Termination of higher-order probabilistic programs

Charles Grellois
(joint work with Ugo Dal Lago)

Aix-Marseille Université

Séminaire LIRICA
6 novembre 2017

Functional programs, Higher-order models

Imperative vs. functional programs

- Imperative programs: built on finite state machines (like Turing machines).

  Notion of state, global memory.

- Functional programs: built on functions that are composed together (like in Lambda-calculus).

  No state (except in impure languages), higher-order: functions can manipulate functions.

(Turing machines and λ-terms are equivalent in expressive power)

Imperative vs. functional programs

- Imperative programs: built on finite state machines (like Turing machines).

  Notion of state, global memory.

- Functional programs: built on functions that are composed together (like in Lambda-calculus).

  No state (except in impure languages), higher-order: functions can manipulate functions.

(Turing machines and λ-terms are equivalent in expressive power)

Example: imperative factorial

```
int fact(int n) {
  int res = 1;
  for i from 1 to n do {
    res = res * i;
    }
  }
  return res;
}
```

Typical way of doing: using a variable (change the state).

Example: functional factorial

In OCaml:

```
let rec factorial n =
    if n <= 1 then
      1
    else
      factorial (n-1) * n;;
```

Typical way of doing: using a recursive function (don't change the state).

In practice, forbidding global variables reduces considerably the number of bugs, especially in a parallel setting (cf. Erlang).

Advantages of functional programs

- Very mathematical: calculus of functions.
- ...and thus very much studied from a mathematical point of view. This notably leads to strong typing, a marvellous feature.
- Much less error-prone: no manipulation of global state.

More and more used, from Haskell and Caml to Scala, Javascript and even Java 8 nowadays.

Also emerging for probabilistic programming.

Price to pay: analysis of higher-order constructs.

Advantages of functional programs

Price to pay: analysis of higher-order constructs.

Example of higher-order function: `map`.

`map` φ $[0, 1, 2]$ returns $[\varphi(0), \varphi(1), \varphi(2)]$.

Higher-order: `map` is a function taking a function φ as input.

Probabilistic functional programs

Probabilistic programming languages are more and more pervasive in computer science: modeling uncertainty, robotics, cryptography, machine learning, AI. . .

What if we add probabilistic constructs?

In this talk: $$M \oplus_p N \to_v \{ M^p, N^{1-p} \}$$

Allows to simulate some random distributions, not all. In future work: add fully the two roots of probabilistic programming, drawing values at random from more probability distributions (typically on the reals), and conditioning which allows among others to do machine learning.

Using higher-order functions

Bending a coin in the probabilistic functional language Church:

```
var makeCoin = function(weight) {
  return function() {
    flip(weight) ? 'h' : 't'
  }
}
var bend = function(coin) {
  return function() {
    (coin() == 'h') ? makeCoin(0.7)() : makeCoin(0.1)()
  }
}
var fairCoin = makeCoin(0.5)
var bentCoin = bend(fairCoin)
viz(repeat(100,bentCoin))
```

Motivations

- Quantitative notion of termination: almost-sure termination (AST) which is notably required to do probabilistic inference...
- AST has been studied for imperative programs in the last years...
- ...but what about the functional probabilistic languages?

Goal of the talk. Go towards verification of probabilistic functional programs. We give an incomplete method for termination-checking.

Roadmap

1. A few words on the λ-calculus with recursion

2. A type system for termination of probabilistic functional programs

A few words on the λ-calculus

Definition, simply-typed fragment, recursion, natural numbers

λ-terms

Grammar:

$$M, N \ ::= \ x \ \mid \ \lambda x.M \ \mid \ M \ N$$

Calculus of functions:

- x is a variable,
- $\lambda x.M$ is intuitively a function $x \mapsto M$,
- $M \ N$ is the application of functions.

λ-terms

Grammar:

$$M, N \ ::= \ x \ \mid \ \lambda x.M \ \mid \ M \ N$$

Examples:

- $\lambda x.x$: identity $x \mapsto x$,
- $\lambda x.y$: constant function $x \mapsto y$,
- $(\lambda x.x) \ y$: application of the identity to y,
- $\Delta \ = \ \lambda x.x \ x$: duplication.

β-reduction

$$(\lambda x.x)\ \ y$$

is an application of functions which should compute y:

$$(\lambda x.x)\ \ y \rightarrow_\beta y$$

Beta-reduction gives the dynamics of the calculus.
(= the evaluation of the functions/programs).

This calculus is equivalent in expressive power, for functions $\mathbb{N} \rightarrow \mathbb{N}$, to Turing machines.

β-reduction

Formally:

$$(\lambda x.M) \ N \ \rightarrow_\beta \ M[x/N]$$

Examples:

$$(\lambda x.y) \ z \rightarrow_\beta y$$

β-reduction

Formally:

$$(\lambda x.M)\ \ N\ \ \rightarrow_\beta\ \ M[x/N]$$

Examples:

$$\begin{array}{rl} & (\lambda f.\lambda x.f\ (f\ x))\ (g\ g)\ y \\ \rightarrow_\beta & (\lambda x.g\ (g\ (g\ (g\ x))))\ \ y \\ \rightarrow_\beta & g\ (g\ (g\ (g\ y))) \end{array}$$

The looping term Ω

Just like with Turing machines, there are computations that never stop.

Set $\Omega \;=\; \Delta\;\Delta \;=\; (\lambda x.x\;x)(\lambda x.x\;x)$.

Then:

$$\begin{array}{rl} & \Omega = (\lambda x.x\;x)(\lambda x.x\;x) \\ \rightarrow_\beta & (x\;x)\,[x/\lambda x.x\;x] \;=\; \Omega \\ \rightarrow_\beta & \Omega \\ \rightarrow_\beta & \cdots \end{array}$$

The looping term Ω

Just like with Turing machines, there are computations that never stop.
But that may depend on how we compute.

$$(\lambda x.y)\ \Omega \quad \rightarrow_\beta \quad y$$

if we reduce the first redex, or

$$(\lambda x.y)\ \Omega \quad \rightarrow_\beta \quad (\lambda x.y)\ \Omega$$

if we try to reduce the second (inside Ω)...

- Weak normalization: at least one way of computing terminates
- Strong normalization (SN): all ways of computing terminate.

Simple types and strong normalization

Problem with Ω: it contains $x\ x$.
So x is at the same time a function and an argument of this function.

Simple types forbid this: you have to be a function $A \to A$ or an argument of type A, but not both.

It is enough to guarantee strong normalization:

$$M \text{ has a simple type} \Rightarrow M \text{ is SN.}$$

It's an incomplete characterization: $\Delta \ = \ \lambda x.x\ x$ is SN (no way to reduce it!) but not typable.
(simple typing is decidable, so it couldn't be complete).

Simple types

Simple types: $\sigma, \tau \ ::= \ o \ \Big| \ \sigma \to \tau.$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}$$

Recursion

We can add recursion with a new construct:

$$M, N \ ::= \ \cdots \ \mid \ \text{letrec } f \ = \ M$$

a new rewrite rule:

$$\text{letrec } f \ = \ M \ \to \ M[f/\text{letrec } f \ = \ M]$$

and a new typing rule:

$$\frac{\Gamma, f : \sigma \to \tau \vdash M : \sigma \to \tau}{\Gamma \vdash \text{letrec } f \ = \ M : \sigma \to \tau}$$

which does not guarantee SN: letrec $f \ = \ f$ is typable and loops forever.

Natural numbers

A way to add natural numbers: add them as constructors built inductively, together with a destructor (pattern-matching).

$$M, N \ ::= \ \cdots \ \mid \ 0 \ \mid \ \mathsf{S}\ M \ \mid \ \mathsf{case}\ M\ \mathsf{of}\ \{ \mathsf{S} \to N \mid 0 \to L \}$$

Reductions:

$$\begin{array}{rcl} \mathsf{case}\ \mathsf{S}\ M\ \mathsf{of}\ \{ \mathsf{S} \to N \mid 0 \to L \} & \to & N\ M \\ \mathsf{case}\ 0\ \mathsf{of}\ \{ \mathsf{S} \to N \mid 0 \to L \} & \to & L \end{array}$$

Note: all we do in this talk can be done with inductive types (lists, trees. . .)

Natural numbers

A way to add natural numbers: add them as constructors built inductively, together with a destructor (pattern-matching).

$$M,\ N \quad ::= \quad \cdots \quad \Big| \quad 0 \quad \Big| \quad \mathsf{S}\ M \quad \Big| \quad \text{case } M \text{ of } \Big\{ \mathsf{S} \to N \ \Big|\ 0 \to L \Big\}$$

Typing:

$$\frac{}{\Gamma \vdash 0 \,:\, \mathsf{Nat}} \qquad\qquad \frac{\Gamma \vdash M \,:\, \mathsf{Nat}}{\Gamma \vdash \mathsf{S}\ M \,:\, \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M \,:\, \mathsf{Nat} \qquad \Gamma \vdash N \,:\, \mathsf{Nat} \to \sigma \qquad \Gamma \vdash L \,:\, \sigma}{\Gamma \vdash \text{case } M \text{ of } \Big\{ \mathsf{S} \to N \ \Big|\ 0 \to L \Big\} \,:\, \sigma}$$

where we consider $o\ =\ \mathsf{Nat}$.

Sized Types and Termination

A sound termination check for the deterministic case

Sized types: the deterministic case

Sized types: a decidable extension of the simple type system ensuring SN for λ-terms with letrec.

Fundamental idea of typing: types describe properties of programs.
In sized types: properties linked with termination properties.

See notably:

- Hughes-Pareto-Sabry 1996, *Proving the correctness of reactive systems using sized types*,
- Barthe-Frade-Giménez-Pinto-Uustalu 2004, *Type-based termination of recursive definitions*.

Sized types: the deterministic case

Sizes: $\qquad \mathfrak{s}, \mathfrak{r} \quad ::= \quad \mathfrak{i} \mid \infty \mid \widehat{\mathfrak{s}}$

+ size comparison underlying subtyping. Notably $\widehat{\infty} \equiv \infty$.

Idea: k successors = at most k constructors.

- $\mathsf{Nat}^{\widehat{\mathfrak{i}}}$ is 0,
- $\mathsf{Nat}^{\widehat{\widehat{\mathfrak{i}}}}$ is 0 or S 0,
- ...
- Nat^{∞} is any natural number. Often denoted simply Nat.

The same for lists,...

Sized types: the deterministic case

Sizes: $\mathfrak{s}, \mathfrak{r} \quad ::= \quad \mathfrak{i} \mid \infty \mid \widehat{\mathfrak{s}}$

+ size comparison underlying subtyping. Notably $\widehat{\infty} \equiv \infty$.

Fixpoint rule:

$$\frac{\Gamma, f \,:\, \mathsf{Nat}^{\mathfrak{i}} \to \sigma \vdash M \,:\, \mathsf{Nat}^{\widehat{\mathfrak{i}}} \to \sigma[\mathfrak{i}/\widehat{\mathfrak{i}}] \qquad \mathfrak{i} \text{ pos } \sigma}{\Gamma \vdash \mathsf{letrec}\ f \ = \ M \,:\, \mathsf{Nat}^{\mathfrak{s}} \to \sigma[\mathfrak{i}/\mathfrak{s}]}$$

"To define the action of f on size $n+1$,
we only call recursively f on size at most n"

Sized types: the deterministic case

Sizes: $\quad \mathfrak{s}, \mathfrak{r} \quad ::= \quad \mathfrak{i} \mid \infty \mid \widehat{\mathfrak{s}}$

+ size comparison underlying subtyping. Notably $\widehat{\infty} \equiv \infty$.

Fixpoint rule:

$$\frac{\Gamma, f \,:\, \mathsf{Nat}^{\mathfrak{i}} \to \sigma \vdash M \,:\, \mathsf{Nat}^{\widehat{\mathfrak{i}}} \to \sigma[\mathfrak{i}/\widehat{\mathfrak{i}}] \qquad \mathfrak{i} \text{ pos } \sigma}{\Gamma \vdash \mathsf{letrec}\ f \ = \ M \,:\, \mathsf{Nat}^{\mathfrak{s}} \to \sigma[\mathfrak{i}/\mathfrak{s}]}$$

Typable $\Longrightarrow$ SN. Proof using reducibility candidates.

Decidable type inference: no completeness, but of practical use.

Sized types: example in the deterministic case

From Barthe et al. (op. cit.):

$$
\begin{array}{lll}
\mathrm{plus} \equiv (\mathsf{letrec} & plus_{:\mathrm{Nat}^{\imath}\to\mathrm{Nat}\to\mathrm{Nat}} = & \\
 & \lambda x_{:\mathrm{Nat}^{\widehat{\imath}}}.\ \lambda y_{:\mathrm{Nat}}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y & \\
 & \qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x'_{:\mathrm{Nat}^{\imath}}.\ \mathsf{s}\ \underbrace{(plus\ \ x'\ \ y)}_{:\mathrm{Nat}} & \\
 & \qquad\qquad\qquad\qquad \} & \\
)\ : & \mathrm{Nat}^{s}\to\mathrm{Nat}\to\mathrm{Nat} &
\end{array}
$$

The case rule ensures that the size of x' is lesser than the one of x.
Size decreases during recursive calls $\Rightarrow$ SN.

Probabilistic Termination

A probabilistic λ-calculus

$$
\begin{array}{rcl}
M, N, \ldots & ::= & V \mid V\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid M \oplus_p N \\
& & \mid \mathsf{case}\ V\ \mathsf{of}\ \{ \mathsf{S} \to W \mid 0 \to Z \}
\end{array}
$$

$$
V, W, Z, \ldots \quad ::= \quad x \mid 0 \mid \mathsf{S}\ V \mid \lambda x.M \mid \mathsf{letrec}\ f = V
$$

- Formulation equivalent to λ-calculus with $\oplus_p$, but constrained for technical reasons (A-normal form)
- Restriction to base type Nat for simplicity, but can be extended to general inductive datatypes (as in sized types)

A probabilistic λ-calculus: operational semantics

$$\frac{}{\mathsf{let}\ x\ =\ V\ \mathsf{in}\ M\ \ \to_v\ \ \left\{\,(M[x/V])^1\,\right\}}$$

$$\frac{}{(\lambda x.M)\ \ V\ \ \to_v\ \ \left\{\,(M[x/V])^1\,\right\}}$$

$$\frac{}{(\mathsf{letrec}\ f\ =\ V)\ \left(c\ \overrightarrow{W}\right)\ \ \to_v\ \ \left\{\left(V[f/(\mathsf{letrec}\ f\ =\ V)]\ \left(c\ \overrightarrow{W}\right)\right)^1\right\}}$$

A probabilistic λ-calculus: operational semantics

$$\frac{}{\text{case S } V \text{ of } \{\, \text{S} \to W \mid 0 \to Z \,\} \;\to_v\; \left\{ (W\ V)^1 \right\}}$$

$$\frac{}{\text{case } 0 \text{ of } \{\, \text{S} \to W \mid 0 \to Z \,\} \;\to_v\; \left\{ (Z)^1 \right\}}$$

A probabilistic λ-calculus: operational semantics

$$\frac{}{M \oplus_p N \;\to_v\; \left\{ M^p,\, N^{1-p} \right\}}$$

$$\frac{M \;\to_v\; \left\{ L_i^{p_i} \;\middle|\; i \in I \right\}}{\text{let } x = M \text{ in } N \;\to_v\; \left\{ (\text{let } x = L_i \text{ in } N)^{p_i} \;\middle|\; i \in I \right\}}$$

A probabilistic λ-calculus: operational semantics

$$\frac{\mathscr{D} \overset{VD}{=} \left\{ M_j^{p_j} \mid j \in J \right\} + \mathscr{D}_V \qquad \forall j \in J, \quad M_j \rightarrow_v \mathscr{E}_j}{\mathscr{D} \rightarrow_v \left(\sum_{j \in J} p_j \cdot \mathscr{E}_j\right) + \mathscr{D}_V}$$

For $\mathscr{D}$ a distribution of terms:

$$[\![\mathscr{D}]\!] = \sup_{n \in \mathbb{N}} \left(\left\{ \mathscr{D}_n \mid \mathscr{D} \Rrightarrow_v^n \mathscr{D}_n \right\} \right)$$

where $\Rrightarrow_v^n$ is $\rightarrow_v^n$ followed by projection on values.

We let $[\![M]\!] = [\![\{ M^1 \}]\!]$.

M is AST iff $\sum [\![M]\!] = 1$.

Random walks as probabilistic terms

- Biased random walk:

$$M_{bias} \;=\; \Big(\text{letrec } f \;=\; \lambda x.\text{case } x \text{ of } \Big\{\, \mathsf{S} \to \lambda y.f(y) \oplus_{\frac{2}{3}} (f(\mathsf{S}\,\mathsf{S}\,y)) \;\;\Big|\;\; 0 \to 0 \Big\}\Big) \;\; \underline{n}$$

- Unbiased random walk:

$$M_{unb} \;=\; \Big(\text{letrec } f \;=\; \lambda x.\text{case } x \text{ of } \Big\{\, \mathsf{S} \to \lambda y.f(y) \oplus_{\frac{1}{2}} (f(\mathsf{S}\,\mathsf{S}\,y)) \;\;\Big|\;\; 0 \to 0 \Big\}\Big) \;\; \underline{n}$$

$$\sum [\![\, M_{bias} \,]\!] \;=\; \sum [\![\, M_{unb} \,]\!] \;=\; 1$$

Capture this in a sized type system?

Another term

We also want to capture terms as:

$$M_{nat} \;=\; \Big(\text{letrec } f \;=\; \lambda x.x \oplus_{\frac{1}{2}} \mathsf{S}\ (f\ x)\Big)\ 0$$

of semantics

$$[\![\, M_{nat} \,]\!] \;=\; \Big\{(0)^{\frac{1}{2}}, (\mathsf{S}\ 0)^{\frac{1}{4}}, (\mathsf{S}\ \mathsf{S}\ 0)^{\frac{1}{8}}, \ldots\Big\}$$

summing to 1.

Remark that this recursive function generates the geometric distribution.

Beyond SN terms, towards distribution types

First idea: extend the sized type system with:

$$\text{Choice} \quad \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M \oplus_p N : \sigma}$$

and "unify" types of M and N by subtyping.

Kind of product interpretation of $\oplus$: we can't capture more than SN...

Beyond SN terms, towards distribution types

First idea: extend the sized type system with:

$$\text{Choice} \qquad \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M \oplus_p N : \sigma}$$

and "unify" types of M and N by subtyping.

We get at best

$$f \;:\; \mathsf{Nat}^{\widehat{\widehat{\mathfrak{i}}}} \to \mathsf{Nat}^{\infty} \;\vdash\; \lambda y.f(y) \oplus_{\frac{1}{2}} (f(\mathsf{S}\,\mathsf{S}\,y))) \;:\; \mathsf{Nat}^{\widehat{\mathfrak{i}}} \to \mathsf{Nat}^{\infty}$$

and can't use a variation of the letrec rule on that.

Beyond SN terms, towards distribution types

We will use distribution types, built as follows:

$$\text{Choice} \quad \frac{\Gamma \,|\, \Theta \;\vdash\; M : \mu \qquad \Gamma \,|\, \Psi \;\vdash\; N : \nu \qquad \{\!|\, \mu \,|\!\} = \{\!|\, \nu \,|\!\}}{\Gamma \,|\, \Theta \oplus_p \Psi \;\vdash\; M \oplus_p N : \mu \oplus_p \nu}$$

Now

$$f \;:\; \left\{ \left(\mathsf{Nat}^{\mathfrak{i}} \to \mathsf{Nat}^{\infty}\right)^{\frac{1}{2}},\; \left(\mathsf{Nat}^{\widehat{\widehat{\mathfrak{i}}}} \to \mathsf{Nat}^{\infty}\right)^{\frac{1}{2}} \right\}$$

$$\vdash$$

$$\lambda y. f(y) \oplus_{\frac{1}{2}} (f(\mathsf{S}\,\mathsf{S}\,y))) \;\;:\;\; \mathsf{Nat}^{\widehat{\mathfrak{i}}} \to \mathsf{Nat}^{\infty}$$

Designing the fixpoint rule

$$f \; : \; \left\{ \left(\mathsf{Nat}^{\mathfrak{i}} \to \mathsf{Nat}^{\infty}\right)^{\frac{1}{2}}, \; \left(\mathsf{Nat}^{\widehat{\widehat{\mathfrak{i}}}} \to \mathsf{Nat}^{\infty}\right)^{\frac{1}{2}} \right\}$$

$$\vdash$$

$$\lambda y. f(y) \oplus_{\frac{1}{2}} (f(\mathsf{S}\,\mathsf{S}\,y))) \;\; : \;\; \mathsf{Nat}^{\widehat{\mathfrak{i}}} \to \mathsf{Nat}^{\infty}$$

induces a random walk on $\mathbb{N}$:

- on $n+1$, move to n with probability $\frac{1}{2}$, on $n+2$ with probability $\frac{1}{2}$,
- on 0, loop.

The type system ensures that there is no recursive call from size 0.

Random walk AST (= reaches 0 with proba 1) $\Rightarrow$ termination.

Designing the fixpoint rule

$$\text{LetRec}\quad \frac{\begin{array}{c}\{\!|\,\Gamma\,|\!\} \;=\; \mathsf{Nat} \\ \mathfrak{i} \notin \Gamma \text{ and } \mathfrak{i} \text{ positive in } \nu \\ \color{red}{\left\{\, (\mathsf{Nat}^{\mathfrak{s}_j} \to \nu[\mathfrak{i}/\mathfrak{s}_j])^{p_j} \;\;\middle|\;\; j \in J \,\right\} \text{ induces an AST sized walk}} \\ \Gamma \,|\, f \;:\; \left\{\, (\mathsf{Nat}^{\mathfrak{s}_j} \to \nu[\mathfrak{i}/\mathfrak{s}_j])^{p_j} \;\;\middle|\;\; j \in J \,\right\} \vdash V \;:\; \mathsf{Nat}^{\widehat{\mathfrak{i}}} \to \nu[\mathfrak{i}/\widehat{\mathfrak{i}}]\end{array}}{\Gamma \,|\, \emptyset \vdash \mathsf{letrec}\ f \;=\; V \;:\; \mathsf{Nat}^{\mathfrak{r}} \to \nu[\mathfrak{i}/\mathfrak{r}]}$$

Sized walk: AST is checked by an external PTIME procedure.

Generalized random walks and the necessity of affinity

A crucial feature: our type system is affine.

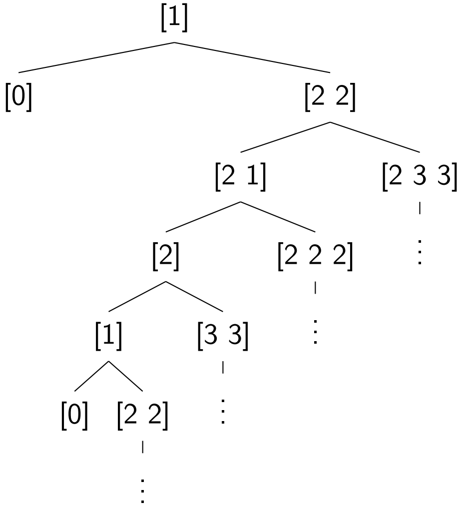
Higher-order symbols occur at most once. Consider:

$$M_{naff} \;=\; \text{letrec } f \;=\; \lambda x.\text{case } x \text{ of } \left\{ \mathsf{S} \rightarrow \lambda y.f(y) \oplus_{\frac{2}{3}} (f(\mathsf{S}\,\mathsf{S}\,y)\,;\, f(\mathsf{S}\,\mathsf{S}\,y)) \;\middle|\; 0 \rightarrow 0 \right\}$$

The induced sized walk is AST.

Generalized random walks and the necessity of affinity

Tree of recursive calls, starting from 1:



Leftmost edges have probability $\frac{2}{3}$; rightmost ones $\frac{1}{3}$.

This random process is not AST.

Problem: modelisation by sized walk only makes sense for affine programs.

Key properties

A nice subject reduction property, and:

Theorem (Typing soundness)

If $\Gamma \mid \Theta \vdash M : \mu$, then M is AST.

Proof by reducibility, using set of candidates parametrized by probabilities.

Conclusion

Main features of the type system:

- Affine type system with distributions of types
- Sized walks induced by the letrec rule and solved by an external PTIME procedure
- Subject reduction + soundness for AST

Next steps:

- type inference (decidable again??)
- extensions with refinement types, non-affine terms
- and use implicit complexity to give type systems for probabilistic complexity classes

Thank you for your attention!

Conclusion

Main features of the type system:

- Affine type system with distributions of types
- Sized walks induced by the letrec rule and solved by an external PTIME procedure
- Subject reduction + soundness for AST

Next steps:

- type inference (decidable again??)
- extensions with refinement types, non-affine terms
- and use implicit complexity to give type systems for probabilistic complexity classes

Thank you for your attention!