How should declarative and imperative programming be used together?

July 18, 2018

Peter Van Roy
Université catholique de Louvain
LPOP Workshop (colocated with FLoC 2018)
Oxford, UK

Towards a formal justification and foundation for the architectures we saw today for the RBAC (Roll-Based Access Control) challenge
Definitions of declarative and imperative

- **Declarative programming**
  - Any computation model that *infers true consequences from an initial expression* (e.g., doing logical deduction from a set of axioms)
  - For example, functional programming (\(\lambda\) calculus) or logic programming (pure subset of Prolog, constraint solving)

- **Imperative programming**
  - Any computation model that *has memory cells with read and write operations*, such that each read returns the argument of the last previous write done during the execution
  - For example, the \(\lambda\) calculus extended with memory cells or standard object-oriented programming

- How should these two paradigms be used together when building software systems?
Client/server example

- Each client sends requests to the server and receives replies
- The server must handle each client’s requests in reasonable time
- The server cannot be programmed declaratively (e.g., with λ calculus)
  - Because the order of client requests is not known in advance
  - The server needs a nondeterministic choice, else clients are not independent
Declarative programming: \( \lambda \) calculus

- Process calculus for expressing any computation based on functional abstraction and application

- Syntax
  - \( x ::= \) (variables)
  - \( t ::= x \mid (\lambda x. t) \mid (t_1 t_2) \)

- Semantics (using substitution operation \( t[x] \))
  - \((\lambda x. t[x]) \rightarrow (\lambda y. t[y])\) \(\alpha\)-conversion
  - \(((\lambda x. t_1) t_2) \rightarrow t_1[x:=t_2]\) \(\beta\)-reduction
  - \(((\lambda x. (t x)) \rightarrow t \ (if \ x \ not \ free \ in \ t)\) \(\eta\)-conversion
Properties of $\lambda$ calculus

- It is confluent
  - Church-Rosser theorem: The final result of a reduction is the same for all reduction orders (up to variable renaming)
  - This implies that $\lambda$ reduction can be done concurrently without affecting the result

- It cannot express real-world interaction
  - Real-world interaction requires handling inputs coming from the real world during the execution
  - The inputs arrive during the reduction process because \textit{reduction steps take nonzero time}
  - The \textit{inputs are not known in advance}, so they cannot be part of an initial expression
Imperative programming: read-extended $\lambda$ calculus

- We add a read operation to the lambda calculus
  - The read operation models an input from the real world
    (we omit the write operation to simplify our example)

- Syntax
  - $x ::= \text{(variables)}$
  - $t ::= x \mid (\lambda x. t) \mid (t_1 \ t_2) \mid (\rho x. t)$

- Semantics
  - $(\lambda x. t[x]) \rightarrow (\lambda y. t[y]) \quad \alpha$-conversion
  - $((\lambda x. t_1) \ t_2) \rightarrow t_1[x:=t_2] \quad \beta$-reduction
  - $((\lambda x. (t \ x)) \rightarrow t \ (if \ x \ not \ free \ in \ t) \quad \eta$-conversion
  - $(\rho x. t_1) \rightarrow t_1[x:=t_2] \ (t_2 \ is \ external \ input) \quad \rho$-reduction (read)
Software system design

- The $\lambda$ calculus has many desirable properties, such as confluence, easy reasoning, specification, analysis, debugging and verification, easy optimization and maintenance, distributed implementation, etc.
- But it cannot express real-world interaction

- The read-extended $\lambda$ calculus can express real-world interaction, but it is more difficult to program in
  - For most programs, $\rho$-reduction is only needed in a very few places, so the advantages of $\lambda$ calculus are retained for most of the execution
Solution of client/server

- Server uses $\rho$-reduction to read each client request
- The rest of the server can be pure $\lambda$ calculus
General principle

- Use $\lambda$ calculus everywhere, except when interacting with the real world
  - Because inputs arrive during the reduction process that cannot be known in advance

- Use an imperative calculus when interacting with the real world
  - Read/write operations (memory cells) that can be invoked during the reduction process

- A software system should be declarative except where it interacts with the real world
The principle applied to today’s workshop

- All systems presented today follow this structure (serendipity!)
  - The RBAC challenge in XSB, JASP, IDP, LogiQL, Python, DMN, DistAlgo
  - They all have a declarative core and an imperative real-world interface
  - We have justified this structure and we give an approach to formalize it

- The whole system (core + interface) can be defined in one formalism
  - The real-world interface needs an extra operation (such as $\rho$-reduction)
  - Multi-language implementations are just surface differences (Java+Prolog)

- It is important to separate the declarative and imperative parts
  - The separation does not necessarily respect the language boundaries
  - “Declarative” system is partly imperative (e.g., nonmonotonic reasoning)!
    - For example: Prolog is a declarative language with imperative extensions
  - “Imperative” system is partly declarative (e.g., ontology in Java)!
    - For example: JASP is an imperative language with declarative extensions