Climbing Up the Semantic Tower — at Runtime

François-René Rideau
TUNES
fare@tunes.org

Abstract

Software exists at multiple levels of abstraction, where each more concrete level is an implementation of the more abstract level above, in a semantic tower of compilers and/or interpreters. First-class implementations are a reflection protocol to navigate this tower *at runtime*: they enable changing the underlying implementation of a computation *while it is running*. Key is a generalized notion of *safe points* that enable observing a computation at a higher-level than that at which it runs, and therefore to climb up the semantic tower, when at runtime most existing systems only ever allow but to go further down. The protocol was obtained by extracting the computational content of a formal specification for implementations and some of their properties. This approach reconciles two heretofore mutually exclusive fields: Semantics and Runtime Reflection.

CCS Concepts • **Theory of computation → Operational semantics**; **Categorical semantics**; Type theory; • **Software and its engineering → Reflective middleware**; **Runtime environments**; *Just-in-time compilers*;

Keywords First-class, implementation, reflection, semantics, tower

1 Introduction

Semantics predicts properties of computations without running them. Runtime Reflection allows unpredictable modifications to running computations. The two seem opposite, and those who practice one tend to ignore or prohibit the other. This work reconciles them: semantics can specify *what* computations do, reflection can control *how* they do it.

2 Formalizing Implementations

An elementary use of Category Theory can unify Operational Semantics and other common model of computations: potential states of a computation and labelled transitions between them are the nodes ("objects") and arrows ("morphisms") of a category. The implementation of an abstract computation A with a concrete one C is then a "partial functor" from C to A, i.e. given a subset O of "observable" safe points in C, a span of an interpretation functor from O to A and the full embedding of O in C. Partiality is essential: concepts atomic in an abstract calculus usually are not in a more concrete calculus; concrete computations thus include many intermediate steps not immediately meaningful in the abstract.[1]

OBT 2018, January 13, 2018, Los Angeles, California
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

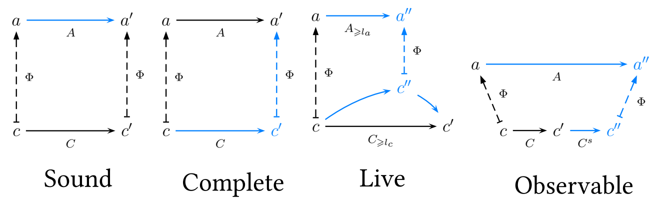

Figure 1: Some properties for implementations to have or not

The mandatory *soundness* criterion is, remarkably, the same as functoriality. Many other interesting properties may or may not be hold for a given implementation: variants of *completeness* guarantee that abstract nodes or arrows are not left unimplemented in the concrete (e.g. can express the notion of simulation [4]); variants of *liveness* guarantee that progress in the abstract is made given enough progress in the concrete (e.g. can express "real time" behavior); and variants of *observability* guarantee that an observable abstract state can be recovered given any intermediate state at which the concrete computation is interrupted. These properties can be visualized using bicolor diagrams such as in figure 1.[2]

3 Extracting a Runtime Protocol

The above properties can be formalized using dependent types; their constructive proofs will then have a *computational content* as per the Curry-Howard Correspondence [3]. *Observability* could thus be formalized in Agda [5] as the type of the following function `observe` where: `.o` and `.⇒` denote node-wise and arrow-wise components; Φ is the interpretation functor opposite the implementation of `A` with `C`; `a` is the starting abstract state concretely implemented by `c` (implicit inputs); `c'` is the concrete state in which `C` was interrupted after effects `f` (explicit input); `c''` is the observable safe point that is being recovered after effects `g` (explicit

[1] For instance, languages in the ALGOL tradition have no notion of explicit data registers or stacks, yet are typically implemented using lower-level machines (virtual or "real") that do; meanwhile their high-level "primitives" each require many low-level instructions to implement.

[2] In these diagrams, computation is from left to right; abstract is above and concrete below; property premises are in black and conclusions in blue; and all diagrams commute. While an implementation is notionally from abstract to concrete, the opposite arrows of Abstract Interpretation are drawn, because functoriality goes from concrete to abstract, which is what matters when diagrams commute; for more details on the diagrams see [6].

output); `safe.⇒` guarantees that g cannot take too much resources or do blocking I/O or require user intervention; and `a''`, h and the last property ensure the diagram commutes (implicit outputs).

```
observe : ∀ {a : A.o} {c : C.o} {Φ.o c a}
 {c' : C.o} (f : C.⇒ c c') →
 ∃ (λ {c'' : C.o} → ∃ (λ (g : C.⇒ c' c'') →
 ∃ (λ {a'' : A.o} → ∃ (λ {h : A.⇒ a a''} →
 ∃ (λ {safe.⇒ g} → Φ.⇒ (C.compose g f) h)))))
```

Erasing dependencies, implicit arguments, compile-time and redundant information, the content can be extracted as a function in a programming language with less precise types:

```
observe : (f : C.⇒) → (g : C.⇒)
```

In lay words, `observe` takes the interrupted fragment of concrete computation and shows how to complete it into one that is observable as an abstract computation.

Similarly, the computational content of completeness is a function that allows to control the concrete computation as if it were the abstract computation. The computational content of liveness is a function that advances the concrete computation enough to advance the abstract computation. All these functions and more form an API that allows arbitrary implementations of arbitrary languages to be treated as first-class objects, usable and *composable* at runtime.

4 Simulating or Performing Effects

Traditional reflection protocols [7] offer interfaces where only state can be reified, and effects always happen as ambient side-effects, except sometimes for limited ad hoc ways to catch them. By contrast, when extracting a protocol from a categorical specification, it becomes obvious that effects too deserve first-class reification, being the arrows of the reified computation category.

One simple way of reifying effects is as a journal recording I/O that happened during the computation — or would happen were the computation to actually run (or run again). More abstract representations can be symbolic, at a higher-level than a low-level logger could record; they could be monadic functions or arbitrary Kliesli arrows. In the end, there are two complementary approaches in which effects are either *simulated* or *performed*. Two functions `simulate` and `perform` may translate one approach into the other: but while `perform` can be written on top of any expressive enough system (at a cost), achieving `simulate` requires rewriting the entire system if the existing implementation does not offer a suitable reflection protocol.

Now the reflection protocol itself includes effects beyond those of the computations being reified and reflected — if only partiality and its dual non-determinism. Implementations and interpretations are partial and may fail on some nodes or arrows. And even if a computation is itself deterministic or at least confluent, running or advancing it includes non-determinism as to how much work will be done according to what evaluation strategy. A logical specification of the protocol must therefore expose these effects.

5 Applications

The protocol, thanks to its crucial notion of observability, enables navigating up and down a computation's semantic tower *while it is running*. Developers can then zoom in and out of levels of abstraction and focus their tools on the right level for whatever issue is at hand, neither too high nor too low. Computations can be migrated from one underlying implementation to the other, one machine or configuration to the other — changing a running engine. Recovering an abstractly observable safe point also enables safely killing threads and upgrading code, thus achieving a robustness that only Erlang [8] can currently provide. Code instrumentations can be seen as the categorical opposites of Natural Transformations; they can be written in a generic way, added to running code, configured independently from code; they can provide orthogonal persistence, access control, time-travel debugging, and other capabilities to all languages. etc.

Each of these applications has been done before, but in heroic ways, available only to one implementation of one language, using some ad hoc notion of safe points (PCLSRing [1], Garbage Collection [9], etc.). The promise of this runtime reflection protocol is to achieve these applications in comparatively simple yet general ways, and made available universally: tools such as shells, debuggers, or code instrumentations, can then work on all possible implementations of all languages, specialized using e.g. typeclasses.

Finally, rooting a reflection protocol in formal methods means it is now possible reason about metaprograms, and maybe even feasably prove them correct; they need no longer invalidate semantic reasoning nor introduce unmanageable complexity.

6 Conclusion and Future Work

The ideas above remain largely unimplemented. But they already provide a new and promising way of looking at either the semantics of implementations or the design of reflection protocols — and more importantly, at the synergy between those two estranged fields. My plan is to further implement the protocol in Gambit Scheme: it already implements observability and migration at the level of its GVM [2], and there is a Racket-like module system called Gerbil to develop closed languages on top of it.

See my presentation at https://youtu.be/heU8NyX5Hus.

Bibliography

[1] Alan Bawden. PCLSRing: Keeping Process State Modular. 1989.

[2] Marc Feeley. Compiling for Multi-language Task Migration. 2015.

[3] William A. Howard. The formulae-as-types notion of construction. 1980.

[4] Robin Milner. An Algebraic Definition of Simulation between Programs. 1971.

[5] Ulf Norell. Dependently typed programming in Agda. 2008.

[6] François-René Rideau. Reconciling Semantics and Reflection. 2018.

[7] Brian Cantwell Smith. Procedural Reflection in Programming Languages. 1982.

[8] Joe Armstrong and Robert Virding and Claes Wikström and Mike Williams. Concurrent Programming in ERLANG. 1993.

[9] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. 1992.