Verification of (probabilistic) functional programs

Charles Grellois

Aix-Marseille Université

MABioS team seminar
June 3rd, 2019

Introduction

- Type theory: allows to label parts of a program to prove properties about it.
- Model-checking: abstract a program as a model, and (try to) prove automatically properties about it.
- Both will meet in this talk, to allow the verification of functional programs, in which functions can take functions as inputs.

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.

Roadmap

1. A few words on the λ-calculus and an introduction to type systems
2. Intersection type systems for higher-order model-checking
3. Towards the verification of probabilistic programs

A few words on the λ-calculus

Definition, simply-typed fragment, towards intersection types

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:

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

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:

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

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 $A \rightarrow A$ 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: $\sigma, \tau \ ::= \ o \ \Big| \ \sigma \to \tau.$

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

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

Intersection types and strong normalization

A complete (and undecidable) characterization of SN: intersection types.

Now, $\lambda x.\, x\ x$ has type $((\tau \rightarrow \tau) \wedge \tau) \rightarrow \tau$ for all (intersection) types τ...

A term is SN iff it is typable in an appropriate intersection type system. Δ is typable, Ω isn't.

Crucial feature: intersection type systems enjoy both subject reduction and subject expansion.

In other words: typing is invariant by reduction. We'll use that to do static analysis!

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.

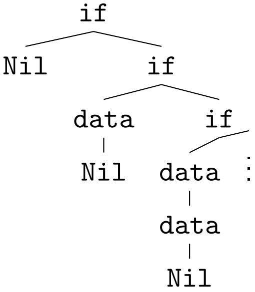
An example

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

An example

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

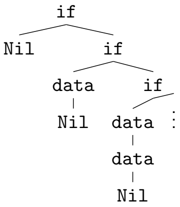
A tree model:



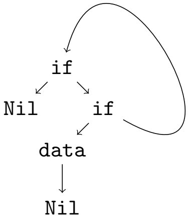
We abstracted conditionals and datatypes.

The approximation contains a non-terminating branch.

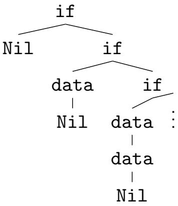
Finite representations of infinite trees



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).

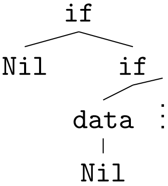
Higher-order recursion schemes

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

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

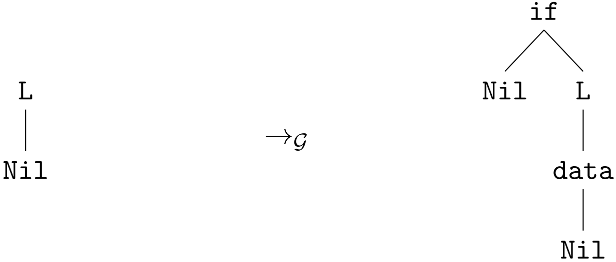
$$\mathcal{G} = \begin{cases} \mathtt{S} & = \mathtt{L\ Nil} \\ \mathtt{L}\ x & = \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ )\ ) \end{cases}$$

Rewriting starts from the start symbol S:

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

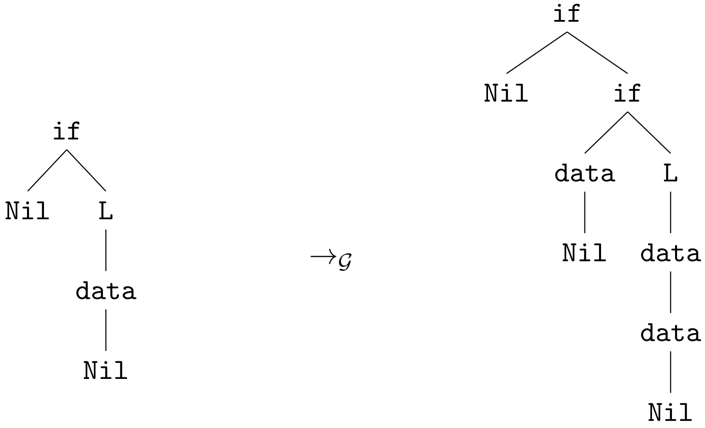
Higher-order recursion schemes

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



Higher-order recursion schemes

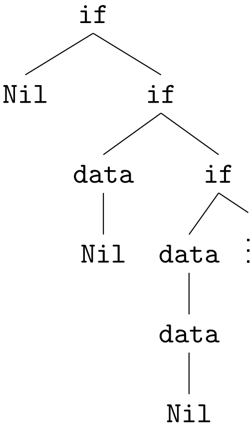
$$\mathcal{G} \quad = \quad \begin{cases} \mathtt{S} & = \quad \mathtt{L}\ \mathtt{Nil} \\ \mathtt{L}\ x & = \quad \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ )\ ) \end{cases}$$



Higher-order recursion schemes

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

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



Higher-order recursion schemes

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

can be rewritten in λ-calculus style as

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

HORS can alternatively be seen as simply-typed λ-terms with

simply-typed recursion operators $Y_\sigma \ :\ (\sigma \rightarrow \sigma) \rightarrow \sigma$.

Note that, in general, arguments may be functions of functions of functions. . .

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.

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

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

Alternating tree automata

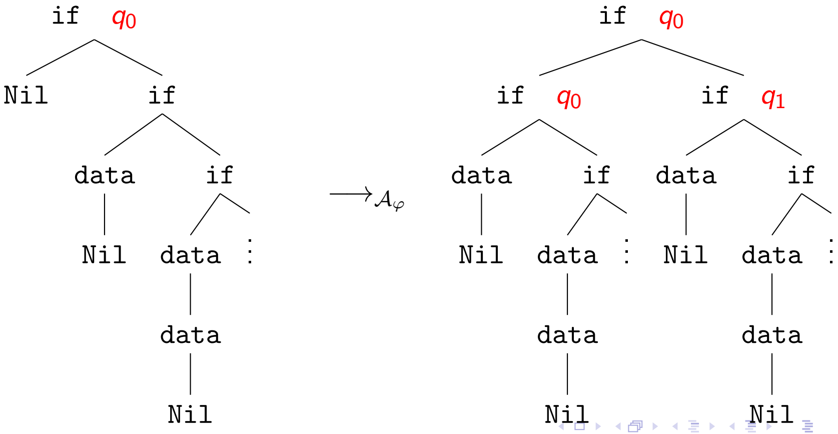
ATA: non-deterministic tree automata whose transitions may duplicate or drop a subtree.

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

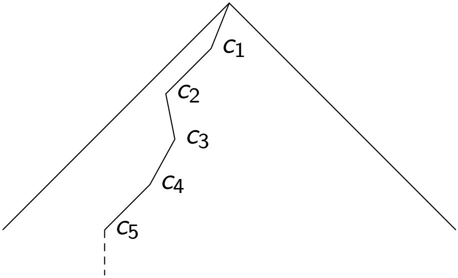


Alternating parity tree automata

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 problem

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 "

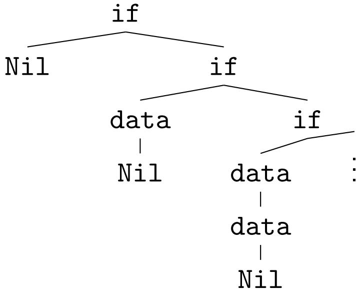
Output: `true`.

The (local) HOMC problem

Input: HORS $\mathcal{G}$, formula φ.

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

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



Output: `true`.

Our line of work

This problem is decidable (Ong 2006), and its complexity is n-EXPTIME where n is the order of the HORS of interest.

But there are practical algorithms that work quite well!

Our contributions (with Melliès, Clairambault and Murawski):

- A connection with linear logic and its models, based on a refinment of an intersection type system and on a connection between intersection types and linear logic
- Explain why it works: in fact, complexity depends on the linear order of the HORS
- For this, we introduce a linear-nonlinear version of HORS and of APT. This framework allows us to give simpler proofs of existing results of HOMC, and allows to unify these existing approaches.

Intersection types and alternation

A first connection with linear logic

Alternating tree automata and intersection types

A key remark (Kobayashi 2009):

$$\delta(q_0, \texttt{if}) \;=\; (2, q_0) \wedge (2, q_1)$$

can be seen as the intersection typing

$$\texttt{if} \;:\; \emptyset \rightarrow (q_0 \wedge q_1) \rightarrow q_0$$

refining the simple typing

$$\texttt{if} \;:\; o \rightarrow o \rightarrow o$$

Alternating tree automata and intersection types

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

$$\dfrac{\text{App}\;\dfrac{\dfrac{}{\emptyset \vdash \texttt{if} : \emptyset \to (q_0 \wedge q_1) \to q_0}\,\delta \qquad \emptyset}{\emptyset \vdash \texttt{if}\ T_1 : (q_0 \wedge q_1) \to q_0} \qquad \dfrac{\vdots}{\emptyset \vdash T_2 : q_0} \qquad \dfrac{\vdots}{\emptyset \vdash T_2 : q_1}}{\emptyset \vdash \texttt{if}\ T_1\ T_2 : q_0}\,\text{App}$$

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

Theorem (Kobayashi 2009)

$\vdash \mathcal{G} : q_0$ *iff* *the ATA* $\mathcal{A}_\varphi$ *has a run-tree over* $\langle \mathcal{G} \rangle$.

A form of static analysis!

A type-system for verification: without parity conditions

$$\text{Axiom} \qquad \frac{}{x : \bigwedge_{\{i\}} \theta_i :: \kappa \vdash x : \theta_i :: \kappa}$$

$$\delta \qquad \frac{\{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \quad \text{satisfies} \quad \delta_A(q, a)}{\emptyset \vdash a : \bigwedge_{j=1}^{k_1} q_{1j} \to \ldots \to \bigwedge_{j=1}^{k_n} q_{nj} \to q :: o \to \cdots \to o}$$

$$\text{App} \qquad \frac{\Delta \vdash t : (\theta_1 \wedge \cdots \wedge \theta_k) \to \theta :: \kappa \to \kappa' \qquad \Delta_i \vdash u : \theta_i :: \kappa}{\Delta, \Delta_1, \ldots, \Delta_k \vdash t\, u : \theta :: \kappa'}$$

$$\lambda \qquad \frac{\Delta, x : \bigwedge_{i \in I} \theta_i :: \kappa \vdash t : \theta :: \kappa'}{\Delta \vdash \lambda x . t : \left(\bigwedge_{i \in I} \theta_i\right) \to \theta :: \kappa \to \kappa'}$$

$$\mathit{fix} \qquad \frac{\Gamma \vdash \mathcal{R}(F) : \theta :: \kappa}{F : \theta :: \kappa \vdash F : \theta :: \kappa}$$

A closer look at the Application rule

In the intersection type system:

$$\text{App} \qquad \frac{\Delta \vdash t : (\,\theta_1 \ \wedge \cdots \wedge \ \theta_n) \to \theta \qquad \Delta_i \vdash u \ : \ \theta_i}{\Delta\,,\,\Delta_1\,,\,\ldots\,,\,\Delta_n \ \vdash \ t\,u \ : \ \theta}$$

This rule could be decomposed as:

$$\frac{\Delta \ \vdash \ t \ : \ (\,\bigwedge_{i=1}^{n} \ \theta_i\,) \to \theta' \qquad \dfrac{\Delta_i \ \vdash \ u \ : \ \theta_i \qquad \forall i \in \{1,\ldots,n\}}{\Delta_1,\ldots,\Delta_n \ \vdash \ u \ : \ \bigwedge_{i=1}^{n} \ \theta_i} \ \text{Right} \bigwedge}{\Delta,\,\Delta_1,\ldots,\,\Delta_n \ \vdash \ t \ u \ : \ \theta'}$$

A closer look at the Application rule

In the intersection type system:

$$\text{App} \qquad \frac{\Delta \vdash t : (\ \theta_1 \ \wedge \cdots \wedge \ \theta_n) \rightarrow \theta \qquad \Delta_i \vdash u \ : \ \theta_i}{\Delta\,,\, \Delta_1\,,\, \ldots\,,\, \Delta_n \ \vdash \ t\, u \ : \ \theta}$$

This rule could be decomposed as:

$$\frac{\Delta \ \vdash \ t \ : \ (\ \bigwedge_{i=1}^{n} \ \theta_i\ ) \rightarrow \theta' \qquad \dfrac{\Delta_i \ \vdash \ u \ : \ \theta_i \qquad \forall i \in \{1, \ldots, n\}}{\Delta_1, \ldots, \Delta_n \ \vdash \ u \ : \ \bigwedge_{i=1}^{n} \ \theta_i} \ \text{Right} \bigwedge}{\Delta, \Delta_1, \ldots, \Delta_n \ \vdash \ t \ u \ : \ \theta'}$$

A closer look at the Application rule

$$\frac{\Delta \vdash t : (\bigwedge_{i=1}^{n} \theta_i) \rightarrow \theta' \qquad \dfrac{\Delta_i \vdash u : \theta_i \qquad \forall i \in \{1, \ldots, n\}}{\Delta_1, \ldots, \Delta_n \vdash u : \bigwedge_{i=1}^{n} \theta_i} \text{ Right } \bigwedge}{\Delta, \Delta_1, \ldots, \Delta_n \vdash t\ u : \theta'}$$

Linear decomposition of the intuitionistic arrow:

$$A \Rightarrow B \quad = \quad !\,A \multimap B$$

Two steps: duplication / erasure, then linear use.

Right $\bigwedge$ corresponds to the Promotion rule of indexed linear logic.
(see G.-Melliès, ITRS 2014)

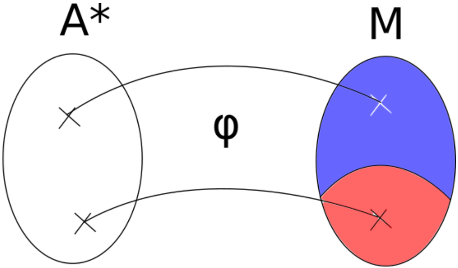
Overview of our results

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

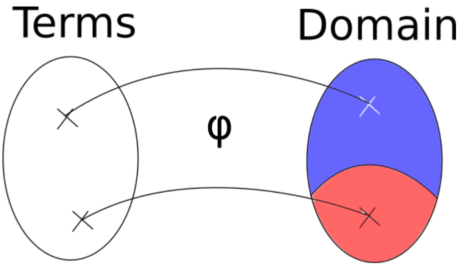
if and only if...



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.

Finitary semantics of linear logic

In ScottL (a finitary model of linear logic), we define $\Box$, λ and $\mathbf{Y}$ in an appropriate way.
$ScottL_{\lightning}$ is a model of the λY-calculus.

Theorem

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

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

Corollary

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

See Grellois-Melliès: CSL 2015, Fossacs 2015, MFCS 2015, and my thesis.

Linear order and the true complexity of HOMC

Clairambault, G., Murawski, POPL 2018: order isn't the good measure for complexity. We can use linear order.

Idea: when the automaton doesn't use alternation, complexity doesn't increase that much. . .

A big advantage: allows to reprove several works on HOMC in a much simpler way!

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.

- $\mathsf{Nat}^{\widehat{\mathfrak{i}}}$ 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: $\quad \mathfrak{s}, \mathfrak{r} \quad ::= \quad \mathfrak{i} \mid \infty \mid \widehat{\mathfrak{s}}$

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

Fixpoint rule:

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

"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.):

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

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

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.

Probabilistic termination of probabilistic HORS

Termination analysis for functional programs, one more step:

Kobayashi - Dal Lago - Grellois, LICS 2019: a probabilistic extension of HORS and approximate techniques for termination verification.

Conclusion

- Type theory is perfectly fit to verify functional programs, by considering appropriate type systems.
- Type theory can be extended to accomodate the probabilistic setting.
- Linear logic is very connected to intersection types, and allows to consider more carefully the behavior of programs
- Contributions: models of LL for HOMC, refinment of the complexity measures, a new framework to carry proofs in HOMC, termination analysis for probabilistic programs.

Thank you for your attention!

Conclusion

- Type theory is perfectly fit to verify functional programs, by considering appropriate type systems.
- Type theory can be extended to accomodate the probabilistic setting.
- Linear logic is very connected to intersection types, and allows to consider more carefully the behavior of programs
- Contributions: models of LL for HOMC, refinment of the complexity measures, a new framework to carry proofs in HOMC, termination analysis for probabilistic programs.

Thank you for your attention!