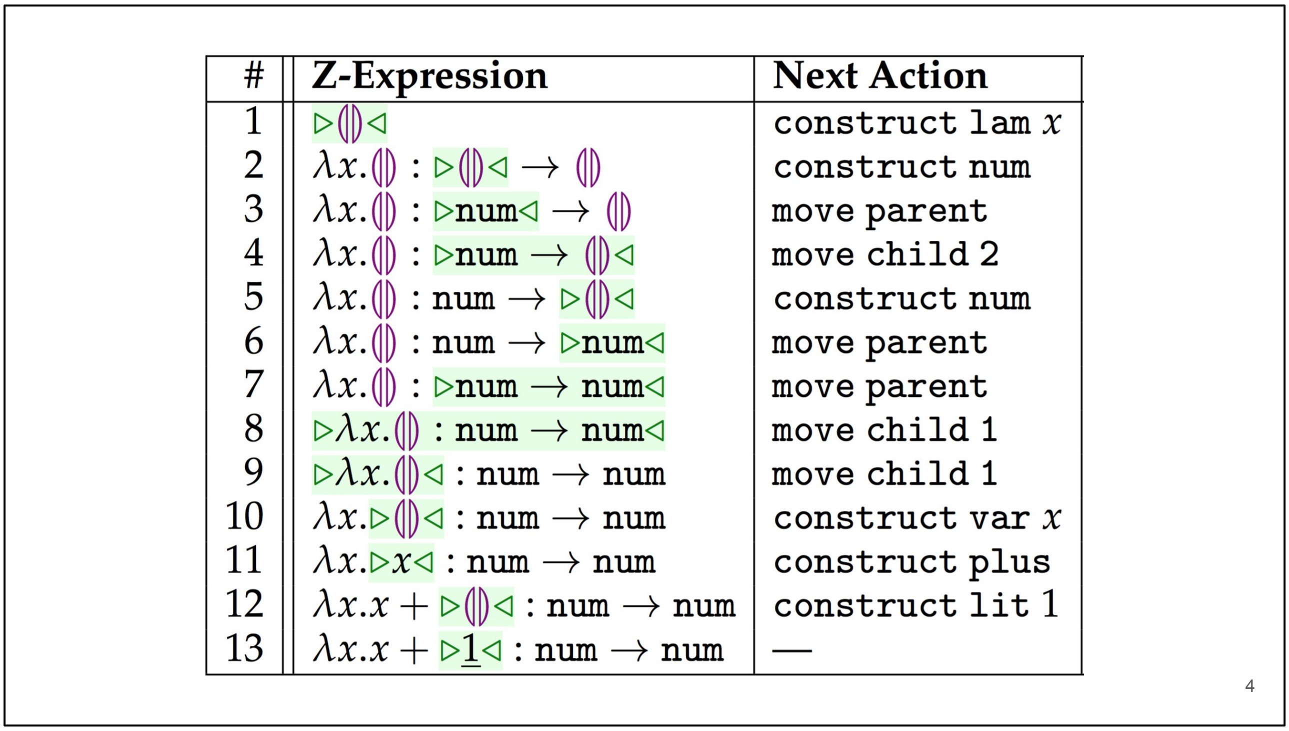
- purple hotdog bun lookin' thing is a hole, which we use to mark places in an expression where a subterm must go but hasn't been written yet
- green thing + wedges is the cursor, which indicates where the action is taking place
- each line is a (possibly partial) expression
- right hand column is the name of an action
- move between lines / programs by applying edit actions to the current edit state
- end (hopefully) in a complete expression.



continued..

$\triangleright\lambda x.(\!\|\,\|\!)\triangleleft : \mathtt{num} \to \mathtt{num}$	$\mathtt{move\ child\ 1}$
$\lambda x.\triangleright(\!\|\,\|\!)\triangleleft : \mathtt{num} \to \mathtt{num}$	$\mathtt{construct\ var}\ x$
$\lambda x.\triangleright x\triangleleft : \mathtt{num} \to \mathtt{num}$	$\mathtt{construct\ plus}$
$\lambda x.x + \triangleright(\!\|\,\|\!)\triangleleft : \mathtt{num} \to \mathtt{num}$	$\mathtt{construct\ lit}\ 1$
$\lambda x.x + \triangleright\underline{1}\triangleleft : \mathtt{num} \to \mathtt{num}$	

(continued)



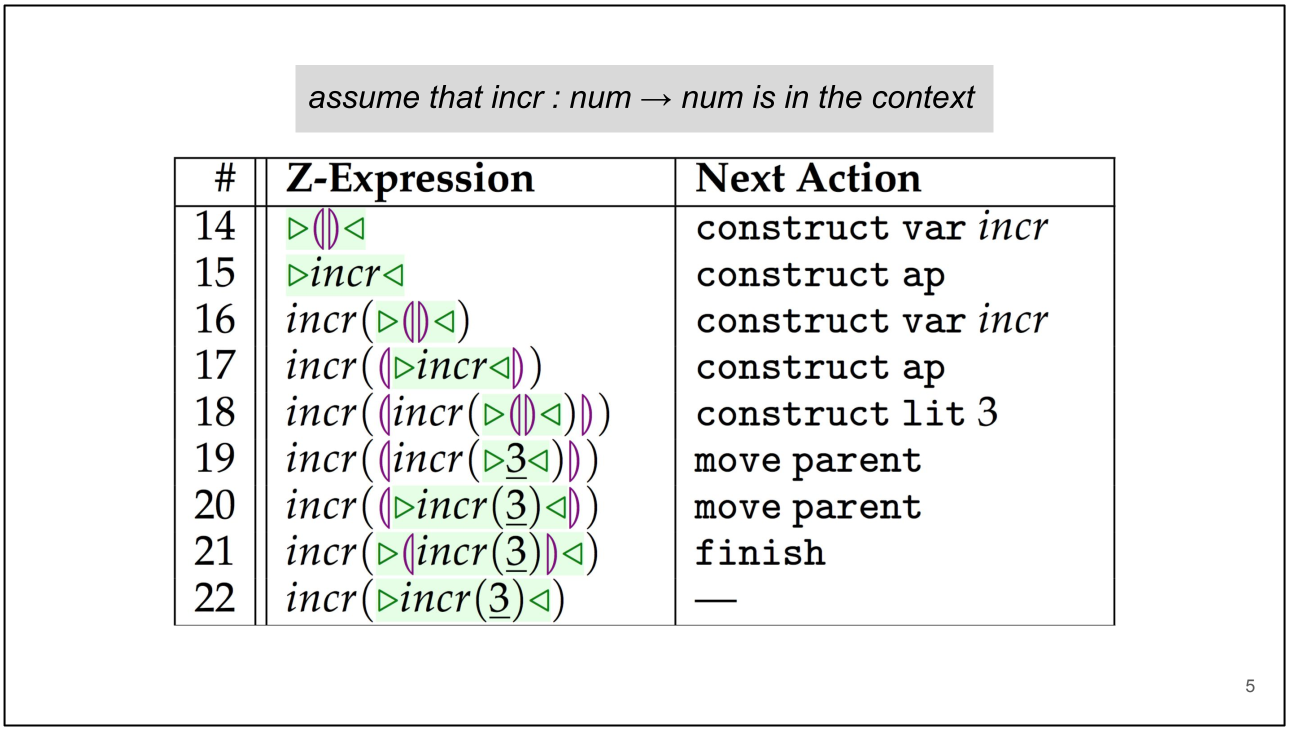
#	Z-Expression	Next Action
1	$\triangleright\llparenthesis\rrparenthesis\triangleleft$	construct lam x
2	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft \to \llparenthesis\rrparenthesis$	construct num
3	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\mathtt{num}\triangleleft \to \llparenthesis\rrparenthesis$	move parent
4	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\mathtt{num} \to \llparenthesis\rrparenthesis\triangleleft$	move child 2
5	$\lambda x.\llparenthesis\rrparenthesis : \mathtt{num} \to \triangleright\llparenthesis\rrparenthesis\triangleleft$	construct num
6	$\lambda x.\llparenthesis\rrparenthesis : \mathtt{num} \to \triangleright\mathtt{num}\triangleleft$	move parent
7	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\mathtt{num} \to \mathtt{num}\triangleleft$	move parent
8	$\triangleright\lambda x.\llparenthesis\rrparenthesis : \mathtt{num} \to \mathtt{num}\triangleleft$	move child 1
9	$\triangleright\lambda x.\llparenthesis\rrparenthesis\triangleleft : \mathtt{num} \to \mathtt{num}$	move child 1
10	$\lambda x.\triangleright\llparenthesis\rrparenthesis\triangleleft : \mathtt{num} \to \mathtt{num}$	construct var x
11	$\lambda x.\triangleright x\triangleleft : \mathtt{num} \to \mathtt{num}$	construct plus
12	$\lambda x.x + \triangleright\llparenthesis\rrparenthesis\triangleleft : \mathtt{num} \to \mathtt{num}$	construct lit 1
13	$\lambda x.x + \triangleright\underline{1}\triangleleft : \mathtt{num} \to \mathtt{num}$	—

this is a summary of what i just showed you, but in a more tabular form, so you can think of it like a trace of a bunch of edit actions that ends in a complete program -- one with no holes

at first swipe, this is just structure editing -- we get to modify the AST itself, directly. but with hazelnut, we're doing more.

structure editing traditionally only makes sure that you don't make syntactic errors -- like forgetting parenthesis. it does not offer **static reasoning principles about the intermediate states and edit actions**. in hazelnut, we guarantee that every intermediate edit state is statically well-defined

and i've proven everything in Agda. as a result of that, we can prove theorems about features of a structure editor that are usually hand-waived.



assume that incr : num → num is in the context

#	Z-Expression	Next Action
14	▷⦇⦈◁	`construct var` *incr*
15	▷*incr*◁	`construct ap`
16	*incr*(▷⦇⦈◁)	`construct var` *incr*
17	*incr*(⦇▷*incr*◁⦈)	`construct ap`
18	*incr*(⦇*incr*(▷⦇⦈◁)⦈)	`construct lit` 3
19	*incr*(⦇*incr*(▷$\underline{3}$◁)⦈)	`move parent`
20	*incr*(⦇▷*incr*($\underline{3}$)◁⦈)	`move parent`
21	*incr*(▷⦇*incr*($\underline{3}$)⦈◁)	`finish`
22	*incr*(▷*incr*($\underline{3}$)◁)	—

before we leave this example entirely, i want to use it to talk about another design goal that we had that sets hazelnut apart from traditional structure editors.

we do not want to prescribe a particular order in which a programmer must write their program

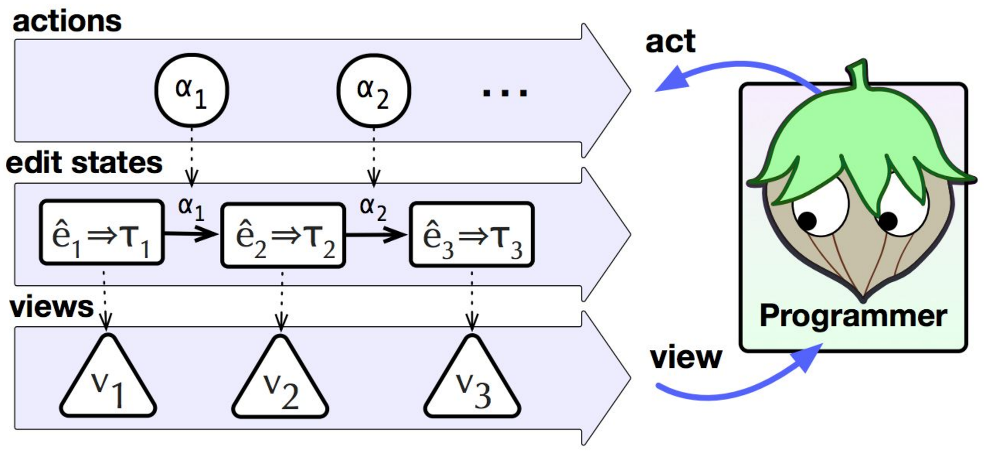
specifically in this example … (step through)

being functional programmers we think of this as composition. it's incr after incr. it's pretty natural to write the next function call next, that's how you do it in text. no one thinks of writing an application site.

the type you get from the analytic structure is num, but you want to write something that's num -> num.

so when we're in a place where the typing and structure of the surrounding term tells us that an action makes sense but doesn't result in an appropriate type, we **DO NOT BAN THAT ACTION**. instead we **delay the consistency check** by putting the result in a hole and let editing progress. if what's in that hole never grows up to be consistent, then tough noogies, you can't replace the hole with its contents. otherwise, you can when you choose to.

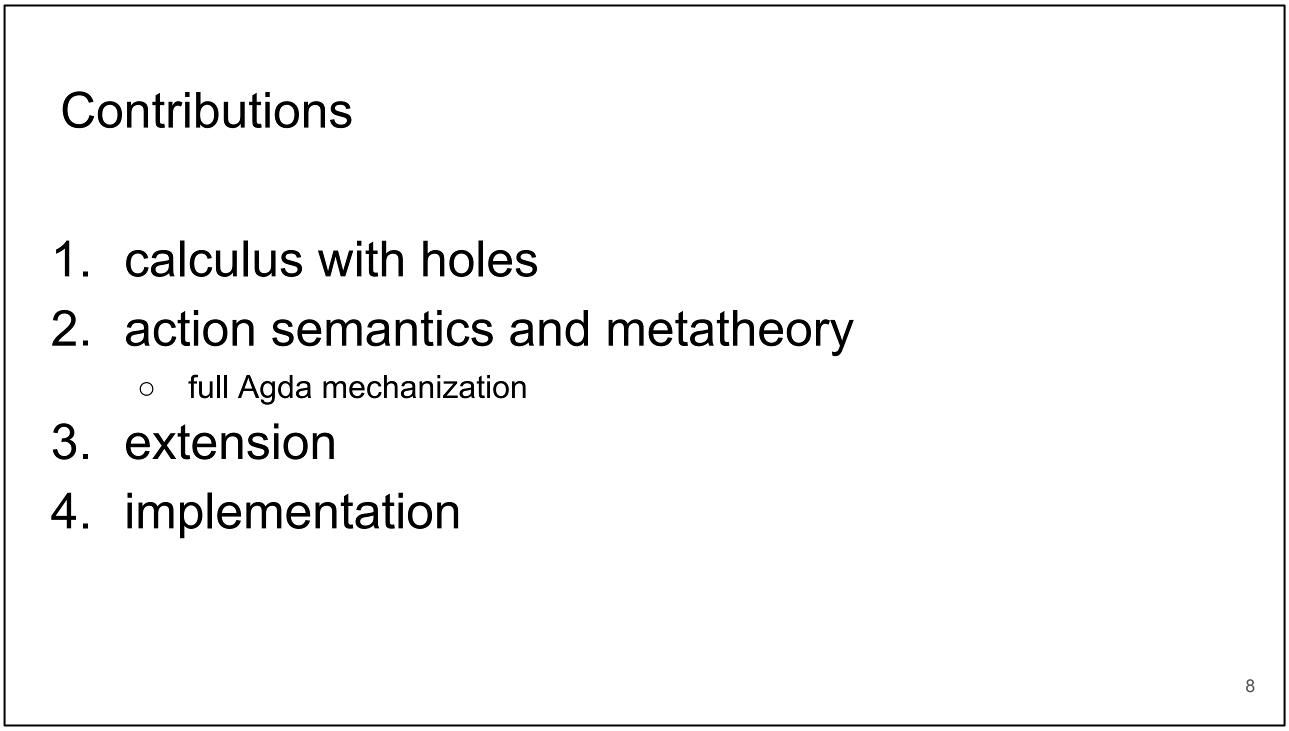
(we also do allow the other trace as well, where you choose to write ap first; that's perfectly fine, we just don't want to force the programmer either way as long as we can make sense internally to the calculus of both things)



high level: we’re internalizing the above picture, and be able to statically reason about each part of it and each intermediate state.

specifically, reason statically about: a programmer inspecting a partially written program of some type, determining what sensible action they would like to perform, applying that action, observing the resultant partial program (and type)

and we want to do this, as in the incr incr example, while using non-empty holes to avoid constraining the programmer to writing things in some one specific way.



Contributions

1. calculus with holes
2. action semantics and metatheory
   - full Agda mechanization
3. extension
4. implementation

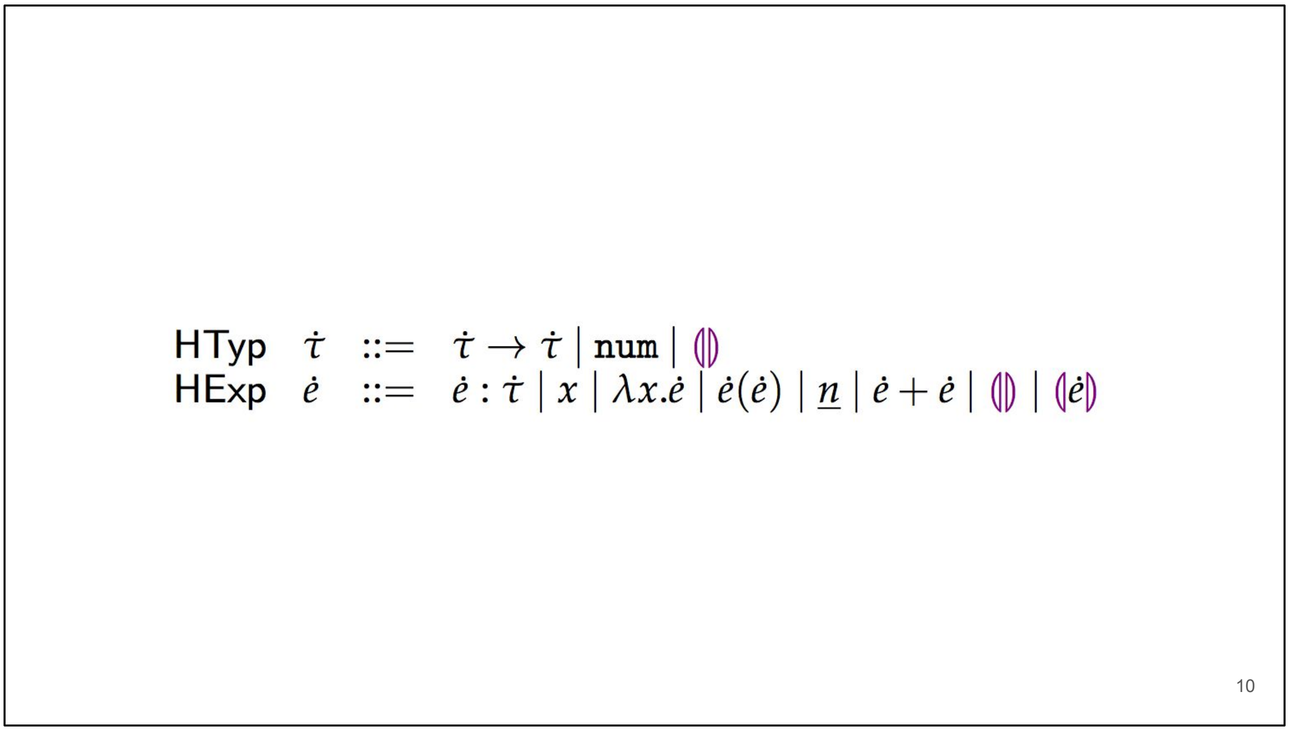
the specific contributions that we offer towards this goal are

- a calculus of with two kinds of holes as i described above
- a semantics for actions that gives static meaning to each phase in the interaction i just described (and meta theory that guides that development, that i proved entirely in agda)
- using this metatheory as a guide, an extension of the language with a new feature (also guided by metatheory)
- the implementation of a toy editor using this semantics

… and this amounts to the structure of my talk

Calculus with Holes

OK! Let’s start with the core language.



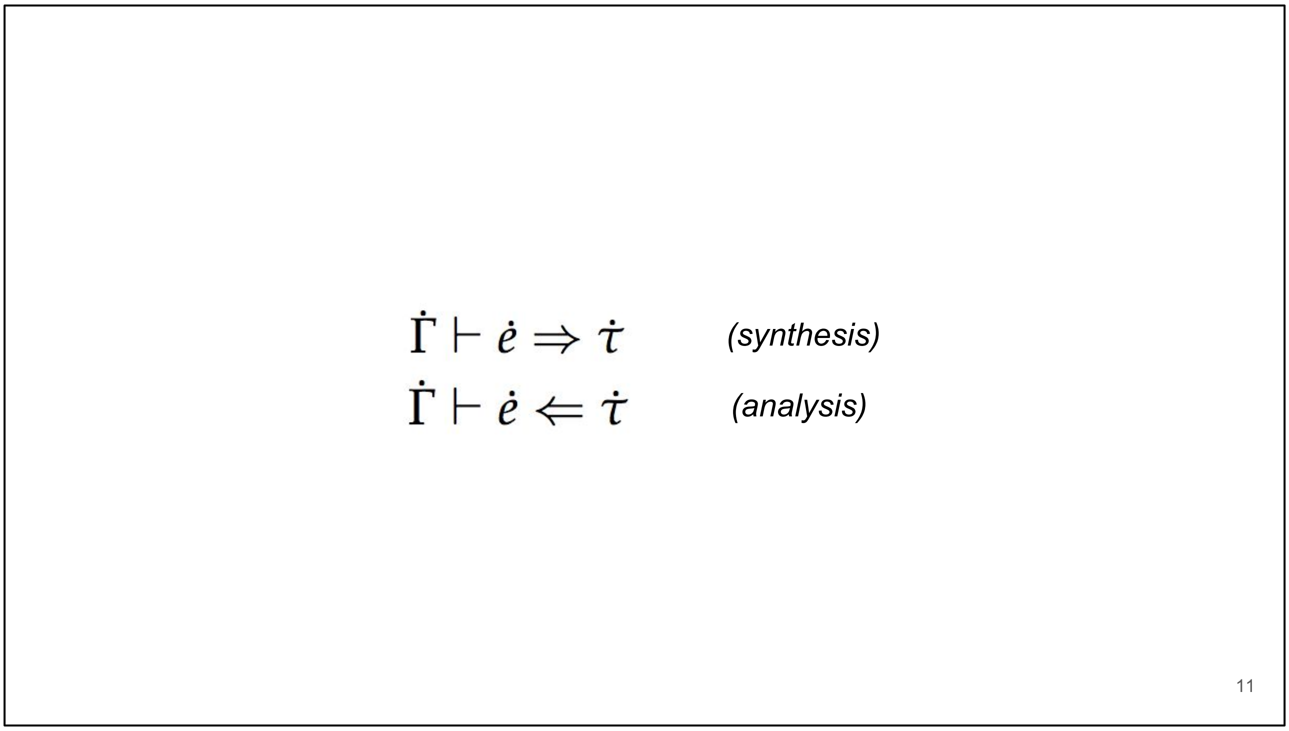
$$
\begin{array}{lll}
\textsf{HTyp} & \dot{\tau} & ::= \quad \dot{\tau} \to \dot{\tau} \mid \mathtt{num} \mid \llparenthesis\rrparenthesis \\
\textsf{HExp} & \dot{e} & ::= \quad \dot{e} : \dot{\tau} \mid x \mid \lambda x.\dot{e} \mid \dot{e}(\dot{e}) \mid \underline{n} \mid \dot{e} + \dot{e} \mid \llparenthesis\rrparenthesis \mid \llparenthesis\dot{e}\rrparenthesis
\end{array}
$$

the language of hazelnut is a stripped down, largely standard, lambda calculus. here's the grammar.

the non-standard things are holes, non-empty holes, and the hole type. i motivated these above but will give them a formal meaning as the talk progresses.

holes aren't new -- beluga, agda, idris, CMTT. NEholes are new. they don't appear previously because you don't "need" them proof theoretically--you can replace <| e |> with <||> e--but we want them because it lets us give a really clean description of what we're interested in.

this work is more intended to present a *methodology*. language is very austere, but like i promised i'll describe how to extend it with a new construct and how the metatheory we've proven guides that extension.



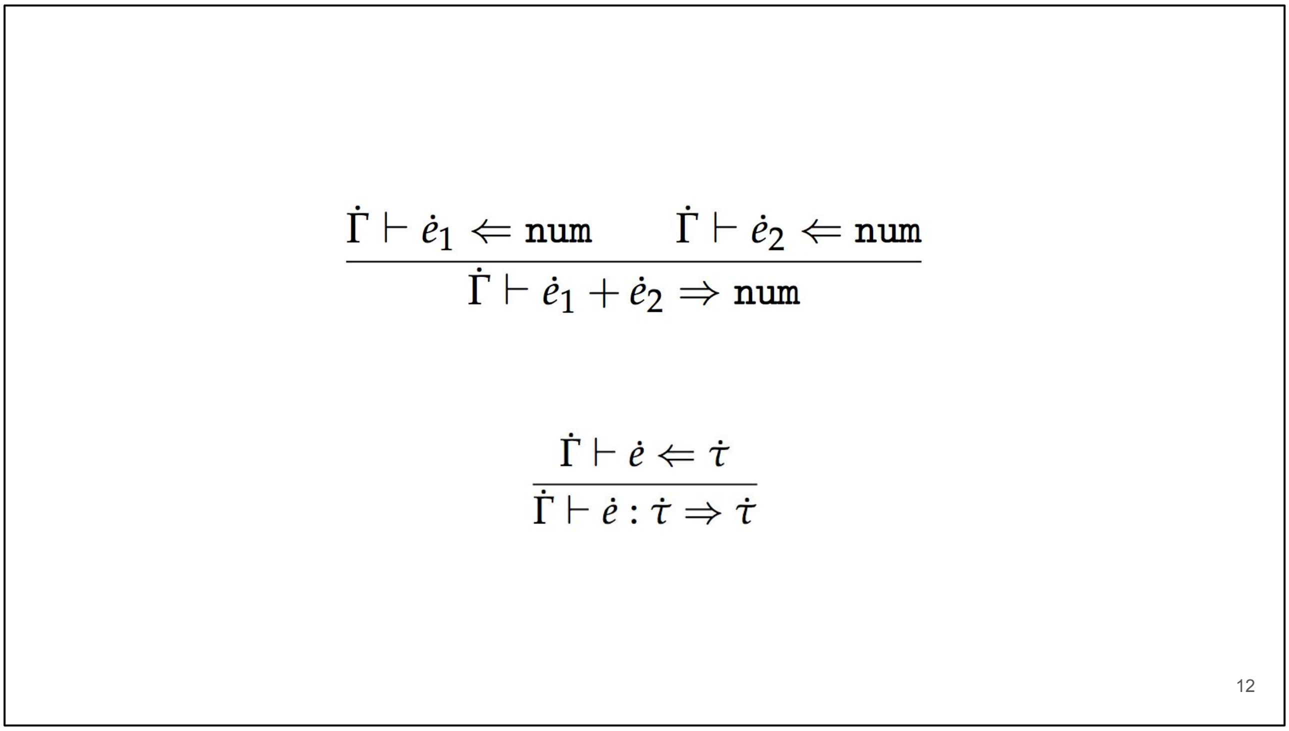
$$\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau} \qquad \textit{(synthesis)}$$
$$\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau} \qquad \textit{(analysis)}$$

hazelnut is built around a bidirectional type system.

this is the type system that results from the standard process of taking a vanilla declarative system and making it algorithmic by distinguishing between two modes: “synthesis”, where we think of the type as output determined by the structure of the expression; and “analysis”, where we think of the type as input that the expression must conform to based on its environment.

NB: programmatically at the top level of a typechecker for a bidirectional system, we always start out in the synthetic mode. we know nothing a priori, and also need discover what the type is from the structure (and then output it).

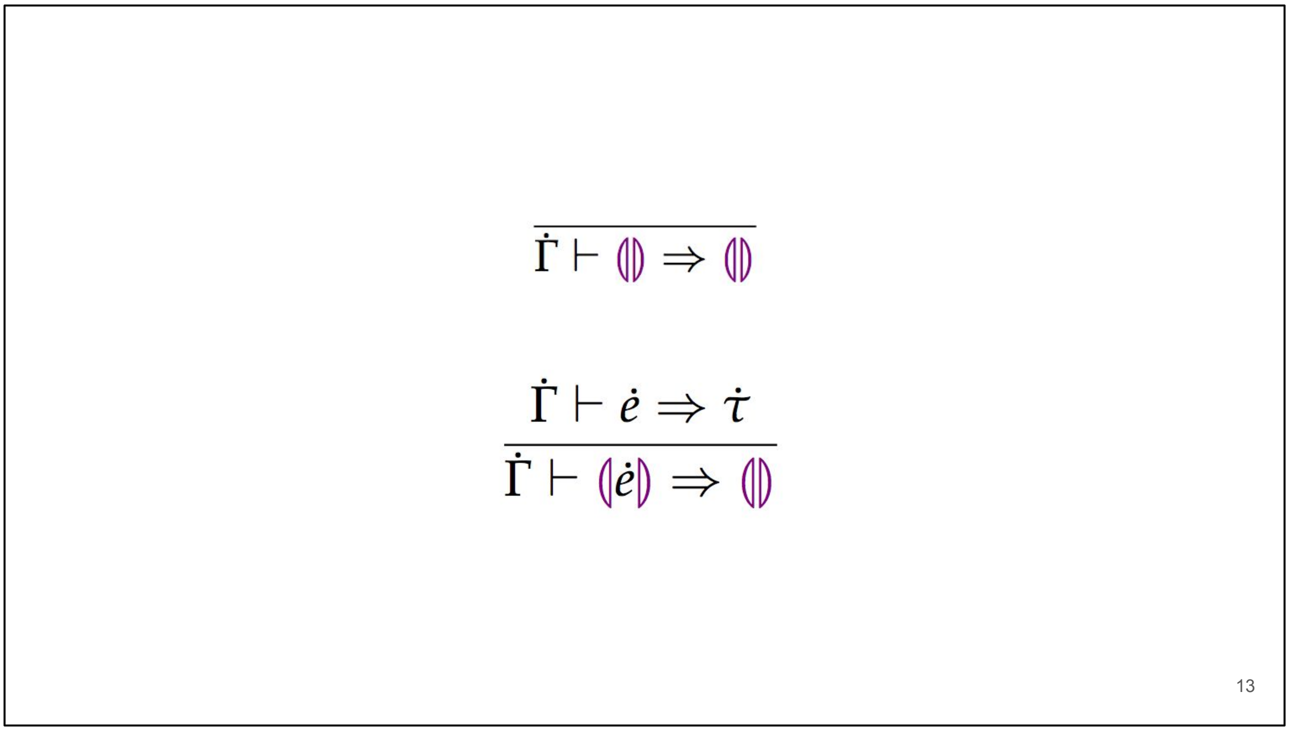
in the context of this work, we chose to use a bidirectional system because when it propagates the type information down into the terms in the typing rules themselves, we always have enough local type information to describe type-directed edit actions. we want to know when a type of a hole is pinned down because of how it’s used.



$$\frac{\dot{\Gamma} \vdash \dot{e}_1 \Leftarrow \texttt{num} \qquad \dot{\Gamma} \vdash \dot{e}_2 \Leftarrow \texttt{num}}{\dot{\Gamma} \vdash \dot{e}_1 + \dot{e}_2 \Rightarrow \texttt{num}}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \dot{e} : \dot{\tau} \Rightarrow \dot{\tau}}$$

most of the rules are standard, here are a couple representative ones that should look familiar.

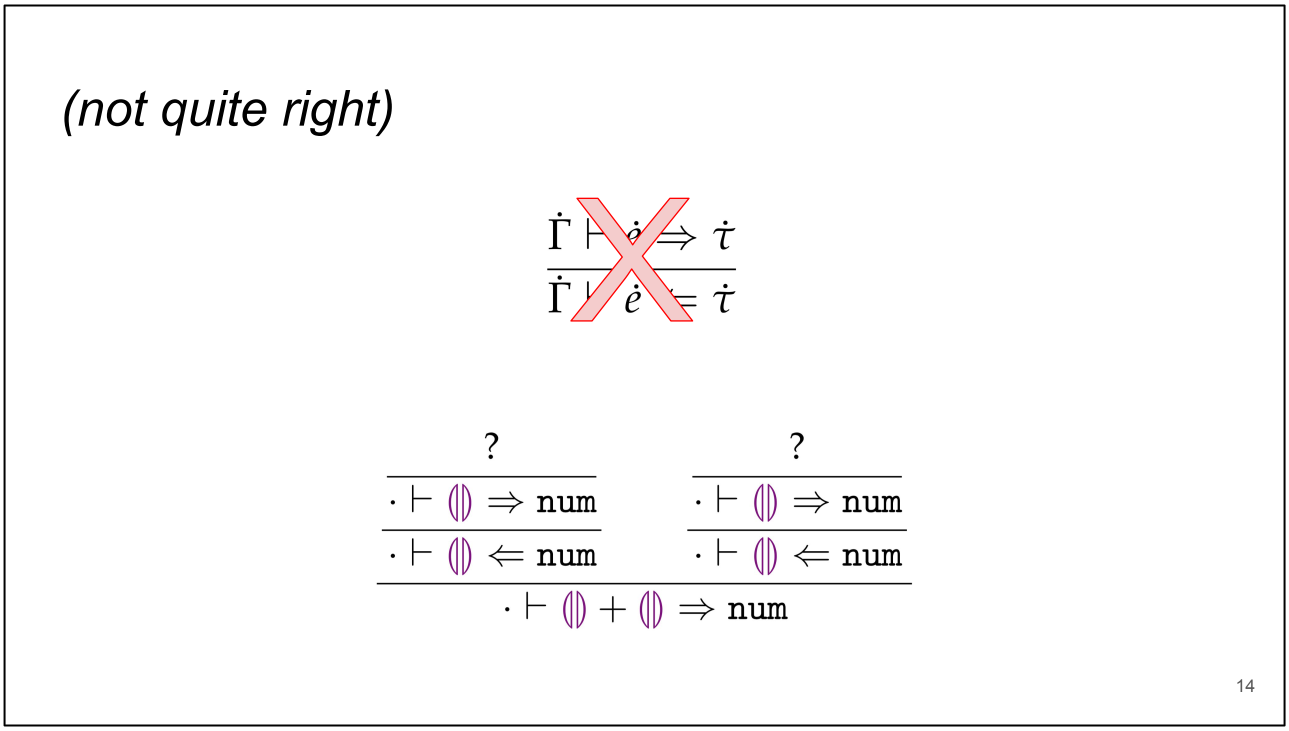


$$\frac{}{\dot{\Gamma} \vdash \llparenthesis\rrparenthesis \Rightarrow \llparenthesis\rrparenthesis}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}}{\dot{\Gamma} \vdash \llparenthesis\dot{e}\rrparenthesis \Rightarrow \llparenthesis\rrparenthesis}$$

the less standard rules are the ones that describe holes and hole types. here we say that the hole has hole type, but if it’s non-empty then the expression inside must have any type. this is how we use the hole type to track exactly where the incomplete parts of the program are.

because holes track places where a program is still under construction, we call an expression that does not use these forms “complete”



(not quite right)

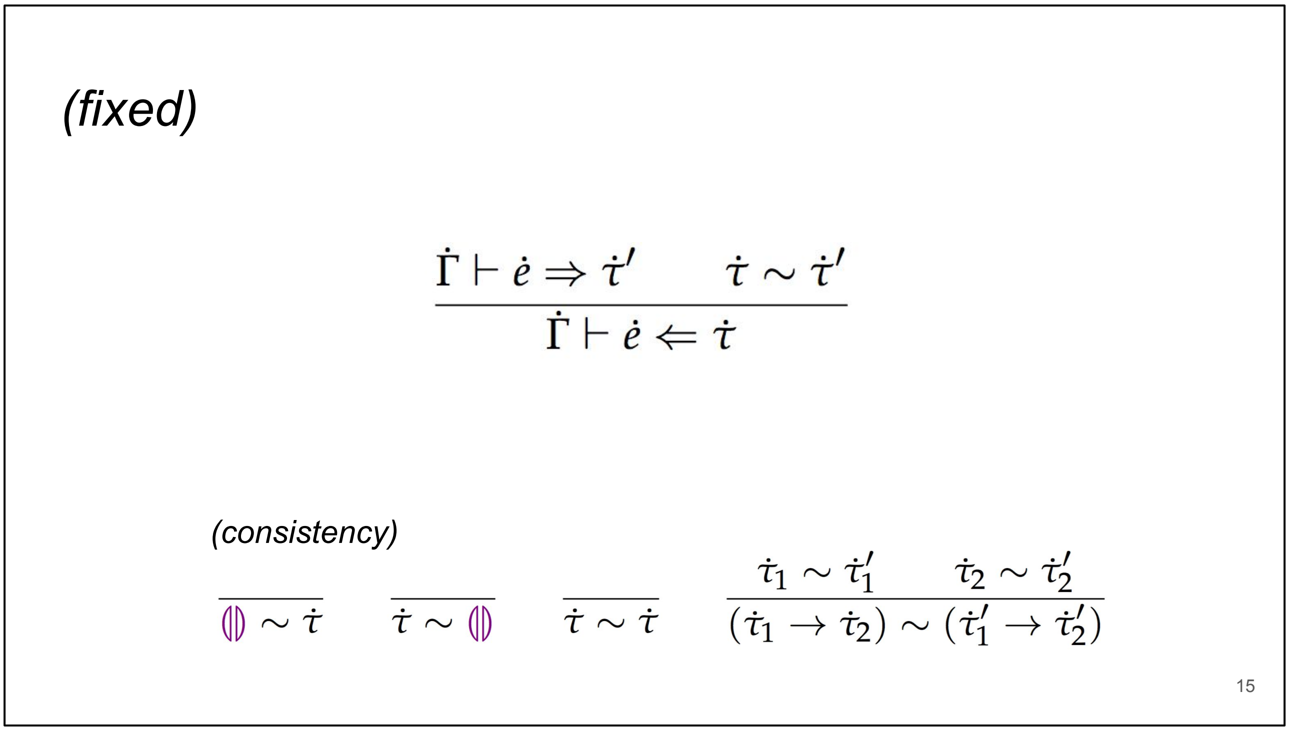
$$\xcancel{\frac{\dot\Gamma \vdash \dot e \Rightarrow \dot\tau}{\dot\Gamma \vdash \dot e \Leftarrow \dot\tau}}$$

$$\dfrac{\dfrac{\dfrac{?}{\cdot \vdash \llparenthesis\rrparenthesis \Rightarrow \texttt{num}}}{\cdot \vdash \llparenthesis\rrparenthesis \Leftarrow \texttt{num}} \qquad \dfrac{\dfrac{?}{\cdot \vdash \llparenthesis\rrparenthesis \Rightarrow \texttt{num}}}{\cdot \vdash \llparenthesis\rrparenthesis \Leftarrow \texttt{num}}}{\cdot \vdash \llparenthesis\rrparenthesis + \llparenthesis\rrparenthesis \Rightarrow \texttt{num}}$$

the main structural rule that you'd expect to see in a bidirectional system is subsumption, that lets you switch from synthesis to analysis. in the presence of holes this normal rule (shown here) is not unsound, but it also doesn't really capture the notion of a hole that we have intuitively. it's not unsound, but it is incomplete, so here's an example of something we'd like to be able to derive but cannot.

if we go back to our example, consider giving a type to just a small part of that expression, (||) + (||). by the + rule, it should synthesize num, because that's what the form + always means.

but it doesn't -- we get stuck!



(fixed)

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

(consistency)

$$\frac{}{\llparenthesis\rrparenthesis \sim \dot{\tau}} \qquad \frac{}{\dot{\tau} \sim \llparenthesis\rrparenthesis} \qquad \frac{}{\dot{\tau} \sim \dot{\tau}} \qquad \frac{\dot{\tau}_1 \sim \dot{\tau}_1' \qquad \dot{\tau}_2 \sim \dot{\tau}_2'}{(\dot{\tau}_1 \to \dot{\tau}_2) \sim (\dot{\tau}_1' \to \dot{\tau}_2')}$$

instead, we have to allow the subsumption rule to accept types up a weaker relation than equality. this relation is usually called type consistency-- it captures the notion that we're willing to accept hole instead where ever we expected to see a complete type.

if this looks familiar to you, it's because the same judgement form appears in the foundational gradual typing work by Siek and Taha to describe the type ?.

and that should make sense: gradual typing attempts to codify **"the intuition that the structure of a type may be partially known or unknown at compile time and the job of the type system is to catch incompatibilities"**

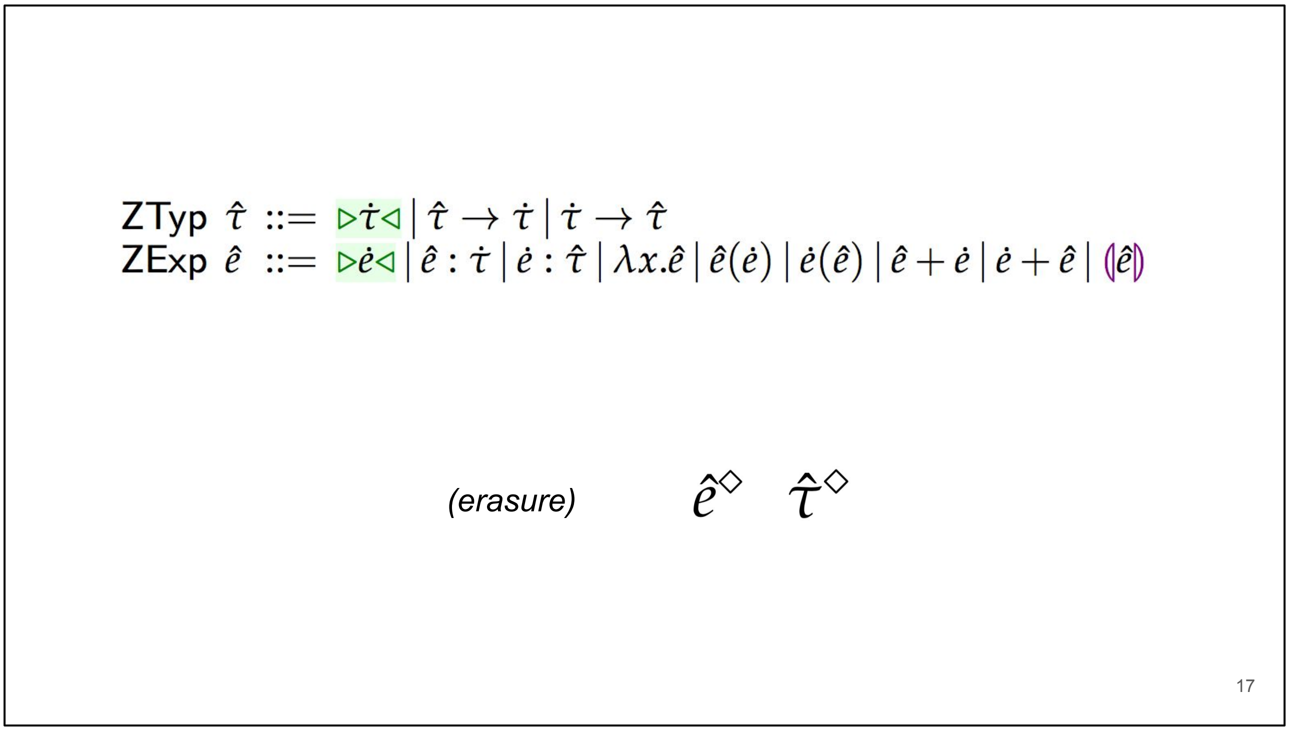
with complete expressions, this is pretty hard to swallow philosophically. but! if you're trying to statically understand each step of a program as it's written and evolves over time, you really do not have a full derivation of the type at compile time. the program isn't done yet! so it's natural here.

Action Semantics and Metatheory

so far i’ve given enough to describe the core language -- just the shape of the terms on each line of the example, including the fancy purple but not yet the green.

now i’m going to describe actions on hexpressions, and the metatheory that argues that for the rules of actions. these were designed at the same time, so i’ll interleave them as appropriate.

and remember that all this is supposed to be **EXTENSIBLE**. so it should be easy to add new things to the language, and the metatheory should guide us in doing that.



$$\text{ZTyp}\ \hat{\tau} ::= \triangleright\dot{\tau}\triangleleft \mid \hat{\tau} \to \dot{\tau} \mid \dot{\tau} \to \hat{\tau}$$

$$\text{ZExp}\ \hat{e} ::= \triangleright\dot{e}\triangleleft \mid \hat{e} : \dot{\tau} \mid \dot{e} : \hat{\tau} \mid \lambda x.\hat{e} \mid \hat{e}(\dot{e}) \mid \dot{e}(\hat{e}) \mid \hat{e} + \dot{e} \mid \dot{e} + \hat{e} \mid (\!|\hat{e}|\!)$$

(erasure) $\hat{e}^{\diamond}$ $\hat{\tau}^{\diamond}$

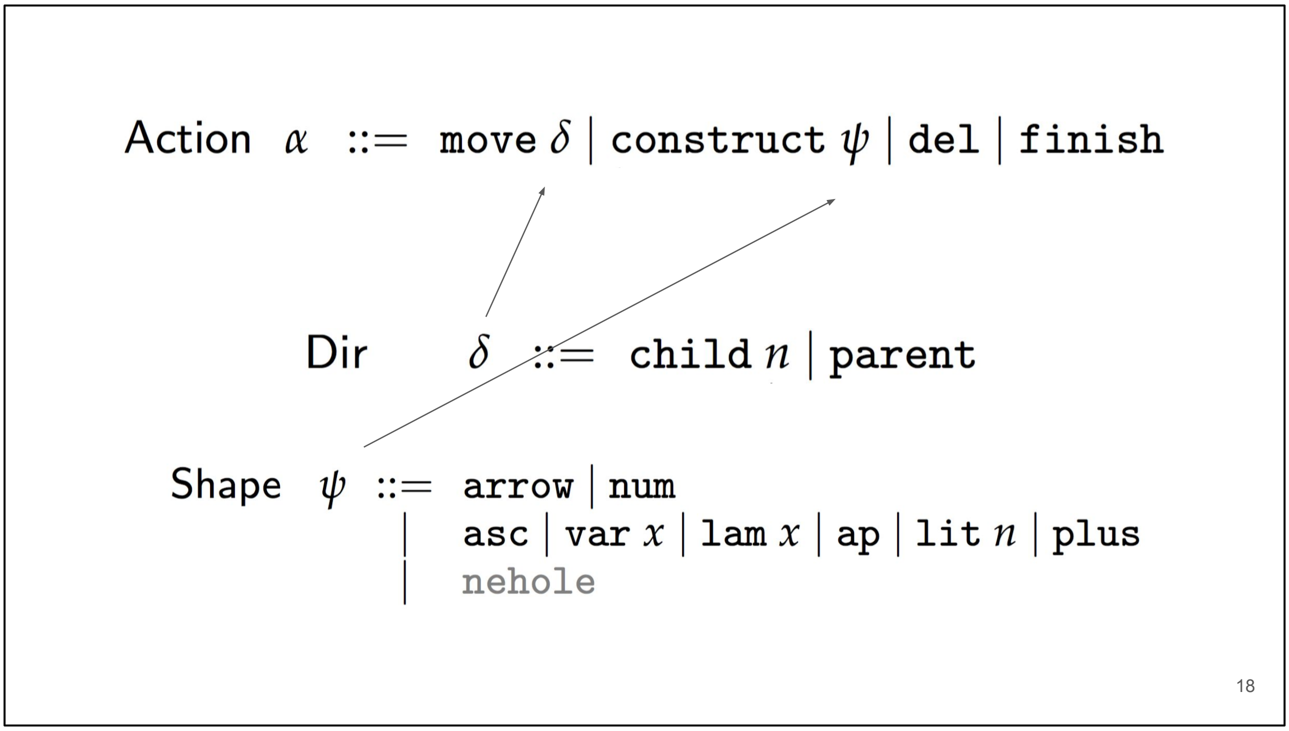
edits on an hexpression are made **in a particular place**. so we superimpose a single cursor on htypes and hxpressions (dotted) to get ztypes and zexps (hatted)

here the cursor is the thing in green. the base cases of the grammar identify a particular hexp / htype (dotted)that the cursor is on; the inductive forms each identify exactly one of the subtrees of the AST as containing the cursor (hatted), while all others do not.

if you've seen it, this is the zipper pattern from Huet's functional pearl -- which, perhaps not coincidentally, he encountered while writing a structure editor.

h-types because of holes; ztypes because of zippers

erasure is defined by the obvious structural recursion on the form of the z(exp/type). we write this with a little superscript diamond. we need to do this just because typing rules only type terms without a cursor in them.



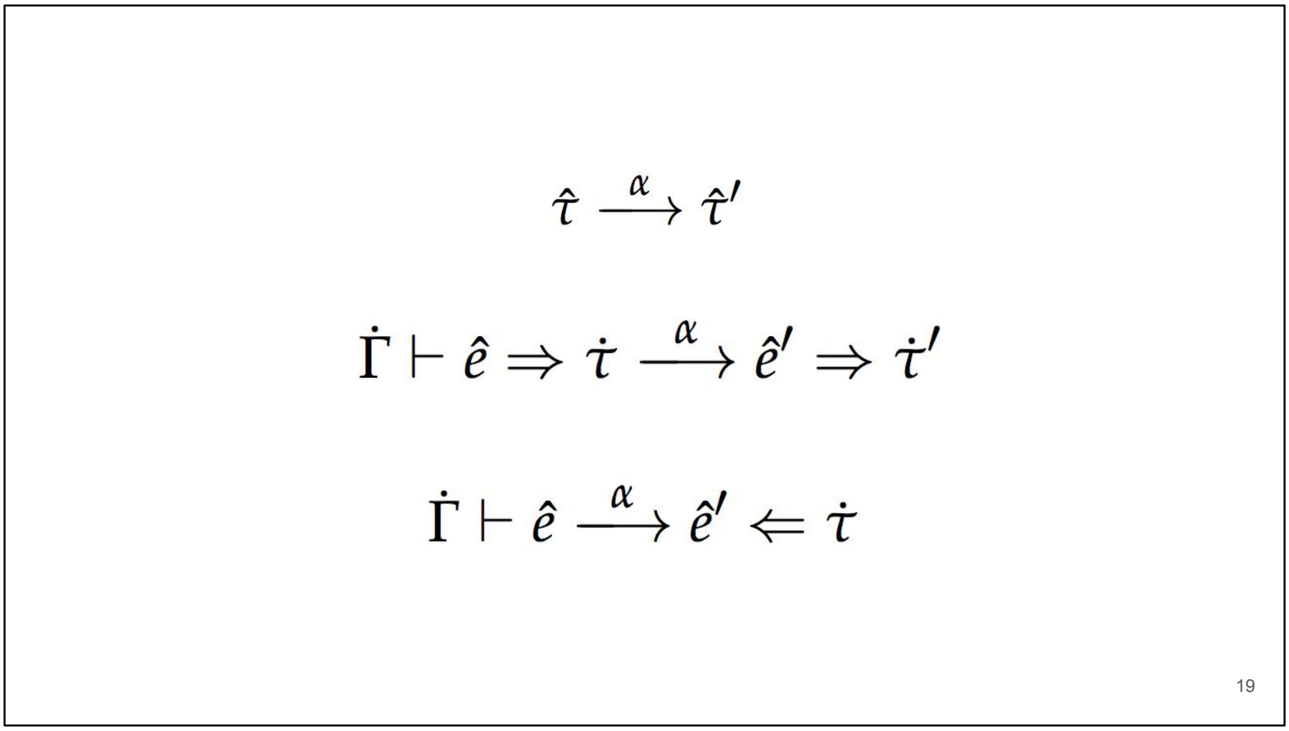
$$\begin{array}{llcl}
\text{Action} & \alpha & ::= & \mathtt{move}\,\delta \mid \mathtt{construct}\,\psi \mid \mathtt{del} \mid \mathtt{finish} \\
\text{Dir} & \delta & ::= & \mathtt{child}\,n \mid \mathtt{parent} \\
\text{Shape} & \psi & ::= & \mathtt{arrow} \mid \mathtt{num} \\
 & & \mid & \mathtt{asc} \mid \mathtt{var}\,x \mid \mathtt{lam}\,x \mid \mathtt{ap} \mid \mathtt{lit}\,n \mid \mathtt{plus} \\
 & & \mid & \mathtt{nehole}
\end{array}$$

the system i’ve described so far is enough to give a static semantics to edit states. it does not describe how to ***move between* edit states**

to that end, there are four kinds of actions given by this grammar:

- move moves the cursor in a direction (delta);
- construct builds a new term when given its name;
- delete replaces whatever is under the cursor with a hole;
- finish replaces a non-empty hole with its contents ONLY IF the type of the contents matches the context.

incidentally, this is the last key component in our terrible acronym pun: Holes Actions and Zippers in an expression language give you HAZ-el. and “nut” because it’s like the core small thing.



$$\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'$$

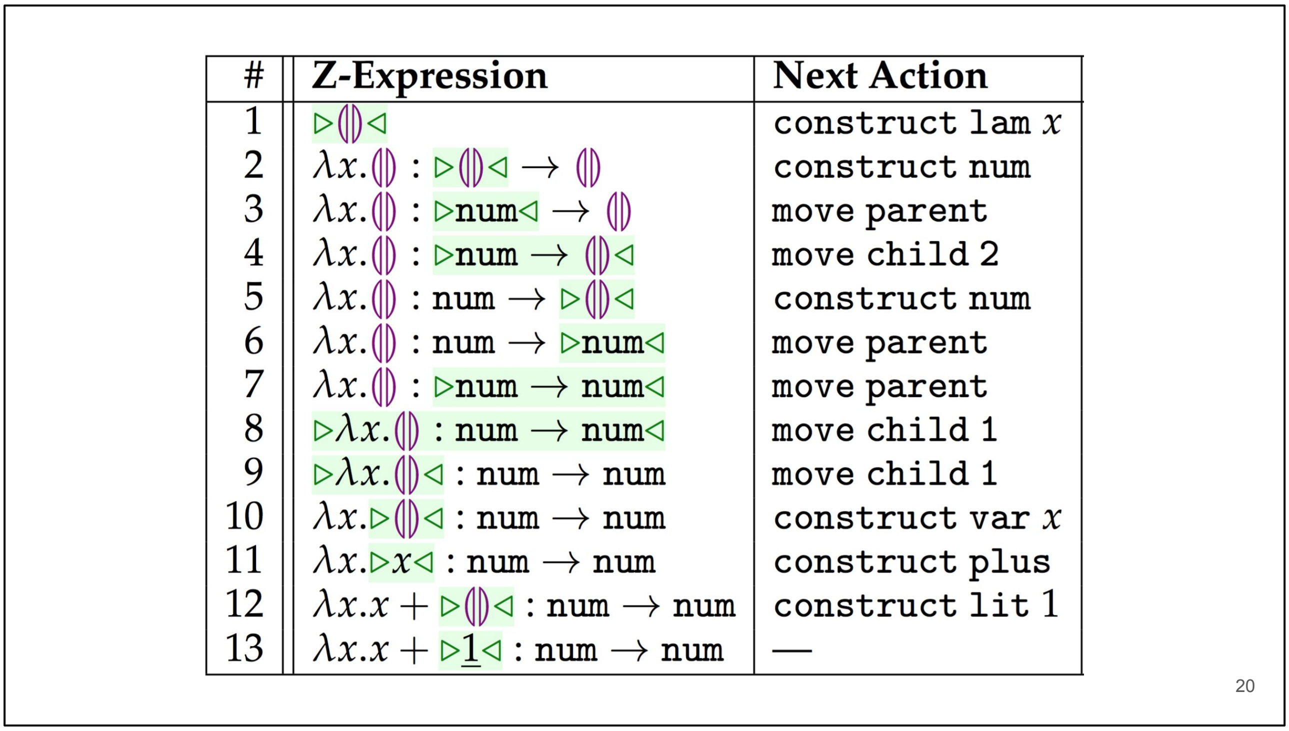
$$\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$$

$$\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}$$

the rules that define how these actions act on programs are given in three judgements based on where the action is occurs**, following the bidirectional shape**

if \alpha is some action, there are three judgements that describe how \alpha moves from one edit state to another:

- performing alpha on t gets you t’
- alpha on e that synthesizes t gets you e’ that synthesizes t’, which **might be different depending on what you make**
- alpha on e while checking against t gets you e’, still at t because the type is **pinned by the ana**



#	Z-Expression	Next Action
1	$\triangleright\llparenthesis\rrparenthesis\triangleleft$	`construct lam` x
2	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft \to \llparenthesis\rrparenthesis$	`construct num`
3	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\mathtt{num}\triangleleft \to \llparenthesis\rrparenthesis$	`move parent`
4	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\mathtt{num} \to \llparenthesis\rrparenthesis\triangleleft$	`move child 2`
5	$\lambda x.\llparenthesis\rrparenthesis : \mathtt{num} \to \triangleright\llparenthesis\rrparenthesis\triangleleft$	`construct num`
6	$\lambda x.\llparenthesis\rrparenthesis : \mathtt{num} \to \triangleright\mathtt{num}\triangleleft$	`move parent`
7	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\mathtt{num} \to \mathtt{num}\triangleleft$	`move parent`
8	$\triangleright\lambda x.\llparenthesis\rrparenthesis : \mathtt{num} \to \mathtt{num}\triangleleft$	`move child 1`
9	$\triangleright\lambda x.\llparenthesis\rrparenthesis\triangleleft : \mathtt{num} \to \mathtt{num}$	`move child 1`
10	$\lambda x.\triangleright\llparenthesis\rrparenthesis\triangleleft : \mathtt{num} \to \mathtt{num}$	`construct var` x
11	$\lambda x.\triangleright x\triangleleft : \mathtt{num} \to \mathtt{num}$	`construct plus`
12	$\lambda x.x + \triangleright\llparenthesis\rrparenthesis\triangleleft : \mathtt{num} \to \mathtt{num}$	`construct lit 1`
13	$\lambda x.x + \triangleright\underline{1}\triangleleft : \mathtt{num} \to \mathtt{num}$	—

revisit the example from the top of the talk quickly.

we have enough now to recognize the forms of the zexpressions in the left column, and the names of the actions on the right. we know that the judgmental forms will describe the semantics of how to go between rows, but we haven’t really seen any of the rules yet.

Sensibility

edit actions preserve well-typedness

1. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$ *then* $\dot{\Gamma} \vdash \hat{e}'^{\diamond} \Rightarrow \dot{\tau}'$.
2. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}$ *then* $\dot{\Gamma} \vdash \hat{e}'^{\diamond} \Leftarrow \dot{\tau}$.

before i go through the rules for the semantics judgements, though, i want to state our first theorem. this is the main point of the whole system, and therefore a design principle i want you to keep in mind as i go through the rules.

anything that you might want to do in a rule that would break sensibility is strictly verboten.

our goal is to have a system where each edit state can be given meaning statically. that is to say: edit actions preserve well typedness.

this is the theorem that lets us reason about how we move between rows

Move

$$\frac{}{\triangleright\dot{\tau}_1 \rightarrow \dot{\tau}_2\triangleleft \xrightarrow{\texttt{move child 1}} \triangleright\dot{\tau}_1\triangleleft \rightarrow \dot{\tau}_2}$$

$$\frac{}{\triangleright\dot{\tau}_1 \rightarrow \dot{\tau}_2\triangleleft \xrightarrow{\texttt{move child 2}} \dot{\tau}_1 \rightarrow \triangleright\dot{\tau}_2\triangleleft}$$

$$\frac{}{\triangleright\dot{\tau}_1\triangleleft \rightarrow \dot{\tau}_2 \xrightarrow{\texttt{move parent}} \triangleright\dot{\tau}_1 \rightarrow \dot{\tau}_2\triangleleft}$$

$$\frac{}{\dot{\tau}_1 \rightarrow \triangleright\dot{\tau}_2\triangleleft \xrightarrow{\texttt{move parent}} \triangleright\dot{\tau}_1 \rightarrow \dot{\tau}_2\triangleleft}$$

a few example movement actions -- just directions in the AST.

they're nice because they are pretty darn mechanical and go more or less like you'd think. this is just a few.

they're also not so nice because they're so mechanical. there's a real worry that you'll make a mistake.

so what could go wrong? how could we stray from our intent? what's the theorem that checks this?

Movement Erasure

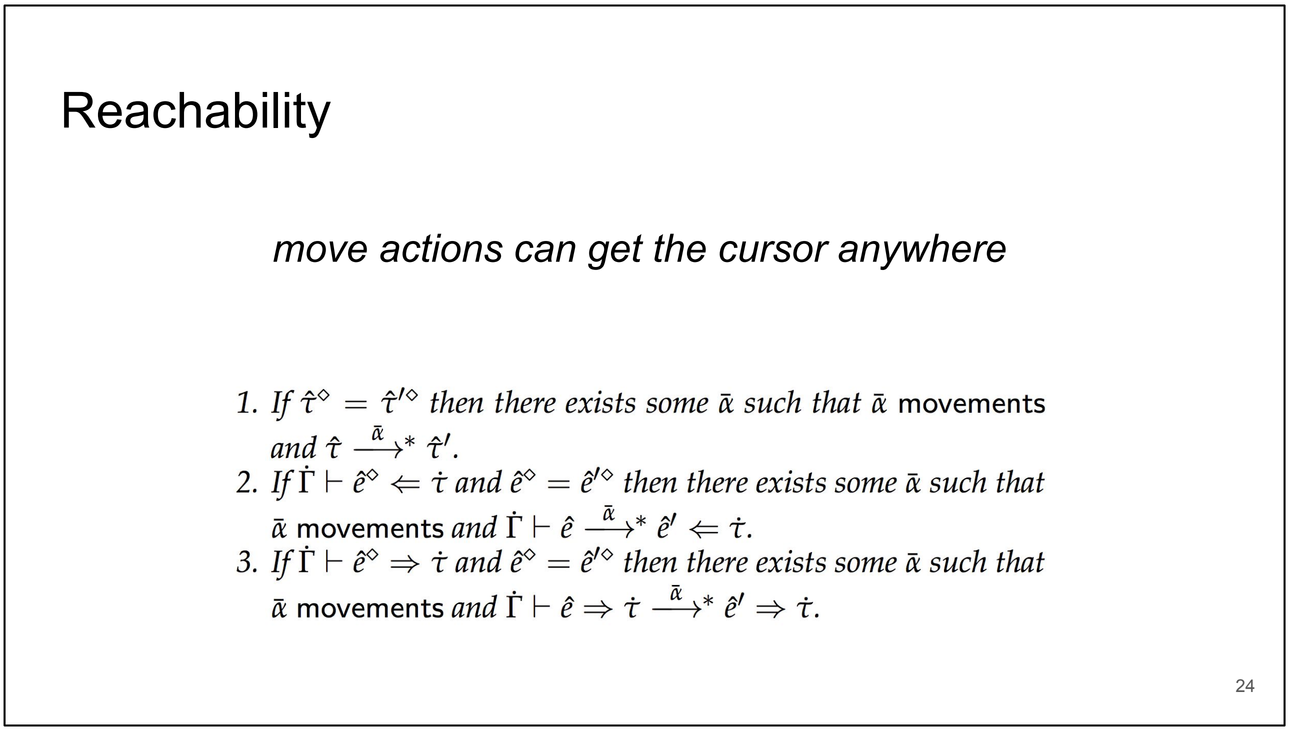
move actions only move the cursor

1. *If* $\hat{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{\tau}'$ *then* $\hat{\tau}^{\diamond} = \hat{\tau}'^{\diamond}$.
2. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Rightarrow \dot{\tau}'$ *then* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$ *and* $\dot{\tau} = \dot{\tau}'$.
3. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Leftarrow \dot{\tau}$ *then* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$.

checksum

well. you might do something other than just move the cursor -- turn everything into 1 or flipping arguments or whatever. some but not all of those kinds of errors would cause sensibility to fail.

more technically, you want to say that after any move action in any judgement, the erasure is the same.



Reachability

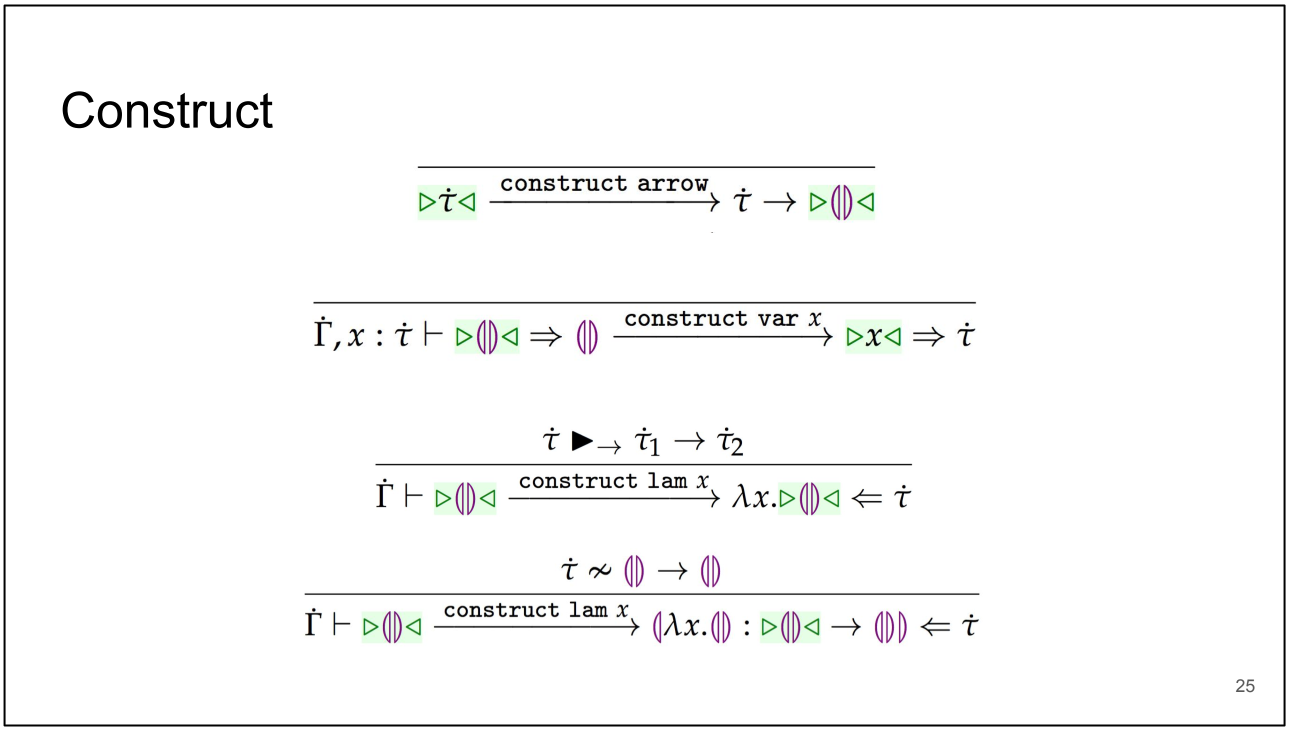
move actions can get the cursor anywhere

1. *If* $\hat{\tau}^{\diamond} = \hat{\tau}'^{\diamond}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\hat{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \hat{\tau}'$.
2. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\bar{\alpha}}{}^{*} \hat{e}' \Leftarrow \dot{\tau}$.
3. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \hat{e}' \Rightarrow \dot{\tau}$.

the other thing that could go wrong is that you might not have enough movement rules, you might forget some.

so we also have a checksum thrm that says that you can get everywhere -- if two terms agree on their erasure there's a sequence of just movement actions that brings you from one to the other

(note that --->* is just standard lifting of ---> to the reflexive transitive closure)



Construct

$$\frac{}{\triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{construct arrow}} \dot{\tau} \to \triangleright\llparenthesis\rrparenthesis\triangleleft}$$

$$\frac{}{\dot{\Gamma}, x : \dot{\tau} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \Rightarrow \llparenthesis\rrparenthesis \xrightarrow{\texttt{construct var } x} \triangleright x\triangleleft \Rightarrow \dot{\tau}}$$

$$\frac{\dot{\tau} \blacktriangleright_{\to} \dot{\tau}_1 \to \dot{\tau}_2}{\dot{\Gamma} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct lam } x} \lambda x.\triangleright\llparenthesis\rrparenthesis\triangleleft \Leftarrow \dot{\tau}}$$

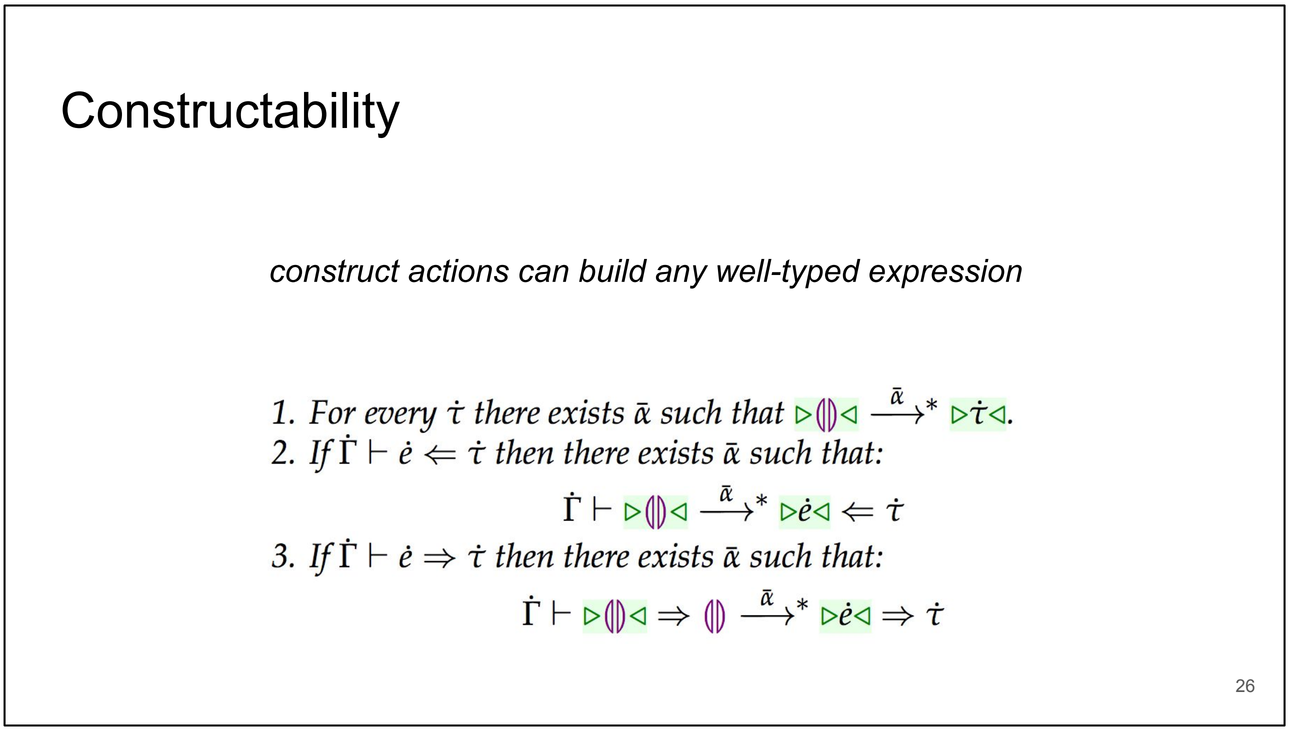
$$\frac{\dot{\tau} \nsim \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis}{\dot{\Gamma} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct lam } x} \llparenthesis\lambda x.\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft \to \llparenthesis\rrparenthesis\rrparenthesis \Leftarrow \dot{\tau}}$$

25

here are some example construction rules:

- one on how to build types, pretty obvious.
- if you have a var in the context, and are trying to synthesize anything, you're allowed to use it and synth that type.
- this one makes good on our promise about not-outside-in. if you want to construct a lambda but you're in a situation where you know the type that's been propagated down disagrees with that, we construct a lambda but put that lambda inside a hole.

what's the checksum here? well. the rules are less boring, so there's not that danger, but we could easily forget something. so we need to argue that we have *enough* construct rules.


Constructability

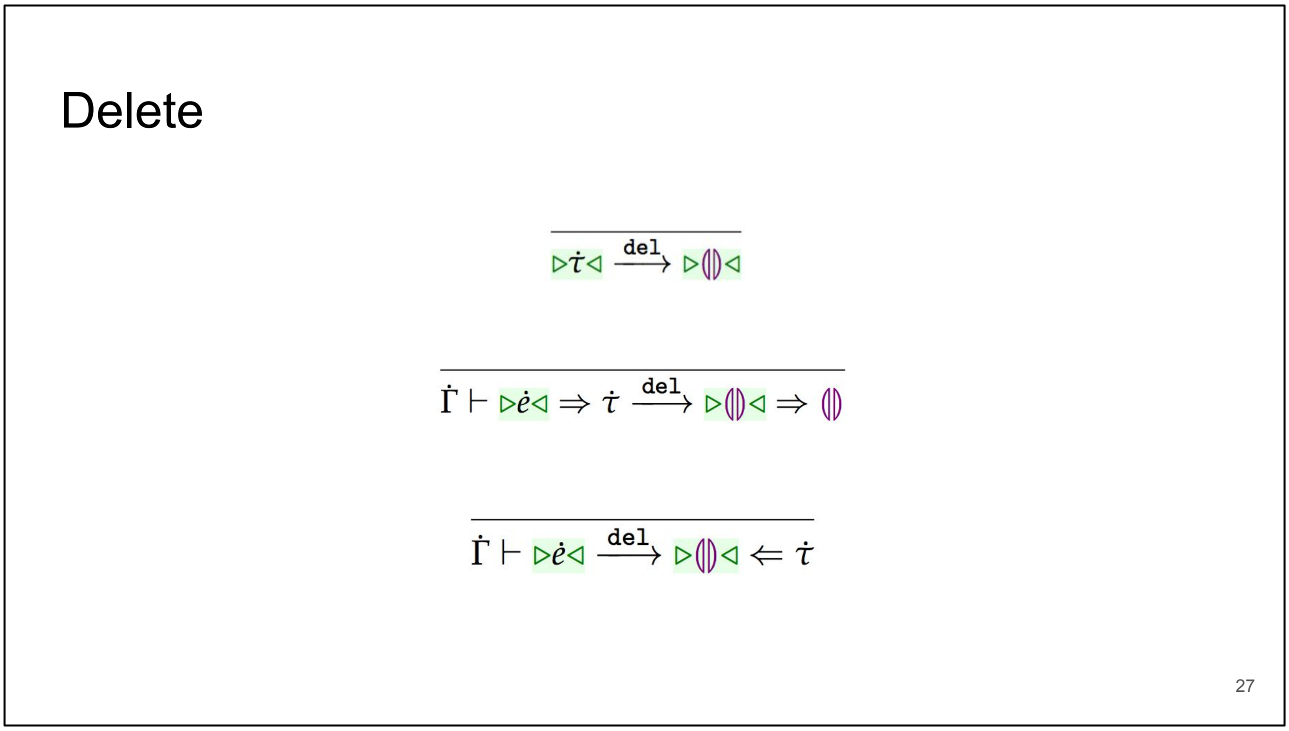
construct actions can build any well-typed expression

1. *For every* $\dot{\tau}$ *there exists* $\bar{\alpha}$ *such that* $\triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\bar{\alpha}}{}^{*} \triangleright\dot{\tau}\triangleleft$.
2. *If* $\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}$ *then there exists* $\bar{\alpha}$ *such that:*
$$\dot{\Gamma} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\bar{\alpha}}{}^{*} \triangleright\dot{e}\triangleleft \Leftarrow \dot{\tau}$$
3. *If* $\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}$ *then there exists* $\bar{\alpha}$ *such that:*
$$\dot{\Gamma} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \Rightarrow \llparenthesis\rrparenthesis \xrightarrow{\bar{\alpha}}{}^{*} \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau}$$

(note that this doesn’t say you have all that you might want .. just that you didn’t forget anything. so there’s still some wiggle room here.)

not just a checksum! this is also a feature that you would want for your editor and we can really prove that you have it with this collection of rules.

the implementation of this theorem therefore coughs up a witness. we have a fairly naive one that is sufficient to show the checksum goes through; could imagine some cool engineering effort to produce smallest trace or other things.



Delete

$$\frac{}{\triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{del}} \triangleright(\!|\,|\!)\triangleleft}$$

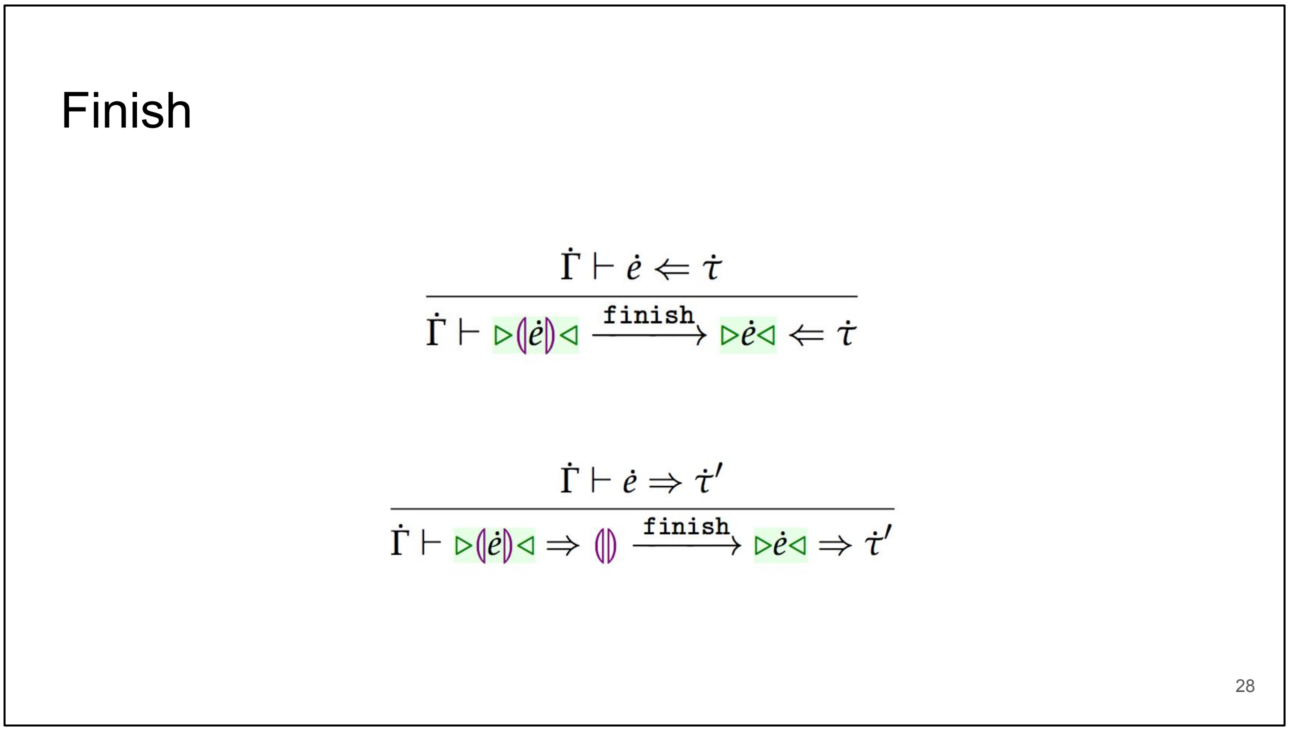
$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{del}} \triangleright(\!|\,|\!)\triangleleft \Rightarrow (\!|\,|\!)}$$

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \xrightarrow{\texttt{del}} \triangleright(\!|\,|\!)\triangleleft \Leftarrow \dot{\tau}}$$

the delete action makes whatever is in the cursor into a hole.

what can we prove about these rules as a checksum?

the motivation for the other theorems has been that, hey, you can really mess these rules up. because these rules **do not inspect the structure of the terms or types** once you believe them you’re pretty much done. and they’ll survive to any extension.



Finish

$$\frac{\dot\Gamma \vdash \dot e \Leftarrow \dot\tau}{\dot\Gamma \vdash \triangleright(\!|\dot e|\!)\triangleleft \xrightarrow{\texttt{finish}} \triangleright\dot e\triangleleft \Leftarrow \dot\tau}$$

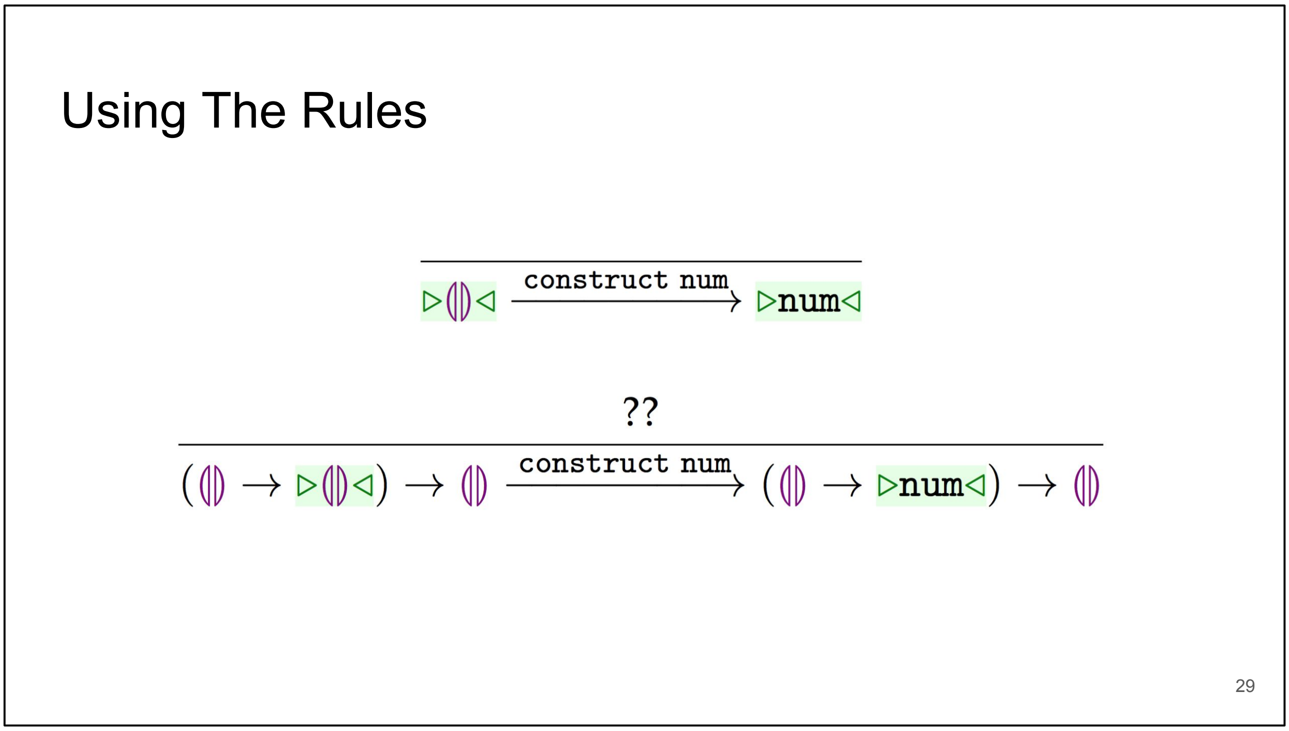
$$\frac{\dot\Gamma \vdash \dot e \Rightarrow \dot\tau'}{\dot\Gamma \vdash \triangleright(\!|\dot e|\!)\triangleleft \Rightarrow (\!|\,|\!) \xrightarrow{\texttt{finish}} \triangleright\dot e\triangleleft \Rightarrow \dot\tau'}$$

(no finish rule for types because non-empty type holes don’t appear)

finish replaces a non-empty hole in the cursor with its contents --- if the type of the contents matches the type required by the context. this is what we mean by “delaying the type consistency check” -- and we delay it until the programmer says not to.

note that these rules absolutely might not fire -- you can write junk code til the cows come home, it just can’t get finished.

again, because finish doesn’t inspect the structure of its contents, there’s no particular checksum theorem for these rules. they’ll work for any extension to the language as-is.



Using The Rules

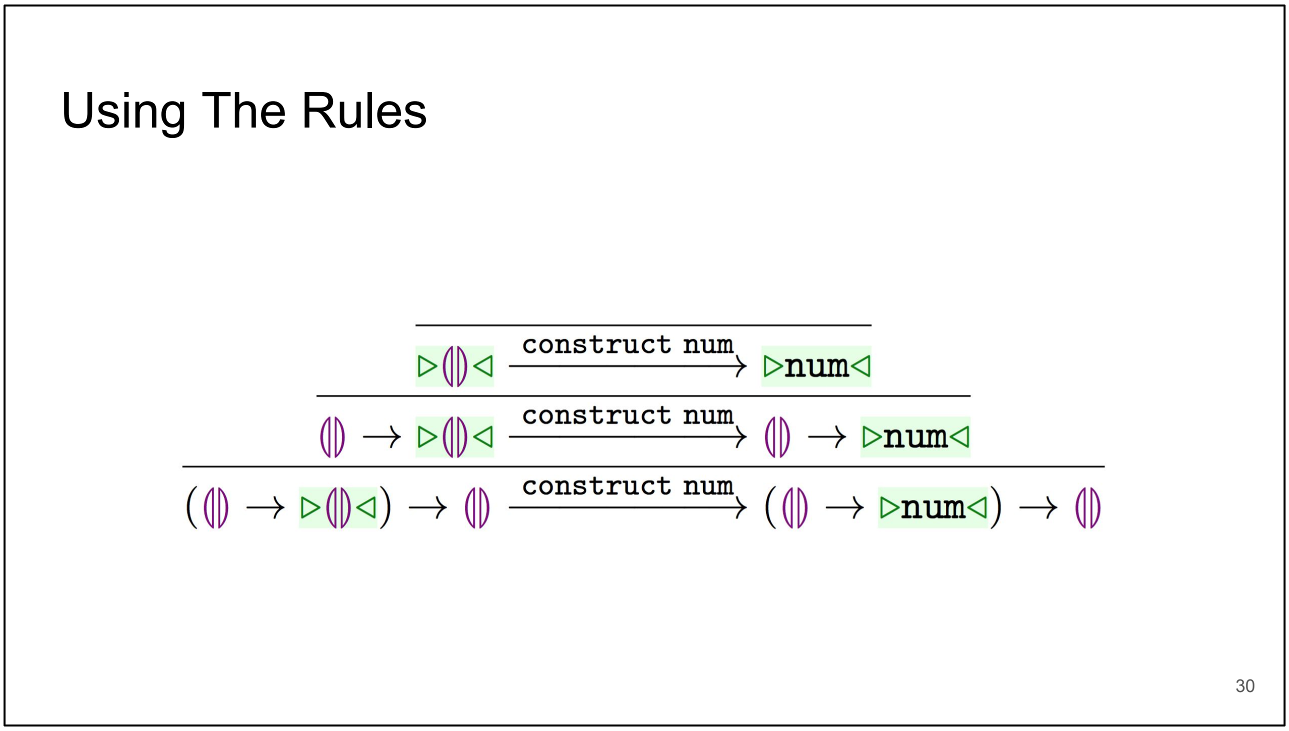
$$\frac{}{\triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct num}} \triangleright\texttt{num}\triangleleft}$$

$$\frac{??}{(\llparenthesis\rrparenthesis \to \triangleright\llparenthesis\rrparenthesis\triangleleft) \to \llparenthesis\rrparenthesis \xrightarrow{\texttt{construct num}} (\llparenthesis\rrparenthesis \to \triangleright\texttt{num}\triangleleft) \to \llparenthesis\rrparenthesis}$$

so there’s one detail that’s missing here. the rules i’ve shown you are really kind of just the base cases. to see what’s missing, let’s imagine trying to find a derivation for just a little part of the example, where you want to change one hole in an arrow type to `num`.

so you get stuck immediately, right?

the rule you *want* to apply is (show it) but the bottoms don’t even unify because the cursor is deep inside this other expression.



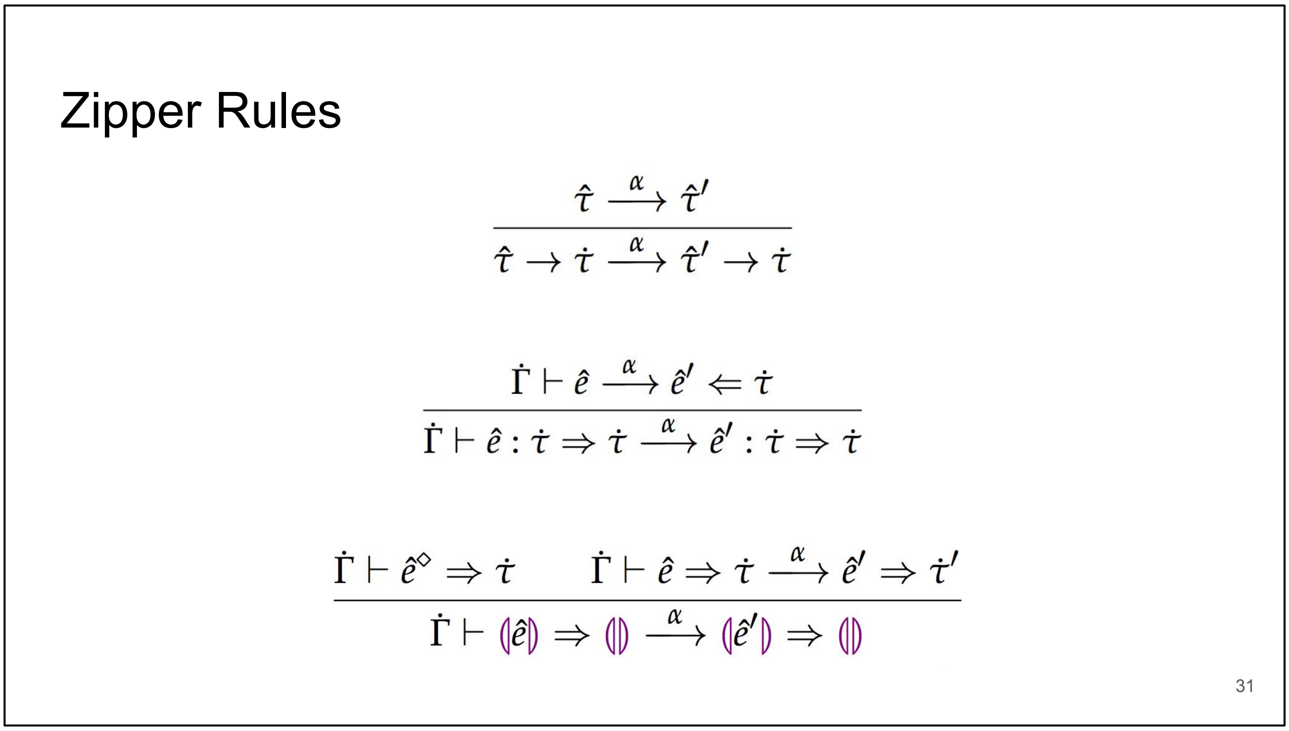
Using The Rules

$$\dfrac{\dfrac{\dfrac{}{\triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct num}} \triangleright\texttt{num}\triangleleft}}{\llparenthesis\rrparenthesis \to \triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct num}} \llparenthesis\rrparenthesis \to \triangleright\texttt{num}\triangleleft}}{(\llparenthesis\rrparenthesis \to \triangleright\llparenthesis\rrparenthesis\triangleleft) \to \llparenthesis\rrparenthesis \xrightarrow{\texttt{construct num}} (\llparenthesis\rrparenthesis \to \triangleright\texttt{num}\triangleleft) \to \llparenthesis\rrparenthesis}$$

what you need is a rule that says that if a change happens at some subexpression, it propagates through the zipper structure, so you can write something like this.

you have to unzip the expression down to the place you want to apply one of the local rules, apply the rule, and then zip it back up again and make sure that everything still fits together.

you also need to reflect the zipper pattern at the judgemental level of the action semantics, or else you can’t ever apply the local rules you want. (this is the flip side to being able to state rules that are simple and cursor-and-type-directed; but it’s ok because the zipper rules are pretty mechanical)



Zipper Rules

$$\frac{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'}{\hat{\tau} \to \dot{\tau} \xrightarrow{\alpha} \hat{\tau}' \to \dot{\tau}}$$

$$\frac{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \hat{e} : \dot{\tau} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' : \dot{\tau} \Rightarrow \dot{\tau}}$$

$$\frac{\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau} \qquad \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'}{\dot{\Gamma} \vdash \llparenthesis \hat{e} \rrparenthesis \Rightarrow \llparenthesis\rrparenthesis \xrightarrow{\alpha} \llparenthesis \hat{e}' \rrparenthesis \Rightarrow \llparenthesis\rrparenthesis}$$

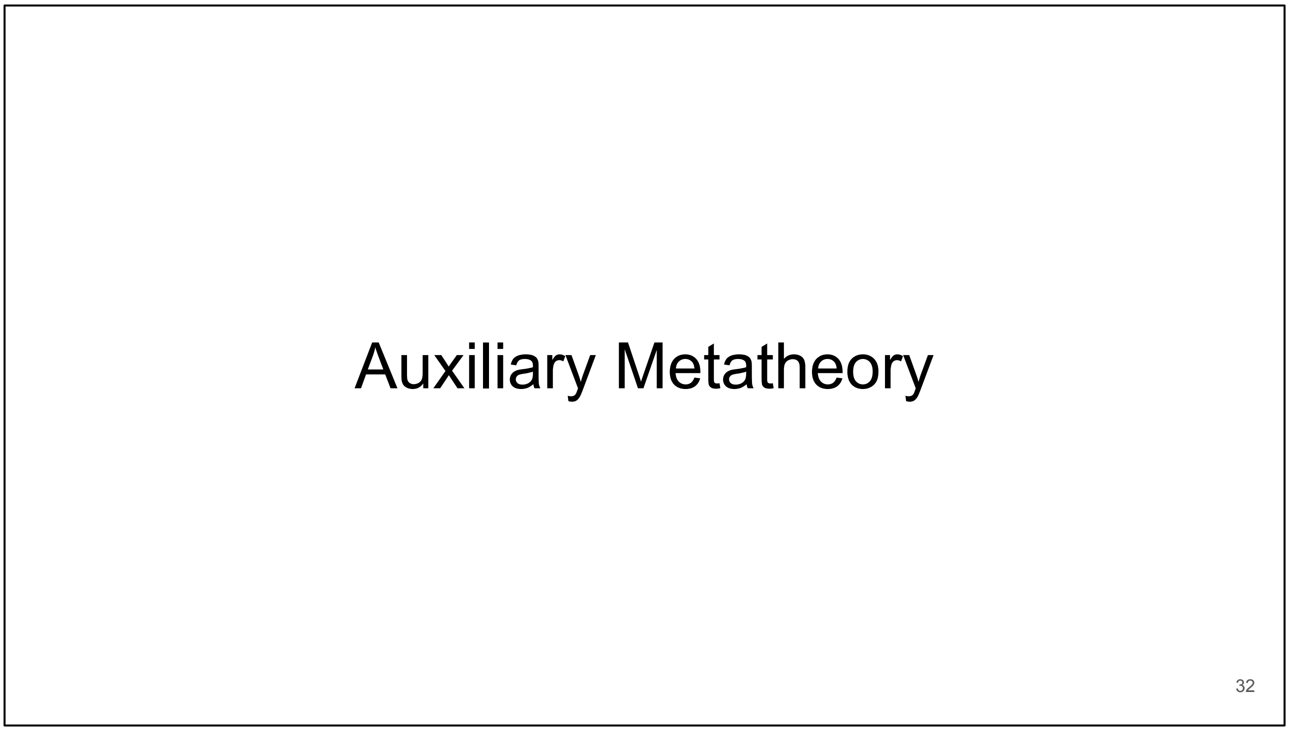
here are a few representative sample rules of how that goes; it follows the structure of the grammar of zexps very closely.

note that in some cases, like the rule shown for non-empty holes, we have to enforce that the expression be well-typed. sensibility us that e' will be as well --- but only if the input to the action is. leaving out premises like this would lead to soundness problems.

how do we have a checksum for these rules?

well. to prove anything interesting you need to use all of them. so actually the proofs of constructability and reachability demonstrate that these rules are sufficient, by proving a suite of lemmas to lift them to the RTC of the action judgements.

so we already did it!


Auxiliary Metatheory

so all the theorems that i’ve talked about so far have either been design principles or checksums that really show that we’re doing what we’ve said we’re doing. (that **features of the editor really are** what they claim to be)

one really nice thing about having a mechanized metatheory is that you can also have theorems that you just dream up and then check to see if they really are true.

Determinism

actions result in only one edit state

$$\textit{If } \hat{\tau} \xrightarrow{\alpha} \hat{\tau}' \textit{ and } \hat{\tau} \xrightarrow{\alpha} \hat{\tau}'' \textit{ then } \hat{\tau}' = \hat{\tau}''$$

here's one.

you might well believe me if i told you that the action semantics are deterministic, that is to say that applying a fixed action to a fixed edit state results in only one other expression.

it's true for actions on types, but turns out to be false for expressions.

(we were surprised too!)

Determinism *(expressions)*

$$\frac{}{\cdot \vdash \triangleright(\!|\,|\!)\triangleleft \xrightarrow{\texttt{construct lam } x} \lambda x.(\!|\,|\!) \Leftarrow \texttt{num} \to \texttt{num}}$$

$$\frac{\dfrac{}{\cdot \vdash \triangleright(\!|\,|\!)\triangleleft \Rightarrow (\!|\,|\!) \xrightarrow{\texttt{construct lam } x} (\lambda x.(\!|\,|\!) : \triangleright(\!|\,|\!)\triangleleft \to (\!|\,|\!)) \Rightarrow (\!|\,|\!) \to (\!|\,|\!)}}{\cdot \vdash \triangleright(\!|\,|\!)\triangleleft \xrightarrow{\texttt{construct lam } x} (\lambda x.(\!|\,|\!) : \triangleright(\!|\,|\!)\triangleleft \to (\!|\,|\!)) \Leftarrow \texttt{num} \to \texttt{num}}$$

here’s a little counter example for expressions

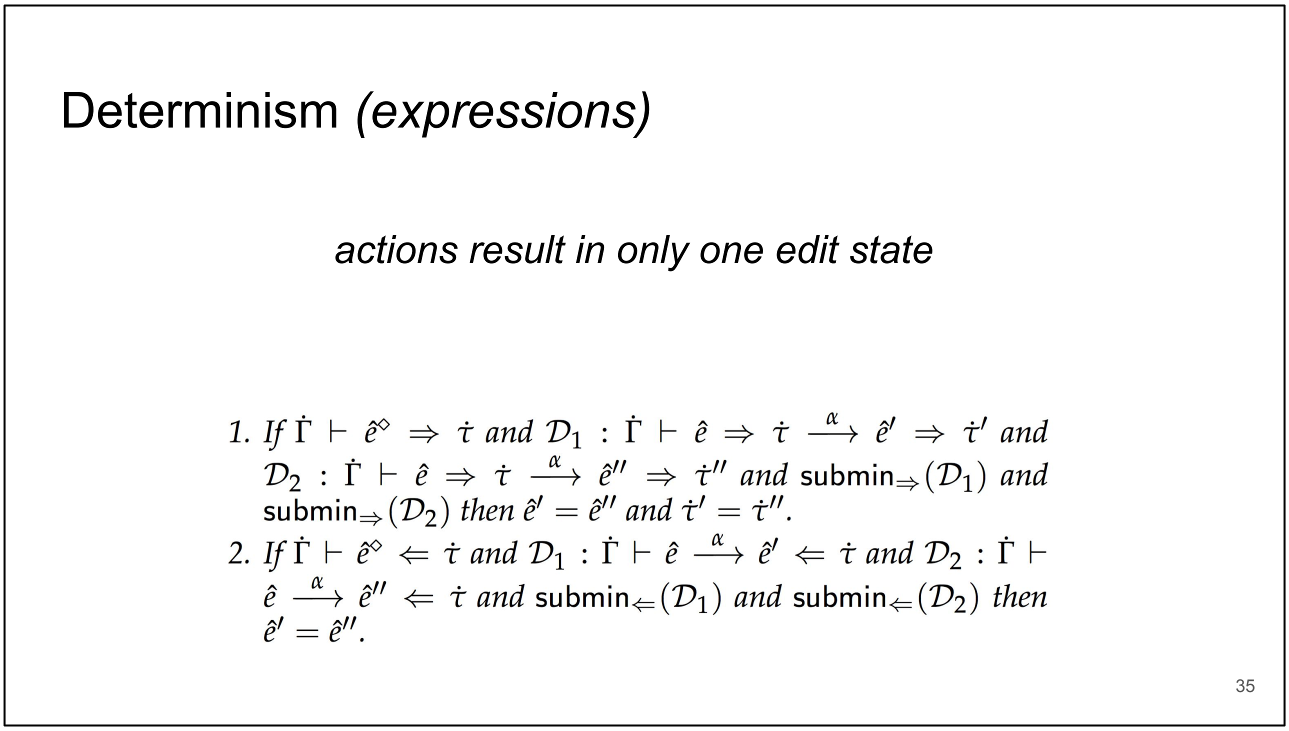
let’s say you’re an an analytic position, so you know that the type you’re pinned to is arrow. so you know that you do not need to have an ascription on a lambda.

but.

you could also apply the action rule that corresponds to subsumption to appeal to a synthetic rule.

in a synthetic position you have to have an ascription to be able to check a lambda -- so the syntheic rule adds one!

so you have two different derivations of the action in the conclusion and they do not produce the same expression.



Determinism *(expressions)*

actions result in only one edit state

1. *If* $\dot\Gamma \vdash \hat{e}^{\diamond} \Rightarrow \dot\tau$ *and* $\mathcal{D}_1 : \dot\Gamma \vdash \hat{e} \Rightarrow \dot\tau \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot\tau'$ *and* $\mathcal{D}_2 : \dot\Gamma \vdash \hat{e} \Rightarrow \dot\tau \xrightarrow{\alpha} \hat{e}'' \Rightarrow \dot\tau''$ *and* $\mathsf{submin}_{\Rightarrow}(\mathcal{D}_1)$ *and* $\mathsf{submin}_{\Rightarrow}(\mathcal{D}_2)$ *then* $\hat{e}' = \hat{e}''$ *and* $\dot\tau' = \dot\tau''$.
2. *If* $\dot\Gamma \vdash \hat{e}^{\diamond} \Leftarrow \dot\tau$ *and* $\mathcal{D}_1 : \dot\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot\tau$ *and* $\mathcal{D}_2 : \dot\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}'' \Leftarrow \dot\tau$ *and* $\mathsf{submin}_{\Leftarrow}(\mathcal{D}_1)$ *and* $\mathsf{submin}_{\Leftarrow}(\mathcal{D}_2)$ *then* $\hat{e}' = \hat{e}''$.

what to do? well. we just ban that **over eager** use of the **rule of last resort** of subsumption. (subsume is a last resort in type checking; the corresponding rule in the analytic judgment that follows the type system is too.) these just aren't the sorts of things you're supposed to subsume.

we call derivations that obey this convention "subsumption minimal" (see agda), and then so this **is** a theorem. it's a little bit messy.

BUT.

it's not as ad hoc as it feels! it corresponds to proof theory!

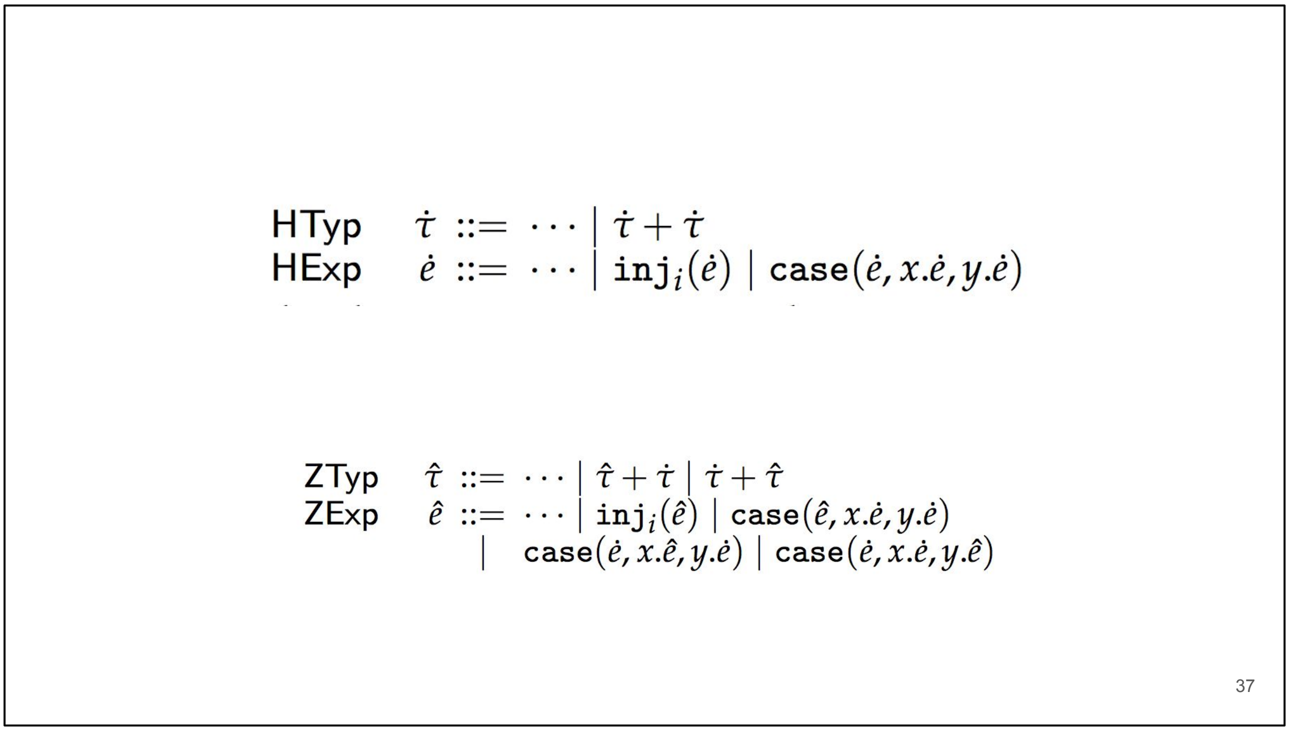
bidirectional systems are deeply related to focusing. but the difference between focusing and chaining is that you don't apply a rule too eagerly -- that you only switch modes when it's the only thing that might work. that's exactly what we do-- we only prove determinism for derivations that obey the intent of subsumption.

conveniently, this is also in correspondence with what you'd do as an implementor; you'd want to check the analytic rules first before trying anything else. so this is a spec that's easy to follow, if you think about it as one.

Extension

Now I want to talk about what happens when you really add something to the language.

recall, part of the motivation for the checksum theorems was that they should guide us through the process of adding things! those should be theorems of any extension, too, and looking at where the proofs we have break and how to write the rules!



$$\begin{array}{lrcl}
\textsf{HTyp} & \dot{\tau} & ::= & \cdots \mid \dot{\tau} + \dot{\tau} \\
\textsf{HExp} & \dot{e} & ::= & \cdots \mid \mathtt{inj}_i(\dot{e}) \mid \mathtt{case}(\dot{e}, x.\dot{e}, y.\dot{e})
\end{array}$$

$$\begin{array}{lrcl}
\textsf{ZTyp} & \hat{\tau} & ::= & \cdots \mid \hat{\tau} + \dot{\tau} \mid \dot{\tau} + \hat{\tau} \\
\textsf{ZExp} & \hat{e} & ::= & \cdots \mid \mathtt{inj}_i(\hat{e}) \mid \mathtt{case}(\hat{e}, x.\dot{e}, y.\dot{e}) \\
& & \mid & \mathtt{case}(\dot{e}, x.\hat{e}, y.\dot{e}) \mid \mathtt{case}(\dot{e}, x.\dot{e}, y.\hat{e})
\end{array}$$

extend the grammars, add new rules, etc. (note that it’s pretty mechanical)

$$\triangleright\dot\tau_1 + \dot\tau_2\triangleleft \xrightarrow{\texttt{move child 1}} \triangleright\dot\tau_1\triangleleft + \dot\tau_2$$

$$\triangleright\dot\tau_1 + \dot\tau_2\triangleleft \xrightarrow{\texttt{move child 2}} \dot\tau_1 + \triangleright\dot\tau_2\triangleleft$$

$$\triangleright\dot\tau_1\triangleleft + \dot\tau_2 \xrightarrow{\texttt{move parent}} \triangleright\dot\tau_1 + \dot\tau_2\triangleleft$$

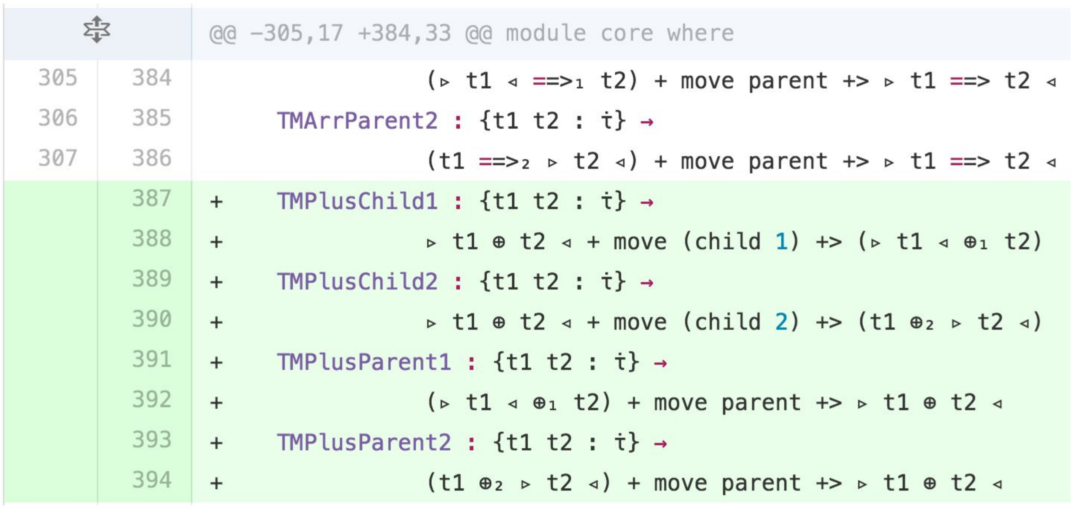
$$\dot\tau_1 + \triangleright\dot\tau_2\triangleleft \xrightarrow{\texttt{move parent}} \triangleright\dot\tau_1 + \dot\tau_2\triangleleft$$

$$\frac{\dot\tau \nsim (\!|\!) + (\!|\!)}{\dot\Gamma \vdash \triangleright(\!|\!)\triangleleft \xrightarrow{\texttt{construct inj } i} (\!|\texttt{inj}_i((\!|\!)) : \triangleright(\!|\!)\triangleleft + (\!|\!)|\!) \Leftarrow \dot\tau}$$

$$\frac{}{\dot\Gamma \vdash \triangleright(\!|\!)\triangleleft \xrightarrow{\texttt{construct case } x\ y} \texttt{case}(\triangleright(\!|\!)\triangleleft, x.(\!|\!), y.(\!|\!)) \Leftarrow \dot\tau}$$

example movement rules written in a (slightly more equational style) and construction rules for sums.

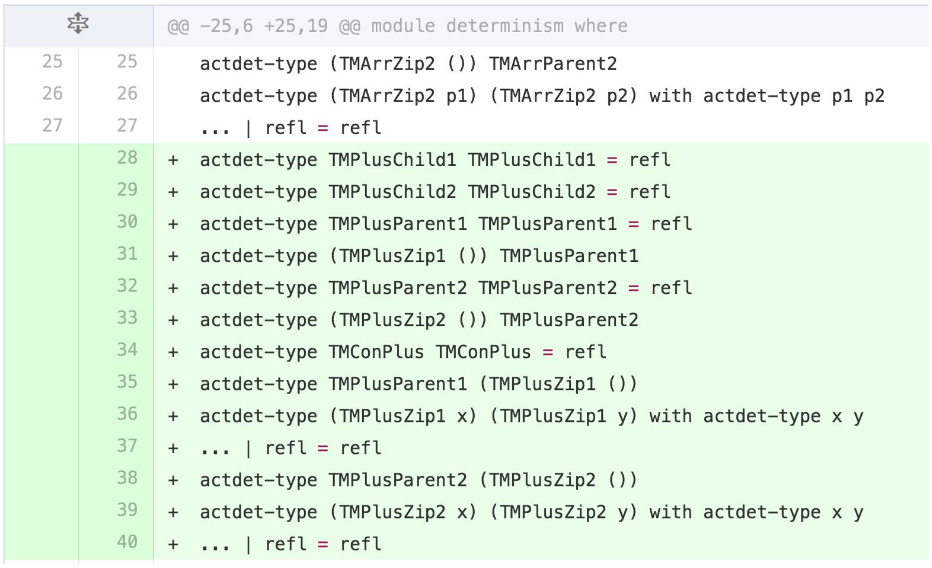
think about what you need to do to make the theorems true!



```
@@ -305,17 +384,33 @@ module core where
                   (▹ t1 ◃ ==>₁ t2) + move parent +> ▹ t1 ==> t2 ◃
     TMArrParent2 : {t1 t2 : τ̇} →
                   (t1 ==>₂ ▹ t2 ◃) + move parent +> ▹ t1 ==> t2 ◃
+    TMPlusChild1 : {t1 t2 : τ̇} →
+                  ▹ t1 ⊕ t2 ◃ + move (child 1) +> (▹ t1 ◃ ⊕₁ t2)
+    TMPlusChild2 : {t1 t2 : τ̇} →
+                  ▹ t1 ⊕ t2 ◃ + move (child 2) +> (t1 ⊕₂ ▹ t2 ◃)
+    TMPlusParent1 : {t1 t2 : τ̇} →
+                  (▹ t1 ◃ ⊕₁ t2) + move parent +> ▹ t1 ⊕ t2 ◃
+    TMPlusParent2 : {t1 t2 : τ̇} →
+                  (t1 ⊕₂ ▹ t2 ◃) + move parent +> ▹ t1 ⊕ t2 ◃
```

another payoff for having everything mechanized is that you can really use that mechanization to understand extensions.

this is a screenshot from the github pretty printed diff between the branches with and with sums added. so these are the lines that add the movement actions i just showed you. it’s a **conservative extension** because the **diff of the branches only adds code, it doesn’t delete anything**. this is just a screen cap from github!

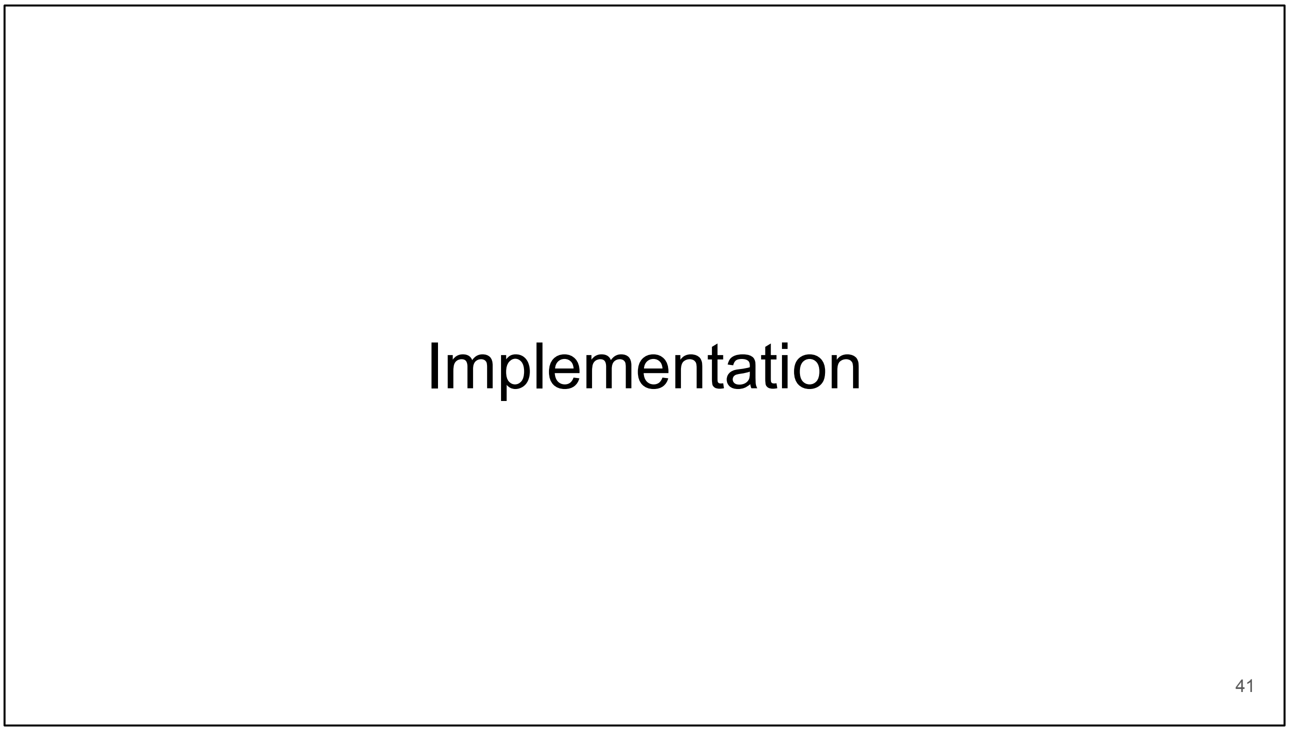


```
@@ -25,6 +25,19 @@ module determinism where
    actdet-type (TMArrZip2 ()) TMArrParent2
    actdet-type (TMArrZip2 p1) (TMArrZip2 p2) with actdet-type p1 p2
    ... | refl = refl
+   actdet-type TMPlusChild1 TMPlusChild1 = refl
+   actdet-type TMPlusChild2 TMPlusChild2 = refl
+   actdet-type TMPlusParent1 TMPlusParent1 = refl
+   actdet-type (TMPlusZip1 ()) TMPlusParent1
+   actdet-type TMPlusParent2 TMPlusParent2 = refl
+   actdet-type (TMPlusZip2 ()) TMPlusParent2
+   actdet-type TMConPlus TMConPlus = refl
+   actdet-type TMPlusParent1 (TMPlusZip1 ())
+   actdet-type (TMPlusZip1 x) (TMPlusZip1 y) with actdet-type x y
+   ... | refl = refl
+   actdet-type TMPlusParent2 (TMPlusZip2 ())
+   actdet-type (TMPlusZip2 x) (TMPlusZip2 y) with actdet-type x y
+   ... | refl = refl
```

and these are the new lines for the new cases in the proof of determinism. again, this is just github, but it’s letting us reason about our metatheory.

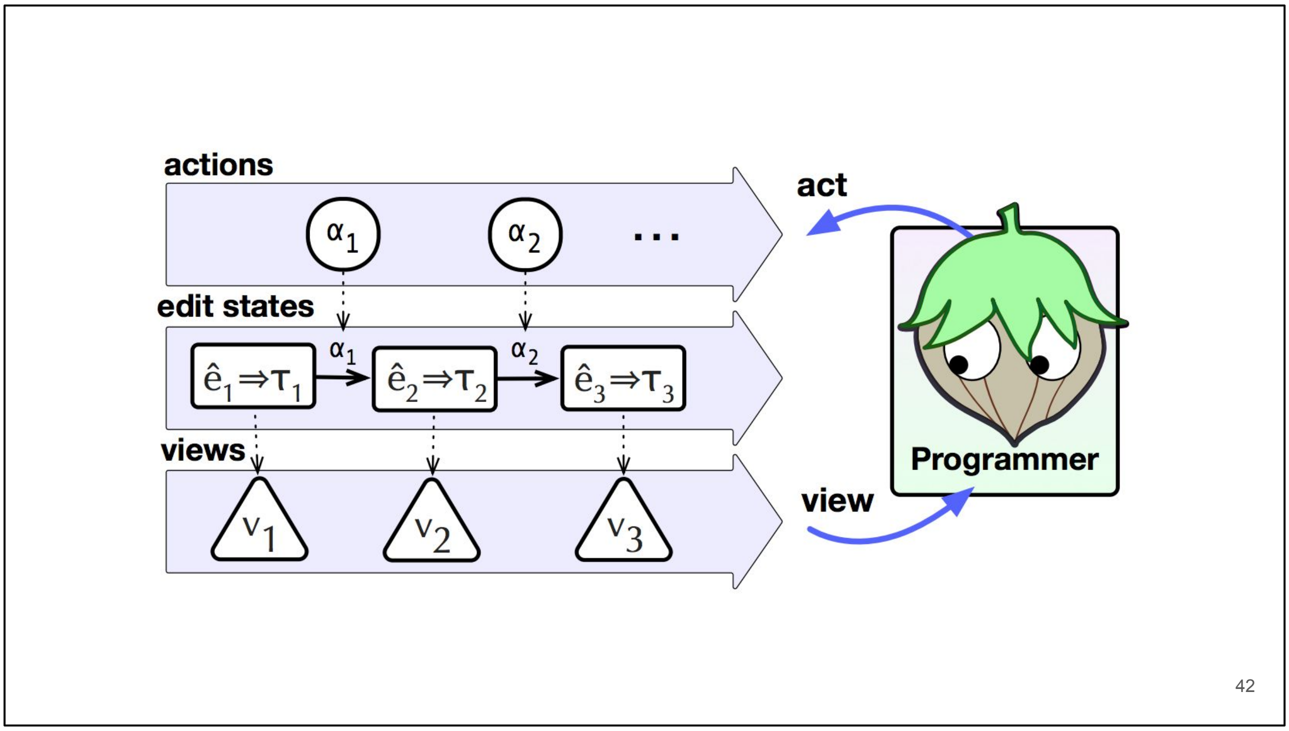
so when we added sums, we really did this. i sat down and did sums after getting everything else settled, and it took about a day. when we make new changes they merge in, because it’s a **conservative extension** and we **know how big that is.**

we got some stuff wrong -- forgotten premises, typos, etc.-- and only caught it because i mechanized it.



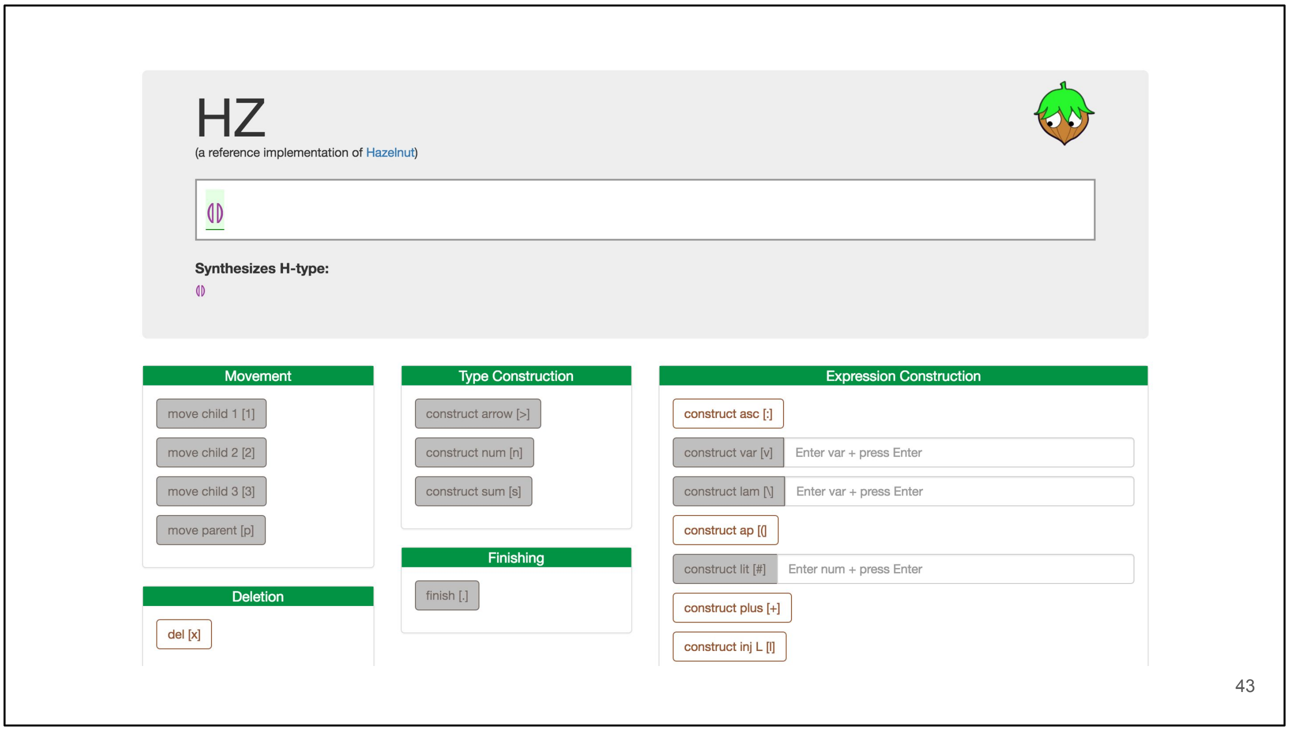
Implementation

We have an implementation of hazelnut called HZ, so that you can play with the rules and build some expressions and get a sense for what this feels like when it’s in motion. It’s obviously not an editor, but lets you get your hands dirty with the particular action semantics we have and understand it better.



remember that picture i showed you at the beginning?

if you squint slightly, it’s **coincidentally** pretty much exactly the classic FRP picture!



HZ

(a reference implementation of Hazelnut)

⦇⦈

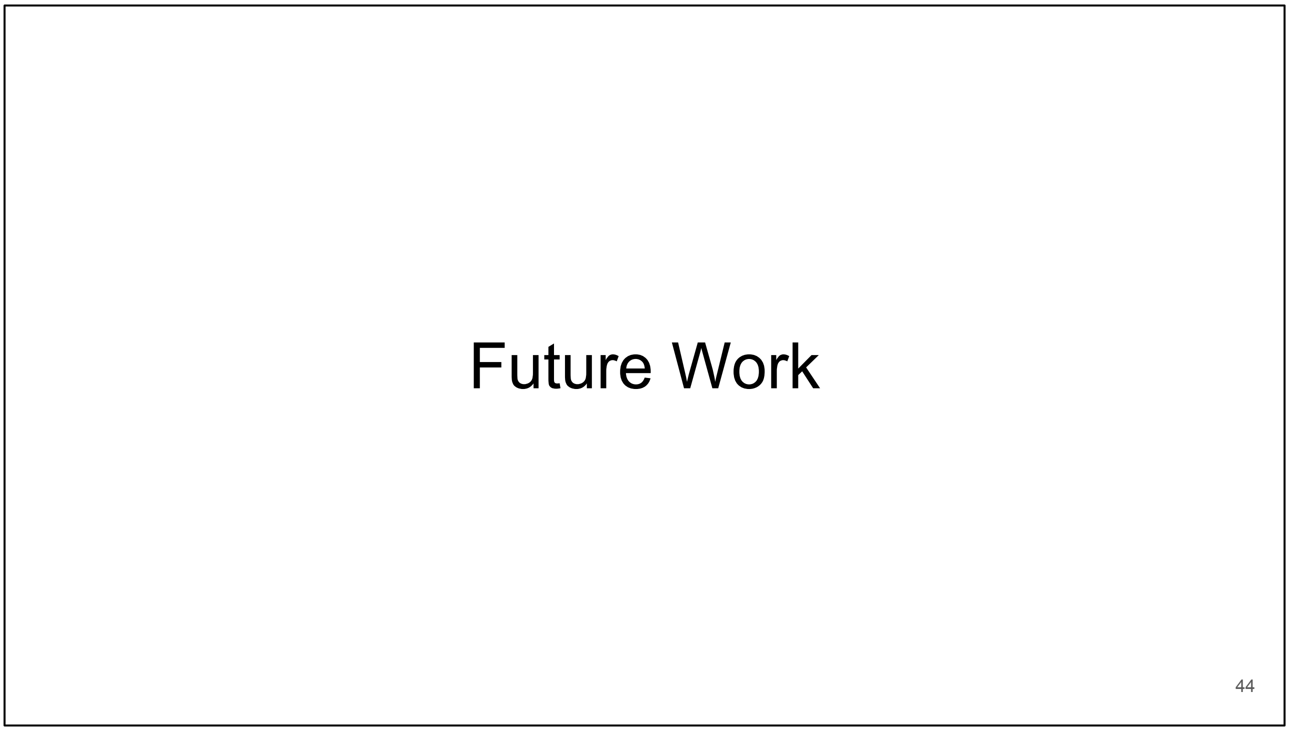
Synthesizes H-type:

⦇⦈

so we've taken that idea and used the FRP library js_of_ocaml to implement a web-based toy editor that lets you actually use hazelnut.

actions you can't take greyed out dynamically, including if you enter a variable name that doesn't make sense or that sort of thing.

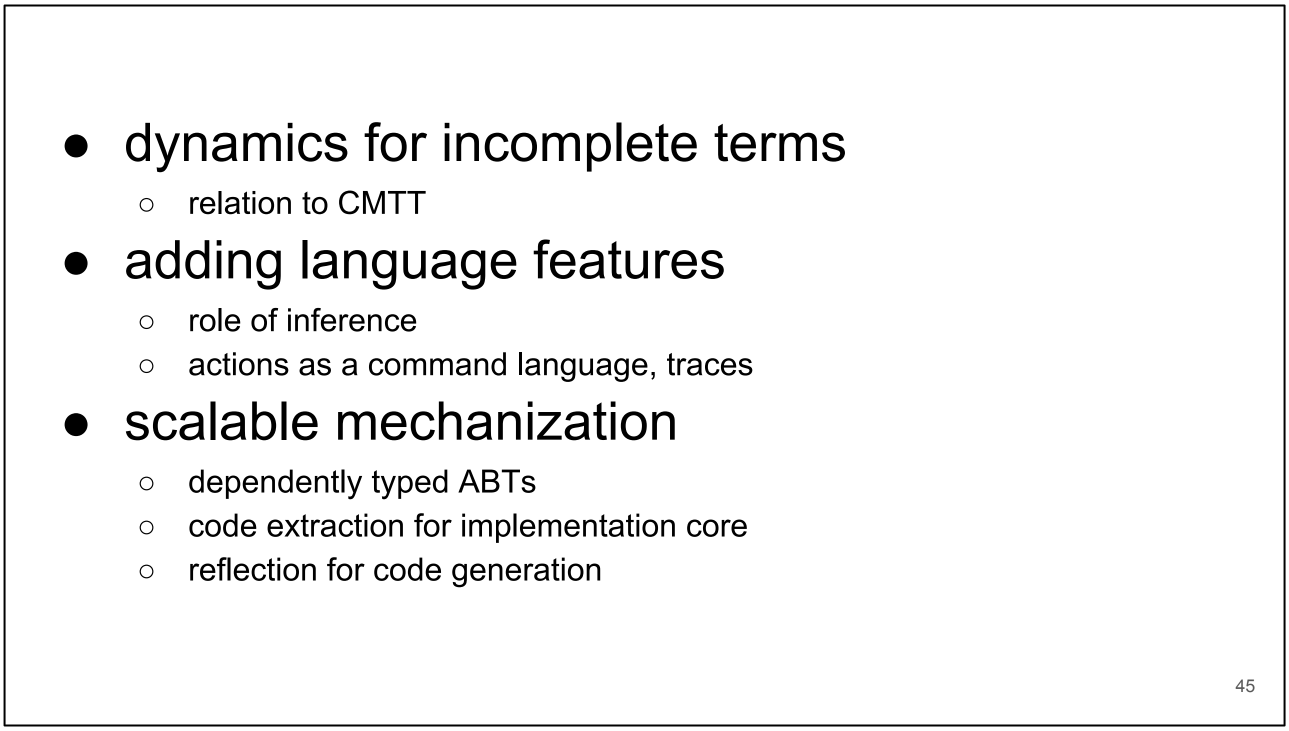
it's on our website so you can play with it!



Future Work

44

This is very much still a work in progress, so here are a just a couple of things we’re thinking about.



- dynamics for incomplete terms
  - relation to CMTT
- adding language features
  - role of inference
  - actions as a command language, traces
- scalable mechanization
  - dependently typed ABTs
  - code extraction for implementation core
  - reflection for code generation

- there’s the standard dynamics for complete terms, obviously. coincides with standard work.
- some way to evaluate terms with holes in them. agda and others have the “abort” semantics; just try it but give up if you can’t. CMTT (**Nanevski, pfenning, pientka**) gives some way forward, but only gives local reductions and only doesn’t allow holes in the *types* like we do.
- add things! action macros / derived actions, think about actions as a command modality; programming over / reasoning about traces of actions;
- the mechanization we have in agda is complete
  - we sidestepped the classic problem of dealing with bound variables by using brendrecht’s convention, but a more robust approach (like Sterling) with something like 2OAS or HOAS or PHOAS will be needed going forward. (and this clouds the issue with inference somewhat.)
  - it’s readable and great --- but probably doesn’t scale.
  - repeated work, ott style things, generate core parts of the implementation. (rel to synthesizer generator but modernized, or gradualizer)

----
inference stuff::

you could imagine using type inference to a similar end as bidirectional type checking.

we did not do this for a couple of reasons:

- conceptual simplicity of the presentation while working out the details of the new contributions: the bidirectional system is simpler to describe and good enough to reason about edit actions statically, so this avoids distraction.
- type inference for terms with holes in them is weaker -- you just have less information because things haven't been used yet (and not well understood)
- filling in one hole could affect constraints very non-locally. at best you'd have to re-infer the whole thing
- you then have to come up with error messages, which is known to be hard
- type inference doesn't scale to subtyping or dependent types, so it's not a panacea either, and it would be bad to place it too centrally to the design

Questions?

hazelgrove.github.io

(arXiv, github, and HZ)

Our draft paper is on arXiv; the source code of the paper, HZ, all the proofs are on github with **unexpurgated** commit messages; and a copy of HZ is live and you can run it in your browser.

This is all on our website, URL here.

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