Verifying properties of functional programs: from the deterministic to the probabilistic case

Charles Grellois
(partly joint with Dal Lago and Melliès)

FOCUS Team – INRIA & University of Bologna

Séminaire au LIP, ENS Lyon
May 12, 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.

(recall that Turing machines and $\lambda$-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.

(recall that Turing machines and \(\lambda\)-terms are equivalent in expressive power)
Example: imperative factorial

```c
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:

```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: \texttt{map}.

\texttt{map} $\varphi$ [0, 1, 2] returns [$\varphi(0)$, $\varphi(1)$, $\varphi(2)$].

Higher-order: \texttt{map} is a function taking a function $\varphi$ as input.
Advantages of functional programs

Price to pay: analysis of higher-order constructs.

- Function calls + recursivity = deal with stacks of calls → approaches for verification using automata with stacks of stacks of stacks... or with Krivine machines that also have a stack of calls

- Based on \( \lambda \)-calculus with recursion and types: we will use its semantics to do verification

That's the first goal of the talk.

(but that's only an approach among many others)
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: $M \oplus_p N \rightarrow v \{ M^p, N^{1-p} \}$

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.
Probabilistic functional programs

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

What if we add probabilistic constructs?

In this talk: \[ M \oplus_p N \rightarrow_v \{ M^p, N^{1-p} \} \]

Second goal of the talk. Go towards verification of probabilistic functional programs. We give an incomplete method for termination-checking and hints towards verification of more properties.
Using higher-order functions

Bending a coin in the probabilistic functional language Church:

```javascript
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))
```
Roadmap

1. Semantics of linear logic for verification of deterministic functional programs
2. A type system for termination of probabilistic functional programs
3. Towards verification for the probabilistic case?
Semantics of **linear logic** and higher-order **model-checking**

**Linear logic:** a logical system with an emphasis on the notion of *resource*.

**Model-checking:** a key technique in *verification* — where we want to determine *automatically* whether a program satisfies a specification.

**My thesis:** linear logic and its semantics can be enriched to obtain new and cleaner proofs of decidability in higher-order model-checking.
What is model-checking?
The halting problem

A natural question: does a program always terminate?

Undecidable problem (Turing 1936): a machine can not always determine the answer.

What if we use approximations?
Model-checking

Approximate the program \( \rightarrow \) build a model \( M \).

Then, formulate a logical specification \( \varphi \) over the model.

Aim: design a program which checks whether

\[ M \models \varphi. \]

That is, whether the model \( M \) meets the specification \( \varphi \).
An example

$$\begin{align*}
\text{Main} & = \text{Listen Nil} \\
\text{Listen } x & = \text{if } \text{end_signal}() \text{ then } x \\
& \quad \text{else } \text{Listen received_data}() :: x
\end{align*}$$
An example

\[
\begin{align*}
\text{Main} & = \text{Listen Nil} \\
\text{Listen } x & = \begin{cases} 
\text{if } \text{end_signal}() \text{ then } x \\
\text{else Listen received data()::x}
\end{cases}
\end{align*}
\]

A tree model:

We abstracted conditionals and datatypes. The approximation contains a non-terminating branch.
Finite representations of infinite trees

is not regular: it is not the unfolding of a finite graph as
Finite representations of infinite trees

but it is represented by a higher-order recursion scheme (HORS).
Modeling functional programs using higher-order recursion schemes
Higher-order recursion schemes

\[ G = \begin{cases} 
S &= L \text{ Nil} \\
L \ x &= \text{if } x (L \ (\text{data } x)) 
\end{cases} \]

Rewriting starts from the start symbol \( S \):

\[ S \rightarrow_G L \text{ Nil} \]
Higher-order recursion schemes

\[ G = \begin{cases} 
S &= \text{L Nil} \\
\text{L x} &= \text{if x (L (data x))} 
\end{cases} \]
Higher-order recursion schemes

\[ G = \begin{cases} 
    S & = \text{L Nil} \\
    \text{L} \ x & = \text{if } x (\text{L} \ (\text{data} \ x)) 
\end{cases} \]
Higher-order recursion schemes

\[ G = \left\{ \begin{array}{l}
  S = L \text{ Nil} \\
  L \times = \text{if } x (L (\text{data } x)) \\
\end{array} \right. \]

\[ \langle G \rangle = \text{if } \]

\[ \text{Nil} \quad \text{if} \quad \]

\[ \text{data} \quad \text{if} \quad \]

\[ \text{Nil} \quad \text{data} : \]

\[ \text{data} \quad \text{Nil} \]
Higher-order recursion schemes

\[
\mathcal{G} = \left\{ \begin{array}{ll}
S &= \text{L Nil} \\
\text{L } x &= \text{if } x (\text{L } \text{data } x)
\end{array} \right.
\]

HORS can alternatively be seen as simply-typed \( \lambda \)-terms with simply-typed recursion operators \( Y_\sigma : (\sigma \rightarrow \sigma) \rightarrow \sigma \).

They are also equi-expressive to pushdown automata with stacks of stacks of stacks... and a collapse operation.
Alternating parity tree automata

Checking specifications over trees
Monadic second order logic

MSO is a common logic in verification, allowing to express properties as:

“all executions halt”

“a given operation is executed infinitely often in some execution”

“every time data is added to a buffer, it is eventually processed”
Alternating parity tree automata

Checking whether a formula holds can be performed using an automaton.

For an MSO formula $\varphi$, there exists an equivalent APT $A_\varphi$ s.t.

$$\langle G \rangle \models \varphi \text{ iff } A_\varphi \text{ has a run over } \langle G \rangle.$$

$$\text{APT} = \text{alternating tree automata (ATA) + parity condition}.$$
Alternating tree automata

ATA: non-deterministic tree automata whose transitions may duplicate or drop a subtree.

Typically: $\delta(q_0, \text{if}) = (2, q_0) \land (2, q_1)$. 
Alternating tree automata

ATA: non-deterministic tree automata whose transitions may duplicate or drop a subtree.

Typically: $\delta(q_0, \text{if}) = (2, q_0) \land (2, q_1)$. 

\[
\begin{array}{c}
\text{if} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\rightarrow A_\varphi \\
\text{if} \\
\end{array}
\begin{array}{c}
\text{if} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\end{array}
\begin{array}{c}
\text{if} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\end{array}
\begin{array}{c}
\text{if} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\text{data} \\
\text{Nil} \\
\end{array}
\]
Alternating parity tree automata

Each state of an APT is attributed a color

\[ \Omega(q) \in \text{Col} \subseteq \mathbb{N} \]

An infinite branch of a run-tree is winning iff the maximal color among the ones occurring infinitely often along it is even.
Alternating parity tree automata

Each state of an APT is attributed a color

$$\Omega(q) \in \text{Col} \subseteq \mathbb{N}$$

An infinite branch of a run-tree is winning iff the maximal color among the ones occurring infinitely often along it is even.

A run-tree is winning iff all its infinite branches are.

For a MSO formula $\varphi$:

$$A_\varphi \text{ has a winning run-tree over } \langle G \rangle \iff \langle G \rangle \models \varphi.$$
The higher-order model-checking problems
The (local) HOMC problem

**Input:** HORS $\mathcal{G}$, formula $\varphi$.

**Output:** true if and only if $\langle \mathcal{G} \rangle \models \varphi$.

Example: $\varphi = \text{ “there is an infinite execution”}$

Output: true.
The (local) HOMC problem

**Input:** HORS $\mathcal{G}$, formula $\varphi$.

**Output:** true if and only if $\langle \mathcal{G} \rangle \models \varphi$.

Example: $\varphi = \text{"there is an infinite execution"}$

```
if
    Nil
    if
        data
            if
                Nil
                data
                    data
                        data
                            Nil
```

**Output:** true.
The global HOMC problem

**Input:** HORS $\mathcal{G}$, formula $\varphi$.

**Output:** a HORS $\mathcal{G}^\bullet$ producing a marking of $\langle \mathcal{G} \rangle$.

Example: $\varphi = " \text{there is an infinite execution} "$

Output: $\mathcal{G}^\bullet$ of value tree:
The selection problem

Input:  HORS $\mathcal{G}$, APT $A$, state $q \in Q$.

Output: $false$ if there is no winning run of $A$ over $\langle \mathcal{G} \rangle$. Else, a HORS $\mathcal{G}^q$ producing a such a winning run.

Example: $\varphi = " \text{there is an infinite execution "}$, $q_0$ corresponding to $\varphi$

Output: $\mathcal{G}^{q_0}$ producing

```
if^{q_0}
  |
if^{q_0}
  |
if^{q_0}
  |
  ...
```
Our line of work (joint with Melliès)

These three problems are **decidable**, with elaborate proofs (often) relying on **semantics**.

**Our contribution:** an excavation of the semantic roots of HOMC, at the light of **linear logic**, leading to refined and clarified proofs.
Recognition by homomorphism

Where semantics comes into play
Automata and recognition

For the usual finite automata on words: given a regular language $L \subseteq A^*$,
there exists a finite automaton $A$ recognizing $L$

if and only if...

there exists a finite monoid $M$, a subset $K \subseteq M$
and a homomorphism $\varphi : A^* \rightarrow M$ such that $L = \varphi^{-1}(K)$. 
Automata and recognition

The picture we want:

(\textit{after Aehlig 2006, Salvati 2009})

but with \textbf{recursion} and \textit{w.r.t.} an APT.
Intersection types and alternation

A first connection with linear logic
Alternating tree automata and intersection types

A key remark (Kobayashi 2009):

\[ \delta(q_0, \text{if}) = (2, q_0) \land (2, q_1) \]

can be seen as the intersection typing

\[ \text{if} : \emptyset \rightarrow (q_0 \land q_1) \rightarrow q_0 \]

refining the simple typing

\[ \text{if} : o \rightarrow o \rightarrow o \]
Alternating tree automata and intersection types

In a derivation typing the tree $\text{if } T_1 \ T_2$:

\[
\begin{align*}
\delta & \quad \emptyset \vdash \text{if } \emptyset \to (q_0 \land q_1) \to q_0 \quad \emptyset \\
\text{App} & \quad \emptyset \vdash \text{if } T_1 : (q_0 \land q_1) \to q_0 \\
\text{App} & \quad \emptyset \vdash \text{if } T_1 \ T_2 : q_0
\end{align*}
\]

Intersection types naturally lift to higher-order – and thus to $G$, which finitely represents $\langle G \rangle$.

**Theorem (Kobayashi 2009)**

$\vdash G : q_0$ iff the ATA $A_\varphi$ has a run-tree over $\langle G \rangle$. 
A closer look at the Application rule

In the intersection type system:

\[ \Delta \vdash t : (\theta_1 \land \cdots \land \theta_n) \rightarrow \theta \]
\[ \Delta_i \vdash u : \theta_i \]
\[ \Delta, \Delta_1, \ldots, \Delta_n \vdash t u : \theta \]

This rule could be decomposed as:

\[ \Delta \vdash t : (\land_{i=1}^n \theta_i) \rightarrow \theta' \]
\[ \Delta_i \vdash u : \theta_i \quad \forall i \in \{1, \ldots, n\} \]
\[ \Delta_1, \ldots, \Delta_n \vdash u : \land_{i=1}^n \theta_i \]
\[ \Delta, \Delta_1, \ldots, \Delta_n \vdash t u : \theta' \]
A closer look at the Application rule

In the intersection type system:

\[
\begin{array}{c}
\frac{\Delta \vdash t : (\theta_1 \land \cdots \land \theta_n) \rightarrow \theta}{\Delta, \Delta_1, \ldots, \Delta_n \vdash t u : \theta} \\
\text{App}
\end{array}
\]

This rule could be decomposed as:

\[
\begin{array}{c}
\frac{\Delta \vdash t : (\bigwedge_{i=1}^n \theta_i) \rightarrow \theta'}{\Delta, \Delta_1, \ldots, \Delta_n \vdash t u : \theta'} \\
\frac{\Delta_i \vdash u : \theta_i}{\Delta_1, \ldots, \Delta_n \vdash u : \bigwedge_{i=1}^n \theta_i} \\
\forall i \in \{1, \ldots, n\}
\end{array}
\]

Right \land
A closer look at the Application rule

\[
\Delta \vdash t : (\bigwedge_{i=1}^{n} \theta_i) \rightarrow \theta' \\
\Delta, \Delta_1, \ldots, \Delta_n \vdash u : \bigwedge_{i=1}^{n} \theta_i \\
\Delta_i \vdash u : \theta_i \quad \forall i \in \{1, \ldots, n\}
\]

Right \ \bigwedge

Linear decomposition of the intuitionistic arrow:

\[
A \Rightarrow B = !A \multimap B
\]

Two steps: duplication / erasure, then linear use.

Right \ \bigwedge \text{ corresponds to the Promotion rule of indexed linear logic. (see G.-Melliès, ITRS 2014)}
Intersection types and semantics of linear logic

\[ A \Rightarrow B = ! A \multimap B \]

Two interpretations of the exponential modality:

**Qualitative models** (Scott semantics)

\[ ! A = P_{\text{fin}}(A) \]

\[ [ o \Rightarrow o ] = P_{\text{fin}}(Q) \times Q \]

\[ \{ q_0, q_0, q_1 \} = \{ q_0, q_1 \} \]

**Quantitative models** (Relational semantics)

\[ ! A = M_{\text{fin}}(A) \]

\[ [ o \Rightarrow o ] = M_{\text{fin}}(Q) \times Q \]

\[ [ q_0, q_0, q_1 ] \neq [ q_0, q_1 ] \]

Order closure

Unbounded multiplicities
An example of interpretation

In $\text{Rel}$, one denotation:

$$([q_0, q_1, q_1], [q_1], q_0)$$

In $\text{ScottL}$, a set containing the principal type

$$([\{q_0, q_1\}, \{q_1\}, q_0)$$

but also

$$([\{q_0, q_1, q_2\}, \{q_1\}, q_0)$$

and

$$([\{q_0, q_1\}, \{q_0, q_1\}, q_0)$$

and $\ldots$. 
Let $t$ be a term normalizing to a tree $\langle t \rangle$ and $A$ be an alternating automaton.

$A$ accepts $\langle t \rangle$ from $q \iff q \in \llbracket t \rrbracket \iff \emptyset \vdash t : q :: o$

Extension with recursion and parity condition?
Adding parity conditions to the type system
An example of colored intersection type

Set $\Omega(q_0) = 0$ and $\Omega(q_1) = 1$.

\[
\begin{align*}
\lambda x \\
\lambda y \\
a \quad q_1
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
a \\
\begin{array}{c}
x \\
\quad q_0 \\
y \quad q_1
\end{array}
\end{array} & & \begin{array}{c}
a \\
\begin{array}{c}
x \\
\quad q_1
\end{array}
\end{array}
\end{align*}
\]

has now type

\[
\square_0 q_0 \land \square_1 q_1 \rightarrow \square_1 q_1 \rightarrow q_1
\]

Note the color 0 on $q_0$...
A type-system for verification

We devise a type system capturing all MSO:

**Theorem (G.-Melliès 2014, from Kobayashi-Ong 2009)**

\[ S : q_0 \vdash S : q_0 \text{ admits a winning typing derivation iff the alternating parity automaton } A \text{ has a winning run-tree over } \langle G \rangle. \]

We obtain decidability by considering idempotent types.

Our reformulation
- shows the modal nature of □ (in the sense of S4),
- internalizes the parity condition,
- paves the way for semantic constructions.
Colored semantics

We extend:

- \textit{Rel} with \textbf{countable} multiplicities, \textbf{coloring} and an \textbf{inductive-coinductive} fixpoint
- \textit{ScottL} with \textbf{coloring} and an \textbf{inductive-coinductive} fixpoint.

Methodology: think in the relational semantics, and adapt to the Scott semantics using Ehrhard’s 2012 result:

the \textbf{finitary} model \textit{ScottL} is the extensional collapse of \textit{Rel}.
Finitary semantics

In ScottL, we define □, λ and Y using downward-closures. ScottL↓ is a model of the λY-calculus.

Theorem

An APT A has a winning run from q₀ over ⟨G⟩ if and only if

\[ q₀ \in \llbracket \lambda (G) \rrbracket. \]

Corollary

The local higher-order model-checking problem is decidable (and is n-EXPTIME complete).

We could also obtain global model-checking and selection.

Similar model-theoretic results were obtained by Salvati and Walukiewicz the same year.
Probabilistic Termination

Checking a first property on probabilistic program
Motivations

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

- **Quantitative notion of termination**: almost-sure termination (AST)

- AST has been studied for imperative programs in the last years...

- ...but what about the **functional** probabilistic languages?

We introduce a **monadic, affine sized type system** sound for AST.
Sized types: the deterministic case

Simply-typed $\lambda$-calculus is strongly normalizing (SN).

\[
\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \quad \frac{}{\Gamma \vdash \lambda x. M : \sigma \to \tau}
\]

\[
\frac{}{\Gamma \vdash M : \sigma \to \tau} \quad \frac{}{\Gamma \vdash N : \sigma} \frac{}{\Gamma \vdash M \ N : \tau}
\]

where $\sigma, \tau ::= o \mid \sigma \to \tau$.

Forbids the looping term $\Omega = (\lambda x. x \ x)(\lambda x. x \ x)$.

**Strong normalization**: all computations terminate.
Simply-typed $\lambda$-calculus is strongly normalizing (SN).

No longer true with the letrec construction...

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

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: \( s, r ::= \ i \ | \ \infty \ | \ ^\wedge s \)

+ size comparison underlying subtyping. Notably \( \ ^\wedge \infty \equiv \infty \).

Idea: \( k \) successors = at most \( k \) constructors.
- \( \hat{\text{Nat}}^i \) is 0,
- \( \hat{\text{Nat}}^i \) is 0 or \( S \ 0 \),
- \( \ldots \)
- \( \hat{\text{Nat}}^\infty \) is any natural number. Often denoted simply \( \text{Nat} \).

The same for lists,\ldots
Sized types: the deterministic case

Sizes: \( s, r ::= i \mid \infty \mid \hat{s} \)

+ size comparison underlying subtyping. Notably \( \hat{\infty} \equiv \infty \).

Fixpoint rule:

\[
\Gamma, f : \text{Nat}^i \rightarrow \sigma \vdash M : \text{Nat}^\hat{i} \rightarrow \sigma[i/\hat{i}] \quad \text{i pos } \sigma
\]

\[
\Gamma \vdash \text{letrec } f = M : \text{Nat}^s \rightarrow \sigma[i/s]
\]

“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: \[ s, r ::= i \mid \infty \mid \hat{s} \]

+ size comparison underlying subtyping. Notably \( \hat{\infty} \equiv \infty \).

Fixpoint rule:

\[
\frac{
\Gamma, f : \text{Nat}^i \to \sigma \vdash M : \text{Nat}^{\hat{i}} \to \sigma[i/\hat{i}] \quad i \text{ pos } \sigma
}{
\Gamma \vdash \text{letrec } f = M : \text{Nat}^s \to \sigma[i/s]
}
\]

Typable \( \implies \) SN. Proof using reducibility candidates.

Decidable type inference.
Sized types: example in the deterministic case

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

\[
\text{plus} \equiv (\text{letrec} \quad \text{plus} : \text{Nat}' \rightarrow \text{Nat} \rightarrow \text{Nat} = \\
\quad \lambda x : \text{Nat}'. \quad \lambda y : \text{Nat}'. \quad \text{case } x \text{ of } \left\{ \begin{array}{l}
\quad \text{o } \Rightarrow \ y \\
\quad | \text{s } \Rightarrow \lambda x' : \text{Nat}'. \quad \text{s } \underbrace{\text{plus } x' \ y}_: \text{Nat}
\end{array} \right\}
\quad \}
\quad ) : \quad \text{Nat}' \rightarrow \text{Nat} \rightarrow \text{Nat}
\]

The case rule ensures that the size of \( x' \) is lesser than the one of \( x \). Size decreases during recursive calls \( \Rightarrow \) SN.
A probabilistic $\lambda$-calculus

\[ M, N, \ldots \ ::= \ V \mid V \; V \mid \text{let } x = M \text{ in } N \mid M \oplus_p N \]
\[ \quad \mid \text{case } V \text{ of } \{ S \rightarrow W \mid 0 \rightarrow Z \} \]

\[ V, W, Z, \ldots \ ::= \ x \mid 0 \mid S \; V \mid \lambda x. M \mid \text{letrec } f = V \]

- Formulation equivalent to $\lambda$-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 $\lambda$-calculus: operational semantics

\[
\text{let } x = V \text{ in } M \Rightarrow_v \left\{ (M[x/V])^1 \right\}
\]

\[
(\lambda x. M) V \Rightarrow_v \left\{ (M[x/V])^1 \right\}
\]

\[
(\text{letrec } f = V) \left( c \ W \right) \Rightarrow_v \left\{ \left( V[f/(\text{letrec } f = V)] \ (c \ W) \right)^1 \right\}
\]
A probabilistic $\lambda$-calculus: operational semantics

\[\text{case } S \ V \text{ of } \{ S \rightarrow W \mid 0 \rightarrow Z \} \rightarrow_v \{ (W \ V)^1 \}\]

\[\text{case } 0 \text{ of } \{ S \rightarrow W \mid 0 \rightarrow Z \} \rightarrow_v \{ (Z)^1 \}\]
A probabilistic $\lambda$-calculus: operational semantics

\[
\begin{align*}
M \oplus_p N & \rightarrow_v \{ M^p, N^{1-p} \} \\
M & \rightarrow_v \{ L_i^p \mid i \in I \} \\
\text{let } x = M \text{ in } N & \rightarrow_v \{ (\text{let } x = L_i \text{ in } N)^{p_i} \mid i \in I \}
\end{align*}
\]
A probabilistic $\lambda$-calculus: operational semantics

$$
\mathcal{D} \overset{VD}{=} \left\{ M^j_p \mid j \in J \right\} + \mathcal{D}_V \quad \forall j \in J, \ M_j \rightarrow_v \mathcal{E}_j
$$

$$
\mathcal{D} \rightarrow_v \left( \sum_{j \in J} p_j \cdot \mathcal{E}_j \right) + \mathcal{D}_V
$$

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

$$
\mathbb{E}[\mathcal{D}] = \sup_{n \in \mathbb{N}} \left( \left\{ \mathcal{D}_n \mid \mathcal{D} \Rightarrow^n_v \mathcal{D}_n \right\} \right)
$$

where $\Rightarrow^n_v$ is $\rightarrow^n_v$ followed by projection on values.

We let $\mathbb{E}[M] = \mathbb{E}[\left\{ M^1 \right\}]$.

$M$ is AST iff $\sum \mathbb{E}[M] = 1$. 
Random walks as probabilistic terms

- **Biased** random walk:

\[ M_{bias} = \left( \text{letrec } f = \lambda x.\text{case } x \text{ of } \left\{ \begin{array}{l} S \to \lambda y.f(y) \oplus \frac{2}{3} (f(S S y)) \mid 0 \to 0 \end{array} \right\} \right)^n \]

- **Unbiased** random walk:

\[ M_{unb} = \left( \text{letrec } f = \lambda x.\text{case } x \text{ of } \left\{ \begin{array}{l} S \to \lambda y.f(y) \oplus \frac{1}{2} (f(S S y)) \mid 0 \to 0 \end{array} \right\} \right)^n \]

\[ \sum \llbracket M_{bias} \rrbracket = \sum \llbracket M_{unb} \rrbracket = 1 \]

Capture this in a sized type system?
Another term

We also want to capture terms as:

\[ M_{\text{nat}} = \left( \text{letrec } f = \lambda x. x \oplus \frac{1}{2} S (f x) \right) 0 \]

of semantics

\[ \llbracket M_{\text{nat}} \rrbracket = \left\{ (0)_{\frac{1}{2}}, (S \ 0)_{\frac{1}{4}}, (S \ S \ 0)_{\frac{1}{8}}, \ldots \right\} \]

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:

\[ \Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma \]

Choice: \[ \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M \oplus_p N : \sigma} \]

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:

\[
\frac{
\Gamma \vdash M : \sigma \\
\Gamma \vdash N : \sigma
}{
\Gamma \vdash M \oplus_p N : \sigma
}
\]

and “unify” types of \(M\) and \(N\) by subtyping.

We get at best

\[
f : \text{Nat}^i \rightarrow \text{Nat}^\infty
\]


\[
\lambda x. \text{case } x \text{ of } \begin{cases}
S & \rightarrow \lambda y. f(y) \oplus \frac{1}{2} (f(S S y)) \\
0 & \rightarrow 0
\end{cases} : \text{Nat}^i \rightarrow \text{Nat}^\infty
\]

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:

\[
\begin{array}{c}
\text{Choice} \\
\frac{\Gamma \mid \Theta \vdash M : \mu \quad \Gamma \mid \Psi \vdash N : \nu \quad \{\mu\} = \{\nu\}}{
\Gamma \mid \Theta \oplus_p \Psi \vdash M \oplus_p N : \mu \oplus_p \nu}
\end{array}
\]

Now

\[
f : \left\{ (\text{Nat}^i \to \text{Nat}^\infty)^{\frac{1}{2}}, \left(\text{Nat}^{\tilde{i}} \to \text{Nat}^\infty\right)^{\frac{1}{2}} \right\}
\]

\[
\vdash \\
\lambda x. \text{case } x \text{ of } \left\{ S \to \lambda y. f(y) \oplus_{\frac{1}{2}} (f(S \ S y)) \mid 0 \to 0 \right\} : \text{Nat}^{\tilde{i}} \to \text{Nat}^\infty
\]
Designing the fixpoint rule

\[ f : \left\{ \left( \text{Nat}^i \rightarrow \text{Nat}^\infty \right) \frac{1}{2}, \left( \text{Nat}^\hat{i} \rightarrow \text{Nat}^\infty \right) \frac{1}{2} \right\} \]

\[ \vdash \lambda x. \text{case} \ x \ \text{of} \ \left\{ \begin{array}{c} S \rightarrow \lambda y.f(y) \oplus \frac{1}{2} (f(S\ S\ y)) \mid 0 \rightarrow 0 \end{array} \right\} : \text{Nat}^\hat{i} \rightarrow \text{Nat}^\infty \]

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 (\( \equiv \) reaches 0 with proba 1) \( \Rightarrow \) termination.
Designing the fixpoint rule

\[ \{ \Gamma \} = \text{Nat} \]

\[ i \notin \Gamma \text{ and } i \text{ positive in } \nu \]

\[ \{ (\text{Nat}^s_j \rightarrow \nu[i/s_j])^p_j \mid j \in J \} \text{ induces an AST sized walk} \]

LetRec

\[
\frac{\Gamma \vdash f : \{ (\text{Nat}^s_j \rightarrow \nu[i/s_j])^p_j \mid j \in J \} \vdash V : \text{Nat}^\hat{i} \rightarrow \nu[i/\hat{i}]}
{\Gamma \vdash \text{letrec } f = V : \text{Nat}^\tau \rightarrow \nu[i/\tau]}
\]

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:

\[ M_{\text{naff}} = \text{letrec } f = \lambda x.\text{case } x \text{ of } \begin{cases} S \rightarrow \lambda y. f(y) \oplus \frac{2}{3} (f(S S y); f(S S y)) & | 0 \rightarrow 0 \end{cases} \]

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)**

\[ \text{If } \Gamma \mid \Theta \vdash M : \mu, \text{ then } M \text{ is AST.} \]

Proof by reducibility, using set of candidates parametrized by probabilities.
Conclusion of this part

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
Towards Higher-Order Probabilistic Verification
IntList random_list() {
    IntList list = Nil;
    while(rand() > 0.1) {
        list := rand_int()::list;
    }
    return list;
}
Probabilistic HOMC

Allows to represent probabilistic programs.

And to define higher-order regular Markov Decision Processes: those bisimilar to their encoding represented by a HORS.

(encoding of probabilities + payoffs in symbols)
Current directions

- In a talk at Singapore last September: a notion of probabilistic tree automata + a sound and complete type system, but
  - no result of complexity / decidability
  - automata not formally related to logic, and the corresponding logic seems a bit “weak” from a probabilistic perspective

- Two leads:
  - Probabilistic $\mu$-calculus of Castro, Kilmurray, Piterman + connection with obligation games (Chatterjee-Piterman)
  - Recursive Markov Chains (Etessami-Yannakakis) for which there exists verification result (wrt Büchi automata)
Conclusions

- Multiple approaches for higher-order model-checking, from theory to practice. Here, using semantics of linear logic to make the theory clearer.

- A type system for checking termination of affine probabilistic programs.

- Some preliminary hints to check for more than just termination properties.

Thank you for your attention!
Conclusions

- Multiple approaches for higher-order model-checking, from theory to practice. Here, using semantics of linear logic to make the theory clearer.

- A type system for checking termination of affine probabilistic programs.

- Some preliminary hints to check for more than just termination properties.

Thank you for your attention!