Π: 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 Oliet 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

$$
\begin{aligned}
p_L &\xrightarrow{\textit{lexical analysis}} \textit{tokens}_p \\
&\xrightarrow{\textit{syntactical analysis}} \textit{ast}_p \\
&\xrightarrow{\textit{semantic analysis}} \textit{ast}_p\checkmark \\
&\xrightarrow{\textit{code generation}} p_{asm} \\
&\xrightarrow{\textit{code optimization}} p'_{asm}
\end{aligned}
$$

… and Chomsky's hierarchy

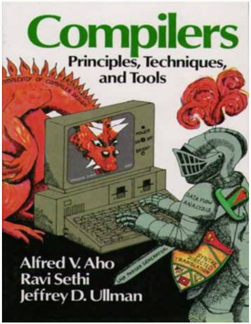
$$p_L \xrightarrow{\substack{\textit{lexical analysis}\\ \textit{Linear Grm.}}} tokens_p$$

$$\xrightarrow{\substack{\textit{syntactical analysis}\\ \textit{Context-free Grm.}}} ast_p$$

$$\xrightarrow{\substack{\textit{semantic analysis}\\ \textit{Context-sensitive Grm.}}} ast_p\checkmark$$

$$\xrightarrow{\substack{\textit{code generation}\\ \textit{Context-sens. Grm.}}} p_{asm}$$

$$\xrightarrow{\substack{\textit{code optimization}\\ \textit{Context-sens. Grm.}}} p'_{asm}$$



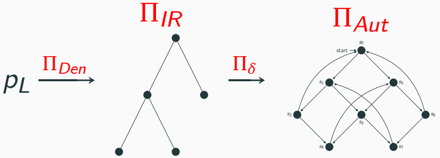
What about fighting the dragon ...

http://github.com/ChristianoBraga/PiFramework

… with *semantic* weaponry instead?

Π

Π approach to compiler construction i



Π approach to compiler construction ii

$$
p_L \xrightarrow{\Pi_{Den}} p_{\Pi_{IR}} \begin{cases} \xrightarrow{validation} \checkmark \\ \xrightarrow{interpretation} output \\ \xrightarrow{code\ \ generation} p_{asm} \xrightarrow{code\ \ optimization} p'_{asm} \end{cases}
$$

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

Π approach to compiler construction iii

Π denotations:

$$[\![\cdot]\!]_{\Pi_{Den}} : G_L \to G_{\Pi_{IR}}$$

Π automata: Let $L(G_{\Pi_{IR}}) = \Pi_{IR}$ programs, $L(G_{\Pi_\iota}) = \Pi_{IR}$ programs with computed values and opcodes[1], the set **Q is the disjoint union of semantic components**, such as $\kappa = L(G_{\Pi_\iota})^*$ is the control stack, $\nu = L(G_{\Pi_\iota})^*$ is the value stack, σ is the memory store, ρ is the environment, a Π automaton $\mathscr{A}$ is the tuple

$$\mathscr{A} = \langle L(G_{\Pi_\iota}), Q, \delta, q_0, F \rangle$$

where $q_0 \in Q$, $F \subseteq Q$, $\delta : \mathbf{Q} \to \mathbf{Q}$.

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

Π_{IR} i

```
<PiIR> ::= <Exp> | <Cmd> | <Dec> | <Abs>
```

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

  ```
  <Dec> ::= Bind(<Id>, <Exp>)
  <Cmd> ::= Blk(<Dec>, <Cmd>)
  ```

- 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
  $VS = L :: V$,

$$\delta(Blk(D, M) :: C, V, E, S, L) = \delta(CS, VS, E, S, \varnothing),$$

$$\delta(\#BLKDEC :: C, E' :: V, E, S, L) = \delta(C, E :: V, E/E', S, L),$$

$$\delta(\#BLKCMD :: C, E :: L :: V, E', S, L') = \delta(C, V, E, S', L),$$

$$\text{where } S' = S/L.$$

- Transition function for abstractions:

$$\delta(Abs(F, B) :: C, V, E, S, L) = \delta(C, Closure(F, B, E) :: V, E, S, L)$$

Π automata for blocks and functions ii

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

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

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

A simple example

How to write a compiler using Π?

1. Write a context-free gramar 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:

$$[\![fn(ast)]\!]_{\Pi_{Den}} = Bind([\![fst(ast)]\!], \mathbf{mkAbs}(snd(ast), trd(ast))$$

- Block command:

$$[\![let(ast)]\!]_{\Pi_{Den}} = Blk([\![left(ast)]\!], \mathbf{mkCSeq}(right(ast))), \\ \text{if } right(ast) \in < cmd >^{+}$$

Imp AST in Π_{IR}

```
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 i

- 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

- $\mathscr{A} = (L(G_{\Pi_{IR}})^*, \delta, q_0, F) \Rightarrow \mathscr{T} = (A, \longrightarrow)$, 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.

Writing a compiler in Maude i

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] $pp : AbstractSyntax_T \rightarrow Qid$

Writing a compiler in Maude with Π

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

$$\mathit{AbstractSyntax}_S = \Pi_{IR}.$$

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

The B Maude tool i

(Joint work with Narciso Martí Oliet.)

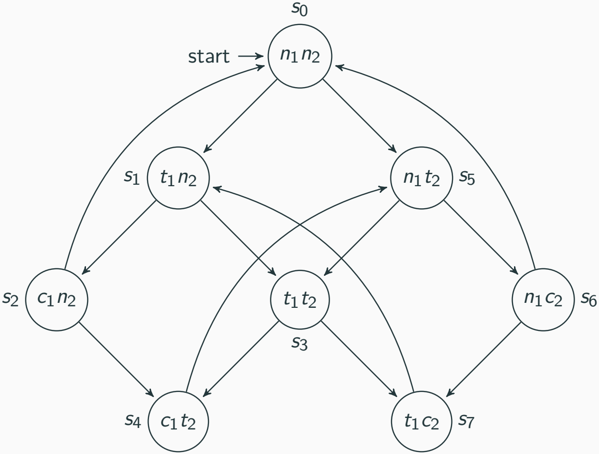
http://github.com/ChristianoBraga/BMaude

- 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.

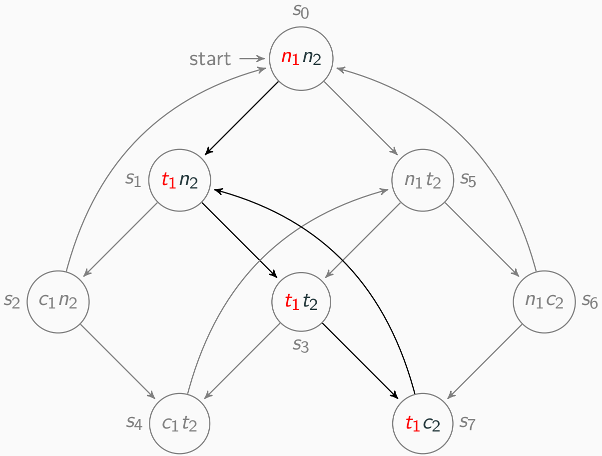
The B Maude tool ii

- 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)
```

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.

A few pros and cons of Π ii

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.

Related work

- 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.
- Roșu's K Framework has similar foundations, but it evolved towards specific notation to hide context and rebase its foundations on top of Matching Logic.

Conclusions i

- 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.

Conclusions ii

- 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.

Π: Towards a Simple Semantic Framework for Compiler Construction

Christiano Braga

April 12, 2019

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