Vérification des programmes d'ordre supérieur

Charles Grellois
(travaux réalisés avec Dal Lago et Melliès)

Aix-Marseille Université - LSIS

Visite des étudiants de l'ENS Paris-Saclay
23 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.

(recall that 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.

(recall that 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.

Advantages of functional programs

Price to pay: analysis of higher-order constructs.

- Function calls + recursivity = deal with stacks of stacks... of calls
- Based on λ-calculus with recursion and types: we can use its semantics to do verification

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.

To be fully general: add 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))
```

Roadmap

1. Semantics of linear logic for verification of deterministic functional programs
2. A type system for termination of probabilistic functional programs

Modeling functional programs using higher-order recursion schemes

Model-checking

Approximate the program $\longrightarrow$ build a model $\mathcal{M}$.

Then, formulate a logical specification φ over the model.

Aim: design a program which checks whether

$$\mathcal{M} \ \models \ \varphi.$$

That is, whether the model $\mathcal{M}$ meets the specification φ.

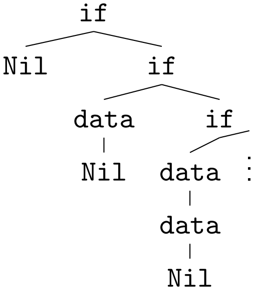
An example

$$
\begin{array}{rcl}
\texttt{Main} & = & \texttt{Listen Nil} \\
\texttt{Listen}\ x & = & \texttt{if end\_signal() then}\ x \\
 & & \texttt{else Listen received\_data()} :: x
\end{array}
$$

An example

$$\begin{array}{rcl} \texttt{Main} & = & \texttt{Listen Nil} \\ \texttt{Listen } x & = & \texttt{if end\_signal() then } x \\ & & \texttt{else Listen received\_data()::}x \end{array}$$

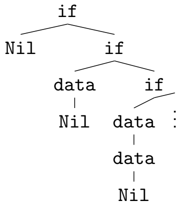
A tree model:



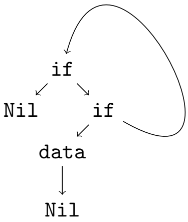
We abstracted conditionals and datatypes.

The approximation contains a non-terminating branch.

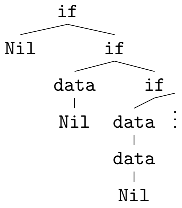
Finite representations of infinite trees



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).

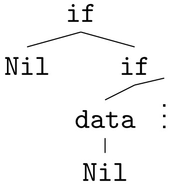
Higher-order recursion schemes

$$\begin{array}{rcl} \texttt{Main} & = & \texttt{Listen Nil} \\ \texttt{Listen}\ x & = & \texttt{if end\_signal() then}\ x \\ & & \texttt{else Listen received\_data()} :: x \end{array}$$

is abstracted as

$$\mathcal{G} = \left\{ \begin{array}{lcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L}\ x & = & \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )\ ) \end{array} \right.$$

which represents the higher-order tree of actions



Higher-order recursion schemes

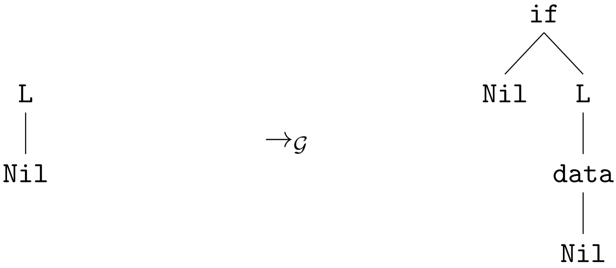
$$\mathcal{G} \;=\; \begin{cases} \texttt{S} & = \quad \texttt{L Nil} \\ \texttt{L}\ x & = \quad \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )\ ) \end{cases}$$

Rewriting starts from the start symbol S:

$$\texttt{S} \quad \rightarrow_{\mathcal{G}} \quad \begin{array}{c} \texttt{L} \\ | \\ \texttt{Nil} \end{array}$$

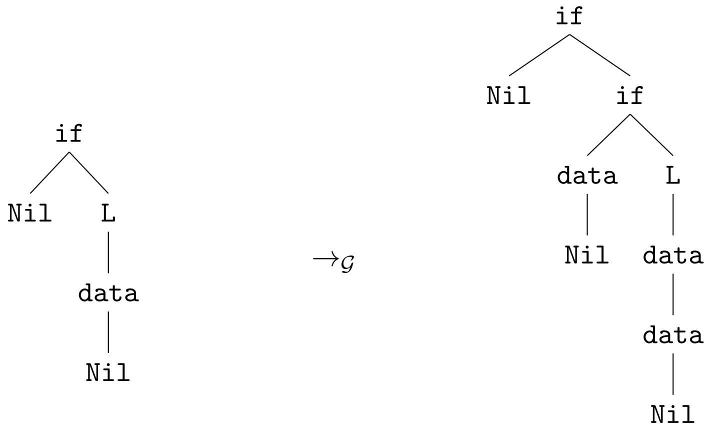
Higher-order recursion schemes

$$\mathcal{G} \quad = \quad \begin{cases} \mathtt{S} & = \quad \mathtt{L}\ \mathtt{Nil} \\ \mathtt{L}\ x & = \quad \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ )\ ) \end{cases}$$



Higher-order recursion schemes

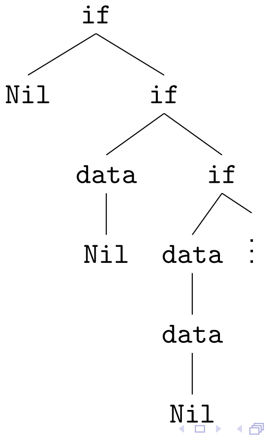
$$\mathcal{G} = \begin{cases} \texttt{S} & = \quad \texttt{L Nil} \\ \texttt{L}\ x & = \quad \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )\ ) \end{cases}$$



Higher-order recursion schemes

$$\mathcal{G} = \begin{cases} \texttt{S} & = \texttt{L Nil} \\ \texttt{L } x & = \texttt{if } x \, (\texttt{L } (\texttt{data } x\,)\,) \end{cases}$$

$\langle \mathcal{G} \rangle =$



Higher-order recursion schemes

$$\mathcal{G} = \begin{cases} \texttt{S} & = \texttt{L Nil} \\ \texttt{L}\ x & = \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )\ ) \end{cases}$$

HORS can alternatively be seen as simply-typed λ-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 φ, there exists an equivalent APT $\mathcal{A}_\varphi$ s.t.

$$\langle \mathcal{G} \rangle \models \varphi \quad \text{iff} \quad \mathcal{A}_\varphi \text{ has a run over } \langle \mathcal{G} \rangle.$$

APT = alternating tree automata (ATA) + parity condition.

Alternating tree automata

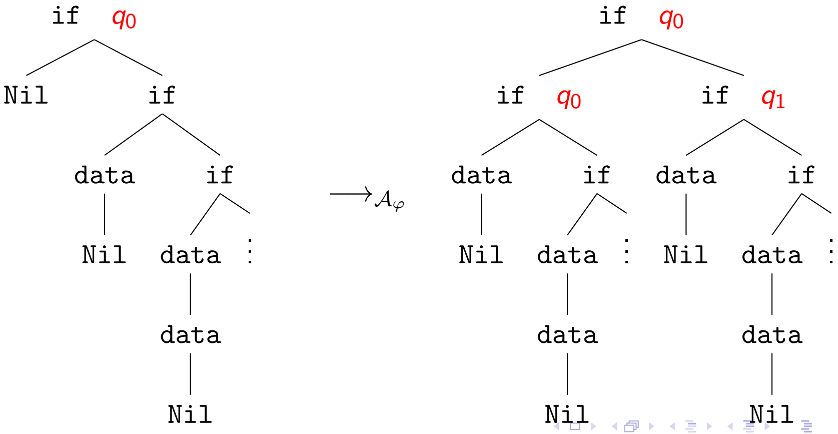
ATA: non-deterministic tree automata whose transitions may duplicate or drop a subtree.

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

Alternating tree automata

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

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

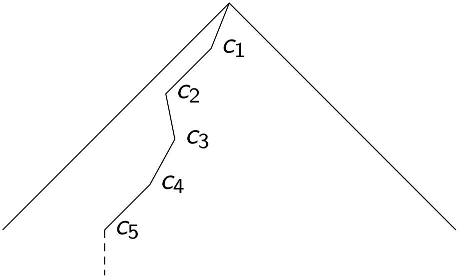


Alternating parity tree automata

Each state of an APT is attributed a color

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

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



Alternating parity tree automata

Each state of an APT is attributed a color

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

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

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

For a MSO formula φ:

$$\mathcal{A}_\varphi \text{ has a winning run-tree over } \langle \mathcal{G} \rangle \qquad \text{iff} \qquad \langle \mathcal{G} \rangle \vDash \varphi.$$

The higher-order model-checking problems

The (local) HOMC problem

Input: HORS $\mathcal{G}$, formula φ.

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

Example: φ = " there is an infinite execution "

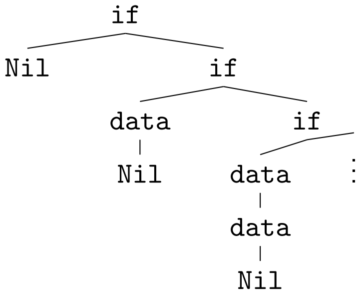
Output: `true`.

The (local) HOMC problem

Input: HORS $\mathcal{G}$, formula φ.

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

Example: φ = " there is an infinite execution "



Output: `true`.

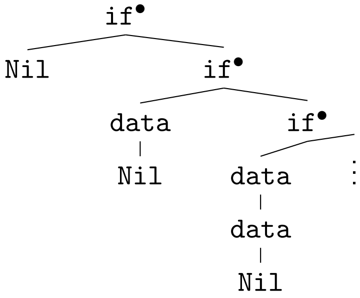
The global HOMC problem

Input: HORS $\mathcal{G}$, formula φ.

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

Example: $\varphi \;=\;$ " there is an infinite execution "

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



The selection problem

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

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

Example: $\varphi \;=\;$ " there is an infinite execution ", q_0 corresponding to φ

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

$$\begin{array}{c} \mathtt{if}^{q_0} \\ | \\ \mathtt{if}^{q_0} \\ | \\ \mathtt{if}^{q_0} \\ | \\ \vdots \end{array}$$

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

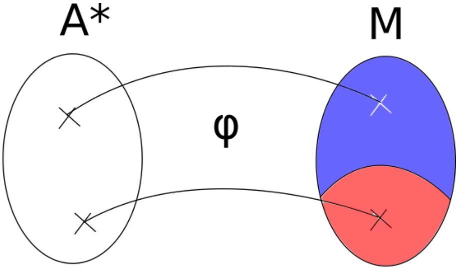
Where semantics comes into play

Automata and recognition

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

there exists a finite automaton $\mathcal{A}$ recognizing L

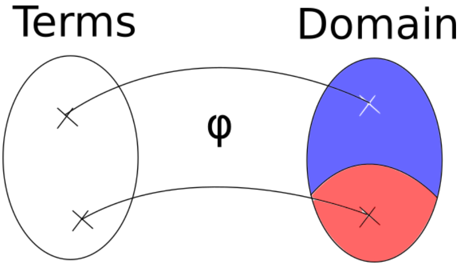
if and only if. . .



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

Automata and recognition

The picture we want:



(after Aehlig 2006, Salvati 2009)

but with recursion and w.r.t. an APT.

Our contribution

Using semantics of linear logic

Finitary semantics

ScottL is a model of linear logic, from which we obtain $ScottL_{\lightning}$, a model of the λY-calculus (the algebraic structures we look for!).

Theorem

An APT $\mathcal{A}$ has a winning run from q_0 over $\langle \mathcal{G} \rangle$ if and only if

$$q_0 \in [\![\mathcal{G}]\!].$$

Corollary

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

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

Work together on the selection property?

Probabilistic Termination

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 λ-calculus is strongly normalizing (SN).

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

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

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.

Sized types: the deterministic case

Simply-typed λ-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 λ-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: $\mathfrak{s}, \mathfrak{r} \quad ::= \quad \mathfrak{i} \mid \infty \mid \widehat{\mathfrak{s}}$

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

Idea: k successors = at most k constructors.

- $\mathrm{Nat}^{\widehat{\mathfrak{i}}}$ is 0,
- $\mathrm{Nat}^{\widehat{\widehat{\mathfrak{i}}}}$ 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:

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

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

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

Fixpoint rule:

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

Sound for SN: typable $\Rightarrow$ SN.

Decidable type inference (implies incompleteness).

Sized types: example in the deterministic case

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

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

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

A probabilistic λ-calculus

With Dal Lago, we studied a call-by-value λ-calculus extended with a probabilistic choice operator.

We designed a type system, inspired from sized types, in which

typability $\Rightarrow$ AST

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:

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

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

This is checked by our type system.

Another term

We also 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.

A Perspective

The sized type system for the deterministic case has a decidable type inference.

We conjecture that its extension to the probabilistic case should be decidable too. We could do it together!

Another Perspective

If you like proof theory, a new team called LIRICA has started in Marseilles. With Nicola Olivetti, we propose to work on non-normal intuitionnistic modal logics.

- Modal: special operators change the meaning of formulas. Example, in a temporal perspective: $\Box\varphi$ means that φ is true all the time.
- Non-normal: some of the usual axioms of modal logics are not assumed to be true.

Proposition: for one of these logics, there exists a semantics but no known proof theory. Let's design a sound-and-complete associated calculus together!

Conclusions

- We can use semantics to do verification of functional programs, by defining appropriate models.
- Possible perspective: selection property
- We can give a type system for functional programs ensuring almost-sure termination.
- Possible perspective: type inference algorithm
- Last perspective: work on proof theory of modal logics

Thank you for your attention!

Conclusions

- We can use semantics to do verification of functional programs, by defining appropriate models.
- Possible perspective: selection property
- We can give a type system for functional programs ensuring almost-sure termination.
- Possible perspective: type inference algorithm
- Last perspective: work on proof theory of modal logics

Thank you for your attention!