Whatever happened to meta-programming? (Invited talk)

Gallagher, John Patrick

Published in:

Publication date:
2002

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Whatever Happened to Meta-Programming?

John Gallagher
*University of Bristol*

### Youthful Expectations

**10-15 years ago**
- Meta-interpreters + partial evaluation
- Meta-interpreters “for real” -- to realise language implementation and extensions
- Compiling control, clever and efficient program control
- Self-applicable partial evaluation for generating compilers and compiler-compilers
- Inventing abstract machines
- Reflection (demo, holds, etc) for knowledge management and problem solving
- Conferences for meta-programming (META, Programs as Data,..)

**Progress and continued activity, but…..**

### Middle-age realism

- The dreams live on - the problems are becoming better understood
  - how to express semantics
  - how to make more powerful partial evaluators
  - the essential role of static analysis
  - how to design good object program representation

### Object Language and Meta Language

Object program
P written in
language L

Meta program
Q written in
language M (possibly L=M)

The representation of P in Q
[P] is some data structure in Q

P
Q

AGP’02, Madrid

AGP’02, Madrid

AGP’02, Madrid

AGP’02, Madrid
The real target for tools

- We usually talk of developing language-specific tools, e.g.
  - an analyser for C
  - a transformer for Prolog
  - a verifier for VHDL
- Each tool is complex and hard to develop
- We should analyse, transform, prove etc. in the meta-language
- One meta-language can serve many object languages

One program executing another

- An interpreter for language L, Int_L
- Takes a (representation of) a program in L and some input data, and computes some output data
  \[ \text{Int}_L : \text{Prog}_L \times \text{Input} \rightarrow \text{Output} \]
- The familiar partial evaluation equations (Futamura projections) show that we can “compile” a program
  \[ \text{PE}(\text{Int}_L, p) = \text{“p compiled into the language of Int}_L \text{”} \]

Language Semantics as a Parameter

- A systematic method for developing object language processors
- Explicit object language definitions, written down in a meta-language
  - syntax and semantics definitions
- Construct language independent tools, parameterised by semantics

One more generalisation

- A universal interpreter U
- Takes a language definition (syntax and semantics) as parameter
- Parses and executes a program according to the given syntax and semantics
  \[ U : \text{LangDefs} \times \text{Progs} \times \text{Input} \rightarrow \text{Output} \]
- PE wrt a fixed language definition
  \[ \text{PE}(U, \text{Def}_L) = \text{Int}_L \]
Object language definition

★ Syntax
★ abstract syntax = terms, parse trees
★ utilities (tokenizers and parsers) to convert concrete programs (e.g. strings, flowcharts) to abstract syntax

★ Semantics
★ transitions
★ derivations
★ abstract machine states
★ denotations

Big-step vs. small-step

★ Big-step
★ the total effect of each program construct is expressed
★ initial state ⇒ final state

★ Small-step
★ the immediate effect of each statement is expressed
★ initial state ⇒ next state

Transition Systems

★ Operational semantics
★ Define abstract machine
★ Computation states
★ Transitions - binary relation on states
★ Derivations
★ Transitions can be Big-step or Small-step

Small-step semantics

★ Operational semantics (small-step or structural)
★ a set of computation states Q (configurations) including terminal states F
★ a set of transitions - a relation ⇒ in Q × Q
★ each transition describes a basic computation step
★ derivations (computations)
★ q₀ ⇒ q₁ ⇒ q₂ ⇒ ....
★ terminating derivation if qₙ ∈ F
Big-step semantics

- Operational semantics (big-step or natural)
- a set of computation states \( Q \) (configurations) including terminal states \( F \)
- a set of transitions - a relation \( \Rightarrow \) in \( Q \times Q \)
- each transition describes a complete computation for some program construct
- derivations and transitions are not distinguished

Example - simple imperative language

(Peralta, Gallagher, Saglam, SAS’98, Peralta, PhD 2000)

- Computation state is a pair <Code, Store>
- In a terminal state, the Code is empty (\( \varepsilon \))

\[
\begin{align*}
\langle x := \text{expr}, \tau \rangle & \Rightarrow \langle \varepsilon, [x/\text{val}('x')] \rangle \\
\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \tau \rangle & \Rightarrow \langle s_1, \tau \rangle \text{ if } \text{val}(b, \tau) = \text{true} \\
\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \tau \rangle & \Rightarrow \langle s_2, \tau \rangle \text{ if } \text{val}(b, \tau) = \text{false} \\
\langle s_1; s_2, \tau \rangle & \Rightarrow \langle s_1; s_2, \tau \rangle \text{ if } \langle s_1, \tau \rangle \Rightarrow \langle s_11, \tau \rangle \text{ (} s_11 \neq \varepsilon \text{)} \\
\langle s_1; s_2, \tau \rangle & \Rightarrow \langle s_2, \tau \rangle \text{ if } \langle s_1, \tau \rangle \Rightarrow \langle s_11, \tau \rangle \text{ (} s_11 = \varepsilon \text{)} \\
\langle \text{while } b \text{ do } s, \tau \rangle & \Rightarrow \langle \text{if } b \text{ then } (s; \text{while } b \text{ do } s) \text{ else } \text{skip}, \tau \rangle \\
\langle \text{skip}, \tau \rangle & \Rightarrow \langle \varepsilon, \tau \rangle
\end{align*}
\]

Interpreter for small-step semantics

- a predicate \( \text{transition}(S1, S2) \) to represent \( S1 \Rightarrow S2 \)
- a predicate \( \text{derivation}(D) \) where \( D=[q0,q1,q2,...] \)

Syntax representation

\[
\begin{align*}
x & := \text{expr} & \text{assign}(X, \text{Expr}) \\
\text{if } b \text{ then } s_1 \text{ else } s_2 & \Rightarrow \text{ifte}(B, S1, S2) \\
s_1; s_2 & \Rightarrow \text{compose}(S1, S2) \\
\text{while } b \text{ do } s & \Rightarrow \text{while}(B, S) \\
\text{skip} & \Rightarrow \text{skip}
\end{align*}
\]

\[
\begin{align*}
i & := 2; & \text{compose}(\text{assign}(i, 2),) \\
j & := 0; & \text{compose}(\text{assign}(j, 0),) \\
\text{while } (n^* n > 1) & \Rightarrow \text{ifte}(n^* n = 2, \text{assign}(i, i+4), \\
\{ \text{if } (n^* n = 2) \text{ then } i := i+4; \text{ else } \} \\
\{ j := j+1; \} \\
\}) \\
\} & \Rightarrow \text{compose}(\text{assign}(i, i+2), \text{assign}(j, j+1))
\end{align*}
\]
Representation of transitions

Represent configuration $c(\text{stmt}, t)$ as $c(\text{stmt}, t)$

$$
\text{transition}(c(\text{assign}(X, \text{Expr}), T), c(\text{empty}, T1)) :- \\
\text{val}(\text{Expr}, T, V), \\
\text{update}(T, T1, X, V).
$$

$$
\text{transition}(c(\text{if}(B, S1, S2), T), c(S1, T1)) :- \\
\text{val}(\text{Expr}, T, \text{true}).
$$

$$
\text{transition}(c(\text{if}(B, S1, S2), T), c(S2, T1)) :- \\
\text{val}(\text{Expr}, T, \text{false}).
$$

$$
\text{transition}(c(\text{while}(B, S), T), c(\text{compose}(B, \text{while}(B, S, \text{skip})), T)).
$$

$$
\text{transition}(c(\text{compose}(S1, S2), T), c(\text{compose}(S11, S2, T1)) :- \\
\text{transition}(c(S1, T), c(S11, T1)), S11 \neq \text{empty}.
$$

$$
\text{transition}(c(\text{compose}(S1, S2), T), c(S2, T1)) :- \\
\text{transition}(c(S1, T), c(S11, T1)), S11 = \text{empty}.
$$

Program Specialization

- $\text{PE}(P, \text{input}_1) = P_{\text{input}_1}$
- $P(\text{input}_1, \text{input}_2) = P_{\text{input}_1}(\text{input}_2)$
- when applied to language interpreters, can be regarded as generalised compilation
- $\text{PE}(\text{Int}, P) = \text{Int}_P$
- $\text{Int}_P$ is a translation of $P$ into the language of $\text{Int}$
- self-application yields compiler
- $\text{PE}(\text{PE}, \text{Int}) = \text{PE}_{\text{Int}_{\text{Int}}}$
  and $\text{PE}_{\text{Int}_{\text{Int}}}(P) = \text{Int}_P$

Analysing at Meta-Level

The program above was submitted to a convex-hull analyser for CLP.
Analysis returns a set of linear constraints associated to each predicate call. E.g. (partial result).

$$
\text{d7\_query}(I, J, N) :- J >= 0, J-0.5*I =< -1.0
$$

$$
\text{d9\_query}(I, J, N) :- J >= 0, J-0.5*I =< -2.0
$$

Each predicate $d1$, $d2$, etc. corresponds to an imperative program point. Thus the results can be interpreted back to the object level easily.
Big-step semantics

- The total effect of a complete language construct is specified
- Often specified in compositional style
- `<Statement, Init, Final>`

\[
\begin{align*}
\langle x := \text{expr}, t, t[x/\text{val}(x,t)] > & \quad <s_1, t, t_1 > <s_2, t_1, t_2> <s_1 ; s_2, t, t_2> \\
\text{val}(b,t) = \text{true}, <s_1, t, t_1> & \quad <\text{if } b \text{ then } s_1 \text{ else } s_2, t, t_1> <s_1 ; s_2, t, t_2> \\
\text{val}(b,t) = \text{false}, <s_2, t, t_1> & \quad <\text{if } b \text{ then } s_1 \text{ else } s_2, t, t_1>
\end{align*}
\]

Specialized Big-step interpreter

i := 2;
j := 0;
while (n*n > 1) do
  {if (n*n = 2) then
    i := i+4;
  ...
\}

More complex analysis?
Variables are duplicated
Answers must be propagated during analysis

Big-step interpreter

- The big-step transition relation functions also as a derivation relation.

- transition(S, Init, Final) can be specialized w.r.t. a given program S.

- The resulting program differs from the small-step semantics
- intermediate states are visible.
- program is not tail recursive

Labelled Transition Semantics

- A compositional style of small-step semantics
- Used extensively for process algebras
- Process language with constructs a.P, P1+P2, P|Q, P\L, P[f] (Milner)
- (Labelled) transitions, and derivations, are defined similarly to the imperative language
- Specialized meta-programs have been “model checked” using static analysis techniques (e.g. Leuschel et al.)

\[
\begin{align*}
\text{a}.P & \Rightarrow \text{a}.P \\
P & \Rightarrow P_1 + P_2 \\
\text{P|Q} & \Rightarrow \text{P|Q} \\
\text{Q} & \Rightarrow \text{Q1} \\
\text{P|Q} & \Rightarrow \text{P|Q1}
\end{align*}
\]
“Low-level” meta-language

★ We can use a low-level meta-language to represent the semantics of a high-level object-language

★ E.g. proof mechanism for full first order logic realised by Horn clause program
★ Search space of prover pruned using analysis of Horn clause meta-program.

---

Static Analysis of Term Structure

Aim of set-based analysis (a.k.a. type inference- to find a safe approximation of the set of values that can appear at a given program point (work goes back to [Reynolds, 1968])

\[
\begin{align*}
q([]\,X\,X) & . \\
q([c\,(X1)\,Y]\,Acc\,X) & : - \\
& \text{integer}(X1), \ q(Y\,c\,(X1\,Acc)\,X). \\
q([d\,(X1)\,Y]\,Acc\,X) & : - \\
& \text{integer}(X1), \ q(Y\,d\,(X1\,Acc)\,X). \\
p(X\,Y) & : - q(X\,0\,Y).
\end{align*}
\]

\[
S_Y ::= 0 \mid c(\text{Int}, S_Y) \mid d(\text{Int}, S_Y) \quad (S_Y \text{ is an infinite regular set of terms})
\]

---

Requirements of meta-language

★ convenient representation of syntax
★ important only for readability
★ representation of transitions
★ relational style fits better with traditional style of semantic specification
★ non-determinism
★ fixpoint semantics
★ framework for abstract interpretation

---

Uses of Set-based Analysis

For imperative program analysis the environment stack is modelled as a component of the state.

The stack can be unbounded in the presence of recursive procedures (hence, stack must be approximated)

Over approximation means that control is undefined at procedure exit.

Details - Gallagher-Peralta (HOSC, 2001)
Current and Future Work

- Using set-based analysis of meta-programs
- Verification experiments
- infinite state model checking [Charatonik & Podelski]
- cryptographic protocols [Abadi & Blanchet]
- using meta-language representation of protocol
- Shape analysis in imperative language

Some day….

- Programming will cease to be an error-prone and time-consuming handcraft
- Most programs will be generated, verified by other programs
- Meta-programming is one of the unifying principles of computing
- Systematic meta-programming is key

Related Research

- The Boyer-Moore prover (ACL2) (and other provers) is essentially a meta-level approach
- First, systematically translate the object program/system to a LISP-like functional program
- Verify the functional program
- Interpret the results back to the object program
- The Supercompiler project (Turchin et al.)
- target Java - execute Java via meta-level interpreter, and optimise at that level