Proofs-Programs correspondance and Security

Jean-Baptiste Joinet

Université de Lyon
&
Centre Cavaillès, École Normale Supérieure, Paris

Third Cybersecurity Japanese-French meeting
Formal methods session
Keiô University
24/04/2017

Simple aims of this 15 minutes talk :

to advocate the relevance of
the Proofs as Programs paradigm
to understand programs behavior
with in view applications
to security questions

“Proofs-as-Programs” paradigm

- 1969. “Curry-Howard isomorphism” :

The process of analytization of Proofs
in Intuitionistic Natural Deduction
=
The process of computation
in Simply typed Lambda Calculus

The conclusion of a proof
=
The type of the
corresponding program

“Proofs-as-Programs” paradigm

- 1969. “Curry-Howard isomorphism” :

  The process of analytization of Proofs
  in Intuitionistic Natural Deduction
  =
  The process of computation
  in Simply typed Lambda Calculus

  The conclusion of a proof
  =
  The type of the
  corresponding program

- 2017. Generalized to almost all parts of Logic (including Set theory)
  - Second-order quantification (Polymorphic types, Girard’s System F. . .)
  - Generalization to Classical Logic (e.g. Lambda-Mu-calculus. . .)
  - Subsystems of Classical Logic with a lightened complexity (designed through Linear Logic’s decomposition of computation), etc. . .

Propositions-as-Types : a first approach

- First approach of types (the "external" one) :
  - types are given by an additional "second level" grammar
  - used to externally submit the construction of programs to constraints

Propositions-as-Types : a first approach

- First approach of types (the "external" one) :
  - types are given by an additional "second level" grammar
  - used to externally submit the construction of programs to constraints

- Typing then is a way to avoid :
  - some programs
  - thus some particular computational dynamics
  - thus some undesired properties of computation :
    - typically non termination
    - termination within a too long runtime

Propositions-as-Types : methodological use of the first approach

The program extraction methodology (to guarantee to get a correct program wrt an equational specification)

- Data types : second order types whose shape determines all the terms of that type
- Define equationally a *recursive function* on data in first order logic
- Prove the formula that states that the function is terminating
- We then know that the program corresponding to the proof does satisfy the specification

Propositions-as-Types : methodological use of the first approach

The program extraction methodology (to guarantee to get a correct program wrt an equational specification)

- Data types : second order types whose shape determines all the terms of that type
- Define equationally a *recursive function* on data in first order logic
- Prove the formula that states that the function is terminating
- We then know that the program corresponding to the proof does satisfy the specification

What about other (non arithmetical functional) theorems ?

Propositions-as-Types : a second approach

Slogan :

a type

$=$

a set of programs

with some common behavior

with respect to

some set of tests

Propositions-as-Types : a second approach

Slogan :

a type
=
a set of programs
with some common behavior
with respect to
some set of tests

How to characterize abstractly which set of programs are types :

- idea : a type is a set of programs "orthogonal" to some set of programs (i.e. which is closed by bi-orthogonality)
- types constructors are operations on sets of programs that preserve the fact to be a type

Krivine's specification methodology

Goal : prove that all programs of a given type have a given common behavior

Krivine's classical realizability could be used as a device to infer behaviors from the type :

- a classical typing discipline is needed : second order (classical) predicate calculus, formalized by adding Peirce law to the intuitionistic natural deduction

Krivine's specification methodology

Goal : prove that all programs of a given type have a given common behavior

Krivine's classical realizability could be used as a device to infer behaviors from the type :

- a classical typing discipline is needed : second order (classical) predicate calculus, formalized by adding Peirce law to the intuitionistic natural deduction
- then, in order to realize classical proofs :
  - a "classical" extension of Lambda-calculus (means : with control, exceptions treatment)
  - Behaviors described in terms of three categories (coming from $\neg\neg$-translation of intuitionistic logic into itself !) :
    1. terms,
    2. stacks (of terms),
    3. executables (pairs made of a term and a stack)
  - and w.r.t. a particular evaluation strategy (Call-by-name weak head evaluation with a stack-save-and-restore abstract machine) which preserves this description in three categories.

Krivine's specification methodology

Solving specification problems :

- Interpretation of atomic formulas by set of terms : $|\, X \,|$
- $|\perp|$ is a chosen set of executables closed by retro-reduction
- define $|\, A \,|^{-}$ (orthogonal) as the set of stacks that will form nice executables when paired with terms in $|\, A \,|$ with respect to $|\perp|$
- define inductively the interpretation "as usual", but through a $\neg\neg$ translation
- adequacy theorem : for any $|\perp| \subseteq \{\textit{terms}\} \times \{\textit{stacks}\}$: it the term t is of type A and π is a stack in the orthogonal of A, then the executable (t, π) is in the interpretation of $\perp$.
- adequacy theorem is then used to prove that a particular common behavior is shared by all terms :
  - introduce a new combinator
  - describe its postulated computational behavior in terms of stack-save-and-restore manipulations (within the frame of the chosen cbn evaluation)
  - show that it is a realizer of the corresponding type, i.e. choose a relevant $|\perp|$ and show the combinator belongs to the interpretation of the corresponding type.

Fin