Réconcilier Sémantique et Réflexion

Formalismes, Protocoles, Applications et Architecture

Reconciling Semantics and Reflection

Formalisms, Protocols, Applications and Architecture

François-René RIDEAU ĐẶNG-VŨ Bân

École Nationale Supérieure des Télécommunications
Département Informatique et Réseau

Thèse présentée pour l'obtention du grade de
Docteur en Informatique

BROUILLON: THÈSE NON ENCORE SOUTENUE
DRAFT: THESIS NOT YET DEFENDED

Prepared from LaTeX sources at version `thesis-draft-119-ga030e4e` committed on
`2023-07-18T21:44:24+02:00`

Abstract

I propose an approach to reconciling semantics and reflection in the design of computing systems, two topic often considered mutually exclusive. My approach distinguishes between three dimensions of meta-computation: beyond the well-known ante/post dimension of staged evaluation and code-generation, I identify a hypo/hyper dimension of semantic implementation and refinement, and a back/fore dimension of runtime effect control. My thesis is divided in four parts.

First, I make elementary use of Category Theory (no advance knowledge required) to unify common formalisms for computational semantics. I can then specify a notion of *Implementation* as a partial correspondance between an abstract computation being implemented and a concrete computation implementing it. I study common composable properties that implementations may have; these notably include a key yet often neglected property that I dub *Observability*, by which a state of the abstract computation can be recovered from the interrupted state of concrete computation implementing it.

Second, I use the Curry-Howard Correspondance to systematically extract from the previous formalism a protocol to manipulate implementations as *first-class* entities at runtime. This protocol notably allows to navigate up and down the semantic tower of a computation and zoom in or out of levels of abstraction while the computation is running. I then discuss how to use this protocol to reinterpret classic topics involving implementation, from compilation and static type analysis to aspect-oriented programming and refactoring.

Third, I present potential applications of these formalism and protocol to specify, verify, generalize, compose, or otherwise reinterpret many existing techniques. Notably, *Migration*, whereby a implementation is replaced by another one while the computation is running, can express process migration, garbage collection, zero copy routing, dynamic configuration or JIT compilation. The categorical concept of Natural Transformation, whereby a implementation can be naturally transformed into another, offers an approach to making *Code Instrumentation* uniformly available at all levels of abstraction, which applies to debugging, logging, access control, concurrency control, robustness, orthogonal persistence, and more.

Fourth and last, I explore a reflective architecture, in which software is systematically organized around the use of the previous protocols. Every computation has an explicitly associated semantic tower at runtime; dynamic changes to this tower are migrations controlled by a meta-computation we dub its background controller. Software is then written, distributed and evolved not in terms of applications each embodying an entire tower of semantics, but in terms of smaller components that interact through the reflective protocols. This architecture may apply to any kind of development platform, user interface shell, operating system, or distributed and virtualized application manager. I discuss the potential benefits of this architecture in terms of performance, features, robustness and social organization.

Résumé

Je propose une approche pour réconcilier sémantique et réflexivité dans la conception de systèmes informatiques, deux sujets souvent considérés comme mutuellement exclusifs. Mon approche distingue trois dimensions du méta-calcul : au-delà de la dimension ante/post bien connue de l'évaluation par étapes et de la génération de code, j'identifie une dimension hypo/hyper d'implémentation et de raffinement sémantique, et une dimension arrière/avant de contrôle des effets d'exécution. Ma thèse est divisée en quatre parties.

Primo, je fais un usage élémentaire de la théorie des catégories (aucune connaissance préalable requise) pour unifier des formalismes habituels pour la sémantique des logiciels. Je peux alors spécifier une notion d'*Implémentation* comme correspondance partielle entre un calcul abstrait à implémenter et un calcul concret l'implémentant. J'étudie les propriétés composables courantes que les implémentations peuvent avoir ; ceux-ci incluent notamment une propriété clé mais souvent négligée que je baptise *Observabilité*, par laquelle un état du calcul abstrait peut être récupéré à partir de l'état interrompu du calcul concret l'implémentant.

Secundo, j'utilise la correspondance de Curry-Howard pour extraire systématiquement du formalisme précédent un protocole pour manipuler les implémentations en tant qu'entités *de première classe* au moment de l'exécution. Ce protocole permet notamment de naviguer de haut en bas dans la tour sémantique d'un calcul et d'effectuer un zoom avant ou arrière sur les niveaux d'abstraction pendant l'exécution du calcul. Je discute ensuite de la manière d'utiliser ce protocole pour réinterpréter des sujets classiques impliquant l'implémentation, de la compilation et de l'analyse de type statique à la programmation orientée aspect et à la refactorisation .

Tertio, je présente des applications potentielles de ces formalismes et protocoles pour spécifier, vérifier, généraliser, composer ou autrement réinterpréter de nombreuses techniques existantes. Notamment, la *Migration*, où une implémentation est remplacée par une autre pendant que le calcul est en cours, peut exprimer migration de processus, récupération de place, routage sans copie, configuration dynamique ou compilation JIT. Le concept catégorique de Transformation Naturelle, par lequel une implémentation peut être naturellement transformée en une autre, offre une approche pour rendre l'*Instrumentation de Code* uniformément disponible à tous les niveaux d'abstraction, ce qui s'applique au débogage, à la journalisation, au contrôle d'accès, au contrôle de la concurrence, de la robustesse, de la persistance orthogonale, etc.

Quarto et in fine, j'explore une architecture réflexive, dans lequel les logiciels sont systématiquement organisés autour de l'utilisation des protocoles précédents. Chaque calcul a une tour sémantique explicitement associée au moment de l'exécution ; les changements dynamiques de cette tour sont des migrations contrôlées par un méta-calcul que nous appelons son contrôleur d'arrière-plan. Le logiciel est alors écrit, distribué et évolué non pas en termes d'applications incarnant chacune une tour entière de sémantique, mais en termes de composants plus petits qui interagissent à travers les protocoles réflexifs. Cette architecture peut s'appliquer à tout type de plate-forme de développement, shell d'interface utilisateur, système d'exploitation ou gestionnaire d'applications distribuées et virtualisées. Je discute des avantages potentiels de cette architecture en termes de performances, fonctionnalités, robustesse et organisation sociale.

Contents

Preface

You are reading a draft of my undefended PhD thesis. Indeed this thesis is *not* currently in a state that is defensible as an academic contribution.

It is notably lacking (a) some application, theorem or proof that would be both original and non-trivial, *and* (b) more in-depth discussions of related works and bibliographical references.

Instead, this thesis offers an original *point of view* that generalizes known techniques in *a posteriori* trivial ways, and suggests an original way of architecting software development that remains so far unimplemented.

I originally worked on this PhD thesis between 1997 and 2000 under Jean-Bernard Stefani and Elie Najm, introducing and formalizing the core notions of an implementation and its properties, that constitute the essence of the earlier parts of the thesis, plus a few intuitions of the rest. I completely rewrote the thesis on my own between 2016 and 2017, clarifying and extending the original ideas, and rewriting the latter parts from scratch.

During this rewrite, important changes compared to previously circulated early drafts include: (1) Using category theory to generalize an initial work on rewrite systems with no side-effects. (2) Over time, changing the nomenclature for some concepts so that "soundness" and "observability" are now the respective names for properties previously dubbed "safety" and "soundness" in early drafts. (3) Also, distinguishing the three independent axes of reflection, and the according nomenclature for "implement", etc., instead of "reify" and "reflect". (4) Identifying code instrumentations as the opposites of natural transformations.

I gave talks based on this work at the BostonHaskell meeting of May 2016 `https://youtu.be/heU8NyX5Hus`, at the Lisp NYC meeting or March 2017 `https://www.meetup.com/LispNYC/events/237759785/`, at the Off-the-Beaten-Track workshop at POPL 2018 `https://github.com/fare/climbing`, and at LambdaConf 2018 `https://youtu.be/fH51qhI3hq0`.

I also wrote a very informal blog using swiftian storytelling to expound many of the ideas underlying this thesis: *Houyhnhnm Computing* `https://ngnghm.github.io` (the blog name is pronounced like "Hunam Computing", the protagonist name like "Ann").

A recent draft of this thesis should be available at `http://fare.tunes.org/tmp/phd/thesis.pdf` for which `https://bit.ly/FarePhD` should be a valid shorthand.

Chapter 1

Introduction

Reflection is about inspecting and changing a computation, including at runtime. Semantics is about having a fixed meaning to the computation even before it is run, that is guaranteed not to change at runtime. The two seem antithetical, to the point that people interested in one tend to accept that they won't be interested in the other, at least not at the same time. Yet, the present thesis is about reconciling the two and having both runtime reflection and compile-time semantics at the same time. Informally, the secret to this reconciliation is that semantics can study *what* the computation does, while reflection will control *how* the computation does it.

1.1 Basic Intuitions

1.1.1 Semantic Towers

A piece of modern software can be mindboggingly complex, made of tens or hundreds of millions of lines of code, more than any one developer can hold in his mind, much less possibly understand. However, one powerful way that software is kept simple enough for developers to be able to handle at all, a small piece at a time, is to decompose it into *semantic towers*[citation needed], wherein each layer is the implementation of some more abstract computation using some more concrete computation.

Hence, a typical program implements a user-visible application (a more abstract computation) on top of a programming language (a more concrete computation). The compiler for that programming language implements that language (now seen as the more abstract computation) in terms of a virtual machine (a yet more concrete computation). Then a lower layer (interpreter, JIT compiler, or compiler backend pass) expresses this virtual machine (in turn seen as an abstract computation) in terms of a somewhat portable view of the hardware as provided by the operating system (here seen as a concrete computation). The operating system provides this portable view of the hardware (then seen as an abstraction) by mapping it to the semantics of the actual hardware, CPU, chipset and many attached devices. This actual hardware realizes the documented CPU semantics in term of the underlying digital circuits — though it may itself involve one or more intermediate abstraction levels involving firmware or microcode, wherein instructions as seen by the user are actually made of smaller instructions seen by the actual hardware digital circuits.

These digital circuits are implemented as transistors in terms of analog electrical circuits. The analog electrical circuits are implemented in terms of quantum mechanics. This quantum mechanics might be implemented by God in terms of his own digital physics computer à la Ed

Fredkin[citation needed], though any access to that layer of abstraction would be out of reach of mere humans. Many more abstraction layers may exist above, below, or in the middle, that were omitted in the list above, yet may be usefully added when to analyze some phenomenon: each observation is best done at a suitable layer of the semantic tower — one with the relevant details you care about, yet without the noise of details you don't, with a wider point of view than layers below, yet with a finer resolution than the layers above.

Viewing software in terms of such decompositions and recompositions into layers of abstractions allows developers to only have to focus on one layer at a time, using a programming language adapted to said layer of semantics. Within each tower, they can further divide their computations in terms of smaller entities, whether they are functions and variables, instructions and memory addresses, gates and signals, or fields and particles.

1.1.2 Mentally Navigating the Towers

It is said that good developers can mentally zoom in and out of these levels of abstraction. As they do, they understand that none of these levels possibly contradicts the others, since they are distinct but coherent views on a same overall computation. Thus, they can imagine the behavior of their systems at the level of abstraction that befits the issue at hand, be it a user-visible application feature (or bug), the way it is implemented in a Domain-Specific Language (DSL), or some behavior in a general purpose programming language, a compiler pass, a virtual machine, the processor in user-mode or in system-mode, etc.

It is also said that the most interesting mathematical theorems establish a correspondence between two different structures. With these theorems, mathematicians can then consider those distinct structures as multiple points of view on a same underlying object; then, at least in theory, they can freely switch from one point of view to the other. Hence, they can pick whichever point of view makes a theorem easiest to prove, and use that result in whichever point of view it is most useful.

These two complementary abilities are very important when designing or maintaining computing systems: with them developers may at any time choose the most appropriate way to think about the system to keep the issue at hand tractable, yet the most efficient way to write it to keep its development and/or execution costs affordable.

However, these abilities so far remain purely mental, mostly unaided by software automation and disconnected from the interactive development loop. Developers have to cultivate these multiple points of view in their head (sometimes assisted by pen and paper), and the thinking has to all happen way before runtime, before compile-time, even before code-writing time. Yes, an iterated process can be used whereby the code-writing time for the next version of some code is after the runtime of the previous version; but often developers still have to deal mentally with a lot of abstraction levels at which they receive scant feedback from such runtimes. Developers have to develop models of the software in their brain, that (pending any significant improvement to the capacities of the human brain) remain limited to a increasingly small portion of the software written as software gets more sophisticated.

1.1.3 Navigating the Towers at Runtime

Now, what if those abilities, instead of being required from developers before they may even think about programs, where instead provided and supported by their programming environments? Then, developers would be able to interactively handle and debug programs that are much more complex than they could master mentally. Moreover, the ability to change the point of view on a program at runtime would have wider consequence: it would open new territories for software architecture.

Indeed with this “superpower”, developers could migrate computations from one underlying software and hardware stack to another, while it is running, with all its dynamic state. They could then automatically enable and disable various kinds of code instrumentations, based on various varying conditions: They could change how they monitor computations at runtime, or enable and disable time-travel debugging on a running computation when specific triggers are tripped. They could dynamically tune the parameters controlling how their data persists or is replicated (latency, throughput, number of copies, overall cost, encryption keys, trust in cloud providers, optimism vs transaction guarantees, etc.)

What more, each computation needs a controller meta-computation to manage when it is migrated to what new implementation. This controller can also serve to virtualize declarative I/O. Programs can then be factored in ways that split functionality between program and metaprogram, rather than between program and library. The important distinction here is that when using a library a program includes all its semantics, whereas when relying on a metaprogram, their semantics remain well separated, making for simpler reasoning and safer, more secure programs.

1.1.4 Reconciling Semantics and Reflection

The above runtime abilities are the promise of this thesis. Our approach to achieve them is to reconcile two subfields of Computer Science often considered antithetical: Semantics and Reflection. Indeed, a general belief is that when you use Reflection, anything goes: all language invariants can be subverted, any reasoning is voided, the logical theory becomes trivial, and study of semantics is vain; therefore, for the semantics of a language to be well-defined, it must avoid any reflective features. Conversely, those who use reflection often feel that semantics can only limit them, when they seek to break out of the limits to what they can express; therefore, their efforts at exploring semantics usually stop at explaining how their systems are or could be implemented, with no regards for semantics or invariants.

Yet, this thesis proposes a general model for reflection that is resolutely rooted in semantics. The first part is dedicated purely to the semantics of implementations. The second part builds on the first to extract a runtime protocol that allows for reflection. The third part explores what semantically meaningful operations are enabled or facilitated by this protocol. The fourth part describes a reflective architecture to take advantage of the protocol.

But beyond what this thesis proposes, the hope is to reconcile the two subfields of semantics and reflection: they need not be alien to each other; and they must cease to be, for both are required in large enough systems.

1.2 Reflection

1.2.1 What it is

This thesis is *about* Reflection in computational systems.

Informally, Reflection is the phenomenon by which some formal systems can reason or compute *about themselves* — for some meaning of “themselves”. The metaphor underlying the name is that the system (or an agent in the system) can look at “itself” as if in a mirror. More formally, Reflection is the use of self-reference, fixed points, or “strange loops” in formal systems, allowing for meta-reasoning or meta-computing about “themselves” — and often about any other (logical or computational) formal system, in a universal way. We may speak of Reflection when discussing of meta-reasoning or meta-computing in a universal system, even when no self-reference or loop has been introduced (yet) besides the loop implicit in using formal systems to reason about other formal systems.

Note that formal systems encompass both logical systems and computational systems; the distinction between the two can often be soft and blurry, a matter more of point of view and intent from outside the system than of specific properties of the system itself: not only does the Curry-Howard correspondence establish an isomorphism between proofs and programs; computer programs routinely implement proof and validation in logical systems, and logical systems are used to reason about computer programs.

A universal meta-computing system, is thus one where programs and programmers can see code as data, and data as code. A little bit of theory[25] shows that in any sufficiently expressive *base* computing system (where any system capable of expressing simple arithmetics suffices), it is always possible to go from data to code at the cost of writing an interpreter for the data, and to go from code to data at the cost of doing everything in said interpreter. Any system capable of expressing simple arithmetic functions can therefore be seen as a universal *meta* computing system. However, the approach consisting in using such an interpreter and using only it can be extremely costly, and reflection is much better done when the programming language supports it natively: not only is the implementation of reflection much faster then, but it is also less fragile as the programming language evolves.

1.2.2 Some History

Reflection is the instrumental device relied upon by the the very foundational works of Kurt Gödel[10] and Alan Turing[25], that started both modern logic and modern computer science in the 1930s. In the 1960s, John McCarthy's LISP programming language[20] led to the discovery of simple and cheap ways of manipulating and evaluating program representations. Brian Cantwell Smith's 1982 thesis[24] made explicit the thinking *about* reflection in Programming Languages.

These days, not only Lisp dialects, but all major Programming Language platforms feature *some* support for reflection at both runtime and compile-time: not just "scripting languages" based on interpreters for which it is relatively cheap to "open" the internals to enable reflective features; not only popular compiler platforms like C++, .NET, or Java, that couldn't escape the practical necessity of such features; but even languages with "formal" background like ML or Haskell, whose authors may have initially tried to denigrate and avoid reflection as a dangerous source of paradoxes (that just like Bertrand Russell a century earlier[citation needed] they tried to exclude using types).

1.2.3 Going Meta

The prefix traditionally used to indicate the use of reflection is *meta*: thus, when a formal system is used to reason or compute about another, it is called the *meta* system, and the other one, in contrast is called the *base* system. In illustrative diagrams, the *meta* system is often pictured *above* the *base* system, according to the metaphor of a God or superior being overseeing his creatures and interfering as needs be.

Now, etymologically, *meta* is a Greek word that means "after", "above" or "beyond". But in the XXth century, it has come to be widely used among scientists and poets in English and other languages as a prefix to mean something different: discourse *about* a topic.

This use has its origin with "Metaphysics"[citation needed], a collection of books by Aristotle, that when his complete works were first compiled, were published *after* his treatise on nature — "Physics" in Greek. The relation of Metaphysics to Physics was thus that of a textual sequence: one volume came after the other. The name had not been given by Aristotle himself, but by whoever compiled his complete works, and failed to imagine a better name to fit the unusual topics in that volume. Indeed, in "Metaphysics", Aristotle discussed various topics of philosophy that didn't fit in any known, named domain: he started with what he called *first philosophy*, which in modern terms we might call *the foundations of philosophy*, principles on which the rest of philosophy rests, and went on discussing the nature of the sciences, of reality, of "being" itself.

Now for people who studied Philosophy, and who kept reading "Metaphysics" (in translated versions) long after anyone interested in nature had stopped reading "Physics", and who weren't fluent in Greek, the prefix "meta" was inferred as denoting a semantic relation between "Metaphysics" and "Physics": if Physics was the Science that studies Nature, then Metaphysics was the Science that studies the Science of Nature. In the early XXth century, the Science that studies Mathematics and its foundations, itself using a lot of Mathematics, was dubbed Metamathematics. Since then, meta-`X` came to be used to mean anything science or activity *about* `X`, especially (but not only) when it involves a lot of `X`, whatever `X` may be. A metalogical variable is a logical variable in a model of logic; a meta-discussion is a discussion about a discussion, etc. This use was made famous by Douglas Hofstadter's works *Gödel, Escher, Bach* and *Metamagical Themas*[citation needed], and by computer scientists who deal with such situations all the time.

1.2.4 Representation

An entity m in a meta system may *represent* a first-class entity or second-class phenomenon b in the base system, such that based on which logical statements can be made and proven (or disproven) about the entity or phenomena in the base system, and transformations may be computed.

The verb "represent" and noun "representation" have a clear traditional usage, that long predates Smith's thesis, and continues beyond, that may set the pattern for other words.

Here, *first-class* means that an entity is embodied as a value that a system can itself compute over or reason about, and bind a variable to if applicable; phenomena that are not first-class are called *second-class* and typically include types, environments, continuations, and other contextual data that is necessary or useful to specify the behavior of the system or reason about it, yet not embodied in it.

The noun *representation* refers to a notional function from the base system, or an extension thereof that include its second-class phenomena, to first-class values in the meta system. When the context makes it clear and unambiguous, a particular meta-level entity that represents a particular base-level entity or phenomenon is itself called a representation, while the base-level entity or phenomenon being represented is just called entity, object, type, environment, continuation, context, or whatever noun describes its embodied or disembodied kind of notion.

A representation typically includes more details, aspects, distinctions, than matter to the phenomenon being represented. For instance, the representation of functions and variables may be or have labels or names as a hook to manipulate or display these entities, whereas the corresponding base-level entities possess no such attribute. We say of these extra aspects, distinctions, etc., that they are *syntactic*, whereas those that matter in the base-level entity are *semantic*. The representation may thus involve *purely syntactic* entities such as labels and names, paths and line numbers, that have no meaning at the base-level (if a total mapping is required, they can be forgetfully mapped to the same element of a unit type). A representation that maps much of the semantics of the base system directly to semantic elements of the meta system implicitly processed by the meta system semantics without introducing many syntactic entities or aspects is said to be *shallow*; on the other hand, a representation that transforms some base system elements into syntactic entities that are to be explicitly processed by outside functions is said to be *deep*. The shallowest representation is of the system by itself, making no processing step explicit; the deepest representation is of the system in terms of data structures, making all processing steps explicit.

The opposite of representation is *meaning* or *denotation*. The base-level phenomena b is the meaning denoted by the meta-level representation m. The meaning or denotation function goes from meta-level first-class value to base-level phenomena (first-class or second-class). Note however that as for the verb form, we still say that m denotes b, that m means b, rather than the opposite. The grammatical relationship between the verb and noun forms of these correspondances between base and meta level therefore does not vary uniformly. We also sometimes say even that m evaluates to b, when b is a first-class value and no further context is necessary to go from representation to meaning.

1.2.5 Reification and Reflection

Reflection is the phenomenon of self-reference within a system of the system itself, that then plays the role both of meta system and base system. There can be reflection without explicit representation, as Brian Cantwell Smith insists in his seminal thesis[24]: a system may behave in ways that refer to its own behavior, without involving a representation of that behavior as well-defined first-class "symbolic" entity that is explicitly manipulated by the system. However, most of the research of reflection, including the present thesis as well as Cantwell Smith's involves an explicit representation.

As far as nomenclature goes, Cantwell Smith says that "the system (as a meta system) reflects about itself (as a base system)" (parenthetical remarks ours), and that a program may "*reflect* and thereby obtain fully articulated "descriptions" [...] of the state of interpretation process that was in effect at the moment of reflection". The verb's subject is the system as meta system, and it has itself as base system as implicit direct or indirect object. Furthermore his thesis is written in the context of a *procedural tradition* wherein computation happens through procedures that side-effect a *model*: the model *reifies* the state of the system as base system; modifications to that model within the meta system and to the underlying base system are then "causally connected". Cantwell Smith does not exactly have a word for the opposite of *reify*, but the verb *absorb* is used for that (and also for somewhat different uses). Further, Cantwell Smith layers each layer of meta computation notionally *above* the base layer that it reflects — a convention followed by much of subsequent literature[citation needed].

Wand and Friedman [8, 28] in their efforts to clarify Cantwell Smith's work, define the verbs *reify* and *reflect* as inverses: A program may *reify* the state of its base-level computation, including second-class information such as environment and continuation, and pass that state as a first-class value to some function run at its meta-level (which is notionally *up* in this classic literature). *Reification* is that process that takes a base-level phenomenon and turns it into a meta-level value. Conversely, a program may take such first-class values as result from reification and other meta-level computations, and *reflect* those values into a base-level computation (notionally one step *down* in this classic literature). *Reflection* is that process that takes a meta-level value and turns it into a base-level phenomenon.

1.2.6 Directional Confusion

Now, which is reification and which is reflection is neither obvious nor context-independent, and the same pair of functions can sometimes be viewed both ways.

Consider the popular Haskell library `Data.Reflection` written by Edward Kmett, following Kiselyov and Shan[15]. Kiselyov and Shan seem to themselves follow the Wand and Friedman nomenclature for reify and reflect (though they do not cite them), with the point of view that type-level programming is some compile-time meta-level for generating base-level code that will be evaluated at runtime. Their reflect function goes from types to runtime values, whereas their reify functions go from runtime values to types (using some clever continuation encoding to remain expressible in Haskell). Examples provided include a compile-time type-level representation of integers and runtime integers, or similarly for lists of integers, or for any Storable value.

Now, another use of reflection[citation needed] may have several meta-level each add increasingly concrete implementation details to previous more abstract computations. In such points of view, each meta-level is more concrete, each base-level more abstract, and types are even more abstract. Reification then goes toward the concrete and reflection toward the abstract, in directions opposite to the Haskell library above.

Furthermore, a reflective optimizer that would take a type-level representation of integers and try to extract an actual meta-level value of an integer, so it can inline that value in various operations, would consider the type as a second-class base-level phenomenon to be reified into a first-value integer value at its meta-level. Reification and reflection would then also go the opposite direction than the Haskell library above.

Finally, "reification" etymologically means turning a sometimes vague concept into a actual thing, and it isn't always obvious whether the meta-level abstractions even when first-class values are more "things" than the concrete base-level phenomena even when second-class. Meanwhile, "reflection" through a mirror suggests that a system is looking at itself, and that the base-level is the "thing" and the meta-level representation a model, yet the way we use the concept is more generally useful in any stratified situation where a meta system looks at a base system that is well distinguished and not "itself" but other.

To resolve confusion, we will therefore eschew the use of the terms "reification" and "reflection", and instead introduce and use unambiguous terms that work wherever a "meta" system and a "base" system are involved, with or without "self-reference", and instead clearly distinguishing between three different kinds of situations.

1.3 Three Dimensions of Meta-Computation

For the purposes of this thesis, we will distinguish three very different kinds of meta-computations, that don't seem to have been explicitly distinguished in existing literature. This distinction will clarify the relationship between software elements when many programmers might otherwise be confused when "reflection" or "meta" are involved.

1.3.1 Generation

Our first dimension of "meta" in software is when a program, the metaprogram, manipulates the code of another program, the base program, or group of programs: reading, injesting, analyzing source code into internal representations, modifying and transforming those representations, and often in the end generating new code, either more "source" code in the same language or a different one, or "object" code in a usually lower-level language closer to execution by the hardware.

The prototypical such metaprograms are compilers, that transform source code into object code. Lisp macros, C preprocessor macros, C++ templates, Haskell type-level programming, all kinds of code generators, code reformatters, transpilers, preprocessors and expanders, are more such metaprograms. Interpreters, that take a source program as input and evaluate it, are also such metaprograms: they process a program and directly generate its effects and output without generating a user-visible object program as an intermediate step, though many generate one internally, representing the program as an abstract syntax tree (AST) for "naive" interpreters, or as a somewhat more elaborate bytecode for more advanced interpreters.

The important way that the metaprogram is "meta" is that it runs *before* the base program it processes, in an earlier stage within a sequence or more generally hierarchy of staged evaluations.[1] This is ironic, since in Greek, meta means *after*, which is the opposite of before. To describe this phenomenon, I thus propose that the (Latin) prefixes *ante* (before) and *post* (after) be used.[2]

Thus, we will call *anteprogram* the metaprogram that generates code, and *postprogram* the code that is generated, the evaluation of which comes later. When the metaprogram analyzes code of the base program rather than generates it, we will still say *anteprogram* for the metaprogram and *postprogram* for the base program, considering that the metaprogram is still in this same evaluation space that exists before that of the base program.

We will call *generation* the process of computing a *postprogram* from an *anteprogram*. Conversely we will call *quotation* the process of computing an *anteprogram* that (trivially) generates a given *postprogram*.[3] In the common case that evaluating an anteprogram outputs a postprogram, we will say that the anteprogram *generates* the postprogram, and that the postprogram is generated by the anteprogram. To obtain an anteprogram that generates the postprogram, we will quote it.

This dimension of metaprogramming we will call *generative*.[4]

We will not discuss this dimension much in the following thesis, not because it is insignificant, but because there is already a quite large and very active body of literature on the topic, from compilers to macros and everything else. This dimension of metaprogramming is well-understood and we do not have much in terms of new lights to provide on it. Thus, we will readily assume all affordances of the modern generative metaprogramming, use them in the system we build, and make them available to programmers using it, even though most programming environments only allow limited forms of such metaprogramming to regular programmers, and most programmers fail to use any.

[1] Though for a very trivial and thin notion of "before" in the case of an naive interpreter.

[2] I strongly rejected the pair *prin/meta* or *pro/meta* (Greek for before/after) as prefixes for this dimension of metaprogramming, because of how the usage for *meta* conflicted with common usage for metaprogramming, which would only increase confusion. I also rejected *proto* (Greek for "first", a prefix commonly used in such situation) because the closest opposite would be *deutero* (Greek for "second"), which is not very well-known, and causes cognitive dissonance in a sequence of more than one metaprogramming stages. I rejected *pre/post* because preprogramming sounds too close to preprocessing which is related but more restricted, and *pre* is also too close to *prae* which was a candidate for another dimension; so all in all more confusions to have to explain away. I rejected *paleo/neo* (Greek for ancient/new) that refer to evolution in time, *argo/teleo* (Greek for start/end) that are too unfamiliar and have absolute connotations that don't resonate nicely with composition of metaprograms, *ana/kata* (Greek for up/down) that correspond more to our second dimension of metaprogramming below, *geno/pheno* (Greek for producing/showing) that do match the biological analogy with genotype and phenotype but would require more explaining, *gene/here* (Latin for genitor and heir) or some variant of *goneo*, *matro*, *patro* / *paedo*, *pedo*, *tekno*, *oo* (Greek for parent, mother, father / child, child, son, egg) also for requiring too much explanation, *dino/lamvano* (Greek for give/receive) for being too obscure, and *stoma/procto* (Greek for mouth/anus) that nicely clarify the flow of information but induce a disturbing scatological metaphor when composing metaprogramming stages.

[3] The usual notion of quotation, as per the Lisp special form `quote` is a syntactic process that works at compile-time on what is already a meta-level representation of a base program to yield a meta-meta-level representation of a meta-program generating that base-program. A corresponding reification function that works at runtime on a base-level value to yield a meta-level representation is called `kwote` is old Lisp lore, so if we wanted to stick closely to established usage, we might have to call *kwotation* the process of computing an anteprogram from a postprogram.

[4] I rejected "evaluative" or "temporal" which fit somehow but cause too much confusion with related but different concepts in computer science, or "horizontal" which works well with the second dimension below but not when taking the third into account.

1.3.2 Implementation

Our second dimension of "meta" in software is when a program, the metaprogram, precisely implements at runtime the semantics of another program, the base program. The metaprogram may handle all kinds of additional lower-level implementation details, such as the allocation of various base-level objects and variables into memory locations, or the scheduling of underlying threads of execution, or the mapping between language-level entities and database storage, etc. These behaviors are not aspects of the base program that users care about, and are not even expressible at the level of abstraction at which users interact with the base program. Yet they are necessary for the efficient evaluation of the base program on the underlying hardware.

The important way that such a metaprogram is "meta" is that it runs *below* the level of abstraction of the base program above it. This is ironic, since people often picture a metaprogram as being "superior" and "above" the base program, due to it having more expressive power, and one of the meanings of the Greek word "meta" is indeed "above". To describe this phenomenon, I propose that the (Greek) prefixes *hypo* (below) and *hyper* (above) be used.[5]

We call *interpretation* the process of taking a low-level (state of the) *hypoprogram* and computing the high-level (state of the) *hyperprogram* that it *implements*, *represents*, *denotes*, *means*, or otherwise corresponds to. As Cantwell Smith notes[24], this is well-established meaning of the word *interpretation* though one different from the also well-established meaning used previously when discussing interpreters as anteprograms. The two meanings while related are different enough in different enough contexts not to be the occasion for much confusion. In this thesis we mean mostly this semantic denotation rather than the previous runtime execution, unless explicitly specified otherwise.[6] We say that a hypoprogram *implements* a hyperprogram, and that the hyperprogram is implemented by the hypoprogram, or is the interpretation of the hypoprogram, or that we can *interpret* (a state of) the hypoprogram as having for meaning (a corresponding state of) the hyperprogram.

Conversely, we call *implementation* the process of taking a high-level (state of the) *hyperprogram* and computing a corresponding low-level (state of the) *hypoprogram*. As we will see, this also matches the common use and understanding of the term "implementation".

This dimension of metaprogramming we will call *implementative*.[7] Note how for this dimension we choose the word for the "reification" process going from base to meta, because "implementation" more precisely suggests what this dimension is about than "interpretation" that while correct leaves too much to interpretation and imagination. This contravariance will reappear many times in this thesis.

[5] I would have prefered be consistent in using either all Latin or all Greek prefixes, but the common use of meta conflicts with Greek in the first case. The somewhat common use of the prefix *sub* to denote temporal inclusion in a subprogram or subroutine precludes the choice of the latin *super* and *sub* as prefixes. I could have proposed *ana* (up) and *cata* (down) but I figure that *hypo* and *hyper* are more familiar prefixes, and there is also an existing use of *ana* and *cata* in category theory, that is unrelated, whereas *hypo* and *hyper* seem to be up for grabs. I could have proposed the latin prefixes *supra* (above) and *infra* (below), and that could have worked well with the notion of infrastructure, but I figured the greek prefixes sound better with fewer opportunities for confusion. Other prefixes were rejected for requiring too much explanation, including *rhizo*/*clado* (Greek for roots/branches) which actually work quite well in many ways including the foundational and branching aspects, or *steato*/*ligno* (Greek for fat/slim) which alludes to the increasing amount of details in implementations, or *symbolo*/*hermeneu* (Greek for symbol/explanation) which is literally true but too obscure.

[6] As we'll see, this meaning is also related to *abstract interpretation*, though the latter usually denotes compile-time analyses from the source program without runtime information to lossy approximations thereof above it in terms of types and such, rather than runtime correspondances from a detailed level below that of the source program to the level of semantics the source program (plus other runtime information such as environment, continuation, etc.).

[7] I rejected "interpretive", "abstractive", or "semantic", which fit somehow but cause too much confusion with related but different concepts in computer science, or "vertical" which works well with the first dimension above but not when taking the third into account.

While countless publications touch the topic of Implementation, none seems to have a general theory of what Implementation is or should be. The first two parts of this thesis will be dedicated to exploring a universal theory of this second dimension of metaprogramming, *Implementation*.

1.3.3 Control

Our third dimension of "meta" in software is when a program, the metaprogram, surveys and controls at runtime the behavior of another program, the base program. Many *Meta-Object Protocols* and *Meta-Level Architectures* have been proposed in the past[citation needed] and even though these terms haven't been en vogue since the early 1990s, today's software is ripe with many such kinds of controlling meta-objects, though under other names, without a unified architecture, and with less dynamic control than in the old language-based meta-object systems: graphical control panels, configuration or settings menu, language servers, interactive debuggers, tracers, loggers, monitors, consoles, virtualization managers, user-level filesystems, data buses, middleware, etc. Configuration files and command-line options that are checked at startup by the program itself can be considered a degenerate case where a notionally distinct one-shot controller metaprogram was transcluded in the program itself, for lack of better infrastructure to achieve the same purpose within the context of distributing "executable files" to the end-user on current operating systems.

To describe this phenomenon, I propose that the (English) prefixes *fore* (front) and *back* (back) be used. If the fore-program is what occupies the front stage that users want to interact with, the back-program is all the supporting work that happens in the back stage that most users do not usually want to have to deal with, yet that pull the invisible threads that control the puppets that the users do interact with.[8]

We will call *control* the association from a *foreprogram* (base program) to the *backprogram* (metaprogram) that controls it. Conversely, we will call *manifestation* the correspondance from a backprogram to the foreprogram that it controls. We will say that a backprogram *controls* a foreprogram, and that the foreprogram *manifests* the backprogram.

This dimension of metaprogramming we will call *controlling*.[9]

While many publications describe specific control gadgets, the topic of a general architecture for runtime control mechanisms seems largely abandoned since the demise in the mid-1990s of various object systems sporting meta-object protocols; and even these systems were isolated general-purpose but system-specific experiments that didn't attempt a universal theory of Runtime Control. The last two parts of this thesis will be dedicated to exploring a universal theory of this third dimension of metaprogramming, *Control*.

1.3.4 Three Independent Dimensions

Generation, Implementation and Control are three independent dimensions that one may go "meta" about a program: One may go "meta" by going "ante" and neither "hypo" nor "back"; One may go both "ante" and "hypo" but not "back"; etc.

[8] I rejected the *piso* / *empro* (Greek for back and fore directions), *plati* / *prosopo* (Greek for a body's back and face) or as wholly unknown and unsuggestive to English speakers. I also rejected *retro* / *prae* (Latin for back and fore directions), and *para* / *pro* (Greek prefixes for the back and fore of a scene in theater), because these prefixes have conflicting connotations as otherwise used in English. Finally, I rejected *inner* / *outer* or the Greek *endo* / *exo* as suggesting too much of a static, essential, irreplaceable relationship between the two, when the relationship between backprogram and foreprogram is intended as very dynamic in general; maybe they could be used if we ever need a name for a static variant of control.

[9] I rejected "contextual" to qualify this dimension. There was no proper name for a dimension to complement "horizontal" and "vertical" and "depth" didn't quite work.

One may also combine several steps of “ante” and “post” with a different number of steps of “hypo” and “hyper” and of “back” and “fore”, and build three dimensional circuits along all three axes. When considering such transformations, it might not even make sense to distinguish a particular program as “meta” rather than “base”: that status only makes sense relationship to another program in the given transformation circuit, and changes in a different relationship.

We will be careful to specify precisely what we mean when we speak of “metaprograms” and “base programs”. Actually, in the rest of this thesis we will avoid the terms “meta” and “base” and instead systematically specify “ante”, “post”, “hypo”, “hyper”, “back” and “fore” as appropriate. We will similarly avoid the nouns “reflection” and “reification” to use the nouns “generation” and “quotation”, “interpretation” and “implementation” or “control” and “manifestation” as appropriate. Similarly we will avoid the verbs “reflect” and “reify” to use verbs “generate” and “quote”, “interpret” and “implement”, or “control” and “manifest” as appropriate.

We will represent the first dimension of generation horizontally, with *ante* to *post* going from left to right, as is common when representing stages of evaluation in time.

We will represent the second dimension of implementation vertically, with *hypo* to *hyper* going from bottom to top, as matches the notion that concrete implementations with more details are lower-level than more abstract interpretations with fewer details that are higher-level.

And we will represent the third dimension of control as notional depth, with *back* being distant from the reader and *fore* closer to him, as matches the notion that the backstage is hidden from spectators behind the front stage.

Note that beside common convention, there is a good reason why the relationship between “meta” and “base” system indeed goes in the direction we specified and not the other way around: in each case, the “base” system is closer to what the end-user sees, it expresses fewer details that “reflection” from “meta” to “base” forgets and that the “reification” from “base” to “meta” has to reconstitute often involving some arbitrary choices or programming context that the end-user never sees nor cares about.

Also note that these three axes do not precisely match any of the notions of “computational”, “procedural”, “structural” or “behavioral” reflection sometimes seen in the literature[citation needed]. We couldn’t see a definition of these notions that multiple authors agree on, nor a formalization that would give them an objective meaning.

Finally note that since we introduce in this thesis the concept of these three dimensions of “meta”, other works in the field of reflection should not be expected to use a compatible nomenclature at all. We will consistently use the nomenclature in this thesis. The above definitions are only authoritative in the current thesis, and while we will take care to use them consistently, readers should be careful when matching our terms with the same terms or different terms as used by other authors.

1.3.5 Metaprogramming as a Social Activity

Now of course, the most common case for metaprogramming is syntactic abstraction: a higher-level language, usually a Domain-Specific Language (DSL) or a syntactic extension to an existing language, is offered to users to express their concerns as a *hyper*-program at a suitably high level of abstraction; simultaneously, an *ante*-program defines how this *hyper*-program is to be translated into a *hypo*-program in the lower-level language. The resulting *post*-program is what will actually be run by the end-users. Furthermore, some effects can be factored out into *back*-programs to keep the *fore*-programs simpler but also easier to distribute and upgrade.

Note how to every *ante* there is a *post* and to every *hypo* there is a *hyper*. In the end, metaprogramming is not about “moving” programs in one direction so much as *factoring* program development, along three dimensions: Without metaprogramming, these dimensions are often left implicit, confused or interspersed together, for instance using unsafe and inefficient naive interpreters, often fraught with dangerous inadvertent puns. Using metaprogramming, the three dimensions can be clearly distinguished, labor can be divided along well-defined surfaces, and it becomes possible to reason about code while abstracting away each side from the other.

Metaprogramming therefore has an important social role: it enables more efficient *division of labor*, and thereby more affordable production of more robust and more elaborate programs. Its division of labor enables better specialization of tasks in an otherwise increasingly complex world. It allows each developer to cultivate and leverage their comparative advantage, whether at understanding a domain in which to write or reason about domain-specific hyperprograms, or at growing expertise about anteprograms that can more efficiently generate hypoprograms for those hyperprograms, or at creating or managing backprograms that alleviate the need for other programmers to manage some aspects of computations so they can focus on simpler foreprograms.

1.3.6 The Difficult Direction

The “reflecting” direction of meta-computation, whether it means “generating”, “interpreting” or “manifesting”, is usually well-defined, straightforward, possible and easy in any existing computing system: the processors that do it are readily available without any special care given to implement “reflection”. Your existing compiler, your existing runtime, your existing configuration manager, already support it.

Moreover, many manuals, tutorials, books, articles, essays, explain how to write your own compiler for your language, your own program execution runtime, or to a point your own virtualization or configuration engine — all in ways that will be fully compatible and interoperable with the system-provided equivalent, without having to resort to any magic trick or unsupported system feature.

However, “reifying” is the operation that is seldom supported, or only in very limited ways. It is a lot of work to quote a program, with no common convention for program representation, no common library to draw upon to manipulate programs except invoking the usual compiler the hard way on the entire program. The implementation of a program state is usually completely opaque, with unreliable ways to inspect and modify the low-level state that void any implementation warranty when used. And the notion of controlling meta-object is mostly absent from common language design and operating system architecture, except for very ad hoc tools.

Moreover, even if you implement this “reifying” the hard way, which you will mostly have to do without documentation on how to do it, your code will in no way interoperate with system-provided tools, unless you spend a lot of time reverse-engineering how things are done, and even then the result will be fragile as the system evolves. You will have to reimplement the entire system (possibly as a “virtual” variant on top of the existing system), or learn and embrace the internals of the current system (if published and accessible at all) to achieve “reification” in a way such that users can modularly reify small bits of programs without each having to reify the entire system.

The challenge in designing a “reflective” architecture will thus be to all for *modular* reification of code — quotation, implementation, control — in ways that can be modularly reflected back — generated, interpreted, manifested — such that developers can safely divide labor in smaller components without each having to reimplement a complete reflective system.

1.4 Our Contribution

1.4.1 Our Approach

Our approach differs from previous presentations of Reflection in literature, in at least following ways:

- We put an emphasis on formal semantics, where previous works remained largely informal or semi-formal, at best formally describing a single reflective system.
- We explicitly distinguish those three dimensions of metaprogramming above, considered independently from any the notion of self-reference.
- We seek a modular architecture that enables division of labor along finer component divisions thanks to these three dimensions of metaprogramming.
- Our formalism is not aimed at describing a single system that would happen to be reflective; instead, it describes a universal framework for metaprogramming, that applies to all systems.
- Most of the systems we describe using our formalism are not "reflective" as such, as they involve no self-reference at any point. However the overall system we describe is itself necessarily reflective as a consequence of being universal.
- Reflection thus naturally emerges as we seek fixed points, which naturally appear when building general meta systems that reason about other systems (but neither more nor less so than any meta system).
- We apply the notions of reflection to modules of computations, well delimited along each of the axes of metaprogramming: a computation after (post) a certain point, above (hyper) a certain level, in front (fore) of given scene. We do not specifically seek notions of reflection that apply to "the" computation (all-encompassing without reified context), to individual objects (too small-grained for our approach), or anything else in between.
- As part of our categorical approach, our formalization of reification notably captures not just states but also transitions between states, with their side-effects.

1.4.2 Our Plan

In the first part of our thesis, we present a general-purpose formal theory to describe the operational semantics of various computing systems, and relations of implementation between them. This theory as such is largely but a restatement of well-known techniques; however, it allows us to introduce the key concept of *Observability*, thanks to which the abstract state of a system can be recovered from the concrete state of its implementation.

In the second part of our thesis, we present a protocol to describe and manipulate implementations as *first-class* entities at runtime. This protocol can be seen as the computational content that can be extracted from the previous logical theory. We discuss how to reinterpret compilation, static type analysis or aspect-oriented programming in the context of this protocol.

In the third part of our thesis, we present potential applications of this protocol, and how it can be used to reinterpret many existing techniques, and reimplement them in simpler ways. For instance, a general concept of *migration* subsumes process migration, garbage collection, zero copy routing, dynamic configuration or JIT compilation. It makes it possible to modify at runtime the semantic tower that implements a program. Other applications include the modelling of fault-tolerance, optimistic evaluation.

In the fourth part of our thesis, we propose a runtime meta-programming architecture based on the previous protocol, wherein every program has an associated semantic tower at runtime, and an associated controller meta-program. This architecture may apply to any kind of development platform, user interface shell, operating system, or distributed and virtualized application manager. It may be considered as a reenactment in a more formal setting of the reflective towers from the 1980s.

Part I

The Semantics of Implementations

Chapter 2

Formalizing Computations

2.1 Categorical Approach

2.1.1 Intent

We provide a general framework to represent arbitrary computations as categories, and implementations as relations between categories (category theorists say "profunctors").

By "computation", we mean the process of running any kind of a program in any programming language, any Turing machine (universal or not), any expression in a variant of λ-calculus or π-calculus or any other calculus, any bytecode running on a virtual machine, any binary code running on digital devices, or even any configuration running on substrates other than electronic devices, such as mechanical devices, analog devices, quantum devices or biological devices.

We identify a computation with its operational semantics: a category whose objects (nodes) are the possible states of the system, and whose morphisms (arrows) are valid labelled transitions between states of the computation.

So as to avoid confusion with other meanings of the word "object" in computer science, we will herein mostly use the layman terms "node" and "arrow" instead of the terms more commonly used by category theorists, "object" and "morphism" (or "homomorphism"). We still call "functor" an application from one category to another that preserves the composition of arrows.

We require no prior knowledge of category theory, and remain elementary throughout.

2.1.2 Computations as Categories

When viewing computations as categories, nodes express the internal state of the system; arrows between two nodes express computation paths that lead from one state to the other. Side-effects and other interactions with the external world (if any) are encoded in these arrows, and distinct arrows will represent distinct interactions, whereas indistinguishable interactions will be represented by identical arrows.

At the level of abstraction offered by typical computer hardware, computations constitute a category where a node is a record of the state of each CPU register, memory bank, or peripheral; arrows are transitions between states, with I/O. More abstract systems could be described in terms of a virtual machine with a data heap, a control stack, and program code; or in terms of environment, store, and continuation; or in terms of whatever concepts suit the system at hand.

We will use diagrams such as the following one to describe assertions about operational semantics. For instance, in the following diagram, we describe three nodes x, y and z in a category S:

$$\begin{tikzcd} x \arrow[r, "S"'] & y \arrow[r, "f", "S_e"'] & z \end{tikzcd}$$

There is an unnamed transition from x to y with an arrow in S, and a transition from y to z, where the arrow is member of a subset S_e of S, and has an effect f: presumably, S_e is a subset of S that allows certain kinds of side-effects, and the label f represents one of these side-effects. For instance, when printing `"Hello, World"` on some virtual machine, you may start from an initial state x, push the string `"Hello, World"` to the stack as you transition to state y, then pop that string from the stack and print it as a side-effect, as you transition to final state z.

2.1.3 The Category of Computations

The very purpose of our categorical approach is to describe structure-preserving correspondence between completely different systems, the states of which may be described in completely different ways.

A structure preserving correspondence Φ from a computation C to a computation A will be called an *interpretation* of C as A, while the inverse correspondence will be called an *implementation* of A with C.

Given such an interpretation or implementation, we will say that C is the concrete computation, or that it is low-level and A is the abstract computation, or that it is high-level. In the context of several implementations that may be composed, we may also say that C is more concrete or lower-level than A, or that A is more abstract or higher-level than C. We will draw a diagram with C at the bottom and A on top as follows:

$$\begin{tikzcd} A \\ C \arrow[u, dashed, "\Phi"] \end{tikzcd}$$

The reason the line is dashed is because these correspondences are *partial* functions, rather than total functions. In other words, an interpretation of C as A is a total function from a *subset* of C to A. See next chapter 3 for details.

Computations and interpretations are themselves the nodes and arrows of a larger category, the *category of computations*. Implementations are the arrows of the *opposite*, dual, category. Viewed as functions, they would in general be non-deterministic partial functions (i.e. relations), mapping abstract computations to the many concrete computations that may implement them.

2.1.4 Internal and External State

For some formal semantics to properly capture the meaning of a computation, they must somehow represent any relevant interactions with other systems. Depending on what one cares about, these interactions may include or not include such things as external data inputs and outputs, other side-effects such as printing, manufacturing a physical item, or otherwise controlling a robot, or even timing, power consumption, and any other resource usage.

The formalization these interactions can be done in two ways. One approach is to *internalize* these interactions into the category, by encoding that which is being modified (messages received or sent, resources used, etc.) as part of the internal state represented by a node. The opposite approach is to *externalize* such state by collapsing together nodes that only differ in terms of such interactions, and only distinguishing these interactions (if at all) through the arrows that represent distinct interactions. From a categorical point of view, internalizing interactions is more concrete: and there is an obvious forgetful functor from the more concrete view with internal state to the more abstract view with external side-effects only.

When modelling interactions, it is sometimes useful to consider a separate category, called the *action category*, the arrows of which represent all the possible interactions that one may care about. This action category often has only one node (it is a *monoid*); but it can also have several nodes, each representing a distinguishable "mode" in which the system might operate, that might change during computation.

A computation is then specified as a category of states and state transitions, as well as a functor from said category to an action category, called a *labelling* functor. The action associated to a state transition is also called its *label*. The actions represent the visible interactions observable from the outside, whereas the state transitions represent the internal state of the computation, that might not be directly visible, yet that drives the visible interactions.

Some action categories might themselves have more details than other action categories and be more concrete, or have fewer details and be more abstract. When considering an interpretation of a computation as another one, we will require that they be accompanied by a corresponding interpretation of the concrete computation's actions as the abstract computation's actions, that makes the obvious diagram commute. *(Draw this diagram)* For such purposes, we can identify a category without action category to a category whose action category is itself in a trivial way.

2.1.5 Russell-Whitehead, not Curry-Howard

Note that our interpretation of computations as categories is distinct from the Curry-Howard correspondence, and possibly the opposite.

In the recently very popular and wildly successful Curry-Howard correspondence$^{\text{[citation needed]}}$, a node in a category is a type of computational objects or equivalently the logical proposition stating that the type is not empty; and an arrow is a total function from one type to another or equivalently an implication proof of one proposition given the other. This is a very powerful paradigm with a lot of applications.

Our approach, however, relies on a different correspondence, whereby a node in a category is a state of a computation, or equivalently the proposition stating that some computation can reach a valid end state; and an arrow is a trace that shows how one state can lead to the other. Since Bertrand Russell's and Alfred North Whitehead's foundational *Principia Mathematica*$^{\text{[citation needed]}}$ can be seen as using this approach to equating a proof to a trace of what can be obviously seen as a non-deterministic computation (though they did not call it that, for their work pre-dates computer science), we will call this the Russell-Whitehead correspondence.

The two correspondences have a very similar flavor and are obviously related, but it is not immediately clear how. We suspect that they can both be expressed as views of the same phenomenon in Game Semantics[13]: In Game Semantics, each proposition describes a game between two opponents, who respectively try to prove and disprove the proposition: a "run" of the game determine who wins the argument; and the proposition is true if there is a winning strategy using which the player arguing the positive may always win the argument whatever moves his opponent chooses. It seems to us that by squinting hard enough, we can view the strategies grouped with the runs as programs, and naked propositions as types, whereas strategies grouped with the propositions can be viewed as programs, and naked runs as traces. However, this intuition may have to be formalized before it is either accepted or rejected.

2.2 Existing Formalisms as Categories

Category Theory provides a notion of computation that is simple and general; it makes it possible to unify, relate, combine and generally use together computations originally described with different formalisms. Here are some familiar paradigms and how they map into our presentation.

2.2.1 Labelled transition systems

Our categories directly include labelled state-transition systems.[citation needed] We identify a labelled transition system as a category where the nodes are the states of the system, and the arrows are the labelled transitions from one state to another. The labels may represent observable side-effects that happen when running the state-transition system.

Our categorical notion of Implementation (see next chapter 3) is broadly related to the familiar notions of *simulation* and *bisimulation* between labelled state-transition systems, yet distinct from either of them, as it fulfills very distinct purposes.

Bisimulations fulfill much stronger criteria than implementations: they try to model some observational equivalence between computations (i.e. whatever you may observe about one, you may observe about the other), whereas implementations model observational inclusion of the concrete computation in the abstract computation (i.e. whatever you may observe about the concrete computation is indeed valid about the abstract computation). Indeed, an implementation can make (implementation) choices such that the set of potential observable behaviors of the concrete computation is a strict subset of the set of potential observable behaviors of abstract system. For instance, the abstract computation may leave some evaluation order unspecified and allow for many different orderings of observable effects, when the concrete computation would pick a specific ordering, or a subset of the allowed orderings. The abstract computation may also leave some parameters unspecified, such as the width of system integers, the length of buffers, the bandwidth, latency and timeouts of communication channels, the number of processors and their connection architecture, the names and passwords of users, etc., when some concrete computation might either choose specific values for these parameters or merely refine the constraints on them.

As for simple simulations, they have limited purpose beside being used as half the specification of a bisimulation. They do not have to satisfy the basic soundness criterion for implementations (see subsection 3.1.4), whereas implementations do not have to satisfy theirs. On the one hand, a computation can have many more transitions than the computation it simulates, whereas for an implementation cannot, as that would be considered the concrete computation lying about what the abstract computation can do. In an extreme case, a computation with one node and arrows of every label from itself to itself can trivially simulate any other computation; but it cannot implement anything but a superset of itself except with an empty (and useless) implementation that has no observable state. On the other hand, implementations do not have to satisfy the basic soundness criterion for simulations, and generally do not, for an implementation is allowed to optimize away and skip intermediate abstract states. The property of an implementation that guarantees simulation in the sense of labelled transition systems, we dubbed Completeness (see subsection 3.2.2).

In our categorical setting, labels can be expressed as a *labelling* functor from the category S to a monoid Λ of labels. (A monoid is category with only one node and usually many arrows.) Labelled transition systems typically use a free monoid, one generated by the atomic labels without any simplification rule. The subcategory of transitions with a null label can be distinguished as that of "internal" computations, without observable side-effect. An additional constraint that will be required of candidate implementations will be that the obvious labelling diagrams should commute.

Variants of labelled transition systems that can similarly be viewed as categories include Term Rewriting Systems or Rewriting Logic[citation needed], Hidden Algebra[citation needed], Abstract State Machines[citation needed], and small-step operational semantics.

2.2.2 Operational semantics

There are many flavors of operational semantics[citation needed], but they can all be viewed as categories.

Most obviously, small-step semantics or reduction semantics can be viewed as labelled state-transition systems, and hence as categories, as above. A node or state of the category corresponds to a record of a term being reduced and any accompanying reduction context. An arrow or transition is a reduction. If there is any side-effect, it is represented as a label of this transition. In the absence of side-effects, the labelling functor is trivial and can be omitted. The set of atomic steps constitutes a generator of the category (and is thus not itself a (sub)category since it isn't closed by composition).

Big-step semantics or natural semantics can be automatically translated to small-step semantics as follows: a state of the system is the data of a sub-expression being evaluated, the set of records of expressions reduced so far and the values to which they were reduced, and the structural context of expressions that remain to be reduced. In the case that the order in which to descend the structure is not specified by the big-step semantics, either an order can be chosen, or transitions with an empty label can be introduced for each of the possible orders.

Alternatively, big-step semantics can be viewed as a category with very few arrows, only those that go from expression to fully reduced value. Our categorical approach can also work in this case; however such categories have fewer arrows, and are thus easier to use as abstract systems to be implemented, and harder to use as concrete systems to implement other systems.

2.2.3 Modal Logic

A Modal Logic system can be viewed as a category. Nodes can be considered as state of knowledge of the system, and arrows as modal events that cause the system to change and the knowledge about the system to be updated accordingly. Temporal logic [citation needed] can use quantifiers to discuss possible futures of a state. Linear logic [citation needed] can model non-monotonous knowledge evolution. Hoare logic [citation needed], and more elaborate variants such as Dynamic Logic [citation needed] are specifically designed to model computations. The subset of arrows with trivial modality represent internal manipulation of knowledge (deduction, etc.) without gain (or loss) of information. Refinements are then implementations between such systems [18].

2.2.4 Partial Order

If we don't care about transition labels, and identify morphisms with common start and end objects, the category is a partial order, where $a \leq b$ iff b is a valid future or specialization for a. This applies for systems where the state is completely internalized within the node-set and there is no need to distinguish I/O effects using arrows. [citation needed]

2.2.5 Denotational Semantics

Our approach directly models common uses of denotational semantics[citation needed].

One common use, the definitional interpreter, is to consider the mapping of some source code to a known domain of computations, and posit the result as *the* meaning or *denotation* of a program by definition. Such a definitional interpreter thus reduces the problem of understanding the source domain to that of understanding the (usually known) (co)domain of computations and that of understanding the (usually simple) mapping. The target computations will then be the initial states of a computation in some well-known operational semantics, for instance, terms of the lambda calculus or bytecode of some virtual machine, with an evaluation context that is either implicitly empty or to be given as a parameter. This definitional mapping can be seen as an implementation either way (see next chapter), and be made functorial by definition, with arrows being decreed among source objects if and only if they exist among the target objects. It effectively identifies the source domain to a full subcategory (i.e. a subset) of the target domain. Thereafter, all study of the source computation has been reduced to the study of the target computation.

The other common use of denotational semantics, for static analysis, is equivalent to abstraction interpretation.

2.2.6 Abstract Interpretation

Our notion of Implementation is directly related to Abstract Interpretation [5]; in fact, in a way they are categorical opposite: i.e. they are very same phenomenon, but considered from the opposite direction, as going from abstract to concrete rather than from concrete to abstract. An Abstract Interpretation is also a partial functor between two categories, from one embodying a concrete computation to another one embodying an abstract computation. However, there are usually important differences in traditional usages of Abstract Interpretation and (Concrete) Implementation:

- Both Abstract Interpretation and (Concrete) Implementation often take as input a fully expressive programming language used by humans; but Abstract Interpretation goes “up” from that language to a more abstract one, whereas (Concrete) Implementation goes “down” from that language to a more concrete one.

- The underlying partial functor from concrete to abstract must preserve structure, must be determistic, and may be lossy (forget information), when going from the concrete to the abstract; the other way around, the opposite function is partial (an abstract state or transition may fail to be implemented), non-deterministic (an abstract state or transition may be implemented by multiple concrete ones), injective (a concrete state or transitions implements a single abstract one), co-functorial (implementation need not preserve structure, only interpretation must), and noisy (it may add new information or noise that is irrelevant to the abstract computation). Therefore an Abstract Interpretation usually does approximations whereas a (Concrete) Implementation adds new implementation details. For instance, the implementation may add a notion of memory addresses, execution time, or specific processor that does each part of the computation in a multiprocessor system. There maybe sometimes infinitely many possible ways to thus enrich the computation structure. The interpretation may forget those details.

- Abstract Interpretation is usually used for static analysis of a computation, where the dynamic evaluation context is unknown, and must be approximated by a set of all possible evaluation contexts. The first step is then to consider the (usually non computable) way that programs operate on sets of values (rather than just values). Then, to itself yield computable answers in finite time about potentially infinite computations in an infinite number of contexts (and not being able to solve the halting problem), an Abstract Interpretation *must* make significant approximations and lose information. By contrast, a (Concrete) Implementation of a computation can neither forget nor approximate the given dynamic evaluation context, but can only add new aspects to it.

- An Abstract Interpretation usually focuses on denotational semantics, and the end of an analysis often yields types in an approximation lattice (wherein the arrows embody the inclusion of sets of values and effects), or even in a discrete category (where said sets are mutually disjoint). Not all source terms or intermediate products of analysis are well-typed, hence the functor being partial. The whole point of the abstract interpretation is to *not* compute, but predict facts about the computation before the computation even takes place (if ever). By contrast, a (Concrete) Implementation usually focuses on operational semantics; it evaluates a lower-level system that involves additional details, often with performance as an important concern; it is only interested in a single evaluation context at a time; it usually only yields *one* of the possible answers is supposed to be computable (as contrasted with getting them all, especially when the language is non-deterministic). The whole point of the implementation is to compute it indeed. The implementation is not allowed to approximate, except in that it may sometimes fail to give an answer at all.

- An Abstract Interpretation usually has an adjunct functor going the other way, associating to each approximation the set of values it includes; this is an artefact of the approximation used. In the case of a (Concrete) Implementation, the opposite partial functor of interpretation does not usually have an adjunct (partial) functor, unless there is a *canonical* way (up to equivalence) to implement an abstract computation using an concrete computation. When this adjunct exists, it defines a compiler for the implementation.

2.3 Combining Computing Systems

It is useful to study algebraic operations that build more elaborate computing systems from simpler ones. These operations can be used either as synthesis tools, or as analysis tools. Category theory offers a wide tool box of such operations, including products and co-products (sums), initial and terminal objects, equalizers, limits, pullbacks and pushouts, monomorphisms and epimorphisms, functors, natural transformations, duality, monads, lifting, etc. These usual operations all apply to Computations considered as a Category; but we won't discuss most of them, since there's nothing special about applying them to Computations in particular. Instead, we will focus on a few operations that are particularly common or remarkable when applied to Computations.

2.3.1 Subcomputations and Supercomputations

It is often useful to consider cases where a computation is included in another. The included computation is then called the subcomputation, and the including computation is called the supercomputation. Formally, given a computation C as a category, a subcomputation of C is the data of a computation S and a "canonical" embedding (i.e. injective total functor) j from S into C. Typically j is assimilated to the identify function.

A subcomputation is "full" if the embedding is "full" in the usual categorical meaning: all the arrows between two reached nodes are also reached. A full subcomputation is characterized by the subset of nodes it contains. Then the identity function is also a trivial injective *partial* functor from the supercomputation to the subcomputation (being total on a subset of the supercomputation, namely the subcomputation). The two computations are therefore just as concrete or abstract as each other. The full subcomputation may then for instance represent a computation where some intermediate states have been eliminated, making for larger computation steps. If however some choices are removed in the subcomputation, then the supercomputation may still implement the subcomputation, but one that can get "stuck" into cases not supported by the subcomputation (see section 3.2.3).

If your category is the computation of one program written in a language, it is a full subcomputation of the more general category consisting of the computation of *any* program in that language: The latter category contains all the possible states of all the programs and all their state transitions; those of the specific program are an obvious subset. Of course, any given state only represents a single program at a time (or many programs that are thereupon equivalent at that point in the computation); in other words, when you pick a node in that category, you've chosen a specific program (or equivalence class of programs).

Similarly, given a language, a larger language, containing more programs, will be a supercomputation, whereas a more restricted language, containing fewer programs, will be a full subcomputation.

Hypotheses about the execution environment can be modeled by restricting the arrows in a supercomputation (e.g. some erroneous transitions are assumed never to happen). We already saw that *internal* computations, those without any input/output, could be modeled by the subsystem of transitions with a null label. This can be generalized in various ways, to model *internally directed* computations, those that do not depend on any input/output operation not guaranteed to be feasible: for instance, input of ticks from the wall clock, read/write operations on persistent media, communication with trusted servers, etc., are allowed, but input from a human user or untrusted client are not considered.

More subcategories can model various kinds of restrictions on computations: resource limitations, invariant preservations, hypotheses about the external world, effects previously approximated away, "magic" behavior, etc.

These concepts of subcomputations will be used, but once again, this formalization is voluntarily large, so as to allow to express mappings between computations of various different abstraction levels.

Note that whether adding arrows to a computation to achieve a supercomputation, or removing arrows from a computation to achieve a subcomputation, it is not just individual arrows, but a coherent set of arrows that must be added or removed, so that the resulting set is closed via composition and indeed constitutes a category. A proper way to do that is to consider generators of the computations at hand (as in small step semantics), and add new generators in the case of extension, or restrict the product set of generators (plus possible identity transitions) in the case of restriction. A subset of a computation that is not a category will be called a subset, and not a subcomputation.

2.3.2 Concurrent Computations

A family of computations can be made to run in parallel. When the computations run independently, the resulting computation consists in the usual cartesian product of the family of computations, considered as categories. [citation needed]

Expressing interaction between computations running in parallel can be expressed in two complementary ways. In the first approach to expressing interaction, we start from computations that do not include any arrows for I/O, an extend their cartesian product with arrows that express internal communication by simultaneously modifying the state of several computations.

Arrows that do not involve such communication are called local transitions or communication-free arrows. Arrows added to the computation are non-local transitions or communication arrows. In the second approach, we start from computations that each communicate with its outside world, and restrict their cartesian product so that the effect I/O operation done by one of them is reflected by simultaneous corresponding I/O operations in other ones. In categorical terms, the resulting system is a *pullback*, i.e. a constrained product where some interactions in one computation are each identified to a complementary interaction in the other computation. The cartesian product is then the particular case where the two computations are independent, with no interations in either computation to identify to interactions on the other.

A concurrent computation is a computation thus achieved by modifying or constraining the arrows of a cartesian product. The computations in the cartesian product are the components or processes.

Note that being a concurrent computation is not an intrinsic property of the computation as a category, but an extrinsic property of its formalization, that depends on the way it is built or viewed as the modification of a cartesian product. Indeed, up to isomorphism, any computation could be butchered into components in arbitrarily many ways, with ad-hoc arrows being added or removed to fit the desired computation. Most such decompositions into components are useless; sometimes, several different decompositions of the same computation as a concurrent computation can be useful each to prove an appropriate property. We'll see examples of that in sections 2.3.2 and 6.2.3 below. As says Lamport, processes are in the eye of the beholder [17].

Chapter 3

Properties of Implementations

This chapter is mostly an extension of the formal parts of our 1999 article "Formalizing the Notion of Implementation" [22], and a generalization of its concept to computations with I/O.

3.1 Definitional Properties

3.1.1 Intent

Given a higher-level "abstract" computation A and a lower-level "concrete" computation C, an implementation of A with C is the inverse Φ^{-1} of a partial functor Φ from C to A. It is then said that Φ^{-1} implements A, or by extension that C implements A (using Φ, which may be implicit). Φ being a (partial) functor means that it preserves the observational semantics of the abstract computation: observations done on the concrete computation C are consistent with observations that would have been done "directly" on the abstract computation A.

In one class of implementations of interest, the abstract computation is a modern high-level programming language, and the concrete computation is a process running on some given virtual machine. In another class, the abstract computation is the process on the virtual machine, and the concrete computation is a process running on some actual CPU. In yet another class, the abstract computation is whatever the user may observe, and the concrete computation is the program as evaluated by the modern high-level programming language in which it was written.

More generally, the two computations are any two levels of program representation that are respectively input and output of some compiler pass or series of passes. Each implementation can thus represent either the entire semantic tower rooting the execution of some high-level program onto a low-level machine, or a thin layer that focuses on the details of one implementation, or any small or large "slice" between two levels in that tower.

Note however that these computations are not compile-time only entities: they include runtime state such as a control stack, a set of bindings, a data heap, memory mappings, open file descriptors, other interactions with the Operating System, etc. Programming languages must be considered together with their actual runtime support, and with any linked libraries written in the same or a different programming languages, or with non-standard extensions, using extended operating system features, various configuration settings, or machine-specific capabilities. When analyzing the semantics of a program, anything that partakes in its behavior must be taken into consideration. On the other hand, we often are only interested in a small fragment of a larger computation, at which point a whole lot of irrelevant details can be omitted.

3.1.2 Partiality

An implementation is a *partial* functor, which means a structure-preserving function (functor) where not all point in the domain have an image (partial). It is important to not consider only total functors, because the operations that are atomic in the abstract computation are not generally available as atomic operations in the concrete computation, and *vice versa*, as illustrated in the diagram below:

$$\begin{tikzcd}
\texttt{\{puts("Hell");puts("o");\}} \arrow[r, "{\texttt{"Hell"}}", "C"'] & \texttt{\{puts("o");\}} \arrow[r, "{\texttt{"o"}}", "C"'] & \texttt{\{\}} \\
\texttt{\{(Hello) show\}} \arrow[u, dashed, mapsto] \arrow[r, "PostScript"'] & \texttt{(Hello) \{show\}} \arrow[r, "\texttt{(Hello)}", "PostScript"'] & \texttt{\{\}} \arrow[u, dashed, mapsto]
\end{tikzcd}$$

In this diagram, a computation in a hypothetical C-like language is implemented by computation in a hypothetical stack machine language. The abstract computation outputs a string, then another one. The concrete program directly writes the concatenation of the two strings. The abstract operational semantics is defined in terms of a continuation k and with the effect of outputting a stream of characters. The concrete operational semantics is defined in terms of a data stack s, a continuation k, and also the effect of outputting a stream of characters. The abstract state where one of two strings only is output is not implemented. The concrete state where a string is pushed to the stack is not represented. Yet the concrete computation reaches an end state which can be interpreted as having had all the specified effects of the abstract computation.

3.1.3 Bicolor Diagrams

An implementation Φ^{-1} of A with C is a partial functor from C to A, which once again we represent with this diagram, where the dashed line indicates partiality:

$$\begin{tikzcd}
A \\
C \arrow[u, dashed, "\Phi"]
\end{tikzcd}$$

In traditional category theory, arrows usually represent total functors. Thus we have to draw the following diagram to represent the same thing, where O is the full subcategory of C the elements of which are "observable" by having a meaning in A, j is the canonical inclusion of O in C (the arrow fin signifies that j is injective), and ϕ is a total functor from O to A:

$$\begin{tikzcd}
 & A \\
O \arrow[ur, "\phi"] \arrow[dr, hook, "j"] & \\
 & C
\end{tikzcd}$$

Now we can *define* a partial functor as being just the above and draw it in a dashed line. The logical *introduction rule* for such a diagram would look like this:

$$\begin{tikzcd}
 & A & & & A \\
O \arrow[ur, "\phi"] \arrow[dr, hook, "j"] & & {} \arrow[r] & {} & \\
 & C & & & C \arrow[uu, dashed, "\Phi"]
\end{tikzcd}$$

However, for the sake of smaller, more readable diagrams, we will adopt the convention of *writing such rules by drawing hypotheses in black ink and conclusions in blue ink.* Thus, the above introduction rule we will more simply draw as follows:

$$\begin{tikzcd}
 & A \\
O \arrow[ur, "\phi"] \arrow[dr, hook, "j"] & \\
 & C \arrow[uu, dashed, blue, "\Phi"']
\end{tikzcd}$$

Similarly, we can *define* element association by a "partial function" in terms of element association by the underlying function and injection, with the following diagram:

$$\begin{tikzcd}
 & a \\
o \arrow[ur, mapsto, "\phi"'] \arrow[dr, mapsto, "j"'] & \\
 & c \arrow[uu, mapsto, dashed, blue, "\Phi"']
\end{tikzcd}$$

Note that in the above diagram, j is notionally an inclusion, and thus o is *equal* to c, at least morally: $o = c$. In this thesis, we usually speak only of c; we omit to explicitly distinguish o as a separate entity. We similarly also only speak of Φ or Φ^{-1}, and leave ϕ and j implicit unless when specifically needed in formulas.

Given a node c of C we say that it is *observable* when there exists an element o of O that is equal to c; otherwise we say that c is not observable. We similarly extend the notion of being observable to arrows of C. O being a full subcategory of C, an arrow is observable if and only if its domain and co-domain (starting and ending nodes) are both observable. When an observable arrow in C can be decomposed as the product of two arrows, then the node where those two arrows meet is called an *intermediate* computation state, like c' in the diagram below:

$$\begin{tikzcd}
a & & a'' \\
c \arrow[u, dashed, |->, "\Phi"'] \arrow[r, "C"'] & c' \arrow[r, "C"'] & c'' \arrow[u, dashed, |->, "\Phi"']
\end{tikzcd}$$

Partiality is important; it is the fact that not every state is observable: the concrete system C may well contain transitions that are *subatomic* with respect to the invariants exposed by the abstract system A: in the proverbial transfer of money from one account to the other, the abstract system A is not allowed to see the accounts while some amount has been debited from the source and not yet credited to the destination (or the other way around); but at the level of C no such atomic operation may be available, and there *will* be intermediate states that *temporarily* violate the invariants of the abstract system A.

Partiality thus accounts for the fact that not every concrete state is "stable" or "observable" with an abstract meaning; not every concrete transition corresponds to an abstract transition; intermediate concrete states may break abstract invariants; intermediate concrete transitions break the atomicity of abstract transitions; concrete optimizations may short-circuit intermediate abstract transitions; etc.

3.1.4 Soundness

The most essential property for an implementation is to never give wrong answers.

It's often acceptable for an implementation to fail to give a correct answer: the computation might take too much time, or consume too much memory, or run out of some other scarce resource; some malfunction or external event may cause the computer to crash. But a wrong answer may cascade into other computations and cause a wider downstream computation to fail, and cascade to the user with potentially catastrophic results: some human may take the wrong decision, some device may cause a deadly accident. Even in more day to day scenarios, every wrong answer is a failure of the computing system to do what it was supposed to do.

The diagram that describes this property is as follows:

$$\begin{tikzcd}
a \arrow[r, "A"'] & a' \\
c \arrow[u, dashed, |->, "\Phi"'] \arrow[r, "C"'] & c' \arrow[u, dashed, |->, "\Phi"']
\end{tikzcd}$$

In informal terms, any observable (intermediate or final) result that may be reached by computing in the concrete system must correspond to a valid abstract result that could legally have been obtained by computing in the abstract system.

We will call this property *soundness*. It is such an essential property that we will require from every candidate implementation that it shall be *sound*, or we will not consider it an implementation at all.

This property can be equivalently restated in different ways:

- If by computing from an (observable) concrete implementation c of a, we can observe an (intermediate or final) concrete state c' that can be interpretable as abstract state a', then a' must be a correct result that could have been found by doing computations purely within the abstract system.

- Any abstract observation made by observing and interpreting the concrete system must be valid in the abstract system.

- The basic reduction structure of the computing systems must be preserved by ϕ.

- In categorical terms, this is summarized by saying that ϕ must be a (covariant) functor. Actually, *functoriality* also implies preservation of arrow composition; we will indeed require that all diagrams involving arrow composition shall commute.

- All the answers computed thanks to the implementation are correct.

Notice that structure preservation happens in a way that is contravariant to the direction usually thought of as an "implementation": it is the "interpretation (partial) function" Φ that preserves structure; and it goes in a direction opposite to that of the "implementation (non-deterministic) function" Φ^{-1}. In categorical words, the theory of implementation is inherently *decompositional*, rather than compositional. We may argue that it explains the utter failure of so many attempts to build compositional meta-objects framework. Another way of seeing things is that this approach explores semantics in a way reverse to that followed by abstract interpretation and static analysis, that study interpretation functions and their nice properties. All this justifies our privileging Φ as the object of interest, as far as nice semantics are concerned.

Note that by definition of O as a full subcategory, it is given that j be structure-preserving.

Soundness is a *composable* property: if Φ^{-1} is a sound implementation of A with C, and Ψ^{-1} is a sound implementation of C with D, then $\Psi^{-1} \circ \Phi^{-1}$ is a sound implementation of A with D.

To illustrate the difference between sound and unsound, consider computations on natural numbers, where the states we observe are those when the system yields results. An implementation with fixed-precision integers will be sound if it traps on overflow; any result it will may yield will be correct. An implementation with fixed-precision integers that silent wraps computations that overflow is not sound (at least, not in the eventuality of such overflow), and may yield wrong results when initial conditions imply too large numbers during computation. Note that soundness does not mandate termination. It only mandates that in case of termination or legal observation, the implementation must yield a correct result. It is always sound (according to this definition) for an implementation to fail to answer; of course, answers are desirable when possible, but misleading incorrect answers are worse than no answer. When designing mission-critical systems, it is sound (according to this definition) not to come with a design, but unsound to come with a design that might kill people under intended use conditions.

This notion of soundness is well established under this name in computational logical [12]. It has been called "safety" by Lamport [16]. Goerigk called it "partial correctness" [11], where "partial" corresponds to the fact that Φ is a partial function rather than a total function.

The computational content of soundness is that we can use (partial) operational execution of the concrete computing system as a valid substitution (modulo translation via Φ) to (partial) operational execution of the abstract computing system — which is precisely what implementations are all about.

3.2 Optional Properties

3.2.1 Intent

We study several properties that may or may not be interesting for an implementation to have.

Some of these properties are pretty obvious and well-known: *completeness* and its many variants are properties that tell how closely you can go from the abstract computation to the concrete computation; *liveness* and its many variants are properties that tell how progress in the concrete computation guarantees progress in the abstract computation. *co-liveness* and its many variants are properties that tell how termination in the concrete computation guarantees termination in the abstract computation. We will use them as benchmarks to illustrate how our formalism can help reason about these common concerns that implementers face.

However, another of these properties, *observability*, tells how you can go from the concrete computation back to the abstract computation. We believe it is vastly underrated — and pinpointing its importance might be the single major contribution of this thesis. Observability generalizes the notion of *safe point*, as a node from which an abstract meaning can be recovered from a concrete computation. All who develop of concurrent implementations of programming languages have to deal with one or multiple such notions. This notion is crucial to the rest of this thesis, in which we will argue that having formalized it trivializes a variety of so-far difficult applications.

The properties in this section do not in any way exhaust the range of what one may find interesting about implementations. For instance, some programmers will care a lot about some kind of side-effects, such as I/O channels, user-interface behavior, real-time constraints, etc.; then an all-important property they would require from every implementation might be that some promises made by the abstract computation regarding those side-effects are indeed fulfilled by the concrete computation. Now, the previous formalization framework in terms of categories is general purpose. Whether with diagrams such as those we use, or with more general logical formulas, one can capture the relevant aspects of computations, and specify any desired properties pertaining to these aspects Of course, some aspects may not be particularly easy to formalize, such as whether a concrete computation does a good job of providing a usable graphical interface when implementing an abstract "interface" to data structures as concrete pixels on a screen that change color with time. But that's an issue with the limits of mathematical formalization in general, not with this particular approach.

In the rest of this section, we'll consider as defined above an implementation Φ^{-1} of an abstract computing system A with a concrete system C, given by $\Phi = \phi \circ j^{-1}$ where O is the subcategory of observable states with a canonical injection $j : O \longrightarrow C$, and an interpretation function $\phi : O \longrightarrow A$.

3.2.2 Completeness and its variants

Rename "Completeness" to "Controlability", "Complete" to "Controlable" ??? XXXX

A family of properties

Completeness and its variants form family of properties that one may desire in an implementation, that follow the general pattern: "if you can do *something* in the abstract computation, then you can the *same* something in the concrete computation". In informal terms, you can control your interaction with the concrete computation in terms of the concepts meaningful to the abstract computation.

All these properties are composable (except noted otherwise): if two implementations have it and can be composed, then the product of their composition has the same property too, because from the abstract *something* we can get the *same* something in through the first implementation in the intermediate computation, then the *same* something through the first implementation in the concrete computation.

These property is important to show that an implementation captures all of an aspect that one cares about; or, when composing many implementations, this property can help ensure that blame for failing to capture all of that aspect must lie in a particular implementation step that fails to possess this property.

In particular, it is not possible to have a total implementation of an infinite computation with a finite computation (by a simple counting argument). Therefore, the best that can be done usually is either to restrict oneself to finite computations (and e.g. prove bounds on memory and time required), or to prove totality at every step but one in a tower of implementations: that crucial step will represent a potentially infinite graph into finite memory by assigning memory addresses to graph nodes; it is not total and may fail with a memory overflow error; but all implementations above are total from one infinite computation to another, and all implementations below are total from on finite computation to another. Totality failures have been contained, if not eliminated.

Completeness

The following version of this property, we'll call *completeness*, as defined by the following diagram:

$$\begin{tikzcd}
a \arrow[r, "A"'] & a' \\
c \arrow[u, dashed, |->, "\Phi"'] \arrow[r, "C"'] & c' \arrow[u, dashed, |->, "\Phi"']
\end{tikzcd}$$

Informally, if you currently start from a concrete computation state c that is observable and has meaning a, and if you consider a transition from a to a' in the abstract computation A (with whatever side-effects allowed in A), then there is a state c' that is observable and has meaning a', and a transition from c to c' in the concrete computation C (the side-effects of which map into the side-effects of the previous abstract transition).

Completeness is useful when you want to manually control how and how much a computation advances, and what choices it takes along the way (if evaluation isn't deterministic). This is important when the computation is controled by some outside system, such as an interactive debugger, a recording of some past run, a search heuristic, etc.

When the computation is a labelled state transition system, completeness means that the implementation is a simulation (not the interpretation).

Totality

Totality is an even simpler variant in that theme, as defined by the diagram below:

$$\begin{tikzcd}
a \\
c \arrow[u, dashed, mapsto, "\Phi"']
\end{tikzcd}$$

This property "just" means that ϕ is surjective, when considered as an application from node to node. In other words, any given abstract computation state a can be implemented as a concrete computation state c.

This property is very important e.g. to show that a compiler can handle all cases specified in the abstract language, or to show that the deficit in cases that can't be handled is confined to a few well-specified cases.

You could consider a variant of totality that works on any arrow, and given a and a' finds both c and c'. But then, c could never be guaranteed to be what you want, and c' could never be guaranteed to be a useful node from which to continue computation — unless you are willing to depend on further properties to provide such guarantees. But at that point, you are better off requiring from the implementation both totality and fullness (see Strong Completeness below).

Fullness

Fullness is being able to implement all arrows between implementable nodes. This means that ϕ is a full functor, i.e. it is surjective when restricted to arrows between any two objects. The corresponding diagram is as follows:

$$\begin{tikzcd}
a \arrow[r, "A"'] & a' \\
c \arrow[u, dashed, mapsto, "\Phi"'] \arrow[r, "C"'] & c' \arrow[u, dashed, mapsto, "\Phi"']
\end{tikzcd}$$

The implementation preserves the full richness of the abstract computation, so that many properties can be transfered from one computation to the other.

However, fullness is computationally expensive, since it requires arrows between any two concrete representations of arrow-related abstract states, and in particular any two concrete representations of a same abstract state. So, any path-dependence or non-deterministic choice in representation must be reversible inside the system.

Strong Completeness

Strong Completeness is the conjunction of fullness and totality: surjectivity of ϕ for both nodes and arrows between given nodes. It implies completeness: given a, c, a', use totality to deduce c' then fullness to deduce the arrow from c to c'. In general, it is computationally less practical than completeness, since using totality will pick c' without knowledge about c, then may have to pay a high price undoing any choice embedded in c to instead implement the choices embedded in c'. This may not be much of an issue if there are no implicit choices made when representing an abstract state, and it all representations are "canonical"; but that's not the general case.

Weak Completeness

Weak Completeness is a weaker variant of completeness whereby the abstract user can specify an abstract arrow as a "direction" in which to compute, and the concrete computation must be able to compute "in that direction", but need not stop exactly at the tip of that arrow: instead the concrete computation may go "past" this tip as it keeps computing, and only stop some time after; how far it stops after the target may be controlled by a subset A^s of A in which that extra step is constrained to remain. The corresponding diagram is as follows:

$$\begin{tikzcd}
a \arrow[r, "A"'] & a' \arrow[r, "A"'] & a'' \\
c \arrow[u, dashed, mapsto, "\Phi"'] \arrow[rr, "C"'] & & c'' \arrow[u, dashed, mapsto, "\Phi"']
\end{tikzcd}$$

3.2.3 Liveness and its variants

Another family of properties

Liveness and its variants form a family of properties that one may desire in an implementation, that follow the general pattern: "The abstract computation will advance given sufficient advance in the concrete computation".

All these properties require some measurable notion of advancing, for instance based on a subset of strictly advancing arrows, itself possibly generated from a subset of "atomic" arrows. Given compatible enough notions of advancing, these properties will usually be composable.

They are important to show that the implementation cannot get "stuck", that it will keep advancing and making progress until it reaches an answer from which no further progress can be made.

Sometimes, the advance will be conditional on some adverse event not happening (e.g. running out of memory) or not happening too often (e.g. the power going out). As with the previous families of properties, proving (unconditional) liveness for each step but one (or a few) in a tower of implementations allows to contain liveness failures to one set of well-understood failure modes at one (or a few) levels of abstractions.

Advancing

The liveness family of properties assume that each computation X comes with a subset X^+ of arrows that "advance" in X. Importantly, X^+ is closed by composition on either side with arrows in X (this encodes the fact that advance is irreversible), and X^+ doesn't contain identity arrows. The complementary subset $X^0 = X \setminus X^+$ of arrows that don't advance is also notable: it always contains the identity arrows, but it may contain additional arrows. Depending on what one is interested in viewing as "advancing", the non-advancing arrows may or may not include arrows that encode reversible changes in representation, "administrative" changes, uncommitted transactions, internal computations without "observable" input or output or side-effect, or interruptions and recovery from interruptions.

A very same computation (e.g. as an isomorphic category) can typically be equipped with several (infinitely many) notions of "advancing", depending on what one is interested in. The “advance” subset only includes arrows that correspond to a change in some abstract view (or combination of abstract views) of the state of the system and/or other systems it interacts with; such change might be: internal change in one or more among identified interesting variables that define the system; more generally change in the internal state of the system up to some equivalence function; consumption of input from a particular channel or in general; production of output onto a particular channel or in general; observation of some non-deterministic choice; deterministic update of the state of the system after some computations; etc. Arrows that do not affect the considered abstract view of the system are then not considered as “advancing”.

In the context of implementing advancing computations, we will often require that non-advancing shall preserve observability, as per the following diagram:

$$\begin{tikzcd}
a \arrow[r, "A^s"'] & a' \\
c \arrow[u, dashed, maps to, "\Phi"'] \arrow[r, "C^s"'] & c' \arrow[u, dashed, maps to, "\Phi"']
\end{tikzcd}$$

Liveness

Liveness is the property that if C will advance, then A will advance. We will define it formally with the following diagram:

$$\begin{tikzcd}
a \arrow[rrr, "A^+"'] & & & a' & & \\
 & & & c' \arrow[u, dashed, maps to, "\Phi"'] \arrow[rd] & & \\
c \arrow[uu, dashed, maps to, "\Phi"'] \arrow[rrru, bend left] \arrow[r, "C^+"'] & c_1 \arrow[r, "C^+"'] & c_2 \arrow[rr, dotted] & & c_n \arrow[r, dotted] & {}
\end{tikzcd}$$

The property asserts that for any infinite sequence of advancing concrete computation transitions, there is an integer n such that the abstract computation has strictly advanced before concrete computation reaches the n^{th} transition — the concrete computation will spontaneous “go past” more advanced abstract states as it itself advances.

Note that given the hypotheses, it is not usually possible to guarantee that any of the c_k will be observable. Indeed, unless all (or most) elements of C are observable, then it is possible to select every c_k so that they are all non-observable intermediate states. However, for a stronger property see Strong Liveness below.

Liveness is a composable property. If we look at temporal logic statements on transitions, liveness ensures preservation of increasing “must eventually” statements by Φ^{-1}: if after sufficient advance, the state of the abstract computation must eventually satisfy some invariant, then after sufficient advance, the state of the concrete computation that implements it will satisfy it (more precisely, at some point, any observable concrete state reachable from that point will be mapped by Φ to an abstract state that satisfies the predicate).

Bounded Liveness

In *Bounded Liveness*, we depend on an additional notion of *length* for computations, and require that any long enough concrete computation goes past an abstract computation of at least some minimum length.

For each computation, length is function from arrows to non-negative real number. The function is additive (or sometimes subadditive), which means that the length of the composition of two arrows is equal to (respectively no greater than) the sum of the lengths of individual arrows. Real-time liveness is then the following property:

$$\begin{tikzcd}
a \arrow[r, "A_{\geqslant l_a}"'] & a'' & \\
 & c'' \arrow[u, dashed, mapsto, "\Phi"'] \arrow[dr] & \\
c \arrow[uu, dashed, mapsto, "\Phi"'] \arrow[ur, bend left] \arrow[rr, "C_{\geqslant l_c}"'] & & c'
\end{tikzcd}$$

In other words, that there is an an abstract length l_a and a concrete length l_c such that if $f : c \longrightarrow c'$ is a concrete arrow of length no less than l_c, then it can be decomposed into $f = g \circ h$ where h is an observable arrow the meaning of which has length no less than l_a. (NB: in the subadditive case, we have to modify the sentence above so that l_a is universally quantified rather than existentially quantified.)

Bounded liveness is composable, given matching notions of length. Bounded liveness implies liveness where advancing corresponds to an arrow having strictly positive length.

Strong Liveness

Strong Liveness is a variant of liveness whereby an infinite sequence of *atomic* concrete transitions actually contains an observable arrow the meaning of which is advancing.

Strong Liveness assumes that the concrete computation's advancing arrows are generated by a set of *atomic* arrows, a subset of C noted C_1. The corresponding diagram is then as follows:

$$\begin{tikzcd}
a \arrow[rrr, "A^+"'] & & & a' & & \\
c \arrow[u, dashed, mapsto, "\Phi"'] \arrow[r, "C_1"'] & c_1 \arrow[r, "C_1"'] & c_2 \arrow[r, dotted] & c_m \arrow[u, dashed, mapsto, "\Phi"'] \arrow[r, dotted] & c_n \arrow[r, dotted] & {}
\end{tikzcd}$$

Strong Liveness is a useful property to have when C is a sequential (single threaded) computation, where you prove that the implementation will spontaneously reach observable "safe points" at which e.g. garbage collection may happen.

Strong Liveness is a composable property. On the other hand, Strong Liveness will not usually be preserved when considering the product of two computations (i.e. running two or more computations in parallel), whereas Liveness will. Strong Liveness will be preserved by parallelism if there are no unobservable intermediate states, or if all concrete transitions are precisely timed and synchronized between the parallel implementations (e.g. in a synchronous digital circuit, with a clock).

Bounded Strong Liveness

Bounded Strong Liveness is a combination of Strong Liveness and Bounded Liveness, whereby a finite sequence of *atomic* concrete transitions of total length at least l_c contains an observable arrow the meaning of which has length at least l_a.

$$\begin{tikzcd}
a \arrow[rrr, "A_{\geqslant l_a}"'] & & & a' & \\
c \arrow[u, dashed, |->, "\Phi"'] \arrow[r, "C_1"'] \arrow[rrrr, bend right, "C_{\geqslant l_c}"'] & c_1 \arrow[r, "C_1"'] & c_2 \arrow[r, dotted] & c_m \arrow[u, dashed, |->, "\Phi"'] \arrow[r, dotted] & c_n
\end{tikzcd}$$

Bounded Strong Liveness assumes that the concrete computation's arrows are finitely generated by a set of *atomic* arrows.

Bounded Strong Liveness is a composable property and trivially implies Strong Liveness.

(Bounded) Strong Step Preservation

Strong Step Preservation is a variant of Strong Liveness assuming that we have notions of "atomic" computations for A as well as for C, and requiring that the arrow from a to a' be atomic (in A_1 rather than merely A^+). This is useful for straightforward implementations of "virtual machines". Strong Step Preservation is a composable property and trivially implies Strong Liveness.

Similarly, *Bounded Strong Step Preservation* will offer the same guarantee with a static time bound. Bounded Strong Step Preservation is a composable property and trivially implies Strong Step Preservation and Bounded Strong Liveness.

3.2.4 Co-Liveness and its variants

Yet another family of properties

Liveness and its variants provided guarantees that the abstract computation is advancing when the concrete computation is advancing; co-liveness provides guarantees that the abstract computation terminates when the concrete computation terminates. Co-liveness complements liveness: liveness says that concrete computations are relevant when they keep advancing; co-liveness says that concrete computations are relevant when they stop advancing.

Co-Liveness

We'll call *Co-Liveness* the property according to which if a concrete evaluation terminates, then it is observable, and its meaning also terminates in the abstract computation.

Here, terminates means that there are no advancing arrows starting from where the evaluation stops. A concrete evaluation here is any concrete arrow starting from an observable node.

Interestingly, our formalism so far does not allow for a nice of insightful diagram for co-liveness as defined above. However, the (equivalent) contraposition of co-liveness could almost be a simple diagram: it says that if there is an advancing abstract arrow from the meaning of a concrete node, then an advancing concrete arrow starting from that node must either have advancing arrows that continue it, or it must be observable. If we devised some way of representing negation, or at least negation of a simple existential, or disjunction, or higher-order diagrams, we could represent the property and/or its contraposition. But we will leave devising such a graphical representations as an exercise for the reader. Instead, we will find variants of co-liveness that do possess simple yet interesting diagrams in our formalism.

Relevance

Relevance is the notion that unless an abstract state is terminal, any concrete transition from a state implementing it will make progress "toward" some abstract transition. It corresponds to the following diagram:

$$\begin{tikzcd}
 & & a''' \\
a \arrow[urr, "A^+"'] \arrow[rr, "A^+"'] & & a'' \\
c \arrow[u, dashed, maps to, "\Phi"'] \arrow[r, "C"'] \arrow[rr, bend right, "C^+"'] & c' \arrow[r, "C"'] & c'' \arrow[u, maps to, "\Phi"']
\end{tikzcd}$$

The diagram supposes that the abstract computation state a isn't terminal, i.e. that advancing the abstract computation is possible from a, as posited by the otherwise unused arrow from from a to a''', being in the advancing subset A^+. It further supposes that some concrete state c implementing a and some concrete transition f from c to c' that "starts computing from there". It then requires that this transition be a prefix to a larger computation $h = g \circ f$ from c to c'', that is advancing in C, that is observable, and the meaning $\Phi(h)$ of which is advancing in A from a to some node a''. In informal terms, any concrete transition is relevant as part of the implementation of a larger observable abstract transition.

Relevance implies co-liveness, *assuming that C^0 preserves observation and termination*, i.e. that if c is observable (respectively terminates) and there is an arrow from c to c' in the non-advancing subset C^0 of C, then c' is also observable (respectively terminates). Indeed, consider the premises of the contraposition of co-liveness. If there is an advancing abstract arrow from a to a''', and if c to c' is advancing, then we use relevance to deduce c'' and a''; either the arrow g from c' to c'' is advancing, and QED, or it's in C^0 and then c' is observable like c'', and also QED.

The diagram for relevance is complex and ugly, yet it has the advantage of fitting in a single diagram a sufficient condition for co-liveness, that still has an intuitive interpretation. Now, we can simplify things a bit by separating the concerns in two: *advance preservation* for the case where $c' = c$ and we require advancing (the diagram above removing c'), and *observability* for the case c' is arbitrary and we do not require advancing (the diagram above removing a''', the curved arrow, and the remaining A^+ restriction).

Advance Preservation

Advance Preservation is the property that if advancing is possible in A, then advancing is possible by evaluating in C.

The diagram is as follows:

$$\begin{tikzcd}
 & a'' \\
a \arrow[ur, "A^+"'] \arrow[r, "A^+"'] & a' \\
c \arrow[u, dashed, maps to, "\Phi"'] \arrow[r, "C^+"'] & c' \arrow[u, maps to, "\Phi"']
\end{tikzcd}$$

In other words, if advancing from a is possible in A, and c implements a, then advancing is possible in C starting from c, in a way that is observably advancing in A.

Note that in the above diagram, a'' is a priori unrelated to a' — that is, if the system can advance, it will advance, but if there is any choice involved, it may be following a path completely different from any particular path one might desire. If you want to force a' to be the same as a'', then it's completeness; however, completeness and advance preservation have a very different computational content: with advance preservation, the evaluation is driven by the concrete implementation, which is the whole point of an implementation indeed; by contrast, with completeness, the evaluation is driven by the abstract implementation, which as a regular evaluation strategy would defeat the purpose of implementation. However, if A is (a) deterministic and sequential or at least (b) confluent, then indeed the difference between completeness and advance preservation is minor, since in a way, there is no real choice left to the implementation on where to take the computation.

Advance preservation as such isn't composable, because if an implementation of A with C and an implementation of C with D have it, then you can advance in C thanks to the implementation in D, but that isn't enough to guarantee an advance in A. Now, if you consider the subset of observable states to be part of the invariant to preserve, then A, O-advance preservation is composable with O, O'-advance preservation, etc.

3.2.5 Observability

Rename to Safety, in light of it matching the Game-Theoretic Safety of blockchain applications? See AGEFp properties in Computation Tree Logic (CTL)

A last family of properties

Observability and its variants constitute a last family of properties in this series, but not the least. Observability guarantees that some observable abstract meaning can always be extracted from an intermediate concrete computation state.

We saw above that, combined with advance preservation, it implies relevance that implies co-liveness. But it is a much more general property with many important applications, from garbage collection to process migration.

We are not aware of any general variant of this property having been studied in the past, and believe we discovered its importance in our 1999 article “Formalizing the Notion of Implementation, as Illustrated with Concurrent Garbage Collection” [22] (unpublished), where we originally dubiously dubbed it “soundness” — and called “safety” what we now call soundness, after Lamport [16]). Actually, this property of “observability” corresponds closely to a “safety” in multiplayer games where a player (in this case, the implementation) wants to be able to reach a “safe” point (in this case, one that is observable) regardless of what the other players do (thus, without depending on inputs from users and other peripherals).

However, a specific variant of this property was famously discovered by the MIT hackers who developed ITS in the 1960s [7], and dubbed it “PCLSRing” (pee-cee-luser-ing); this discovery was described by Alan Bawden in his 1989 report [3]: a process could stop another process and observe its state, and the observed process would never be in the middle of a system call; any ongoing system call would be either completed, or restarted, or interrupted in such a way that it could be resumed from state purely in userland (e.g. with registers pointing to the end of a partially read or written buffer before restart of the system call). The metaphor used by ITS hackers was that a process could never be caught ”with its pants down”.

Observability

Observability is the property that given an intermediate concrete computation state c', an observable computation state c'' can be extracted by applying a non-advancing transition. The diagram is as follows:

$$\begin{tikzcd}
a \arrow[rrrr, "A"'] & & & & a'' \\
& c \arrow[ul, dashed, mapsto, "\Phi"'] \arrow[r, "C"'] & c' \arrow[r, "C^s"'] & c'' \arrow[ur, dashed, mapsto, "\Phi"'] &
\end{tikzcd}$$

That is, if a concrete computation starting from a stable state c has reached an intermediate state c', then it must be possible to stabilize the computation from c' to a further observable state c'' by computing in C without advancing. This diagram is somewhat analogous to that of Weak Completeness (see subsubsection 3.2.2), with abstract and concrete switched (however, mind that the direction of functoriality of Φ is not switched, which limits the analogy).

Note however that the notion of advancing used with Observability is often different from the notion of advancing used in other contexts (e.g. as used for Liveness or Co-Liveness): often, when considering Liveness, a wide view of state and I/O is considered, where any change to any aspect of the computation is considered as advancing; by contrast, often, when considering Observability, a more narrow view of state and I/O is considered, where only input (or synchronous output) is typically excluded from C^0, and any kind of change that involves no input and has low enough latency is included. Indeed, a typical way to implement Observability is for an interrupted computation to be resumed and allowed to run and otherwise make some progress until it reaches the next "safe point". For all these reasons, we will call C^s the subset of "safe arrows", by contrast with the regular subset C^0 of non-advancing arrows for some regular notion of "advancing". Therefore, when using Observability in other contexts, the deduced arrow must be considered as being an arbitrary arrow in C^s, rather than an arrow in C^0 with respect to advancing in that context.

In other words, the concrete computation mustn't have meaningless transitions into the permanently unobservable; it mustn't spontaneously and unrecoverably go wild or enter a deadlock; it mustn't require the external world to do special magic I/O so as to allow it to be observable again. Instead, starting from an observable state, it must always keep the possibility of evolving into an observable state, of "stabilizing", in a way that doesn't require the special knowledge by the external world.

Observability by itself is *not* a composable property: given an observable implementations Φ^{-1} of program P with abstract computation A, and Ψ^{-1} of A with concrete computation C, we can from an interrupted concrete state c' extract a stable abstract state a'', and from there an abstract program state p''' — but at that point, we haven't identified a concrete state c''' that has p''' as image, and so we do not satisfy the condition for an observable implementation of P with C:

$$\begin{tikzcd}
p \arrow[rrrrrrr, "P"'] & & & & & & & p''' \\
& a \arrow[ul, dashed, mapsto, "\Phi"'] \arrow[rrrr, "A"'] & & & & a'' \arrow[r, "A^s"'] & a''' \arrow[ur, dashed, mapsto, "\Phi"'] & \\
& & c \arrow[ul, dashed, mapsto, "\Psi"'] \arrow[r, "C"'] & c' \arrow[r, "C^s"'] & c'' \arrow[ur, dashed, mapsto, "\Psi"'] & & &
\end{tikzcd}$$

Now, if the notions of advancing for A and C are such that Ψ induces an implementation of A^s with C^s that is complete (see above subsection 3.2.2), then we can compose the observability to get a''' and p''' as previously, and use completeness to deduce a c''' from a''', thus fulfilling the requirements. Given suitable subsets A^s of A and C^s of C, the combination of observability and completeness is therefore composable:

$$\begin{tikzcd}
p \arrow[rrrrrr, "P"'] & & & & & & p''' \\
& a \arrow[ul, dashed, mapsto, "\Phi"'] \arrow[rrr, "A"'] & & & a'' \arrow[r, "A^s"'] & a''' \arrow[ur, dashed, mapsto, "\Phi"'] & \\
& & c \arrow[ul, dashed, mapsto, "\Psi"'] \arrow[r, "C"'] & c' \arrow[r, "C^s"'] & c'' \arrow[u, dashed, mapsto, "\Psi"'] \arrow[rr, "C^s"'] & & c''' \arrow[ul, dashed, mapsto, "\Psi"']
\end{tikzcd}$$

Note however that most programming languages do not provide a standard interface to declare safe points and safe arrows, and let users provide observability for the languages (whether domain-specific or general purpose) that they define on top of their existing platform. Thus, even though modern programming language implementations often go to great length to provide some notion of observability internally this notion cannot be exposed in a usable way to the end-user who will be using a program written on top of the platform.

Observability for correct programs

Observability for correct programs is a weaker variant, whereby the concrete system is allowed to go wild for abstract programs that do not satisfy some correctness property. Usually, this is formalized by just considering an implementation of the proper subcomputation of A.

Unhappily, some programming languages (such as the widely used C programming language [citation needed]) make it easy to inadvertently stumble upon "undefined behavior" [citation needed], at which point typical implementations will go wrong, and it is not possible to recover a meaningful abstract observable state anymore from the computation — what more, implementations of the same languages may make it hard to distinguish whether you're still observing correct computations, or are being fed garbage due to previous undefined behavior. When such a language, directly or indirectly, Observability and other useful implementation properties are only guaranteed if not otherwise triggering one of these cases of "undefined behavior".

Bounded Observability

Bounded Observability is a stronger variant of Observability, whereby we assume we have a notion of "length" of transition paths between two states, and we require that the exhibited transition path from c' to c'' be of length less than some maximum admissible response time, or more generally that it fits within some maximum pre-allocated amount of resources. Bounded observability is important for real-time applications, particularly in presence of global synchronization of concurrent threads. [citation needed]

Strong Observability

Strong Observability is a variant whereby up to we can associate to every state in C a unique "closest" state in A. Essentially, Φ can be extended into a total function such that every state in C is observable, e.g. by always forcing the evaluation to continue to the next "safe point".

Strong Observability is a simple and cheap property to require from a sequential (single-threaded) implementation of a deterministic programming language. But Strong Observability can be quite complex and expensive to achieve in a non-deterministic, concurrent and distributed system, where the branching structure of evaluation choices may be richer in the concrete computation than in the observable subset of the abstract computation it implements — which itself may be poorer than the branching structure of choices in the complete abstract computation, but in different ways. Some concrete states may then represent a choice between many valid abstract states, in a way that does not reflect a specific state in the abstract computation.

More formally, we associate to each reachable state c in C the subset of abstract states in A that may be observed in future executions in C^s starting from c, or equivalence classes up to reduction in A^s. We will call this set the set of interpretations of c, and we will call its elements interpretations of c. Observability states that for any c that is reachable from O, this set is non-empty. Strong observability is a stronger variation that requires that the set of interpretations of a reachable concrete state should not only be non-empty, but to have only one element up to equivalence. That is, for every reachable concrete state c', we can precisely identify an abstract state a' that corresponds to c', even if c' is not observable yet. It then becomes trivial to extend ϕ from O to the whole subset of elements in C reachable from O.

3.2.6 Contrasting Observability with Other Properties

Some previous properties, like soundness and completeness, only involved observable elements of the concrete system: they only concerned relative properties of O and A, notwithstanding any relationship between C and O; they were property of ϕ alone, independently of j. By contrast, liveness and observability really involve relative properties of O and C, notwithstanding any relationship between O and A; they were property of j alone, independently of ϕ.

Also note the temporal dissymmetry of Observability, Liveness or Completeness, whereas e.g. Soundness and Fullness were symmetric with respect to the direction of arrows. The notion of implementation is meant to formalize efficient (or at least adequate) means of executing abstract computations with more concrete computations. Observability is a property that is definitely oriented toward proper evaluation of computations, notwithstanding other properties of computations. If we wanted to capture the pure and perfect semantics of the abstract computation, we would just stick to the pristine source in the original language, or go “upwards” toward more abstract concepts; going “downwards” toward more concrete implementations is done for execution purpose only or mainly.

All in all, Observability is an essential notion for implementations, that involves the very essence of what to implement is about. It was the main contribution of our 1999 article[22].

3.3 Combining Implementations

Implementations being the arrows of a Category (see 2.1.2), they themselves constitute a Category the arrows of which are natural transformations. All the usual algebraic operations then also apply to the Category of Implementations, just as they did to the Category of Computations (see 2.3). Once again, we will focus on just a few operations that are particularly common or remarkable when applied to Implementations. Examining ways that implementations can be combined is most useful to build more complex implementations from simpler ones, or conversely to analyze a complex implementation into simpler parts.

3.3.1 Composition

Implementation Composition

A most common way to combine implementations is to compose them, and a most common way to analyze them is to decompose them. In our diagrams, this corresponds to introducing more implementations along the vertical axis. We saw in the previous chapter that Computations constitute a Category where Interpretations are morphisms (or, reversing the arrows, Implementations). The full diagram for the composition of Interpretations (and thus, Implementations) is as follows:

$$\begin{tikzcd}
 & & A \\
 & O_M^A \arrow[ur, "\phi"] \arrow[dr, hook, "j_\phi"] & \\
O_C^A \arrow[uurr, bend left, "\xi"] \arrow[ur, "\psi_o"] \arrow[dr, hook, "j_o"] \arrow[ddrr, bend right, "j_\xi"'] & & M \arrow[uu, dashed, "\Phi"'] \\
 & O_C^M \arrow[ur, "\psi"] \arrow[dr, hook, "j_\psi"] & \\
 & & C \arrow[uu, dashed, "\Psi"'] \arrow[uuuu, dashed, bend right, "\Xi"']
\end{tikzcd}$$

Formally, if $\Phi^{-1} = j_\phi \circ \phi^{-1}$ is an implementation of an abstract computation A with a middle computation M through observable subset O_M^A, and $\Psi^{-1} = j_\psi \circ \psi^{-1}$ is an implementation of middle computation M with a concrete computation C through observable subset O_C^M, then $\Xi^{-1} = \Psi^{-1} \circ \Phi^{-1} = (\Phi \circ \Psi)^{-1}$ is an implementation of A with C through an observable subset O_C^A, as constructed below.

O_C^A is the full subcategory of O_C^M such that node-wise $O_C^A = \psi^{-1}(j_\phi(O_M^A))$; j_o is the canonical embedding of O_C^A into O_C^M (itself a full category of C); and $\psi_o = {j_\phi}^{-1} \circ \psi \circ j_o$. If we define $j_\xi = j_\psi \circ j_o$ and $\xi = \phi \circ \psi_o$ then $\Xi^{-1} = j_\xi \circ \xi^{-1}$ is the implementation sought after. Checking that this construction behaves correctly on arrows (soundness) is left as an exercise to the reader, as the formal details of such demonstrations are not essential to this thesis; however, note that the fullness of j_ϕ is essential in establishing that the above construction for ξ actually leads to a functor.

Of course, in practice, we will use partial functions, elide the details about total functions involving an observable subcategory, and only write the simpler:

$$\begin{tikzcd}
A \\
M \arrow[u, dashed, "\Phi"] \\
C \arrow[u, dashed, "\Psi"] \arrow[uu, dashed, bend right, "\Xi"']
\end{tikzcd}$$

Now, most of the properties and conjunction of properties we previously discussed are *composable*: if Φ^{-1} is a implementation of A with M that has all properties in a conjunction, and

Ψ^{-1} is a implementation of M with C that has the same properties, then $\Phi^{-1} \circ \Psi^{-1}$ is an implementation of A with C that also has those properties. Therefore, Computations, equipped with additional features like advancing or length, where appropriate, also constitute a (somewhat different) Category, where the morphisms are Implementations having those properties. As for soundness, it's a property we require of all Interpretations, so the Category of Computations with sound interpretations is the same as the Category of Computations.

Decomposing Implementations

The above composition diagram is most often used with decomposition as an intent: the object of ultimate interest is the implementation Ξ^{-1} of some abstract computation A with some concrete computation C, where A and C are given as well as constraints on Ξ. When synthesizing Ξ, the intermediate entities M, Φ, Ψ are largely variable elements that the programmer chooses at his convenience, to simplify the problem of achieving his goal. When analyzing Ξ, establishing properties of the intermediate entities M, Φ, Ψ is just intermediate goals in establishing those properties for Ξ.

This decomposition can be repeated many times. Indeed, when compiling a programming language, the compiler is often broken down in multiple *passes*; passes, or pairs of analysis and transformation passes, can often be thought of as implementations (in our sense), where at each slice, the computation (considered up to non-advancing rewrites) is being implemented with a somewhat lower-level computation. These passes together constitute a tower of implementions the composition of which implements the abstract program with the concrete machine. "Nanopass" compilers[citation needed] emphasize the process by decomposing the same overall semantics in more passes; adding more passes allows to zoom in on specific details of the implementation, making it easier to write individiual passes and to reason about them.

Given a set of composable properties, and a decomposition of an implementation as a composition of simpler implementations that we'll call *passes*, we can analyze each pass and prove that is has each property, then deduce that the overall implementation has all those properties. If the passes and the properties are each simple enough, the reasoning should be trivial for most pairs of pass and property — only in cases where the pass does something subtle with one aspect of the computation will an according property require non-trivial analysis, at which point the decomposition will hopefully keep that analysis relatively simple. Often, it may happen that all passes but one satisfy a property (for instance, completeness or totality may fail at a pass mapping object graphs to memory, due to out-of-memory error); but then, at least failures to satisfy the property has been narrowed down to one cause with a well-identified failure mode, and if that particular failure isn't observed, users can be confident that the property could be relied upon so far.

Composing up: Writing Programs

Now, there is a completely different way of looking at the very same diagram: instead of starting from an existing abstract computation A and concrete computation C and adding an intermediate computation M in the middle, we can start from an programmable abstract computation A (i.e. language, or configurable application), and a concrete computation C and add a program P *on top*:

$$\begin{tikzcd}
P \\
A \arrow[u, dashed, "\Phi"'] \\
C \arrow[u, dashed, "\Psi"'] \arrow[uu, dashed, bend right, "\Xi"']
\end{tikzcd}$$

With this change in point of view, A and C are given as well as the implementation Ψ^{-1} of A with C, and we are exploring the programs implemented on top of the programming language or programmable system A. Then it becomes apparent that the interesting properties that we are interested to have in implementations require not just Ψ to possess them, but also Φ, which in turn requires A as a programming language to expose suitable interfaces for the programmer to implement and provide said properties.

Now, most programming languages lack any interface to the useful properties that an implementation may have. Providing such an interface is the topic of the next chapter. Then, casual users, rather than advanced compiler writers, can also ensure that their program P has an implementation Ξ^{-1} with the properties they desire. Otherwise, to achieve the same effect they'll have to use ugly design patterns to implement on top of A a virtual machine V that provides the hard way all the properties they wanted from Ψ that were not exposed, then use them to implement P awkwardly on top of that.

Composing down: Virtualizing

By symmetry, a third way of looking at the diagram, of course, is as be adding an arrow *down*, where A and C are known, and you add a virtualization layer V below C:

$$\begin{tikzcd}
A \\
C \arrow[u, dashed, "\Phi"'] \\
V \arrow[u, dashed, "\Psi"'] \arrow[uu, dashed, bend right, "\Xi"']
\end{tikzcd}$$

In this final point of view, some existing program has been implemented on some concrete machine C, but that machine is virtualized, and itself implemented on top of an underlying virtualization layer V. Note, that this point of view may apply whether or not C was explicitly called a "virtual machine" or not before this virtualization happened: there can be direct hardware implementations of what was previously considered a "virtual machine" bytecode, and there can be software implementations of what was previously meant as hardware instructions, with all kinds of mixed strategy in between. And of course, V can itself be implemented "directly" in hardware, or virtualized and implemented in any kind of way whatsoever — and so on, *ad libitum*.

When virtualizing, the virtualization layer takes care of implementing the user-visible inputs and outputs. Additional outputs may be added, for instance to logging usage of some resources, or to track modifications to some variables being watched while debugging. The evaluation may be interrupted when some condition is detected, whether again for debugging, or for lack of resources, or for security reasons. Inputs may be added to control this additional output only. But if the behavior of the program is modified, then what you have is not an implementation of the program, it's an implementation of a different, modified program, or of a larger program that includes (parts of) the initial program (see below) — which is also an interesting and related thing, but a different thing. An implementation as such may not modify the higher-level behavior.

If the upper layers of the program implementation, i.e. Φ, do not offer some useful property, then the overall implementation, i.e. Ξ, can't provide it. But that doesn't mean that the lower layers, i.e. Ψ, can't or shouldn't provide such a useful property: it might still be relied upon by users to provide some weaker overall property, and it might still be useful to have these properties, even at the level of abstraction of C only rather than A. For instance, Φ might not be total nor complete, but you might still want Ψ to be total or complete, to not add new unwanted and unhandled failure modes. Or, Φ might not have real-time liveness in general, but fragments of it may have it, and having Ψ provide it may then be essential to preserving the meaning of these fragments. Moreover, Observability may be used to enable process migration or replication at a low-level, even where these features cannot be provided at a higher-level of abstraction.

3.3.2 Control

There are many ways of composing or decomposing implementations that in our diagrams do not add more implementations on the vertical axis, but combine or modify what happens along the horizontal axis, i.e. defining the semantics of the computation. We unhappily do not have very good diagrams for these operations, but happily building and implementing computations with interesting and/or appropriate semantics is already a well-known topic, that of programming languages. We will thus focus on those ways of composing and decomposing implementations that do not involve building new semantics, but preserving existing semantics, yet in useful and relevant ways.

Inclusion

A most obvious way for the semantics of an implementation Φ^{-1} to be preserved when one instead considers an implementation ${\Phi'}^{-1}$ is when ${\Phi'}^{-1}$ is included in Φ^{-1}. If Φ^{-1} is an implementation of A with C and ${\Phi'}^{-1}$ is an implementation of A' with C', then A' is included in A, C' is included in C, and Φ' is included in Φ.

The restricted scope of C' as compared to C can be used to model many things: the respect of some kind of local invariant or scope; use in normal (or correct) situations as opposed to exceptional (or erroneous) situations, or vice versa; some context that doesn't change often during execution; the temporary, expected, desired, or necessary satisfaction of some constraint;

etc. C may contain additional nodes and arrows as compared to C': the local invariant may be violated and the scope exited; the situation may change from normal to exceptional and vice versa; the context may eventually change; the constraint may be broken; etc. The computation C may thus be much larger than C'.

A particular use of inclusion is to allow "administrative" rewrites between equivalent nodes, that do not usually happen during normal execution, but may be forced from an outside controller (see 8.3) e.g. to achieve completeness. Thus for instance, in a non-deterministic program, the implementation left to run without intervention would follow some heuristic for choosing the execution path, and that would be C', but it actually also allows some external agents to override this heuristic or force different choice.

Many of the desirable properties listed in the previous section need only be proven for either one of Φ^{-1} or Φ'^{-1} (depending on the property) and be easily deduced for the other. On the other hand, the two implementations will usually have been distinguished precisely because one will have properties that the other has not: For instance, it may be that the "same" implementation has real-time behavior during normal use as modeled by C' having real-time liveness but not in the general case exceptional situations may arise (e.g. packet drop, disconnection, power loss, etc.) such that C as a whole does not have that property.

Selection

Selection is particular case of multiple inclusion, whereby the sum of a family of implementations $({\Phi_x}^{-1})_{x \in X}$ is included into a common implementation Φ^{-1} of A with C. (Note that, in the terms of Category Theory, this sum, being constrained, is a pushout, rather than a co-product.) Additional arrows may then allow the computation to select between these sub-implementations. If these arrows depend on external input, the selection is externally controlled by an outside process; if the arrows do not depend on external input, the selection is internally controlled by a algorithm inside the implementation itself.

This models the fact that at every moment that the computation A is being implemented by one of the sub-implementations ${\Phi_x}^{-1}$ in the sum, that sub-implementation has been selected to implement the current state of computation. But, with additional arrows, at some times the computation may switch to another implementation ${\Phi_y}^{-1}$ (and, with additional nodes, if the inclusion is not nodewise surjective, the computation may switch to parts not included in the sum). Such change in the current sub-implementation may model situations in which the previous choice of sub-implementation is replaced by another that is better adapted to a modified situation, one that is better optimized, or one that somehow corresponds to a temporal decomposition of an overall computation in several local phases, or one that reflects a change in the execution environment.

In such a setting, we are usually interested in establishing properties of the complete implementation Φ^{-1}, by studying the partial implementations ${\Phi_x}^{-1}$, and the additional transition arrows.

Selection is a very useful tool, that allows to focus on essential issues at hand when synthetizing or analyzing the local behavior of an implementation. One notable use of it is to model migration, as described below in chapter 6.

Stopping

Given an implementation Φ^{-1}, the implementation ${\Phi_{stopped}}^{-1}$ is defined as $\Phi_{stopped}$ having the same set of observable nodes as Φ, but having no non-trivial arrows (i.e. no arrows besides the identity arrows). There is trivially a natural transformation from $\Phi_{stopped}$ to Φ and there is a forgetful functor that to Φ^{-1} associates the stopped implementation ${\Phi_{stopped}}^{-1}$. Once again, the direction that preserves structure is contravariant to the intent-following notation: from C to A, from $\Phi_{stopped}$ to Φ. Note that $\Phi_{stopped}$ has no liveness property at all — and that's the whole point: from an evaluation point of view, it's stopped.

A *stoppable implementation* is a selection (as in 3.3.2 above) of an implementation and its stopped implementation, where the co-product is extended with transition arrows labelled by two distinct actions "stop" and "resume" between homologous nodes. These transition arrows are labelled so an external computation (scheduler or other metaprogram) can thus stop and resume the (concrete) computation. There is an obvious functor from implementation to stoppable implementation.

Timesharing

Stoppable implementations can notably model timesharing, by which an operating system implements an arbitrary number of concurrent processes on a monoprocessor or a multiprocessor with a fixed number of processors: given for each process an implementation of the process on the machine, consider the constrained product of the corresponding stoppable implementations (i.e. in Category Theory, a pullback, where the constraint is that inter-process communication and other interactions are properly identified); then add the restriction that among the implementations in the product, only one may be running at the same time per processor available, including the scheduler itself, a privileged additional computation in the product.

In *cooperative* multiprocessing, scheduler computations may only happen when the computations execute some special *yield* actions; in a *preemptive* multiprocessing, scheduler computations may happen at any time via *interrupt* transitions.

Runtime Metaprogramming

Beyond timesharing, stopped implementations can also model debugging, or any kind of control of an implementation by some kind of runtime metaprogram: a scheduler, a debugging monitor, a container (such as a hypervisor), a kernel, etc.

The runtime metaprogram can stop the computation, examine it, resume it, determine the strategy according to which to resolve some non-deterministic choices, maybe make it go back in time and follow different choices. It may also contain or use additional transitions, corresponding to the computational content of observability, completeness, liveness, etc.: all of these transformations trivially preserve the computation's semantics. However, the metaprogram may not modify the computation, otherwise it's not a correct implementation of the computation anymore — though it may be a correct implementation of some super-computation, if *that* is what it may have been specified to preserve.

As with selection above, the metaprogram can be an external computation or can be part of the computation itself: An external metaprogram would control its concurrent computations via input or output interactions triggering the stopping and resuming of each individual computation's stoppable implementation, and other transitions within, between or across implementations. An internal metaprogram would do as much, but the transitions would be internal without involving externally observable input or output.

3.3.3 Concurrency

Observability as a Synchronization Primitive

In the paper in which we originally proposed the current formalization of implementation[22], we explained how the computational interpretation of the observability property was as a synchronization primitive, and how it could be used to model essential phenomena in the implementation of concurrent and distributed systems. Here is a restatement of this approach.

Consider an abstract concurrent computation A made of several communicating parallel subcomputations A_i. In categorical terms, A is a pullback of the product $\Pi_{i \in I} A_i$, where various states or interactions have been identified in the way natural for A. We want a "modular" implementation of A, based on combining separate implementations ${\Phi_i}^{-1}$ of each A_i with a concrete computation C_i. However, for parallelism to mean anything at all, we want to *run* the concrete computations C_i in parallel on the given underlying processor architecture — in other words, to use the implementation for running the computation, we can't arbitrarily choose which states or interactions we can identify between the C_i, and which pullback we get, we have to use *the* pullback of the C_i where the identifications are implicit from the common algebra in which they are written, as embodied by the computer architecture on which they are meant to run.

Now, if the threads run independently, the odds that any one C_i will be in an observable state at a given moment are low, and for any non trival number of factor computations, the odds that all relevant factor computations C_i will be in an observable at the same time are vanishingly small. This is not a problem as long as any interactions that are atomic between the A_i are also atomic between C_i for the implicit way that the C_i are run on said underlying architecture; then, the product $\Pi_{i \in I} C_i$ implements A. But what if the interactions between the subcomputations A_i are not directly expressible as a pullback in terms the underlying common architecture in which the C_i are to run?

The solution is that we can run the independent fragments of C_i, those that don't involve the non-atomic "high-level" communication events, as separate *threads*, and have a special activity, a *kernel*, implement the communication events. For these high-level communication events to happen, all threads involved may have to respect all the suitable high-level invariants — in other words, they must each be in an *observable* state. To get the relevant individual threads in observable state, the kernel K will be able to *stop* and *restart* each individual thread (as in 3.3.2 above); when a thread is stopped, the kernel can *observe* it, that is, synchronize it in an observable state; and when all relevant threads are in observable state, then the kernel can enact the communication, in a way that appears atomic to each of these threads.

Thus, in the general case, to implement a (parallel) product of computations, we need more than the product of their implementations, but a product of their stoppable implementations *and* a communication kernel. The set of state of the system will thus be a pullback not of $\Pi_{i \in I} C_i$ but of $K \Pi_{i \in I} (C_i + C_{i,stopped})$.

Observability in Existing Concurrent Systems

Synchronizing to a safe point is a well-known issue faced by concurrent implementations of computing systems, whether it's for the sake of garbage collection, database transactions, snapshotting, process migration, code or data schema upgrade, non-local communication, precise accounting, or any other global operation that depends on the high-level invariants of the system being preserved. The software implementation of such synchronization to a safe point dates back at least to the 1960s with ITS [7] and its "PCLSRing" [3]. When running any kind of user-provided code, synchronizing to a safe point can also be a matter of security: if code meant to run while safety invariants are satisfied is actually allowed to run when they are broken, it can result in catastrophic behavior. See for instance how the recent attack on the ethereum DAO cost tens of millions of dollars in damages. [citation needed]

Observability is a formal way to reason about this issue and to address it in a systematic and modular fashion.

Multiple Observability Properties

When only a subset of the components are involved in a particular abstract operation, only these components must be coherently stopped in an observable state for the implementation of the abstract operation to be possible. Several operations on disjoint subsets of components may even be attempted in parallel. Hence, there are as many useful notions of partial observability as there relevant sets of components to be synchronized.

What more, when there are several composed layers of implementations, or when there is an inclusion or selection of implementations, then even for the same subset of components of the same system, there may be many distinct observability properties, all of them relevant.

For each way to observe a same concrete system as the implementation of some more abstract system, there corresponds a different notion of observability that may be used for synchronization while implementing that more abstract system. For efficiency, the kernel may then be able to use as weak an observability property as required by the considered abstract operation, but no weaker.

Part II

First-Class Implementations

Chapter 4

From Semantics to Protocol

4.1 Computing with Categories

4.1.1 Intent

From our diagrams about the properties of implementations in general, we extract a runtime protocol to interact with such implementations at runtime. This illustrates how theoretical considerations can have direct practical applications. The extraction is type-directed, as per the Curry-Howard isomorphism[citation needed]: the protocol is the *computational content* of the properties we presented.

First, we formalize these properties in a dependently-typed system. We chose Agda as our formalization language, because of its relative simplicity and the fact that thanks to its type system having dependent types, it is simultaneously a programming language and a proof verification environment. We could as well have used Coq or Idris.

Second, we can omit logical constraints, that in Agda are resolved at compiled time, and approximate away type dependencies, that cannot be expressed in languages with less expressive type systems. This yields types that can be used to specify the protocol functions in these languages. Depending on the language, various approximations must be consistently used to extract different variants of the protocol.

In this section, we will show how to can extract a runtime protocol from our formalization of computations, yielding first-class computations. Then in the next section, we can apply the same technique to extract a runtime protocol from the more elaborate formalization of implementations, yielding first-class implementations.

4.1.2 Categories

A Category can be formalized as a record of the following items:

- A set `Node` of the nodes of the category.
- For every pair of nodes `A` and `B`, a (possibly empty) set `A` $\Longrightarrow$ `B` of arrows.
- For every node `A`, an identity arrow `id A` in `A` $\Longrightarrow$ `A`.
- A composition function $\circ$ that composes compatible arrows.
- Logical laws: `id` arrows being identity (left and right), and composition being associative.

In the Agda library the definition for a Category could be as follows:

```
record Category (n a : Level) : Set (suc (n ⊔ a)) where
  field
    Node : Set n       -- nodes, at level n
    _⟹_ : Rel Node a  -- binary relations between nodes, at level a
    id : ∀ {A} → (A ⟹ A)
    _∘_ : ∀ {A B C} → (B ⟹ C) → (A ⟹ B) → (A ⟹ C)
  ...
```

Note how in Agda we had to explicitly deal with type hierarchies necessary to avoid the Russel Paradox: `n`, `a` and `(suc (n ⊔ a))` are manually managed type levels, so that a category lives at a level strictly above that of both nodes and arrows. The underscores help declare infix syntax, and the curly braces declare implicit arguments that can usually be omitted and inferred from the context.

Also note that in the actual Agda library `categories`, the `Node` type is actually called `Obj` (and its level `o`), to reprise the common mathematical terminology where nodes are called objects, and arrows are called morphisms (or homomorphisms).

The Agda definition also includes logical laws, as additional parameter fields containing proofs of the required properties:

```
    identityˡ : ∀ {A B} (f : A ⟹ B) → (id ∘ f) ≡ f
    identityʳ : ∀ {A B} (f : A ⟹ B) → (f ∘ id) ≡ f
    assoc : ∀ {A B C D} {f : A ⟹ B} {g : B ⟹ C} {h : C ⟹ D} →
                (h ∘ g) ∘ f ≡ h ∘ (g ∘ f)
```

4.1.3 Extraction

Erasing dependent types

From specifications, we can *extract* programs that embody their *computational content*, by erasing proofs and types that are not relevant at runtime. While Coq has popularized automated program extraction[citation needed] based on formal rules, we will manually and informally extract an API from the previous specification of implementation properties.

Not to introduce an extraneous programming language, the target language of our informal extraction will be a subset of Agda without dependent types, yet with type parameters. The result should be easy to map to any sufficiently advanced programming language such as Haskell, OCaml, Scala, Java, C++, etc. Well-known techniques could also further adapt the code to languages with less advanced static type systems (by replacing every parametrized type with one type or a finite enumeration of types), or to "dynamically typed" languages without any non-trivial static type system (by erasing types altogether).

We leave it to the readers to adapt our protocol to the language of their choice: what matters is that the computational content of the functions in our protocol will remain the same: one function returns the domain of an arrow, another function advances a computation, etc. Thus, our initial specification will have made it possible to identify a coherent set of functions that can be used together to express things not previously possible.

Extracting Categories

Without dependent types, and when nodes represent values that can change at runtime within a type `Node`, the type for arrows in a category cannot depend on the values of nodes. Therefore, for each category, there will be a type `Arrow` for all the arrows in the category. A category representing a runtime computation would therefore be a record of just the following types and functions:

```
record &Category : Set₀ where
  field
    Node : Set₀
    Arrow : Set₀
    &id : Node → Arrow
    &compose : Arrow → Arrow ↛ Arrow
```

Note that in this text we name our extracted functions with an ampersand `&` prefix to clearly distinguish them from the functions in our original specification on an obvious syntactic basis. The extraction in some other language may eschew that prefix (that might not be syntactically valid in that language), since there would be no clash with identifiers in this specification; on the other hand, it might also rename functions to avoid name clashes with standard library functions or reserved keywords. We will usually omit the ampersand prefix for types, that will have different names and shapes.

Also note how `&compose` returns an `Arrow`, but is a partial function, denoted by ↛: the result is only guaranteed to make sense when the right arrow's codomain is the left arrow's domain, which cannot be enforced by the type system when simplifying away dependent types. The way to express that with dependent types is through type constraints between arguments, and extra arguments as witnesses of additional logical constraints. In the absence of dependent types, several solutions may apply. We could use a total function; but then `&compose` would have to return some non-sensical result when that's not the case. We could have made it return the option type `Maybe Arrow` and use the value `nothing` when the arrows fail to compose; then we have to constantly "lift" functions that process such results to handle that case. This lifting could be implicit in a language where or where the `Arrow` type implicitly (or explicitly) includes a null case. Finally, in a language where functions can have side-effects such as throwing exceptions, these might have been used instead, which we could denote using `--→`.

Runtime checking

Since without dependent types the logical correctness of programs cannot usually be proven at compile-time, it would befall on the programmers to that they only do operations on compatible nodes and arrows. One solution would be to first write their program in a dependently-typed language that ensures safety, then extract it; another would be for the programmers to check at runtime that nodes and arrows operated on are indeed compatible. The check could be done either by those who implement the categories (which is safer), or by those who use it (which is more efficient, but unforgiving of mistakes they might make). To check whether two arrows can be composed, the domain and codomain may be determined at runtime, using the below functions:

```
    &domain : Arrow → Node
    &codomain : Arrow → Node
```

The two functions could have been *defined* as follows in the original dependently-typed protocol, rather than be required function parameters to be provided as part of the category implementation:

```
    domain : ∀ {A B} → (A ⟹ B) → Node
    domain = λ {A B} f → A
```

```
codomain : ∀ {A B} → (A ⟹ B) → Node
codomain = λ {A B} f → B
```

Furthermore, if, lacking sufficient type abstraction, the types `Node` and `Arrow` have to be shared across categories, then functions that use nodes and arrows may have to check that they are in the same category before they may be somehow used together. This can be done using the following functions as part of the `&Category` signature whereby a category can recognize which nodes are part of it[1]:

```
&contains-node : Node → Bool
&contains-arrow : Arrow → Bool
```

Another (non-exclusive) approach could be that nodes and arrows know which (principal) category they are part of, which however precludes these nodes and arrows being part of more than one:

```
&node-category : Node → &Category
&arrow-category : Arrow → &Category
```

4.1.4 First-Class Computations

Computations

Now, when we consider categories as operational semantics, then a node ("object" in categorical terms) is a state of a computation, and an arrow ("morphism" or "homomorphism" in categorical terms) is a labelled state transition.

As a first-class object, a node may therefore encode the captured state of a light-weight thread in some high-level programming language or its virtual machine: the registers, stack and heap or a low-level program, the continuation of a scheme program (and also its store if it's stateful), the S, E, C, D registers of a SECD machine, etc. It may also encode the frozen state of a separate operating system process being debugged, or otherwise attached e.g. with the Unix `ptrace(2)` system call.

An arrow is then a first-class frozen representation of a change that may happen, that takes one from one state to another. For instance, it would record that some registers and memory addresses were modified from one value to another, and that some I/O operations happened. You could deduce the arrow from a complete trace of every change that happened between those two states, by summarizing those changes that matter to the computation at hand.

In an analogy to physics, one can see a node as a point in phase space that summarizes (what can be observed of) the state of the system. It is a tiny point in a huge space, that grows exponentially with the informational content in each node. An arrow is then a path between two points along a trajectory of allowed system evolution; or rather, it is an equivalence class of such paths that are indistinguishable to the observer. Observation can equate many nodes, and its uncertainty can introduce non-determinism; or the non-determinism can equivalently be considered as part of the system. If there was observable input of matter, energy or information from outside the system or output of same to outside the system, then it will be recorded as part of the arrow; similarly if there were turns around a topological singularity or whatever other observable physical phenomenon.

What distinguishes a computation from a random category, though, is that somehow these categories are the operational semantics of some computations, that can be actually *run*.

[1] These two functions satisfy the following identity:
`∀ {node} → &contains-node node ≡ &contains-arrow (&id node)`

Running Computations

The Category protocol so far allows one to express first-class representations of computation states and of transitions between pairs of such states. Now the point of having a computation is for it to do the computing for you, not for you to tell it what transition to examine and effect and when. Therefore, to *run* the computation, one must have some mechanism of starting from the current state and having a computer walk as far as it can through these transitions. To embody a first-class computation, a category must therefore possess a function `run` that given a starting node `A`, "runs" the first-class computation for some time, and returns (the dependent product of a node `B` and) an arrow `A ⟹ B`.

```
run : (A : Node) --→ ∃ (λ {B : Node} → A ⟹ B)
```

The signature for an extracted function in a non-dependent setting would be as follows, where the node `B` can be omitted since it can be deduced from the arrow as its domain:

```
&run : Node --→ Arrow
```

Now, notice the above use of a dashed arrow. Running the computation may take an arbitrary amount of time; it may be non-deterministic; it may not terminate; it may involve communication with other computations; it may or may not have other side-effects. For all these reasons, it cannot be a simple function, but must be some effectful computation. The arrow `--→` therefore represents native functions with general side-effects in the meta-language that manipulates the first-class computation. It is therefore an arrow at the meta-level (more precisely the *ante*-stage). Implicit in the notion of first-class computation is therefore the notion of evaluative reflection, which we'll explore in the next section (4.1.5). In Haskell, for instance, `x --→ y` could be extracted as the Kleisli arrows `x -> m y` for some suitable monad `m`, typically the `IO` monad. [citation needed] In a meta-language with unrestricted side-effects, such as an ML dialect, it would be that meta-language's regular function type `x -> y`.

Notice also how the signature for `run` says nothing about when that function stops. It may return immediately without effecting any transition and without advancing in any way. It may never return, vainly trying to complete a computation that loops forever and never stops. It may return an intermediate step, stopping after a while for whatever reason. Or it may run the computation to completion until it returns with a result. While the informal contract of the function may be to run until either the program completes or some meaningful interruption happens, the current type does not say anything; a model of what advancing, completing or interrupting means is necessary to improve on that type.

Advancing Protocol

We can express progress in running the computation with some notion of advancing, as per section 3.2.3. We can formalize this notion with functions that can decide whether an arrow is advancing or non-advancing, and whether a node is done or not-done, where it is done if and only if there is no advancing arrow with it as domain:

```
advancing? : ∀ {A B} (A ⟹ B) → Bool
done? : (A : Node) → Bool
```

For reasoning, we can define the following predicates:

```
advancing : ∀ {A B} (A ⟹ B) → Set a
advancing f = (advancing? f ≡ true)
```

```
not-advancing : ∀ {A B} (A ⟹ B) → Set a
not-advancing f = (advancing? f ≡ false)
done : ∀ {A B} (A ⟹ B) → Set a
done f = (done? f ≡ true)
not-done : ∀ {A B} (A ⟹ B) → Set a
not-done f = (done? f ≡ true)
```

The following laws hold for `advancing`:

```
advancing-absorbsˡ : ∀ {A B} {f : A ⟹ B} advancing f
                      {C} {g : C ⟹ A} → advancing (f ∘ g)
advancing-absorbsʳ : ∀ {A B} {f : A ⟹ B} advancing f
                      {C} {g : B ⟹ C} → advancing (g ∘ f)
identity-not-advancing : ∀ {A} → ¬ (advancing (id A))
```

And the following law relates `advancing` and `done`:

```
done-iff-cannot-advance :
   ∀ {A} (done A) ↔ ∀ {B} (ar : A ⟹ B) → (not-advancing ar)
```

Then we can specify the functions `step` and `advance` as follows, where `step` (wherever it is defined) deterministically advances one step, whereas `advance` (wherever provided) non-deterministically advances some unspecified amount, depending on the particulars of the implementation, on interruptions, etc., which side-effects are represented using the meta-language arrow `--→`:

```
step : (A : Node) →
  ∃₂ (λ {B : Node} (ar : A ⟹ B) → (done A) ⊎ (advancing ar))
advance : (A : Node) --→
  ∃₂ (λ {B : Node} (ar : A ⟹ B) → (done A) ⊎ (advancing ar))
```

The simplified API function in a non-dependent setting would make advancing and done runtime predicates returning a boolean, and would omit `B` and the correctness proofs:

```
&advancing? : Arrow → Bool
&done? : Node → Bool
&step : Node → Arrow
&advance : Node --→ Arrow
```

Evaluation

Given the notions of advancing and done above, we can define a function with the following type, that will keep running and advancing the computation until it completes and reaches a state where it is done:

```
eval : (A : Node) --→ ∃₂ (λ {B : Node} (ar : A ⟹ B) → done B)
```

The extracted API function would simply have the following type:

```
&eval : Node --→ Arrow
```

Note how `&run`, `&advance` and `&eval` have the same type: without dependent types, the three cannot be distinguished at compile-time merely from their signature; what distinguishes them is the logical guarantees about the result they return; more variants could be produced, for instance, to take into account time and other resources.

4.1.5 Evaluative Reflection

Performing vs Simulating

In the above protocol to "run" a computation, we kept manipulating and returning first-class nodes and arrows, and didn't *actually* run the code and have it do its side-effects. In some way, it was a simulation, not the "real thing". Of course, if the computation involves no actual side-effect, and if the simulation is efficient enough, there is no difference. But if the computation involves having actual effects on the world, such as sending messages to machines and people outside the system, printing documents, controlling a robot, effecting monetary transactions, or launching nuclear missiles, then clearly there is a difference between simulating a computation based on a first-class representation and actually running the computation on the actual computer system it's supposed to run on.

Inherent in the notion of first-class computation, therefore, is the fact that these first-class computations, as data items manipulated or reasoned about by some meta-system, can be instantiated as code into actual computing system, that will run with actual side-effects. In other words, first-class computations implicitly assume what we called *Evaluative Reflection* in section **??**. Whatever program manipulates a first-class computation is an *ante*-computation, and these first-class computation *simulate* computations that an actual computer will presumably *perform* later, with actual side-effects, as a *post*-computation.

Formalizing Evaluative Reflection

When describing comptations, we use two kinds of computation arrows. First, the double-arrow $\Longrightarrow$ denotes first-class computations seen as data recording potential changes and side-effects; being data, it's mostly inert by itself, and its meaning fairly independent of the meta-system where the ante-computations take place. Second, the dashed-arrow $\dashrightarrow$ denotes actual computations seen as code to be performed with actual side-effects; being code with side-effects, its precise semantics may vary a lot depending on the meta-system used for ante-computations. (As for the regular arrow $\rightarrow$ it will keep denoting both logical implication and pure total functions without side-effects — equivalently as per the Curry-Howard isomorphism.)

The difference between the two kinds of computation arrows is largely invisible at the level of logic: all that logic ever manipulates is data, anyway, and from a logical point of view, both first-class and actual computations have the same semantics. But there is a lot of difference from the point of view of program extraction. Notably, by the time a computation was actually performed and had actual irreversible effects on the world, it is too late to undo what is done, and it is too late to record the computation if it hadn't been recorded already. The "same" program may at times be seen from different points of view, depending on the context.

Performing Computations

The first and foremost function that is implicit in any formalization of an actual computation is thus *perform*: the function that takes a recording of a computation and makes a computing device actually perform the computation with all its side-effects. The formalization is useless and calling what was formalized a computation at all is dubious unless such function exists in one form or another, inside the system or in a separate meta-system: it's not a computation if you can't perform it.

Now, `perform` is actually cofunctorial, and has both node-wise and arrow-wise components. Its signature involves the type `State`, of the states of the operating system processes, programming language threads, or virtual or real machines that are to run the code for the current computation; this type `State` may depend on the Category of computation being performed, as well as on the machine on which it is performed. It can be largely opaque to a lot of users of the protocol, who only care that the computation has the desired observable effect, and aren't otherwise interested in the details when they perform the effects.

```
perform.● : Node → State
perform.⟹ : ∀ {A B} → (A ⟹ B) → (State --→ State)
```

In other words, the node-wise component of perform instantiates a first-class representation of a computation node into a actual computation state; and the arrow-wise component of perform actually realizes all the changes encoded in a first-class representation of computation arrow into an actual computation change with all its actual side-effects. Extracting the computational contents, we have the simpler:

```
&perform.● : Node → State
&perform.⟹ : Arrow → (State --→ State)
```

Simulating Computations

Now, the inverse function simulate isn't implicit in the formalization of computation. Most implementations of a computational system don't provide it. Indeed, it's usually impossible to express this function without virtualizing the entire machine, because there is otherwise no way to capture all the side-effects of all the libraries that your code may indirectly depend upon. It is functorial, and the signature of its components are:

```
simulate.● : State → Node
simulate.⟹ : State → (State --→ State) --→ ∃_2 (λ {A B} → A ⟹ B)
```

The extracted computational contents are:

```
&simulate.● : State → Node
&simulate.⟹ : State → (State --→ State) --→ Arrow
```

In other words, the node-wise component of simulate takes the state of a stopped process and freezes it as a first-class object that simulates that state in sufficient detail to perform it in the future, or otherwise reason about it. And the arrow-wise component explicitly takes an initial state as well as a function to transform that state with side-effects, and returns a record of the internal changes and external side-effects that running this function would have on the state, without actually performing those effects.

Note however, how simulate.⟹ in general itself may have side-effects: the computation being simulated itself may have side-effects, and when simulating them without performing them, simulate.⟹ has to build a model of what these effects will be or can be; this model may be a precise or approximate formal model, may consult an oracle of what inputs will be and outputs will do, may do one or multiple runs with random inputs, may fail or stop for further instructions if some unmodelled, unauthorized or otherwise watched effect happens. Thus simulate.⟹ is an effectful function from State to State, though its effects are *not* those of the input arrow, but those of simulating it without actually performing it.

It is sometimes useful to actually perform the effects and merely record which they were; in those case, a specialized version record of simulate may be used, that relies on a trivial oracle to actually perform the effects (as long as recognized and authorized) instead of trying to predict what they will be and do. As compared to just performing the computation and returning the final state, record, just like simulate in general, also returns a record of all the effects that happened on the way from the initial state to the final state.

Implemented Computations

For every function that computes on first-class computations and returns first-class computations, there corresponds evaluatively reflected variants that do notionally the same thing, but rely on actually performing the computation rather than playing with representations.

Informally, these variants correspond to compiling a program and running it as code in a post-computation, rather than interpreting the program by manipulating it as data in the ante-computation. Of course, a valid "compilation" strategy is always to ship an interpreter with the program being interpreted; but indeed, then the interpreter becomes part of the post-program doing the actual running, rather than of the ante-program reasoning about the running.

When there is a protocol that ensures that a computation can be actually run, we will say that the computation is *actually implemented.* If the computation is a selection of all the programs in a programming language, then the language is *actually implemented.*

Reflected Protocol

Let us suppose that the node-wise components of `perform` and `simulate` are simple and cheap, and that instantiating `Node` or `Node` into a `State` or extracting an `Node` or `Node` back from a `State` is trivial. This is especially the case in a monadic or linear setting, where there is only one active `Node` or `Node` at a time, and it is already embedded in a `State` when used. Then, we can abstract away any underlying `State` and have an interface that works directly on `Node`.

Now, importantly, when extracting protocol functions, instead of returning arrows, they are immediately performed. Thus, using `!` as prefix for such reflected interfaces, the corresponding functions would be:

```
!run : State --→ State
!step : State --→ State
!advance : State --→ State
!eval : State --→ State
```

The previous functions `&run`, `&step`, `&advance`, `&eval` could then be seen as instrumented variants that do not effect the changes but capture them into an arrow with some variant of `instrument`, only simulating the computation. Meanwhile, the evaluative reflection protocol allows users to both "actually" and "directly" run the code, rather than to either merely simulate it, or having to go through the indirection of explicitly instantiating a separate process.

4.2 A Protocol for Implementations

4.2.1 Soundness Protocol

Having seen how to extract a protocol from a specification using the Computations as an illustration, we can now apply the same to Implementations.

An implementation of an abstract hyper-computation with a concrete concrete hypo-computation is first the datum of a partial functor of the concrete computation to the abstract one. The functoriality automatically gives us soundness:

```
record Implementation (ao aa co ca : Level) : Set (suc (ao ⊔ aa ⊔ co ⊔ ca)) where
  Abstract : Category ao aa
  Concrete : Category co ca
  interpret : PartialFunctor Concrete Abstract
```

This partial functor from Concrete to Abstract is the same as a (total) functor from a full sub-category Observable of Concrete to Abstract. A PartialFunctor, like all functions, be they total or partial, deterministic or non-deterministic, will define a relation ↦ between objects in their domain and those they map to in their codomain. A PartialFunctor, like all functors and profunctors, will also define a relation ⤇ between arrows in their domain and those they map to in their codomain.

It can be extracted as follows, where X ⇸ Y denotes a partial function from X to Y, which could be expressed as X → Maybe Y or something equivalent in the target language; such a partial function f would define a relation f.↦ between objects in X and in Y:

```
record &Implementation : Set₀ where
  &Abstract : &Category
  &Concrete : &Category
  &interpret.• : &Concrete.Node ⇸ &Abstract.Node
  &interpret.⟹ : &Concrete.Arrow ⇸ &Abstract.Arrow
```

4.2.2 Totality Protocol

Recall the diagram for totality:

$$\begin{tikzcd}
a \\
c \arrow[u, mapsto, dashed, "\Phi"']
\end{tikzcd}$$

The specification for totality is that given what's in black, i.e. the abstract node a, we can deduce what's in blue, i.e. the concrete node c, with the constraint that $a = \Phi(c)$. In Agda, we can formalize it as a total function totally-implement-• from Abstract.Node to Concrete.Node, with an additional law:

```
totally-implement-• : Abstract.Node → Concrete.Node
totally-implement-•-left-inverse-of-interpret :
  ∀ {a : Abstract.Node} {c : Concrete.Node}
  (totally-implement-• a = c) → (c interpret.↦ a)
```

It can be extracted as follows, dropping the corresponding law:

```
&totally-implement-• : &Abstract.Node → &Concrete.Node
```

But it suggests that even without totality, you probably want a partial function:

```
try-implement-• : Abstract.Node ⇸ Concrete.Node
try-implement-•-left-inverse-of-interpret :
  ∀ {a : Abstract.Node} {c : Concrete.Node}
  (a try-implement-•.↦ c) → (c interpret.↦ a)
```

Or instead with arbitrary meta-level side-effects (e.g. partiality, non-determinism, exceptions, etc.):

```
implement-• : Abstract.Node ⇢ Concrete.Node
implement-•-left-inverse-of-interpret :
  ∀ {a : Abstract.Node} {c : Concrete.Node}
  (a implement-•.↦ c) → (c interpret.↦ a)
```

Which extract as:

```
&try-implement-• : &Abstract.Node ↛ &Concrete.Node
```

And:

```
&implement-• : &Abstract.Node ⇢ &Concrete.Node
```

4.2.3 Completeness Protocol

Now recall the diagram for completeness:

$$\begin{tikzcd}
a \arrow[r, "A"'] & a' \\
c \arrow[u, dashed, mapsto, "\Phi"'] \arrow[r, "C"'] & c' \arrow[u, dashed, mapsto, "\Phi"']
\end{tikzcd}$$

The specification for completeness is that given what's in black, i.e. a concrete node c that interprets into abstract node a, and an abstract arrow f from a to some a', we can deduce what's in blue, i.e. a concrete arrow g from c to some node c', with the constraint that $f = \Phi(g)$. In other words, we have total function `completely-implement-⟹` from `Concrete.Node` and `Abstract._⟹_` to `Concrete._⟹_`:

```
completely-implement-⟹ :
  ∀ (c : Concrete.Node) {a a' : Abstract.Node}
    {c interpret.↦ a} (a Abstract.⟹ a') →
    ∃ (λ {c' : Concrete.Node} → (c Concrete.⟹ c'))
completely-implement-⟹-left-inverse-of-interpret :
  ∀ (c c' : Concrete.Node) {a a' : Abstract.Node}
    {f : a ⟹ a'} {g : c ⟹ c'} →
    (f completely-implement-⟹.⤇ g) → (g interpret.⤇ f)
```

Omitting proofs and implicit arguments that can be deduced from the explicit ones, it can be extracted more simply as:

```
&completely-implement-⟹ : &Concrete.Node → &Abstract.Arrow → &Concrete.Arrow
```

But as with totality, this suggests that even without completeness, you probably want a partial function:

```
try-implement-⟹ :
  ∀ (c : Concrete.Node) {a a' : Abstract.Node}
    {c interpret.↦ a} (a Abstract.⟹ a') ↛
    ∃ (λ {c' : Concrete.Node} → (c Concrete.⟹ c'))
```

```
try-implement-⟹-left-inverse-of-interpret :
  ∀ (c c' : Concrete.Node) {a a' : Abstract.Node}
    {f : a Abstract.⟹ a'} {g : c Concrete.⟹ c'} →
    (f try-implement-⟹.↦ g) → (g interpret.⤇ f)
```

Or instead with arbitrary meta-level side-effects (e.g. partiality, non-determinism, exceptions, etc.):

```
implement-⟹ :
  ∀ (c : Concrete.Node) {a a' : Abstract.Node}
    {c interpret.↦ a} (a Abstract.⟹ a') --→
    ∃ (λ {c' : Concrete.Node} → (c ⟹ c'))
implement-⟹-left-inverse-of-interpret :
  ∀ (c c' : Concrete.Node) {a a' : Abstract.Node}
    {f : a Abstract.⟹ a'} {g : c Concrete.⟹ c'} →
    (f implement-⟹.↦ g) → (g interpret.⤇ f)
```

Which extract as:

```
&try-implement-⟹ : &Concrete.Node → &Abstract.Arrow ↛ &Concrete.Arrow
```

And:

```
&implement-⟹ : &Concrete.Node → &Abstract.Arrow --→ &Concrete.Arrow
```

4.2.4 Fullness Protocol

We'll go faster over fullness. Fullness was this property:

$$\begin{tikzcd}
a \arrow[r, "A"'] & a' \\
c \arrow[u, dashed, maps to, "\Phi"'] \arrow[r, "C"'] & c' \arrow[u, dashed, maps to, "\Phi"']
\end{tikzcd}$$

It can be formalized and extracted as follows, and can also have weaker variants ending with ↛ or --→, that we'll omit.

```
fully-implement-⟹ :
  ∀ (c c' : Concrete.Node) {a a' : Abstract.Node}
    {c interpret.↦ a} {c' interpret.↦ a'} (a Abstract.⟹ a') →
    (c Concrete.⟹ c'))
fully-implement-⟹-left-inverse-of-interpret :
  ∀ (c c' : Concrete.Node) {a a' : Abstract.Node}
    {c interpret.↦ a} {c' interpret.↦ a'}
    {f : a ⟹ a'} {g : c ⟹ c'} →
    (f fully.↦ g) → (g interpret.⤇ f)

&fully-implement-⟹ :
  &Concrete.Node → &Concrete.Node → &Abstract.Arrow → &Concrete.Arrow
```

We now see how we could deduce Completeness from Strong Completeness, the conjunction of Fullness and Totality. The computational content of it would be equivalent to defining &completely-implement-⟹ as follows:

```
&completely-implement-⟹ c f =
  let c’ = &totally-implement-• (&codomain f) in &fully-implement-⟹ c c’ f
```

And then we see that if there are any non-deterministic choices in implementation, this will be very inefficient when the choices made for c’ = (&totally-implement-• (&codomain f)) don’t match those made for c, whereas an efficient implementation of &completely-implement-⟹ could let the choices made in c plus a path-dependence on f determine the choices made in c’. Implementing Completeness via Strong Completeness is over-constraining c’ without taking into account the information in c and f; and &totally-implement-• can be much more expensive than implementing f by following steps from a previously computed implementation c.

4.2.5 Liveness Protocol

It will be somewhat more subtle to extract a protocol from Liveness. Recall the diagram:

$$\begin{tikzcd}
a \arrow[rrr, "A^+"'] & & & a' \\
& & & c' \arrow[u, dashed, |->, "\Phi"'] \arrow[dr, bend left] & & \\
c \arrow[uu, dashed, |->, "\Phi"'] \arrow[urrr, bend left] \arrow[r, "C^+"'] & c_1 \arrow[r, "C^+"'] & c_2 \arrow[rr, dotted] & & c_n \arrow[r, dotted] & {}
\end{tikzcd}$$

A direct formalization goes something like this, assuming an appropriate definition for StrictMonotonicFunction:

```
liveness :
  ∀ {c : StrictMonotonicFunction ℕ.< Concrete.advancing}
    {a : Abstract.Node} {c_0 : Concrete.Node}
    {0 c.↦ c_0} {c_0 interpret.↦ a}
    ∃_2 (λ (c’ c’’ : Concrete.Node) →
    ∃_2 (λ (n : ℕ) (n c.↦ c_n) →
    ∃_2 (λ (f : c Concrete.⟹ c’) (g : c’ Concrete.⟹ c_n) →
    ∃_2 (λ (a’ : Abstract.Node) (h : a Abstract.⟹ a’) →
    ∃ (λ (c’ interpret.↦ a’) →
    (f interpret.⤇ h))))))
```

Now, positting an infinite sequence of advancing concrete steps isn’t very constructive; in other words, its computational contents aren’t very usable: first, no one is going to actually provide an infinite sequence of transitions as reified input; second, unless in practice there’s a small enough bound on n, the property alone is too weak to be directly useful. Still the property being weak is a good criterion to reject as less-than-generally-useful implementations that do *not* possess the property.

However, a more useful point of view is to realize that this infinite sequence of advancing concrete steps is to be understood not as reified progress that the meta-level can coldly reason about, but reflected progress that the meta-level is experiencing in all its side-effectful glory:

something that intrinsically requires one to use --→ instead of merely →. Formally, it would look like that, where the dashed arrow is analogous to that of `advance` in subsection 4.1.4:

```
advance-abstract-computation :
  ∀ (c : Concrete.Node) {a : Abstract.Node} {c interpret.↦ a} --→
    ∃_2 (λ (c’ c’’ : Concrete.Node) →
    ∃_2 (λ (f : c Concrete.advance.↦ c’) (g : c’ Concrete.⟹ c’’) →
    ∃_2 (λ (a’ : Abstract.Node) (h : a Abstract.⟹ a’) →
    ∃ (λ (f interpret.⤇ h) →
    a Abstract.advance.↦ a’))))
```

The formula above just says that you can achieve an abstract advance from `a` (the final term of the dependent product) by a meta-level computation (the first, dashed, arrow, after the hypotheses) starting from a concrete implementation of the abstract computation. The extracted interface is then simply as follows, a way to emulate the abstract computation's `&Abstract.&advance` just by acting on the concrete computation:

```
&advance-abstract-computation : &Concrete.Node --→ &Concrete.Arrow
```

The implicit contract (lost during extraction) being that the result implements an advance in the concrete computation that goes past (a concrete arrow that interprets as) an advance in the abstract computation.

Variants of Liveness

Bounded Liveness, Strong Liveness, Bounded Strong Liveness, Strong Step Preservation (see subsection 3.2.3) have stronger contracts than Liveness, that make them more usable in many ways. A proper formalization of their respective contracts would require more effort than we're interested in putting for that purpose, and we're leaving it as an exercise to the reader: you need to equip your operational semantics with some kind of additive measure and/or set of atomic computation arrows, and assert that an adequate advance in the abstract computation can be achieved through a sufficient advance in the concrete computation. Interestingly, though, when you extract the computational content of these properties, you still get the same type after erasing the logical part of the contract:

```
&advance-abstract-computation-in-bounded-time :
  &Concrete.Node --→ &Concrete.Arrow
&advance-abstract-computation-to-observable-point :
  &Concrete.Node --→ &Concrete.Arrow
&advance-abstract-computation-to-observable-point-in-bounded-time :
  &Concrete.Node --→ &Concrete.Arrow
&step-abstract-computation-in-bounded-time :
  &Concrete.Node --→ &Concrete.Arrow
```

It's just that your advance variants have slightly different requirements and guarantees attached to both the abstract and concrete computations under consideration, with respect to real-time bounds, coherence, synchronization. For all its weakness, the virtue of Liveness is to embody the common spirit of these many properties, their common computational content, precisely without imposing any overly specific guarantee.

4.2.6 Co-Liveness Protocol

Co-liveness was not expressible as a diagram in our simple language of diagrams, because that language cannot directly deal with either negation or logical disjunction, and therefore couldn't express either the done predicate or its relationship with advancing. (That language of diagrams could of course be extended to support one kind of logical "or" or another, but that's a different issue). The logical language of Agda has no such limitation, and it can express co-liveness simply as follows, where done is the same as in subsection 4.1.4:

```
done-means-done :
  ∀ (c c' : Concrete.Node) {a : Abstract.Node} {c interpret.↦ a}
  (c Concrete.⟹ c') Concrete.done c →
  ∃₂ (λ (a' : Abstract.Node) {c' interpret.↦ a'} → Abstract.done a')
```

Here is a case where there is no computational contents to extract: co-liveness is a promise made that a logical constraint is respected, which leaves nothing to express when you drop logical constraints. It just means that the function &Concrete.&done, when it returns true on a node that results from running the computation, ensures that that node has an observable meaning for which &Abstract.&done also returns true (and itself satisfies the specification for being done).

Advance Preservation

We'll skip relevance, which was a conceptual stepping stone from co-liveness to observability but on its own is a bit too complex to formalize, and its computational contents accordingly awkward to use. Instead, we will directly extract the computational contents of advance preservation and observability, that better factor the concept.

The diagram for advance preservation was as follows:

$$\begin{tikzcd}
 & a'' \\
a \arrow[ur, "A^+"'] \arrow[dr, "A^+"'] & \\
 & a' \\
c \arrow[uu, dashed, maps to, "\Phi"'] \arrow[r, "C^+"'] & c' \arrow[u, maps to, "\Phi"']
\end{tikzcd}$$

It can be formalized as follows:

```
advance-abstract-computation-if-possible :
  ∀ (c : Concrete.Node) {a : Abstract.Node} {c interpret.↦ a}
  {a' : Abstract.Node} {f : a Abstract.⟹ a'} (Abstract.advancing f) →
  ∃₂ (λ (c'' : Concrete.Node) (g : c Concrete.⟹ c'') →
  ∃₂ (λ {c'' interpret.↦ a''} {h : a Abstract.⟹ a''} →
  ∃ (λ {g interpret.⤇ h} →
  Abstract.advancing h)))
```

But in the end, the extraction has the same type as for liveness: given a concrete node for which the abstract computation isn't done, return a concrete arrow from that node that has an abstract meaning that is advancing the abstract computation. As usual the subtle difference in meaning is left implicit in the poorer extraction target type system.

```
&advance-abstract-computation-if-possible : &Concrete.Node → &Concrete.Arrow
```

4.2.7 Observability Protocol

The crown jewel of our implementation properties, Observability, has the following diagram:

$$\begin{tikzcd}
a \arrow[rrr, "A"'] & & & a'' \\
& c \arrow[ul, dashed, mapsto, "\Phi"'] \arrow[r, "C"'] & c' \arrow[r, "C^s"'] & c'' \arrow[u, dashed, mapsto, "\Phi"']
\end{tikzcd}$$

It can be formalized as follows:

```
observe :
  ∀ {c : Concrete.Node} {a : Abstract.Node} {c interpret.↦ a}
  (c’ : Concrete.Node) {f : c Concrete.⟹ c’} →
  ∃₂ (λ (c’’ : Concrete.Node) (g : c Concrete.⟹ c’’) →
  ∃₂ (λ (a’’ : Abstract.Node) {h : a Abstract.⟹ a’’} →
  ∃ (λ (not-advancing g) →
  (g Concrete.○ f) interpret.⤇ h)))
```

Its computational contents can then be reduced to the following:

```
&observe : &Concrete.Node → &Concrete.Arrow
```

The function takes a concrete node `c’` and returns a non-advancing transition arrow `g` from `c’` to an observable state `c’’`.

Note how we’re not passing any of `c` or `a` or `f` as parameters, or returning any of `a’’` or `h`, anymore than the correctness proofs: we assume that they are part of the prerequisite logical constraints excised during the extraction. Indeed, in practice we assume that the protocol function `&observe` will only be called on a node `c’` that is indeed the interrupted intermediate state of a computation; and we assume the returned values are correct.

We can then define the following utility functions:

```
&observe.● : &Concrete.Node → &Concrete.Node
&observe.● c = (&Concrete.&codomain (&observe c))

&observe.⟹ : &Concrete.Arrow → &Concrete.Arrow
&observe.⟹ f = (&observe.● (&Concrete.&codomain f)) &Concrete.○ f
```

`&observe.●` synchronizes a concrete node to a safe point that is observable. `&observe.⟹` takes a concrete arrow `f` from an observable node `c` to an arbitrary node `c’`, and returns an arrow from `c` to an observable state `c’’` that is the composition of a non-advancing transition arrow `g` with `f`.

4.2.8 Variants of Observability

There again, Observability has many variants, such as Observability for correct programs, Bounded Observability, or Strong Observability. Formalizing them logically will require a lot of additional formalism, such as a definition of correct program, or a notion of length for concrete computations, and some specific notion of advancing that may differ from that used in other contexts. We leave this formalization as an exercise to the reader who cares. In the end, all these logical variants involve passing as extra arguments and/or receiving as extra results some compile-time values that witness a proof of correctness of the extra invariants required. But when you extract functions for a computational protocol, all these these extra arguments and results disappear, and the function signature remains:

```
&observe-variant : &Concrete.Node → &Concrete.Arrow
```

In the end, all these variants of Observability correspond to taking the concrete computation at an arbitrary intermediate point, and synchronizing to the nearest safe point, at which the concrete computation is observable, and an interpretation can be obtained.

4.3 Implementation Reflection

4.3.1 Properties as Abstractive Reflection

If we look closely at the computational content of all the nice properties we may require from an implementation, it appears they constitute a protocol for Abstractive Reflection (see section 1.3).

Indeed, underlying Totality and Completeness, or even partiality and incompleteness, is an implementation protocol: specifications for a profunctor `implement` and its extraction as functions `&implement.•` and `&implement.⟹`. And underlying Soundness, combined with Observability, is an interpretation protocol: specifications for a partial functor `interpret` and its extraction as functions `&interpret.•` and `&interpret.⟹`. The computational content of these properties is the ability to navigate up and down the levels of a tower of abstraction.

4.3.2 Lifting Evaluative Reflection Protocols

As for Liveness, combined with Observability, it allows to *lift* the Concrete computation's evaluative reflection protocol into an evaluative reflection protocol for the Abstract computation. For simplicity, we will only show how to lift the extracted versions.

First we can lift the abstract perform and simulate functions from the concrete ones:

```
&Abstract.&perform.• : &Abstract.Node → State
&Abstract.&perform.• a = &Concrete.&perform.• (&represent.• a)

&Abstract.&perform.⟹ : &Abstract.Arrow → State --→ State
&Abstract.&perform.⟹ a s =
  let c = (&represent.• s) in
    &Concrete.&perform.⟹ (&represent.⟹ c s) c

&Abstract.&simulate.• : State → &Abstract.Node
&Abstract.&simulate.• s = &interpret.• (&Concrete.&simulate.• s)

&Abstract.&simulate.⟹ : State → (State --→ State) --→ &Abstract.Arrow
&Abstract.&simulate.⟹ s a =
  &interpret.⟹ (&observe.⟹ (&Concrete.&perform s a))
```

Then we can lift the `&run` and sibling functions:

```
&Abstract.&run : &Abstract.Node --→ &Abstract.Arrow
&Abstract.&run a =
  &interpret.⟹ (&observe.⟹ (&Concrete.&run (&represent.• a)))
```

```
&Abstract.&step : &Abstract.Node --→ &Abstract.Arrow
&Abstract.&step a =
  &interpret.⟹ (&advance-abstract-computation-one-step-in-bounded-time (&represent.• a))

&Abstract.&advance : &Abstract.Node --→ &Abstract.Arrow
&Abstract.&advance a =
  &interpret.⟹ (&observe.⟹ (&advance-abstract-computation (&represent.• a)))

&Abstract.&eval : &Abstract.Node --→ &Abstract.Arrow
&Abstract.&eval s =
  &interpret.⟹ (&Concrete.&eval s)
```

Finally with the help of a helper function `!observe`, we can lift the evaluatively reflected function `!run` and its siblings:

```
!observe : State --→ State
  !observe = &Concrete.&perform.⟹ (&observe (&Concrete.&simulate.• s)) s

&Abstract.!run : State --→ State
&Abstract.!run s = !observe (&Concrete.!run s)

&Abstract.!step : State --→ State
&Abstract.!step s =
  &Concrete.&perform
    (&step-abstract-computation-in-bounded-time (&Concrete.&simulate.• s)) s

&Abstract.!advance : State --→ State
&Abstract.!advance s = (&advance-abstract-computation
  !observe
    (&Concrete.&perform
      (&advance-abstract-computation (&Concrete.&simulate.• s)) s)

&Abstract.!eval : State --→ State
&Abstract.!eval s = &Concrete.!eval s
```

Note that the above definitions equality are meant as a specification, but not as a naive computational recipe: it can be computationally expensive to simulate the entire computation state only to perform back a small change to it; but if the reification is done lazily, if meta-level computation is inlined, if the intermediate data structures are deforested away, or if the specification is otherwise properly "optimized", then yes, this specification can be compiled into code of acceptable quality that may synchronize the computation being currently executed in a provably correct way.

4.3.3 Implementations as Implementations

In subsection 4.1.5 above, we called it for a computation to be *actually implemented* if there existed an evaluative reflection protocol for it. We just justified why the name *Implementation* befits the entity we called "Implementation": if there is an Implementation of an abstract computation `A` with a concrete computation `C` and `C` is *actually implemented* (down to e.g.

being the states of an Operating System process), then indeed `A` is also *actually implemented* thanks the the lifting described in the previous subsection.

An Implementation thus allows us to *reduce* the problem of actually implementing a computation `A` to that of actually implementing a computation `C`.

However, we also realize the importance of some of the properties that were otherwise neglected in existing software and previous literature:

- Observability, notably, is necessary to maintain full abstraction of the details of hypo-computations when reasoning about hyper-computations.

- The ability to interpret, implement and reason about not just nodes, but also arrows, is also crucial in maintaining this protocol.

Chapter 5

Programming with First-Class Implementations

5.1 Expressing Known Phenomena

5.1.1 Testing a theory

You don't test the validity of a theory by seeing that it says correct things, but by seeing that it doesn't say incorrect things. What you test by seeing that it does say correct *and previously unpredicted* things, is the interest of a theory you've tested to be valid.

In logic and mathematics, any theory is valid as long as it is consistent; this as such is a very low bar, but then it can be quite hard to discover something *interesting* that can be proven to be valid. In computer science, logical constraints are even laxer, and any code that compiles can be considered valid, as long as no overly broad promise is made as to what the code does; as for criteria of interest, they tend to be even more subjective than in mathematics — which makes for a large *market* in ideas that someone, anyone, finds interesting enough to spend resources on.

The theory we propose is quite poor in terms of new results. The few relatively new constructs it proposes, mainly having to do with how Observability generalizes PCLSRing, are quite trivial, at least after the fact. Subsequent chapters will build on these constructs, but provide even less theory – and even then, most of the applications we propose have already been achieved somehow even in absence of our theory. The question is then: is our theory interesting, beside obvious making the importance of this one concept, Observability?

5.1.2 A Simplifying Paradigm

We will argue that our theory is useful in that it allows to think in simple terms a lot of phenomena that are well-known in computer science, yet seldom discussed clearly, by lack of an adequate paradigm. We will thus suggest a few ways that our theory underlies a simplifying paradigm that can clarify existing techniques and concepts.

All the techniques we'll discuss are very well known. Yet we contend that our approach can be bring some new insight, or at least make some old insight simpler and more obvious. This insight can be used informally when designing or analyzing systems or formally when developing correctness proofs of systems, or systems that automatically prove correctness of what they do. And this insight leads to developing and using composable metaprogramming frameworks at both compile-time and runtime. Of course this is all easier said than done; nonetheless it is better said than kept implicit, or worse, ignored.

5.2 Elucidating Compilation

5.2.1 Intent

Our notion of implementation provides an obvious semantic framework to reason about the correctness of compilers, and the many properties that these compilers may or may not guarantee with respect to the implementation they provide for the programs they compile. Once again, Totality, for instance, is out of reach for the implementation of an infinite language on a finite machine; but the failures may be clearly confined to one level of abstraction at which point there could be better defined coping strategies compared to a situation where the looming existence of failure cases inevitably exists but escapes specification.

But thinking in terms of implementations can bring broader insight as to the nature of the activity of implementing and using a programming language. We will use some of our diagrams to illustrate such insight.

5.2.2 Compilation as Implementation

A compiler is clear meta-level concept: it is an ante-program, running in the ambient operating system, that given a (first-class representation of a) program generates a (first-class representation of a) machine-runnable hypo-program that preserves its semantics. The simplest way of thinking of a compiler is as an implementation profunctor, that takes computations in an abstract category A, that embodies the programs as specified by the programmer's source code, and returns computations in a concrete category C, that embodies the binary machine code actually runnable by the computer.

In our diagrams, we will keep representing the functorial interpretation arrows rather than the co-functorial implementation arrows. The diagram for this simple way of thinking is therefore as follows:

$$\begin{tikzcd}
A \\
C \arrow[u, dashed, "\Phi"']
\end{tikzcd}$$

And the signature would be:

```
compile : {A C} Implementation A C → (A.Node --→ C.Node)
compile I a = I.&implement.● a
```

Of course, this compiler may be made of many *passes*, which means that it can be seen as the composition of many implementations, each further transforming the computation while preserving its semantics. But since functions can always be decomposed into further more elementary functions, we will not usually represent that in our diagrams — that will remain the obvious implicit. Instead, we will focus on diagrams that can't be simplified by merging passes.

5.2.3 Source as Representation

Now, the previous diagram assumed that programmers can directly specify the computation they mean, in the abstract category of programs with their associated semantics. But actually, what programmers manipulate is not abstract programs, but concrete *source code*.

Source code contains many details that are irrelevant to the abstract program that is *meant* for the computer to run. For instance, comments are directed to other programmers, whereas identifiers are interchangeable labels for the sake of the running program: some of them matter when displaying debugging information, but that's usually not all of them, and only in a special failure mode; in a deep sense they don't matter to the semantics of the program. The source program may also include a lot of distinctions that will hopefully be *optimized away*, such as the order in which some operations are done, or the fine details of some arithmetic formula.

The compiler is allowed and *expected* to transform the program "up to" equivalence with respect to the semantics of some abstract computation A, then emit an implementation of this equivalent program in the concrete computation C. But the programmer actually writes in a programming language S that is in a way "lower level" than A, since it includes many additional details and distinctions: it's just lower level in a different way than matters to the computer; it's lower-level in a way that matters for human brains.

The diagram for this slightly more elaborate way of thinking is as follows:

$$\begin{tikzcd} & A & \\ S \arrow[ur, dashed, bend left, "\Sigma"] & & \\ & & C \arrow[uul, dashed, bend right, "\Phi"'] \end{tikzcd}$$

And the signature would be:

```
compile : {S A C} Implementation A S → Implementation A C →
          (S.Node --→ C.Node)
compile IAS IAC = IAC.&implement.● ○ IAC.&interpret.●
```

5.2.4 Semantic Gap between Meaning and Understanding

Now, the above picture is an improvement in explaining what a compiler is about, but it still doesn't tell the entire stories.

So, programmers write their programs in S, but actually intend to denote some abstract program in A where equivalent computations are identified together as a same ideal program. The hope is that the compiler can then consider all computations equivalent to the meaning source program, and *rewrite* the program into one that can be most efficiently translated to machine code that the computer can actually run. From the point of view of the compiler specification, these rewrites can be seen as non-advancing arrows mutually relating all starting nodes in each equivalence class of the computation: the compiler may pick any of these semantic-preserving "optimization" rewrites when it implements the program.

However, equivalence of computations is not a computable criterion: an algorithmic way to identify all operationally equivalent programs would trivially solve the halting problem which we know is impossible [25]. Therefore, what an algorithmic compiler does is necessarily *weaker* than that. The algorithmic compiler can only consider computations *up to* some computable equivalence relation that is weaker than operational equivalence. This computable equivalence relation corresponds to a computable subcategory U of A, where only *some* of the valid rewrites are considered, and most potential rewrites are actually ignored. The "optimization" rewrite arrows included in U are a strict subset of the rewrite arrows allowed in A. In other words, U is not a *full* implementation of A; there is necessarily a semantic gap (denoted by the arrow Gap below) between the full semantics and the little that the compiler can actually take into account when optimizing while keeping the problem tractable.

The diagram for this yet more elaborate way of thinking is as follows:

$$\begin{tikzcd}
 & A & \\
 & U \arrow[u, dashed, "\mathsf{Gap}"'] & \\
S \arrow[uur, dashed, bend left, "\Sigma"] \arrow[ur, dashed, "\Sigma_U"] & & C \arrow[uul, dashed, bend right, "\Phi"'] \arrow[ul, dashed, "\Phi_U"]
\end{tikzcd}$$

And the signature would be:

```
compile : {S U C} Implementation U S → Implementation U C →
          (S.Node --→ C.Node)
compile IUS IUC = IUC.&implement.● ∘ IUC.&interpret.●
```

The Understanding Gap is also for Humans

Note that conversely, programmers have to specify their programs in the logic of A, but their actual intent could be both wider or narrower than what the logic of A allows to express.

On the one hand, a programmer might actually want "any program that solves his problem", and there might be a large class of such programs beyond the one he writes; the differences might be cosmetic, as in using icons of a different color; or they might be profound, as in using a radically different approach to realizing a real-world solution (such as designing a cheap analog physical sorter to distinguishing zipper tabs instead of an elaborate robot with digital AI vision software to recognize them then pick them when improving the robotized manufacture of zippers). [citation needed] Ideally (from the point of view of the programmer), he could just use one instruction "DWIM" [citation needed] and, taking into account the context, the compiler could be able to rewrite that into a program that best solves his issue.

On the other hand, the programmer might be mistaken in what rewrites A allows and write a program that happens to work in U because the compiler won't rewrite the program into one that doesn't work anymore for the programmer's purpose; but an update to the compiler might include deeper analyses and a wider set of optimizing rewrites, resulting in a larger semantics U' closer to A, in which the same program is indeed rewritten into one that fails catastrophically. This latter phenomenon is unhappily frequent in C programs, whereby compilers are allowed to arbitrarily rewrite programs in the advent of a large class of specified "Undefined Behaviors" (UB); earlier compilers do not rewrite some programs despite their UB, and the compiled programs then happen to match the intent of the programmer; then a latter compiler is able to recognize the UB, and rewrites programs in ways that defeat the intent of the programmer, which can introduce computer crashes, data loss, data corruption, or security vulnerabilities.[citation needed]

And of course, humans can fail to properly understand any of those computation categories at all, or to act on this understanding — but that constitutes a more trivial class of bugs about which the above diagram bring no enlightenment.

5.2.5 Static Typing

In chapter 4, our implementations ultimately rested upon a same type `State` into which all computations were reduced, whatever their level of abstraction was. However, this doesn't mean that in the end our implementation framework necessarily reduces down to a trivially typed or otherwise "dynamically typed" system. Indeed, while all abstraction levels *for a given computation* reduce to a same single type, different computations may each be reduced to their own distinct type.

Indeed, be reminded that our computations, as categories, embody *operational semantics*. Types, by contrast are *denotational semantics*, which as we saw in subsection 2.2.5, are more abstract categories with no non-trivial arrows. Therefore, when a computation is typed, we have an *abstract interpretation* (see 2.2.6) from said computation to the the type system. If a node in the computation is mapped to a given type, all the arrows reachable from a node in this computation are necessarily mapped to the identity arrow for that type (which the only arrow with that type as a domain), and all nodes reachable from said node are therefore also mapped to the same type. This property, known as *subject reduction*, is a basic soundness property for static type systems.[citation needed]

We can thus visualize compilation of a typed language as per the following diagram:

$$\begin{tikzcd}
T & & \\
& \{\tau\} \arrow[ul, hook] & \\
A \arrow[uu, dashed] & & \\
& A_\tau \arrow[ul, hook] \arrow[uu] & \\
U \arrow[uu, dashed] & & \\
& U_\tau \arrow[ul, hook] \arrow[uu] & \\
S \arrow[uu, dashed] & & \\
& S_\tau \arrow[ul, hook] \arrow[uu, dashed] & \\
& & C_\tau \arrow[uuul, dashed]
\end{tikzcd}$$

Let's consider the source language S, the abstract computation A representing the desired operational semantics of all well-formed programs in S, and the type system T for valid programs in A. We have partial functors from S to A and A to T, where the partiality indicates that not all programs are well-formed, and not all well-formed programs are well-typed. Now, the well-formed and well-typed programs of type τ constitute a subcategory S_τ or S, and their operational semantics constitute a subcategory A_τ of A, and by definition they type into a single type τ. Knowing that the program is well-typed and that its static analysis yields type τ, the compiler can apply a number of type-specific rewrite strategies specific to A_τ, not all of which would be valid in all of A. Of course, as per subsubsection 5.2.4 above, the actual strategies it *will* are necessarily a computable subset U_τ of A_τ (where U once again stands for Understood semantics).

Thus, when compiling an expression of type τ from a high-level programming language to a low-level programming language, the actual type `State` of all the nodes in the concrete computation at the bottom may actually be `ConcreteState` τ where `ConcreteState` is some appropriate functor parametrized by τ. Note that τ here is not necessarily the human-readable type used by in static type system designed for humans to specify; it can actually be an even finer type resulting from some arbitrarily elaborate static flow analysis of the program behavior.

5.3 Further Compilation Topics

5.3.1 Aspect Oriented Programming

An interesting case of multiple programming with multiple abstraction levels is Aspect-Oriented Programming [14], where the multiple levels are used not for software analysis, but for software synthesis.

Aspect-Oriented Programming (AOP) is a programming paradigm according to which programmers will separately specify several distinct *aspects* of their programs, such as authentication and access policies, logging of various events, allocation of resources, strategies to follow for various tasks, etc.

The hope is that by keeping each of many aspects separate, it becomes easier to reason about the program than if the programmer has to understand all the aspects at once; therefore, harder problems may be tackled with fewer bugs. The cost of it is creating and maintaining an architecture in which these aspects are and remain independent enough indeed so that reasoning about one aspect doesn't involve pulling in all the other aspects into the reasoning context.

The problem would be trivial if the aspects were independent (or "orthogonal"). with a solution being simply (an implementation of) the cartesian product of the aspects. But what makes AOP interesting is precisely the automation of weaving when the aspects are not independent. Aspects are then said to *cross-cut*. This can be stated formally by adding constraints on the cartesian product of the aspects that restrict what computations are valid in the intended system.

A useful tool for specifying the cross-cutting of aspects in a declarative way is *join-points*. A specific join-point is are events such as a named function being called with certain parameters, or its returning. Various aspects may then specify additional behavior that has to happen at, before, after or around such join-points: checking the caller has proper credentials, logging the event, tracking ownership of the computation, allocating then deallocating some resources, etc. Whatever that specific aspect deals with.

Once the aspects are specified, they are *weaved* together by an ante-program called an *aspect weaver* to generate the actual program. Priority rules may further constrain how the weaver may combine or schedule the behaviors specified by the various aspects. That weaver may then fail with an error if the aspects specify an invalid combination (or if any aspect is invalid by itself).

Using our notion of implementations, we can model the semantics of AOP in terms of simultaneously implementing several programs, one per aspect. Formally, a programming language is specified by having a number of aspects, say A_1, A_2, A_3. Join-points correspond to forgetful functors, say Ψ_1, Ψ_2, Ψ_3, linking each aspect to an aspect-less computation J that specifies the join-points (each Ψ_n forgets the specifics of aspect A_n). We are looking for some forgetful functors, say Φ_1, Φ_2, Φ_3, that will map a more concrete language C to each of these aspects (each Φ_n forgets the specifics of all aspects *except* A_n). The diagram is as follows:

$$\begin{tikzcd}
 & J & \\
A_1 \arrow[ur, dashed, "\Psi_1"] & A_2 \arrow[u, dashed, "\Psi_2"] & A_3 \arrow[ul, dashed, "\Psi_3"'] \\
 & C \arrow[ul, dashed, "\Phi_1"] \arrow[u, dashed, "\Phi_2"] \arrow[ur, dashed, "\Phi_3"'] &
\end{tikzcd}$$

Given abstract programs in each of these aspects, say, a_1, a_2, a_3, the problem is then to find a concrete program c that simultaneously implements each of those abstract programs:

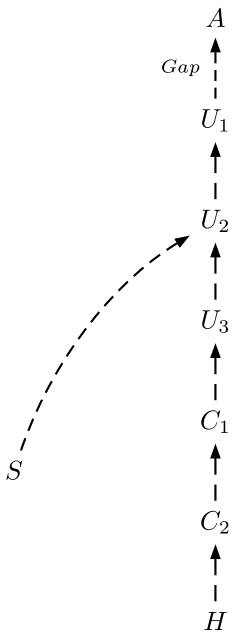
$$\begin{tikzcd}
 & j & \\
a_1 \arrow[ur, dashed, mapsto, "\Psi_1"'] & a_2 \arrow[u, dashed, mapsto, "\Psi_2"'] & a_3 \arrow[ul, dashed, mapsto, "\Psi_3"'] \\
 & c \arrow[ul, dashed, mapsto, "\Phi_1"] \arrow[u, dashed, mapsto, "\Phi_2"'] \arrow[ur, dashed, mapsto, "\Phi_3"'] &
\end{tikzcd}$$

AOP can thus be formalized using our framework. Within that framework, the aspects $a_1 \ldots a_n$ appear as constraints on the specified hypo-program c; and weaving the aspects appears as a problem in Constraint Logic Programming [citation needed], except at the meta-level (the ante-stage, in our nomenclature): the aspect weaver is a constraint solving logic metaprogram. And indeed, custom Aspect-Oriented Programming systems can be built quite effectively by writing ante-programs in a suitable logic programming framework [26]. What our approach suggests is a way that aspect constraints can be formalized, to enable reasoning about correctness proofs for logical properties of aspect-oriented programs.

5.3.2 Navigating the Semantic Tower

When figuring out how a program works or fails to work, so as to use it or improve it, programmers have to work at several levels of abstraction. The ability of navigating the many levels of abstraction involved in a program is perhaps the defining quality of good programmers; it is certainly a factor that limits how elaborate programs can get before they become impossible to either write or maintain.

Our framework makes it possible to formalize this problem to a point; and it suggests a class of tools that could be developed to help with the issue. Consider the following diagram:



When specifying a computation, programmers provide the computer system source code S in some programming language (or set of such). The formal specification for S defines an abstract semantics A for it.

Programmers also specify at what level of abstraction the program should be considered. This level of abstraction, U_2 in the above diagram, indicates what rewrites are permissible to the compiler for the sake of "optimization" (see previous subsection 5.2.4).

Above are higher levels of abstractions, that allow for more rewrites, but sacrificing some details that the programmer might (or then again might not) care about. That's U_1, A in the diagram, but of course, there is no limit to how many levels and sub-levels could be considered instead, and A is a false top because no finitely describable system can encode all the true logical properties of the computation[10].

Below are lower levels of abstractions, that add more details that the programmer doesn't usually care about (or else he would have specifically selected a higher level of abstraction and enabled more compiler optimizations; then again, he might only trust the compiler so much). That's U_1, C_1, C_2, H in the diagram, where the C_n are supposedly more concrete representations, from high-level language down to virtual machine, to linkage with some low-level language like C, to assembly language, and bottoming out to H, which stands for the hardware.

However, at the bottom no less than at the top, there is no limit to how many levels and sub-levels could be considered, and you could always descend lower: not just into hardware instructions, not just into VHDL specifications or silicon masks, but into equations describing electric potentials, or the behavior of quantum particles, or superstrings, or hypothetical entities computing digital physics [citation needed].

Note that programmers can select not just between the levels U_n of formal understanding of the language by the compiler, but also between the levels C_n of concrete representation used by the compiler: not only can programmers inspect the bytecode or assembly files produced by the compiler, they can also carefully pick which version of which compiler is used — and even modify the compiler. It is possible for programmers to have as much (or as little) control as they want at compile-time. They can even change the hardware level H by running on a different CPU, whether through physical hardware, or through software emulation.

Navigating Abstraction Levels at Runtime

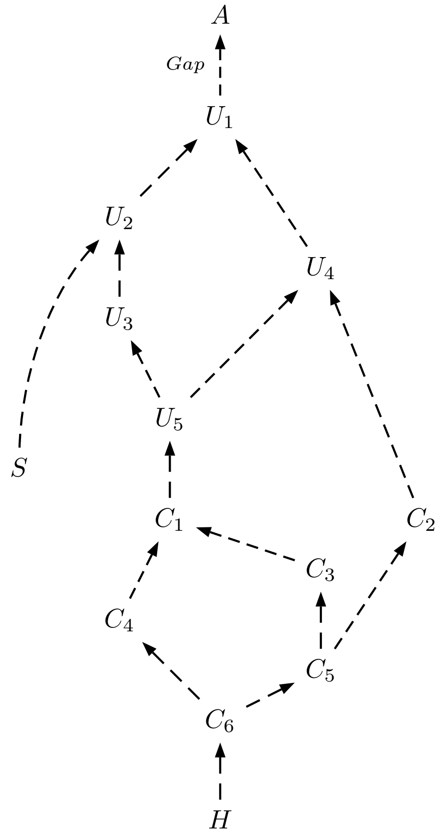
Now, this control doesn't have to be at compile-time! If each implementation in the semantic tower provides a full reflection protocol both abstractive and evaluative, then it becomes possible for programmers to decide *at runtime* what level of abstraction they are interested in.

This means that when they are debugging a difficult issue, they can simulate and re-perform the interaction, zoom in and out at will, locate the issue and focus on it until they pinpoint the bug at the level of detail that best explains it. If they have enough simulation records of what good looks like, they could even train programs to do most of the detection work automatically.

We'll elaborate on that topic in the following chapters.

Non-Linearity in the Tower

Of course, we already saw when discussing Aspect-Oriented Programming that abstraction needs not be a linear order, but can be of any partial order. Thus the semantic tower needs not be linear, but can have any shape. Consider the following diagram — without bestowing too much importance to its arbitrary particulars, meant solely to illustrate cases of non-linearity:



Not every pair of node is comparable: in the above diagram, U_2 and U_4 deal with mutually irreducible aspects, although if you descend to level U_5 you can express both concern; Yet again, if you descend into C_1 or C_2 you have mutually irreducible computations. Mutually irreducible computations could take into account aspects such as serializing thousands of concurrent computations into events on a handful of CPUs, taking garbage collection cycles into account, the accounting of various resources, the logging of progress messages, the intermediate states in a distributed transaction protocol, the exact addresses of various data structures in memory, etc.

The programmer may at times want to focus on any of those levels of abstration. What more, while debugging an issue, he may be considering some lower level of abstraction on the part of the program experiencing the issue than on the rest of the system that is behaving as expected. What more, if there are a distinct abstraction levels available at which to consider one piece of the program (that would be 14 including source code level in the above arbitrary diagram), then for every way that one may factor part of the computation as the interaction of n similarly abstractable pieces, there will be a^n ways of abstracting the overall system.

Therefore, while the simple diagram of a linear semantic tower with a small number of totally ordered abstraction levels might make sense when describing the architecture of a single compiler's passes, it just isn't applicable in the more general case of navigating abstraction levels at runtime, as becomes possible with our protocol: there are infinitely many ways to decompose and recompose software, to rearrange existing abstraction layers into new hierarchies; and there are even more ways to contrive and use wholly new abstraction layers, including ones that hadn't been invented yet when the program was first started.

5.3.3 Refactoring

Refactoring a program can be understood in terms of our semantic diagrams:

$$\begin{tikzcd}
& A & \\
& U \arrow[u, dashed, "Gap"] & \\
S \arrow[ur, dashed, bend left, "\Phi"] & & S' \arrow[ul, dashed, bend right, "\Phi'"']
\end{tikzcd}$$

A source program S has some understood semantics U, that itself is a computable subset of some abstract semantics A. Another source program S' is sought that is better in some way, but within the constraint that the change shall preserve the very same semantics U.

A suitable ante-system in which U is formalized can help automate part or totality of this change, while ensuring that the constraint is satisfied in the end. Even when automated decisions can't be made for sure (e.g. splitting an identifier in two initially equivalent identifiers), it might keep track of how far you are from the goal, by maintaining a checklist of places where decisions need be made (the occurrence of the old identifier), and offer sensible options (pick one or the other of the new options).

Sometimes U and A differ slightly from the semantics used during execution, which might reflect a limitation in the refactoring tool being used, or a choice by the programmer to somewhat actually modify the execution semantics or the program while preserving some of its other aspects.

It is often useful to alternate modifications to a program that provably preserve its semantics and mofifications that don't (yet that preserve some other aspects). Ante-programming a first-class computation can help in both cases.

Note that refactoring is usually done at compile-time by an ante-program, but that the same principle can be used *at runtime* by a controlling hypo-program; the corresponding phenomenon is then *migration*, which we will study in detail in next chapter.

5.3.4 Developing

The above semantic tower was all about exploring the semantics of the program as it exists, i.e. was lies below A. But of course, we could instead explore the semantics of the program as it should be, i.e. was lies above A. Here is a diagram to illustrate what we might find:

$$\begin{tikzcd}
 & & \top & & \\
 & R \arrow[ur, dashed] & & D \arrow[ul, dashed] & \\
A \arrow[ur, dashed] & & & & A' \arrow[ul, dashed] \\
S \arrow[u, dashed] & & & & S' \arrow[u, dashed]
\end{tikzcd}$$

At the bottom of the semantic tower, you have the current computation in source code form S. What it does is embodied in an abstract computation A above it. The goal is to find another source program S', corresponding to an abstraction computation A', that itself is included in a larget set of computations D, that the D stands for "DWIM", or "Do What I Mean": programs that solve the programmer's problems.

For that, the programmer can hopefully rely on the space R of programs that can be automatically obtained by algorithmic "refactoring". R represents what is easy to achieve incrementally; it includes importing libraries of existing functions, refactoring strategies provided by an Integrated Development Environment, and any tricks made possible by the editor, made easier or harder by the language syntax, and by any available combination of parsers, processors and pretty-printers.

Ideally, the intersection between R and D would not be empty, and the computer system may help semi-automatically find a solution. But this need not be the case, and the programmer may just have to manually write a lot of code. From a semantic preservation stand point, the only semantics preserved are the fact of being in the space of all source programs possible, i.e. the top category $\top$, as represented on the top the diagram: it's the terminal category, a trivial category with one node and one arrow; all programs are mapped to that same node, and thus are seen as equivalent inasmuch as they all equally implement that same most abstract of nodes. Any source program goes.

But in the end, the programmer is responsible for determining what programs are acceptable and describing those criteria in a way that is constructive enough for the computer to synthesize such a program from the given specifications. And that's D.

When writing a new program "from scratch", then S is initially the empty program — Ehud Shapiro indeed formalized the idea of synthesizing programs as *debugging the empty program* [23]. But when developing a program, at most points, S is a previous iteration of the program and S' the next, where D itself is approximated in a series of iterations each times with more features and (hopefully) fewer bugs.

Dynamic Software Upgrade (DSU) XXXX

Part III

Runtime Applications

Chapter 6

Migration

6.1 Achieving Migration

6.1.1 Intent

The main innovation of our protocol for using first-class implementations was runtime evaluative reflection, the non-trivial part of which is the `simulate` functions (both on nodes and on arrows). Now what good is a simulation of a concrete computation state for? Most obviously, interpreting it up into an abstract computation state. And what good is an abstract computation state for? Most obviously, implementing it down into a concrete computation state. Of course if it's being implemented back the very same way, nothing was changed, nothing was gained, and it's just a waste of effort. However, the implementation going down can be different from the inverse of the interpretation going up, at which point a very useful application is achieved: *Migration*.

Migration is the change of a program's underlying implementation *while it's running*. In systems lacking either runtime evaluative reflection (`simulate`) or compile-time abstractive reflection (`observe`), migration is considered an impressive stunt: a metaphor often used is that it is akin to *changing a car's engine and wheels while it's going full speed on the highway*. But on systems that *do* possess these two kinds of reflection, migration as such is logically trivial. Of course, making migration *efficient* can be a lot of work; but not *more* so than making any software efficient.

In this and subsequent sections, we'll discuss Migration from the point of view of a reflective system, and contrast it with particular applications of Migration that are traditionally implemented without using general reflective mechanisms.

6.1.2 Migration Formally

Migration can be represented with the following diagram, in which an abstract computation A is initially implemented by a concrete computation C, but while it is running, C is interrupted and is replaced by a different (presumably more suitable) concrete implementation K:

$$\begin{tikzcd}
a \arrow[rrr, "A"'] & & & a'' & & \\
c \arrow[u, dashed, mapsto, "\Phi"'] \arrow[r, "C"'] & c' \arrow[r, "C"'] & c'' \arrow[ur, dashed, mapsto, "\Phi"'] & & k \arrow[ul, dashed, mapsto, "\Psi"'] \arrow[r, "K"'] & k'
\end{tikzcd}$$

The computation starts in abstract state a in A, implemented by concrete state c in C. The concrete computation C is run, and interrupted while in state c'. Observability is then used to stabilize the concrete computation to an observable state c'', that can be interpreted as the abstract state a''. Then this state a'' can be implemented with the state k of a different implementation K, and K is run until it finishes or is interrupted.

Note that this diagram is just the same as Observability for the implementation Φ^{-1} of A with C, followed by Totality for the implementation Ψ^{-1} of A with K (at least, restricted to the domain of Φ^{-1}), followed by running the implementation K:

```
&migrate : ∀ {A C K : Computation} {Φ : Implementation A C}
  {Ψ : Implementation A K} C.Node --→ K.Node
&migrate = Ψ.&run ∘ Ψ.&implement-● ∘ Φ.&interpret.● ∘ Φ.&observe.●
```

6.1.3 Comparison with ad hoc Migration

The key step that most existing systems seem to be missing is the use of Observability. Without Observability, migration is altogether impossible.

Now, many systems do implement some kind of migration at a fixed abstraction level; for that they reinvent an ad-hoc form of Observability, just for that level of abstraction — typically one very low-level virtual machine (such as 370, x86, JVM or .NET), Then, their main problem is that, not having a general framework to think about runtime semantics, they cannot automate very little of the development and verification for semantics-preserving ante-programs; they have to fight very hard to ensure that their migration doesn't corrupt the meaning of programs in some subtle way. And at the end of all their efforts, they implemented just on kind of migration at one single level of abstraction between a small fixed set of implementations.

By contrast, using a general framework for reflection, it becomes clear what the contract is, and how it can be factored into many simpler composable layers. Formal verification becomes possible, and Migration can be implemented in a way that is correct by construction, whether the logical verification is fully automated, just partially assisted by a type system rejecting large classes of errors, or left informal in terms of manual code organization. And the result of the effort is generally usable beyond a single use-case.

6.1.4 Example: Migrating Processes between Machines

An obvious example of migration would be to migrate running processes from one machine to another. In traditional systems without a reflective protocol, this would be extremely expensive: with proper hardware and software virtualization, and with heroic efforts, a snapshot of the machine at a very low-level might be taken and moved to run in another identical machine. In a reflective system using our protocol, migration would come for free, without any special hardware support, or additional software trickery: the abstract state of the computation can be recovered at any time to be migrated to a different underlying machine, and that machine not only doesn't have to be identical, but can sport a completely different kind of processor (e.g. ARM vs x86 vs a GreenArray processor vs ad hoc FPGA or something completely different).

Whereas a traditional system could always emulate one processor with another, this would come at a steep price, and every successive migration to a different processor would incur a large slowdown factor, making the scheme wholly impractical. By contrast, a reflective system can always use all the best known compilation techniques to implement the abstract computation specified by the user in the most efficient way for the target computer (as far as can be known). Indeed, migration might consist solely in updating an existing computation to use a newly released compiler with an improved set of optimizations (or try it out, and revert if there was no improvement, or try out many different combinations of flags, and see which work best, all automatically).

Of course, with a some special purpose case of Observability, a traditional system could decide on one processor as "the" reference virtual processor (e.g. IBM 370) then make sure that whatever real processor emulates it, it's always possible to recover the "abstract" state of the computation at the level of abstraction of a reference processor. Thus, there would be no compounding of slowdown factors at each migration; instead each target processor would have one slowdown or speedup factor with respect to the reference architecture (that would also vary with the application depending on which architectural features it most uses). But then, what was achieved in these traditional virtual machine systems is a particular subset of our reflection protocol, albeit one limited to a particular low-level processor architecture as being "the" "abstract" computation that the system cares about, with no reflective support for computations at higher (or lower) levels of abstraction, at the cost of an expensive mainframe architecture (and accompanying software licenses). Reflection offers the promise of all the benefits of virtual architectures, and much more, at whichever level of abstraction users actually care about, without the cost of expensive hardware architectures and accompanying software.

6.2 Making Migration Efficient

6.2.1 Optimizing the Abstract away

Migration may be logically trivial to achieve with the above formula, but a naive direct implementation thereof would be particularly slow: indeed, it concept involves reifying the entire state of the application, then processing that entire state include all runtime data to interpret it into a more abstract representation, then processing that to recompile it back into a new concrete representation, that is then executed. On an application that processes mega-, giga-, peta- or exa- bytes of data, a direct application of this recipe, that eagerly computes of each of these steps on a sequential machine (what more multiple times, for each intermediate layer of representation) cannot possibly be quick, much less instantaneous. Maybe the framework will ensure correctness by construction, and the application itself can never notices that its implementation has changed underneath; but the user will notice that the application is frozen for seconds, minutes, days, years, while the data is being processed.

However, if the above formula is seen as a logical specification, the implementation of which is subject to optimization, then it can indeed be used as the template to generate migration procedures that are correct by construction. The many steps can to be merged, the intermediate representations as e.g. syntax trees can be deforested away[27], remaining virtually conceptualized, but never actually realized. The diagram for optimized migration is as follows: where migration is represented as the squiggly association arrow from c'' to k by M:

$$\begin{tikzcd}
a \arrow[rrr, "A"'] & & & a'' & \\
c \arrow[u, dashed, mapsto, "\Phi"] \arrow[r, "C"'] & c' \arrow[r, "C"'] & c'' \arrow[ur, dashed, mapsto, "\Phi"] \arrow[r, squiggly, mapsto, "M"'] & k \arrow[u, dashed, mapsto, "\Psi"'] \arrow[r, "K"'] & k'
\end{tikzcd}$$

That arrow is an association arrow, as indicated by its vertical bar at the beginning (as in $\mapsto$). The arrow is also squiggly, indicating that M is not just partial but also non-deterministic. Indeed, $M = \Psi^{-1} \circ \Phi$, and while interpretation Φ from C to A is a (deterministic, but partial) function, Ψ^{-1} on the other hand, as the inverse of a function that is not (known to be) injective, is non-deterministic: there may be many ways to implement a node or arrow in A with one in

K that are consistent with inversing the partial application Ψ. association of one element to another by a function, rather relation of one set to another by a function.

For instance, the initial concrete computation C might consist in interpreting some function as bytecode; noticing that the function is taking time (by profiling the time spent in it, or watching some progress counter), the system might decide to compile that function to native code, (and if that function is still a time sink, it might further spend more time optimizing it). The system with a new function implementation with constitute a concrete computation K, and the transition from C to K is predicated on this transformation not having modified the operational semantics of the program at the level of abstract syntax tree at which the evaluation contract is defined. But that doesn't mean a representation of the program at said abstract level will be generated (then dropped after it is re-compiled); that only means that the logical consistency of the underlying representation with this conceptual contractual level is maintained at all times. Thus, the optimizing runtime monitor, considered as an outside program that modifies the implementation of the program being run, can change the underlying implementation in an incremental, local, way.

6.2.2 Grounded Migration

Some people may object to the above diagrams, or at least find them distasteful or disconcerting, because they display arrows in different categories C and K at the "same" level. What more, the latter diagram also mixes at the same level not a arrow, morphism or function, but an association between nodes of two different categories:

How can that be reconciled with the desire of having at the same level entities of the same nature? The following diagram explains how:

$$\begin{tikzcd}
a \arrow[rrrrrr, "A"'] & & & & & & a'' & & & \\
& c \arrow[ul, dashed, mapsto, "\Phi"'] \arrow[rr, "C"'] & & c' \arrow[rr, "C"'] & & c'' \arrow[ur, dashed, mapsto, "\Phi"'] \arrow[rr, squiggly, "M"'] & & k \arrow[ul, dashed, mapsto, "\Psi"] \arrow[rr, "K"'] & & k' \\
& \Phi, c \arrow[u, dashed, mapsto, "\Theta"'] \arrow[rr, "H"'] & & \Phi, c' \arrow[u, dashed, mapsto, "\Theta"'] \arrow[rr, "H"'] & & \Phi, c'' \arrow[u, dashed, mapsto, "\Theta"'] \arrow[rr, "H"'] & & \Psi, k \arrow[u, dashed, mapsto, "\Theta"'] \arrow[rr, "H"'] & & \Psi, k' \arrow[u, dashed, mapsto, "\Theta"']
\end{tikzcd}$$

In this diagram, we consider that "concrete" computations C and K are actually both implemented on top of the very same hardware, and thus ultimately to the same hardware implementation; but we can also identify a higher level of abstraction above the hardware yet below both C and K, and that's what we'll call H — a common *hypo*-computation. We then consider the "optimizing runtime monitor" as part of the program doing the implementation rather than a separate program watching it "from above" — indeed, it verily *is* an essential part of the hypo-computation H *below* both C ad K. And the way that H implements both C and K is by its state being a pair of some concrete state and of some datum specifying how to interpret the state back into an abstract state: thus c, Φ implements a (through c) and k', Ψ implements a'' (through k). c, Φ and k', Ψ implement states in different computations C and K at the intermediate level, yet are states of the same concrete computation H and implement states of the same abstract computation A. Moreover, the transition from c'', Φ to k, Ψ indeed implements the migration M and is at the same level as the transitions from c to c' to c'' or from k to k'. Indeed, association by a partial and non-deterministic function is of the same nature as the transition function *run* (or *advance*), a process also itself partial and non-deterministic. Therefore, those who find the mixing of heterogeneous entities objectionable may simply factor out the problematic level and write the following simplified diagram (but they are missing out on the fun and on useful insight):

$$\begin{tikzcd}
a \arrow[rrr, "A"'] & & & a'' & \\
\Phi, c \arrow[u, dashed, mapsto, "\Xi"'] \arrow[r, "H"'] & \Phi, c' \arrow[r, "H"'] & \Phi, c'' \arrow[ur, dashed, mapsto, "\Xi"] \arrow[r, "H"'] & \Psi, k \arrow[u, dashed, mapsto, "\Xi"'] \arrow[r, "H"'] & \Psi, k'
\end{tikzcd}$$

Actually, once changes of representation are understood as transitions among others at the same underlying level, then there is no reason why it should be a big and catastrophic event rather than a small incremental change. At the level of H, some changes might slightly modify the state, whereas other changes might slightly modify the representation.

6.2.3 Example: Garbage Collection

A case at hand is incremental garbage collection (GC)$^{\text{[citation needed]}}$: consider an abstract computation the state of which includes a directed graph of objects pointing at each other; consider a slightly more concrete computation where each object is labelled with an address in memory, and additional, "dead" objects may exist, unreachable from those in the abstract graph. A moving garbage collection is a relabelling the object graph, that eliminates unreachable objects by marking them with a special "color" (e.g. black) to indicate they are dead (at which point they don't have any address in memory anymore, or it's no longer meaningful), as distinguished from "white" to indicate they are known to be reachable. A relabelling a graph is a typical instance of migration, and modelling the correctness criteria of a moving garbage collector can thus be expressed as a particular case of migration, reusing the same conceptual model (and, hopefully, proof tools).[1]

An *incremental* moving garbage collector modifies the representation in a way interleaved with computation by the *mutator* (i.e. the rest of the program, that actually implements the abstract computation). One class of moving garbage collection algorithms may break down "white" objects into three distinct colors red, green and blue: at the beginning of a garbage collection cycle, all objects are marked red, indicating their reachability status is under question — except the roots of the collection, that start green (and, if they are fast-changing registers, may remain forever so). when the tracing garbage collector reaches an object, or the mutator (i.e. the rest of the program) modifies it, it is changed to color green and it is added to a green queue; the tracing garbage collector visits objects in the green queue, ensures that all the objects they point to are either green or blue, then turns them blue. When the green queue is empty, the cycle is complete: all objects that are still red are turned black and their memory is recycled since they were provably not reachable; then all blue objects are turned red, and a new cycle is started. The work of the garbage collector as well as that of the mutator are on

[1] As an anecdote for how it is not an abuse to identify Garbage Collection as Migration, in a speech at SPLASH 2010 in Reno, hackers from Linden Lab explained how in Second Life, migrating a script from one server to another server (or to a restarted version of the server) was achieved by reifying (simulating) the script state (using an exception handler to catch and process reification events in each lexical scope), shipping the reified data and reflecting (performing) it back in the new server. They also explained that the very same technique was how the garbage collection of unused memory happened for said scripts: by reifying and reflecting the script state as you migrate it from a server to another server or the very same.

the same level; the mutator advances the abstract computation, whereas the garbage collector advances the incremental migration that is each cycle.

Of course, note that the address annotations and color annotations are largely independent. There are many ways to encode and organize objects in memory even given the previous annotations. Therefore, there is there again the opportunity for many different implementations of what at a higher level of abstraction is the "same" algorithm. And there is the opportunity for migrating from one of these implementations to the next — at runtime.

Ensuring Observability at the abstract level of the object graph is a common problem for programming language implementors writing garbage collectors: whenever an interrupt happens, or a debugger tries to inspect the graph, or a concurrent process wants to access some shared data, or the garbage collector attempts some change how that data is represented, it is important that the program should be synchronized to a "safe point" where computation is observable, at the garbage collection invariants are preserved. GC implementors possess a large collection of tricks, including read barriers, write barriers, memory transactions, locks, and ensuring that you can always roll back from inside critical sections or forward out of them, etc. All these tricks can be viewed in terms of Observability, and most of them can be reused when implementing Observability in a wider context.

6.3 Uses of Migration

6.3.1 Recasting Things as Migration

Once you have a general enough understanding of what constitutes Migration, i.e. a change of implementation strategy at runtime, then a whole lot of computing activities commonly occurring in computers or discussed in computer science start to look like particular cases of Migration.

Process Migration

In the case of distributed system, migration of mobile activities from one host to another, on a same or different architecture. The migrated activities can scale from a single process jumping moving between a handheld device and a desktop computer, to an large set of coordinated servers moving between data centers on different continents.

Garbage Collection

Tracing garbage collection can be viewed as migrating the heap from an old space full of garbage to a new fresh space.

Zero Copy Routing

Large amounts of data can be routed from one process to the next by changing the conceptual ownership of the data pages without copying any data, maybe even without touching mapping tables; at an abstract level the data may have moved, but the actual implementation can be extremely efficient.

Dynamic Configuration

While the program is running, its environment can be change, its inputs and outputs can be reconnected: its "environment variables", the (graphical or text) terminal it's connected to, the sound devices it uses, the network connections to various servers, the logging or debugging settings it uses, many user preferences, etc. — they can all change without the program noticing, having to notice, or even being able to notice, because they are handled wholly by hypo-programs, and a change is just migration from one hypo-program to another.

JIT Compilation

With a Just-In-Time Compiler (JIT), a same function can be implemented differently at different times: in the beginning, it is implemented using a bytecode interpreter; if it keeps getting called, soon the JIT compiler will replace it by some quickly generated native code; and if the dynamic profiler reveals that a lot of time is spent in the function, then that code will in turn be replaced by some aggressively optimized native code. That's migration.

Dynamic type-directed compilation

A dynamic JIT can select completely different representations for general data types, depending on the actually observed access patterns and the actually observed types of various data elements. This can be seen as migrating code from some inefficient or invalidated setting to a new setting that allows speedier or type-correct setting[citation needed].

Database schema upgrade

The data migrates from an old representation to a new extended (or restricted) one. Also file format upgrade, etc.

Software version upgrade

The entire fleet of servers are upgraded to a new version of the software; this can be seen as migration of the service architecture from one implementation to the next — assuming you have a high-level enough view of the program that allows for large incompatible changes in computational behavior to keep satisfying customer need (and satisfy it better).

Refactoring

Similarly, code refactoring, whether statically before it's running, or dynamically while a program is running, and change in data representation and upgrade of virtual machine code can likewise be seen as change in implementation.

6.3.2 A Fruitful Change in Perspective

Even when we're considering but existing applications, and without imagining any kind of "migration" that has never been implemented before, there is are many advantages in view these existing applications from the perspective of Migration.

Correctness

The Migration perspective provides a semantic framework based on which it becomes possible to ensure correctness by construction, whether formally or informally, despite an exponential number of configuration combinations.

Dynamism

The Migration perspective makes it possible *at runtime* to adapt software to relevant changes, without losing any session data, without interrupting fragile interactions or expensive connections, etc.

Retroactivity

Migration makes it possible to apply and combine these many different techniques *after the fact*. Traditional systems demand that developers should specially add support for migration in programs they write, and/or that users should configure configure the execution environment before the program is started, to enable a few specific enumerated kinds of migration. Reflective systems allow migration to be defined and to happen after the program is started, even if the program wasn't specifically written with that kind of migration in mind.

Composability

A general framework for migration makes it possible to mix and match and compose several features developed separately, when traditional systems can only migrate according to fixed combinations of features developed together. For instance, a general approach to migration will allow one to move some arbitrary computation from a laptop to a desktop while keeping its video output to the laptop but redirecting the sound output to some loudspeakers on yet another machine; it might also change its garbage collection algorithm to a real-time algorithm to support music generation (which affects memory representation and code generation), all the while the program keeps running. An ad hoc approach to migration might typically allow one of these changes, at most.

Predictable Cost-Reduction

It is entirely predictable that any *successful* software *will* have to be migrated: data centers will become obsolete, overly expensive, go bankrupt, have accidents; data schemas will have to be upgraded eventually to match a reality that changes in predictably unpredictable ways; users will switch from one device to the next, they will want to share with friends on a big screen videos, music or games that they were previously consuming alone on a small screen; etc. Yet most traditional systems (with exceptions like Erlang[2]) make it extremely hard to retrofit any kind of migration onto computations that weren't specifically designed to be migrated, and thus force enormous future costs on all successful software.

6.4 Managing Migration

6.4.1 Transparent Migration

But in a reflective system, just like in traditional systems, most migration is managed automatically without the user having to explicitly control it: whether memory is migrated from one garbage-collection color to another, or a process is migrated from one server to another, or a function is migrated from one implementation to another, this is usually transparent to the user. Sometimes the migration is directly caused by a user request, such as a user deciding to disconnect his phone from his laptop, his laptop from his desktop, or his desktop from some server. But user requests are only a small fraction of the changes in the environment that a system usually takes into account when deciding what to migrate where and when.

Transparent migration can be illustrated with the following tower diagram:

$$\begin{tikzcd}
& A & \\
& U \arrow[u, dashed, "Gap"] & \\
& V \arrow[u, dashed] & \\
M \arrow[ur, dashed] & & M' \arrow[ul, dashed, blue] \\
C \arrow[u, dashed] & & C' \arrow[u, dashed, blue] \\
& H \arrow[ul, dashed] \arrow[ur, dashed, blue] \arrow[uuuu, dashed, bend left=90, "\Phi"] \arrow[uuu, dashed, bend right=90] &
\end{tikzcd}$$

The user specifies an abstract computation A, whether via some source code S or otherwise via a series of interactions. Since the computer cannot directly express A, the user actually specifies the computer-understandable computation U that implements A (with a semantic *Gap*), that he cares to interact with. The level of abstraction of U is also specified, implicitly or explicitly, via the user's interactions.

Along the tower of implementation that implements A onto the hardware H via U, an intermediate abstraction level V offers some kind of virtual machine. In the tower of implementation layers between V and H, at some point in time are the levels M and C (the letters stand for middle and concrete; in practice there may be any number of implementation layers in there). Then some meta-object decides to adapt to some change in the environment, and migrate the implementation from M and C to alternate levels M' and C' (there again in practice, it could be any new number or new implementation layers). The user doesn't see any change, because the system has maintained the semantics of U: the overall implementation Φ of U with H is unchanged. Actually the system even maintained the semantics of V underneath U. But unbeknownst to him, the implementation was changed beneath his feet.

Migration is transparent to the user.

6.4.2 Implementation Meta-Objects

The above paragraphs suggest the existence of entities we called the *implementation meta-objects*: computations that control the implementation of other computations: the garbage collector, the process scheduler, the JIT compiler, etc.

When considered as *internal* to the computation specified by the user, these implementation meta-objects appear as details added in hypo-computations, as part of the kernel or monitor that controls the low-level computation and selects between implementations of the high-level computation.

When considered as *external* to the computation specified by the user, they are recognized as entities that are interesting in their own right, as ante-computations that manage the implementation of the computation as a post-computation.

Then, it appears that some users may specify some computations, and that the same or other users can independently specify implementation meta-objects or *strategies* that will control the implementation of the former computations. The constraint that said strategies shall correctly implement the semantics of the user-provided computation may or may not be automatically enforced.

Most existing systems are not capable of expressing the constraint, much less enforcing it. They mostly do not even try, because the cost of developing proven-correct programs is considered prohibitive compared to the value at risk in most applications should correctness be found lacking. But as software applications scale to ever more valuable activities and sizes, with ever increased catastrophic risk from failure of their small shared core, the fixed cost of proving at least this core correct will soon be overwhelmed by the cost of leaving it at risk.

In any case, the notion of managing migration naturally leads to the question of an architecture to organize these implementation meta-objects. See part IV, *Architectural Implications of First-Class Implementations.*

Chapter 7

Natural Transformations of Implementations

7.1 Automatable Transformations

7.1.1 Code Instrumentation

Many common tools used by developers in refining their programs rely on some kind of code *instrumentation*, whereby additional instructions are systematically inserted by the compiler to achieve certain effects, typically to give the user more visibility or more control over the execution of a computation: tracing, logging, filtering or intercepting some events such as function calls, trapping when some situation is detected, single-stepping through the execution, profiling the performance of various code paths, accumulating usage statistics based on which to drive future (automated or manual) optimizations, computing code coverage of various test suites, etc.

Unhappily, each of these tools only ever works at a single level of abstraction, usually for one general purpose programming language that the tools authors cared to painfully implement the instrumentation for. To be general purpose enough as to make the instrumentation effort worthwhile, said level of abstraction ends usually up being several layers of abstractions below what end-users actually care about. For instance, if the computations that one cares about are written in some "configuration language" itself implemented by an interpreter written in C++, then all the wonderful profiling tools available for C++ won't help the user find out which part of his "configuration" is abnormally slow, only which parts of the interpreter are most used while running it — which does provide a hint indeed, but a flimsy one, compared to the profiler could tell if the program were written directly in C++. Conversely, if the phenomenon being studied happens below the level of abstraction provided by the instrumentation (say, a bug in the compiler, the kernel, or the processor itself), then the instrumentation is also much less useful than if it happens at the designed level of abstraction, since the tool won't be able to express the precise anomaly that is occurring.

In either case, all these instrumentations are tied to one level of abstraction, and not very usable if at all to help address software issues at a different level. Since most issues are at a different level, the return on investment of the existing tools is low, and equivalent tools are sorely missed at most levels of abstraction that users care about. Some developers may try to force development to happen in a language that is fully supported by instrumentation tools, but then may face complexity explosion, robustness issues, security issues, etc., as said language is a mismatch to the task at hand in terms of expressiveness.

7.1.2 Abstracting Instrumentation

Now, if you consider implementations as first-class entities, it becomes possible to automate, generalize and compose code instrumentation, and make it usable at whichever level of abstraction a developer is interested in. The many known techniques that people traditionally implement the hard way in a few special cases, that usually can't be combined; with first-class implementations, these techniques become general purpose; they are readily available at whichever level of abstraction developers need them; they can be composed, their inputs can be the result of composition, and their outputs can partake in compositions, etc. One instrumentation technique can be implemented once or twice, and become available to any program in any language. Our theory of implementation makes it possible to commoditize instrumentation practices that would otherwise remain artisanal.

The key is first to understand an instrumentation as a *Natural Transformation* of an implementation into a modified implementation (with a twist explained below), and second to abstract away the specific implementations on which a transformation operates and make them parameters of a generalized instrumentation technique. Thus, instead of being able to trace or profile function calls in C++, you'll be able to trace or profile function call in any language at any abstraction level of your development environment for which you can describe some piece of the operational semantics as calling a function or returning from it.

7.1.3 Natural Transformations

In category theory, if ϕ and ψ are functors between the categories A and C, then a natural transformation η from ϕ to ψ is a family of morphisms that satisfy two requirements. The natural transformation must associate to every object a in A a morphism $\eta_a : \phi(a)ß\psi(a)$ between objects of C. The morphism η_a is called the component of η at a. Components must be such that for every morphism $f : aßa'$ in A we have:

$$\eta_{a'} \circ \phi(f) = \psi(f) \circ \eta_a$$

The notion has to be generalized a little bit to apply to implementations, that are inverse of *partial* functors (that in category theory can be viewed as spans or as profunctors). Let Φ^{-1} and Ψ^{-1} be implementations of A with C, then a natural transformation η from Φ^{-1} to Ψ^{-1} is a family of arrows that satisfy two requirements. The natural transformation must associate to every abstract node a in A a morphism $\eta_a : \Phi^{-1}(a)ß\Psi^{-1}(a)$ between objects of C. Or more precisely, since Φ^{-1} is non-deterministic, for every c such that $\Phi(c) = a$, there exists d in C such that $\Psi(d) = a$ (we write $d = \eta_{a,c}(c)$), and there is a $\eta_{a,c} : cßd$, the component of η at a, c.[1] Components must be such that for every arrow $f : aßa'$ in A, we have:

$$\eta_{a'} \circ \Phi^{-1}(f) \subseteq \Psi^{-1}(f) \circ \eta_a$$

[1] Note how the quantifier for d is existential and not universal. Indeed, if Φ adds some state constraining arrows such that the state of the domain must be prefix to the state of the codomain (i.e. there is a functor from C to that state category), amd if Ψ similarly adds another state that may or may not be orthogonal to that of Φ, then η clearly cannot map arbitrary pairs c, c' to arbitrary pairs d, d': c' can only be reached from c if it has compatible Φ-state, d' can only be reached from d if it has compatible Ψ-state, d only makes sense from c if their Φ- and Ψ- states are compatible, and d' only makes sense from c' if their Φ- and Ψ- states are compatible.

Also note how this existential quantifier means that Ψ is defined wherever Φ is defined, and that underlying it is a function. We could weaken that by defining "partial natural transformations", or "natural pro-transformations", where the transformation is a partial functor or a profunctor, instead of a functor.

More precisely if $g : c\text{ß}c'$ and $\Phi(g) = f$ there exists $h : d\text{ß}d'$ with $\Psi(h) = f$, such that $d = \eta_{a,c}(c))$, $d' = \eta_{a',c'}(c')$, and $\eta_{a',c'} \circ g = h \circ \eta_{a,c}$. The transformation is surjective if the converse condition holds, that if $h : d\text{ß}d'$ and $\Psi(h) = f$ there exists $g : c\text{ß}c'$ with $\Phi(g) = f$, such that $d = \eta_{a,c}(c))$, $d' = \eta_{a',c'}(c')$, and $\eta_{a',c'} \circ g = h \circ \eta_{a,c}$.

The component of η at a, c represents the removing the effects that Φ^{-1} has that Ψ^{-1} doesn't have, and adding the effects that Ψ^{-1} has that Φ^{-1} doesn't have. However, in the case of total deterministic natural transformations (the classic notion from category theory), any adding of effect must be trivial (cannot add information), whereas removal of effects can be significant (can, uniformly, lose information). Thus we find that once again, the instrumentations that we are interested in go against the functorial flow, and are actually the *dual* of natural transformations. Said otherwise, a code instrumentation usually isn't a natural transformation, but the forgetful functor that erases a code instrumentation is.

Also note that remarkably, these dual of natural transformations are between implementations, with are dual of interpretation (partial) functors: indeed in all our implementation properties, we previously emphasized how it was the inverse of an implementation that was functorial, and as far as specifying correctness goes, we mostly followed the interpretation arrows "up" from concrete to abstract; but with these natural transformations, we actually follow the implementation arrows "down" from abstract to concrete then apply the inverse of a natural transformation between two such downward implementation profunctors. Thus in the "cone" a, c, d the common origin is the abstract node a and the two distinct images are the concrete nodes c and d (via distinct implementations). The instrumentation is the inverse of the natural transformation, because it is "natural" (in a category theory sense) to lose the information from the instrumentation and retrieve the uninstrumented program; but it is not (usually) natural to add that information.

A trivial example would be to consider a system with one node and an infinite loop, and an instrumentation that counts the number of times the system went through the loop. If the count is part of the concrete state, then the instrumented system distinguishes nodes by count; yet they all map to the same abstract node.

Cite Conor McBride's 2010 paper on Ornaments or this 2013 follow-up: https://arxiv.org/pdf/1212.3806.pdf

7.1.4 Instrumentation vs Aspect Oriented Programming

Note the relationship between Instrumentation and Aspect-Oriented Programming (5.3.1): the "events" that may be traced or logged or otherwise instrumented are called "join-points" in Aspect-Oriented Programming; in both cases they are typically function calls, but may be arbitrary points during the evaluation, as long as the developer can somehow recognize and define them as such. The developer can then specify a set of join-points, called a pointcut in AOP parlance, at which some code, called an advice, will be run before, after, around or instead of the code normally run at the join-point (though hopefully preserving the high-level invariants of the code as specified by the developer, whatever they may be).

The difference is that AOP is usually used somewhat more statically to define the program that is to run, at which point there is more freedom for the advice to override or modify the behavior of the original program in ways that do *not* fully preserve its semantics, but transform it into something better suited to the developer's needs. By contrast, the kind of tracing, logging, and other forms of instrumentation that are used outside of AOP typically are configured in a more dynamic way, and do not interfere with the semantics of the non-instrumented program in any substantial way. Yet, in the end, it is the same set of techniques, just viewed from a different angle and applied at a slightly different time by different people within slightly different constraints.

7.2 Common Transformations

7.2.1 Tracing and Logging

Tracing and logging are instrumentation techniques whereby some "events" are recognized, either nodes or arrows in the operational semantics, upon which the instrumentation adds additional code that displays a summary of the event to the developer, or stores it in a log file for further analysis. An event is typically a function being called with some arguments, or the same function returning a value to its caller; but an event can also be a variable being modified, some system call being issued or returned from, or special I/O action taking place, etc.

Tracing is often used while debugging a computation, to determine whether the control flows as intended or fails to, and if so at which precise point it fails. Logging is often used to audit the performance or security of running services, or to obtain a baseline to which to compare the evolution of the program and of its test runs.

Now, a good interactive development platform will let you enable and disable tracing for individual functions, maybe even allowing the tracer to inspect the event data whenever deciding whether to log it or not, and which details to include or not in the trace. All the interacting developer has to do is name some functions; then any function call event involving these functions will be traced; or will cease to be when the developer decides not to trace them anymore.

In the case of logging, the developer explicitly defines the events in the source code; however the logging infrastructure can typically be configured independently from the program and often dynamically; it can then decide where to route the log messages, depending its own criteria, based on their content and on various annotations provided by the developer, that detailing their degree of seriousness, confidentiality, etc.

Now, whether events are displayed to the user or saved in a file, if you consider that tracing and logging are sending output to a channel invisible to the computation's abstract semantics, then the addition or substraction of this output trivially preserves said abstract semantics. Adding the tracing or logging of various functions is a Natural Transformation that adds these systematic effects; removing the tracing or logging of the same functions is another Natural Transformation, a Forgetful Natural Transformation, that removes these effects (as if redirecting the invisible channel to the bit bucket).

7.2.2 Single-Stepper

A single-stepper instruments a computation so it will stop before every step, and wait for the human operator (or a controlling program) to tell it to continue processing. Some single-steppers rely on hardware support to do the stepping itself; other steppers do it all in software. If tracing adds outputs, single-stepping adds inputs: between every two consecutive steps of computation there will now be a wait on external input.

A single-stepper allows a human to inspect the state of the computation and visualize its progress, either to learn the proper behavior of a working computation, or to understand a debug a buggy computation. Single-stepping may also be a preliminary transformation that later enables further transformations to e.g. simulate state changes between every consecutive steps, keep running until the very first time some condition is met, synchronize two slightly out-of-synch copies of a computation, etc.

Of course, there may be infinitely many interesting notions for what "a step" is. One view may see the evaluation of some function as a single step, whereas another view might see it as decomposed into millions of steps.

Given any implementation with a Strong Step Preservation property (see 3.2.3), there is a simple Natural Transformation from the initial implementation to one that requests input between every step, on a channel that is invisible to the abstract program. And given the single-stepping implementation listening to such a channel, we can retrieve the original implementation with a forgetful Natural Transformation where the invisible channel is bound to an ever-ready channel that always requests the continuation of the computation.

7.2.3 Omniscient Debugging

Once the computation has been divided in steps, it is possible to record at every step the changes in the state of the computation. And it becomes possible to keep a relatively compact, indexed representation of all these changes, that one can search and query according to various criteria such as: When was some variable bound to some unacceptable value, and what were the other variables in scope bound to at that time, and the intermediate values and continuations from surrounding expressions being computed? Through what chain of intermediate transfers was one value transfered from one place to another where it shouldn't have landed? What if you replay from one point with this change in variable bindings, or that change in code? What if you undo or redo smaller parts of the computation until you pinpoint what exactly went wrong and where?

Debugging based on such a complete model of all the steps during a computation is called *Omniscient Debugging*, also known as *Time-Travel Debugging* or *Reverse Debugging*.

There is another natural transformation from an implementation that preserves steps to one that can record and index the computation at every step. And this transformation can be abstracted given a description of an abstract computation, a model of the space of computation state and how to index or query it, and a specification of how each step affects the model. Then, instead of having Omniscient Debugging just for the low-level virtual machine several levels of abstraction below one's application, one could achieve it a exactly the level of abstraction that one cares for.

7.2.4 Profiling

Instead of tracking every single change in the state of the computation, it is cheaper and sometimes sufficient to track only a synthetic summary thereof: how much memory was used, how much cpu time was spent, how many blocks were read from disk, how many packets were sent or received on the network, what operating capabilities were used, how well various caches behaved, how many errors of each kind were caught, etc. When this resource accounting is correlated to which parts of the code or data caused the resources to be consumed, it yields *profiling* information. Profiling can be crucial to detect and address performance bugs and security issues, to direct the developer's focus to parts of the system with the most urgent issues, or to forecast the usage of some resources and provision them accordingly.

There too, the collection of Profiling information, being a reduction of the information collected via Omniscient Debugging, can be viewed as a natural transformation of implementation from one that doesn't collection information to one that does — or once again, back, via a forgetful transformation that drops the collected information. This means that you can automatically extract a useful profiling tool applicable to an arbitrary DSL for which you possess an implementation with a suitably declarative description. And there again, the logical specification of the profiling tool, that can be made correct by construction, can yield an efficient implementation of the tool, using a suitably declarative programming language, with a compiler capable of deforestation of intermediate representations: specify the reduction in terms of the richer Omniscient representation, but compute and collect only the much cheaper reduced representation.

7.2.5 Code and Data Coverage

When the reduced information that you profile and index is about which pieces of code (respectively data) were used during a collection of (test or actual) runs, what you have is code (respectively data) *Coverage*. Coverage is usually profiled systematically using a single-bit count, or indexing for each piece of code (respectively data) covered some summary of the context in which it was used; but it could also be profiled statistically using approximated frequencies.

Given a code base, coverage allows you to improve your tests, to make sure your code is well-tested and will not experience regressions. Given a set of tests, Coverage allows you to safely remove dead code (resp. data), which can simplify your codebase and enable refactorings formerly blocked by the dead code (resp. data). Given some actual usage data, coverage allows you to determine how you can modify your application to better suit user demand, by simplifying away overly complex parts of the code, or elaborating overly undevelopped parts.

Constructing coverage tools in a declarative way makes it possible for any program in any language to benefit from coverage, at whichever level of abstraction developers care about.

7.2.6 Access Control

A different kind of instrumentation consists in filtering certain subcomputations such as system calls, input/output, access to certain variables, reduction of certain terms, etc., to ascertain that the computation is authorized to run those subcomputations, that it possesses proper access rights, credentials or capabilities, that it isn't trying to use a forbidden or dangerous combination of parameters, etc. In the context of this *Access Control*, system calls are not limited to calls into "the" operating system kernel, but can be an arbitrary interface between a "user" space and a "system" space, as delimited by the developer to distinguish between what is under the control of an relatively untrusted user, and what is fully under control of the "system".

As usual, writing one set of access control tools once makes them available at all levels of abstraction. However, in the case of security, any savings in terms of engineering resources pale before the ability of describing and automatically enforcing security properties at the suitable level of abstraction for each program: having the enforcement focus at the wrong level, or having to manually translate the enforcement between two levels of abstraction (and what more manually maintaining it as the code evolves) pretty much guarantees that the software *will* be subverted.

7.2.7 Concurrency Control

Parallelizing a computation written in an otherwise sequential language based on declared or inferred dependencies between subcomputations, or serializing a computation written in a concurrent language to achieve a deterministic result can both be seen as Natural Transformations that preserve the more abstract semantics of the computation. By specifying computational semantics that are compatible with the considered reordering of subcomputations, programmers can thus decouple the overall meaning of the computation from the evaluation strategy that may be used to compute it, depending on the situation.

A software debugger, a hardware FPGA or ASIC, a multicore computation by a trusting developer, a single-threaded evaluation by an untrusting server, a massively distributed computation in a private farm, an interactive demonstration in a browser, etc.: They each provide a different context in which the very same computation can be implemented with very different characteristics with respect to concurrency as well as other settings.

There exist many tools to automate concurrency control of some or some other specific language, often specially crafted for the purpose. These tools could be made more general with a framework for first-class implementations: by making explicit the dependencies in the flow of values being computed, and making these dependencies an explicit parameter of the control tool, the tool is then factored into a general concurrency-handling part that can be reused on any computation for which the dependencies are specified, which is applicable to a wide variety of programming languages.

7.2.8 Optimistic Evaluation

Optimistic Evaluation is natural transformation of implementations, whereby speculative fragments of computations are evaluated before some information is known as to whether said fragments are valid or not. If at some later point it is found that the premise of the fragment was invalidated, while the fragment is still running or after it has completed, then the intermediate or final results of this fragment are thrown away, and any side-effects by this fragment are reverted (which in particular means that speculative fragments may not issue irreversible side-effects).

Optimistic Evaluation can model many things from user interfaces and online games (assuming that the behavior local or remote user could be correctly predicted), to in-memory or persistent transactions (assuming that the current transaction will complete successfully), to CPU architecture (assuming that branches can be predicted). As usual, by framing these phenomena as special cases of a common pattern, many strategies can be shared for either implementing these techniques or proving their correctness properties.

7.2.9 Virtualization

In Virtualization, a universal concrete computation C used as the substrate for a lot of abstract computations (even "all" of them), itself has an implementation on top of C (or some variant C'), allowing many computations to run concurrently, and to interact while instrumented in arbitrary ways. Moreover, the transformation from a computation A running "directly" on C to a computation running "virtualized" C in the context of some given management software, is itself a natural transformation (that is, assuming that $C' = C$ indeed).

Now, while Virtualization in general allows for arbitrary kinds of instrumentation, specific virtualization techniques may only make some kinds of instrumentation easy and/or cheap, while most remain hard and/or prohibitively expensive, even though other virtualization techniques could enable them. First-class computations are a generalization of virtualization, so that it may happen at arbitrary levels of abstraction, rather than always at the level of zeros and ones of an extraordinarily complex yet desperately low level of abstraction of modern microprocessors. Thus, instrumentation can happen at the level of concepts that the user cares about, rather than at a lower-level where users have to manually manage the correspondence, with a lot of work, a lot of additional opportunities for mistakes, and a lot of additional opportunities for being misled by an adversary. First-class computations therefore make it easier and cheaper to express the virtualization architectures that users care about.

7.3 Orthogonal Persistence

Orthogonal Persistence can be better supported by a family of transformations opposite natural transformations. This largely unknown, misunderstood and undervalued concept requires a section of its own; only at the end of it can we explain why transformations of first-class implementations can make Orthogonal Persistence more relevant.

7.3.1 Orthogonality between Persistence and Semantics

Persistence is the property of a computing system in which all data objects (including code) persist until they are no more needed by the users — neither before, which would make the system fail, nor after, which would waste (eventually all) the system's limited resources. Persistence is meaningful in the context of adverse events that are more or less likely to happen: system shutdown, power loss, hardware failure, theft, tampering by vandals, etc.

Orthogonality means that two aspects are independent from each other — in this case, the persistence of data, and the evaluation semantics of the code. In other words, users and programmers who manipulate some data should not usually have to worry about the persistence of some data, they should only have to care about its structure and meaning, while the system will implicitly ensure the persistence of all data objects for them. Orthogonal persistence is when users don't need to specify anything for all data objects to persist[6]. Users and programmers might have to explicitly specify or alternate settings for persistence (or, sometimes, non-persistence), where performance, robustness or confidentiality matters. By default, though, all data is persisted without any user or programmer intervention.

7.3.2 Two Stories about Orthogonal Persistence

Here are two short stories to illustrate the idea. When my mother first used a computer, she spent the whole night entering in a database (dBase II) the references of books in her library. When she got too tired, she shut the computer down and went to sleep. The next day, she turned her computer back on, and the data had disappeared! She didn't ”know” she had to explicitly ”save” the data before to quit the application. Well, *persistence* means that the data would have been still there despite the quite expected computer shutdown. *Orthogonal* persistence means that she wouldn't have had to care about it.

At about the same time, I had a wonderful battery-backed programmable pocket calculator (an HP28C), that never lost its data during normal operation; actually it would lose its whole patiently typed programs when the battery die, or if using the `SYSEVAL` escape to write buggy programs in assembly; and there were no means to backup data to tape or to disk to restore data after it was lost. So that my calculator's memory was orthogonally persistent against high-level software bugs, regular power-off events and scheduled battery replacement, but not against complete battery loss or low-level software bugs — and so it was ultimately unsatisfying.

7.3.3 Orthogonal Persistence as a Natural Expectation

Orthogonal persistence is quite a natural expectation, because this is exactly what people need when manipulating objects, as what counts to serious people is not (just) the fun they have during a computer session, but the work they accumulate during successive sessions. Untrained people, like my mother was, expect data to be persistent, and hope to have to care as little as possible about keeping it persistent. Indeed, they are not generally qualified to deal with technical aspects of this persistence when they aren't orthogonal; and progress, in computer systems like elsewhere, is in not having to care.

If you're not convinced in orthogonal persistence being a natural expectation, imagine that there would be two kinds of papers, one that would persist, the other that would self-destroy after it's no more on the top of your desk; you could use the latter for short-lived drafts, but you'd use the former for anything that has any worth; and even when you're not writing things that ought to last, you might be using the persistent paper, just because it might unexpectedly turn up as more valuable than initially expected, and you don't want to waste your precious mind resources at assessing the real shortlivedness of your information, nor at making a copy: your time is much more precious than the paper. So the only time you use short-lived paper would be in specific, scheduled events, as part of streamlined business use (for instance, when exchanging passwords or processing secret self-destruct messages, or when computing intermediate results destined to be thrown away).

The same should eventually happen with computers: only as part of developing optimized applications would people ever care about using memory that isn't orthogonally persistent.

7.3.4 Traditional Manual Persistence

Most traditional computing systems do *not* support orthogonal persistence at all; instead, users must manually manage persistence of their data.

Typically, most state will only persist as long as an application is "open"; but to "close" the application is a regular, frequent, event: the user is expected to do it eventually, and may easily do it by mistake; an automatic system upgrade, necessary for security purposes, may force it to happen; all too frequently, the application may crash, or the graphical interface; an out-of-memory event triggered in one application (e.g. a web browser) may cause other applications to die, or some essential system service; or even without any process crashing, memory or other resources may leak until the system will thrash like mad and crawl to a halt, and the user will restart it.

Thus, most state fails to persist at a scale directly visible to the user, who is supposed to explicitly manage the persistence: "save" state into files, move these files around, make local copies, transfer them to other machines, etc. Making remote backups is an important responsibility of the user, often ignored at the user's peril, despite the foreseeable high probability of the machine experiencing at least one of a catastrophic software failure, a hardware break down, a theft, or a takeover by hackers, over the few years of its lifespan.

Traditional database servers do offer persistence services for structured data; but this persistence can only be seen as "orthogonal" for the "stored procedures" written in each of their builtin language. However, these programming languages that are generally too limited, too awkward or too expensive to use for writing complete general-purpose applications.

Yet manual persistence is usually expensive, error-prone and insecure: an old 1978 IBM report once evaluated the overhead of explicitly marshalling and unmarshalling data structures to save and restore or send and receive them to waste 30% of total program code, not taking into account checking, converting, or recovering data. It is unclear that the situation has substantively improved since; meanwhile, parser and unparser issues account for a large number of security advisories.[citation needed] What more, users must still constantly worry about saving their data, then retrieving their data, or safely sharing it between their many devices.

Modern applications tend to hide the necessity of explicit saving from end-users in the simple case of configuration changes, and to offer "recovery" options for documents not explicitly saved when the application previously died. However, they fail to relieve the user from having to handle any but very simplest cases of persistence; and they do it at great cost for developers, who keep using the same underlying programming model of manual persistence.

7.3.5 The Cost of Orthogonal Persistence

Many systems have successfully implemented Orthogonal Persistence since as far back as the late 1970s[19, 6]. Yet the feature never made it to mainstream operating systems and programming languages. It is interesting to speculate on the reasons they failed to reach wider use.

One argued obstacle to the adoption of Orthogonal Persistence is performance, or, more broadly speaking, costs. But the problem is not that Orthogonal Persistence as such costs a lot to implement: the price of automating persistence can be quite small, and arguably smaller than the price of dealing with persistence manually. However, Orthogonal Persistence does make low-level errors extremely undesirable: memory corruption, type mismatches, bad indexes, resource leaks, fork-bombs, security breaches, etc., will persist and permanently corrupt the system state. It is achieving safety against these low-level errors that indeed has a large negative impact on the cost and performance of computations, compared with running wild with unsafe languages like C or C++ and letting programs crash without corrupting any essential persistent state.

For many decades, the cost of safety made it not competitive compared to recklessness, at least for mainstream software. The market demanded compilers for low-level languages yielding faster code, mainly C and C++, even at the expense of catastrophic loss of safety and correctness in the inevitable presence of bugs. Consequently Orthogonal Persistence was not competitive compared to manual persistence. And for those decades, operating systems and development environments have therefore evolved in a way that is hostile to adding Orthogonal Persistence and makes adopting it costly out of path dependence. Yet, nowadays (late 2010s), most of the price of safety is already being paid by most software: most software use "scripting" languages, typesafe languages, or languages otherwise "hosted" on a typesafe virtual machine (JVM or CLR). And even when these technologies do not currently eliminate low-level failures, they already pay the most of the price for this elimination. The time may thus have come to reevaluate Orthogonal Persistence as a feature to include into operating systems and development environments.

7.3.6 Development Ecosystem for Orthogonal Persistence

Orthogonal Persistence requires developers to use a safe language, but that's not all. It also requires that this safe language's implementation should cooperate with the persistence engine. And it also requires that the libraries, the colloquialisms, the coding patterns, etc., should play well with Orthogonal Persistence rather than against it. For instance, there can be no more expectation that restarting an application or rebooting a computer will be a panacea that cures all software ailments. The entire programming culture is affected by what managers might qualify as a "non functional" feature, i.e. one that doesn't directly affect any program's user-visible interactions.

Each existing orthogonally persistent system thus comes or came with its own implementation of its own programming language, with its own libraries and programs and environment. Even those that were based on a variant of an existing language (such as Java or C++), had with their own modified implementation, a modified set of libraries and data structures to account for a modified coding style, new concepts to address new concerns (such as transactions, schema upgrade, versioning, etc.), and new restrictions as to what is culturally acceptable to do in an application.

This means that these systems intrinsically cannot reuse existing programming language ecosystems as is. Which in a way is kind of the point: orthogonal persistence is a whole-system property that is indeed supposed to deeply change the way people write software, *for the better* (claim its proponents). Ultimately, systems with orthogonal persistence compete with entire other *systems*, including operating system, programming language and development platform at the same time.

7.3.7 Historical Failure for Orthogonal Persistence

Now this essential incompatibility and rivality with other systems means that Orthogonal Persistence could not "simply" reuse existing software infrastructure. Implementers of orthogonal persistence, interested primarily in that topic, could implement the persistent aspect of the system very well; but, not being as well versed in the other aspects of programming system, they could not rival with other programming languages with respect to having a good and modern set of features, a quality implementation, a wide range of libraries, static analyzers, dynamic debuggers, refactoring tools, interactive editors, optimizing and parallelizing backends, etc.

To remain relevant, the Orthogonal Persistence community should have tried hard to work alongside prominent members of the Programming Language community, so they could keep up with progress in that community, and not be left far behind along those aspects that they necessarily did not have the resources to all address. For reasons that would require historical investigation, this collaboration did not happen. Maybe the Orthogonal Persistence community failed to recognize that they desperately needed to work with otherwise healthy programming language communities to become and remain relevant. Maybe they recognized this necessity but failed to convince these communities that they had something worthwhile to offer.

Maybe they failed to focus enough on usability aspects whereby orthogonally persistent systems should be easily deployable to entry-level programmers non-technical users. Maybe they required too many system administration skills from new users, raising the barrier to adoption. Maybe they failed to provide good data exchange capabilities, local and remote backup capabilities. Maybe they were lacking in means of safe interoperation with other non-persistent systems. Maybe they did not make it easy for a single system to handle programs that have a wide range of different persistence and performance profiles, easily configurable by the user.

Stepping back, the essential failure of Orthogonal Persistence can be traced to the fact that it had to compete with entire systems, without having the resources to be relevant on other aspects, and without the means to embrace or import existing technology as is. In other words, the adoption of Orthogonal Persistence as a feature wasn't *incremental*, thus making transition too costly.

7.3.8 Orthogonal Persistence via Implementation Transformation

All the above discussion about Orthogonal Persistence was but context to this thesis's claim that First-Class Implementations can make Orthogonal Persistence relevant again, by making it incremental, and user-configurable, rather than the matter of adopting an entire system that uniformly provides of Orthogonal Persistence for every user.

Indeed, Orthogonal persistence can be achieved as a code instrumentation transforming a transient implementation to a persistent one, whereby modifications are journaled and checkpointed. And that code instrumentation is a the opposite of a natural transformation, for which all the previously developed theory applies.

For performance purposes, the persistent transactions are usually evaluated optimistically (see above section 7.2.8), assuming that indeed the computer won't crash before the evaluation results are persisted. This is another co-natural transformation. Of course, some irreversible effects cannot be handled optimistically, but must be synchronized to persistent storage before the computation may proceed to further irreversible effects that depends on the former: talking to external systems that do not partake in the same optimism, controlling I/O devices, printing, using robotic actuators, etc.

Different variants of these transformations correspond to as many different persistence policies: policies with different granularities on the persistent transactions; with different replication strategies; with different latency profiles; with different safety guarantees; with different domains of optimism; with different levels of trust in various vendors, or in various pieces of software or hardware; with different costs, and prices.

Yet in all cases, an implementation is instrumented so that the state of the computation is preserved even in the erratic yet predictably common advent of a power loss or other kind of system crash. Though they ostensibly implement the same abstract computation on top of the same concrete computation, the differently instrumented (or non-instrumented) implementations possess different liveness properties in the presence of a wider domain of errors.

Thanks to runtime migration between carefully developed such implementations, users can therefore achieve precisely the liveness properties they care about, at the price they are ready to pay, without being limited to the choices provided (if any) by those who provide a fixed semantic tower below them, whether system administrators, application developers, library or infrastructure providers, operating system vendors, etc. End-Users can choose their persistence policy independently from implementers of abstraction layers below, and providers of these abstraction layers do not have to care for the details of persistence anymore, except to follow the observability protocol of first-class implementations.

As a hurdle of adoption, there remains of course the question of whether this observability protocol can be promoted in an incremental way. But on the one hand that is another debate, and on the other hand, first-class implementations potentially provide a much higher value than Orthogonal Persistence alone, since it enables not just Orthogonal Persistence, but also a whole lot of other features.

7.4 Erlang-style Resilience

Erlang and its derivatives are somewhat unique in supporting a particularly robust style of programming, where failure is ubiquitously expected as a normal thing to occur [21], and handling these failures by killing and restarting actors is supported deep inside the language and its standard libraries[2].

7.4.1 The Actor Model and its Fragility

Over the decades since it was introduced in 1973 [citation needed], the actor programming model has been implemented in many languages, and most modern languages possess libraries that offer some variant of it: in this model, computation happens concurrently in independent entities called "actors", that interact primarily by exchanging asynchronous messages, or creating more actors. In a pure actor model, message exchange is the only means of interaction, and there is otherwise no mutable state shared between actors. In practical implementations, actor libraries preserve the means of interaction from the underlying programming language, and even languages built around the actor model may include extensions for reasons of performance, interoperation with the underlying system or libraries in other languages, etc.

Now, the actor programming model can be a pleasure to work with, since it nicely models a lot of situations that programmers face; this is increasingly true as computations have started to grow larger than fits on one processor, and span many processor, sometimes scaling to a great number of machines spanning a world-wide distributed system. Yet, typical approaches to implementing the actor model often lead to awkward performance or fragile software.

For instance, spawning one Operating System process per actor can be very expensive: there is a lot of memory overhead involved in every OS process, and a relatively small limit to the number of processes simultaneously usable in a system, which limits the applicability of the actor model; communication typically requires many context switches from process to kernel to process, and every context switches tend to be very slow and throw away expensive execution context; there is often a large impedance mismatch between the semantics of language actors and that of OS processes, each tending to evolve very different kinds of options with time; papering over these differences can be costly in terms of the complexity of interfaces to support. Finally, processes are entities that may survive independently from each other and from whatever interface developers use to control them; they can therefore go astray in a wide range of configurations, and there can be a lot of complexity in managing them, keeping them in synch with each other, handling version discrepancies, assessing whether they are functioning in order, making sure that resources don't leak.

Another strategy, more often adopted, is to spawn one thread per actor within the same process, using whatever notion of thread the underlying programming language offers. Much of the cost and complexity related to dealing with many operating system processes go away. However this strategy can be very fragile: threads, whether managed by the operating system or by special user libraries, share with each other the entire mutable state of their common process; this mutable state typically requires a lot of caution to be taken by each and every thread in respecting all the relevant invariants and conventions. One mistake, error or unexpected condition happening in one actor, and the entire computation, including completely unrelated threads, can go wrong in catastrophic ways, that can at times lead to massive financial losses or even death.

Whichever way actors are implemented, but even more so when using threads, there are many reasons why actors may fail: its execution may hit a software bug; it may hit a hardware bug; it may be hit by cosmic rays or outer forces; it may fall victim to some ”wrench” thrown by software like Chaos Monkey [citation needed] that deliberately introduces random failures into the system to ensure that robustness issues are found and addressed earlier rather than later; it may be targetted by some denial-of-service attack; it may exceed some resource threshhold; it may experience a conflict between conventions followed by subtly incompatible libraries; it may otherwise enter a state where it fails to correctly respond to queries.

Hybrid strategies are possible, but introduce a lot of complication, without really solving the issues of either simple strategy above, only allowing the programmer to choose which set of issues he wants to face while writing each part of a given program. This is an improvement, but a limited improvement at a high price that few care to afford.

And yet, among all the systems that implement the actor model, one stands out, that remarkably manages to solve these fragility issues where most others fail: Erlang [citation needed].

7.4.2 Robust Actors: Erlang

Erlang is a programming language that provides a mostly pure actor model, except that it calls its actors “processes”. Each actor is specified in terms of a pure functional applicative language, where the only side-effects are message-passing and spawning new actors.

However, unlike other implementations of the actor model, Erlang (and its derivatives like Elixir or LFE, Efene, Joxa[citation needed]) allows programmers to build extremely robust actor systems. Its approach to robustness is to start from the assumption that exceptions, errors, failures and mistakes *will* happen, and that the system should as matter of course be able to easily recover from them — which it does by having supervisors monitor the status of some other actors, detect when they fail, and then thoroughly kill them and restart them. To elucidate the depth of the paradigm shift in this approach to software, see notably the 2017 paper on *miscomputations* by Tomas Petricek [21].

One key mechanism to achieving this very robust style of actor programming, that distinguishes Erlang from about every other implementation of the actor model, is Erlang's ability to safely kill an actor at any point in time, which other implementations do not possess.

With this ability to kill, Erlang can indeed express the concept of a supervisor in the language. A supervisor may simply wait for the supervised actor to crash, or actively probe the supervised actor by checking its answers to simple regular "pings" or to more elaborate semi-random semi-periodic queries. Importantly, though, in Erlang the supervisor can *do something* when it detects failure, namely kill the failed actor, whereas in other languages it can only cry as the failed actor corrupts the entire system. The dead process will be unregistered from whatever service brokers it was subscribed to, and the incoming request traffic will be picked up by its healthy registered peers, until its replacement is fully operational.

Because interesting services are made of many actors that play in concert and have mutually-dependent state, when an actor dies (whether of natural or super-natural causes), all the actors linked to it (typically but not necessarily, the parent that spawned it and the children it spawned) are in turn sent a signal to terminate gracefully. They can explicitly catch and handle this signal if they really care to survive or to execute some cleanup code before they die; but by default, the linked actor just dies immediately, freeing all its resources; when it dies, so will a graceful termination signal be sent to its own linked actors, recursively, in a tree of related actors (called a "process tree" in Erlang). When actors fail to gracefully terminate, they can be killed summarily, at which points their resources are freed without the opportunity to execute some cleanup code (and to fail while trying to execute it). This ability to safely terminate or kill entire trees of linked actors is essential to building an extremely robust architecture where large services made of many coordinated actors automatically restart in a coherent way when failures happen — but also when regular system upgrades happen.

7.4.3 On Killing

Why is killing actors easy in Erlang and hard in other actor systems? Because Erlang strictly follows a pure actor model, and no shared state is jeopardized when an actor is killed.

In Erlang, the actor termination and kill signals work *asynchronously*: unless an actor explicitly catches and handles a termination signal, it will die immediately; and it will die immediately if it receives a kill signal, that it cannot handle. This means that a regular process may die in the middle of whatever operation it is executing. Because the Erlang programming model is pure and processes interact exclusively via message passing (or nearly so), there is, by construction, precious little shared state that may be left in an invalid state when an asynchronous signal happens: only some system state to track actors and atomically transmit messages to mailboxes, and a few narrowly used extensions for shared buffers or calls to C libraries. The system implementation can take care of this little shared state, and ensure updates to this state are atomic with respect to asynchronous signal delivery; that state is completely hidden *below* the abstraction provided by the language; there is no way a program written in Erlang can possibly break these abstractions or leak resources (though an extension written as a C library of course can).

Now, in most languages, that do not strictly adhere to a purely functional paradigm (like Haskell does), there can be an arbitrary amount of shared mutable state that would be left in disarray if a thread were shutdown or killed in the middle of its execution. Moreover, this shared mutable state is not under the control of the system implementation, but arbitrarily created by user software. Stateful data structures are usually a common thing; system calls or and foreign libraries often involves a lot of state; the language implementation's runtime environment itself has plenty of shared mutable state, that often gets more complex as people extend this implementation. Unless the language, its implementation, its libraries, its foreign function interface, its extensions, all take great care in supporting asynchronous abort signals, odds are these asynchronous interrupts will lead to catastrophic failure of the overall computation.

Indeed, if an asynchronous signal is received, there is a non-negligible probability that some of this large shared mutable data will be in some intermediate state, such that killing the current thread would then leave the overall computation unstable, and unable to operate correctly. A lock may be held that will never be released. The state protected by that lock may violate necessary invariants to the operation of its high-level interface. Some resource borrowed from another actor may never be released or may otherwise never complete its life cycle, and even in the best scenario hoard a growing amount of leaking resources. The program may be experience a deadlocked or live lock. Some distributed protocol that was previously initiated may never complete. Another thread waiting on a spinlock may spin the processor forever in a tight loop consuming lots of power. If the invariant broken is low-level enough, the program may crash in ugly low-level ways, or, worst of all, it may delete important information, corrupt data, return wrong answers, issue undesired side-effects, and do the wrong thing to your system — which can conceivably cause damage up to great financial loss or death. On the other hand, failing to terminate or kill the actor when it is obviously failing can also lead to other variants of the same catastrophic failure modes.

If the probability of corrupting shared state is small enough in a given application, it may be acceptable to summarily kill a failed actor. But even a tiny probability of failure for a single killing operation can mean a large probability of failure over time if the idioms of Erlang are used, whereby tens of thousands of processes are constantly being supervised on every machine in a large distributed system, and asynchronous termination and killing are regular activities, rather than rare desperate fallback solutions. Therefore, the programming model that enables Erlang programs to be robust is not available unless the probability of failure from killing an actor is entirely negligible.

7.4.4 Workarounds to the Unavailability of Asynchronous Signals

In languages other than Erlang (and its derivatives), there are some limited workarounds that may enable use of actors while providing an ersatz of Erlang-style robustness, by cooperatively emulating the evaluation model that these languages cannot directly express.

- First, the programming language will *not* allow for asynchronous killing signals (except maybe as a desperate debugging tool). Termination signals will have to be explicitly and synchronously handled by each and every actor that partakes in the protocol.

- Developers must socially abide by and enforce programming conventions, whereby all actors should regularly call the message loop; programmers must manually take care that there should never be a deadlock, live lock, non-terminating computation or runaway code execution between two consecutive calls home to the message loop.

- If some algorithm may involve indefinitely long computations, its implementation must maintain a discipline of "cooperative multitasking", like in the bad old days of the 1980s, whereby these long-lived computations will be specially modified to periodically ”yield” execution and poll for termination signals, giving the message loop the opportunity to process such a signal while the computation is in a stable state.

- The programming language at hand must be considered as a replacement not for Erlang itself, but only for the lower-level language in which the Erlang VM, BEAM, is implemented (i.e. C). This means that developers will be exposed to all the ugly synchronization details, that they will have to manually handle, following some conventional design pattern, without being able to fully abstract over these patterns.

- Any further abstractions over the programming language can only be enforced by social convention. Most languages have no means to express the conventions that must be respected and prevent users from reaching into implementations and breaking the invariants that these abstractions depend on. Only one language, PLT Racket, allows developers to build and enforce a full abstraction for module, thanks to its Racket `#lang` feature, that imposes global (rather than merely local) restrictions on what software inside the module can express.

- If you really want a group of actors that live and die together, they can be put them in a same Operating System (OS) level process (and either use OS process groups to implement trees of related actors, or implement yourself that notion using some kind of supervisor process). Then you can kill and restart the OS process and its entire set of actors. Unlike Erlang "processes" or OS threads, OS processes are heavy weight; but at least, this programming style works, and sometimes that's exactly what's needed.

- In general, as much as possible, developers should use a pure functional style and restrict side-effects to local state that is private (not shared), thus reducing issues related to shared state for processes that use this style. However, because the programming language's implementation's runtime and the available libraries were never designed for asynchronous interrupts, their own use of shared resources can still cause catastrophic failures in case of asynchronous termination signals.

These strategies of course work, but they lead to code that is awkward, inefficient, not modular, tiresome and error-prone to write, impractical except at a small scale, and still fragile. It is not satisfactory to only provide fragile constructs that will explode when developers fail to respect non-trivial coding conventions, and to maintain these conventions as the software evolves.

In the end, these strategies are "design patterns" that transform programmers into manual code generators who instrument their computations to follow a cooperative shutdown protocol. But Rich Hickey said: [**?**] What this issue really calls for some robust abstraction mechanism inside the programming language, that will automatically enforce any invariant through coherent automated code instrumentation rather than manual discipline.

7.4.5 Observability as Necessary and Sufficient Solution

Automatically generating code so that it always strictly follows some additional conventions is the very definition of *code instrumentation*, also known as the "opposite of natural transformations of implementations".

Just like support for automatic Garbage Collection can be added by instrumenting control around every access to the heap using e.g. read or write barriers, support for asynchronous interruption can be added by instrumenting control around every access to shared resources using e.g. roll-back or roll-forward of atomic transactions. Indeed, Garbage Collection *is* a particular case of an operation on a shared resource. However, in the case of aborting threads, unlike in the case of collecting garbage, it is not enough to achieve observability at one fixed low level of abstraction: indeed, the thread's execution partakes in protocols at every level of abstraction; preserving only the invariants up to one low level of abstraction while breaking the more restrictive invariants at higher levels of abstraction would therefore leave the system unable to correctly make further progress for all operations at those higher levels.

The solution to enabling the killing of threads in presence of user-defined shared mutable state is therefore user-defined observability: whoever defines the shared mutable state must also define safe-points, transactions, or some other means of achieving observability around accesses to that state, as a part of the requirements for this shared mutable state to be usable. Note that "shared mutable state" here does not have to specifically be the bits at some address on a shared memory multiprocessor or monoprocessor. It can be any part of any distributed protocol, at any level of abstraction. If a group of machines vote to coherently maintain an evolving mapping from some abstract symbols to a weighted set of likely used definitions, then no fixed memory address may be involved on any machine, yet the result of the distributed protocol is very much shared mutable state, and there might be a well-defined meaning to killing one of the activities that partakes in this protocol on one of the machines.

Observability is thus a condition both necessary and sufficient to achieving robustness in killing threads — assuming that killing threads is possible at all, which is itself a necessary condition of a robust distributed system, where you may never fully trust a process and its threads not to go wrong, nor the granularity at which it will go wrong.

Now, what if there is a bug in the user code for observability, and the computation fails to achieve observability? Then the situation is about as bad as if observability was not attempted (as is the case in traditional systems without first-class implementations); the situation is actually slightly worse, for all the bad side-effects of the buggy attempt to achieve observability. In the end, the same bugs that make the eventual failure of a thread inevitable also make the system as a whole unstable; and they make it so in an unrecoverable way, since killing of bad threads couldn't be safely achieved.

Of course, observability can always be still achieved *up to some robust abstraction level* R, even when it cannot be achieved above R. Thus, any program P that depends on higher-level invariants not encoded in R may be broken; but the system continues to work just fine at level R and lower levels of abstraction, with their laxer invariants. Other high-level programs P_i above R that do not share state with P (at least, none that P is currently breaking) may also be able to keep running. At that point, the computation P can be *reset* to some state defined at level R, but that any state it possesses at a higher level of abstraction than R may get stuck and be lost; partial recovery of the lost state might still be achieved through special steps manually taken by a competent system administrator; but it is a costly expense that few can afford.

In most cases, the hardware at the very least will preserve its physical properties independently of software failures — though, there have been software failures known to cause irreparable damage, and malicious attacks exploiting those failures: overheating a computer and even causing fires, physically crashing hard disk heads, stuffing a printer, making nuclear centrifuge devices spin to their death, etc. Beyond mere hardware safety, the operating system usually provides a stable, observable abstraction, and on top of that, language implementations often provide robust observable virtual machines. Yet, since these programming languages provide no cheap and easy way for users to extend this observability to the computations implemented by their programs, therefore, programming languages offer no stable basis for state above what the operating system provides, and programmers must constantly write their own layers of software to save and restore state that they care about (see above section 7.3 on Orthogonal Persistence).

If a given programming language or programming environment is to help with more than a single layer of abstraction, it must therefore enable the expression of all the distinct relevant notions of a safe point, one for each level of implementation in the semantic tower — all within the same programming language and runtime infrastructure. In other words, it must support our observability protocol.

7.4.6 Full Observability Protocol Required

The simplest way to achieve observability in a multithreaded setting, as used by all implementations that provide garbage collection, is to support blocks of code that are atomic at the higher-level, despite not being atomic at the lower-level. Any point outside these blocks (or at either end of one) is a safe point, i.e. is observable; points inside the blocks are unsafe, and the block will be rolled forward (or, sometimes in some more elaborate implementations, rolled back) to reach a safe point. In some implementations, all points are safe by default unless explicitly part of an atomic block; the "blocks" are often statically defined in terms of ranges of the program counter, but may be dynamically defined in terms of an "interrupt disabled" flag (for instance, the lower bit of a register usually aligned to memory addresses). In other implementations, all points are unsafe by default unless explicitly marked as safe, which again can be static (known program counter addresses) or dynamic (calls to a special trampoline, possibly conditional on checking an observation flag).

Now asynchronous signals, when received, require observation at the level of abstraction of the signal handler, and thus synchronization to a point that is safe for said level of abstraction. The default handler will kill the thread (after reporting or logging an error and a stack trace), and requires observation at the highest level of abstraction relevant to that thread. There again, asynchronous signal delivery must adapt to the semantics of the application, and there can be no "one size fits all" notion of observability: the notion that works at one level of abstraction won't work at higher levels, and low-level libraries that partake in the atomicity protocol for the implementation's garbage collector do not thereby suffice to provide any help with respect to the atomicity that matters, at the application level.

What more, reaching a safe point is usually enough for the purpose of killing threads defined at that level without having to extract the abstract state; but reaching a safe point is not usually enough for other purposes, including that of killing threads defined at a higher-level than that of the safe point: Indeed, reaching a safe point for that higher-level might itself require extracting data about the current state at said higher-level, for instance to identify shared resources to release, run cleanup forms, or to otherwise commit or rollback the current transaction. Since there is no upper limit to what abstractions users might want to build on top existing computations, there is no point where you can for sure stop respecting the interpretation protocol in addition to the safe point protocol — at least, not until you reach the actual end-user interaction.

A complete solution therefore necessarily uses the entire first-class implementation protocol, including observability to synchronize to safe-point, interpretation to extract higher-level state, and completeness to carry the high-level synchronization back down to the low-level. Only this full protocol can "lift" the notion of safe point, so that a higher-level safe point may be recovered from a lower-level safe point, and so that threads may be safely killed.

7.4.7 Performance Tricks for the Observability Protocol

A naive understanding of "recovering the state at some abstract level" can be too expensive to be feasible: you don't want to serialize the entire state of the virtual machine (potentially gigabytes of memory or more) every time you process an asynchronous interrupt handler. The cost of eagerly evaluating a high-level representation for the state of a large computation, at every observed safe-point, would be prohibitive; the cost would be even higher in concurrent systems where abstract observation might require synchronization between many computing devices. The abstract state recovery must to be lazy, so the system only extracts the bits of abstract state actually required by the automated handler or inspected by the human operator.

A naive implementation of safe points would create a closure to express this recovery, at every safe point. A slightly less naive implementation would only create that closure *if* an interrupt was actually caught at that safe point. Therefore, a general protocol for a safe point is therefore to have some kind of special form `(safe-point` ***`abstract-level concrete-level state`***`)` to declare safe points; the level arguments somehow identify the level of abstraction of the safe point (these arguments should if possible be known at compile-time, and might even be implicit), and `state` is a form only evaluated when an interrupt is caught at said level, that permits recovery of the state at specified abstraction level, if possible lazily.

For lower-level implementations, and especially where most points are safe by default, the state recovery can conceivably be done based on off-code debugging annotations in the style of the DWARF[citation needed] format. Notably, for performance reasons, tight loops and highly used code might eschew any polling for interrupts; when an interrupt is received, the off-code annotations can be used to synchronize to a safe point: the assembly code might be "interpreted" until a point is reached and control escapes; on some architectures, the escape code could be generated dynamically; on other architectures, tight loops would have a shadow copy that escapes rather than loops, so that the interrupt handler could, based on the meta-data, add an offset to the program counter before calling the code, causing the current loop iteration to be completed before an escape to the higher level.

So as to avoid polling for interrupts too often and maintaining many copies of state-extracting code, the compiler will hopefully knows how merge safe-points for multiple levels of abstractions. Thus, tests for asynchronous interrupts at higher level safe-point and creation of corresponding higher-level state objects will only be evaluated if an asynchronous interrupt was already caught at the corresponding lower-level safe-point, yet wasn't handled already by a lower-level handler.

An even better compiler would elide redundant consecutive safe points, and only poll for interrupts at the beginning of functions and loop iterations — just like the implementation presumably already does for its own lower-level checkpoints. To allow single-stepping with a finer granularity than the concrete computation allows after elision of these redundant safe points, the implementation of the abstract computation may have to maintain a higher-level representation of the code, e.g. in term of an expression graph or of portable bytecode, that can be interpreted more slowly but with finer granularity than is available with the lower-level code.

7.4.8 Language and Library Support for Asynchronous Interrupts

Now, it is not enough to have compiler support. The runtime library must also be written in a way that supports asynchronous interrupts, and the programming language must provide suitable abstractions. Indeed, when allocating *any* kind of resource that an asynchronous interrupt may necessitate to release, the atomic operation with respect to interrupts should be not merely allocating the resource, but allocating it *and* atomically binding some variable to it, *and* registering cleanup forms using e.g. a `finally` clause, to properly release the resource without a leak should an asynchronous abort be received. The `finally` clause will in turn have to either be atomic with respect to the high-level interrupts, or to catch and synchronously handle these interrupts, or to otherwise reenable interrupts after some initial processing.

Potentially long-running library functions, and especially higher-order functions, may also have their own issues with respect to declaring safe-points for higher levels of abstraction within the dynamic extent of their function call. When an abstraction level reexports such functionality from lower levels, it may have to subtly wrap this functionality in variants that suitably handle safe-points. And the compiler may have to be able to suitably optimize away most wrappers.

There is also the case when a thread receives a further asynchronous interrupts in the middle of processing an existing one; or when it gets stuck while executing cleanup forms in general. Asynchronous interrupts are specified with a target level of abstraction. By default, an abort signal (as a Unix `kill -TERM`) works at the highest level of abstraction that the programmer cares about, and should run all the cleanup forms. If the operator gets impatient, he may send signals with lower levels of target abstraction (down to a Unix `kill -KILL`), at which point levels of abstractions higher than the target level are invalidated, their cleanup forms are eschewed, and all linked processes at this level of abstraction are killed (and hopefully restarted by their supervisor). It is therefore possible to "lose" a layer of abstraction — if there was a bug in the implementation of this layer of abstraction, at which point, well, that is exactly what "having a bug" means.

All in all, supporting asynchronous interrupts requires special care while writing software libraries as well as deep support in the compiler itself. This can be a lot of non-trivial work, but the result might be worth it, as this enables not only Erlang-style resilience, but all the other applications suggested in this thesis.

7.5 Further Transformations

7.5.1 Optimization

Many optimization passes can be viewed as natural transformations of implementations: take an implementation Φ^{-1} of an abstract computation A with a concrete computation C, and identify a "better" implementation Ψ^{-1}, that minimizes some metric, applies some normalization strategy, some local rewrites, some change in representation, etc. The order of evaluations is locally modified, but the sequence of abstract state at observable safe points is preserved (or graph of possible paths through those states).

This realization can help specify and prove the correctness of compiler passes. But by treating these passes as first-class entities that users can manipulate, rather than hiding them behind an opaque compiler, it becomes possible to let programmers or meta-programmers simplify or enhance existing programs, to develop efficient domain specific languages or programming language extensions, or to guide a compiler towards the more efficient evaluation of specific computations in a way that remains correct by construction.

7.5.2 Higher-Order Transforms

Now that we have identified a universal concept of natural transformation of implementation and their opposite code instrumentations, it becomes possible to envision libraries of typed functions that may apply to arbitrary such code instrumentations, or that may uniformly generate such code instrumentations. The usual categorical toolkit may apply to help develop them, work with them, compose them, build higher transformations between transformations, etc.

7.5.3 An Open Topic

This chapter has only skimmed over how the notions of first-class computation and first-class implementation can help bring about a coherent new paradigm, that provides a unifying framework to think past and future techniques.

Part IV

Architectural Implications of First-Class Implementations

Chapter 8

A Reflective Runtime Architecture

8.1 Runtime Architecture

8.1.1 From Blocks to Buildings

What kind of system architecture is necessary for all (or most, or a lot of) software to have first-class implementations maintained at all times? What kind of interactions are made possible and affordable by such an architecture that would otherwise be impossible or prohibitively expensive?

Just like new building materials (wood, stone, reinforced concrete, etc.) or new building techiques (scaffoldings, arches, cantilevers, etc.) can usher new eras of architectural styles in buildings, new primitives (closures, objects, processes, distributed map/reduce services, neural networks, etc.) can lead to new ways of organizing software, by expanding the realm of the possible, and modifying the balance of the affordable.

We will offer suggestions for how the architecture of a computing system would be different from what is currently mainstream if the system were to maintain at runtime fully abstract first-class implementations for all computations, at least by default — and then to take advantage of this capability. In the following chapters, we will call the suggested architecture a “reflective architecture”.

8.1.2 Interactive Systems

A reflective architecture, one that maintains a fully abstract first-class implementation for all computations, makes sense if and only if the software can be dynamically inspected or modified after it is initially started. But that’s indeed the case in a wide variety of common situations.

Development Platforms

In a Development Platform, such as GNU Emacs, or an Integrated Development Environment (IDE), etc., sentients interact with a computing system to develop software. The software is being modified; the developer tries to make sense of unexpected behavior, or of unsatisfying performance, and to evaluate how some or some other changes affect the parts he’s trying to fix, while not disturbing the parts he’s trying to preserve. The developer often has to navigate the execution of the software at multiple layers of abstraction, until he can pin-point the precise level at which to make a change.

User Interface Shell

In a User Interface Shell, whether it is graphical like the macOS Finder, or it is a "console" text interface like ZSH, users also control applications and various smaller programs, and sometimes even simple scripts that they write. Users may configure these programs, sometimes compose them, control their execution environment, interrupt their execution, monitor the resources they use, etc.

Although the granularity at which users typically employ these shells is usually coarser than the granularity at which developers use development platforms, in the end, a user interface shell is a form of development platform, and a development platform is a user interface shell. Indeed, *the difference between a user and a programmer is that the programmer knows that there is no difference between using and programming.*

Operating System

Rather than defining high-level abstractions expressed in a high-level language, an operating system typically defines its primitives in terms of bits and bytes, of "system calls" and register and memory modifications at the level of CPU instructions, or almost equivalently at the level of calls to functions in a low-level language like C, and in terms of an "executable file format" combining the above.

Yet in the end, low-level as its interface might be, an operating system is the same as a development platform: it also allows its users to develop new software, configure it, compose it, start computations, control them, etc. — if not, it would just be called "firmware", not "operating system".

Distributed and Virtualized Application Management

Larger distributed computations also require not just middleware to effect communications, but executive interfaces to start computations, monitor their resource usage, control them, stop them, restart them, etc. These layers are also interactive systems, though they tend to have leaky abstractions such that the complexity of all the underlying layers is compounded rather than abstracted away.

8.1.3 Reflective Interactive Systems

Interactive systems can be augmented with runtime reflection (as described in previous chapters), yielding some primitives missing in existing interactive systems. Whereas existing systems can destroy an existing computation (`SIGTERM` and `SIGKILL` in Unix), sometimes stop and restart it (`SIGSTOP` and `SIGCONT` in Unix), a system with runtime reflection can also take a running program, and: (a) stop it and recover its state at a higher level of abstraction, or (b) migrate it to a different implementation while it's running.

The minimal API for that is that the system maintains for every running computation not just the state of the computation in its current concrete implementation, but also a structure of more abstract interpretations of this computation, or more concrete implementations. And each level transition in this structure is an interpretation or implementation (depending on the direction it's taken) that annotated with its own properties and the computational content thereof (e.g. totality, completeness, liveness, observability, etc.).

These abilities and their applications have been described in previous chapters; the following chapters will explain how they may affect software architecture.

8.2 Every Computation has a Semantic Tower

8.2.1 First-Class Semantic Tower

A (traditional) irreflective interactive system will only keep track of computations at a single canonical low level of abstraction, that of its "real" or "virtual" machine. Inasmuch as there is a "semantic tower" implicit in every computation, all of it is destroyed at compile-time, and only the ground floor remains.

A reflective interactive system considers that no level of abstraction is either more real or more virtual than any other: they can all be considered as being actually run, and they can all be virtualized without notice. What matters is that a reflective interactive system will keep track of each computation simultaneously at all potentially interesting abstraction levels in its semantic tower. We saw in previous chapters that this tower need not be a linear order and need not be finite (much less so with cardinality countable with one's fingers); rather, this tower can itself be an arbitrary category, possibly infinite.

Now, the user or the system may at any time change the current abstraction level of a running or stopped computation from one level of the semantic tower to another. When that happens, the system picks an efficient transformation from the old level to the new level, that avoids going through unnecessary or slow intermediate representations. The computation is now considered at the new level, and a new set of potential transformations is available, to reach the other levels in the semantic tower, nodes of its category.

Conceptually, the system could represent the semantic tower as a labelled set of transformations from the current level to the other ones. When a transition to a new level happens, the new of available transformations could thus be computed by composing each element of the previous set of transformations with the inverseof the transformation just applied. A naive implementation of this strategy would of course be terribly inefficient in time, and grow linearly in space as more transitions occur; if the composition is computed eagerly, there is also a linear growth in the size of the semantic tower.

A better model is to have a first-class representation of a semantic level (i.e. of a node in the category constituted by the semantic tower). Then, a transition consists more simply of applying the proper transformation, which can be computed by composing elementary transition functions, then tracking the new semantic level. Its complexity remains constant in time and space. Moreover, if done lazily and symbolically, this composition of elementary functions can also be "optimized" by "deforesting" intermediate results.

8.2.2 The World on top, and Turtles all the way down

The semantics tower associated to each computation may have many distinguished levels, beside the one "currently running".

At the top of the semantic tower, the sentient user specifies the computation they desire. This computation embodies the world as as they will experience it. The system is supposed to implement this specified semantics; an obedient servant, it has to conform to whatever the sentient user has specified so far. Any discrepancy from that specification is a damning bug for the system.

At the bottom of the semantic tower, is whatever computation is currently running. This is the only level that traditional irreflective systems keep track of. But in a reflective system, this is not a permanent bottom, because the system may always dynamically virtualize the current machine, and add an implement layer beneath, whether for the purpose of instrumentation or performance or migration to a different machine. When this happens, the system is then lifting the entire semantic tower up one level (or several levels) and adding new stories underneath, while it's running, without the user-visible world being observably affected. Thus, *there is a fixed world on top of the semantic tower, but below it's turtles all the way down with no fixed bottom.*

Between this top and this bottom, and optionally also above this top, stands the "current tower" of abstraction levels into which the current bottom computation can be interpreted without the need to migrate.

8.2.3 Static and Dynamic Implementation Control

Often, the user, or the implementer for him, is interested not just in the user-visible world, but in a specific implementation of this user-visible world. He then specifies not the conjunction of this user-visible world and of an implementation strategy to apply to it.

Together, this abstract computation and implementation strategy constitute a distinct, more specific, more concrete computation; one where already two related levels are distinguished: on the one hand the "concrete" computation, obtained by applying the implementation strategy to the high-level computation, and that constitutes the specification as far as the lower levels of the system are concerned; and on the other hand, the "abstract" computation, that constitutes the world as visible to the end-user, and that may be at any time recovered to effect a change in implementation strategy, at the bequest of the user or of the implementer he delegates to.

This implementation control by specifying an implementation strategy can be static or dynamic: It can be static, by specifying the strategy at compile-time, at which point the strategy is wired into the object code that runs on the low-level machine. Or it can be dynamic, as the user changes their opinion, or something changes in the environment, and the user, or one of his proxies, refines or otherwise modifies the implementation strategy. Alternatively, they may decide that they trust the system or not be interested in implementation anymore, because the system's default strategy now proved it works well, and the user doesn't care in the details anymore. Then, the user may drop the implementation strategy, or vastly simplify it, and thus move the specification back from a lower-level specification to a higher-level specification.

If we squint our eyes, we can even see all the interactions of the user as changes in the computation he is interested in: his access rights define the top of the semantic tower of his computation, where one amongst all possible algorithms relate the resources and I/O devices that he has access to; but the initial implementation strategy is the trivial unit category with one point and its identity arrow, without any side-effect. Every change in the effective computation requested from the system can be seen as a change in the implementation strategy for that top computation.

8.2.4 Intermediate Management

In between the user's bidding and its ultimate low-level executors, there may be any number of intermediate layers of management, each specifying an implementation strategy for the computation it is specified to implement.

Each layer is thus limited in its power to implementing what the higher management layers require; and each layer may be invalidated by any of the upper layers, and required to do something different, or replaced with different workers altogether. There is thus a hierarchy of management, where each actor has limited capabilities in a hierarchy of access rights. The top of the hierarchy is specified by the User; the bottom is controlled by the System; in between there can be a continuum of agents closer to the User or further from Him.

For instance, a simple management layer for cloud computations may monitor the resource usage of a long-runnning computation and the prices of resources as available from various cloud providers; based on current state of the market, it may stop, migrate and restart the computation. The user may provide some parameters that control the spending strategy; or may even override specific parts of the code that the system would otherwise use as a default strategy.

In another common instance, computation workers may include some JIT (Just-In-Time compiler) controlled by a monitor, that decides which parts of the code to compile or to aggressively optimize, and which parts of the code to just interpret naively. There again, the user, or some intermediate optimization layer, could conceivably provide explicit implementation hints to the workers' JIT, e.g. based on data accumulated from profiling typical runs. In the end, each management layer must strictly implement what the upper layers specify (up to what the user specified), but there can be a lot of leeway and fluidity in how the responsibilities are laid out below what the user specifies.

8.2.5 Dividing Control

When multiple parties are involved in a computation, they may each have specific access rights regarding which parts they control, and which they don't. This truism is valid when parties control disjoint components at a given level of abstraction (vertical stripes of our diagrams); but it is also valid when parties control disjoint "slices" of the semantic tower (horizontal stripes of our diagrams); or even the intersection of components and slices (rectangles of our diagrams).

In such a setting, parties handling directly superposed layers would be bound by a contract such that (1) the party responsible for the high-level slice of the computation has no access to low-level aspects of the computation through which one could potentially subvert the system, whereas (2) the party responsible for the low-level computation must strictly abide by the high-level specification of the computation and is not authorized to leak secrets to third parties. In other words, the higher-level and lower level slices must be related by a *full abstraction*.

For instance, when delegating execution of some programs to the cloud, the contract between the user and his cloud provider will typically limit what control the user has on the implementation by locking the programs in a virtualization layer that isolates him from the hardware, and maybe subjects him to eviction based on counter-bids in a fluctuating market. Meanwhile the provider is required to implement virtual machines that completely and correctly implement the promise computation model, and performs within guaranteed levels of service (speed and latency distribution, etc.). The user may specify some implementation strategies, but is not allowed to go below whatever interfaces the implementer provides; the user may not use low-level tricks to circumvent any barriers the implementer sets between users or uses to ascertain the integrity of his systems. The implementer in turn may not be allowed to leak any of the secrets of the user (his implementation must be "fully abstract" within observable parameters). If the implementer in turn leases resources from a yet lower-level provider, then a similar contract will bind them where he now has the role of user.

The semantic tower in general is therefore not just a pure stateless mechanism that "transparently" implements semantics without adding any information; to the contrary, it can be a rich dynamic stateful ecosystem, where the interactions between actors can be ruled by elaborate contracts.

8.2.6 Ubiquitous Reflection

In a reflective interactive system, each and every implementation layer of the semantic tower of every computation must expose a reflective interface, constituted by the computational content of suitable variants of completeness, liveness, observability, etc. Thus, every programming language, every virtual machine, every translation layer, every macro-expansion pass, must enable developers who use it to prove these properties, or at least to express the computational content of these proofs.

For sequential computations, this means that languages must provide some primitives to declare safe points or check points, at each of which some function may be defined that can recover the higher-level state. For concurrent computations, this means that languages must provide atomic operations or transactions, at the ends of which some function may be defined that can recover the higher-level state. A naive implementation of these safe points and atomic operations can be quite expensive, and so can an overly eager implementation of abstract state recovery functions. Therefore good languages must provide good declarative ways to express them, that allow for efficient implementation.

Now reflective primitives are not just required of the languages and tools that developers use; it is also required of the languages and applications that they write. And indeed, every application can be seen as a domain-specific language in which user interactions specify appropriate system responses; and every language, whether general-purpose of domain-specific, may be used a computation back-end for higher-level computations.

8.2.7 Bootstrapping Reflection

When a language fails to provide suitable reflective interfaces, maintaining reflective functionality becomes more expensive, though not impossible.

First, as long as some the computation runs on top of a suitably virtualized CPU at the bottom, it is always possible to simulate, record, replay, checkpoint, persist, analyze, instrument or otherwise manipulate the computation as a first-class entity. The problem is that the cost of retrieving the high-level semantics from the low-level execution state, while finite, is tantamount to that of reimplementing the entire computation stack, including every application and programming language involved. This cost is compounded when the source code is not available, or of very bad quality, or itself a complex entanglement of many layers of leaky abstractions. In the end, it may simply not be worth it — though often, simple instrumentations can rely on said bad quality code being naive in its implementation strategy, so that structure can be retrieved by injecting high-level changes and automatically comparing how that affects runs of the code.

Second, when some abstraction level lacking support for reflection is targetted for the concrete computation of a semantic tower (or of a slice of a semantic tower), it is always possible to implement the reflective primitives the hard way on top of the irreflective substrate: generated code can include at regular safe-points checks for an escape flag, which if set causes causes the program to jump out to the monitor, providing it a function or other descriptor that will reify the abstract state; if the state must be recursively reconstructed at every frame of the call tree, special return values can be recognized, or special exceptions can be caught, that will recursively trigger the reification and escape; if some of the computation state is implicit in the programming language, generated code may maintain an explicit shadow for the implicit state, allowing it to be reconstructed; at worst, the low-level computation may be used to interpret a virtual machine with well-defined semantics and explicit state, that may then be made the actual target for higher-level computations.

These are all well-known techniques, routinely used to achieve special purpose reflective results: garbage collection, migration, redundancy, white-box testing, and any of those techniques mentioned in previous chapters. We propose that the very same implementation techniques may instead be used to achieve general purpose reflection.

8.3 Every Tower has its Controller

8.3.1 Controller Meta-programs

It is usually desirable for migration to happen automatically as directed by an external program, rather than to be manually triggered by the user, or to follow a hardwired heuristic. Indeed, most users, and most programs, could not care less about most of the details that are dealt with using migration, whether they are about cache lines, Just-In-Time (JIT) representation of code or data, availability of cloud resources, etc. Moreover, a heuristic hardwired in the computation would make that computation both more complex and more rigid than necessary, making it simultaneous hard to reason about the program and hard to adapt it later to the precise needs of the users.

For all these reasons, a reflective architecture includes the notion that every computation has a *Controller*: a meta-program that controls this semantic tower. The controller will follow relevant signals and dynamically control when to incrementally or totally migrate the computation from its current implementation to another one. As it does, the topmost computation is fixed — or rather, its state keeps making progress while its semantics remain constant and/or are refined monotonically.

The Unix equivalent of a controller would be a process that controls another using the `ptrace` system call,[citation needed] such as a debugger, or a virtual machine monitor (like `qemu` [citation needed]), etc.

8.3.2 A new dimension of metaprogramming

Controllers are arbitrary computations of their own. A controller can be developed in an arbitrary programming language, in which distinguished input/output primitives are available to interact with the controlled program.

Since controllers are themselves computations, they themselves each have a controller, that is itself a computation that has its own controller, and so on. Thus the system maintains for every computation not a tower but another sequence of controller, controller of controller, controller of controller of controller, etc. If you think of controllers as puppeteers in the background who control the puppet computations in the foreground so they perform the show that users wish to interact with, then it's puppeteers all the way back, with no ultimate back wall: the system can always spawn new puppeteers to control existing ones, and even the operating system kernel's debugging console is less ultimate than an oscilloscope probing into the hardware.

This adds another dimension to all our computation diagrams, of back-to-front runtime control of computations, different from either the left-to-right progress of computations or from the up-to-down concretization of computations (or down-to-up abstraction). There again, users specify the "front" of that dimension, whereas the "back" is ultimately handled by the system. Users can always explicitly control a larger "depth" of control in "front", and add new layers in the back to handle additional aspects of control.

8.3.3 Shared controllers

A controller can be a stateful computation, and controllers of distinct computations may or may not share state or otherwise communicate with each other.

Most notably, the multiple computations of a same semantic tower can be handled by the same controller or by distinct controllers. Indeed, the bottommost computation of a semantic tower is itself the topmost computation of another tower (with that tower becoming trivial when the topmost computation hits the "bare metal"); there is thus a tower of controllers, that corresponds to the a tower of distinguished current implementations: each "slice" of the computation has its controller.

An essential reason why several computations may share the same controller is that a finite computer cannot implement an infinite explosion of ever-branching controllers of controllers. For practical bootstrapping purposes as well as for the sake of formal well-groundedness, the system must ultimately have a finite number of computations controlled by a finite number of controllers. On the other hand, an essential reason why controllers must sometimes be distinct from each other is that they are processes controlled by distinct sets of people: one company may provide an application by relying on a development and deployment system provided by another company that in turns rely on the provider of "cloud" services who uses chips from a hardware vendor with their own basic virtualization and management layer, backdoored by their own government's three-letter agency who rootkits everyone's hardware; each layer would have its own controller, distinct from the others, that has its own level of abstraction, its own state, its own responsibilities, its own release cycle, none of which can possibly be the same as those of the other controllers.

8.3.4 Dynamic Invalidation of Controllers

When migrating the implementation it controls, a controller has to preserve the topmost computation; but it may arbitrarily abandon any of the computations below, to replace them with different ones — at least, until it hits the hardware, the cloud virtual machine, or whatever low-level platform the user is paying, for which the offering is limited, and necessarily changes slowly compared to software.

Now, when a controller changes the bottommost computation of its implementation, all the computation slices below become invalid, and so are their own controllers invalidated; new computation slices are created instead, and with them either a new controller or an existing one. Alternatively, the controller may reuse the same bottommost computation while completely changing the implementation of the topmost computation; when reusing the bottom computation, its controller might be reused or recycled, its state preserved or reset; or a new controller my be created, or some different preexisting controller may be attached.

The set of active controllers thus varies dynamically, with the lifetimes of front and down controllers depending on migration decisions by back controllers and up controllers.

8.3.5 Controlling Evaluation Strategy

Among the things that a controller can control is the evaluation strategy. Any non-determinism or underspecification present in the abstract computation can be determined and specified by the controller when it picks the computation's implementation — and then re-determined and re-specified differently when it migrates to a different implementation.

For instance, if the abstract computation is expressed in terms of the lambda-calculus without a reduction strategy being specified, then the controller could pick any reduction strategies, including but not limited to applicative (eager), normal (lazy) or optimal. Likewise, if the abstract computation is a search in some recursively defined space without a search strategy being specified, then the controller could pick a depth first along some order, or breadth first search based on some estimated likelihood, or a mini-max algorithm with alpha-beta pruning, or some heuristic search based on analysis by a neural network, or any combination of the above, and more.

The ability to decouple the computation from its evaluation strategy makes it possible to declare a computation that can be composed with other computations and used in many contexts such that the strategy only needs be specified when the final overall computation is determined, at which point a more adapted strategy can be found than would be possible if each subcomputation had to pick an evaluation strategy as it was read and immediately compiled from source code into machine code. Late binding of strategy thus keeps code more declarative, more modular, and potentially more efficient.

One particular way that a controller can affect evaluation is to never commit to any progress until the user ends the computation, and instead to always allow the user to rewind the state of the computation back in time, from where it can be replayed or fast-forwarded. Adding time travel can thus be done automatically for all computations, including but not limited to deterministic video animations and audio performances, as long as no costly external side-effects are involved in replaying (e.g. sending payments, dropping bombs, reusing one-time-pads, etc.).

When thus exploring the evaluation semantics of an underspecified, non-deterministic or probabilistic computation, the controller may of course change between many evaluation strategies and modify any parameters those strategies use (including any pseudo-number generator seed), so that different outcomes may be observed in each of many explored branches of the evaluation. Yet more advanced controllers (e.g. for probabilistic programming) may synthetize results from many such evaluations. A controller might also use various metrics to measure the progress of a heuristic strategy or detect lack thereof, to automatically tune some parameters or decide to switch to a different strategy.

At the opposite end of the spectrum, evaluation strategies can be devised to guarantee timely termination, ensure better memory locality, minimize use of some resources, avoid leak of information, or otherwise provide additional security when evaluating code that is not fully trusted.

8.3.6 Controlling Effects

In addition to the internal evaluation strategy of a computation, a controller can also control the external side-effects of the computation. Effects controlled include effects as a computation emits sound or records it, as it interactively displays information to the user, as it exchanges network packets, etc. Control on these effects include the ability to filter, transform, analyze, synthetize, create, destroy, replace these effects. The controller can also connect, disconnect and reconnect the computation's inputs and outputs. All this control happens while the computation is running, and without the computation itself being able to know that any of it is happening: to the fore-computation, these are all interactions with the "outside world"; the controlling back-computation can arbitrarily define what this outside world is, as long as it never interferes with the fore-computation's "inside world" is in ways contrary to the fore-computation's specified meaning.

Thus, a computation may "just" specify that some information is displayed, and the controller will arrange for a window to be popped up, a message to be read on a synthetic voice, a braille terminal to be updated, etc., as appropriate. The computation itself has no control over this display: most computations wholly lack the capability to talk to any display, network, filesystem or I/O device whatsoever; they do not possess the relevant capabilities or access rights, and their code does not contain the relevant libraries or key data. What computations do is declare their output to their controller, that will be transmitted to relevant displays (if any) by its controller and the display controller (when applicable).

Controllers are thus very much like operating system kernels, or virtual machine emulators (e.g. QEMU or MAME), in that they control all the "real" inputs and outputs of the controlled computation, whereas said controlled computation merely declares them and hopes for the controller to do its job. However, whereas operating system kernels and virtual machine emulators work at the very low level of abstraction of a CPU or virtual machine, a controller works at the level of abstraction of the programming language being abstracted; it can therefore intercept and implement atomic operations much higher-level than its lower-level cousins could without costly pattern recognition and speculative strategies to dynamically reverse-engineer the translation from high-level language to low-level primitives.

8.3.7 Removing Complexity

Moving functionality from programs to controllers enables a tremendous simplification of most programs; this makes it easier and cheaper to write, debug and document them, but also to reason about them and to ascertain their correctness or security properties. Programs do not have to deal with graphical interfaces, do not have to be linked against huge libraries that are hard to audit, do not have to have unrestricted access rights to storage and I/O subsystems, do not have to possess access keys to cloud providers, just to interact with the user or safely persist data.

Now, the astute reader may remark that while programs have been simplified, that the complexity was moved to these new entities, controllers. Is it then a net gain? Or was the complexity merely swept under the rug, to come back later with a vengeance? Of course it is always possible to misuse the controller mechanism to introduce gratuitous complexity. The question is whether it is *possible* for developers to use the mechanism for good. And the answer is that yes, it is possible, for the following reasons.

First, a lot of the controller code base can be shared between most programs, which means a better pooling of resources to secure a smaller attack surface. Second, that shared controller code base can itself be divided into many smaller modules, each of them a computation with limited capabilities, which makes them easier to work with than parts of a large monolithic computation. Third, the part of control code that is not shared can be reduced to plumbing that routes data from one computation to another; while it requires some access rights not available to random computations, this routing code can most usually be kept very simple, and doesn't need unrestricted access rights either.

The "dangerous" unrestricted code can thus be pushed back to the controller's controller, etc., back to the general purpose user or developer shell shared by all computations. Such a shell is a necessary component of any system whether it is reflective or not, so the attack surface has been reduced to the its very minimal size, whereby new access rights need only ever be granted but with explicit user input and verification. For production code, this "shell" can also enforce additional security measures such as managerial oversight, review by several experts, verification by an entire QA infrastructure, alerts if new rights are used or anything special happens, precise logging of all inputs and their origins, sufficient to reconstitute the computation, etc.

Move the below to some other section, e.g. in chapter 9? XXX

All in all, for a system of equivalent functionality, no architecture can (by hypothesis) simplify any evaluation path in the system, since the equivalent functionality means data and control must flow from inputs (and state) to outputs (and state) through equivalent decision and transformation trees. But a better architecture can still enable overall simplification (compared to other architectures) by allowing sharing and reuse of code along new dimensions, by making easier for developers not to introduce incidental complexity. Less visibly yet perhaps more importantly, a better architecture can help make each component of the system easier to develop by reducing the cognitive load required to think about each component; minimizing the number of interactions possible at any moment, and thus the size of the space that programmers have to understand when editing any component or assembly of components. Thus, a better architecture cannot possibly enable more local simplifications than another one, but it can enable better global simplifications when you consider the entire programming toolchain rather than simply the state of the target software at a given time. I argue that a reflective architecture is better indeed than its irreflective counterpart.

8.4 Implicit Effects

8.4.1 Communication: Implicit vs Explicit

A Reflective Architecture can enable communication at a higher level of abstraction by keeping most of it *implicit*. This requires some explanation as to what I mean by implicit and explicit.

In the case of explicit communication, a process specifically names another process, whether an existing one or a new one to be started; it then opens a communication channel with that other process, and proceeds to exchange data. Explicit communication does exactly what the programmers want (or at least say, since there is no DWIM); thus programmers control how much complexity they will afford; but it requires tight coupling between the programs (and thus programmers) on all sides of the communication, and is difficult to extend or adapt to suit the dynamic needs of the end-user.

Conversely, communication with other processes can be implicit: something outside some process grabs data from it, and makes it available to some other process. This is the case with copy-pasting, or with piping the standard output of one process into the standard input of another. Implicit communication is controlled by the users of a computation rather than by the programmers who write it, and is therefore adapted to their needs. It sometimes require complex support from the computations that partake in it (or, we'll argue, their controller); but programmers don't have to worry about computations on the other side, as long as they abide by some general protocol (and keep up with its updates).

Note that implicit vs explicit is a continuum rather than a clear cut distinction: every communication is partly explicit, because it necessarily involves grabbing data that was somehow published by the first process, the publishing of which wasn't optimized away; and every communication is partly implicit, because it always relies on something in its context to effect that communication, in the controller, "at the meta-level" (as known from famous paradoxes, no consistent formal system is perfectly self-contained). Another name for this dimension of software design is declarative vs procedural programming: In the declarative approach, programmers describe what is being computed, without specifying how it is going to be computed or how it will be further processed, which will be determined by strategies at the meta level. In the procedural approach, programmers describe the steps of the computation without specifying what is going to be computed, and all the operational semantics remains at the base level.

A Reflective architecture recognizes the importance of both aspects of communication, implicit and explicit. Traditional irreflective architectures tend to have very limited support for implicit communication, because they lack a general approach (i.e. a meta-level protocol) to handling declarative programs in general and implicit communication in particular. Thus, support for implicit communication requires ad hoc protocols that are quite complex to develop, even harder to standardize, yet ultimately extremely limited in expressive power for the end-user.

8.4.2 Implicit Communication: Copy-Paste

The kind of implicit communication most visible to end-users in traditional systems is copy-paste: applications interact with a graphical interface, and may allow the user to either copy or cut part of a document being displayed; the clipping is then stored in a global clipboard (with space for a single clip). Another application interacting with the graphical interface may then allow the user to paste the clipping currently in the clipboard into its own document. The two programs may know nothing of each other; as long as they properly partake in the protocol, they will have communicated with each other as per the desires of the end-user. Copy-pasting alone provides user-controllable implicit communication between most applications, and is an essential feature in traditional computer systems.

Now, on traditional computer systems, copy-paste requires every participating application to specially implement large chunks of graphical interface support. Every application then becomes somewhat bloated, having to include large graphical libraries; in modern systems these libraries can to a point be shared between applications, though "version hell" may actually limit the amount of actual sharing. Applications also have to properly initialize these libraries, follow their protocols, abide by the strictures of their event loop, etc. They have to be able to negotiate with the clipboard server the kinds of entities they can copy and paste, and/or convert between what the server supports and what they can directly handle. This architecture where all features are implemented at the same level of abstraction contributes significantly to the complexity of applications; applications are therefore hard to reason about, brittle and insecure. The overall graphical environment will in turn inherit the unreliability of the applications that partake in it. And despite all this complexity, often some application will fail to support copying for some of the information it displays (e.g. an error message); the feature is then sorely missed as the user needs to copy said information by hand, or falls back to some low-level means of information capture such as screen copy (assuming the information fits in one screen), or memory dump (for more advanced developers, assuming suitable access rights).

An interesting exception to the rule of the above paragraph is the case of "console" applications: these applications display simple text to a "terminal emulator" straight out of the 1970s, at which point all the output can be copied for further pasting. The terminal emulator thus serves as the back-computation responsible for presentation of the application output, and handling copy-paste. This comes with many limitations: Only plain text is supported, not "rich text", not images. Lines longer than the terminal size may or may not be clipped; or may have an end-of-line marker or escape character inserted; layout artefacts may be included (such as spaces to end-of-line, or graphic characters that draw boxes in which text is displayed). Selecting more than a screenful may be an issue, though you can sometimes work around it by scrolling the terminal, by resizing it, or by switching to tiny fonts. Standard output and error output may be confusingly mixed, and interspersed with output from background programs. Connecting and disconnecting from terminals is possible, but only if the program is started inside of the `screen` or `tmux` utility, at which point the program cannot use any of the extensions provided by the underlying terminal. Still, the principle of a back-computation to handle display already exists in some traditional computer systems; its protocol is just limited, unreliable, baroque and antiquated.

A reflective architecture generalizes the idea that presenting data to the end-user is the job of a back-computation separate from the computation that displays the data; this back-computation, the controller, is part of a common extensible platform, rather than of the self-contained "application" that underlies each activity. The display manager will thus manage a shared clipboard; this clipboard may contain more than just one clip; it may contain an arbitrarily long list of clips (like the Emacs `kill-ring`). Also, clips are annotated with source domain information, so that the user shall not unintentionally paste sensitive data into untrusted activities, and may not paste data of an unexpected kind that would cause errors or security issues. The platform manages interactive confirmations, rejection notifications, and content filters, that are activated when users copy or paste data.

In these aspects as in all others, the platform can be extended by modules and customized by end-users. Other back-computations beside the display manager can reuse the same infrastructure: they can use their own criteria to select data from a program's output; they can use the selected data for arbitrary computations, and store the results into arbitrary variables or data structures, not just a common clipboard; they may consult the history of the selected data, or watch the data continuously as it changes, instead of merely extracting its current value. The fore-computation doesn't have to do anything about it, except properly organize its data so that the external back-computations may reliably search it.

8.4.3 Implicit Communication: Unix Pipes

As another instance of implicit communication in traditional systems, one of the great successful inventions of Unix was the ability to combine programs through *pipes*: regular "console" applications possess a mode of operation where they take input from an implicit "standard input" and yield output into an implicit "standard output", with even a separate "error output" to issue error messages and warnings, and additional "inherited" handles to system-managed entities. A process usually does not know and does not care where the input comes from and where the output is going to: it may be connected to a communication stream with another process, to a terminal, or to a file; the *parent process* setup the connections before the program started to run.

The Unix parent process here plays a bit of the role of a controller, but this role is very limited and only influences the initial program configuration. The `ptrace` utility makes it possible to control another process at runtime after it is started, but it is very unwieldy, non-portable, and inefficient, which may explain why it remains uncommon outside its intended use as a debugging tool. Still, even within this limitation, Unix pipes revolutionized the way software was written, by allowing independent, isolated programs to be composed, and the resulting compositions to be orchestrated into scripts written in some high-level programming language.

A reflective architecture very much acknowledges the power of composing programs; but they are not so restricted as with Unix pipes. Back-computations enable composition of programs of arbitrary types, with arbitrary numbers of inputs and outputs all of them properly typed according to some high-level object schema, rather than always low-level sequences of bytes. (Note that low-level sequences of bytes do constitute an acceptable type; they are just rarely used in practice except in a few low-level programs.) These typed inputs and outputs all provide natural communication points that can be used to compose programs together.

Unlike the typical parent processes of traditional systems, the back-computations of a reflective architecture can control more than the initial configuration of applications. They can at all time control the entire behavior of the fore-computation being evaluated. In particular, side-effects as well as inputs and outputs are typed and can be injected or captured. Virtualization is a routine operation available to all users, not just an expensive privileged operation reserved to system administrators.

8.4.4 Explicit Communication

There are many obstacles to explicit communication in traditional systems.

A first obstacle, is the low-level nature of the data that is exchanged with their communication protocols, which constitutes a uniform obstacle to all communications by making them complex, error-prone, and insecure. But these protocols are not low-level only with respect to the data; they are also low-level with respect to communication channels. Traditional programming languages do not support reflection, and communication channels are selected by passing around *handles*, low-level first-class objects (typically small integers); this makes it harder to define and enforce invariants as to how channels may or may not be used within a given process:

any function having a handle can do anything with it, and handles are often easy to forge; thus you can't reason about security locally.

A programming language supporting reflection, while it may express the same low-level protocols as above, would tend to (fully) abstract over them and instead expose higher-level protocols, where the channel discipline as well as the data discipline are expressed as part of the types of the functions that exchange data. Communication channel names become regular identifiers of the programming language; the usual type-checking and verification techniques apply to enforce protocol invariants not limited to data format; and the language may let programmers use dynamic binding to control these identifiers.

A second obstacle specific to explicit communication is that to be a legitimate target to such communication, a program must specifically implement a server that listens on a known port, or that registers on a common "data bus"; where this becomes really hard is that to process the connections, the server must either possess some asynchronous event loop, or deal with hard concurrency issues. Unhappily, mainstream programming languages have no linguistic support for decentralized event loops, and make concurrency really hard because side-effects in threads can all too easily mess things up. Libraries that implement a centralized event loop are *ipso facto* incompatible with each other; those that rely on concurrency and a locking discipline are still hard to mix and match, and to avoid deadlocks they require an improbable global consensus on lock order when used by multiple other libraries.

The more advanced "functional programming" languages (including Erlang, Racket, Haskell, OCaml, etc.) support decentralized event loops (the crucial feature being proper tail calls, and for even more advanced support, first-class delimited continuations), and make it easier by supporting well-typed concurrency abstractions on top of a functional programming core, which is a big improvement. But a language that further supports reflection would make it possible to move these servers completely to a separate back-computation that controls the computation you interact with; thus the fore-computation can be written as a simple program, with a very simple semantics, easy to reason about, without any pollution by the server and its complex and possibly incompatible semantics; yet it is possible to tell it to invoke exported functions or otherwise run transactions on its state, by talking to its controller back-computation.

A third obstacle specific to explicit communication in traditional computer systems is the difficulty of locating and *naming* one of those target processes available to communicate with. Indeed, inasmuch as communication is explicit, it requires some way to *name* the party you want to communicate with: a named process (in e.g. Erlang), a numbered port or a named pipe or socket on the current machine (in e.g. Unix), a remote port on a named machine (using TCP/IP), etc. Implicit communication only needs to distinguish between local ports: "standard input", "standard output", "file descriptor number 9", "the graphical display manager" (including its copy-paste manager), etc., without having to know what or whom is connected to it on the other side. Reading (or writing to) a file is intermediate between the explicit and implicit: you know the name of the file, but not the identity of who wrote the file (or will read it). Naming a port can also be considered more implicit and less explicit than naming a process.

Now, traditional systems do not have orthogonal object persistence; therefore all their connections and all their names are transient entities that must be reestablished constantly. Traditional systems also have no notion of dynamic environment; there is a static environment, set at the start of a process, but it doesn't adapt to dynamic changes. To track dynamic changes, programs can query servers, but then the behavior is either completely unconstrained or highly non-local. You can try to automate this communication, but every program has to handle a vast array of error cases. In any case, local reasoning about dynamic properties is nearly impossible.

A reflective architecture enables orthogonal object persistence: controllers will transparently save a computation's progress to permanent storage, and make it possible to use a stable name to access a remote service wherein any underlying connection is established or reestablished when needed. Controllers can also use dynamic binding as a language feature to control the behavior of programs or groups of programs in a structured way. How does a reflective architecture deal with transience, reconnection and unreliability at lower levels of the system? It abstracts the issues away by introducing a clear distinction between fore-computation and back-computation: the fore-computation is written in an algebra that can assume these problems are solved, with persistent naming and dynamic reconnection both implicitly achieved; the back-computation takes care of these issues. Local reasoning on small simple programs (whether at the fore or in the back) keeps the overall complexity of the system in check while ensuring robustness.

8.4.5 Explicit Communication: Locally Constant Servers

At the extreme end, opposite to implicit communication, the communication is so explicit that the system knows exactly what's on the other side of a communication portal. The inter-process communication can then be reduced to a static function call, and the listening function on the other side can often itself be inlined. And in a reflective architecture, this may indeed happen, automatically: the runtime division into functions, as optimized for execution speed, need not at all match the source-level division into functions, as optimized for meaningful separation of responsibilities.

Indeed, when it doesn't change very frequently, whatever is on the other side of any communication channel can be considered locally constant; then, whichever back-computation handles connecting the communicating parties, whether a linker or JIT, can optimize all communication into function calls, and function calls into more specific instructions; it can then wholly eliminate unnecessary marshalling and unmarshalling, and reduce all higher-order functions and indirections to efficient loops, until a change in the connection or in the code invalidates these optimizations.

Of course, sometimes the optimization that makes sense goes the other way, transforming function calls into communication with another process: a process on a CPU might delegate computations to a GPU; an embedded device when power is at premium, including a mobile phone, might rather query a server than run an expensive computation itself, etc. Thus local CPU cycles can be saved whenever cheaper, faster and/or more energy-efficient resources are available. And there again, a more declarative approach allows back-computations to automatically pick a better strategy adapted to the dynamic program context.

In the end, factoring the code in terms of fore-computation and back-computation is an essential tool for division of programming labor: The fore-computation developer can focus on expressing pertinent aspects of the computation semantics; he can write smaller programs that are simpler, easier to reason about, easier to compose; they can be written in a domain-specific language, or, equivalently, in a recognizable subset of his general-purpose language with well-defined patterns of function calls. The back-computation developer can focus on implementation strategies and optimizations; he has a relatively simple, well-defined framework to prove their correctness, whether formally or informally; and he can focus on the patterns he is interested in, while leveraging the common platform for all other evaluation patterns, instead of having to reinvent the wheel. Thus, whether the source code for some part of an application is modular or monolithic is wholly independent of whether the implementation will be modular or monolithic at runtime. The former is a matter of division of labor and specialization of tasks between programmers at coding-time; the latter is a matter of division of labor and specialization of tasks between hardware components at runtime.

At every level, each programmer can and must use explicit names each implicitly bound to a value, to abstract any process, function or object that belongs to another programmer. By hypothesis, the programmer never knows for sure what the name will be bound to —

though often that other programmer may well be the same programmer in a different role at a different time. Yet the overall system in time can always see all the bindings and statically or dynamically reduce them, efficiently combining all parts of a programs into one. Names allow to express fixed intent in an ontology where the extent will change (the extent being the value of a variable, or the text of a function, etc.); they are superfluous from the perspective of a static computer system, because for a computer system any name beside memory addresses and offsets is but a costly indirection that is better done away with; names are important precisely because programming is part of a dynamic computing system, where the activities of programmers require abstraction and communication across programmers, across time, across projects, etc.

8.4.6 Declarative I/O

From a programming language point of view, input/output (I/O) primitives can be seen as a set of functions $\texttt{iop}_k$ that each take as parameter some output data type $\texttt{out}_k$, and, with some side-effects (which we represent as a dashed arrow, but could be expressed as a monad or a profunctor), returns a value of the input data type $\texttt{in}_k$:

$$\texttt{iop}_k \ : \ \texttt{out}_k \dashrightarrow \texttt{in}_k$$

Of course, in case of pure input, the output type is the unit type, whereas in case of pure output, the input type is the unit type. Using either dynamic types or dependent types, the index k could be a parameter inside the language, rather than a parameter in the meta-language. there would then be a function `iop` that takes the index k (if the set of indices is larger than a singleton) and a value of type $\texttt{out}_k$, and after side-effects returns a value of type $\texttt{in}_k$. Using type classes or otherwise subclassing, a single function `iop` could similarly take a generic class `out` that has many subclasses $\texttt{out}_k$, and return a value of generic class `in` that has many subclasses $\texttt{in}_k$, though without dependent types, the correlation between the two, if non-trivial, might then be lost.

Whichever way it is encoded, the data r of k (if needed) and of a value of type $\texttt{out}_k$ (if needed) is an I/O request. The data e of d and the corresponding return value of type $\texttt{in}_k$ (if needed) is an I/O event. The controller intercepts the requests and must (synchronously) reply to each with a value of the input data type before computation may continue.

In a closed deterministic context, if the controller provides it, the return value can be deduced from the history of requests, and the interaction can be reduced to a value in the free monad of I/O requests (i.e. two arrows between given states are equal iff they correspond to the same history of I/O requests). But in general the controller does not have to provide such a closed deterministic context. The context it provides can be open in that it depends on communication within a wider context as provided by the controller's controller: communicating with other services on the same computer, or on other machines; input/output with analog devices outside the world of digital computations; ultimately, interaction with humans and the human world. The controller can compute the input value with a simulated computation of I/O that may not be deterministic. It can also have that value depend on the state of the computation, or on the history of the computation, or on multiple evaluation histories of the computation.

The controller doesn't even have to continue the computation. Indeed sometimes it can't, because there was some kind of I/O error, or because there is no useful and valid value of the proper type to return in the given context. And sometimes, it won't, because the computation isn't authorized to acquire resources required, or lacks the access rights to enact this particular action. What the controller may do in these case is modify the evaluation strategy of the computation so it doesn't have to continue. For instance, the controller could stop the computation; or it could abort the current code transaction due; or it could automatically restart said transaction, and try a few times before it gives up, possibly with randomized exponential backoff between attempts. The controller could also schedule a different branch of a non-deterministic evaluation.

8.4.7 Security Considerations

From a security point of view, note though that a controller can only express and enact access restrictions within the universe of abstractions it sees. It is always possible that there be what security experts would call *side channels* that are not part of the formal universe of the controller, but part of a wider universe: a typical example is power consumption as observable in the surrounding physical analog world in which digital devices are implemented, that often allows an attacker with physical access to a running device to extract cryptographic keys from it; with local code access, an attacker might also use timers or cache miss counters to observe other programs or leak information. On the other hand, such invisible side-effects are quite useful indeed and a security asset rather than risk: for instance, logging, tracing and debugging can help detect and debug issues. In the end, controllers are tools, and whether controllers are used for good or evil, whether they are used skillfully or incompetently, depends on he who wields them.

One way to see things for people familiar with programming language implementations, is that a controller can do anything an interpreter can do, except this interpreter is restricted to only evaluating a given program the computation, within a context that binds the high-level side-effects that it must or must not have. Note that the controller is itself a computation that is constrained by the access rights and computation obligations that it may or may not possess besides those associated to the computation it controls. For instance, the controller might have access to a logging service not seen by the fore-computation, but not to the files used by the fore-computation (except inasmuch as it may access them as part of implementing the fore-computation). Or the controller might have full access to some filesystem, and may be implementing the semantics of file access for the fore-computation.

How much access a controller does or doesn't have, what effects remain implicit and what effects become explicit, depend on *its* own controller. Moreover, the effects explicitly allowed and excluded when specifying a computation and its successive back-computations may or may not suffice to ascertain security properties desired of the computation.

But let it be clear though that lies behind or below a computation is out of the control of said computation, and that a computation cannot be blamed (or praised) when some security issue lies in this control context or execution context (or when no such issue exists): by definition this context lies outside the computation, that has no control on it (besides declaring some of them wholly invalid). Some completely secure bugfree program can be deployed in a flawed context that makes it completely insecure; or a malware can be deployed in a sandbox or honeypot that makes it safe to execute and study. Therefore the security properties of a program are not the security properties of the program in context, and vice versa, though the two may be related via the properties of the context. A secure service deployment requires not just the computation but all its back-computations and hypo-computations to be secure, which involves what is usually considered “operations” as well as what is usually considered “development”.

8.4.8 Dynamically reconfiguration

Every implicit communication becomes explicit communication in *some* back-computation, its handler, though it may remain implicit in any given back-computation that doesn't handle it. The back-computation that handles a given kind of implicit communication can specify how that communication is implemented. This implementation can be stateful in ways that the implicit communication isn't: for instance, whereas the fore-computation may have a constant implicit communication channel, the back-computation may back it have a variable explicit communication channel; the implicit communication channel could be "the current audio-video output", and the explicit communication channel that backs it can be at one time the user's cell phone, at another time his laptop or the conference room's large display, or a "tee" that outputs to both the conference room's display and part of the laptop's display (the other part having notes and a preview of what comes next). The back-computation may change from one to the other while the computation is running, without the computation having to notice or being able to notice.

These capabilities already exist to a point in non-reflective systems; but they are only available at fixed levels of abstraction, usually relatively low-level, and most applications must either share the same implicit low-level configuration, or explicitly handle all communications the hard way. A reflective system makes it possible to affordably modify the configuration of individual computations or groups of computations at the level of abstraction that matters to them, at the level of abstraction that matters for these computations, and without having to share the entire configuration with all other computations, by changing their back-computation that handles this aspect of their configuration.

8.5 Full Abstraction

8.5.1 Requirement for Migration

For migration to be possible and meaningful, computations must all have a clear opaque bottom: (1) it must be perfectly clear what the bottom is; and (2) the bottom must be totally opaque, such programs above cannot see below. This requirement is known semi-formally as *Full Abstraction* [1].

The reason for the requirement of a full abstraction is that in presence of migration, what's "below" can change at runtime; therefore any indirect way that it may be detected (e.g. timing) is unreliable and temporary: querying at different times may yield different results, or, if done in the middle of a migration, may yield incoherent results. Depending on those results to remain constant throughout the computation for results to be correct means that either migration is prohibited or the computation will yield incorrect results. Hence the requirement.

8.5.2 Full Abstraction Mechanisms

Some languages such as Haskell or ML use strong static typing with parametric polymorphism to make it possible to express full abstraction over some classes or modules, wherein clients of those classes or modules cannot observe the implementation of the class or module (short of using some special reflection API).[citation needed] The *parametricity* of type-abstraction ensures full abstraction of the definition of a Haskell type class or ML module over the underlying implementation of its parameters. On the other hand, an instance of a Haskell type class or ML module will statically bind the parameter at compile-time and it cannot be changed at runtime.

Racket offers chaperones and impersonators[citation needed] as reflective primitives on top of which language abstractions can be defined the details of which are opaque to clients at runtime. Assuming the implementation indeed dynamically prevents users of these objects from accessing their internal state, and that the implementation indeed fulfills the high-level contract for the objects without exposing any backdoor, then indeed these primitives enabled the developer to write full abstractions.

Computability Logic[13] is a general purpose formalism based on Game Semantics, that generalizes at the same time classical logic, linear logic, and constructive logic. It distinguishes several kinds of quantifiers that other formalisms don't, including a notion of *blind* quantifiers wherein the strategies under the quantifier are not allowed change their behavior based on the value (or type) bound by the quantifier (say matching the value of the bound variable using a `case` or `typecase`), whereas such discriminations is allowed by a regular quantifier. Blind quantifiers therefore correspond to full abstraction, whereas regular quantifiers correspond to regular abstraction.

8.5.3 Leaky Abstractions

The opposite of a full abstraction is a *leaky abstraction.* Most abstraction layers provided in most programming languages and APIs are leaky from all sides, and it takes little effort to peer into the innards of the "abstraction".

Languages like Common Lisp offer primitives to redefine classes at runtime, change the classes of existing objects, or simply re-bind functions associated to a symbol.[citation needed] Yet other languages, such as Erlang, enable and encourage a style where messages are exchanged between mutually isolated processes and some of these processes can be wholly replaced at runtime without other processes being affected.[citation needed] Correct programs that access functionality through the indirection of provided constructs can indeed benefit from the implementation behind those constructs being migrated at runtime. However, these languages *require* the users of these constructs to be fully abstract with respect to their implementation, but they offer no mechanism to *enforce* this full abstraction, which remains the responsibility of both developers who builds them and use them.

Now, in any language, the underlying implementation could (at least in theory) be changed under a running program without the program noticing (as long as all indirect runtime data is modified covariantly). However, most languages do not offer abstraction mechanisms to embody (much less formalize, much less enforce) the contract specifying what properties must be preserved by each side of the computation.

A general mechanism to achieve full abstraction is to write domain-specific languages (DSLs), usually through an interpreter, or (usually with more efforts) through a compiler. However the barrier to entry to implementing these DSLs is high, whereas the resources available to develop and maintain these implementations are limited. Therefore, for economic reasons, these implementations are often of relatively bad quality: they provide little tooling if at all; their abstractions are usually involuntarily leaky, and of course they do not allow for migration.

Better languages, usually those of the Lisp family, have macros, which allow for the incremental and composable definition of compilers. These macros make for much more robust abstractions, but usually limit these abstractions to incremental extensions of the base language; and even these abstractions often leak semantic details in subtle ways that preclude migration at runtime if programmers use primitives below the abstraction, which these languages cannot detect or prevent. Racket, a distant descendent of Lisp, has the most advanced support for defining and supporting domain-specific languages; not only does it provide macros, it also has a notion of first-class languages, such that old constructs can be prohibited as well as new constructs are provided; this mechanism does allow for full abstraction, unlike macros alone, but (at least at this time) not for migration at runtime.

8.5.4 Full Abstraction as Security

Every violation of some abstraction *is* a security risk and every security risk *is* violation of some abstraction: A security risk is when a bad actor may get a program to behave in a way that it shouldn't be allowed to do. This subversion of the program's intent is the same as a violation of the abstraction that the program was supposed to provide — had it been well formalized and correctly implemented.

Therefore, if and when an adverse attempt at subverting the abstraction is detected, security flags shall be raised; the police shall be involved ASAP and the perpetrators arrested and prosecuted. An exception of course is when the perpetrators are part of a legitimate penetration testing team. In particular, customers of hosting services shall reserve the right to test that the abstractions they were provided with are not leaky (or have professionals test it for them).

Recognizing that any program that interacts with users or other systems uses a language be it implicitly, and that this language must provide a full abstraction or be a security risk, is the basis for Language-Theoretic Security[citation needed]: this research domain encourages identifying what language each interaction uses, and making sure that these language is appropriately restricted, that parsers and printers are strict enough that inputs and outputs are sanitized, that said parsers and printers are not mixed with application code in a way that makes it all too easy for abstraction leaks to happen, that evaluators do not have backdoors, hidden side-effects, buffer overruns, cross-site scripting, unauthorized accesses or unwittingly Turing-complete computations, etc.

8.5.5 Breaking Abstractions for Greater Good

There are legitimate reasons to break an abstraction barrier, to query the underlying system, and to use its primitives. Sometimes the primitives provided are semantically incomplete and do not suffice to express the desired program with all required guarantees. Sometimes they do not offer satisfactory performance in terms of speed, latency, energy consumption, resource usage. Sometimes the performance gains from a lower-level approach make a meaningful economic difference. Sometimes the developer wants to include libraries written in a lower-level language, or to use instrumentation available at a lower level of abstraction. Sometimes, the underlying abstractions are themselves leaky or buggy, it might be necessary to detect that this is the case so as to provide a better tighter abstraction. There might be more reasons.

Now, the right way to break the abstraction barrier is to keep the top computation as abstract as possible, and express it in a DSL that abstracts away (hides) all these features, optimizations, bugs and leaks. Then, that DSL can itself have multiple implementations depending on the underlying computing basis, and these implementations indeed can and should take advantage of available features and optimization opportunities while bridging over underlying leaks and bugs. Thus, the top computation can be implemented using a hypocomputation that uses a particular implementation of the DSL that uses knowledge of what's "below", yet it can always be migrated to a different implementation of that DSL, because that DSL itself provides a full abstraction, even though it may be implemented on top of computing bases that don't.

Importantly, even though the emotional motivation was for the developer to break an abstraction, the result in the system is a more robust abstraction. What happens is that a computation formerly defined in terms of an abstract system A, was redefined in terms of a different abstract system B extended or modified with a suitable DSL; and that DSL is itself explicitly defined in terms of a more concrete system C, though in so doing it may reuse most of an implementation of A with C. The notional "breaking of the abstraction" corresponds to extending the "vertical" scope of the specified computation from A down to C: the new computation is specified in greater details than the previous one, in terms of a lower level virtual machine; migration to different low-level virtual machines remains possible thanks to the formalization of that alternate abstract system B, that can have implementations not just with C but also with other low-level machines D, E...

8.5.6 Activity Sandboxing

In a reflective architecture, proper sandboxing ensures that activities may only share or access data according to the rules they have declared and that the system owner agreed to. In particular, applications that are not meant to communicate with anyone but the user (e.g. regular web pages) will not be able to to communicate with anyone but the user. Proper sandboxing also means that users need not be afraid of getting viruses, malware or data leaks via an activity (though he should still avoid running code suspected of being malicious, just because there may be inadvertent bugs or wanton backdoors in the software platform he is using).

Systems with a reflective architecture always run all code in fully abstract sandboxes, as controlled by a user-controlled meta-program. There is no supported way for code to distinguish between “normal” and “virtualized” machines. All machines are “virtualized”, that what’s “normal” in a reflective architecture. If the system owner refuses to grant an application access rights to some or all requested resources, the activity has no direct way to determine that the access was denied; instead, whenever it will access the resource, it will be suspended, or get blank data, or fake data from a randomized honeypot, or a notification of a timeout delay, or whatever its back-computation is configured to provide; the system owner ultimately controls all configuration. If the application is well-behaved, many unauthorized accesses may be optimized away; but even if it’s not, it has no reliable way of telling whether it’s running ”for real”, i.e. whether it’s connected to some actual resource or to some cheap emulation thereof.

Allowing code to make the difference would be a huge security failure; and any time a monitor in a production system recognizes the attempt by a process to probe its environment or otherwise break the abstraction, a serious security violation is flagged; upon detection, the process and all its associated processes are suspended, up to the next suitably secure meta-level; also the incident is logged, an investigation is triggered, and the responsible software vendor is questioned. — Unless of course, the people responsible for the break in attempt are the system’s owners themselves, or penetration testers they have hired to assess and improve their security, which is a recommended practice among anyone hosting computations controlling any important actual resources.

Note that proper sandboxing at heart has nothing whatsoever to do with having “kernel” support for “containers” or hardware-accelerated “virtual machines”; rather it is all about providing *full abstraction*, i.e. abstractions that don’t leak. For instance, a user-interface should make it impossible to break the abstraction without intentionally going to the meta-level. You shouldn’t be able to accidentally copy and paste potentially sensitive information from one sandbox to the next; instead, copy and pasting from one sandbox to another should require extra confirmation *before* any information is transferred; the prompt is managed by a common meta-level below the sandboxes, and provides the user with context about which are the sandboxes and what is the considered content; that the user may thus usefully confirm based on useful information — or he may mark this context or a larger context as authorized for copying and pasting without further confirmations.

Chapter 9

Architectural Benefits

9.1 Performance Improvements

9.1.1 Migration-Time Optimization Opportunities

The benefits from a reflective architecture are many. However, the easiest to sell is probably performance, since it speaks directly to the bottom-line of users of large computations or of small computers. Indeed, a reflective architecture allows for a wide class of runtime optimizations that are not available in non-reflective systems: a reflective architecture introduces the notion of migration-time, distinct from either compile-time and runtime, yet interspersed with runtime, that corresponds to migration events. With this migration-time come a lot of optimization opportunities. From the point of view of software implementation techniques, these migration-time optimizations can be seen as a generalization of the now well-established JIT compilation[citation needed].

When migration happens, a lot may be known about the state of the computation, that is not known at compile-time and therefore cannot be used for static compile-time optimization:

- The configuration data for the current computation, including parameter values, logging level, etc.

- The dynamic state of the current computation, including size and usage statistics of various entities, specialized knowledge about currently active data and control structures, symmetries of the problem at hand, etc.

- The evaluation context of the current computation, including the exact variant of underlying hardware being used, the exact resources available, the exact versions of each library used, the exact addresses of each function and variable linked, the exact types or control flow graph used in the overall computation, etc.

- The context of other computations that the current computation is interacting with, including all the above for each of these computations, whereas they are hidden behind abstractions at compile-time.

The information available at migration-time has both a wider scope and more precise knowledge than is available at compile-time, and thus enables optimizations that are inaccessible at compile-time: First, what is typically considered as "global" information at compile-time can only encompass the current computation and be blind to any specific context, whereas what is considered as "global" information at migration-time can span all the active computations on the computer system. And second, compile-time only has access to statically known information in all possible contexts, whereas migration-time has access to dynamically known information in the current context. Migration-time still has less information to use than available at runtime, since e.g. variable values can be modified more often than a migration happens.

9.1.2 Migration-Time Optimization Constraints

The migration-time optimizer may use all the information discussed above; however, as restriction, all its optimizations must be *reversible* with respect to the declared computation semantics. In other words, it must always be possible to observe the state of the current hypo-computation being executed as implementing the hyper-computation specified by the user (or a computation at any level of abstraction in-between), and from there to migrate to a different implementation. As usual, the specified abstract system may be defined up to some set of acceptable rewrites, and optimizations may cause the concrete computation to be observed as any of the acceptable rewritten variants of the computation.

Thus, local invariants of the current implementation cannot be assumed to be global invariants, since they can be invalidated by the next migration; (unless of course they are global invariants indeed, but then the compile-time optimizer will probably have exploited them already before the migration-time optimizer gets a chance to do anything with it). This restriction is quite constraining indeed: by hypothesis, some irreversible optimizations will have to be eschewed that could have been applied if no observability were needed because the low-level implementation were known to never change. On the other hand, it only takes a safepoint once every so many iterations of a tight loop to achieve observability, so the constraint isn't a significant runtime burden in the kind of code where performance really matters. What the constraint does, mostly, is to forbid "puns" that lose information by identifying entities that are distinct at the high-level but could have been merged at the low-level if observability weren't needed.

Thus for instance, if at migration-time it is found that some variable `a` is always equal to 2 in the context of the new implementation, then any use of variable `a` can be replaced by a use of the constant 2, and the information that this is a constant can ripple through the computation as constant folding propagates it through the program. Thus, if the computation includes a function of the form `x ↦ x + a` then the function can be compiled as if it were `x ↦ x + 2`. Yet some event might cause migration to yet another implementation where `a` may not be 2 anymore, at which point this function's implementation would be migrated as if it were still of the form `x ↦ x + a`, as distinguished from a function of the form `x ↦ x + 2` obtained by other means. If the language allows for function comparison, comparing the two functions must therefore keep returning a signifier of known falsity, even though the current implementation of the two functions is identical.

By contrast functions specified as `x ↦ x + 2` and `x ↦ 2 + x` and maybe even `x ↦ 1 + x + 1` might very well be the very same function under commonly specified sets of acceptable rewrites, at which point the compile-time optimizer would make them equal, and observing them might return the same object — and the migration-time optimizer wouldn't have anything more to do there. Now, note that merging these functions might lead to changing the error message generated by an overflow event (if applicable); therefore the declared semantics that allows such rewrites corresponds to the notion that (at least some) errors are fatal and the message they produce is irrelevant in the abstract — even though of course the implementation will do its best to make them relevant in the concrete. Alternatively, the rewrite happens after (or independently from) error handling being expanded into regular checks to intermediate results, and other rewrites allow to eliminate, fuse, move or otherwise transform the error checks.

In practice, the constraint against losing information may matter most in cases where some bindings are declared as observable and cannot be dropped nor replaced by a more directly actionable summary. To allow optimizing them away, the specification of the operational semantics of the higher-level programming language would then have to somehow declare lexical bindings as not directly observable, or otherwise explicitly allow optimizations that drop information about them.

9.1.3 Migration-Time Optimization Strategies

When implementing an abstract program P using some virtual machine V, the compiler from P to V cannot go below the abstractions provided by V. Compile-time optimizations can only rely on details visible by V; they cannot go lower-level, they cannot peer inside bindings established by blind quantifiers, and more generally cannot break the full abstraction for V.

However, the migration-time optimizer deals with implementing V using a lower-level concrete computation C; it is effectively a compile-time optimizer for implementing V using C. And that implementation may very well break down the abstractions of V into those of C; it can see inside the internals of entities provided by V, and peer at the values bound by what to V are blind quantifiers. The migration-time optimizer will be able to use all the contextual information available and discussed above to efficiently implement the computation.

Thus, not only can locally constant variables be replaced by their values, but constant propagation and partial evaluation may lead to further local simplifications in the resulting hypo-computation. However, to preserve observability, these simplifications must not involve use of these variables past the next safepoint. Now, removing extraneous safepoints is a valid strategy to enable optimization across them; however, in doing so, observability to some further safepoint must be preserved: for instance, it is permissible to eliminate all the safepoints within a tight loop, after unrolling the loop a few times for small enough loops, but there need be a safepoint after only so many iterations of the loop, so it remains possible to observe the computation in a reasonable amount of time.

Now, in a reflective architecture, configuration is often handled by back-computations: the back-computations control which computation are connected to which other computations or to which I/O devices, which window of which display is in use, what is the volume of the sound or the layout of the keyboard, which configuration options are enabled in which instance of each program, etc. The user (or some automated program) may then modify the configuration: e.g. he move, hide or resize a window, connect the audio output to a different device, change the volume, selects a different keyboard input method, pick a different color scheme, tweak persistence parameters, modify the data processing pipeline, etc. Whenever some configuration change invalidates some generated code, the back-computation stops the fore-computation in an observable state, then the old code is replaced or even updated in place (if all code users are migrated at the same time) with newly generated or modified code, all the while preserving the running state of the computation. Since code worthy of being compiled is run frequently, whereas the human-controlled events that cause configuration changes are infrequent, it is often a big win to generate code specialized to the current configuration. Thus, pixels will be blitted directly to the correct video memory location with the correct color depth and cpu optimizations using the specified color palette; bits will be banged directly to the correct device without extra buffering and copying; the correct settings will be used and assumed all around; proper synchronization or suitable optimism will be involved in committing transactions; processes will (usually) communicate directly without involving unnecessary marshalling, context switching, copying and unmarshalling; the resulting combinations of higher-order functions are specialized into tight loops, based on provided argument functions even if these functions come from different processes with different owners and different lifetimes, etc.

Note that there are often good reasons *not* to let computations directly communicate with others or access devices, notably relating to software capabilities and hardware capabilities. As of software capabilities, this device or process may have strict trust requirements that do not allow that particular user to generate code against it from that code base using that compiler, and conversely. while it might be possible to find an acceptable combination that allows for direct access, that combination might be slower than just keeping things in separate processes communicating via a trusted middleman (that nevertheless need not be a runtime "kernel" since it needs not be the same unique master process for all pairs of communicating processes and devices). As of hardware capabilities, depending on the number of physical computing units available, on the geometry of the data interconnects, it may make sense to generate code for a software pipeline of programs in such a way that it maximizes usage of the hardware pipeline; merging all code for a pipeline into a big loop rather than a series of smaller loops might be bad for cache locality as well as usage of available processing resources; while communication between processes does incur communication costs in marshalling, bit-banging and unmarshalling data, this cost is sometimes less than the benefits of better hardware usage (then again sometimes not).

There again, deferring these decisions to back-computations allows for implementation strategies that take into account resources actually available to the computation's owner at the time of the computation request and as the computation keeps running in a changing environment. A static implementation could not adapt to a variable execution context from user to user, machine to machine, day to day, hour to hour, minute by minute. Note that this is what the Borg scheduler or Kubernetes scheduler do at the level of granularity of virtual computers within a distributed system[4],. We argue that the same principle can be generalized and automatized at a finer level of granularity.

9.1.4 Up to Infinitely Faster

The greatest performance savings happen when non-trivial aspects of a computation can be wholly optimized away.

For instance, any kind of explicit interprocess communication requires checking, unparsing, context switching, copying, parsing and checking again. When the processes belong to the same user and are part of the same trust domain, there isn't any reason to go through all these costly steps; they can be wholly optimized away.

When in some user interface a window is hidden from view and no process is watching its contents, then these contents need not be computed at all; all that needs be computed is a record of a recipe to display the window should its contents be shown again. Thus, for instance, when watching a video clip, if the user decides to hide the window or turn off the screen, all the CPU-intensive activity of decoding and scaling the video can be eschewed, leaving only the sound to be decoded; all the video player has to do is keep track of where in the video to resume decoding and display should the user decide to turn the screen back on. On an embedded or wearable device that is being used as a music player, this can not only save a lot of battery life for the device, but also spare the user some serious burns.

In an extreme case, a computation that has been permanently disconnected from any positively-desired output or side-effect can be wholly garbage collected. In his computational-philosophical novel "Permutation City", Greg Egan speculates about what it means for an artificial life simulation to thus permanently disconnect from the outside world while preserving (or indeed increasing) the richness of its internal state. Has this simulation ceased to exist, or does it continue to grow and prosper forever, having been implemented in the most perfect way possible? We do not have to answer the philosophical question. It suffices to note that in a reflective system it is indeed possible to automate the fact that such simulations are run *infinitely faster* than in traditional systems.

Most cases are less extreme, yet it is not unusual that when combining computations and display the results for the user, some of the final and intermediate results are not used, whereas some of the intermediate computations partly cancel each other. In all these cases, the contextual late compilation of code can achieve greater simplifications than possible with either ahead-of-time compilation or runtime interpretation. Of course, simplifying code away requires that the compiler should be able to detect that some code is indeed unused. This is much more easily done when said code is written in a pure lazy functional language like Haskell[citation needed]; but it is possible even in messier programming languages, as long as the programs remain short or that the side-effects are somehow contained; and separating code into fore-computation and back-computation, hyper-computation and hypo-computation, etc., is precisely a way to keep the relevant software components short and to control their side-effects.

9.2 New Features

Obvious to observe yet less obvious to assess among the benefits of First-Class Implementations, are the many software features that they enable. Unlike higher performance, that is evidently worth the costs it reduces while for achieving the same results and the ability to reach previously unatteignable goals, it isn't straightforward it is to see what any particular new feature is worth. Is it just an expensive gadget that only looks nice the first time you see it on display? Or is it a change with a deep and broad impact on how things will thereafter be done? Yet, though it is hard to evaluate these changes, it is still pretty obvious to observe that they exist: some features were indeed enabled (or weren't), and ultimately, new such features are how better software can qualitatively change the life of users rather than improving it in a merely quantitative way.

The features enabled by First-Class Implementations were already described in chapters 5 to 8. The contents of these chapters will not be repeated, but here are some remarks on the significance of these features.

Some software features enabled by First-Class Implementations can be properly considered novel, such as runtime navigation of the semantic tower (see chapter 5). However most of the discussed features are novel on their own, since they each have been implemented before without a Reflective Architecture; still, the combination of these features may itself be novel, and perhaps more importantly their *composability*: First-Class implementations offers a unified framework in which these many features can be expressed, composed, combined, reused, shared, with which these features may build synergies that may not be available without such a framework.

First-Class Implementations make some features cheap and universal when they were previously expensive and reserved for specific uses, or sometimes even prohibitively expensive or impossible without heroic efforts, such as Migration (see chapter 6) or various Code Instrumentations (see chapter 7). Yet, while migrating processes from one machine to another while it's running is the most spectacular immediately understandable application of First-Class Implementations, the features that promise a deeper change in the way people write software might be some that have existed for decades in some systems but never could previously be incrementally added to other systems: Orthogonal Persistence (see section 7.3), and Erlang-style Resilience (see section 7.4).

In any case, First-Class Implementations offer a promiseful approach to implementing a lot of "non-functional" requirements of software that impact not just the quality of software written, but the way software is written.

9.3 Robustness: Separate program and metaprogram

9.3.1 Code Quality by Any Other Name

One less obvious benefit of First-Class Implementations is Robustness. Here by *Robustness* we will attempt to at least partially formalize the notion of Code Quality, including Readability, Writability, Understandability, Reasonability, Debuggability, Maintainability, and whatever other "-abilities" and "non-functional requirements" code may possess that people do not usually know how to usefully measure and compare. The notion is of course partly subjective, as what is easy to understand for one person may be hard to understand for another and vice versa, probably depending on intellectual factors both innate and acquired, the latter including personal knowledge as well as cultural practices. Yet, assuming that at least a few people in a development team can master the concepts, we will argue the contributions, limitations or potentials that a reflective architecture brings on the table that a traditional architecture doesn't.

Now, even taking into account subjective factors, even most practitioners have trouble evaluating the Robustness of a body of code, much less predicting what effects on Robustness certain practices will have. Indeed, evaluating Robustness requires thinking at a different scale than is required for the day-to-day development or operation of software; furthermore, predicting Robustness requires imagining potentials of future development and the counterfactuals of alternative development histories in addition to looking at the results of actual past development. Non-practitioners cannot even fathom Robustness, because in addition to the above, it is an abstract concept in a domain that is foreign to them.

Robustness, when present, is not felt: it consists in met expectations and other non-events. However, all will feel the consequences of the absence of Robustness, when software breaks down badly, its maintenance costs balloon, or some crime or other catastrophe makes this absence all too obvious. Yet when these consequences are felt, the emergency of dealing with these consequences in the short term also makes it a wrong time to think about Robustness in the long run. Therefore the right time to think about Robustness is now: not every minute, not necessarily every day, but probably every week, every month and every year. It must be part of every development team's concerns, and for each scale at which software design and planning happens, people who understand it at that scale must be involved.

Now, when trying to judge a software architecture from the point of view of Robustness, the question asked is: how easy or hard does the architecture make it for developers to write software that is adapted to the domain with which the software interacts in the world at large, and then for developers to maintain this software by modifying it when that domain changes, including changes in adversarial ways? And the main contribution of an architecture is usually to keep things simple and easy to reason about in terms of the concepts of its domain of interaction.

9.3.2 Defining Robustness Negatively

Robustness isn't directly felt, but its absence is — this makes it hard to formalize what Robustness is. By contrast, this makes it relatively easy to define what Robustness isn't, and to identify entire categories of badness the presence of which constitutes the opposite of Robustness. As regards the contribution of software architecture to Robustness, this means looking at the kinds of systemic software failures that software architectures may lead to.

There are many ways that an architecture can fail. A first failure mode, the complexity failure, is for the architecture to introduce too much extrinsic complexity: much more complexity than necessary, that developers have to deal with, which can arbitrarily increase the cost of using the system to deal with the intended domain. In the extreme case, developers can spend a large part of their time dealing with issues created by the system itself, to the point of being distracted from addressing those of the intended domain. A second failure mode, the conceptual failure, is for the architecture to be overly simplistic and fail to adequately describe the intended domain, or to even enable users to properly express the required concepts. Users then have to each develop their own workarounds and extensions (if possible at all), and have difficulty cooperating with each other because the system does not let them communicate using the adequate concepts. Sometimes the conceptual failure turns into the complexity failure, as users eventually develop convoluted ways of expressing the concepts they need, but then have to deal with all the baggage of expressing these concepts on top of a system that didn't allow their direct expression. These failure modes are of course on a continuum: there are infinitely many degrees of extrinsic complexity that a system can introduce on some aspects of the software, and infinitely many degrees of oversimplification and conceptual inadequacy that it can simultaneously have on other aspects of the software.

A different failure mode, the implementation failure, is for the architecture to offer adequate concepts to its users, but to fail to implement them properly: the abstraction provided by the system may leak, and developers will then have to address a large number of edge conditions and weird failure modes; the leaks might then cause frequent system crashes, or worse, they can be as many security vulnerabilities that expose users to attacks; users must therefore constantly work to bridge the unfilled gap between the system's promises and what it actually delivers.

There again, this third failure mode can be based an intrinsic or an extrinsic failure: An extrinsic failure is when there is nothing wrong with the system's concepts, that could logically be implemented right; but the work wasn't done properly or wasn't completed, and the system was used at a time that it wasn't ready to be used; thus users have to face the current failures of the implementation, but these failures can be fixed and with enough efforts, at a fixed (though maybe large) cost, the implementation can be completed and repaired. An intrinsic implementation failure is when the gap between the system's promise and what it delivers is not just unfilled but unfillable; the technology to fill it doesn't exist yet, or is even impossible. At one extreme, the system could offer a button "do what I mean", and while there is a tautological adequacy of concepts between "what I mean" and what I mean, there is an essentially unfillable gap between this concept and what an implementation of it via computer system can offer: the problem is not only AI-complete, but me-complete — even with intelligent slaves at my service, I cannot escape the responsibility of living my life, of defining, communicating, refining and enacting my desires as I interact with the world, etc. Similarly, a button "make this software robust" could have a perfect conceptual fit with the developers' desire for Robustness, but no automated contraption could ever take any randomly written piece of bad software and magically rewrite it in a conceptually clean way that fixes the bugs while preserving the ultimate intent of it.

There are also degrees to which the implementation may or may not be good enough. At one end of the spectrum, the improperly implemented concepts may be illusions, but illusions that remain stable as the world moves, and enable planning and contracting between multiple parties. The abstraction provided might then provide a great benefit well worth sustaining the cost of the implementation failure: the cost of maintaining the illusion when possible, and otherwise the cost of dealing with its inevitable break-downs. These illusions are actually self-sustaining phenomena, continually justifying their existence by their benefits after costs. At the other end of the spectrum, the illusion is a lure, a waste of time and energy that distracts from more worthy endeavors; the resources sunk into trying to maintain it exponentially increase the overall suffering from the deferred but looming disaster that will occur when the illusion inevitably breaks down.

In all cases, whether these two extremes or in-between, the notion of implementation failure remains relevant: it helps understand the underlying fragility of these phenomena should the underlying infrastructure fall apart, and also identify the limits of applicability of these phenomena, the domain of validity and invalidity where the illusions are or are not worth the cost of their maintenance — and to whom (for sometimes, those who decide and benefit are not those who suffer and pay the costs). Furthermore, thinking those domains of validity may require considering the interactions with the considered systems at various scales in time and space: the interaction of one user with one machine in the few seconds or minutes of a usage session; the day to day dynamic between developers and software, between developers and other developers, between developers and users; the month to month economic incentives between users and providers; the long term market prospects as the economic situation evolves.

9.3.3 Comparing Architectures

Two pieces of software that try to do more or less the same thing can be compared with respect to many dimensions including Robustness. Now how can software architectures be compared in terms of Robustness, when they don't embody two fixed collections of programs with pairwise comparable purposes, but instead frameworks, contexts and traditions inside which to write very different programs with different delineations for software features? The effect of using those respective architectures can be compared in terms of the overall cost of software necessary to deliver given functionality with the same quality, or conversely, the amount of software of given quality that can be delivered for given cost — we'll call this latter quantity the *semantic intensity* of the code at given quality (inasmuch as it is quantifiable or at least conceptually so). The cost here can be measured in any relevant metric: dollar amounts demanded by established consultancies, size of code to write, hours of development required, years of training for developers, etc. Can this cost be expected to shrink or to grow in the long run when making one architectural choice over the other?

Different architectures will by definition divide the work to be done in different ways; it is therefore important to understand that the cost metric that matters is total cost. Indeed, it might be that one architecture divides software in smaller tasks than the other one, yet still increases costs, because there are more of these smaller tasks, that these additional tasks introduce extra communication costs between components, extra coordination costs between developers, extra rigidity that prevents adaptation when the world changes. We argued somewhere else [citation needed] that this was the case for microkernel architectures, that by needlessly dividing kernels into runtime components communicating through the microkernel with limited protocols, vastly increase runtime and development costs as well as system rigidity, at little to no benefit compared to using programming language abstractions that provide effect isolation, software modularity and type safety at compile-time rather than runtime.

On the other hand, even without changing the total size of the problem, a better architecture can improve the situation by factoring it in a way that humans can be better organized to deal with it. For instance, one software architecture (or lack thereof) might have each developer (or at least each group of developers) write all aspect of all software as generalists; a better software architecture might allow for division of labor such that parts of the software can be shared between many developers, so it only has to be reasoned about, written and debugged once; an even better software architecture might enable specialization of tasks, whereby each piece of software is further written by an efficient specialist rather than an inefficient generalist, so it is more proficient people doing the reasoning, writing and debugging of each piece of software. By constrast, a bad software architecture might add so much incidental complexity that the system is too big and too hard for any human (or any automated system) to reason about, or to simplify. Code will then accumulate that is ever harder to understand, leading to ever renewed bugs and security vulnerabilities that no one is competent to fully eliminate.

To ensure that the skills and incentives of the many cooperating developers are both aligned and relevant, then comes the question of how developers are funded and how they are kept accountable for their work. What we call "software architecture" at this point might then also include aspects that are more "social" than purely technical, such as licensing models (free software vs proprietary software vs secret software), funding models (software vendors vs service providers vs user consortia), management models (waterfall vs agile vs other models), etc. We will discuss those aspects in the next chapter 9.4, inasmuch as a Reflective Architecture has such social implications.

Note that architectural effects are only visible at a large enough scale. Writing a "hello world" test, a schoolbook exercise, or even a short program, may not exercise much of the differences between two architectures — or it may exercise superficial features of how well the libraries readily available on one architecture (and mastered by the programmer at hand) compare to those readily available on the other (given the ignorance of them by the same programmer) without telling much on the long term prospects of either architecture. To compare architecture requires thinking in terms of software large enough to require cooperation between many developers (or, then again, methods through which software may be kept small enough not to require such cooperation; and also how these two concerns can sometimes be at odds, as some methods might be detrimental in one way and beneficial in the other).

For instance, the expression problem [citation needed] challenges programming languages with the task of allowing the extension of existing programs, to either allow the handling of more data structures by the same functions, or the processing of the same data structures by more functions, or both. This challenge is only relevant because indeed programs have to evolve, sometimes to handle more cases, sometimes handle more elaborate computations on the same kinds of data. The two complementary classic approaches to solving this expression problem, ad hoc polymorphism and parametric polymorphism [citation needed], are each more or less relevant depending on which kind of evolution happens more frequently — and combining the two of them nicely is relevant when future software evolution will not predictably always be of the same one of those two kinds.

Now actual measure of large scale consequences of architectural choices big and small are not generally affordable if possible at all. Yet it is possible to think rationally about such choices to make better choices than random. What more, even "making choices at random" presupposes a random distribution, and thus a framing of the questions that matter. Framing the questions in a different way, based on a different *paradigm* [9], can lead to very different distribution of questions and of "random" answers. He who controls what questions are considered relevant may have more influence on the outcome than those who make the individual decisions based on the paradigm. And especially so without the means to make much in terms of large scale experiments, it is very important to have good theories on what are architectural questions that matter.

9.3.4 New Opportunities and Their Cost

A trivial but ultimately bad argument in favor of a reflective architecture is that this architecture only adds new ways of factoring software without removing existing traditional ways of writing the same software; therefore, goes the argument, at best the architecture will improve the situation of code by offering some better ways of writing software, and at worst it will leave the situation unchanged as the same ways can be used. In other words, because it is an extension of existing architectures, it is better.

The argument however is flawed because it neglects the overhead costs of using the architecture itself: the cost of writing all the infrastructure, the cost of sticking to the discipline of observability, the mental cost in terms of additional concepts for the developers to be aware of, the cost of dealing with many distinct additional ways of doing the same thing and mentally or programmatically bridging the impedance mismatch between the new variants of software. When all these overhead costs are taken into account, the architecture *must* actually bring noticeably better new ways of writing software, or else there is no benefit to match these costs — not to mention the costs of change. A discussion of the effect of reflection on Robustness must therefore study the consequences of actually using the reflective features, and not cop out by merely mentioning the system's potential for not using them.

On the other hand, it is also important to realize that most of the costs involved in using a reflective system are indeed fixed costs, plus maybe a very small factor overhead when developing software and when running it. Therefore, if the architecture does offer sensible benefits on *some* wide enough classes of problems, then its costs can be justified at least for software involving those problems. What more, in the case of a reflective system, these cost are further reduced (or sometimes negated) by the previously discussed performance enhancements, which makes it quite plausible that reflective features may come "for free" (or be paid for already), unlike what would be the case if random useless features had been grafted onto an existing system.

9.3.5 Separations of Meta-concerns

A reflective architecture based on first-class implementations can increase robustness by enabling well-defined separations of concerns that are not possible without them: the separations between hyper-computation and hypo-computation, between post-computation and ante-computation, between fore-computation and back-computation.

Each of these separations can indeed increase *semantic intensity.* On the one hand, they can reduce overall code complexity for a given set of features, by sharing large parts of meta-level code (i.e., hypo-computations, pre-computations and back-computations) between computations that would otherwise have to duplicate the same functionality the hard way: they can share underlying runtimes, compilers, controllers, that in traditional systems would be inlined in each of the programs and hard to impossible to separate from these programs. On the other hand, these separations enable division of labor along lines of responsibility that allow for higher specialization: the developers who write those meta-level components can be more productive writing that code than being application developers, whereas application developers can be more productive using those meta-level components than by inlining their effects by hand (for which they would make a lot of mistakes).

What more, this separation of concerns encourages the development and use of *domain specific languages* (DSLs) that serve as interfaces between those various base- and meta- levels. And each of these DSLs is an opportunity to reduce extrinsic complexity, and instead provide a language that is exactly fit to the purpose of developing the application at hand — and maintaining it as its evolves. Note that DSLs are already possible without first-class implementations, with a lot of the same benefits.

But a reflective architecture based on first-class implementations can also increase the *semantic intensity* of DSLs, i.e. both increase the benefits of and reduce their costs: On the one hand, the reflective architecture makes it possible to write generic tooling to simultaneously support all DSLs: debuggers, static analyzers, dynamic profilers, and all the usual code instrumentations that are so hard to generalize without first-class implementations. On the other hand, a reflective architecture based upon well-defined semantic relationships between first-class implementations makes it easier to reason about code, compared to not using reflection at all, or compared to using reflection without explicit such relationships.

What more, even when there is indeed an intrinsically unbridgeable semantic gap between intensional concepts that humans want to manipulate and extensional realizations of these concepts by metaprograms, a reflective architecture, by increasing the semantic intensity of computations, displaces the equilibria for these illusions from one extreme to the other: towards more useful, self-sustaining phenomena and fewer harmful, self-defeating mirages. That means that more DSLs can be written as language abstractions that serve as contracts between user and implementers, such that users can safely ignore the implementation details while implementers can affordably maintain a bridge across that unfillable gap between human intension and machine extension.

9.3.6 Robustness-enhancing features

The previous subsection was about how reflection improves semantic intensity, and how this by itself benefits Robustness. But a reflective architecture not only brings generally simpler ways to factor code; it also enables features that specifically contribute to Robustness — once you paid the entry cost of using the architecture.

A reflective architecture offers the ability to navigate the semantics of software at multiple composable levels of abstraction along the hypo/hyper dimension and to instrument code with logging, debugging, access control, etc., all *at the level of abstraction that each user cares about*, and thus with minimal overhead in terms of complexity of use. While the ability to log, debug, control access, etc., obviously exists in non-reflective architectures, it usually requires special purpose cooperation or modification of the code being instrumented, which is expensive and intrusive, and/or special tooling that is only available at one level of abstraction, that of a well-supported programming language. A reflective architecture makes this instrumentation universally available at all levels of abstraction, by requiring all the "language" implementation at each level to abide by some minimal declarative protocol from which the other abilities can be deduced in a generic way.

A reflective architecture also offers the ability to dynamically control the effects of software at multiple composable depths along the fore/back dimension, once again *at the depth of control that each user cares about.* Users can thus dynamically disconnect and reconnect computations from the services they depend on, persist sessions and multiplex of I/O (including between heterogenous kinds of terminals), access services locally or remotely, automatically restart failed computations, use scripts and checks to automate interaction with their programs including test them, etc. Universal control tools are pervasively available at all depths of control, all levels of abstraction, etc. This contrasts with the special-purpose variants of control programs available in non-reflective systems, that each only work at one level of abstraction, usually quite low-level: for instance, user interface connection, deconnection and reconnection happens and in terms of pixels for VNC or RDP, or of characters for GNU `screen` or `tmux`; what more users or developers must choose before the program is started whether the interface with be textual or graphical, mobile or "desktop", etc.; when disconnecting and reconnecting from a different computer, that interface will be either too little or too much considering the computing power of the available terminal, the communication bandwidth and latency, the resources of the developer, etc. More generally, user-directed evaluation control in non-reflective systems typically does not usefully compose, and cannot be programmatically scripted; conversely, what scripting languages do exist do not usually offer good interactive control interfaces, and still no good composition mechanism for independent software developers to combine their works: virtualization happens at the level of CPU instructions with `qemu` or VirtualBox; users of encryption and authentication must deal with binary files (GnuPG), point-to-point connections (SSH) and manual key management; various database servers complexify the programming model with a completely different, non modular, language.

More classically, a reflective architecture offers the ability to generate programs at multiple composable stages of evaluation along the ante/post dimension, as always *at the level of staging that users care about.* Users can thus dynamically issue queries that will be properly typechecked and compiled, interact with the code and modify it, partially evaluate code to specialize it to the information they are interested in, optimize code based on actually observed data patterns and user requests at runtime, use the full power of a large interactive environment while running code on small embedded devices, precompute shared computations, etc. A reflective architecture is based on the assumption that one man's compile-time is another man's runtime — sometimes the same man at different times — and does not try to artificially force a single such division onto programmers and users, unlike traditional systems.

In all these cases, a reflective architecture specifically enhances robustness by:

- Offering developers an interface at the level of abstraction they care about, rather than forcing them to think in lower-level terms than they would like, which exponentially decreases the developers's ability to reason about programs and write them without introducing bugs.
- Offering developers composable abstractions, so they can divide work in smaller chunks that can each be taken care of by a specialist, instead of forcing developers to tackle larger tasks at once in most parts of which they have limited expertise.
- Removing arbitrary barriers in the development process, whereby developers are required to make decisions overly early and cover all cases in advance when they can't know the situation of the users, whereas users are prevented from making decisions and are left without recourse if their case wasn't covered.

9.4 Social Architecture

9.4.1 Better Division of Labor

The main benefit we envision for a reflective architecture is in how it enables a different social organization of users and developers, around more modular software: indeed the new ways of factoring software that it enables not only increase semantic intensity and robustness, but do so by dividing tasks in more ways not possible without reflection. This division of labor calls for its own specialization of tasks among developers, with different interfaces between roles. The finger-grained components made possible by a reflective architecture will be the smaller units that people evolve, distribute, share or configure independently from each other (or at least with loose synchronization), thus affecting the social relationships between developers, and between developers and end-users. This social change is simultaneously the greatest potential benefit of reflection and the greatest hurdle to its adoption. It will probably take a generation for the required paradigm shift to happen in academia then in the industry.

9.4.2 Applications as Rigid Towers

Without reflection, each team of developers delivers an "application" that provides the entire semantic tower from the user-visible top to the "bottom" provided by the operating system. An application once delivered is essentially rigid monolithic block that is very hard to modify, instrument or control in any of our three reflective dimensions, at least not by regular developers, much less so by normal users.

That application is typically delivered as a single "executable" file, or a "bundle" of a many files including one or several "executable" files, "libraries", and many "data" files — or as an "installer" executable that installs a bundle on the target machine and possibly further configures it, or a "package" file for use by an existing installer. Once installed, this application embodies the entire semantics of the software, in all possible execution contexts, on top of the operating system. The operating system itself is a collection of a "kernel" that multiplexes abstractions for low-level resources, a semi-standard set of system "libraries" that can be included in the semantics of the application, some semi-standard executable "utilities" for developers and administrators to bootstrap their semantics, and some semi-standard set of background "services" that applications can interact with.

Virtualizing the "bottom" of an application and doing something with it, while possible, requires advanced system administration skills: who wishes to do that must maintain a virtual copy of an entire operating system, with suitable behavior for any imaginable request to the system; and that system is a growing collection of hundreds of haphazard "interfaces" accumulated along decades of accretion with backward compatibility as a strong constraint, even to bad design. Thus in practice virtualization is an expensive ad hoc technique used to save costs on large scale deployments; even then it consists in bulk replication of the same mostly unchanged pyramid of semantics. The complexity of it all makes it very expensive or even out of reach as a means for regular developers to achieve reflective control along the foreground/ground axis.

Developers do not have the resources to manage (much less develop and debug) the combinatorial explosion of potential behaviors that users may want to specialize, of instrumentations that users may desire to enact, or of computational effects that users may desire to control. Yet developers must explicitly encode in their applications all the combinations thereof that users will actually be able to use. They cannot afford to support more than the common cases. They have to choose sensible defaults that will work for the majority of their potential users; they may also offer a few configuration options to satisfy slightly more demanding users, but even then have to keep these options limited or become unusable due to the complexity of understanding what the options do and how they interact (yet without having access to the code). Some major applications may have a large enough user base to themselves become *platforms* that developers may extend in some way, using *plugins* or some kind of *extension language*. but unless these platforms themselves follow a reflective architecture, this only reproduces the same architectural issues, albeit at a smaller scale in more specialized more manageable setting, but also one where the skill pool is also reduced.

9.4.3 Components

With reflection, there is no more fixed bottom for software. All software by construction runs virtualized, just not at the CPU level, but instead any and every level of abstraction that users are interested in, up to the language in which the software is written (and maybe even higher if the software can e.g. be statically or dynamically analyzed). Control and instrumentation can also happen in terms of any of the above abstractions, under the control of the user, with an infinite variety of parameters (and sensible defaults), while developers do not have to care about any of the aspects that can be left to such controllers. Therefore, developers do not have to care about persistence and other "non-functional requirements" (so-called), and users do not have to be frustrated because their requirements are not served by applications. Orthogonal persistence, infinite undo, time-travel debugging, search, interactive completion, and all kinds of other services that can be implemented once as natural transformations are made available "for free" to all programs, at all levels of abstraction, from the lowliest CPU instructions to the loftiest DSLs.

Developers will thus have to write less code; their code can have higher quality and wider applicability; and users will get to enjoy software that has more functionality, and that is better adapted to their needs. But more importantly, this will happen because developers do not deliver "applications" anymore, that embody entire rigid semantic towers of software from the user interface down to a low-level "operating system". Instead, they will be delivering "components", that implement only the original aspects of the software project, while the other aspects are managed by other components connected to the component at hand via reflection.

No such component delivery platform currently exists, but the closest is probably Emacs packages [citation needed]: the editor is extensible using the programming language Emacs Lisp, and "packages" may define arbitrary new functionality, that depends on other packages, and may share state through variables, "buffers", buffer-local variables, hooks called at various points, "frames", property lists associated to various global objects, new global objects in new variables, etc. The way these components interact isn't quite reflection the way we propose: instead of defining for each component an explicit higher-level DSL that other components can interact with using meta-computations, Emacs packages all use the same somewhat lower-level language, and interact through data decorations and code hooks that roughly implement what background control might have been. Large applications such as web browser and document authoring systems also often have "plugins", "extensions" or smaller "applications" that enable an ecosystem of software components that do not each have to define an entire tower of semantics. However, these components tend to be written in languages more rigid than Emacs Lisp, and the extension points through which they can interact tend to have to be pre-determined by the main application.

9.4.4 Bricks of Semantics

If an application is an entire tower of semantics, a component is more like a single brick, or set of bricks combined with each other — the application being divided along three dimensions of metaprogramming. For instance, a given component might be in charge of file selection, and would be used by all end-user visible activities, instead of every "application" having its own file selector.

With the traditional approach, each application expend will expend the efforts of a non-expert (for most do not have file selection as their field of expertise, and neither should they have it) on this task that is decidedly not the core business of the application. Most of the time, that programmer would be a junior programmer; it is only worse if an expert at something different is wasted writing that part of the application. Of course, libraries exist that provide this functionality, but it's never exactly what you want, it is usually lagging far behind in terms of features compared to what the best applications offer, and then if the application is still used a few years later, it will have had its file selection capability frozen from many years earlier.

With a reflective architecture, user-visible activities would not include a file selector but instead request a file from their controller, that would delegate the task to the system's configured component for file selection. Which file selector is used is moved out of the responsibilities of the "application" writer into that of the system administrator who selects, installs and configures the components. A handful of worldwide experts would compete to provide the very best file selectors, that would be shared across all applications. Instead of thousands of bad file selectors being written for each of thousands of applications, a small number of file selectors will rival to be shared by the thousands of equivalent activities across the board. A much better use of talent is made possible where there was previously waste, both saving a tremendous amount of resources and achieving much higher quality.

There is of course nothing specific to file selection in the arguments above: similar circumstances apply to all kinds of behaviors, that can now be achieved using a few modular components written by experts, instead of a lot of monolithic pieces of code written by as many non-experts. Of course, for the equivalent of an application to be produced, not just the individual components, but an assembly of those components must be developed, after the components were modified or adapted to follow mutually compatible interfaces. The interfacing and assembly of components are also kinds of work that require their own expertise, and those kinds of work too will presumably also be done by experts now that a better division of labor allows for specialization of tasks. Thanks to a reflective architecture, a lot of software development can therefore go from amateur to professional. And the key to this change is the ability for some programs to define their activities in terms of requests to a background controller.

Move the following "meta is more than fancy base" subsections in some previous section.

9.4.5 Controllers are more than Servers

Requesting a file or some other value from an outside controller can be seen as the existential dual to what lambda terms are for universal quantification: whereas a λ-expression provides to its context the proof to a universal proposition, an ε-expression requires from the context a witness to an existential proposition. Thus the form $\varepsilon(x : T)P(x)$ returns an element of type T that satisfies proposition P. For instance, the type could be that of a file, and the proposition ensure that the file is a picture in a supported format, etc. Instead of a file, the user may be able to supply an arbitrary stream, such as one obtained by connecting to a network service or by running some arbitrary generator. The programmer may specify the most general abstraction that his program is ready to deal with, and not have to care about how the user comes up with data that fits the bill; this data can be handled separately by a suitable controller back-computation.

A variant of ε-expressions can explicitly pass a number of arbitrary expressions as additional parameters, to be used as "hints" by the controller when determining which witness to return: $\varepsilon(x : T)P(x)|a, b, c$. Depending on the variant of ε, the controller may also have implicit access to the lexical or dynamic environments and other components of the virtual machine state, and use them when choosing a witness; from the point of view of the fore-computation, it behaves like an oracle, that can be cooperative or adversarial or have whatever purpose of its own.

Access to more information about the fore-computation means that the controller can potentially do be configured to do a better job in a wider variety of contexts. Access to less information means that the controller has tighter security properties and cannot be compromised to do things it shouldn't do. The strictest case, where only explicitly passed parameters can be used, is akin to the traditional case of making a request to a server in a separate process. But the whole point of controllers is that they are not limited to such service calls, and may instead be specialized to the activity at hand, use and keep private information.

One may use actors [citation needed] and capabilities [citation needed] to model which activity has access to what information, which is interesting not only for security purposes, but also for architectural design. Controllers can also be modelled as actors to represent how they interact with each other, or how they internally divide their computation in sub-activities that each only cares about a subset of said information. A controller common to many computations can combine the information in all of them for various beneficial interactions (such as caching common subcomputations). A set of controllers private to each computation can ensure that each computation's state remains private. Some multiplexing controllers can ensure that some kinds of state can be shared between many computations but only in safe ways, whereas other kinds of state remain private. Therefore, as opposed to the simplest model of disjoint individual actors, the reflective actor model has the notion of actors being implementations of other actors, of actors controlling one or multiple other actors, etc. The semantics of one actor may thus include and refine the semantics of these other actors — and the inclusion relation does not define a tree, since actors can be refined along several dimensions, and controllers, implementations and generators can multiplex or delegate multiple parts of their semantic contributions to as many other actors.

All in all, background control by concurrent actors with various capabilities interacting with high-level messages offers a model to think about security in a way that is largely independent from the detail of computations. The model is useful as long as the system can affordably be divided into relatively small components that interact according to type signatures specified for the effects being controlled. And in that model, each background controller has intimate knowledge of the computations it controls in a way that traditional servers do not and cannot have; and each controller can itself be dynamically controlled and migrated, individually. These are improvements over much of traditional monolithic application architecture both in terms of expressiveness and of security.

9.4.6 Controllers are more than Effect Handlers

Another traditional concept that is similar to background controllers, but ultimately different, is effect handlers.

For instance, in Common Lisp, a program can *signal* arbitrary *condition*, and dynamically setup *handlers* for these conditions. When a condition is signalled, the closest handler is invoked, that can take arbitrary actions and conclude by aborting the computation through a non-local exit, raising the condition again to be handled by the next closest handler, restarting the computation that failed, or continuing the computation after taking suitable corrective actions. This latter option, which most other languages with exception facilities still can't decades later, makes it possible to use effect handlers to implement background control of sorts; however, in contrast to our background controllers, Common Lisp handlers do not possess a builtin mechanism to return a value; background control could still be expressed using side-effects to a special variable, and these implementation details could syntactically abstracted using macros. But then again, background control could also be implemented more directly using dynamic binding of special variables (a feature there again still missing from most other languages, decades later) to hold a (chain of) effect handlers capable of returning values without side-effect.

Other related works, in a statically typed setting, include instances of a Haskell *monad* (and particularly a "free monad") [citation needed], ore more recently, handlers for "extensible effects" [citation needed]. Using monad typeclasses in general, computations can be divided between regular "pure" computations and computations that have extra effects, as statically handled by the specific monad instance used; dynamic control can be achieved indirectly by using a particular monad instance that explicitly maintains a dynamic chain of handlers for the effects. Extensible effects behave similarly to what such a dynamic monad instance would. These most directly express the equivalent of background control.

One difference, however, between background control and such effect handlers, is that the background controller is considered distinct computation, and can be dynamically migrated while the computation keeps running and without modifying it. Thus for instance, the controller could be handling a text terminal, and be replaced by a controller handling a mobile phone, then a PC console, and the fore-computation would be none the wiser. Migration is not generally possible using traditional effect handlers. To achieve it, the handlers, in addition to having basic safe point and migration primitives, would have to preserve the full abstraction offered by the declared effects: they would have to record the entire sequence of effects while only applying the rewrites and simplifications permitted by the declared interface; they are not apply further simplifications only valid for the current controller and lose information about the declared state of the interaction — or then again they may only apply such simplifications to their own separate copy of this state.

9.4.7 Metaprograms are more than Libraries

In traditional systems, the way for multiple computations to share some behavior is to use either libraries or servers. Servers we saw above. Libraries are collections of code (functions, variables, classes, etc.) that can be statically linked into a program at the time it is compiled, or can be dynamically linked into it at the time it is started and initialized, before it runs. Importantly, the semantics of a library is imported into that of the program before it actually runs, and becomes part and parcel of said semantics from then on. You cannot unlink a library then relink a different version after the program has started running: its effects are pervasive in the entire computation and intimately affect its meaning.

With a reflective architecture, metacomputations offer a different way of sharing behavior, where the semantics of the metacomputation does not change the semantics of the computation[1]). The semantics of the base computation are included in the metacomputation, rather than the other way around as in the case of libraries. Therefore, a same computation can run with very different metacomputations, but only with nearly identical libraries. Of course, if you adjust the scope of your semantic specification to include not just some base computation but also some of its metacomputations, then indeed these metacomputations and the libraries they use tautologically become part of the semantics of your wider computation. And if the metacomputations do not make use of any meta-level features, this might be equivalent to modifying the computation to use libraries; but in general metacomputations can do more than libraries can — as long as the specified semantics for the base computation is preserved.

Because (by hypothesis) it does not change the base computation's semantics, a metacomputation can be modified at runtime — there will be no observable change as far as the specified aspects are concerned, yet may be arbitrary changes for other aspects. This is unlike libraries, that must be fixed at compile-time (for static libraries) or load-time (for dynamic libraries). Thus, if picture processing is defined in terms of background computations, picture formats can be defined or updated at runtime, and all activities will be able to use to use the new formats, even activities that were compiled and started before the new formats were defined. And unlike delegating to a server, this does not require an expensive RPC roundtrip for every operation down to drawing a pixel, but can be compiled to code just as efficient as if a library had been used — because indeed, a library will be used underneath by the back-computation, and requests to it will be properly inlined before runtime; but as far as the fore-computation is concerned, the images remain opaque objects handled through the back-computation, and they can be observed as such. The back-computation can even change the representation of the underlying objects on the fly and the fore-computation will be none the wiser: for instance, the back-computation could adopt a better compression algorithm for all pictures, or cache variants scaled at various reference resolutions.

A library can only see what is in its scope, and the arguments explicitly provided, and doesn't have access to capabilities not specifically granted to it (although in unsafe languages such as C or C++ this latter guarantee might not mean anything at all; even using safer languages such as C# or Java, actually private capabilities require taking many extra steps to secure the global class scope and restrict access to the reflection API). However, once it has access to some capabilities, the library can otherwise do pretty much anything with whatever information it has access to (within its language's specified type constraints if any). A metacomputation can

[1] This might seem paradoxical in the case of a generator ante-computation, that creates and defines the semantics of the post-computation; but actually, this is just as trivially true for ante-computations as for hypo-computations or back-computations: *given the fixed version of the computation*, its static source code and dynamic state, it does not matter which of two distinct ante-computations generated the *same* post-computation, anymore than which of two distinct hypo-computations implement it, or which two distinct back-computations control it: by very hypothesis of they generating, implementing or controlling the *same* computation, these distinctions have been erased as far as the computation's declared semantics are concerned.

see all the semantics of the base computation at all levels of scoping; however it cannot change anything about their specified semantics; it can only refine theses given semantics. It can make use of any and all of the security capabilities that the base computation possesses, but can only use them in the very same ways that the base computation will use them during its execution whichever metacomputation is being used.

A library strictly follows the control flow of the caller (inasmuch as it is specified — which may admittedly not be so much in language like Scheme with first-class continuations). A metaprogram can change the control flow of the base computation — within its semantic constraints; it can use make use of backtracking, multiple retries, concurrency or sequentialization, it can try multiple possibilities in disjoint or interfering parallel universes, it can use optimistic evaluation, or it can contrive worse-case scenarios, etc.

A library can only affect the computation when called, or as a residual side effect of having been previously called. A metaprogram can affect parts of the base computation that don't explicitly make requests to it; for instance, it can log every binding or binding modification to every variable, to e.g. achieve Omniscient Debugging (aka Time-Travel Debugging).

Metaprograms are thus entities very different from libraries, or servers, or effect handlers. None of these concepts is supposed to replace the others. Together they offer multiple distinct dimensions along which to factor code, divide labor and specialize skills.

9.4.8 3D Slicing

Factoring a computation along three previously neglected dimensions of reflection opens new way of organizing software. A reflective architecture promises simpler software, with less development effort, more code reuse, easier proofs of correctness, better defined access rights, and more performance — but also a different social organization.

For instance, a computation generating video and sound can be well separated from the software that plays it to the user. The sound generation can be a simple component, that can combine sounds at a high-level combining notes and samples, or at a low level setting amplitude levels; either way, the computation doesn't need to know anything about sound devices, user interface (translated in 60 languages), storage of samples, filesystem configuration, music licensing, remote sound devices, synchronization with a movie generator, pausing, recording, accelerated or slowed down playback, skipping, adjusting the volume, mixing sounds, filtering sounds, disconnecting and reconnecting I/O devices, multiplexing and demultiplexing, etc. All these aspects can be handled by suitable metacomputations, at runtime, by different components. What more, the sound generating computation does not have to be restarted, much less recompiled, for a change in any of those other aspects to happen. The output can be redirected from a personal mobile device to a desktop computer, to a conference room video projector, to a broadcast with a varying number of subscribees, to a multiway network conferencing session, and back, without the person writing or distributing the sound generator having to know anything about it.

This separation also enables I/O redirection and other effects without the application even having to know about it, or the developer having to prepare for it, besides respecting the observability protocol. The user can seamlessly control the I/O using a uniform interface for all I/O generators. There are no "video viewing applications", but video viewing components that work across all activities of the system. Similarly for all kinds of components, from text editing to multiparty conversation tracking, from automated translation to online shopping, from logical constraint solving to music indexing, etc. Software functionality can be developed, distributed, maintained, updated, serviced, exchanged, in components that are much finer grained than traditional "applications". Development services and software products can therefore be exchanged in finer grains, with fewer specialists of higher and narrower expertise each touching more people than with the traditional armies of generalists.

Just because components can be written at a finer grain doesn't mean that it will always be. There are a lot of reasons why large components will keep getting written, if only inertia, which is already an important reason: people will keep writing software the way they do until competition forces them out of it; large bodies of existing code will be too expensive to break down into small pieces (though some pieces may be easy to chip away) and instead will be wrapped in virtualization layers until they are made obsolete by a new generation of software redesigned from scratch. The transition to a reflective architecture, if it happens, will be incremental, and will offer backward compatibility with traditional architectures through various adapters. These adapters may actually prolong the viability of some traditional software that is made runnable in contexts in which it couldn't function before.

Finally, even though a reflective architecture may promote developing software in smaller pieces, there will always be a market for people who take many such pieces and assemble them together for end-users. Most end-users don't care to do the assembly by themselves, and don't have the skills to do it even if they did; they want to focus on whichever tasks they are good at. Therefore, Software distributors will carefully select and configure coherent sets of components, and polish the interfaces with some consistent distinguishing design. Application developers may deliver end-user devices that are locked down for security purposes. Game designers will bring together comprehensive experiences out of complete configurations of components. Software development will be less monolithic than it currently is, yet inasmuch as there is market demand for one-stop-shops in software, there will still be people providing that smooth experience to end-users.

Part V

Concluding Material

Chapter 10

Conclusion

10.1 Retrospective

10.1.1 The Take Home Points

This thesis was divided in four parts, in which the most salient points I made were as follows:

- (Part I, Chapter 2) I showed how an elementary use of Category Theory, provides a simple and useful way to *unify* the many existing models of computational semantics. I did not have to invoke any of the higher-level abstractions in the Categorical toolkit, though they remain available to who knows how to use them.

- (Part I, Chapter 3) I formalized a notion of Implementation as the opposite of (Abstract) Interpretation, itself *a partial functor* between two categories of computation. I showed how many common concepts can be expressed in this formalism using simple bicolored diagrams. my one innovative concept was a notion of *observability* that generalizes the notions of safe-points and PCLSRing.

- (Part II, Chapter 4) From the previous formalism, I used the Curry-Howard Isomorphism to extract a protocol to deal with first-class implementations *at runtime*. The key property of observability enables "climbing up" the semantic tower of a computation and thence navigating and modifying this tower, *while the computation is running*.

- (Part II, Chapter 5) I explained how the previous formalism and protocol allowed to reinterpret a lot of known techniques in Computer Science, in new productive ways, and suggest how to unify their implementations.

- (Part III, Chapter 6) I proposed a generalized notion of *Migration*, which offers a principled, general and composable approach to understanding and solving a lot of problems currently considered hard and attacked with ad hoc methods.

- (Part III, Chapter 7) I described how the Categorical concept of Natural Transformation yields a general approach to thinking about *Code Instrumentation* techniques. It brings the promise of universal code instrumentations simultaneously available for all languages at all levels of abstraction, when they are currently ad hoc tools available just for the few languages.

- (Part IV, Chapter 8) I anticipated how these new runtime reflective capabilities both require and enable a new kind of software architecture, a *reflective architecture*. A reflective architecture explicitly deals with towers of implementations, using *background controllers* along another semantic dimension of runtime reflection.

- (Part IV, Chapter 9) I argued that a reflective architecture opens new dimensions of modularity, that will positively affect how code is written, not just in terms of Performance and Features but also in terms of *Robustness*, and Social Organization.

10.1.2 A Subtly Coherent Story

This thesis went through a variety of domains, from formal methods, to programming language runtime implementation, to various metaprogramming techniques, to software architecture. There might not obviously seem to be a strong connection between these domains; and yet, each part of my thesis only makes full sense in the context of the other parts:

- My contribution to formal methods is quite modest, my use of categories being not so original a priori and quite trivial a posteriori. My main innovation is to conceptualize a generalized notion of "observability". What makes this property important is that it enables the runtime navigation through computational semantics without which the rest of the thesis would seem to be impossible magic.

- My runtime protocol by itself might look like an arbitrary mathematical construct; the formalism in the previous part makes it both possible and necessary. The fresh point of view it offers on existing phenomena might be a mildly interesting intellectual exercise; the techniques in the following parts make it relevant as a framework to think about new general solutions to previously ad hoc problems.

- My notions of Migration and Code Instrumentation in isolation might look like they are handwaving away known hard problems. The groundwork of the previous parts make them realizable approaches to building composable general solutions to these hard issues. And the following part provides a framework within which these approaches are manageable.

- My discussion of a Reflective Architecture and its consequences, on its own, might sound like abstract nonsense. The formalisms and techniques in the preceding parts give it solid foundations. And it offers them a hope of relevance in the world at large.

The interrelations between the four parts of the thesis make it hard to raise both understanding and interest when explaining any of them independently from the others; and the need to present them together raises the barrier to entry to making them known. Therefore a great many thanks and congratulations to you for being my readers!

10.1.3 Underlying Theme

The work I present makes sense in the context of Software Architecture. It matters for Programming *in the Large*, but not in the Small [citation needed]. That is, large and/or long-running projects will gain from better factoring of software into independent components that can fit nicely together yet can evolve at different paces. The same concerns are not as relevant for small and/or short-term projects not meant to survive the winter.

A lot of projects aim at long-term success. Those that ignore the principles of good software architecture will find, after the funding is secured or the product is launched that they have accumulated *technical debt* whereby bad initial architectural decisions incur constant high maintenance cost and cripple future development. Meanwhile, those that fail to even aim at long-term success will probably not find it anyway.

On the other hand, a project will not live to enjoy its good software architecture if it failed to reach the point where it can be funded and/or launch into self-funding. It is therefore important not to pay too high a price for software architecture, and accumulating technical debt is a valid strategy. Some people recommend even planning for technical bankrupcy of your code base by making "throwing one away" [citation needed] part of the schedule. Meredith Patterson [citation needed] even went so far as planning to write a prototype in one language and the product in a different enough language to ensure the first experimental codebase will not be kept longer than it makes sense.

In any case, when eventually comes the time to write a solid system that resists the test of time, then it matters to pick a good Software Architecture, one that inherently Scales to large amounts of code, written and maintained by a large number of developers, running on a large number of machines, in a large number of different configurations. *That* is when Semantics and Reflection both matter — to keep the codebase manageable, and increase its semantic intensity (i.e. reduce the amount of code and number of developers needed to achieve the desired effect, or achieving formerly unreachable feats given the same amount of code or number of developers), by offering Modularity along relevant dimensions. And that is thus when this thesis, that reconciles Semantics and Reflection, matters.

10.2 Related Works and Opportunities

TODO: Fill in sections about related works, with relevant citations, particularly for works not mentioned in previous chapters, and/or of wider relevance.

10.2.1 Formal Methods

The first part of my thesis touches the domain of Formal Methods for specifying the semantics of programs and proving their correctness.

10.2.2 Computational Reflection

The second part of my thesis touches the domain of Computational Reflection. Open Implementation, AOP, Towers of Interpreters...

10.2.3 Migration, Code Instrumentation, etc.

The third part of my thesis touches a wide variety of techniques that can be simplified or generalized using principled computional reflection: Garbage Collection, Migration, (Orthogonal) Persistence...

10.2.4 Software Architecture

The fourth and last part of my thesis touches the domain of Software Architecture. Virtualization, distribution...

10.3 Future Works

10.3.1 Challenge

The ideas presented in each part of this thesis of mine remain largely unimplemented. I see that as opportunities, and challenges. I may or may not manage to live up to all these challenges, but maybe I can convince some of you to take some of them.

Regarding to the first part of my thesis, the challenge is to complete a formalization of First-class Implementations. One obvious approach is to use Coq or some other tool for mechanized reasoning, that may already possess a model of Category Theory, in which these partial functors and their properties may be defined. Type classes could be used to express the fact that many of these properties or conjunction of properties are composable, and thereby induce a category of the implementations satisfying them. Such a formalization can then be used as meta-theory to study various compilers, runtimes and programming platforms. A yet further challenge would be to meaningfully extend the formalization to include the formalization platform itself. What useful semantic models can be developed that include reasoning as part of the development platform, and new developments as part of the reasoning? Can the loop of self-reference be closed in a way both consistent and useful? [citation needed]

Regarding the second part of my thesis, the challenge is to add First-class Implementations to your development platform. And here, "platform" can mean many things: a programming language made reflective; an integrated development environment growing more principled; an operating system aiming at greater robustness; a system shell looking for better ways to orchestrate activities; a distributed system monitor trying to reduce complexity. Here, instead of starting from pure theory and extending your platform towards more reflection, you'd start from a platform that already has some reflection out of practical necessity, and restrict this reflection to follow good principles that allow it to do more for less in a more robust way.

Regarding the third part of my thesis, the challenge is to factor your software into finer-grained reflective components. Use the three dimensions of reflection to simplify your software: ante/post computations so you can write your software in terms of domain-specific languages [citation needed]; hypo/hyper computations so you can think about software at several levels of abstraction, at each moment using universal code instrumentations at the level that matters; back/fore computations so you can dynamically control the evaluation context of your computations without corrupting the computation itself. Then you may enjoy the increased *semantic intensity*, and the simplification, robustness, security, etc., that it brings.

Regarding the fourth and last part of my thesis, the challenge is to build more efficient communities, markets, ecosystems, based on these new ways to factor software. Not just isolated programs using cool techniques, but shared platforms that leverage them into a new way to practice computing. [citation needed]

10.3.2 The Meta-Story

In the end, though I do claim to make a technical contribution with this thesis, I aim at something more ambitious: *a change of point of view about Computing.*

The point is not to dismiss the relevance of existing points of views in their respective domains, but to offer a complementary, wider point of view, that is relevant at scales where Architecture matters, for software systems that aim to survive in the long run. The reason that Semantics and Reflection had to be reconciled is because on the one hand, the schools of thought that further Computational Semantics tend to focus on puzzles in well-defined fully formalized setting — wantonly ignoring the wider mostly informal context in which their formalisms compete, pushing away any human factors that cannot be neatly squared into boxes, including the need to adapt to dynamic change in an evolving world. On the other hand, the traditions of coding that further Computational Reflection tend to focus on what is artistically expressive for the ones, or what is immediately practical for the others, usually with little respect for what makes sense, can be reasoned about, be maintained in the long run, and be kept safe and secure. Yet these two approaches are in the end complementary, and both are necessary for large projects to succeed. I consider that my ultimate contribution is on how to think these complementary aspects together, an use them in harmony, rather than continue the current multiple personality disorder of only thinking one at a time while ignoring or denying the other.

I have started a blog specifically about this change of point of view, where I purposefully avoid digging too deep into technical details as I describe software architecture in a dialogue with a Computing Practioner from an alien race, the Houyhnhnm (pronounced “Hunam”), of equine appearance: *Houyhnhnm Computing* `http://ngnghm.github.io/`

Bibliography

[1] Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer, 1999. 8.5.1

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang, 1993. 6.3.2, 7.4

[3] Alan Bawden. PCLSRing: Keeping Process State Modular. Technical report, MIT, 1989. `http://fare.tunes.org/tmp/emergent/pclsr.htm`. 3.2.5, 3.3.3

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, April 2016. 9.1.3

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977. 2.2.6

[6] Alan Dearle, Graham N. C. Kirby, and Ron Morrison. Orthogonal persistence revisited. In *Proceedings of the Second International Conference on Object Databases*, ICOODB'09, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag. 7.3.1, 7.3.5

[7] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. ITS 1.5 Reference Manual. Memo 161a, MIT AI Lab, July 1969. 3.2.5, 3.3.3

[8] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, page 348–355, New York, NY, USA, 1984. Association for Computing Machinery. 1.2.5

[9] Richard P. Gabriel. The structure of a programming language revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 195–214, New York, NY, USA, 2012. ACM. 9.3.3

[10] Kurt Gödel. Über formal unentscheidbare Sätze der *principia mathematica* und Verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. 1.2.2, 5.3.2

[11] Wolfgang Goerigk. On the Correctness of Compilers and Compiler Implementations. Technical report, 1995. `http://i44s11.info.uni-karlsruhe.de/~verifix/`. 3.1.4

[12] John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press. `http://www.cs.cmu.edu/~fp/papers/`. 3.1.4

[13] Giorgi Japaridze. Introduction to computability logic. *Annals of Pure and Applied Logic*, 123(1):1 – 99, 2003. 2.1.5, 8.5.2

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, June 1997. http://www.parc.xerox.com/spl/projects/aop/. 5.3.1

[15] Oleg Kiselyov and Chung-chieh Shan. Functional pearl: implicit configurations-or, type classes reflect the values of types. Technical Report TR-15-04, Harvard University, Division of Engineering and Applied Sciences, 2004. http://okmij.org/ftp/Haskell/tr-15-04.pdf. 1.2.6

[16] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. Technical Report SRC-015, Digital, 1988. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-015.html. 3.1.4, 3.2.5

[17] Leslie Lamport. Processes are in the Eye of the Beholder. Technical report, Digital, 1994. http://www.research.compaq.com/SRC/personal/lamport/tla/. 2.3.2

[18] Leslie Lamport. Refinement in State-Based Formalisms. Technical Report SRC-1996-001, Digital, 1996. http://www.research.compaq.com/SRC/personal/lamport/tla/. 2.2.3

[19] Jochen Liedtke. A persistent system in real use: Experiences of the first 13 years. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, 1993. 7.3.5

[20] John McCarthy. Recursive functions of symbolic expressions and their computation by machine (part I). *Communications of the ACM*, April 1960. http://www-formal.stanford.edu/jmc/recursive.html. 1.2.2

[21] Tomas Petricek. Miscomputation in software: Learning to live with errors. *The Art, Science, and Engineering of Programming*, 1(2), 2017. 7.4, 7.4.2

[22] François-René Rideau. Formalizing the notion of implementation, as illustrated with concurrent garbage collection. Rapport de recherche à paraître, France Telecom R&D, 1999. 3, 3.2.5, 3.2.6, 3.3.3

[23] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983. 5.3.4

[24] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, MIT, January 1982. https://dspace.mit.edu/bitstream/handle/1721.1/15961/08995844-MIT.pdf. 1.2.2, 1.2.5, 1.3.2

[25] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 45:161–228, 1936. http://www.abelard.org/turpap2/turpap2.htm. 1.2.1, 1.2.2, 5.2.4

[26] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, August 1998. 5.3.1

[27] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990. 6.2.1

[28] Mitchell Wand and Daniel Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. pages 298–307, August 1986. 1.2.5