Notes for Polynomial Language Talk

I want a language for polynomials

$$e ::= c \mid x \mid e_0 + e_1 \mid e_0 \cdot e_1$$
$$\mid \text{let } x = e_0 \text{ in } e_1$$

Define top-level functions/programs

$$p ::= \text{def } \underline{\text{name}} (\bar{x}_0, \bar{x}_1, \ldots)\ \bar{e}$$

Now, what might I want to do with a program written in this grammar?

- Interpret/Evaluate
- Compile

Semantics

- "Check" it

Safety/Types

- take Derivative
- take Adjoint Deriv.
- Partially Evaluate

Transform Program

(changing semantics)

- "Simplify"
- Canonicalize
- Optimize

Semantics-Preserving Transformations

Much of this remains true if we modify the language.

Modifications?

- Add Arrays?
- Numeric Integration?
- Probability Distributions
- Dist. Computing Primitives
- ...

Interpreting POLY

$[\![e \mid \sigma]\!]$ = value

e – expression

σ – environment (σ: variable names $\rightarrow$ values)

$$[\![c \mid \sigma]\!] = c$$

$$[\![x \mid \sigma]\!] = \sigma(x)$$

$$[\![e_0 + e_1 \mid \sigma]\!] = [\![e_0 \mid \sigma]\!] + [\![e_1 \mid \sigma]\!]$$

$$[\![e_0 \cdot e_1 \mid \sigma]\!] = [\![e_0 \mid \sigma]\!] \cdot [\![e_1 \mid \sigma]\!]$$

$$\left[\!\!\left[\begin{array}{l} \text{let } x = e_0 \\ \text{in } e_1 \end{array} \;\middle|\; \sigma \right]\!\!\right] = [\![e_1 \mid \sigma[x \mapsto [\![e_0 \mid \sigma]\!]]]\!]$$

where

$$\sigma[x \mapsto v](y) = \begin{cases} v, & \text{if } x = y \\ \sigma(y), & \text{otherwise} \end{cases}$$

Show Code

Compile ...

(let's make a simplifying assumption)

e has the form

$$\begin{aligned} Le &::= \text{let } x = e \text{ in } Le \\ &\mid \text{return } e \\ e &::= x \mid c \mid e_0 + e_1 \mid e_0 \cdot e_1 \end{aligned}$$

That is, all let-bindings are at

the top level.

Then we can compile to a C-function

$$C[\text{def name}(x_0, x_1, \ldots)\ e]$$
$$= \text{double name}(\text{double } x_0, \text{double } x_1, \ldots)\{ C[e] \}$$

```
= double name(
      double x0,
      double x1,
      ...
  ){
      C[e]
  }
```

$$C\left[\begin{array}{l}\text{let } x = e\\ \text{in } Le\end{array}\right] = \begin{array}{l}\text{double } x = C[e];\\ C[Le]\end{array}$$

$$C[\text{return } e] = \text{return } C[e];$$

(and you can do the rest)

Checks

① What happens if there's a free variable that isn't bound during evaluation or isn't in the signature during compilation?

(2)

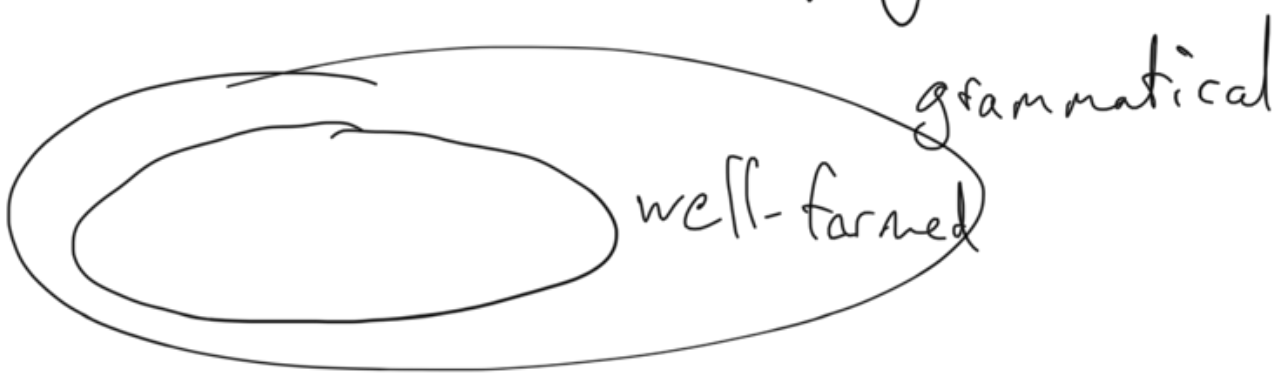
Program → Interpret

Program → Compile

If a program P interprets w/o error will it compile w/o error?

If a program P compiles w/o error will it interpret w/o error?

Ideally these are congruent for all "well-formed" programs



Enforce via "checking" safety (here variable binding)

→ e.g. type-checking

→ "effect-checking"

→ parallel safety   etc...

Defined separately from semantics.

Derivatives

"Total" or "forward" derivative

$$f : \underbrace{\mathbb{R} \times \mathbb{R} \times \cdots \times \mathbb{R}}_{n} \to \mathbb{R}$$

$$Df : \underbrace{\mathbb{R} \times \mathbb{R} \times \cdots \times \mathbb{R}}_{n} \times \underbrace{\mathbb{R} \times \mathbb{R} \times \cdots \times \mathbb{R}}_{n} \to \mathbb{R}$$

e.g. $f(x, y) = \ldots$

$$Df(x, y ; dx, dy) = \ldots$$

as partials...

$$Df(x, y ; dx, dy) = \left.\frac{\partial f}{\partial x}\right|_{x,y} \cdot dx + \left.\frac{\partial f}{\partial y}\right|_{x,y} \cdot dy$$

from first principles...

$Df(x, y)$ is the closest linear approximation to

$$f(x + dx, y + dy) \approx f(x, y) + Df(x, y ; dx, dy)$$

$$D[\![\text{def name}(x_0, x_1, \ldots)\ e]\!]$$

$$= \text{def Dname}(x_0, x_1, \ldots, dx_0, dx_1, \ldots)$$

$$D[\![e \mid [x_0 \mapsto dx_0,\ x_1 \mapsto dx_1,\ \dots]]\!]$$

$$D[\![\text{let } x = e_0 \text{ in } e_1 \mid \sigma]\!] = \text{let } x = e_0 \text{ in let } dx = D[\![e_0 \mid \sigma]\!] \text{ in } D[\![e_1 \mid \sigma[x \mapsto dx]]\!]$$

This is the Chain Rule

$$D[\![c \mid \sigma]\!] = \emptyset$$

$$D[\![x \mid \sigma]\!] = \emptyset \text{ if } x \notin \sigma$$

$$D[\![x \mid \sigma]\!] = \sigma(x)$$

$$D[\![e_0 + e_1 \mid \sigma]\!] = D[\![e_0 \mid \sigma]\!] + D[\![e_1 \mid \sigma]\!]$$

$$D[\![e_0 \cdot e_1 \mid \sigma]\!] = D[\![e_0 \mid \sigma]\!] \cdot e_1 + e_0 \cdot D[\![e_1 \mid \sigma]\!]$$

Leibniz Product Rule

"Adjoint Derivative"

e.g. Gradient

aka. "Reverse Mode"

example: $f(x, y$

$$f(x,y)$$

$$f: \mathbb{R}\times\mathbb{R}\to\mathbb{R}$$

$$Df: \mathbb{R}\times\mathbb{R}\times\underbrace{\mathbb{R}\times\mathbb{R}\to\mathbb{R}}_{\text{linear}}$$

$$D^Tf: \mathbb{R}\times\mathbb{R}\times\underbrace{\mathbb{R}\to\mathbb{R}\times\mathbb{R}}_{\text{linear}}$$

(same as "matrix transpose")

Algebraic definition by a universal property...

Let $g: A\to B$ linear, then $g^T: B\to A$ is the unique linear function s.t.

$$\forall x\in A: \forall y\in B:$$

$$\langle y \,|\, g(x)\rangle = \langle g^T(y) \,|\, x\rangle$$

($\langle\cdot|\cdot\rangle$ is "inner product" aka. dot-product)

Note:

for $f(x,y)$ f^T returns

a pair but we don't have pairs in our language.

Therefore our language is not closed under adjoint derivatives.

We will ignore this, but note that there is no magic reason that when you investigate a language, everything will work out. That's why this is research.

Partial Evaluation

We defined the "total derivative" as the closest linear approximation, but what if we want a plain old rise-over-run/slope derivative?

Let $f : \mathbb{R} \to \mathbb{R}$

$f(x)$.

Then $\frac{df}{dx} = Df(x; 1)$

$$\text{b/c } Df(x, dx) = \left.\frac{df}{dx}\right|_x \cdot dx$$

For a function definition

$$\text{def name}(x_0, x_1, \ldots)\ e$$

the partial evaluation w.r.t. $x_k = v$ is

$$\begin{aligned}&\text{def PEname}(x_0, \ldots, x_{k-1}, x_{k+1}, \ldots)\\ &\quad \text{let } x_k = v \text{ in } e\end{aligned}$$

Note: this definition of partial evaluation is functionally correct, but does not optimize the implementation how we would normally expect.

The preceding program transformations change the signature of a function, so we know they aren't "semantics preserving." By contrast, the remaining transformations will be. Therefore it is useful to define a notion of program equality. The kind of equality we care about (between expressions) is usually called **Observational Equality**.

$e_0 \cong e_1$ iff.

$\forall \sigma : [\![e_0 \mid \sigma]\!] = [\![e_1 \mid \sigma]\!]$

where σ is such that both sides are well defined. (i.e. σ binds all free variables)

Observational Eq. is well-defined for general λ-calculi but we can simplify in our context.

Some Random "Simplification" Rules

$0 + e \overset{\rightsquigarrow}{\cong} e$

$0 \cdot e \overset{\rightsquigarrow}{\cong} 0$

$1 \cdot e \overset{\rightsquigarrow}{\cong} e$

$e + e \overset{\rightsquigarrow}{\cong} 2 \cdot e$

...

(Note: these rules can be justified by appealing to the semantics)

When will I have "enough" simplification rules? Is that possible?

Canonical Forms

Let E be the set of well-formed expressions in POLY. Then a canonicalization procedure/function $\gamma : E \to E$ is a function s.t. $\forall e : e \cong \gamma(e)$

syntactic equality

and $\forall e_1, e_2 : (e_1 \cong e_2) \iff \gamma(e_1) = \gamma(e_2)$

That is all equivalent expressions canonicalize into syntactically identical canonical forms.

Observe:

If we enumerate every rewrite rule used in a canonicalization procedure, then those rewrites effectively axiomatize observational equality.

Claim: This will be possible for POLY, but not for general Turing-complete languages.

The existence of a computable canonicalization function for a language trivially yields a decision procedure for observational equality of terms in that language. However, this is an undecidable problem for Turing-Complete languages (by Rice's Theorem; see any Automata Theory textbook)

Canonicalization of POLY

Any polynomial can be uniquely expressed on the "monomial basis."
For instance

$$(x-y)\cdot(x\cdot z+2) = x^2 z - xyz + 2x - 2y$$

Here is a general procedure.

1. Substitute all let-bindings

$$\text{let } x = e_0 \text{ in } e_1 \cong [x \mapsto e_0]\, e_1$$

(2) Distribute all mult. over all sums

$$e_0 \cdot (e_1 + e_2) \overset{\rightsquigarrow}{=} e_0 \cdot e_1 + e_0 \cdot e_2$$

sim. for $(e_0 + e_1) \cdot e_2$

(3) Canonicalize monomials

$$e_0 \cdot (e_1 \cdot e_2) \overset{\rightsquigarrow}{\cong} (e_0 \cdot e_1) \cdot e_2$$

$$c_0 \cdot c_1 \overset{\rightsquigarrow}{\cong} c' \leftarrow \text{the computed product}$$

$$e_0 \cdot e_1 \cong e_1 \cdot e_0$$

↳ apply in conjunction with associativity until all constants are leading and all variables occur in lexicographic order. If there is no leading constant, make 1 the leading constant.

(4) Canonicalize order of summation

$$e_0 + (e_1 + e_2) \overset{\rightsquigarrow}{\cong} (e_0 + e_1) + e_2$$

$$e_0 + e_1 \cong e_1 + e_2$$

↳ Commute and associate in lexicographic order, with all constants leading

(5) Collapse summation

$$c_0 + c_1 \overset{\rightsquigarrow}{\cong} c' \leftarrow \text{(computed)}$$

$$c_0 \cdot e + c_1 \cdot e \overset{\rightsquigarrow}{=} c' \cdot e$$

6. Constant elimination

$$0 \cdot e \overset{\rightsquigarrow}{=} 0$$

$$1 \cdot e \overset{\rightsquigarrow}{=} e$$

$$0 + e \overset{\rightsquigarrow}{=} e$$

This procedure γ satisfies $\gamma(e) \overset{\sim}{=} e$ by showing that each constituent rewrite preserves equality.

However, we need some special other argument to show that if $\gamma(e_1) \neq \gamma(e_2)$, then $e_1 \not\overset{\sim}{=} e_2$.

For polynomials this is simply the fundamental theorem of algebra.

We now have a complete set of rewrite rules. But if we want to "simplify" or "optimize" which do we apply?

In general, substitution and/or distribution can lead to exponentially large terms.

<u>Note</u>: We are now implicitly assuming a <u>cost-model</u> based on expression size. In more complex languages we may want other cost-models — perhaps even many for the same language.

Rules w/ cost

$$\text{let } x = e_0 \text{ in } e_1 \overset{\rightsquigarrow}{\cong} [x \mapsto e_0]e_1$$

$$e_0 \cdot (e_1 + e_2) \overset{\rightsquigarrow}{\cong} e_0 \cdot e_1 + e_0 \cdot e_2$$

$$e_0 \cdot (e_1 \cdot e_2) \overset{\rightsquigarrow}{\cong} (e_0 \cdot e_1) \cdot e_2$$

$$e_0 \cdot e_1 \overset{\rightsquigarrow}{\cong} e_1 \cdot e_0$$

$$e_0 + (e_1 + e_2) \overset{\rightsquigarrow}{\cong} (e_0 + e_1) + e_2$$

$$e_0 + e_1 \overset{\rightsquigarrow}{\cong} e_1 + e_0$$

$$c_0 \cdot c_1 \overset{\rightsquigarrow}{\cong} c' \leftarrow \text{computed}$$

$$c_0 + c_1 \overset{\rightsquigarrow}{\cong} c' \leftarrow \text{computed}$$

(leading constants are handleable by distribution + computation)

$$0 \cdot e \overset{\rightsquigarrow}{=} 0$$

$$1 \cdot e \overset{\rightsquigarrow}{=} e$$

$$0 + e \overset{\rightsquigarrow}{=} e$$

Given this analysis, a simplification or optimization procedure would run only the cost-reducing rules, possibly augmented by the normalization orderings of the cost-neutral rules.

However, we can do better by running cost <u>negative</u> rules in <u>reverse</u>.

Running let-substitution in reverse is just <u>common subexpression elimination</u>.

What about distribution?

Consider a polynomial in one variable

2 3

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

Then a Horner Scheme is the factorization

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot a_3))$$

A multi-variate Horner Scheme is a (usually greedy) factorization of "common terms" from a sum of products. This is an NP-Hard search problem.

Note: Optimization has largely reduced to a known problem...

Recap

We defined our POLY language

$$e ::= c \mid x \mid e_0 + e_1 \mid e_0 \cdot e_1$$
$$\mid \text{let } x = e_0 \text{ in } e_1$$

To handle adjoints, we need pairs

$$e ::= \ldots \mid (e_0, e_1) \mid \pi_0\, e \mid \pi_1\, e$$

ATL adds

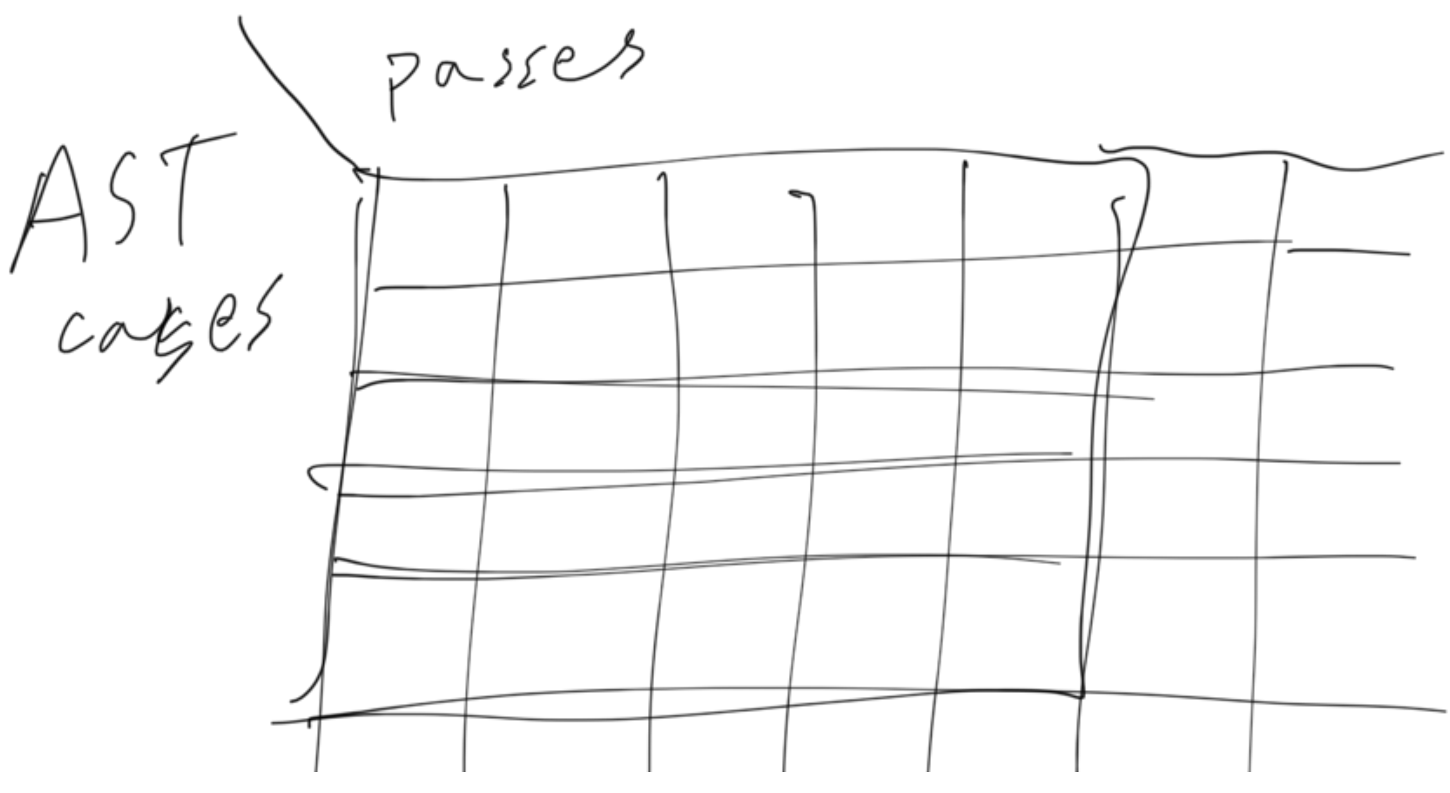
$$e ::= \ldots \mid \boxplus_{i=0}^{n} e \mid \sum_{i=0}^{n} e \mid e[i]$$
$$\mid [p] \cdot e$$
$$p ::= i = j \mid p_0 \wedge p_1$$
$\mid$ ~ affine expressions ~

$\mid R(i_0, i_1, \ldots)$

What about integrals?

The <u>expression problem</u>

is the combinatorial code growth induced by growing compilers



Recap