Π: Towards a Simple Semantic Framework for Compiler Construction

Christiano Braga

April 9, 2019

Universidade Federal Fluminense

Acknowledgements

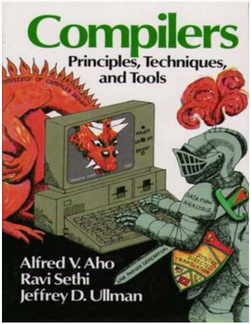
- Thanks to the FADoSS group, and to Professor Narciso Martí Oliet in particular, for partially sponsoring this visit to UCM and the presentation of this work at SAC 2019.
- Thanks to Fabricio Chalub, E. Hermann Haeusler, José Meseguer and Peter D. Mosses, for the long term collaboration that built the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

Standard approach to compiler construction

$$p_L \xrightarrow{lexical\ \ analysis} tokens_p \xrightarrow{syntactical\ \ analysis} ast_p \xrightarrow{semantic\ \ analysis} ast_p\checkmark \xrightarrow{code\ \ generation} p_{asm} \xrightarrow{code\ \ optimization} p'_{asm}$$

… 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 ...

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

… with *semantic* weaponry instead?

http://github.com/ChristianoBraga/PiFramework

Π

Π approach to compiler construction i

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

Π approach to compiler construction i

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

- 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".
- Focus on *semantics* (while teaching), and not so much on syntax, such as (la)LR(k) or PEG grammars and their algorithms.

Π approach to compiler construction ii

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 $q_0 \in Q$, $F \subseteq Q$, $\delta : Q \to Q$.

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

Π approach to compiler construction iii

$$[\![p_L]\!]_{\Pi_{Den}} = p_{\Pi_{IR}} \;\; ; \;\; [\![p_{\Pi_{IR}}]\!]_{\Pi_{Aut}} \models \Lambda$$

$$\Lambda = \begin{cases} \varphi_{DL} & \text{program verification} \\ (\exists\ O), interp(O) & \text{program execution} \\ typecheck() & \text{static analysis} \\ (\exists\ p_{asm}), codegen; optimize_{\omega}(p_{asm}) & \vdots \\ \dots & \end{cases}$$

A simple example

The language Imp

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

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:

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

- Block command:

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

Imp ast

```
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)])))
```

Π automata for blocks and functions i

- Transition function for blocks:
  Let $CS = D :: \#BLKDEC :: M :: \#BLKCMD :: C$ and
  $VS = L :: V$,

$$
\begin{aligned}
\delta(Blk(D, M) :: C, V, E, S, L) &= \delta(CS, VS, E, S, \varnothing), \\
\delta(\#DEC :: C, E' :: V, E, S, L) &= \delta(C, E :: V, E/E', S, L), \\
\delta(\#BLK :: C, E :: L :: V, E', S, L') &= \delta(C, V, E, S', L), \\
&\qquad \text{where } S' = S/L.
\end{aligned}
$$

- 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 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.
  - Narrowing, when the relation is defined by unconditional rules.
  - Built-in Linear Temporal Logic model checking.
  - Metaprogramming.

- A Maude module is comprised by an equational theory, either many-sorted, order-sorted or in membership equational logic, and a set of rules.
  - Equations are assumed Church-Rosser and rules are liberal.
  - In the absence of rules, the module is called a functional module and is called a system module otherwise.

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*.

Writing a compiler in Maude i

- A compiler in Maude is a particular case of a general (functional) metatool in Maude.
  - Essentially, one should define *equational module transformations*.
- Maude's REPL gives us (quoted) identifiers (a.k.a qid) denoting an input.
- The built-in function `metaParse`, when applied to a module denoting a grammar, a set of qids, and a term denoting a type, returns a term well-formed according to the given module, or an error.

Writing a compiler in Maude ii

1. [Parser] $parseC : Qid \to ConcreteSyntax_S$, where S is the source language, $ConcreteSyntax_S$ is the module denoting the context-free grammar of the language S.
   - It essentially calls `metaParse` to produce a term denoting the *concrete syntax* of a given program.
2. [Parser] $parseA : ConcreteSyntax_S \to AbstractSyntax_S$
3. [Compiler] $comp : AbstractSyntax_S \to AbstractSyntax_T$, where T is the target language
4. [Pretty-printer] $pp : AbstractSyntax_T \to Qid$

Writing a compiler in Maude with Π

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

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

The B Maude tool

(Joint work with Narciso Martí Oliet.)

http://github.com/ChristianoBraga/BMaude

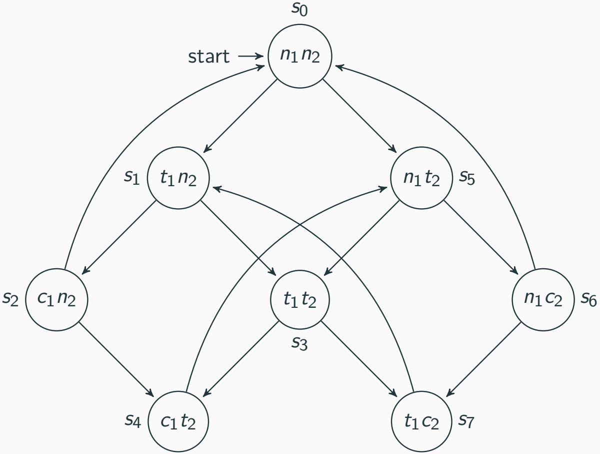
- B Maude is a formal tool for Abstract Machine Notation Descriptions of Abrial's B method implemented in Maude using Π.
- It represents AMN programs as Π_{IR} programs.
- Rewriting, narrowing, and LTL model-checking are computed at Π automata level and then pretty-printed to AMN syntax.

Compiling from AMN to Π_{IR} i

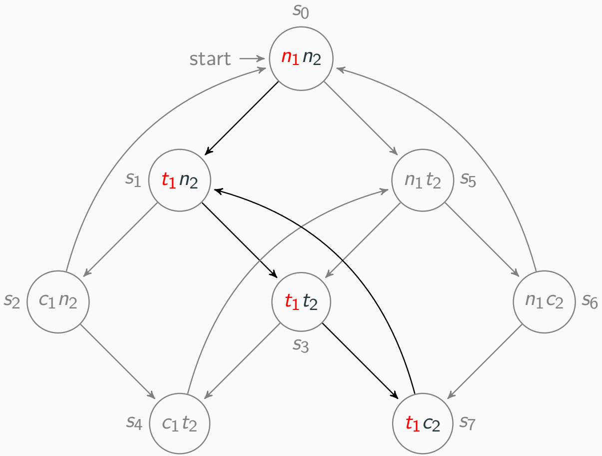
Function `compile` is parseA.

```
op compile : AMNMachine -> Dec .
op compile : AMNClauses -> Dec .
op compile : AMNAbsVariables AMNValuesClause -> Dec .
op compile : AMNAbsConstants AMNValuesClause -> Dec .
op compile : AMNOperations -> Dec .
op compile : AMNOpSet -> Dec .
op compileToFormals : AMNIdList -> Formals .

eq compile( MACHINE I:GSLIdentifiers A:AMNAbsVariables
   C:AMNAbsConstants VS:AMNValuesClause OP:AMNOperations END) =
   dec(compile(A:AMNAbsVariables, VS:AMNValuesClause),
    dec(compile(C:AMNAbsConstants, VS:AMNValuesClause),
     compile(OP:AMNOperations))) .

eq compile(OPERATIONS OS:AMNOpSet) = compile(OS:AMNOpSet) .

eq compile(P:GSLIdentifiers (FS:AMNIdList) = S:GSLSubstitution) =
   prc(compile(P:GSLIdentifiers),
    compileToFormals(FS:AMNIdList), blk(compile(S:GSLSubstitution))) .
```

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 >
```

A Python implementation of Π

Representing Π in Python

- Π denotations are precisely semantic functions in any parsing library for Python.
- Π_{IR} is represented by classes.
    - A Π_{IR} program is an object with a tree structure.
- Π automata are also classes with methods that manipulate objects denoting Π_{IR} programs.
    - One such method is to evaluate a Π_{IR} program.
    - Another such method is a JIT LLVM compiler for a Π_{IR} program. (Joint work with Fernando Mendes.)

Π_{IR} in Python

```
class Loop(Cmd):
    def __init__(self, be, c):
        if isinstance(be, BoolExp):
            if isinstance(c, Cmd):
                Cmd.__init__(self, be, c)
            else:
                raise IllFormed(self, c)
        else:
            raise IllFormed(self, be)
    def cond(self):
        return self.operand(0)
    def body(self):
        return self.operand(1)
```

Π automata in Python i

```
class CmdKW:
    ASSIGN = "#ASSIGN"
    LOOP   = "#LOOP"
    COND   = "#COND"

class CmdPiAut(ExpPiAut):
    def __init__(self):
        self["env"] = Env()
        self["sto"] = Sto()
        ExpPiAut.__init__(self)
```

Π automata in Python ii

```
def __evalLoop(self, c):
    be = c.cond()
    bl = c.body()
    self.pushVal(Loop(be, bl))
    self.pushVal(bl)
    self.pushCnt(CmdKW.LOOP)
    self.pushCnt(be)
```

Π automata in Python iii

```python
def __evalLoopKW(self):
    t = self.popVal()
    if t:
        c = self.popVal()
        lo = self.popVal()
        self.pushCnt(lo)
        self.pushCnt(c)
    else:
        self.popVal()
        self.popVal()
```

An LLVM compiler for Π_{IR} i

(Joint work with Fernando Mendes.)

LLVM code generator uses `llvmlite` library.

```
from pi import *
import llvmlite.ir as ir
import llvmlite.binding as llvm
```

An LLVM compiler for Π_{IR} ii

Class `LLVMExp` gets responsible for creating the basic function block.

```
class LLVMExp():
    def __init__(self, function):
        self.function = function
        self.block =
            function.append_basic_block(name="entry")
        self.builder = ir.IRBuilder(self.block)
```

An LLVM compiler for Π_{IR} iii

Node is Π_{IR} AST.

```
class LLVMCmd(LLVMExp):
    def __init__(self, function):
        self.env = {}
        LLVMExp.__init__(self, function)
```

An LLVM compiler for Π_{IR} iv

```
def compileLoop(self, node):
    loop = self.builder.append_basic_block("loop")
    after_loop =
        self.builder.append_basic_block("after_loop")
    self.builder.branch(loop)
    with self.builder.goto_block(loop):
        cond = self.compile(node.cond())
        block = self.compile(node.body())
        self.builder.cbranch(cond, loop, after_loop)

    self.builder.position_at_start(after_loop)
```

LLVM code generator for Π_{IR} example i

```
# The classic iterative factorial example
let var z = 1
in
    let var y = 10
    in
        while not (y == 0)
        do
            z := z * y
            y := y - 1
```

LLVM code generator for Π_{IR} example ii

```
; ModuleID = "main_module"
target triple = "x86_64-apple-darwin18.0.0"
target datalayout = ""
define i64 @"main_function"()
{entry:
  %"ptr" = alloca i64
  store i64 1, i64* %"ptr"
  %"ptr.1" = alloca i64
  store i64 10, i64* %"ptr.1"
  br label %"loop"
loop:
  %"val" = load i64, i64* %"ptr.1"
  %"temp_eq" = icmp eq i64 %"val", 0
  %"temp_not" = xor i1 %"temp_eq", -1
  %"val.1" = load i64, i64* %"ptr"
  %"val.2" = load i64, i64* %"ptr.1"
  %"tmp_mul" = mul i64 %"val.1", %"val.2"
  store i64 %"tmp_mul", i64* %"ptr"
  %"val.3" = load i64, i64* %"ptr.1"
  %"tmp_sub" = sub i64 %"val.3", 1
  store i64 %"tmp_sub", i64* %"ptr.1"
  br i1 %"temp_not", label %"loop", label %"after_loop"
after_loop:
 ret i64 0
}
```

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.
- Π_{IR} allows us to *focus* the semantic actions on small set of constructions.
- Π underlying formalism is Automata Theory.
  - Different implementations may explore different aspects of the compiler construction process.

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.
    - Take advantage of the underlying coding platform to abstract from parsing issues, such as SLR parsing in Maude and Tatsu library in Python.
- 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 9, 2019

Universidade Federal Fluminense