

II: Towards a Simple Semantic Framework for Compiler Construction

Christiano Braga

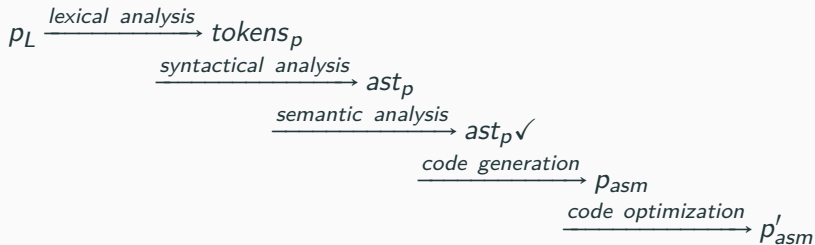
April 12, 2019

Universidade Federal Fluminense and
FADoSS Research Group, Universidad Complutense de Madrid

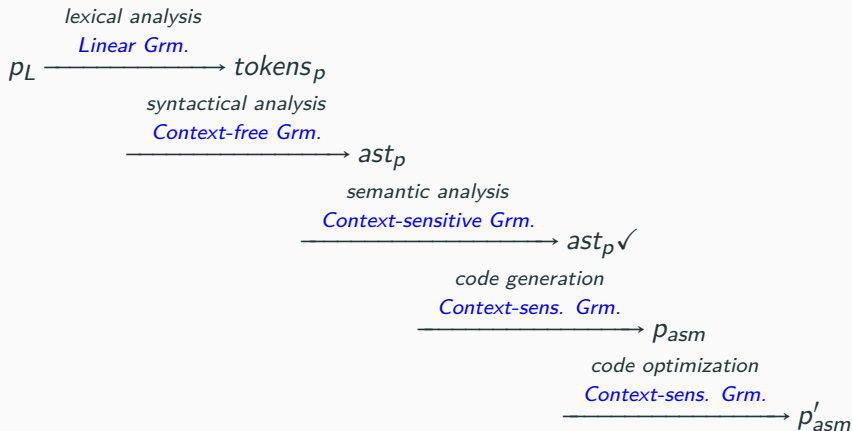
Acknowledgements

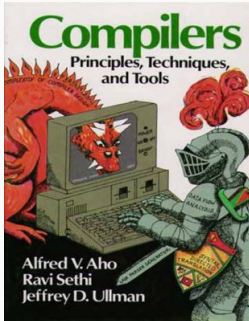
- Thanks to the FADoSS group, and to Professor Narciso Martí Olet in particular, for partially sponsoring this visit to UCM and the presentation of this work at SAC 2019.
- Thanks to Fabricio Chalub, E. Hermann Haeusler, José Meseguer and Peter D. Mosses, for the long term collaboration that built the foundations of this work.

Standard approach to compiler construction



... and Chomsky's hierarchy





What about fighting the dragon ...



... with *semantic* weaponry
instead?

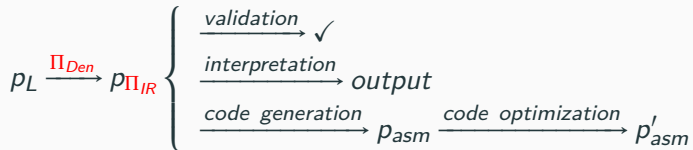
<https://llvm.org/img/DragonMedium.png>



<http://github.com/ChristianoBraga/PiFramework>

Π

Π approach to compiler construction i



- Π_{IR} semantics is given in terms of Π automata: a simple stack-based machine.
- Π automata mimic computation of postfix expressions, like an “HP calculator”.

Π approach to compiler construction ii

Π denotations:

$$\llbracket \cdot \rrbracket_{\Pi_{Den}} : G_L \rightarrow G_{\Pi_{IR}}$$

Π automata: Let $L(G_{\Pi_{IR}}) = \Pi_{IR}$ programs, $L(G_{\Pi_t}) = \Pi_{IR}$ programs with computed values and opcodes¹, the set Q is the disjoint union of semantic components, such as $\kappa = L(G_{\Pi_t})^*$ is the control stack, $\nu = L(G_{\Pi_t})^*$ is the value stack, σ is the memory store, ρ is the environment, a Π automaton \mathcal{A} is the tuple

$$\mathcal{A} = \langle L(G_{\Pi_t}), Q, \delta, q_0, F \rangle$$

where $q_0 \in Q$, $F \subseteq Q$, $\delta : Q \rightarrow Q$.

¹Statements used during the Π automaton-evaluation of a program.

$\langle \text{PiIR} \rangle ::= \langle \text{Exp} \rangle \mid \langle \text{Cmd} \rangle \mid \langle \text{Dec} \rangle \mid \langle \text{Abs} \rangle$

- Declarations: statements that create an environment, binding identifiers to (bindable) values.

$\langle \text{Dec} \rangle ::= \text{Bind}(\langle \text{Id} \rangle, \langle \text{Exp} \rangle)$

$\langle \text{Cmd} \rangle ::= \text{Blk}(\langle \text{Dec} \rangle, \langle \text{Cmd} \rangle)$

- Abstractions: extend Bindables by allowing a name to be bound to a list of formal parameters, a list of identifiers, and a block in the environment. Such names can be called and applied to actual parameters, a list of expressions.

`<Abs> ::= Abs(<Formals>, <Blk>)`

`<Formals> ::= <Id>*`

`<Cmd> ::= Call(<Id>, <Actuals>)`

`<Actuals> ::= <Exp>*`

- Π_{IR} constructions may be *functional* or *relational*.
 - Functional constructions are understood as terminating functions.
 - Relational constructions are understood as relations and may not terminate.

Π automata for blocks and functions i

- Transition function for blocks:

Let $CS = D :: \#BLKDEC :: M :: \#BLKCMD :: C$ and
 $VS = L :: V$,

$$\begin{aligned}\delta(\textcolor{red}{Blk}(D, M) :: C, V, E, S, L) &= \delta(CS, VS, E, S, \emptyset), \\ \delta(\#BLKDEC :: C, E' :: V, E, S, L) &= \delta(C, E :: V, E/E', S, L), \\ \delta(\#BLKCMD :: C, E :: L :: V, E', S, L') &= \delta(C, V, E, S', L), \\ &\text{where } S' = S/L.\end{aligned}$$

- Transition function for abstractions:

$$\delta(\textcolor{red}{Abs}(F, B) :: C, V, E, S, L) = \delta(C, \text{Closure}(F, B, E) :: V, E, S, L)$$

- Transition function for calls:

Let $CS_1 = X_n :: X_{n-1} :: \dots :: X_1 :: \#CALL(I, n) :: C$,
 $A = [V_1, V_2, \dots, V_n]$, $E = [I \mapsto \text{Closure}(F, B, E_1)]E_2$,
 $CS_2 = B :: \#BLKCMD :: C$, and
 $E' = (E_1 / \text{match}(F, [V_1, V_2, \dots, V_n]))$ in,

$$\delta(\text{Call}(I, [X_1, X_2, \dots, X_n]) :: C, V, E, S, L) = \delta(CS_1, V, E, S, L)$$

$$\delta(\#CALL(I, n) :: C, A :: V, E, S, L) = \delta(CS_2, E_2 :: V, E', S, L)$$

A simple example

How to write a compiler using Π ?

1. Write a context-free grammar for the source language.
2. Write a transformation from the source language to Π_{IR} .
 - The implementation of the Π framework will do the heavy lifting.

The language Imp

- Expressions: identifier, arithmetic, Boolean expressions
- Commands: :=, while, if-then-else, ;, let-in
- Declarations: const, var, fn

`<let> ::= 'let' <dec> 'in' <cmd>+`

`<dec> ::= <var> | <fn>`

`<var> ::= 'var' <identifier> '=' <expression>`

`<fn> ::=`

`'fn' <identifier> '(' <formal> ')'` `'=' <cmd>`

`<formal> ::= <identifier> (',' <identifier>)* | ϵ`

Imp code

```
# In this example we encapsulate the iterative
# calculation for the factorial within a call.
let var z = 1
in
    let fn f(x) =
        let var y = x
        in
            while not (y == 0)
            do
                z := z * y
                y := y - 1
in f(10)
```

Some Π denotations for Imp\$

- Function declaration:

$$\llbracket fn(ast) \rrbracket_{\Pi_{Den}} = \textcolor{red}{Bind}(\llbracket fst(ast) \rrbracket, \mathbf{mkAbs}(snd(ast), trd(ast)))$$

- Block command:

$$\begin{aligned} \llbracket let(ast) \rrbracket_{\Pi_{Den}} &= \textcolor{red}{Blk}(\llbracket left(ast) \rrbracket, \mathbf{mkCSeq}(right(ast))), \\ &\quad \text{if } right(ast) \in <cmd>^+ \end{aligned}$$

```
Blk(Bind(Id(z), Ref(Num(1))),  
    Blk(Bind(Id(f),  
            Abs(Id(x),  
                Blk(Bind(Id(y), Ref(Id(x))),  
                    Loop(Not(Eq(Id(y), Num(0))),  
                        CSeq(Assign(Id(z), Mul(Id(z), Id(y))),  
                            Assign(Id(y), Sub(Id(y), Num(1)))))))))  
    Call(Id(f), [Num(10)])))
```

A Maude implementation of Π

- Maude is language and system that implements Meseguer's Rewriting Logic.
- Computations in Maude are realized by *term rewriting*.
 - A term is well-defined and is rewritten with respect to a *module* declaration.
- Some of its features include:
 - Rewriting modulo axioms for identity, commutativity, associativity and idempotence.
 - Built-in Linear Temporal Logic model checking.
 - Metaprogramming.

Π and term rewriting systems

- $\mathcal{A} = (L(G_{\Pi_{IR}})^*, \delta, q_0, F) \Rightarrow \mathcal{T} = (A, \longrightarrow)$, such that $A = Q$ and $\longrightarrow = \delta$.
- Functional and relational constructions in Π_{IR}
 - The semantics of functional constructions are represented by Church-Rosser reduction relations,
 - while relational constructions are represented by an unconstrained reduction relation.
- A Π automaton gives rise to a TRS where its defining rules are *unconditional*.
 - An important characteristic booth from operational and proof-theoretical perspectives.

Write the following *metafunctions*:

1. [Parser] $parseC : Qid \rightarrow ConcreteSyntax_S$, where S is the source language, $ConcreteSyntax_S$ is the module denoting the context-free grammar of the language S .
2. [Parser] $parseA : ConcreteSyntax_S \rightarrow AbstractSyntax_S$
3. [Compiler] $comp : AbstractSyntax_S \rightarrow AbstractSyntax_T$, where T is the target language
4. [Pretty-printer] $pp : AbstractSyntax_T \rightarrow Qid$

1. Write *parseC* and *parseA*, where

$$AbstractSyntax_S = \Pi_{IR}.$$

2. Write a pretty-printer from Π_{IR} to *S*.

The B Maude tool i

(Joint work with Narciso Martí Oliet.)

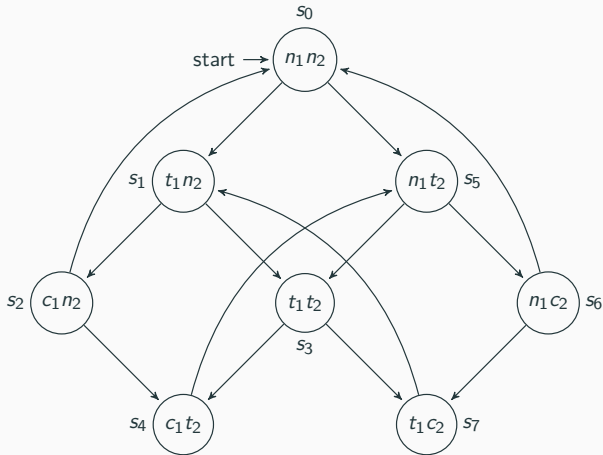


<http://github.com/ChristianoBraga/BMaude>

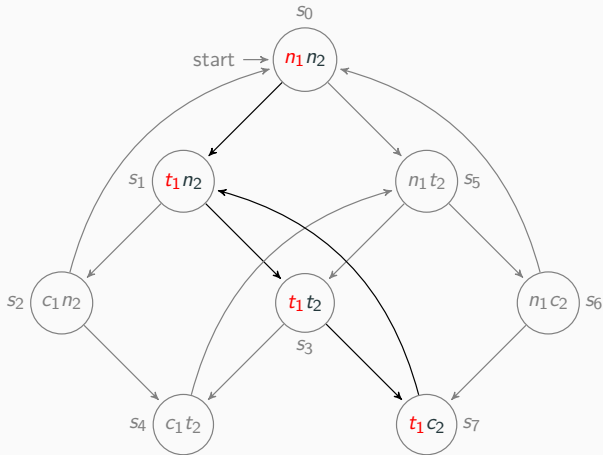
- B Maude is a formal tool for Abstract Machine Notation Descriptions of Abrial's B method implemented in Maude using Π and Maude's metaprogramming facilities.
- It represents AMN programs as Π_{IR} programs.

- We developed a function to compile from AMN to Π_{IR} and a pretty-printer from Π_{IR} to AMN.
- Computations are done at Π automata level.

Mutex analysis in B Maude



Mutex analysis in B Maude - Liveness problem



Mutex analysis in B Maude - specification

```
--- A simple mutual exclusion protocol.
(MACHINE MUTEX
  VARIABLES p1 , p2
  CONSTANTS idle , wait , crit
  VALUES
    p1 = 0 ; p2 = 0 ;
    idle = 0 ; wait = 1 ; crit = 2
  OPERATIONS
    mutex =
      WHILE true DO
        BEGIN
          IF p1 == idle /\ p2 == idle THEN (p1 := wait OR p2 := wait)
          ELSE
            IF p1 == idle /\ p2 == wait THEN p1 := wait OR p2 := crit
            ELSE
              IF p1 == idle /\ p2 == crit THEN p1 := wait OR p2 := idle
              ELSE
                IF p1 == wait /\ p2 == idle THEN p1 := crit OR p2 := wait
                ELSE
                  IF p1 == wait /\ p2 == wait THEN p1 := crit OR p2 := crit
                  ELSE
                    IF p1 == wait /\ p2 == crit THEN p2 := idle
                    ELSE
                      IF p1 == crit /\ p2 == idle THEN p1 := idle OR p2 := wait
                      ELSE
                        IF p1 == crit /\ p2 == wait THEN p1 := idle
                        END END END END END END END END
          END
        END
      END)
END)
```

Mutex analysis in B Maude - checking for liveness

```
(mc mutex() |= [] (p1(1) -> <> p1(2)))
```

BMaude: Model check counter example

Path from the initial state:

```
WHILE(true)...[p1 = 0 p2 = 0]->  
  WHILE(true)...[p1 = 0 p2 = 0]->  
    p2 := wait OR p1 := wait[p1 = 0 p2 = 0]
```

Loop:

```
WHILE(true)...[p1 = 1 p2 = 0]->  
p2 := wait OR p1 := crit[p1 = 1 p2 = 0]->  
WHILE(true)...[p1 = 1 p2 = 1]->  
p2 := crit OR p1 := crit[p1 = 1 p2 = 1]->  
WHILE(true)...[p1 = 1 p2 = 2]
```


Π denotation of Mutex

```
blk(dec(
  dec(ref(gid(bid('p1)), rat(0)),
    ref(gid(bid('p2')), rat(0))),
  dec(dec(cns( gid(bid('idle')), rat(0)),
    dec(cns(gid(bid('wait')), rat(1)),
      cns(gid(bid('crit')), rat(2))))),
    prc(gid(bid('mutex')),
      blk(loop(boo(true),
        if(and(eq(gid(bid('p2')), gid(bid('idle'))),
          eq(gid(bid('p1')), gid(bid('idle')))),
          choice(assign(gid(bid('p1')), gid(bid('wait'))),
            assign(gid(bid('p2')), gid(bid('wait')))),
          if(and(eq(gid(bid('wait')), gid( bid('p2'))),
            eq(gid(bid('p1')), gid(bid('idle')))),
            choice(assign(gid(bid('p1')), gid(bid('wait'))),
              assign(gid(bid( 'p2')), gid(bid('crit')))),
            if(and(eq(gid(bid('p1')), gid(bid('idle'))),
              eq(gid(bid('p2')), gid(bid('crit')))),
              choice( assign(gid(bid('p1')), gid(bid('wait'))),
                assign(gid(bid('p2')), gid(bid('idle')))),
              if(and(eq(gid(bid('wait')), gid( bid('p1'))),
                eq(gid(bid('p2')), gid(bid('idle')))),
                choice(assign(gid(bid('p1')), gid(bid('crit'))),
                  assign(gid(bid( 'p2')), gid(bid('wait')))),
                ...
                if(and(eq(gid(bid('wait')), gid(bid('p2'))),
                  eq(gid(bid('p1')), gid(bid('crit')))),
                  assign(gid(bid('p1')), gid(bid('idle')), nop)))))))))))))
```

Example state of Mutex's Π automaton

```
< env : (gid(bid('p1')) |-> bind(loc(0)), gid(bid('p2')) |-> bind(loc(1))),  
  sto : (loc(0) |-> store(0), loc(1) |-> store(0)),  
  cnt : dec(dec(cns(gid(bid('idle')), rat(0)),  
              dec(cns(gid(bid('wait')), rat(1)),  
                  cns(gid(bid('crit')), rat(2))))), prc(...)) cal(gid(bid('mutex'))) BLK ecs,  
  val : val(noEnv) evs, locs : (loc(0), loc(1)), out : evs, exc : CNT >
```

Coda

A few pros and cons of Π i

Pros

- Semantic functions are *really* semantic functions!
- The semantic framework is quite simple, and uses standard Automata Theory notation.
 - Different implementations may explore different aspects of the compiler construction process.
- Π_{IR} allows us to *focus* the semantic actions on small set of constructions.

Cons

- Π_{IR} is Turing-complete, but it has many limitations in its current form.
- The program transformation step requires engineering.
 - There are libraries in some programming languages that ease this process.
- Currently no support for type-checking.

- Mosses' Component-based semantics: Π_{IR} is a subset of CBS' funcons.
- Plotkin's Interpreting Automata: Π automata generalizes IA, inspired by Mosses' Modular SOS and set-rewriting in Meserguer's Rewriting Logic.
- Roşu's K Framework has similar foundations, but it evolved towards specific notation to hide context and rebase its foundations on top of Matching Logic.

- A proposal to approach compiler construction from a semantics perspective.
- Π is a *semantic* framework for compiler construction.
 - Focus on semantics rather syntax.
- Its underlying formalism is Automata Theory.
 - Different implementations of Π may take advantage of its underlying coding platform, such as rewriting-modulo, narrowing, metaprogramming in Maude and LLVM binding in Python.

- A reduced, but Turing-computable, IR helps make the framework simple to use.
 - Particularly relevant for teaching.
 - We conjecture that its simplicity will allow us to cover more ground, including code generation, static analysis and validation.
 - But also yields a fair framework for developing formal tools (such as B Maude).

Future work

- Π_{IR} : support for reactive programs with co-routines on top of continuations. (Joint work with João Pedro Abreu.)
- Support the complete Π_{IR} in the LLVM code generator.
- Incorporate Dynamic Logic model-checking. (Closer to program validation than Temporal Logic.)
- Incorporate the development of optimization passes into the mix. (Via LLVM infrastructure.)
- From a research perspective: head towards static analysis for functional programs.
- From a teaching perspective: better documentation and tool support to allow us to cover more ground faster to include validation and static analysis in the mix.

II: Towards a Simple Semantic Framework for Compiler Construction

Christiano Braga

April 12, 2019

Universidade Federal Fluminense and
FADoSS Research Group, Universidad Complutense de Madrid