Effects as Judgments

1. Want effects? Use judgments!
2. Nondeterminism
3. Compiling

1. Want effects? Use judgments!

A very important component of a programming language is its ability to use effectful computation. At a minimum we need to print to a screen or blink a light to provide output. So how is this done in a type-theoretic setting? Using judgments!

In particular, recall that judgmental fragments represent a kind of pure (or in self contained) model of some problem domain. The judgment A true, for instance models intuitionistic truth, and when we include terms as M : A true, it represents pure computation with no effects. Many other judgements exist for other domains, such as A eph for consumable resources, A nec for closed code that can be executed at compile time (i.e. staged computation), and A @ t, for quasiquotation and templating.

2. Nondeterminism

One effect that comes up in some contexts is nondeterminism, in the sense of a program which behaves nondeterministically in some defined way. For example, randomness. We can model nondeterminism with a judgment like any other effect.

Let's start by giving the <u>pure</u> fragment of the language, which we will wholly separate from the nondeterministic fragment:

$\boxed{\Gamma \vdash M : A\ \mathrm{true}}$ Term M has type A in context Γ.

$$\frac{}{\Gamma \vdash x : A\ \mathrm{true}}\ \mathrm{var}\ (x : A\ \mathrm{true} \in \Gamma)$$

Term $M, N ::= x$
$\quad \mid \langle M, N \rangle$
$\quad \mid \mathrm{fst}\ P$
$\quad \mid \mathrm{snd}\ P$

$$\frac{\Gamma \vdash M : A\ \mathrm{true} \qquad \Gamma \vdash N : B\ \mathrm{true}}{\Gamma \vdash \langle M, N \rangle : A \times B\ \mathrm{true}}\ \times I \qquad \frac{\Gamma \vdash P : A \times B\ \mathrm{true}}{\Gamma \vdash \mathrm{fst}\ P : A\ \mathrm{true}}\ \times E_1$$

$$\frac{}{\Gamma \vdash n : \mathrm{Nat}}\ \mathrm{Nat}I\ (n \text{ is a natural number}) \qquad \frac{\Gamma \vdash P : A \times B\ \mathrm{true}}{\Gamma \vdash \mathrm{snd}\ P : B\ \mathrm{true}}\ \times E_2$$

+ the usual judgmental fragment for A true:

- reflexivity
- weakening
- contraction
- exchange
- substitution

2. Nondeterminism (cont.)

Nothing has been done yet with the pure fragment, and infact we'll leave it unchanged, with no embedded nondeterministic code. No types that correspond to non-deterministic code are included, tho we could do this later if we wanted.

But now let's model nondeterminism:

$\boxed{\Gamma \vdash K : A \text{ nondet}}$ Program $\underline{K}$ is a nondeterministic computation of an $\underline{A}$ in context $\underline{\Gamma}$.

$$\frac{\Gamma \vdash M : A \text{ true}}{\Gamma \vdash M : A \text{ nondet}} \text{ single value}$$

$$\frac{\Gamma \vdash M_i : A \text{ true} \qquad \Gamma, x : A \text{ true} \vdash K : B \text{ nondet}}{\Gamma \vdash \text{choose } x \in \{\vec{M}\} \text{ in } K : B \text{ nondet}} \text{ choice}$$

Nondet. Program $K ::= M$
$\quad\quad\quad\quad\quad\quad\quad\quad\;\;\mid \text{choose } x \in \{\vec{m}\} \text{ in } K$

2. Nondeterminism (cont.)

2. Non-determinism (cont.)

Before continuing, let's build an intuition for what non-deterministic programs do. Consider the program

```
choose x ∈ {0, 1} in
  choose y ∈ {2, 3} in        : Nat nondet
    ⟨x, y⟩
```

The intended meaning of this program is that is denotes any/all of the values $\langle 0,2\rangle$, $\langle 0,3\rangle$, $\langle 1,2\rangle$, $\langle 1,3\rangle$. It might do this by actually picking x and y randomly, or it might do this by computing the set of all possible values, or maybe something else entirely, like, say, picking only one, given a seed and a PRNG.

So how <u>do</u> we give meanings to this? The above meanings are denotational in nature, or they sound like it, but we can give a big step semantics a little more cleanly, since we can avoid the denotations for types and contexts.

2. Non-determinism (cont.)

First, we must decide the value forms. One option is to define a single value form like so:

$$\begin{array}{lll} \text{Value } U, V ::= & n & \text{(natural numbers)} \\ & \mid \langle U, V \rangle & \text{(pairs)} \\ & \mid \{\vec{U}\} & \text{(sets)} \end{array}$$

And we would likewise have to explain how values relate to judgments. We had previously stated this as

$$\begin{array}{ll} \text{if} & M : A \quad \text{and} \quad M \Downarrow V \\ \text{then} & V : A \end{array}$$

But this only made sense because we had only one judgment, A true, which we abbreviated with just A, and also because all the value forms were also term forms. If we pick the definition of values above, we have to explain what it means to say $V : J$, with a whole new type theory fragment. That's fine! But inelegant. We'd also need to have a confusingly defined version of $M \Downarrow V$ that mixes program types.

2. Nondeterminism (cont.)

Instead, let's look at a far more natural version: we split the value type in two for each judgment:

$$\text{Single Value } \quad U, V ::= n \mid \langle U, V \rangle \qquad \begin{array}{l} \text{(natural numbers)} \\ \text{(pairs)} \end{array}$$

$$\text{Sets of Values } \quad S ::= \{\vec{U}\} \qquad \text{(sets)}$$

Now there's no room for confusion or weirdness in the values.

Here's single value big stepping:

$\boxed{M \Downarrow V}$ Deterministic term M normalizes to single value V.

$$\frac{}{n \Downarrow n}$$

$$\frac{M \Downarrow U \quad N \Downarrow V}{\langle M, N \rangle \Downarrow \langle U, V \rangle} \qquad \frac{P \Downarrow \langle U, V \rangle}{\mathsf{fst}\, P \Downarrow U} \qquad \frac{P \Downarrow \langle U, V \rangle}{\mathsf{snd}\, P \Downarrow V}$$

2. Nondeterminism (cont.)

Big step for nondeterministic programs will be denoted by $K \lightning S$, using a lightning bolt to remind you this requires caution!

$\boxed{K \lightning S}$ Nondeterministic program K normalizes to set of values S.

$$\frac{M \Downarrow V}{M \lightning \{V\}}$$

$$\frac{M_i \Downarrow V_i \qquad [V_i/x]K \lightning S_i}{\text{choose } x \in \{\vec{M}\} \text{ in } K \lightning \bigcup_i S_i}$$

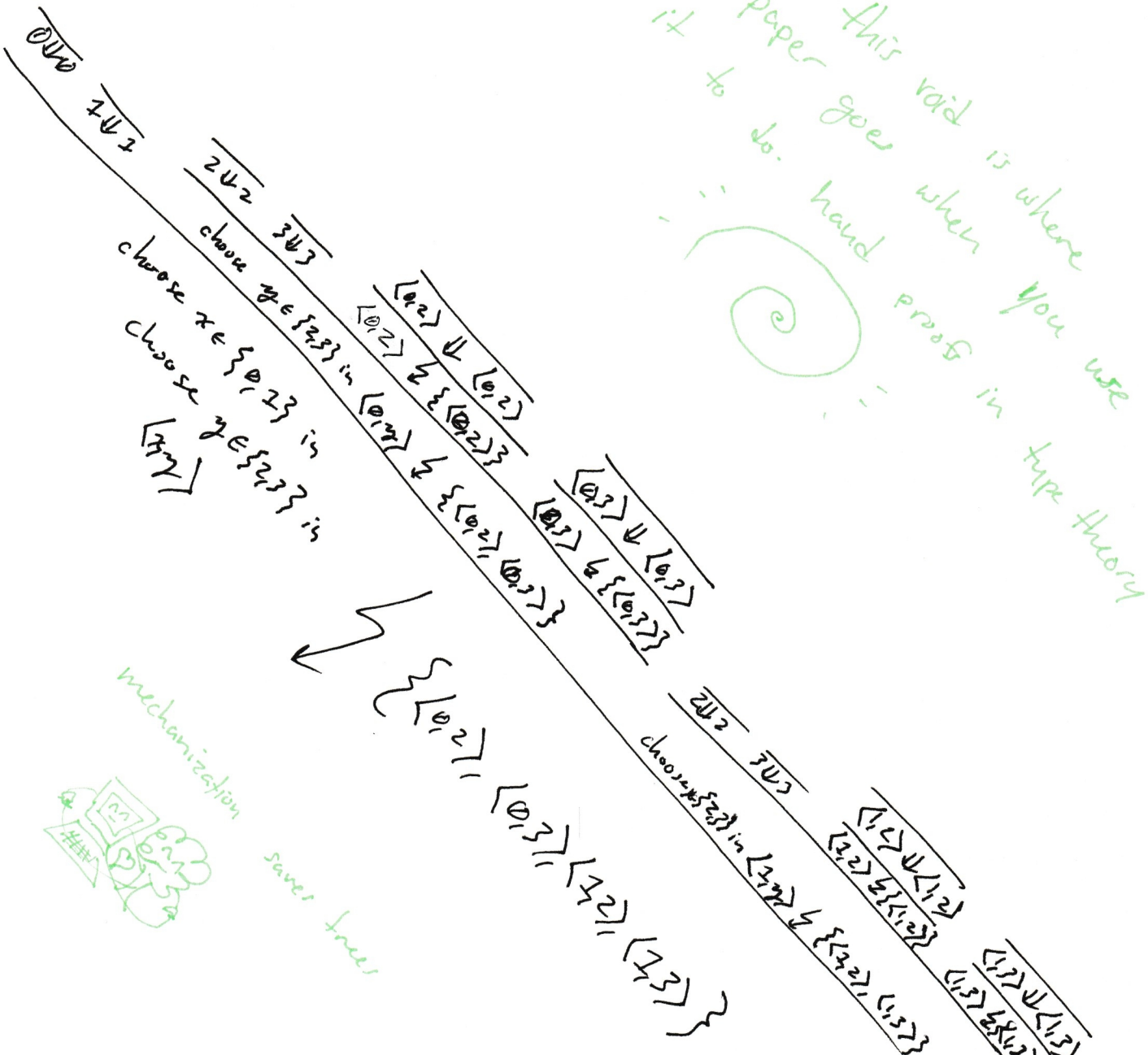
Here, $\bigcup_i$ is a meta-operation on sets, namely union over families of sets. Each S_i is a set, so $\bigcup_i S_i$ is the union of each S_i: $S_0 \cup S_1 \cup \ldots \cup S_n$.

2. Nondeterminism (cont.)

Now we can look at the normalization of our example program

```
choose x ∈ {0, 1} in
  choose y ∈ {2, 3} in
    ⟨x, y⟩
```

This one's wide so...



3. Compiling

How can we now compile nondeterministic programs? Much as we defined two big step relations for each kind of judgment — $M \Downarrow V$ and $K \nleftarrow S$ — we also can simply define two fragments of our operational semantics, one for each judgment.

Our terms and values would embed as normal with

if $M \Downarrow V$

then $\lfloor M \rfloor \longmapsto^* \lfloor V \rfloor$

And nondeterministic programs and sets would likewise embed. However, because some forms of nondeterminism use real randomness, or pseudorandomness, to pick only single values, we can't simply say

if $K \nleftarrow S$

then $\lfloor K \rfloor \longmapsto^* \lfloor S \rfloor$

3. Compiling (cont.)

Or atleast, we can't say that without using exotic logics such as David Hilbert's Epsilon Calculus. Afterall, how do we embed a set? If we can compute sets, sure, no problem, but what if the machine uses an actual RNG? In this setting, we instead would say

$$\text{if } K \not\in S$$
$$\text{then } \lfloor K \rfloor \longmapsto^* \lfloor V \rfloor \quad \text{for some } V \text{ in } S$$

or more formally:

$$\text{if } K \not\in S$$
$$\text{then } \exists V \in S.\ \lfloor K \rfloor \mapsto^* \lfloor V \rfloor$$

If the machine has only pseudo randomness, then this ought to suffice, because it's actually deterministic. But if we have true honest to goodness randomness, our machine's stepping "function" $\longmapsto$ is actually a stepping <u>relation</u> in earnest.

3. Compiling (cont)

In such a situation, it might make more sense to say

$$\text{if } K \not\downarrow S \text{ and } V \in S$$
$$\text{then } \lfloor K \rfloor \mapsto^* \lfloor V \rfloor$$

On top of this, we probably should also demand that nondeterminism is constrained to <u>only</u> produce S:

$$\text{if } K \not\downarrow S \text{ and } \lfloor K \rfloor \mapsto^* \lfloor V \rfloor$$
$$\text{then } V \in S$$

The specifics of the embeddings will now depend on the particular machines, but suppose that we embed to a stack machine with a true choice operator CHOOSE k which picks a random item from the top k stack items and then pops all but the chosen one. Then we might embed like so:

ℓ = return stack

$$\lfloor m \rfloor = \hat{\mathcal{E}}; \mathcal{E} \triangleright \text{compile } m \qquad \lfloor V \rfloor = \ldots V$$

e = environment

3. Compiling (cont.)

And for nondeterministic programs likewise:

$$\lfloor K \rfloor = \varepsilon;\varepsilon \triangleright \text{compile}_{ND}\ K$$

(e: ε, l: ε)

Everything now reduces to compiling terms, which we treat as standard and thus omit, and compiling the nondeterministic programs with compile_{ND}:

$$\text{compile}_{ND}\ m = \text{compile}\ m$$

$$\text{compile}_{ND}\ (\text{choose } x \in \{\vec{m}\} \text{ in } K) =$$

$$\text{compile}\ m_0 \mathbin{+\!\!+} \text{compile}\ m_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \text{compile}\ m_k \mathbin{+\!\!+} [\text{CHOOSE } k, \text{BIND } x] \mathbin{+\!\!+} \text{compile}_{ND}\ K$$

puts the k values of $\vec{m}$ on the stack when run

pops one of those k, discards rest, and thus binds it to variable x

In a very real sense, compiling is not deeply connected to the judgments at all: we could obviously compile with any language we wanted. But the judgmental separation helps force us to produce a <u>sensible</u> language with nondeterminism, rather than a shakey possibly bogus or confusing one.