MPRI 2.4, Functional programming and type systems
Metatheory of System F

Didier Rémy

September 15, 2017

Plan of the course

Metatheory of System F

ADTs, Existential types, GATDs

Logical relations

Subtyping, Row polymorphism

References, Value restriction, Side effects

Introduction

Choosing the meta language of this course. . .

Choosing the meta language of this course...

English or French?

Choosing the meta language of this course...

English or French?

In any case, questions must be asked in the language your speak best
(French by default)

Online material

Visit the course page

https://gitlab.inria.fr/fpottier/mpri-2.4-public/blob/master/README.md

Accessible from the official MPRI 2-4 page

https://wikimpri.dptinfo.ens-cachan.fr/doku.php?id=cours:c-2-4-2

All course material:

- Course notes (will be updated as we progress)
- Calendar of lessons and exams
- Information on the programming task
- All useful information and pointers

Outline of the course (has changed!)

4 parts, each composed of 5 lessons.

1. Programming: Under the Hood (mostly untyped)
2. Metatheory of Typed Programming Languages
3. Advanced Aspects of Type Systems
4. Dependently-typed Functional Programming

Outline of the course (has changed!)

4 parts, each composed of 5 lessons.

1. Programming: Under the Hood (mostly untyped)
   - From operational semantics...
   - ...to interpreters
   - Compiling away first-class fonctions
   - Compiling away the call stack: the CPS transformation.
   - Equational reasoning and program optimizations.
2. Metatheory of Typed Programming Languages
3. Advanced Aspects of Type Systems
4. Dependently-typed Functional Programming

Outline of the course (has changed!)

4 parts, each composed of 5 lessons.

1. Programming: Under the Hood (mostly untyped)
2. Metatheory of Typed Programming Languages
   - System F,
   - ADTs, Existential types and GADTs
   - Logical relations
   - Subtyping and row polymorphism
   - References, value restriction, Side effects
3. Advanced Aspects of Type Systems
4. Dependently-typed Functional Programming

Outline of the course (has changed!)

4 parts, each composed of 5 lessons.

1. Programming: Under the Hood (mostly untyped)
2. Metatheory of Typed Programming Languages
3. Advanced Aspects of Type Systems
   - Exceptions and effect handlers. (Compiled away via CPS.)
   - Typechecking exceptions and handlers.
   - Type inference. (ML. Bidirectional. Elaboration.)
   - Data/control flow analysis.
   - Functional correctness. Intro to dependent/refinement types.
4. Dependently-typed Functional Programming

Outline of the course (has changed!)

4 parts, each composed of 5 lessons.

1. Programming: Under the Hood (mostly untyped)
2. Metatheory of Typed Programming Languages
3. Advanced Aspects of Type Systems
4. Dependently-typed Functional Programming
   - (Effectful, Dependent, Total, Generic) functional programming
   - Open problems in dependent functional programming.

Outline of the course (has changed!)

4 parts, each composed of 5 lessons.

1. Programming: Under the Hood (mostly untyped)
2. Metatheory of Typed Programming Languages
3. Advanced Aspects of Type Systems
4. Dependently-typed Functional Programming

Parts of the course will be mechanized in Coq:

- some knowledge of Coq is not mandatory, but may help:-)
- We won't ask you to do Coq proofs, but you'll be free to do so
- At the end of the course, you should be able to read coq statements

Outline of the course (has changed!)

4 parts, each composed of 5 lessons.

1. Programming: Under the Hood (mostly untyped)
2. Metatheory of Typed Programming Languages
3. Advanced Aspects of Type Systems
4. Dependently-typed Functional Programming

Parts of the course will be mechanized in Coq:

- some knowledge of Coq is not mandatory, but may help:-)
- We won't ask you to do Coq proofs, but you'll be free to do so
- At the end of the course, you should be able to read coq statements

The course is splitable after the two first parts.

Questions!

Questions are welcome!

Please, ask questions...

- during the lesson
- at the end of the lesson
- by email

Didier.Remy@inria.fr

Please, don't wait until the end of the course to tell me about any problems you may encounter.

You are there to learn

and

I am here to help you!

If you have any difficulties during this course:

- do the exercises, check the corrections, ask me if you can't do them.
- discuss with me: the earlier the better.
- don't wait until the exams...

Programming task

A programming task will be given by mid-december

- The solution is due by the end of the course.
- It counts for *about* 1/3 in the final grade (for a full course).
  (We may change this a little depending on the difficulty and amount of work needed for the programming task)
- It is fun! (according to your fellow students from previous years)
- It focuses on one particular topic of the course and usually helps understand it in detail.

Questions?

Contents

What is functional programming?

The term *"functional programming"* means various things:

- it views functions as ordinary data—which, in particular, can be passed as arguments to other functions and stored in data structures.
- it loosely or strongly discourages the use of modifiable data, in favor of effect-free transformations of data.

  (In contrast with mainstream object-oriented programming languages)
- encourages abstraction of repetitive patterns as functions that can be called multiple times so as to avoid code duplication.

What are functional programming languages?

They are usually:

– *typed* (Scheme and Erlang are exceptions),
with close connections to logic.

In this course, we focus on typed languages and types play a central role.

– given a precise *formal semantics* derived from that of the λ-calculus.

Some are *strict* (ML) and some are *lazy* (Haskell) [Hughes, 1989].

This difference has a large impact on the language design and on the programming style, but has usually little impact on typing.

– *sequential*: their model of evaluation is not concurrent, even if core languages may then be extended with primitives to support concurrency.

Contents

What are types?

– Types are:

"a concise, formal description of the behavior of a program fragment."

– For instance:

int	An integer
$int \to bool$	A function that maps an integer to a Boolean
$(int \to bool) \to (list\ int \to list\ int)$	A function that maps an integer predicate to an integer list transformer

– Types must be *sound*.
That is, programs must behave as prescribed by their types.

– Hence, types must be *checked* and ill-typed programs must be rejected.

What are they useful for?

- Types serve as *machine-checked* documentation.
- Data types help structure programs.
- Types provide a *safety* guarantee.

  > *"Well-typed expressions do not go wrong." [Milner, 1978]*

  (Advanced type systems can also guarantee various forms of security, resource usage, complexity, . . .)
- Types can be used to drive *compiler optimizations.*
- Types encourage *separate compilation*, *modularity*, and *abstraction*.

  > *"Type structure is a syntactic discipline for enforcing levels of abstraction." [Reynolds, 1983]*

  Type-checking is compositional. Types can be abstract.
  Even seemingly non-abstract types offer a degree of abstraction
  (e.g., a function type does not tell how a function is represented)

Type-preserving compilation

Types make sense in *low-level* programming languages as well—even *assembly languages* can be statically typed! [Morrisett et al., 1999]

In a *type-preserving* compiler, every intermediate language is typed, and every compilation phase maps typed programs to typed programs.

Preserving types provides insight into a transformation, helps *debug* it, and paves the way to a *semantics preservation* proof [Chlipala, 2007].

Interestingly enough, lower-level programming languages often require richer type systems than their high-level counterparts.

Typed or untyped?

Reynolds [1985] nicely sums up a long and rather acrimonious debate:

> *"One side claims that untyped languages preclude compile-time error checking and are succinct to the point of unintelligibility, while the other side claims that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility."*

The issues are *safety*, *expressiveness*, and *type inference*.

A sound type system with decidable type-checking (and possibly decidable type inference) must be *conservative*.

Typed, Sir! with better types.

In fact, Reynolds settles the debate:

> *"From the theorist's point of view, both sides are right, and their arguments are the motivation for seeking type systems that are more flexible and succinct than those of existing typed languages."*

Today, the question is more whether to stay with rather simple polymorphic types (*e.g.* as in ML or System F) or use more sophiscated types (*e.g.* dependent types, afine types, capabililties and ownership, effect types, logical assertions, *etc.*), or even towards full program proofs!

Explicit *v.s.* implicit types?

Annotating programs with types can lead to redundancy.
Types can even become extremely cumbersome when they have to be explicitly and repeatedly provided.

In some pathological cases, type information may grow in square of the size of the underlying untyped expression.

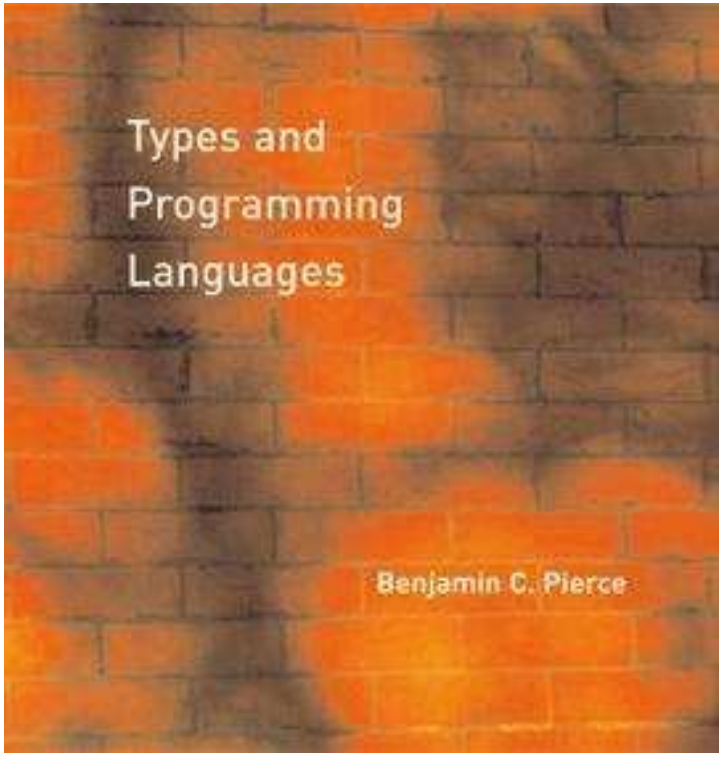
This creates a need for a certain degree of *type reconstruction* (also called type inference), where the source program may contain some but not all type information.

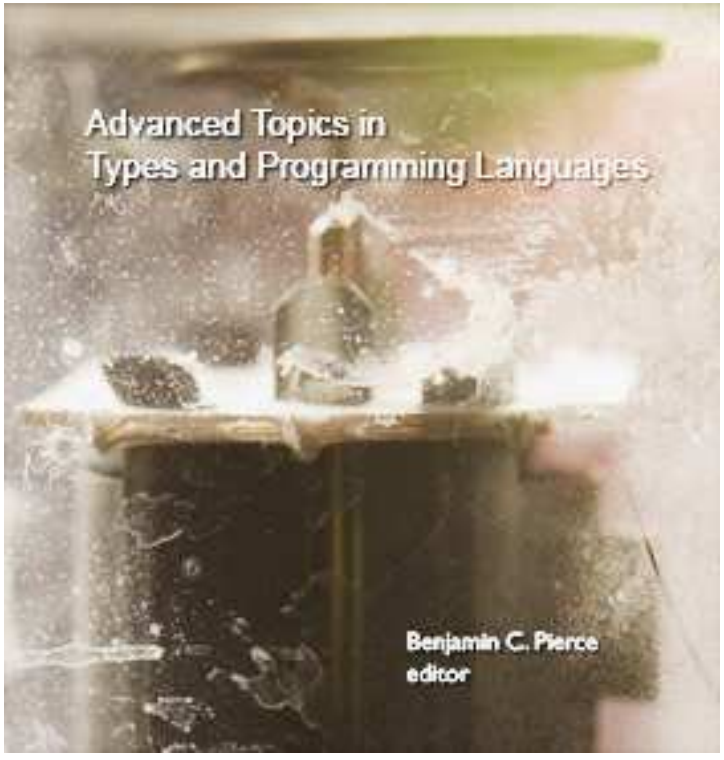
In principle, types could be entirely left implicit, even if the language is typed. A well-typed program is then one that is the type erasure of a (well-typed) explicitly-typed program.

Full type reconstruction is undecidable for expressive type systems.

Some type annotations are required or type reconstruction is incomplete.

Excellent books





Parts of this course are covered in chapters of these books

Online material

Written notes *v.s.* copies of the slides.

Detailed notes from a previous version of the course are available online.

You should rather read the course notes than the slides:

- Contain more details than what I say during the lesson.
- Proofs:
  - often omitted or sketchy on the slides (or on the board).
  - with full details in course notes, as you should write them.
- Exercises:
  - Course notes contain more exercises and solutions to exercises,
  - Only a few of them are mentioned on the slides or done in class.

Bibliography I

(Most titles have a clickable mark "▷" that links to online versions.)

▷ Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

▷ John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

▷ Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

▷ John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

Bibliography II

▷ John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.