Towards proved programs
Introduction

MPRI 2-4-2 (Version Fri Jan 16 11:18:50 CET 2015)

Yann **Régis-Gianas**
yrg@pps.univ-paris-diderot.fr

Paris 7 - PPS
INRIA - $\pi.r^2$

What is a type system?

"A type system is a ***tractable*** *syntactic method of* ***proving the absence*** *of certain program behaviors by classifying phrases according to the kinds of values they compute."*

Benjamin Pierce – *Types and Programming Languages*

What is a type good for?

1. A type is an **invariant** of the computation that classifies values.

$\Rightarrow$ This is ensured by the "Subject Reduction" property.

2. From the type of a value, we can deduce its shape.

$\Rightarrow$ This is the "Classification" lemma.

3. A well-typed expression is either a value or it can be reduced.

$\Rightarrow$ The "Progress" property ensures that what we get when the computation stops is meaningful. The computation cannot get stuck.

$\Rightarrow$ Points 1 and 2 have been tailored to obtain 3.

From that perspective,
well-typedness is (only) a **safety** property.

What is a safe computation exactly?

```
let head = function
  | [] → failwith "Error, there is no head inside an empty list."
  | x :: _ → x
```

This program is safe because the error is cleanly handled by the semantics. Indeed, in presence of exceptions, we write:

Theorem (Progress)

A well-typed, irreducible term is either a value or an uncaught exception.
If $\emptyset \vdash t : T$ and t, then t is either v or raise v for some value v.

Yet, applying `head` on an empty list is certainly a programming error!

What is a correct program?

```
(* This function sorts an arbitrary list.  *)
let rec sort = function
  | [] → []
  | [x] → [x]
  | x :: (y :: _ as ys) when x < y → x :: sort ys
  | x :: (y :: _ as ys) when x ≥ y → y :: sort (x :: ys)
  | _ → assert false
```

Of course, this well-typed program is **not** a valid sorting program. It does terminate normally with a resulting value even though it does not respect its specification.

Should a type system serve as a **proof system** and reject this program?

Towards proved programs

In this second part of the course, we will try to move from the Milner [Milner, 1978]'s slogan :

> "Well-typed programs do not go wrong."

to the formal proof that programs compute correctly :

> "Well-typed programs respect their specification."

How to extend the expressivity of static checking?

1. Type-centric approach: *Types as specification*

   Following the Curry-Howard [Howard, 1980] correspondence, a type can be read as a formula. By augmenting the expressiveness of types, a type-checker can be turned into a proof-checker. This mutation can even be realized in a programming language with side-effects and non terminating terms.

2. Extended static checking: *Logic assertions as specification*

   Keeping the role of types to the only denotation of invariants that (automatically) ensure safety, we will introduce a third class of syntactic objects, namely logic assertions, that will be used to guarantee complex properties using a proof system based on Hoare logic.

Road map

1. Generalized Algebraic Data Types [Xi et al., 2003]
2. Dependent types
3. Higher-Order Hoare Logic

References I

William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.