IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Compilation

A crash course

Fabien Dagnat

ELU 610 – C7

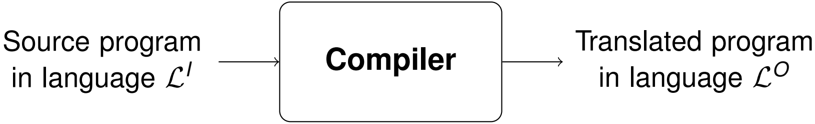
1st semester 2019

1. The structure of a compiler
2. Lexing
3. Parsing
4. Core

1. The structure of a compiler
2. Lexing
3. Parsing
4. Core

What is a compiler?

- It is a program transformer from $\mathcal{L}_1$ to $\mathcal{L}_2$

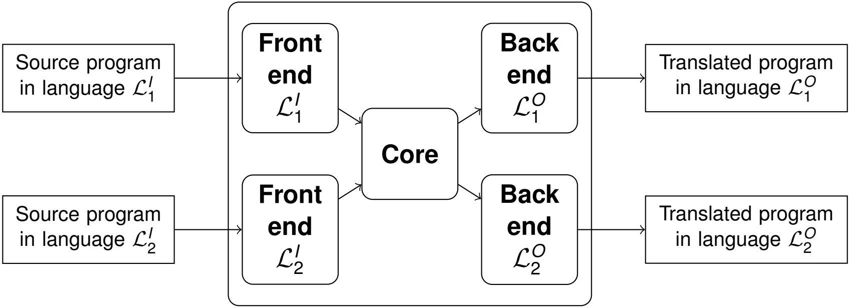


- $\mathcal{L}^O$ is "more" executable than $\mathcal{L}^I$
- Most of the time, a program in $\mathcal{L}_2$ is directly executable
  - either by a machine (*e.g.* $\mathcal{L}^O$ = X64)
  - or by an abstract machine[1] (*e.g.* $\mathcal{L}^O$ = OCaml bytecode)
- Often a compiler is composed of a flow of compilers

$$\mathcal{L}_1 \xrightarrow{\text{compiler}_1} \mathcal{L}_2 \xrightarrow{\text{compiler}_2} \cdots \xrightarrow{\text{compiler}_n} \mathcal{L}_{n+1}$$

[1] an abstract machine is a piece of software acting as a machine

- It is a composed of three stages
  - *front end* in charge of recognizing $\mathcal{L}^I$ (*e.g.* gcc has C, C++, Go, ...)
  - *core* doing the hard work
  - *back end* in charge of emitting $\mathcal{L}^O$ (*e.g.* gcc has X64, ARM, ...)

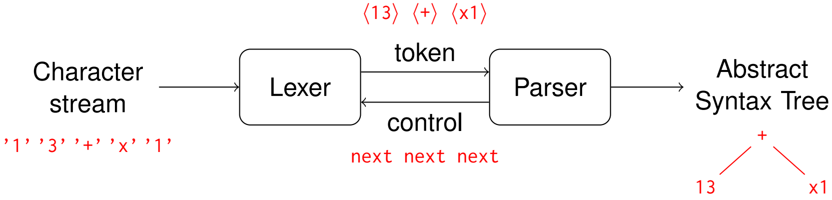


- Several paths are possible

- It checks whether the program is syntactically correct
  - It belongs to the language $\mathcal{L}^I$
- It must build an internal representation of the program
  - It is an internal data structure of the compiler

$\Rightarrow$ It is highly dependent of the input language

- It is decomposed in two parts
  - lexer recognizes tokens in a character stream
  - parser recognizes sentences in a token stream



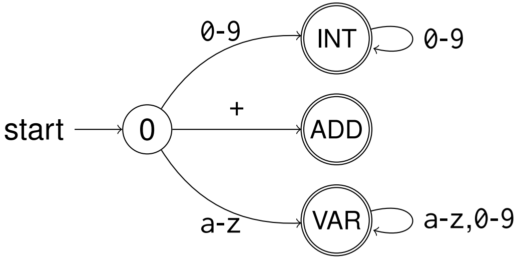
- Works on internal data structures
- Is in charge of the verification of validity (typing) of the program
- In charge of the main transformation work, for instance
  - simplify programs by removing useless elements
  - transform function calls
  - transform object oriented access
- Relatively usual software
- Functional paradigm is very adapted for this kind of code
  - recursive functions for the visiting part
  - sum types for representing the various elements
  - product types to add information to the various elements
  - pattern matching for the recognition of structure
  - ...

- Translates internal data structures in instructions of the target language

```
...
movq $1, %rax
; some code to get the value of x1
addq $26, %rax
...
```

- Implements all optimizations specific to the target
- Complex requiring to master the target machine
- Not in the scope of this introduction...

- A lexer is in charge of reading enough characters from an entry stream to produce a token
- To be efficient it is generally built as an automaton where
  - transitions correspond to the received characters
  - final state corresponds to the production of the token



- When reaching a final state, it produces a token (data structure for the parser)

- One way to define such an automaton is to define a set of translation rules
- A translation rule is composed of a pattern and an action
  - the pattern is defined using a regular expression specifying the accepted input
  - the action defines what to do in case of accepting (often just returning the right token)
- For the previous slide example

```
| ['0'-'9']+ { INT((* input converted in int *)) }
| '+' { ADD }
| ['a'-'z']['a'-'z''0'-'9']* { VAR((* input *)) }
```

- A Domain Specific Language: OCamllex
  - a compiler `ocamllex` producing OCaml code for the automaton

- File with extension `.mll`

```
{ (* OCaml code: optional prelude *) }
(* useful regular expressions only for regexp part *)
let ident = regexp
let ident = regexp
(* a group of rules *)
rule ident [ident1 ... identn] = parse
| regexp { (* OCaml code *) }
| regexp { (* OCaml code *) }
(* another  group of rules *)
and ident [ident1 ... identn] = parse
...
{ (* OCaml code: optional postlude *) }
```

- prelude: as is in the result
- regexp: compiled as an automaton
- OCaml code in rules: actions executed on accepting
- rule: translated as a function
- postlude: as is in the result

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*`

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*` ⇒ C like line comment
- Suppose

```
let digit = ['0'-'9']
let letter = ['a'-'z''A'-'Z']
let id_char = (letter | digit | '_')
```

- `letter id_char* as id`

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*` ⇒ C like line comment
- Suppose

```
let digit = ['0'-'9']
let letter = ['a'-'z''A'-'Z']
let id_char = (letter | digit | '_')
```

- `letter id_char* as id` ⇒ identifiers, `id` contains the result
- integers

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*` ⇒ C like line comment
- Suppose

```
let digit = ['0'-'9']
let letter = ['a'-'z''A'-'Z']
let id_char = (letter | digit | '_')
```

- `letter id_char* as id` ⇒ identifiers, `id` contains the result
- integers ⇒ `digit+ as nb`
- floating point numbers

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*` ⇒ C like line comment
- Suppose

```
let digit = ['0'-'9']
let letter = ['a'-'z''A'-'Z']
let id_char = (letter | digit | '_')
```

- `letter id_char* as id` ⇒ identifiers, `id` contains the result
- integers ⇒ `digit+ as nb`
- floating point numbers ⇒ `digit* '.' digit* (['e' 'E'] ['+' '-']? digit+)? as nb`

- Each `rule name a1 ... an` gives a function `name` taking args
  - `a1`, ... `an`, the user arguments
  - a buffer containing the stream of character of type `Lexing.lexbuf`
- This function matches the characters in the buffer to execute the corresponding accepting action when called
  - it selects the regexp giving the longest part matched
  - in case of equality it selects the first defined
- The standard library module `Lexing` also provides
  - two constructors for buffers: `from_channel` and `from_string`
  - `lexeme buf` returning the matched string of `buf`
- Only one automaton is generated even for several entry points
  - the automaton is determinized and minimized
  - its code is finally put between the prelude and postlude

- File `formulaLexer.mll`

```
{
 type token = EOF | AND | OR | TRUE | FALSE
}
let space = [' ' '\t' '\n']
rule token = parse
  | space+ { token lexbuf }
  | eof    { EOF }
  | "and"   { AND }
  | "or"   { OR }
  | "true"  { TRUE }
  | "false"  { FALSE }
```

- `ocamllex formulaLexer.mll` produces `formulaLexer.ml`
- It then can be compiled using `ocamlc`
  - ⚠ it can contain errors if the `mll` file contained wrong OCaml code

Which kind of grammar?

- For most reasonable language syntax, regular expressions are not sufficient
- We must use more powerful grammars but keep efficiency of parsing

⇒ We use context-free grammars (CFG)

  - defined only by production $A \rightarrow m$ where $A \in V$ and $m \in (X \cup V)^*$

- In this course, we will focus on LR(1) parsing by using Menhir[2]
- Menhir
  - offers a DSL for defining grammars in `.mly` files
  - has a tool compiling a grammar spec. to OCaml code (`menhir`)
- Menhir follows a flow similar to OCamllex

[2] `http://gallium.inria.fr/~fpottier/menhir`

- File `formulaParser.mly`

```
%token AND OR EOF TRUE FALSE
%token <string> IDENT
%start< string > formula
%%
formula: c=disj EOF                        { c }
disj:
 | c=conj OR d=disj                         { "("^c^" or "^d^")" }
 | c=conj                                   { c }
conj:
 | s=ident_or_const AND c=conj             { "("^s^" and "^c^")" }
 | s=ident_or_const                         { s }
ident_or_const:
 | id=IDENT                                 { id }
 | TRUE                                     { "true" }
 | FALSE                                    { "false" }
%%
```

The token type (annotating the two `%token` lines)

An entry point with its return type (annotating `%start< string > formula`)

Grammar (left column) — Actions (right column)

- The token type is now generated within the parser

⇒ The lexer does not define it anymore but imports the parser

- Each entry point (`%start`) gives a parsing function of type `(Lexing.lexbuf -> token) -> Lexing.lexbuf -> string`

⇒ The lexer must be given to the parser

```
let compile file =
 try
  let input_file = open_in file in
  let result = formula token (Lexing.from_channel input_file) in
  close_in (input_file);
  printf "read %s\n" result
 with Sys_error s ->
  printf "Can't find file '%s'" file
let () = Arg.parse [] compile ""
```

The result of parsing

- In general, the result of parsing is a data structure representing programs called an Abstract Syntax Tree (AST)
- Defined using recursive sum types

```
type t =
  | Var of string
  | Bool of bool
  | And of t * t
  | Or of t * t
```

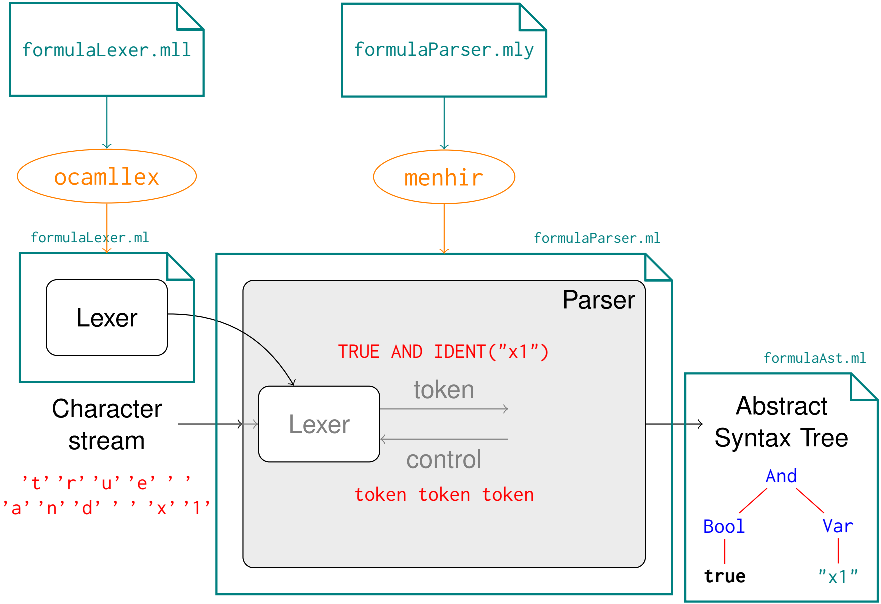
- Manipulated by recursive functions

```
let rec string_of = function
  | Var s -> s
  | Bool true -> "true"
  | Bool false -> "false"
  | And(f1,f2) -> "("^(string_of f1)^" and "^(string_of f2)^")"
  | Or(f1,f2) -> "("^(string_of f1)^" or "^(string_of f2)^")"
```

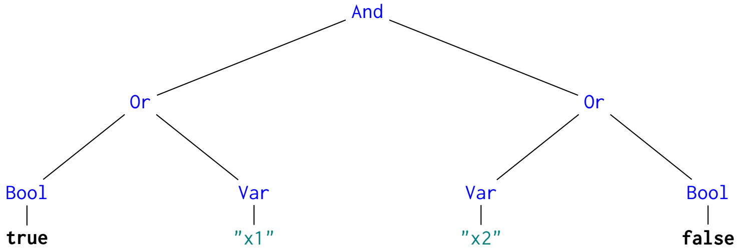
```
%{
  open FormulaAst
%}
%token AND OR EOF TRUE FALSE
%token <string> IDENT
%start< FormulaAst.t > formula
%%
formula: c=conj EOF                   { c }
conj:
 | d=disj AND c=conj                  { And(d,c) }
 | d=disj                              { d }
disj:
 | s=ident_or_const OR d=disj         { Or(s,d) }
 | s=ident_or_const                   { s }
ident_or_const:
 | id=IDENT                            { Var id }
 | TRUE                                 { Bool true }
 | FALSE                                { Bool false }
%%
```

Prelude

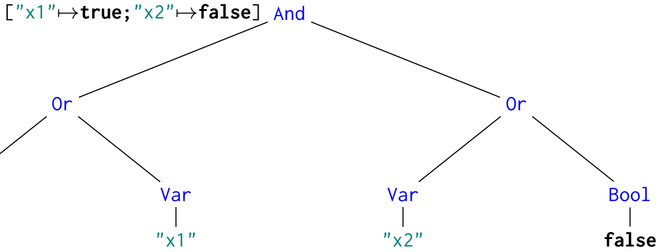
The new complete flow



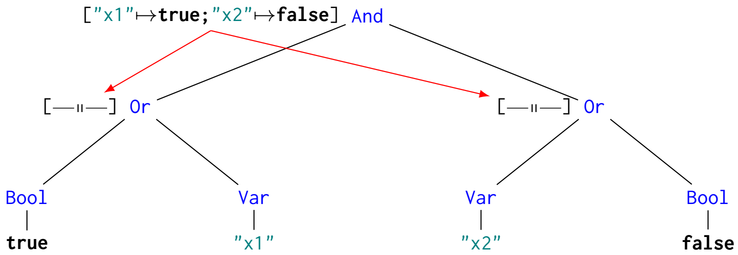
- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected



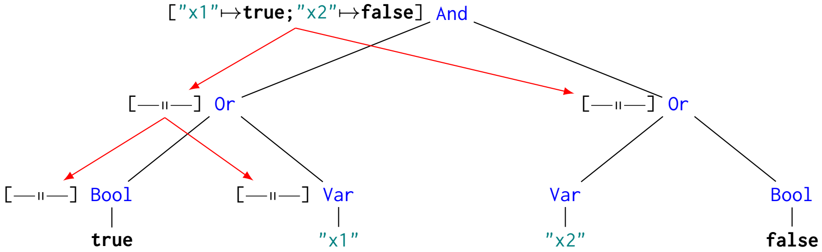
- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected



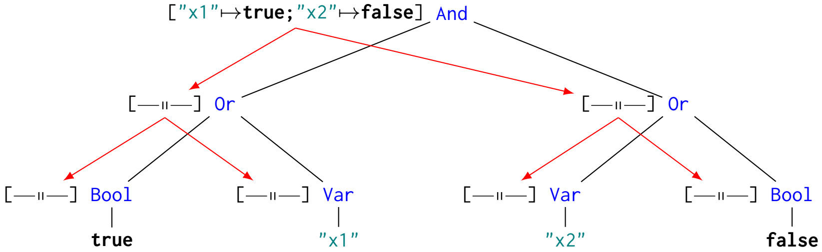
- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected



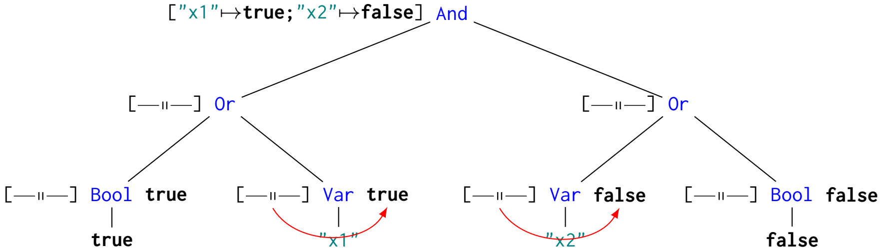
- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected



- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected

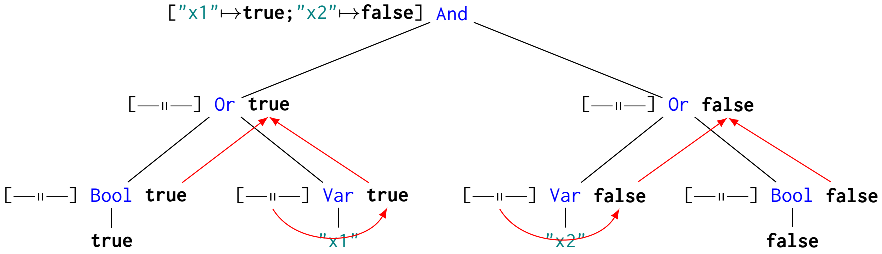


- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected

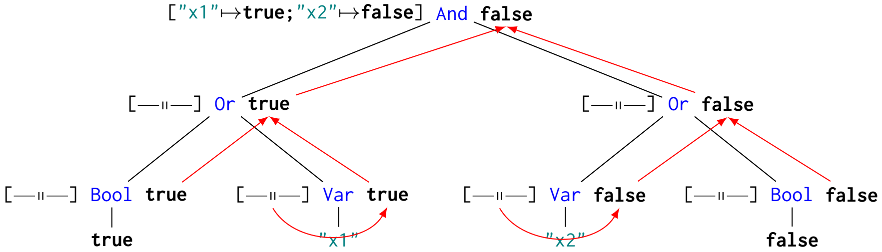


Evaluation

- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected



- One way to execute a program is to use an interpreter
  - often called a Read Eval Print Loop
- Consists in producing a value from an AST
- It uses a recursive visit of the AST to synthesize the value
- While descending into the AST, naming information must be collected



- Such visit can be formalized using big step Structural Operational Semantics
  - values are from the boolean algebra $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$ with $\wedge$ and $\vee$
    - $\mathbf{T} \wedge \mathbf{T} = \mathbf{T}, \mathbf{F} \wedge b = b \wedge \mathbf{F} = \mathbf{F}, \mathbf{T} \vee b = b \vee \mathbf{T} = \mathbf{T}, \mathbf{F} \vee \mathbf{F} = \mathbf{F}$
  - an environment function $\mathcal{E}$ mapping variable names to values
    - *dom* gives its domain, $\mathcal{E}(x)$ gives the value associated to x in $\mathcal{E}$
  - judgements of the form $\mathcal{E} \vdash$ AST term $\Rightarrow$ value

$$(1)\ \mathcal{E} \vdash \texttt{Bool}\ \texttt{\textbf{true}} \Rightarrow \mathbf{T} \qquad (2)\ \mathcal{E} \vdash \texttt{Bool}\ \texttt{\textbf{false}} \Rightarrow \mathbf{F}$$

$$(3)\ \frac{x \in dom(\mathcal{E})}{\mathcal{E} \vdash \texttt{Var}\ x \Rightarrow \mathcal{E}(x)} \qquad (4)\ \frac{\mathcal{E} \vdash F_1 \Rightarrow b_1 \qquad \mathcal{E} \vdash F_2 \Rightarrow b_2}{\mathcal{E} \vdash \texttt{And}(F_1, F_2) \Rightarrow b_1 \wedge b_2}$$

$$(5)\ \frac{\mathcal{E} \vdash F_1 \Rightarrow b_1 \qquad \mathcal{E} \vdash F_2 \Rightarrow b_2}{\mathcal{E} \vdash \texttt{Or}(F_1, F_2) \Rightarrow b_1 \vee b_2}$$

```
open FormulaAst
let eval env formula =
 let rec eval_rec = function
 | Var s      -> List.assoc s env
 | Bool true  -> true
 | Bool false -> false
 | And(f1,f2) -> (eval_rec f1) && (eval_rec f2)
 | Or(f1,f2)  -> (eval_rec f1) || (eval_rec f2)
 in
 eval_rec formula
```

- Here, as the environment is constant during the visit, it is made global to the visting function (`eval_rec`)
- In implementation, we should take care of errors ($x \notin dom(\mathcal{E})$)

- Evaluation can lead to runtime errors
- Typing "approximates" evaluation to detect a maximum of runtime errors in advance
  - the "value" set is simplified and called type set
  - a new "value" is created to represent errors $\bot$
  - operations are defined on this simplified set
- For our example
  - all booleans values are approximated by the type *bool*
  - $\wedge$ and $\vee$ are both transformed in $\otimes$
    - $bool \otimes bool = bool$ and $\bot \otimes x = x \otimes \bot = \bot$
  - the environment is approximated by a type environment Γ

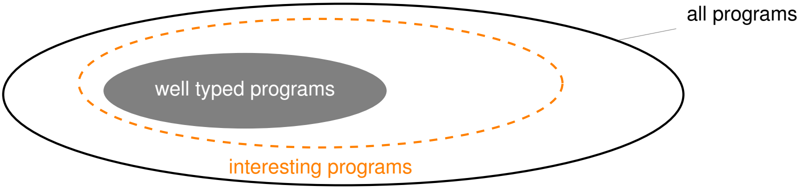
$$(1)\ \Gamma \vdash \texttt{Bool}\ b : bool \qquad (3)\ \frac{\Gamma \vdash F_1 : b_1 \qquad \Gamma \vdash F_2 : b_2}{\Gamma \vdash \texttt{And}(F_1, F_2) : b_1 \otimes b_2}$$

$$(2)\ \Gamma \vdash \texttt{Var}\ x : \Gamma(x) \qquad (4)\ \frac{\Gamma \vdash F_1 : b_1 \qquad \Gamma \vdash F_2 : b_2}{\Gamma \vdash \texttt{Or}(F_1, F_2) : b_1 \otimes b_2}$$

Subject reduction theorem (safety)

$$\varnothing \vdash P : \tau \wedge \tau \neq \bot \implies (\varnothing \vdash P \Rightarrow v \wedge \varnothing \vdash v : \tau) \vee \varnothing \vdash P \overset{\infty}{\Rightarrow}$$

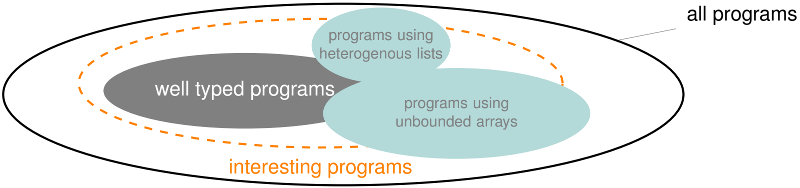
- *Well-typed programs cannot "go wrong"* (produce errors)
- Furthermore computing types is much cheaper than evaluating
- ⚠ In general, Halting and Error discovery are undecidable
- Any typing must rejects correct (but too complicated) programs



Subject reduction theorem (safety)

$$\varnothing \vdash P : \tau \wedge \tau \neq \bot \implies (\varnothing \vdash P \Rightarrow v \wedge \varnothing \vdash v : \tau) \vee \varnothing \vdash P \overset{\infty}{\Rightarrow}$$

- *Well-typed programs cannot "go wrong"* (produce errors)
- Furthermore computing types is much cheaper than evaluating
- ⚠ In general, Halting and Error discovery are undecidable
- Any typing must rejects correct (but too complicated) programs
  - ⇒ needs a compromise between flexibility and safety
  - ⇒ to achieve safety, runtime checks are often needed



- Type algebra are often much more complex
  - more values
  - new types during typing...
  - specific relation between types (*e.g.* subtyping)
- Often the developer must add type annotations
  - type checking: given Γ, P and τ, is $\Gamma \vdash P : \tau$ true?
  - the simply typed λ-calculus
    - syntax: $M ::= \ldots \mid \lambda x\!:\!\tau.M$ and types $\tau ::= \tau \to \tau$
    - typing

$$\frac{x\!:\!\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x\!:\!\tau \vdash M : \tau'}{\Gamma \vdash \lambda x\!:\!\tau.M : \tau \to \tau'} \qquad \frac{\Gamma \vdash M_1 : \tau \to \tau' \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 M_2 : \tau'}$$

- With no annotation, it is Type Inference
  - typeability: given P finds Γ and τ such that $\Gamma \vdash P : \tau$ is true
  - much harder

- The step containing the most difficult algorithms and heuristics
- It consists in transformation of the AST to reduce certain consumption (time, memory, energy, ...)
- Can be generic or specific to a target
- For example
  - for all b, $\mathbf{F} \wedge b = \mathbf{F}$
  - so `And(Bool false,`F`)` can be transformed in `Bool false`
- In real life much more complex!
  - taking out of loop code not depending on the loop
  - loop unrolling
  - propagating constants, inlining small functions
  - removing dead code
  - transforming variables into StaticSingleAssignment
  - tail-call, closure elimination
  - ...

- Transform each element of the AST to machine operation
- For example, let's suppose the following machine
  - it manipulates only one bit
  - it has three registers RA, RB and RC (of one bit)
  - it has a memory of 16 bits (M0 to M15) (initialized before running)
  - it supports the following operations
    - set a register to either 0 or 1 SRxb
    - loading a form memory to a register LMiRx
    - the nand[3] NRxRyRz puts Rx nand Ry in Rz
  - during typing, a formula containing more than 16 variables will be rejected and we will build a mapping firm variable names to memory location denoted $\mathcal{M}$
  - translation rules will be of the following form

$$\mathcal{M}, \mathcal{R} \vdash \text{AST term} \rightsquigarrow \text{instructions sequence}$$

$\mathcal{R}$ carries the register to hold the result, initialized to RA

[3] it is and followed by not (1 nand 1 = 0 and 0 nand b = b nand 0 = 1)

Example

$$(1)\quad \mathcal{M},\mathcal{R} \vdash \texttt{Bool}\ \texttt{\textbf{true}} \leadsto \mathrm{S}\mathcal{R}1$$

$$(2)\quad \mathcal{M},\mathcal{R} \vdash \texttt{Bool}\ \texttt{\textbf{false}} \leadsto \mathrm{S}\mathcal{R}0$$

$$(3)\quad \frac{x \in dom(\mathcal{M})}{\mathcal{M},\mathcal{R} \vdash \texttt{Var}\ x \leadsto \mathrm{L}\mathcal{M}(x)\mathcal{R}}$$

$$(4)\quad \frac{\mathcal{M},\textsc{ra} \vdash F_1 \leadsto is_1 \qquad \mathcal{M},\textsc{rb} \vdash F_2 \leadsto is_2}{\mathcal{M},\mathcal{R} \vdash \texttt{And}(F_1, F_2) \leadsto is_1\, is_2 \textsc{Nrarbrc}\textsc{Nrcrc}\mathcal{R}}$$

$$(5)\quad \frac{\mathcal{M},\textsc{ra} \vdash F_1 \leadsto is_1 \qquad \mathcal{M},\textsc{rb} \vdash F_2 \leadsto is_2}{\mathcal{M},\mathcal{R} \vdash \texttt{Or}(F_1, F_2) \leadsto is_1\, is_2 \textsc{Nrarara}\textsc{Nrbrbrb}\textsc{Nrarb}\mathcal{R}}$$

- `true or x1 and x2 or false` compiles to Sra1Lm0rb NraraRaNrbrbrbNrarbraLm1raSrb0NraraRaNrbrbrb NrarbrbNrararcNrcrcra

- Just a very fast introduction to compilation
- Practice will help concretize!
  - a stack machine language PFX and its execution
  - a micro functional language EXPR, its evaluation and its translation to PFX
- Formalization is important and often forgotten by engineers, that's an error!
- Vocabulary
  - abstract machine, token, sentence, typing, translation rule, pattern, action, abstract syntax tree, interpreter, value, undecidable, type checking, type inference, typeability,
- Acronym
  - LR1, AST, REPL, SOS, SSA