La Sémantique des Systèmes Réflexifs
appliquée à la Construction de Systèmes Répartis

The Semantics of Reflective Systems
as applied to Building Distributed Systems

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

À soutenir

I know that I know not.

— Socrates

Credits

If I had to credit all the people who made this work possible, I would have to describe the contributions from millions of unsuspecting individuals who provided each of the components consumed along the way, from the ink and paper with which I printed this and other documents, to the woodcutters who harvested the trees used to make that paper, to the farmers whose crops fed the woodcutters, to the engineers who designed the bridges on which these crops travelled. Each of these contributions was arguably tiny, yet it is difficult to untangle any of them from the web of their interaction. yet all were significant enough to be necessary. Most of the contributors didn't have any good intentions regarding this thesis and actually most of them didn't even know about my existence; they were only concerned with themselves. But Leonard Read made a better case of it all than I could, in his short essay [**?**, **?**]. Therefore, rather than thanking my contributors, I'll instead begin by thanking my continuators: all the fine people who will do something out of this work of mine; all those who will be enlightened by it, hopefully positively, but also possibly in a negative reaction to it; those who will start from this thesis and take its ideas further, and those who will go yet further away from there, without knowing about my thesis, and so on. In other words, I want foremost to thank those who will keep this work alive, through a tradition of which I will be proud of having been a significant contributor.

Speaking of significant contributors, without which it couldn't be merely said that this work would be different, but that *it* wouldn't have been at all — though no doubt other works would eventually be achieved that would piecemeal cover all the original ideas I could produce if any, I can still name a few people to whom to express a particular gratitude. Firstly, I thank Jean-Bernard Stefani, who hosted me three years in the department "Architecture des Systèmes Répartis" (ASR) — Architecture of Distributed Systems — of the Direction des Techniques Logicielles (DTL) — Direction of Software Techniques — of France Télécom Recherche et Développement (formerly CNET). He helped me start this thesis, and oriented my reflections (pun intended) toward the study of distributed systems; I thank him a second time for making me the honor of presiding this jury.

Then again, I thank Elie Najm, professor at ENST, who advised me during my thesis.

I wish to thank the whole team of DTL/ASR from France Télécom R&D, then led by Jean-Bernard Stefani, who hosted me, for the warm welcome I had there.

I wish to thank all my family, extended family and friends who supported me in my difficult moments.

I also dearly thank the members of the TUNES project, for their interest, discussion, trust. I particularly thank David Manifold for the time resources he spent for me and the hardware and net connectivity he provided. Most of all, I wish to thank Basile Starynkevitch for his friendliest help when I needed it.

Abstract

Taking a dynamic point of view on software engineering, we propose a formal approach to using reflective techniques in the construction of distributed systems. In the first part of our thesis, we present a formal theory in which to describe reflective systems, and with which to discuss the interest of these systems. In the second part, we use our previously developed framework to deduce architectural properties of computing systems in general, and reflective systems in particular, for which we propose a development model. In the next part, we then show how to bootstrap from widely available development tools a self-sustaining reflective infrastructure conforming to our theoretical model. In the last part, we show how to use a reflective infrastructure to build distributed systems, with examples using our system. We hope to thus raise some interest about a development paradigm that is overly neglected and underly understood.

Résumé

Prenant un point de vue dynamique sur le génie logiciel, nous proposons une approche formelle à l'utilisation de techniques réflexives dans la construction de systèmes distribués. Dans une première partie, nous présentons une théorie formelle dans laquelle nous décrivons les systèmes réflexifs, avec laquelle nous discutons l'intérêt de ces systèmes. Dans une deuxième partie, nous utilisons le cadre précédemment développé pour déduire des propriétés architecturales des systèmes de calcul en général et des systèmes réflexifs en particulier, pour lesquels nous proposons un modèle de développement. Dans une troisième partie, nous montrons comment amorcer à partir d'outils de développement largement répandus une infrastructure réflexive auto-suffisante conforme à notre modèle théorique. Dans une dernière partie, nous montrons comment utiliser une infrastructure réflexive pour construire des systèmes répartis, avec des exemples utilisant notre système. Nous espérons susciter ainsi de l'intérêt pour un paradigme de développement trop négligé et trop peu compris.

Contents

Chapter 1

Introduction

Preliminary note: Chunk of texts grayed this way are *meta-remarks*, remarks about the text. Hence, this particular chunk of text is reflective in as much as it describes one of its properties.
Chunks in this font are notes from the author to himself, so as to indicate how "holes" in the text are meant to be filled. If you see them, then you're reading a draft of this text rather than a complete version.
Chunks of text in this police are provoking remarks rather than scientific assertions; they express but the author's personal opinion. If you see them, then you're reading an unofficial version of this text rather than an official document.
This text should be available at the following URL: ☞ `http://fare.tunes.org/tmp/`

1.1 Foreword

For scientific content, you may skip this foreword and jump directly to the problem statement. For technical content proper, skip directly to next chapter.

Retrouver cette citation de Montaigne sur la façon dont l'écriture cristallise nos vaporeuses pensées, leur donnant structure et solidité, quoique restreignant leur portée à une forme concrète limitée, là où l'abstraction laissait entrevoir une infinité d'applications potentielles. Sinon, mettre en citation le poème extrait de "Le ton beau de Marot" sur la contrainte.

This thesis is a blurry picture. It is the unsatisfactory result of a long and difficult process of trying to translate into distinct words and formal concepts the visions of a lofty dream of how computing could be better. This dream has haunted me for years past, and will continue to haunt me for years to come. Of these visions, I tried to take a snapshot; but the crystallization of my vaporous thoughts into a solid text was a longer and more difficult process than I expected, and by the time some picture was beginning to coalesce, the visions were changing, and I had to constantly chase the ghost of my dream. Hence, not only does this thesis not constitute a definitive description of an identified object, it also isn't made as a coherent picture thereof. Each of its parts might be locally consistent; the whole conveys the definite gist of a common purpose; yet the assembly of the parts is lacking in solidity. Nonetheless, it is as good a picture as I could take this far, and most importantly, I tried to make the spirit of it as explicit as I could. My ambition and hope is that after you have read this thesis, you too will be convinced that this chase was not vain, and that though the ghost may not exist to be caught, a valuable scientific

research has been led, that though these visions might never lead to a coherent picture, the elements gathered are worthwhile on their own, and that though there might not be anything to see in the given direction, at least the approach taken itself leads to an enlightening point of view on things as they that actually are.

Then comes the question of identifying the topic of this thesis. This thesis is strongly rooted in computer science, and does propose a few new concepts to clarify some questions of computer science; yet most of it isn't computer science. This thesis does attempt to open new paths in software engineering, and software engineering serves as a motivation for all the concepts discussed, yet most of it isn't software engineering. The thesis stems from a philosophical endeavour, yet any philosophical discussion in it has been actively avoided[1]. So is this a thesis in computer science? in software engineering? in philosophy? I like to think of this thesis as proposing a dynamic or "cybernetic" point of view on the way we may use computers. That is, it doesn't so much proposes new techniques to implement computer systems as it proposes new ways of understanding existing techniques, organizing them, using them, in ways that will hopefully make people more productive and lead to better future designs.

I hope these commencing remarks will help make my thesis more understandable, if not more enjoyable. I also wish at least some of my readers find some joy and enlightenment while reading this thesis.

1.2 Reflection

First, let us introduce the concept at stake in this thesis, Reflection. We will present the current state of the art, consisting of what is known about it (what it is, what it does, what it doesn't), and what is not known about it (what it means, why when and how it is useful). We will then make a statement about what we think we have contributed to the knowledge about reflection.

1.2.1 What it is

Reflection — the ability to think and discourse about oneself — is a concept is older than computer science. It has notably been used for ages in formal or informal logic systems. In the field of computing, it has been used in software engineering or artificial intelligence as a tool to simplify the implementation of difficult tasks and the experimentation of new techniques, and to allow the expression of complex systems in terms of incremental modifications of simpler ones.

Informally, reflection consists in self-reference within a system, whereby some internally accessible algebra of primitives or structures encodes ("references") an aspect (or "feature") of the external semantics of the system ("itself") in a "magic" way that couldn't otherwise be expressed by purely internal means. When this "access" is read-only it is said to be introspective; when it also serves to modify things, it is said to be intercessive.

The effects of such reflective features and their subsequent interest depend closely on the systems considered — there are about as many reflection principles as there are systems and ways to formalize them. Hence, the meaning of the term "reflection" varies a lot, and requires context-dependent explanation in every case.

1.2.2 What it does

Various forms of reflection has been used to implement such things as object systems, logic programming, parallel or distributed programming, garbage collection, interactive debugging,

[1] Though they may appear in footnotes with an explicit *caveat lector*.

persistence and other features in systems that didn't originally support such features.

More interestingly perhaps, it has been used to build *flexible* systems, that can be dynamically adapted to changing needs; adaptations can happen at compile-time or at run-time or both, depending on available reflection features. Starting from a common initial framework and specializing toward divergent goals, a same system can thus hopefully cover a wider range of uses for a given coding effort.

Finally, being able to extend a given system *from within* allows to escape escaping the costs of learning, building and maintaining a large mish-mash of many subsystems each with its own language, that interact in awkward or even counter-productive ways.

1.2.3 What it doesn't

All the above features that could be added to a system through reflection have been implemented without reflection to create integrated systems with a fixed set of feature and with well-defined semantics. If reflection can be useful in quickly exploring new variations of a given feature, each actually useful feature can be extracted afterwards and reimplemented without reflection in each useful case, often with an appreciable performance increase at runtime.

Thus, while proponents of reflection claim that it makes programs simpler, more efficient, more flexible or more powerful, and is therefore a valuable technique, opponents claim that it is not *necessary* to achieve any of the benefits it lays claims to, whereas it incurs an implementation cost that actually makes a reflective program less efficient than a non-reflective implementation of any given technique. Furthermore, opponents of reflection argue that reflection introduces a lot of complexity in the form of unneeded and uncontrollable potential interactions, making programs much more difficult to understand; they therefore conclude that reflection should be avoided, being a dangerous safety issue.

1.2.4 The Paradox of Reflection

Facing these contradictory claims, conventional wisdom seems to be that while reflection might be useful to a few mavericks for experimental exploratory systems in research labs, it is to remain confined there and not be deployed outside. Proponents of reflection are seen as chasing a ghost they cannot ever catch, any interest of reflection residing in its possibly facilitating a chase that has positive side-effects in terms of explored features.

Now, the telling of things in terms of a paradox and their resolution as a compromise is but an emotional approach — categorizing things with links of association or opposition, and laying barriers to emoting where it stumbles on situations for which it isn't fit. The rational solution to paradoxes is to replace emotion with reason, two-state evaluation with implication logic, emotional contradiction with causal structure.

Thus, before we may sort out what truth there is or isn't these mutually incompatible claims, and before we may determine how, when and why Reflection is or isn't useful, we must first build a better understanding of what reflection really is.

1.2.5 What does it mean?

To understand what Reflection is all about, it would be nice to be able to ascribe a clear meaning to Reflective techniques, and thus to have a general framework in which to express this meaning. Yet, to this date, though there exist many reflective systems or features with a well-defined semantic description, many scientific papers each describing a particular reflective technique or proposing a particular reflective framework several reviews of reflective techniques used in particular fields there exists no framework in which to describe reflective systems in

general, and what is perhaps more important, there exists no framework with which to express the concept that a given system or feature be reflective or not. What makes something *reflective* or not? While there seems to be an informal consensus, there is no formal criterion to tell. Thus, in an important way, the concept of computational reflection itself, despite being widely used, hasn't been satisfactorily formalized yet.

This thesis will try to bridge that gap by proposing a semantic framework in which to understand reflection, which is an endeavour in computer science.

1.2.6 Why is it useful?

Can the various claims and arguments of proponents and opponents of reflection be either substantiated or invalidated? Can reflection deliver on its promises of simplicity, flexibility, power and efficiency? Can it at least in some case provide valuable new compromises unavailable without reflective techniques? Can we formalize the kind of tradeoffs involved?

Reflection certainly seems to inspire a lot of faith or fascination in some people, and as much doubt or repulsion in other people. Some people have a quasi-mystical sense of reflection being a superior solution to problems, and other people vehemently dismiss this sense as mysticism indeed. But so as to leave mysticism and go beyond mere intuition, we have to provide rational tools so as to assess the utility or disutility of reflection or lack thereof, to whom and in what context. Then we can see what are substantial justifications and what are explanable delusions in these evaluations, and how to enhance our engineering processes to take this knowledge into account.

In this thesis, based on our understanding of what reflection is, we will be able to conceptualize what can make it useful, which is an endeavour in software engineering.

1.3 Cybernetics

Cybernetics is the science of interaction. It tackles such questions as "what is information?", "how is it used by information processing systems to adjust to their environment?", "how it is discovered by these systems?", "how it can be quantified?", "what principles of conservation apply to it?", etc., though it also rests on fundamental questions like "what is cognition?", "what is interaction?", "what is being?". One essential concept in Cybernetics is Feedback, or retroaction loops between the decisions taken by the system, the actions of the actuators of the system, the effect of these decisions on the environment, the modification on the information perceived by the system from the environment, and the next decisions that will be taken. Negative feedback leads to stable equilibrium. Positive feedback leads to evolution in an open system (or chaos in a closed system). Cybernetics can be used as a point of view to get insight into mechanical systems, but also biological systems or human societies[2]. There have been several schools in Cybernetics, since it was born as an autonomous science, each with a slightly different approach.

The first school focused on the principles of algorithmic information processing; importantly, its approach factors out or takes for granted the adequacy between the world in which the

[2] Indeed, historically, the origins of cybernetics are in the study of human systems, and then the study of biological systems, before it was applied and formalized by physicists. Though the term "Cybernetics" was coined in the 1950's by physicists interested in the use of computers, who formalized it as a distinct science, the principles underlying cybernetics are indeed much older. Darwinism has been widely acknowledged as an eminently cybernetic approach to biological history, predating the physicists' formalization by over one century. Moreover, Friedrich Hayek [**?**] remarks that even the biologists actually borrowed the cybernetic insight from earlier works by historians and economists of the classical liberal tradition. Hayek's own key article "The use of knowledge in society" dates from 1945.

information processing system is used and the system's representation of this world. That is, this school addresses the problems of a designer external to the system, who knows about the environment of use of the system and about the purpose to be fulfilled by the system, and who strives to build a control mechanism to adapt the system's behaviour to this purpose.

The second school ...

move this somewhere else:

Since I will refrain from developing it any further in the body of this thesis, I might as well say a few words now about the philosophical stance that motivates this attitude, least the mystery induces unrest in some philosophy-inclined reader. My stance is a quest for the intricate meanings of liberty, ethics, and information. Cybernetics could be this stance, although it would be a personal vision of cybernetics, that I like to dub "cybernethics".

Reflection is a phenomenon at the heart of this cybernethics, since our self-knowledge defines in both a positive and a negative way the things that we can do, and the way we decide what to do. When studying software systems in general and reflective systems in particular, I've always tried to consider at the same time both the software engineering point of view of the programmers, and the computer science point of view of the programs, in an intertwined way. For the programmers, computing systems are dynamic projects in the making, and the structure of the programmers' "meta"-knowledge about the system crucially impacts this making. For the programs, the meta-knowledge about themselves and their relationship to the external environment is a fundamental data which defines behaviour, hence a paradigm for the analysis and design of software that adapts to changing circumstances. Finally, development of further programs is an activity that involves both humans and computers, and it is the overall system that includes all of them and the their interactive communications that must then be considered.

Indeed, programmers are not perfectly isolated meta-systems: their knowledge is not perfectly thought out, to be dumped exactly as is into perfect computer programs; the knowledge upon which programmers base their decisions includes the knowledge interactively obtained from running computers, that are an active part of the development project. Conversely, humans can be considered as more or less trusted oracles to be used by a computing system while trying to fulfill its purposes.

Moreover, it is always possible, at a cost, to move meta-knowledge from programmers to computers or from computers to programmers: any explanation of why a programmer decided that this was a correct program to achieve such effect could ultimately be encoded and processed in a computer, whereas any computer-generated structure could ultimately be decoded and understood by a human. Hence, moving information in either direction is to be ultimately considered from the economic point of view of the expect immediate cost and long-term impact of moving the information, in systems that comprise both humans and computers. Since development systems, with which further programs are developed, are such systems that comprise both humans and computers, this economic point of view is especially relevant to software engineering.

The point of view followed during this work is rooted in a quest, if not for "Artificial Intelligence", at least for "Intelligence Augmentation" — that is, toward the use computers as intellectual props for human minds. But neither of these topics is directly touched by the thesis, though either may benefit from design principles developed herein.

is interested in building flexible heterogeneous distributed systems. "Flexible" here means that starting from a common initial framework, such diverse features as real-time quality of service (QoS), parallel and distributed evaluation, code mobility in heterogeneous environments, dynamic runtime upgrade, high-availability, dynamic admission tests, persistence, etc., may be added to the infrastructure, depending on the particular needs of applications. And hopefully, each extension the infrastructure in such ways won't require to rewrite the system and its client from scratch. The ability for infrastructure extensions to provide these features implies a that the software architecture be specially designed for flexibility; in particular, implementing some

extensions require full control of resources usually hidden from programmers, whereas hiding these resources again is essential in being able to reuse code that doesn't explicitly use these extensions.

The current generation of distributed systems, as represented by platforms conforming to the CORBA specifications from OMG, *find proper citations* do not provide such required flexibility. There are many research efforts toward flexible distributed system, based on adaptable kernels such as Spring, Spin, Choices, X-kernel, etc. *find proper citations* But in most cases, the flexibility obtained is limited to some parts of the system, according to a system-imposed model, without a systematic framework to design such systems, or for developers to define models specific to their application domain.

A promiseful approach to achieving such a framework would be to use the power of reflective techniques. Reflection in general consists in making details of the system accessible and/or modifiable by user programs that are usually hidden from programs and taken into account manually by the programmer. Run-time reflection consists in the system dynamically maintaining a faithful representation of itself that allows for run-time operations that coherently use whole-system data and propagate modifications to the whole system. Compile-time reflection consists in the system being defined using user-defined languages or language extensions (as with LISP macros, for example).

Up to now, more or less powerful variants of compile-time and run-time structures have been implemented in more or less mainstream dialects of programming languages such as LISP, Smalltalk, C++ or Java; also some Operating Systems have made good use of some reflective techniques (Genera, Apertos, Merlin, Coda, Synthetics). But these existing implementations use such techniques in an ad-hoc way, more as "magic" features than as part of a general theoretical framework. In the case of some implementations, an extensive lore about using such techniques exists [**?**], whereas in the case of some techniques, an extensive scientific literature exists. *find proper citations on partial evaluation, metaprogramming, reflection, etc.* But up to now, implementation and solid science haven't met in a common system; there is still a missing link. And this link is all the more needed so as to use these techniques in a dynamic concurrent system, where run-time and compile-time are blurred somehow, and some techniques do not apply, or raise difficult problems.

To tame the power of reflective techniques, particularly so as to apply them to flexible distributed systems, there still lacks a bit of theory and a bit of implementation. The goal of this thesis is to make a contribution in filling this gap.

1.4 State of the Art

Systems without reflection. Ad-hoc metaprogramming. Interception. Domain-specific languages. Ad-hoc reflective models.

Reflection in some systems: distributed control (ABCL/Rn, Tube). AI (Pitrat).

Theory of reflection: about nothing.

Generic system: Erlang. JOCaml. Have a language that matches the abstraction of distributed systems. Problem: don't match specific

Specific systems: build domain specific languages that closely match the problem set. These languages can evaluate efficiently without extraordinary compiler mastership from the domain expert thanks to partial evaluation and/or dynamic compilation, two techniques that can statically or dynamically transform interpreted expertise into efficient compiled expertise.

Our project is to generalize these results by building a generic metaprogramming framework.

1.5 Our Approach

Solving the paradox: changing the point of view.

Our contribution to the problem of taming reflective techniques for dynamic concurrent systems is at the nexus of theory and practice, of computer science and software engineering.

- Establish a formal model for the semantics of reflective systems.
- Define elements of programming methodology for reflective systems.
- Give an experimental implementation of the model.
- Build a new flexible distributed execution platform on top of it.

Metaprogramming taxonomy necessity? Feature...

1.6 Contribution

State our thesis as such

Dynamics of code development rather than in the static requirements of any given program

1.7 Perspectives

Where to go from now?

1.8 Plan

Plan of the dissertation

1.9 Note on Formalisms Used

In the body of this thesis, we will use several formalisms that deserve clarification. These formalisms do not depend on any conceptual novelty; however, they are not so mainstream as to have universally known and agreed upon conventions and notations, though standard such conventions and notations may be recognized within specific communities. Thus, we had to remind our readers of the concepts involved, and define the conventions and notations that we used. We did so in appendix A. The formalisms we used are (a small subset of) Category Theory, partial and non-deterministic functions, and a graphical formalism for expressing simple assertions in the previous two formalisms.

Part I

A Semantic Framework for Computational Reflection

Split this chapter in two?

Reflection: making more sense of the world by better knowing oneself. And indeed, to make sense out of the world is to be able to interact with it in ways that bring personal subjective satisfactions.

When words are unfit, discourses are empty, and actions are vain.
— Confucius

Move the following paragraphs to the general introduction.

There exists no formal framework for reflection in general in the context of computer science, and the term "reflection" is used by commercial vendors and academics to describe a variety of features that span from the simple availability of dynamically typed access to data in an otherwise statically typed language *cite C++*, to the existence of elaborate tower of metaprogramming levels in a proof system, from concrete running code *cite NuPRL* to ethereal philosophical musings *cite BCSmith-OriginsObjects*.

We hereby propose a theoretical framework that allows the elucidation of key notions that hide behind this vague word "reflection". We most notably formalize notions of *computing system*, *implementation*, *meta-system*, *representation*. *PLAN du chapitre...* Thence, we formalize the power and limits of various currently existing "reflective" systems and techniques, and we study what reflection as we conceive it can really bring to software engineering.

Finally, we discuss a model for a new kind of "reflective" system that purports to be universal within the former framework, and to enable all the previously studied reflective techniques within a clean formalism.

Have a more detailed plan; a guide to the chapter, even.

Chapter 2

Modeling Computing Systems

2.1 Intent

We aim at providing as generic a meta-level framework as possible to internally express the notion of a computing system, the properties of such systems, the relationship between several systems, the combinations of systems, implementation of a higher-level "abstract" computing system by means of a lower-level "concrete" computing system. Our framework attempts to provide insight in programming from several points of view, which covers concurrent and distributed programming as well as programming using multiple different abstraction levels.

By "computing systems", we mean anything from abstract systems like Turing-machines (universal or not) and variations or λ-calculi or π-calculi to concrete systems like binary executable code or semi-conductor devices, from universal systems like specific configurations of PC installed with GNU/Linux, down to the most special-purpose devices like router switches. Note that when formally discussing about systems, there's always some abstraction involved — even when considering masks for engraving silicon, any discussion supposes some abstract behavioral model to describe the semantics of the resulting system. Similarly, the only way we can specify special-purpose devices is by expressing their semantics into a more general, universal formalism.

We choose to use a notion of computing system that is as simple and general as possible, yet fulfills our goal of bringing insight into meta-level architectures. It can be specialized to fit existing formalisms, yet allow to relate, combine together and reuse systems originally devised in different formalisms. We have not implemented this formalism in any computerized proof assistant. However, we do not depend on any particular meta-formalism (computerized or not), and think the presented framework should pose no theorical problem being expressed in any existing computerized formalism.

2.2 Categorical Approach

We use a very simple formalization of a computing system as a category C whose objects (nodes) are the possible states of the system, and whose morphisms (arrows) are valid labelled transitions between states of the system[1].

For instance, the global state of an actual computer, would be the data of the state of each of its registers, memory banks, and peripherals; more abstract systems could be described in

[1] We give in appendix A a short but sufficient description of the formalisms we use, including the little we borrow from Category Theory.

terms of a data heap, control stack, and program code; or in terms of environment, store, and continuation; or in terms of whatever concepts suit the system at hand. The very purpose of this abstract formalization will be to describe mappings between completely different systems, whose states may be described in completely different ways.

So as to avoid confusion with other meanings of the word "object" in computer science (some of which we'll use later on), we will herein use the layman terms "node" and "arrow" instead of the more common respective terms "object" and "morphism" used by category theorist. We still call "functor" an application from one category to another that preserves the composition of arrows.

A common simplifying point of view is to identify every node with the trivial arrow corresponding to the identity morphism for considered object. This allows to only state many definitions, theorems and properties once, for arrows, in which case they naturally extend to nodes, too, instead of having to state them twice (or more), for each valid combination of some supposed parameters being node or arrow.

2.3 Operational Semantics

There are various possible refinements to this formalization, with which it is possible to retrieve well-known paradigms for expressing the operational semantics of computer systems:

- *Operational semantics*: nodes are the data of a system's state; arrows are operational transitions from past state to future states. A class of arrows that generates the category (but might not itself constitute a very structured subcategory) can be distinguished as atomic steps, so as to obtain small-step semantics. Big-step semantics skip intermediate small-steps and isolate those arrows interesting for observation. *find proper citation*

- *Rewrite systems*: the nodes are terms, and arrows are rewrites from term to term. Normal forms are terminal objects in the category. Closely matches small-step operational semantics. *find proper citation*

- *Abstract State Machines*: they are a particular case of operational semantics, where system state and transitions are described in finitary terms of variable to value associations. *find proper citation*

- *Labelled transition systems*: labels can be expressed as a labelling functor from C to a monoid M (category with only one node, but many arrows). The subcategory of transitions with null label can be distinguished as that of "internal" computations, without input/output. *find proper citation*

- *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. *find proper citation*

- *Modal Logic*: nodes can be considered as state of knowledge of the system, and arrows as modal events by which this knowledge evolves. Temporal logic quantifiers can be used to discuss possible futures of a state. Linear logic can model non-monotonous knowledge evolution. The subset of arrows with trivial modality represent internal manipulation of knowledge (deduction, etc.) without gain (or loss) of information.

- Hidden Algebra *?? Goguen et al.*

2.4 Denotational Semantics

Denotational semantics can be deduced from operational semantics in the usual way. Here is a summary of this mostly well-known process, adapted to our general framework. *it subsumes similar processes for other less generic frameworks as described in [?]* Starting from an algebra of syntactic terms with an operational semantics as above, we can define the following intermediate concepts, until we can associate to each term a meaning and to each syntactic combinator from term to terms a corresponding semantic combinator from meanings to meanings.

- Suppose we have an algebra of syntactic terms to which to attach a denotational semantics. This algebra is called the source language and its terms source code.
- Suppose this source language is possibly extended with a few constructs, so as to be able to build a bigger algebra of ground terms. For instance, a ground term might include such things as as an environment and a continuation, together with some source code, which are not directly encodable in the source language.
- Suppose that we have an operational semantics for some subset of ground terms, called valid ground terms, or running programs.
- Suppose that we have defined a notion of ground observation that associates to each valid ground term a unique semantic object, called the answer of the running program. (see below for how we can do it within our framework.)
- Syntactic Combinators: a n-ary syntactic combinator from a n-uple of syntactic algebras $A_1, \ldots, A_n$ to a syntactic algebra A is any term made from the same syntactic constructors as those of A plus special nullary hole constructors $\Box_1 \ldots \Box_n$ that each must be used exactly once. Applying a syntactic combinator k to a n-uple of terms $(t_1, \ldots, t_n)$ in $A_1 \times \ldots \times A_n$ consists in replacing in Γ each hole $\Box_i$ with the corresponding term t_i, yielding a term $kt_1 \ldots t_n$ in A iff the result of this replacement is a valid term in A.
- Contexts: A context is a unary (1-ary) syntactic combinator from source programs to valid ground terms. Γ is said to be a valid context for t iff Γt is a well-defined valid ground term.
- Observational equivalence: two bits of source code t and t' are equivalent iff for any context Γ, Γt and $\Gamma t'$ are equally valid and their ground observation leads to the same answers. Another way to look at it is that we can always exchange t and t' in any context that yields a valid program, and never find a way to observe a difference by running the program.
- Denotation $[\![t]\!]$ of a source code term t: the denotation of a source code term t is the application that to a any given valid context Γ for t associates the answer obtained from the operational semantics of Γt, By straightforward use of the definitions, two source code terms have the same denotation iff they are observationally equivalent. Note that instead of an application from a selected set of contexts to answers, we can equivalently consider a partial function from the set of all contexts to answers that is only defined for some contexts; if answers are actually non-empty sets of basic answers, we can consider instead relations between contexts and basic answers.
- Syntactic invalidity vs semantic errors: Note that in the above definition, the fact that $[\![t]\!]$ be undefined for Γ is notably different of $[\![t]\!]$ yielding for Γ any answer of error or divergence that the operational semantics may define. Often, terms can be grouped in

types or annotated with types terms, such that all terms in a given type have the same set of valid contexts, and types are conserved by the operational semantics.

- Denotational meaning of syntactic combinators: given a syntactic combinator k, for terms $t_1,\ t_2,\ \ldots\ t_n$, the denotation of term $k\ t_1\ t_2\ \ldots\ t_n$ only depends on the respective denotations of terms $t_1,\ t_2,\ \ldots\ t_n$; this stems from the above-mentionned fact that if $[\![t_i]\!] = [\![t'_i]\!]$ then we can substitute t_i by t'_i in any context, including the contexts obtained by using k and other terms among the t_j and t'_j of same denotations. Thus, the function that to $(t_1,\ t_2,\ \ldots\ t_n)$ associates $[\![k\ t_1\ t_2\ \ldots\ t_n]\!]$ can be factored through the quotient structure of terms up to observational equivalence.

- We can thus define the denotational semantics of arbitrary syntactic combinators in the algebra of terms as functions from and to the semantic domain of terms up to observational equivalence.

- Such semantics has referential transparency by construction: the meaning of a term t is context-independent (precisely because the context was abstracted from it), so any term can be replaced by any other term of same denotational semantics in any context, and yield the same ground observations.

- Note: such a denotational semantics deduced from operational semantics can be said to be pure iff there is a "canonical" mapping from source terms to ground terms, so that denotational semantics among source terms matches equivalence of their operational semantics as ground terms. This is a property associated to the data of the extension of the source algebra into ground terms and its operational semantics, and not of the denotational semantics in isolation, since it is very well imaginable a priori that an isomorphic denotational semantics could be deduced from a same grammar extension with a weaker operational semantics, or from a weaker grammar extension where not all terms are ground terms.

There remains to describe the notion of ground observation for ground terms. For this purpose, ground terms are identified to nodes in an Operational Category.

- History Pre-order: given two arrows f and g, $f < g$ iff there exists h such that $fh = g$; in other words, iff g is a future for f. We can similarly define a history pre-order on the action category.

- Future: the future of an arrow f is the set of all arrows g such that $f < g$.

- Basic Observations: we suppose that we have a functor from the operational category to a more abstract Action Category, that represents observable interactions that can be done with the system. Only properties directly or indirectly observable through interaction matter.

- Interaction Theory: supposing we have a logical theory of the Action Category, we can extend it into a theory of the Operational Category, by adding suitable syntactic and semantic constructs to express the history preorder and the action functor. Usually, this is a first order theory with equality for the arrows in the action category; it can be stronger (e.g. second order theory), or weaker (e.g. observation through algorithmic filters only, or probabilistic variants thereof).

- Operational Equivalence: two arrows are said to be operationally equivalent iff their respective futures have the same interaction theory; i.e. iff it isn't possible to distinguish them by further observation of their future behaviour.

- Ground Observation: to any arrow, we can associate the logical interaction theory of its future, that embodies the observations that can be done from this arrow. For ground terms, this gives us a satisfactory ground observation so as to define Denotational Semantics as explained above. Two arrows have the same ground observation theory if and only if they are operationally equivalent.

- Operational Structure: if interaction theories have a structure, then a structure of the same kind can also be found on the arrows considered up to operational equivalence, which form a sub-structure thereof. For instance, intuitionnistic truth values are ordered by implication, probabilistic observations can be ordered variously, etc.

- Termination vs Non-Termination: there is no reason a priori why termination would be directly observable in an interaction theory; in presence of structural rewrites, actual termination of the operational semantics (lack of further rewrite rules) is not even a good criterion to convey successful termination of a computation, at a higher level. If it is important to observe termination in a system, it is much better, and much more in accordance with the practice of computer implementation, to express termination explicitly within the system as special observable transitions from potentially terminal states to states of actual termination.

The problems that one will try to solve will usually be to identify the denotational semantics induced by the operational semantics with another denotational semantics obtained by other ways. For a given operational semantics, several slightly different denotational semantics can be obtained, and obtaining the expected one sometimes involves playing with the action category with which observations are done and the richness of the logical or computational structure of observations expressible over this action category, so as to semantically distinguish more or less ground terms, or with the grammar generating the algebra of terms so as to syntactically distinguish more or less terms and contexts.

Nota Bene : It is also usually possible to go from Denotational Semantics to Operational Semantics: using CPS transformation and other transformations into monadic combinators, embed the calculus into some variant of the π-calculus, where invoking a function is expressed by communicating to a process the function parameters together with a continuation and other environment information. Then syntactic combinations are replaced with some kind of semantic combinations making processes communicate run in parallel, which can be well-defined such that operational equivalence of the processes is the same as denotational equivalence of the original terms.

2.5 Internal and External State

It is essential in our definition that the considered systems are closed worlds, as far as matters to their respective semantics, so that any relevant input or output is formalized into the system. The formalization of input and output (I/O) can be done by having (or adding) proper information either in the set of nodes of the category or in the set of arrows between each pair of nodes. Nodes can be seen as expressing the internal state of the system, whereas the choice of arrows between two nodes then expresses the possible computation paths that lead from one state to the other. Each of these paths may or may not involve interactions with the external world, as encoded by corresponding arrow.

Often, to separate internal state from external state, we consider together with the computing system category an "action category". The action category is a simpler category whose arrows embody the possible observable interactions between the system and the external world;

we relate the computing system to this action category with a functor from the system to this category that "labels" arrows with a visible action in a way that preserves arrow composition. Often the action category is a monoid, that is, a category with only one node (where two arrows can always be composed). Given two systems each endowed with its action category, a functor from one to the other will have to also be accompanied by a functor from the action category of the former to that of the latter, that makes the obvious diagram commute. For such purposes, we can identify a category without action category to a category whose action category is itself in a trivial way.

Also, many of the properties below, and the constructions that depend on them, are expressed much simpler when this internalization is done through enriching the node-set rather than the arrow-set, that is, if the state involved is internal to the system rather than external. For instance, the state of the keyboard, frame-buffer, and any I/O device relevant to the specification of the implementation, can be encoded either by extending the internal state (node-set) with enough registers to encode the device state, or by enriching the external state (arrow-set) with enough transitions to encode communication with the device controller[2].

2.6 Combining Computing Systems

It is useful to study algebraic operations to obtain computing systems from existing ones, so they that can be used either as synthesis tools or as analysis tools. Here are a few common ones.

Subsystems and Supersystems

It is often useful to consider cases where a system is included into another. The included system is then called the subsystem, and the including system is called the supersystem.

For instance, it is possible to model the fact of picking a specific program written in a generic language, or a specific set of programs run in an operating system, a specific configuration in an actual computer system, by considering the subsystem obtained by choosing a subset of the possible states for the system, and considering only nodes and morphisms reachable from these states — that is, the possible future states of the system reachable by computing from these initial states.

Hypotheses about the execution environment can be modeled by restricting the arrows in a supersystem (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 subsystems will be used, but once again, this formalization is voluntarily large, so as to allow to express mappings between computing systems of various different abstraction levels.

[2]Note that it is trivial to implement the latter (system with extended arrow-set) with the former (system with extended node-set), with the below definition of "implementation", so that extending the node-set can rightly be considered as lower-level as extending the arrow-set.

Formally, given a computing system C as a category, a subsystem of C is the data of a computing system S and a "canonical" embedding (i.e. injective functor) j from S into C. A subsystem is "full" if the embedding is "full" in the usual categorical meaning: all the arrows between two reached nodes are also reached. A full subsystem is characterized by the set of nodes it contains.

Note that whether adding arrows to a system to achieve a supersystem, or removing arrows from a system to achieve a subsystem, it is not just individual arrows, but a coherent set of arrows that must be added or removed, so that the resulting set does indeed constitute a category. A proper way to do that is to consider generators of the systems 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.

Concurrent Systems

A family of computing systems can be made to run in parallel. When the systems run independently, the resulting system consists in the usual cartesian product of the family of systems, considered as categories. *cite MEIJE ?*

Expressing interaction between systems running in parallel can be expressed in two complementary ways. In the first approach to expressing interaction, we start from systems 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 systems. Arrows that do not involve such communication are called local transitions or communication-free arrows. Arrows added to the system are non-local transitions or communication arrows. In the second approach, we start from systems 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. The cartesian product is called the free system, and its arrows that are removed are the desynchronized transitions; and the restricted system is called the constrained system, and its arrows the synchronized transitions.

A concurrent system is a system thus achieved by modifying the arrow set of a cartesian product. The systems in the cartesian product are the components or processes.

Note that being a concurrent system is not an intrinsic property of the system 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 system could be butchered into components in arbitrarily many ways, with ad-hoc arrows being added or removed to fit the desired system. Most such decompositions into components are useless; sometimes, several different decompositions of the same system as a concurrent system can be useful each to prove an appropriate property. We'll see examples of that in sections 3.6 and 4.6 below. As says Lamport, processes are in the eye of the beholder [**?**].

Chapter 3

Implementation

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

3.1 Intent

An implementation is a correspondance between two computing systems, one being a higher-level "abstract" computing system and the other being a lower-level "concrete" computing system; it is then said that the latter implements the former. This correspondance somehow preserves the observational semantics of the abstract system: computations done by the concrete system lead to observations that are consistent with computations that would have been done by the abstract system.

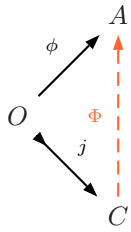
Typically, abstract systems of interest are modern high-level programming languages; for instance, dialects of LISP, ML, Prolog, Smalltalk, Java or C. "Language" here must be understood in a large meaning that includes a language core, but also a suitable set of extensions, libraries and operating-system-dependent features, as needed to specify the behaviour of programs with the precision required for the application at hand. More specific abstract systems of interest are programs and configurations using these languages, that, as seen above, can be considered as subsystems of the larger systems defined by these languages. (Writing such programs is the very purpose of programming languages!) i*DUH!*

Concrete systems of interest are modern microprocessor-based computers, or pools thereof, as programmed in assembly language. Often, these systems are too low-level for a problem at hand, they include too many details that are mostly irrelevant to the problem to solve. Thus, an implementation is often split into layers, factoring the correspondance through intermediate computing systems such as virtual machines or widely available low-level programming languages such as C. Each intermediate layer serves as a pivot so as to implement in a simpler way the layers above with the layers below.

3.2 Definition

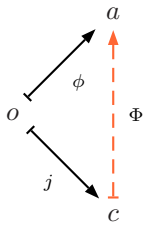
Given a computing system A, and a computing system C, an implementation of A by C is the data of an application ϕ from a subsystem O of C (as defined by its canonical full embedding j) to A. Another way of putting it is to consider the partial function $\Phi = \phi \circ j^{-1}$ from C to A, and its inverse, the non-deterministic function $I = \Phi^{-1} = j \circ \phi^{-1}$ from A to C. All these ways

can be visualized in the following diagram[1]:



With respect to this relation of implementation, we say that A is the *abstract system*, and that its nodes and arrows are *abstract*; we say that C is the *concrete system* and that its nodes and arrows are *concrete*; we say that O is the system of *concrete observable states*, and we usually identify elements of O with j-corresponding elements of C. We call ϕ the *interpretation function*, and j the *canonical injection* from O into C; we only speak about ϕ and j when specifically needed in formulas.

Given an arrow (resp. node) o in O, and given $c = j(o)$ and $a = \phi(o)$, we say that the concrete arrow (resp. node) c is *observable*; we often identify c and o, and say that a is the abstract arrow (resp. node) corresponding to c (or o), and that c (or o) is *an implementation* of a.



If a concrete arrow (resp. node) c' is not in the image-set of j, then we say that it is not observable. If there are arrows c and c'' such that for $x = cc'c''$, x is observable, we say that c' is an *intermediate computation* (of x).

Note that for the sake of being compatible with usual vocabulary, as well as of respecting our original intent, it is I that we call the *implementation* (of A with C through O), even though we'll soon see that it is its inverse Φ (actually ϕ and the implicit j) that will have the nicer structure-preservation properties. We call ϕ the *abstract interpretation* of O with A, and by extension, we call Φ the *partial abstract interpretation* of C with A. *See relationship with abstract interpretation below...* For these reason, we often call Φ^{-1} our implementations, insisting on the fact that we're actually interested in a structure-preserving partial function Φ. We extend the name of a function to apply to the image of an arrow by said function: if $a = \phi(o)$ and $c = j(o)$, we say that c is an implementation of a or that a is the abstract interpretation of c.

3.3 System-specific Constraints

It is important to understand that, when specifying an implementation, the mapping between the abstract system's observable behaviour and the concrete system's is tightly constrained by external factors that cannot always be formalized, at least not without tremedous difficulty, their origin being informal in nature. For instance, the fact that the terminal I/O primitives of a

[1] We remind our gentle reader that the basic notions on non-deterministic functions, as well as the graphical formalism we use are described in appendix A.

programming language should be implemented in terms of electronic-level behaviour that leads to conventional characters being displayed on the computer's console is a constraint strongly tied to very *intent* of said primitives. It would be very easy to build an implementation that verified all (extensional) formal constraints, but failed to encapsulate anything useful, because it wouldn't embody the intentional constraints.

In practice, this matching of intentional behaviour from abstract system to concrete system is specified with lots of ad-hoc axioms wherever they are needed; many of these axioms can be derived from formal and informal specifications and documentation about the source system being implemented and the target system being used for implementation. Particularly when computer-generated implementations are concerned (see Compilation below 4.1), any informal intent will have been over-specified in terms of such ad-hoc axioms enforcing specific implementation choices, matching high-level primitives with proper low-level behaviour such as calls to standard library functions.

[**?**]

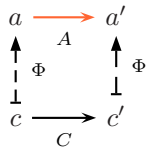
3.4 Properties of Implementations

Now that we have laid out the basic framework for studying the notion of an implementation, we can formalize good properties that we expect from an implementation so it be considered useful. Of course, depending on our (informal) intent for a given implementation, we'll require a different set of properties to be fulfilled by this implementation. In the rest of this subsection, we'll consider as defined above an implementation Φ^{-1} of an abstract computing system A with a concrete system C with system of observable states O, given by $\Phi = \phi \circ j^{-1}$ with the canonical injection $j : O \longrightarrow C$, and an interpretation function $\phi : O \longrightarrow A$.

Safety

The most essential property, that we will require from just every implementation so it be considered correct, is that of *safety*: any observable result that be yielded by concrete evaluation must correspond to a valid abstract result that could legally have been obtained by evaluation in the abstract system.

This corresponds to the following diagram:



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.

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

Let us remark the fact that structure preservation happens in a way contravariant to that of Φ^{-1}: the "interpretation function" ϕ that preserves structure goes in a direction opposite to that of the implementation 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 subsystem, it is given that j be structure-preserving.

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

To illustrate the difference between safe and unsafe, 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 safe if it traps on overflow; any result it will yield will be correct. An implementation with fixed-precision integers that silent wraps computations that overflow is not safe (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 safety does not mandate termination. It only mandates that in case of termination or legal observation, the implementation yield a correct result. It is always safe (according to this definition) for an implementation to not answer; of course, answers are desirable when possible, but misleading incorrect answers are worse than no answer. When designing mission-critical systems, it is safe (according to this definition) not to come with a design, but unsafe to come with a design that might erroneously kill people under intended use conditions.

This notion of safety is the same as found in works by Lamport [**?**]. It is what Goerigk calls partial correctness [**?**], "partial" corresponding to the fact that Φ be a partial function rather than a total function.

Safety is a composable property: if Φ^{-1} is a safe implementation of A with B and Ψ^{-1} is a safe implementation of B with C, then $\Phi^{-1} \circ \Psi^{-1}$ is a safe implementation of A with C.

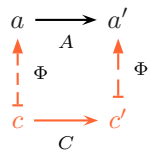
The computational content of safety is that we can use (partial) operational execution of the concrete computing system as a valid substitution (modulo translation via Φ) to (partial) operational execution of the abstract computing system — which is precisely what implementations are all about.

Since safety is such an essential property, and since it fits very naturally the categorical framework that we use, from now on, we'll only consider safe implementations — there is never a good reason to consider implementations that might give wrong answers, anyway.

Completeness

An interesting property that we may require from an implementation, is that of *completeness*, whereby the abstract system is completely implemented by the underlying concrete system.

Formally, a weak version of this property can be summarized by saying that the application ϕ be surjective, which can be expressed by the following diagram:



This (Weak) Completeness is a composable property. Moreover, many other properties of implementations are preserved when composed with (weakly) complete implementations. Completeness is never a property of implementations of infinite abstract systems with finite concrete systems (by a simple counting argument). However, it is a useful property to have in most levels of a tower of successive implementations, so as to isolate the difficult problems relative to implementation properties to a few tricky levels.

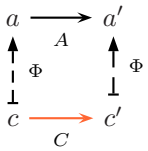
Completeness as is is mostly used in conjunction with other properties; its computational content is then that for any finite abstract computation, an implementation of it can be found in the concrete system. It can also be used as an argument to assess that the systems that it interacts with will have to be prepared for a large set of behaviours, and thus fulfill strong invariants. A variation of completeness with a stronger operational content is "strong completeness", as defined below.

Variations include:

- *Totality* — being able to implement all nodes, but not forcibly all arrows. In other words, ϕ is surjective on nodes, as per the following diagram:

  This variation is notably interesting when implementing non-computable abstract systems with computable ones. Totality is a composable property.

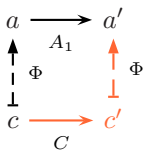
- *Fullness* — being able to implement all arrows between implementable nodes. This means that ϕ is a full functor, and corresponds to the following diagram:



  This variation is notably interesting proving properties of the implemention of a bigger system (e.g. programming language) that will induce complete implementations of a variety smaller subsystems (e.g. specific programs in the previous system). Fullness is a composable property.

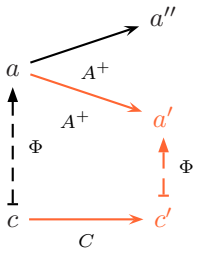
- *Strong Completeness* The conjunction of fullness and totality is a useful stronger variation on completeness that allows to filter out those complete implementations that make "early choices" and lose completeness after choosing the initial state: a weakly complete implementation could choose in advance a maximum number of steps, or some of the non-deterministic choices to be made, whereas a strongly complete implementation must preserve the ability to implement all pending abstract arrows from any observable concrete state.

- *Local Completeness* — being able to implement all "atomic" arrows from an implemented node, assuming there is a notion of atomicity in A (i.e. a generating subset A_1 of A), as per the following diagram:



This variation is helpful for implementing single-steppers. It isn't composable.

- *Advance Preservation* — being able to implement at least one "advancing" arrow from any implemented node, assuming there is a notion of advancement in A (i.e. a subset A^+ of A) and that abstract node has at least one outgoing arrow.

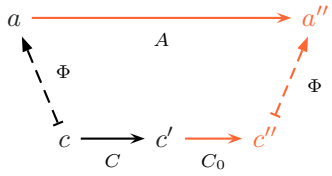


Note that in the above diagram, a'' is a priori unrelated to a' — that is, if the system can advance, it will advance, but if there is any choice involved, it may be following a path completely different from any particular path one might desire. This variation is helpful when implementing non-deterministic systems. It isn't composable.

Soundness

Another important property of implementations is that of *soundness*. An implementation is sound if and only if any possible concrete transition path is somehow "meaningful", and corresponds to part of an abstract transition path[2].

This can be formalized by the following diagram, where C_0 is a suitably specified subsystem of C that abstrains from any high-level I/O:



That is, if concrete computations starting from a stable state c have led to intermediate state c', then they must lead from c' to a further stable state c'' without doing any unnecessary I/O. In other words, the concrete system mustn't have unsound meaningless transitions, whereby it would spontaneously and unrecoverably go wild or enter a deadlock, or require the external world to do special magic I/O so as to stabilize. Instead, starting from a stable state, it must always keep the possibility of evolving into a stable state, of "stabilizing", in a way that doesn't require the special knowledge by the external world.

Soundness as such is not a composable property, since given a sound implementations Φ^{-1} of A with C and Ψ^{-1} of C with E, we cannot know for sure without an additional hypothesis that the observable state in C that we obtain from invoking the soundness of Ψ^{-1} is itself a stable state with respect to Φ^{-1}. However, there are several ways to obtain some degree of composability for soundness results, by invoking additional properties. For instance, composition of a sound implementation of A with C with a sound and complete implementation of C with E.

[2]It so happens that logicians tend to call "soundness" the property that we above called "safety" [**?**]. To avoid confusion, it might have been better to use another name for the present property. The names "metastability" and "stabilizability", albeit long and hard-sounding, have been proposed, since the property ensures that an external monitoring "meta-object" may stabilize the state of the system so that concurrent internal activities will only make safe observations. We are not otherwise aware of this property having been given a name before our 1999 article [**?**].

Also, if Φ^{-1} is an implementation of A with C through O, and Ψ^{-1} is a sound implementation of C with E such that all image values of Ψ in C are also in O, then $\Psi^{-1} \circ \Phi^{-1}$ is sound.

Variations include:

- *Soundness for correct programs*: In this weaker variation, 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 subsystem of A.

- *Real-Time Soundness* in particular and *Resource-bounded Soundness* in general: In such stronger variations, assuming we have a notion of "size" or "duration" of transition paths between two states, we require that the exhibited transition path from c' to c'' be of duration less than some maximum admissible response time or more generally of size less than some maximum pre-allocated amount of resources. Resource-bounded soundness is important for real-time applications, particularly in presence of global synchronization of concurrent threads. *xref section on concurrent GC*

- *Strong Soundness*: We may associate to each reachable state c in C the set of abstract states in A that may be observed in future executions in C_0 starting from c (this set is a "final section" of the set of states, i.e. it is a set S such that $x \in S$ and $x \leq y$ implies $y \in S$). We will call this set the set of (future) interpretations of c, and we will call its elements interpretations of c. Soundness states that for any c that is reachable from O, this set is non-empty. Strong soundness is a stronger variation that requires that the set of interpretations of a reachable concrete state should not only be non-empty, but also have a lower bound; 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 stabilized yet. Given strong soundness, we may extend ϕ from O to the whole subset of elements in C reachable from O. Strong soundness may be useful when closely implementing deterministic systems.

- *B-soundness*: In this generalization, a and a'' are constrained to be in subsystem B of A. Thus plain soundness is the same as A-soundness. This is most useful when composing implementations: so as to achieve overall soundness when composing an implementation Φ^{-1} of A with C through O with an implementation Ψ^{-1} of C with E through Q, we will require Ψ^{-1} to be O-sound rather than sound (i.e. C-sound), and if Φ^{-1} is B-sound, so will be $\Phi^{-1} \circ \Psi^{-1}$. Other variations can usefully be generalized this way, too.

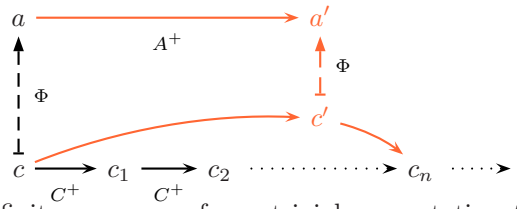
Note that previous properties (correctness and completeness) only involved observable elements of the concrete system: they only concerned relative properties of O and A, notwithstanding any relationship between C and O; they were property of ϕ alone, independently of j. By contrast, soundness really involves relative properties of O and C, notwithstanding any relationship between O and A; they were property of j alone, independently of ϕ. Note however that the generalization of soundness into B-soundness does involve restricting the image-set of ϕ to B, (and hence, the definition domain of j) before applying the usual definition of soundness.

Also note the temporal dissymmetry of the soundness diagram, whereas safety and completeness 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 programs with more concrete systems. Soundness is a property that is definitely oriented toward proper execution of programs, notwithstanding other properties of programs. If we wanted to capture the pure and perfect semantics of the abstract system or program, 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, soundness is an essential notion for implementations, that involves the very essence of what to implement is about. It is a very interesting tool to formalize synchronization in concurrent or distributed systems or even within single-threaded systems designed in modular ways (see section 3.6 below). It was the main contribution of our 1999 article.

Liveness

A useful property to require from an implementation is that of *liveness*: any concrete evaluation must spontaneously advance. Considering a subset A^+ of computations that advance in A and a subset C^+ of computations that advance in C (excluding, for instance, identity arrows), liveness can be expressed by the following diagram:

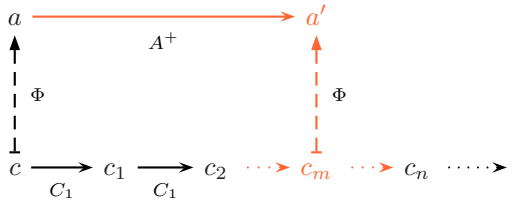


That is, for any infinite sequence of non-trivial computation there is an integer n such that the abstract computation has strictly advanced before concrete computation reaches the n^{th} transition — the concrete computation will spontaneous "go past" more advanced abstract states as it itself advances (liveness could be named "advance preservation").

Liveness is a composable property. If we look at temporal logic statements on transitions, liveness ensures preservation of increasing "must" statements by Φ.

Variations include:

- *Conditional Liveness*: the concrete system is only required to advance if the abstract system can, by adding the hypothesis of an a'' as in advance-preservation above. Conditional liveness implies advance-preservation, and since it is weaker than liveness, so does liveness. This abstract advance condition on the liveness requirement can be added to other variations below, too.
- *Real-Time Liveness*: For "real-time" or otherwise bounded resource variations of liveness, we strengthen the property by replacing the hypothesis that the sequence (c_k) be infinite, with the hypothesis of the sequence being "long enough", for some notion of "length", "size" or "duration" for concrete computations, and similarly requiring that the length of the deduced abstract computation from a to a' be longer than some minimum bound. Such variations compose when the notions of lengths match.
- *Strong Liveness*: assuming that we have a notion of "atomic" computations for C and a notion of advancement for A, and the sequence (c_n) is linked by atomic arrows, we demand that $c' = c_m$ in the liveness diagram. That is, not only must the concrete system "go past" some abstract computation as it goes along, but it must actually reach a stable state in a spontaneous manner. One way to achieve strong liveness, starting from a sound and live implementation, is to run a job in parallel that once in a while ensures that the implementation stabilizes.



Strong Liveness is a natural property to require from single threaded implementations of a computing system, but the requirement of spontaneous synchronization is so expensive as to make it generally out of question for concurrent and distributed implementations.

- *Strong Step Preservation*: assuming that we have notions of "atomic" computations for A as well as for C, we strengthen strong liveness above by requiring that the arrow from a to a' be atomic, too. Strong Step Preservation is typically achieved by straightforward "virtual machine" interpreters.
- There are many other possible variations on the theme of liveness, including combinations of the above.

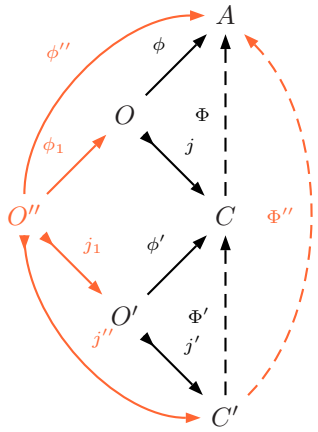
3.5 Combining Implementations

So as to build more complex ones from simpler ones, or conversely to analyze a complex implementation into simpler parts, it is most useful to examine ways that implementations can be combined.

Composition

A common and most priviledged way to combine implementations is to compose them.

Formally, if $\Phi^{-1} = j \circ \phi^{-1}$ is an implementation of A with C through O, and ${\Phi'}^{-1} = j' \circ {\phi^{-1}}'$ is an implementation of C with C' through O', then ${\Phi'}^{-1} \circ \Phi^{-1} = (\Phi \circ \Phi')^{-1}$ is an implementation of A with C' through O'', for a proper O'' defined below.



O'' is the full subcategory of O' such that node-wise $O'' = \phi^{-1}(j(O))$; j_1 is the canonical embedding of O' into O''; and $\phi_1 = j^{-1} \circ \phi' \circ j_1$. If we define $j'' = j' \circ j_1$ and $\phi'' = \phi \circ \phi_1$ then ${\Phi''}^{-1} = j'' \circ {\phi^{-1}}''$ is the implementation sought after. Checking that this construction behaves correctly on arrows (safety) 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 is essential in establishing that the above construction for ϕ_1 actually leads to a functor.

Two implementations can always be composed, but not all properties of implementations are preserved by composition of arbitrary implementations. Usually, to study of a property of a complex implementation, it will be decomposed (sometimes in a way that depends on the property), into a tower of simpler implementations that are composed in sequence (composition is associative) in such a way that only a few crucial steps require attention, whereas the other

steps are (strongly) complete implementations or otherwise such that composition with them preserves the property.

Inclusion

Another most useful way to combine implementations is through inclusion. In such a setting, the concrete system C with which we implement the abstract system A is itself included, through an embedding j' that needs not be full, in a system C'. We then consider both Φ^{-1} and ${\Phi'}^{-1} = j' \circ \Phi^{-1}$ as respective implementations of A with C and with C'.

The restricted scope of C as compared to C' can be used to model the respect of some kind of local invariant, that reflects normal use as opposed to exceptional situations, use within a context that only changes slowly during execution, correct execution environment as opposed to erroneous environment, or the temporary, expected, wished for or necessary satisfaction of some constraint whatsoever. The added arrows and possible added nodes in C' as compared to C conversely reflect a general case that may violate this invariant, include exceptional or non-factual situations and all that can ensue, include a whole family of execution contexts that weren't considered, include transitions that arise from erroneous behaviour (packets dropped, power loss, intruder attack, etc.), or otherwise include a larger class of possible computations.

Many of the above listed desirable properties need only be proven for either one of Φ^{-1} or ${\Phi'}^{-1}$ (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 behaviour during normal use, but not in case of exceptional situations (disconnection), whereas it still has sound behaviour in all cases.

An important application of this way of combining implementations is to model Fault Tolerance, as described below in section 4.4.

In an even more generally setting, we can consider multiple inclusion: we will have a family of systems $(C_x)_{x \in X}$ with various embeddings $j_{x,y}$ between some C_x and some C_y, and will consider a common implementation Φ^{-1}, up to composition by the proper canonical embedding, and will study the various properties that hold or do not hold for whole subfamilies $(C_x)_{x \in Y}$ of considered systems.

Selection

Selection is particular case of multiple inclusion, whereby a family of systems $(C_x)_{x \in X}$ are all included into a common system C, with C adding a few arrows to the disjoint union of the systems (C_x), plus of course all the composed arrows that ensue so as to have a category.

This models the fact that at every moment (for every implemented node in A), one of the implementations C_x has been selected to implement the current state of computation, but that at some times the system may switch to another implementation C_y. Such switch or migration in the current implementation may serve to to cope with situations for which the local choice of implementation wasn't designed, to find a more efficient implementation, to express a temporal decomposition of overall execution in several local phases, or to reflect a change in the execution environment.

In such a setting, we are really interested in the properties of the complete implementation C, rather than those of the partial implementations C_x, although it is by studying these partial implementations that we may establish the properties for C.

Selection is a very useful tool, that allows to focus on essential issues at hand when synthetizing or analyzing the local behaviour of an implementation. One notable use of it is to model migration, as described below in section 4.3.

Control

An special case of selection that important an tool to build larger selections of implementations or concurrent implementations (see section 3.6 below) is Control of an implementation, wherein given an implementation Φ^{-1} of A with C, we consider the selection between C and a copy of C in "interrupted state", whose nodeset is a copy of the nodeset of C, but that doesn't have any non-trivial arrows. The controlled implementation can then be used in an concurrent implementation in parallel with a monitor that may interrupt the running implementation, modify it, and then resume it.

The I/O-less version of controlled implementation (for concurrent systems as extensions) doesn't include any arrow from one copy to the other. The version with I/O (for concurrent systems as restrictions) includes pause actions from an active state to the corresponding interrupted state, and resume actions in the opposite direction. Pause actions can themselves be divided between traps and interrupts; traps are internally generated output actions wherein the implementation C detects that it is incomplete and generates an escape call to the monitor so it switch to another implementation; interrupts are externally generated input actions wherein the monitor stops the implementation for whatever reason. For full control over the implementation, additional arrows allow the monitor to arbitrarily change the state of a stopped implementation thus in the paradigm of concurrent systems as restrictions, the monitor can do whatever it wants — the monitor may also intercept any I/O of C as traps.

Using a monitor to control several implementations in sequence based on traps can implement selection. Using a monitor to control several implementations in parallel based on interrupts can implement concurrent systems (traps also necessary for communication). In any case, controlled implementations serve as bricks to build implementations for systems larger than the one actually implemented by the implementation being controlled.

3.6 Concurrent Implementations

In our the paper in which we originally proposed this notion of implementation [**?**], we explained how the computational interpretation of the soundness 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.

Soundness as a Synchronization Primitive

Consider an abstract concurrent system A (see section 2.6 above) based on the cartesian product $\Pi_{i\in I}A_i$ of the component processes A_i. We want a "modular" implementation of A as a concurrent system, based on separate implementations ${\Phi_i}^{-1}$ of each A_i with a concrete system C_i. We thus consider a concurrent system C whose components are the implementations C_i under Control (see 3.5 above) together with a monitor M that will somehow handle scheduling and I/O. Note that the monitor could itself in be considered as a concurrent system, though for our current purposes it doesn't matter how it can be decomposed; in any case, a concurrent implementation will always need one or several "system" components in addition to those "user" components that reflect activities from more abstract layers.

The problem that concurrent implementations face is with implementing communication and non-local modifications that happen at the abstract level for some set of the A_i's. Interrupting the corresponding concrete implementations C_i's may pause the concrete processes anywhere during their execution, hence generally not at an observable point, whereas the abstract operations are only defined for observable points, and trying to implement them at a wrong point may violate important system invariants: for instance, assuming bank accounts

are implemented in software as cells in (persistent) memory, without any atomic memory-to-memory value transfer operation; then transfer of money, that is atomic at the abstract level, will have to be done in several underlying concrete operations; reading the state of a memory cell while its bank account is in the middle of a transaction may yield false results, and result in money magically appearing in the system or disappearing from it. Once again, the concrete implementation of an abstract operation can only apply when the system when the system is in an observable state. Now, if communication or non-local modifications are originated by a process, then this process may already be in an observable state; but all other processes with which it needs communicate won't.

The solution to this problem is that the monitor must be able to synchronize interrupted processes in a stable state. That is, between the moment that a process is interrupted and the moment where some abstract operation may take place, the process must be rolled back or rolled forward to a stable state compatible with its past observable behaviour. This is exactly what the soundness property (see section 3.4 above) was about! Hence, the computational meaning of soundness is as a synchronization primitive in controlled implementations. By requiring the C_i to be sound, and having for each C_i a computable proof of soundness that takes us from a stopped state in C_i to a future state that is observable through Φ_i, we can ensure that abstract operations are always possible, and that there be no implementation-induced live-lock in the system. All in all, we can build a safe and sound implementation of a concurrent system out of controllable safe and sound implementations of its components.

Of course, 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 so as to begin the implementation of the abstract operation; several operations on disjoint subsets may even be attempted in parallel. Hence, there are as many useful notions of partial soundness as there relevant sets of components to be synchronizepd. Actually, when there is an inclusion or selection of implementations, or embedded implementations, as seen below about optimistic evaluation (see section 4.5), then even for the same system or subset of components, there may be several different soundness properties. To each way to observe a same concrete system as the implementation of a more abstract system, there corresponds a different notion of soundness that may be used for synchronization while implementing that more abstract system. The monitor must then be able to use as weak a soundness property as allows to implement the considered abstract operation, but no weaker.

Soundness in Existing Systems

This theoretical model allows us to cast a new point of view on existing techniques used for synchronization in the implementation of operating systems and languages offering user-level concurrency through time-sharing or parallelism. Time-sharing systems are thus typically implemented with a "kernel" that handles synchronization between a collection of "user processes" and "system services" that together implement the abstract computing system. Even in absence of time-sharing, multiprocessors must sometimes synchronize for global operations such as consensus or garbage collection (see section 4.6 below) that require system-level support.

- Ever since early timesharing systems [**?**], locks have been used as a way to synchronize processes, based on the paradigm of mutual exclusion. This can be interpreted in our paradigm as having one computing system per locked resource, and considering that activities implement abstract operations while putting some subset of the resources in non-observable state, so that each resource may be used by at most one activity at a time; the locking mechanism thus corresponds to a soundness prerequisite.

  Locking is thus a correct solution to enforcing high-level invariants but it the disadvantage

of being a low-level solution that requires a common discipline throughout all components of the concrete system, so as to avoid dead-locks (circular dependencies between locks), live-locks (situations when activities try to grab a lock but in practice almost never get it), or long transaction problems (lock being held longer than other activities can bear).

- A solution that minimizes the need for system support has commonly been used: "cooperative" multi-threading. In systems with cooperative multi-threading, such as FORTH multitaskers, Windows 3.1 or MacOS 7, thread switch can only happen at explicit safe points when the program explicitly polls for a flag indicating it is time to yield execution. This can be seen as an optimized mechanism for threads to release and immediately request back a global execution lock. This forces programmers to actually program at a lower abstraction level than they might like to, and to manually manage the spacing between yields whereas they are not ideally equipped to do so. This may mean poor performance due to routines that poll too often and poor latency due to routines that don't poll often enough. On the other hand, this solution doesn't need require any hardware support at all (like interrupt management). It can be viewed as a declarative way to assert that the system is in an observable state, which works great on single-processor systems, when the invariant defining observable state is manageable by the programmer that writes the code. It can also be viewed as intermixing and inlining the system monitor within user code. All these remarks converge into making this technique a good target for code produced by compilers (as in [**?**]), rather than hand written by humans.

- "Modern" Unix implementations typically forbid interruption of process while it is performing a system call. Pending completion of a system call, processes are put in an "uninterruptible state". This leads to poor real-time behaviour for long system calls, to long transaction problems in some cases (like a disconnected NFS server), and to leak of resources when processes hang indefinitely. One way of viewing this is that, from the point of view of the abstract system being implemented by the system, all synchronization, locking et al. has been moved on the system side and is mostly invisible to the user, barring latency and hanging problems.

- The implementers of the operating system ITS [**?**] had already invented in the 1960's a design that was much improved as compared to systems built afterwards: PCLSRing [**?**]. Whenever a process was interrupted, the system was to roll-back or roll-forward any pending system call, so that the process would be in a meaningful state that could be inspected and modified by other processes or otherwise participate in non-local communication; as the ITS hackers put it, a process could not be caught "with its pants down". As compared to the "modern" Unix approach, this removes the latency and hanging problems. It is also exactly an example use of our notion of soundness: recovering an abstractly observable state from an interrupted concrete process.

- Database systems that allow concurrent access express the abstract atomicity of their high-level operations through transactions (see section 4.4 below), and use locking to achieve fine-grained synchronization. In this context, soundness (forcing a set of locks to be released) can be achieved either by guaranteeing that transactions will finish promptly, or by being able to roll-back or roll-forward a transaction holding a lock.

All in all, formalizing the notion of implementation in general allows us to cast a new view on synchronization for concurrent systems. The soundness criterion appears as something that has already been discovered in practice by many developers of concurrent operating systems and programming languages, yet it is the first time (that we know) that this problem has been formalized in its general case.

More importantly, the full consequences of this discovery had never been taken: soundness is a concept that depends on the abstract system being implemented. Now the abstract system that ultimately interests the end-user is not the operating system or the programming language — it is the application that runs on top of the operating system and written in the programming language. We see why even when particular cases of this soundness principle have been discovered by many system implementers, and achieved in many sophisticated ways (see below), all these previous attempts fail in an essential way, because they provide only soundness for a particular abstraction level, preservation of some static system-wide invariant, that may appear as high-level to the system implementer, but that is low-level to the programmers who use the system[3]. This suggests that programming languages used to build concurrent systems should allow the programmers to explicitly declare their concurrent application's invariants, those that define their application as an abstract computing system, and include tools to enforce consistency and provide soundness with respect to these invariants. See next chapter 8 for more about this.

Achieving Soundness

The general idea behind implementing soundness is that the state of an interrupted process to be stabilized must be rolled back or rolled forward to some previous or next stable state. Roll-back is implemented by making reversible all operations effected since last stable state, which may mean storing all modified components of the stable state, or logging reversal information for every modification. It isn't always possible to roll-back, since some operations are intrinsically irreversible; for instance, you can't usually unprint data that was sent to a line-printer. Roll-forward is typically implemented by reentering the interrupted activity in a special mode that will stop at next safe point. Safe points may detect the necessity of stopping by polling a particular system flag[4], or by releasing and reacquiring locks at the end of atomic blocks, which are ways to manually achieve strong liveness as previously suggested. A more complicated but more efficient efficient technique is to temporarily modify user code (or use a shadow copy of modified user code) so its jump out into the monitor instead of continuing execution as usual; Ogesen used such technique for synchronizing the GC in the EVM implementation of Java [**?**]. These techniques have different tradeoffs regarding spending of space and time, cache coherency issues, development complexity, etc., that make each of them suitede to different situations; however, a same system implementation may use any combination of these techniques so as to achieve soundness in an overall efficient way. All techniques but polling suppose compiler support, and even polling is better done by the compiler than by hand.

[3] A case in point is the failure and removal of the thread stop and suspend primitives from the Java 1.2 language [**?**]: these primitives were proved useless because though a thread you interrupted would be in a safe state with respect to the Java language's semantics, it would be in an insafe intermediate state with respect to the application's semantics. The "solution" proposed by Java designers was to not use any interruption primitive, but for application developers to manually use locking and polling. Hence, even though some Java implementations themselves used very advanced tricks to implement soundness internally [**?**] with respect to Java's semantics, these mechanisms were not available for programmers to enforce their own application-dependent semantics.

[4] Polling a global flag can be made very efficient. The most obvious implementation involves the check of a flag in a reserved register or memory adress followed by a conditional jump that is not taken in the general case, which takes two to three clock cycles per poll. If polling is done very often, it can be done in a single memory read instruction on an address such that the MMU would trap or not. When it doesn't trap, the check takes one cycle, and when it does, the trap's implicit context save does half of the job of the intended high-level context switch.

3.7 Comparison to Existing Approaches

compare LOTOS, TLA+ and Refinement calculi

Whereas plenty of other formalisms have already been developed to specify one aspect or the other of the correctness of such implementations, none is generic enough to apply to to both arbitrary source language and arbitrary target language in a concurrent setting.

Many have completely formalized and verified compilers [**?**, **?**, **?**, **?**, **?**] but only for single-threaded centralized programming models, in ways that do not generalize obviously to specification of arbitrary parallel or distributed programs. Araujo [**?**] discusses correctness of a parallel implementation of Prolog, but in a monotonous logic setting where he needs not bother with atomicity, which is precisely the main problem of concurrent implementations. Lamport has developed notions of atomicity and implementation for concurrent systems [**?**, **?**], but his single-universe formalism isn't immediately suitable to specify higher-order systems, even less metaprogramming or compilation. *XXX - cite ??? lermer97compilation*

As compared to existing formalisms that allow for multiple levels of "refinement" our approach can be viewed as "phenotypical" — based on the observable semantics of programs rather than "genotypical" — based on each level's source code being an incremental modification of the previous level. Apart from being simpler and "cleaner", our approach also allows to express relationships of implementation between systems of a completely different genetic line, including systems developed in completely different languages of formalisms, as long as they have proper operational semantics.[5]

In π-calculus and variants thereof, the notion of simulation can be seen as a particular kind of implementation, and bisimulation as a particular kind of mutual simultaneous complete implementation. *XXX - cite* Our notion or implementation can be considered as a generalization of simulation, in a generalized setting where the calculi of the original and simulated programs need not be both the same variant of π-calculus. The properties of implementations that we defined have equivalent in the vocabulary of simulations in π-calculus that we know of.

The originality of our approach, as compared to the above, lies in several points:

- Our formalism is at the meta-level: it considers whole computing systems are objects. This allows us to express general properties of relations between systems, and combine implementations as components of larger constructions, instead of only being able to study one implementation at a time.
- We have a general setting that can be further specialized in several ways by specifying additional properties, instead of an ad-hoc setting only suited to one specific proof

[5] Another completely misled use of a genotypic approach when a phenotypic approach is required is "inheritance" in statically typed object-oriented languages such as C++. Actually, in the original dynamically typed OO languages such as Smalltalk or CLOS, inheritance *can* indeed be used to model phenotypes. The statically typed OO language Java saves the day by having "interfaces" that allow to specify phenotypes whereas its usual inheritance follows the same misguided effort as in C++. Statically typed languages such as ML and Haskell do have phenotypic abstractions: functors and type classes.

Now, this doesn't mean that genotypic abstractions are useless: indeed, inheritance can be used to model many things, and it is quite possible to have both inheritance and interfaces in a statically typed language, as demonstrates OCaml. But the genotypic approach is more of an ad-hoc hack that is strongly limited by problems of path-dependence, choice of representation, etc. — whereas the phenotypic approach is a universal conceptual tool that really allows to abstract over a problem.

On the other hand, there is an important point in the genotypic approach that is not encoded as such in the genotypic approach: non-intrusive incremental development. But then, this feature of programming systems is much more general than the extremely limited particular cases of it made possible by a genotypic technique such as inheritance of classes in object-oriented. Indeed the genotypic approach allows incremental development only with respect to a predetermined incremental representation of programs, whereas the phenotypic approach allows incremental development with respect to arbitrary incremental changes to programs, including global transformations. See more about global program transformations in our section on metaprogramming.

of implementation. This allows us to simultaneously interoperate with various existing paradigms while being able to adapt to their specificities.

- We handle the general case of concurrent or distributed execution, both for the language being implemented and the language implementing it, and we provide a new framework for expressing synchronization primitives (soundness). In particular, we do not require a global "true" notion of atomicity, but can express each abstract calculus' own notion of atomicity and account for how synchronization may happen at the concrete level.

Chapter 4

Expressing Known Phenomena

We can test the power of our formalism by seeing how it provides a common framework to express known techniques and concepts in computer science.

All the below techniques are very well known. What we are trying to do is to suggest how our approach is relevant to the specification of their behaviour, which can be used informally during design and analysis of systems, or formally when building proofs of correction of systems, or systems that automatically prove correctness of what they do. Our underlying claim is that as soon as it becomes obvious how to relate these existing techniques to our notion of implementation, all the power of the meta-level composable and specializable framework we propose becomes available for the analysis, design and correct development of systems using these techniques. We concede that this is all easier said than done, but nonetheless claim that it is better said than remained implicit, or worse, ignored.

4.1 Interpretation and Compilation

Our formalism directly applies to specification and verification of correctness of interpreting and compiling implementations of programming languages.

Interpretation

Interpretation is the usual case of having the system comprised of the interpreter together with the program source behave as an implementation for the abstract program (whose semantics is defined by its abstract language), with proper constraints as to the way the concrete I/O behaviour matches the abstract I/O behaviour. Here, "together with" means, that the initial state of the concrete system can be seen as the cartesian product of the interpreter's internal data with a straightforwardly encoded equivalent of the program's syntax. Alternatively, the program's syntax can be encoded in a straightforward way into some I/O operations at the concrete level, and the interpreter is the initial state of the concrete system, that after these I/O operations must behave as specified. These two ways are but the internal vs external representation of state as discussed above in section 2.5. The details can be adapted to a particular interpreter, and it only matters to fixate them when comparing the relative properties of two interpreters. Somehow, interpretation is already a meta-level phenomenon: it involves the abstraction of an implementation over an input program. In other words, an interpret is a concrete object of type `Abstract Program Representation -> Behaviour`. Note that as explained above in 3.3, the matching of behaviour between the abstract language's primitives and its interpreted implementation is usually an external constraint over which the implementer

of the interpreter has but limited choice, depending on the human conventions as to the *intent* of the I/O primitives, and to more specific conventions of the computer, its hardware and its operating system.

Compilation

Compilation is a process that takes as an input (the source of) an abstract program, and outputs (a proper representation of) a concrete program that implements it. Compilation is interesting to formalize, because it is neatly a concept at the meta-level: it is about a system that manipulates encodings of the semantics of other systems both as input and output. In other words, a compiler is an object of type `Abstract Program Representation -> Concrete Program Representation` in some computing system possibly unrelated to the source and target systems. Once again, input and output of the compilation process can be encoded either as external I/O or as internal state. What is more interesting is that this data must be defined in terms of an externally constrained encoding convention for programs. This encoding may be obvious in the case of source programs: a stream of (ASCII or whatever) characters, or the equivalent. In the case of the target program, there must also be a similar encoding convention: an "object file" respecting the operating system's ABI for "native" executable code, a system memory image for computer ROM or other bootable device, an archive of bytecode for some virtual machine, another stream of characters for source in a lower-level language (Java, C, assembler, VHDL, silicon mask), etc. What is notable is that as "low-level" as we get, programs are always encoded as data, and the actual code, the semantics, is always dependent on an implicit, external processor: the operating system, the CPU, the virtual machine implementation, the semi-conductor fabrication process, the laws of electricity, the laws of logical reasoning, etc. As usual, the encodings are an external constraint with little choice for the implementer.

Semi-Compilation and Semi-Interpretation

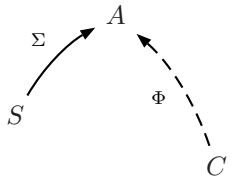
A valid though trivial way to compile a program is to prepend to it the code of an interpreter (or otherwise embed into a same target program the code of the interpreter and the source of the abstract program). It might eschew most of the potential benefits for which compilation is often touted, but it fulfills its role and quickly provides a valid answer to the problem, that can be refined later if needed. At the other end of the spectrum of compilation, a compiler would use artificial intelligence, black magic or godly miracles so as to produce "native code" (that is, code directly using the formalism of the target concrete system) that would be specifically suited to optimally implement the abstract program (for some meaning of "optimal"). In between — and the latter end being more of an unreachable goal than an actual end, about everything is actually in between — compilers will deploy a wide family of ad-hoc techniques and so called "optimizations" so as to achieve reasonable performance for actual software on actual hardware. These two opposite approaches can be combined, by doing "semi-compilation", ad-hoc compilation of source programs into an intermediate format that would then be interpreted (which is what byte-code compilers do), *cite Smalltalk, OCAML, Java, etc.* or conversely optimizing in an intermediate code obtained by combining an interpreter and a source program (which is what partial evaluators do). *cite COMPOSE, etc.* As another symmetry, interpretation of programs can also be done by compiling them and running them immediately afterwards. *CMUCL, JIT, etc.* More generally, interpretation and compilation can be used dynamically as subroutines to implement large systems; a compiled C application will dynamically interpret the printf sublanguage, its own configuration file language, etc., while an interpreted database application will dynamically compile SQL queries, digest user input files, etc.

Simple Diagrams

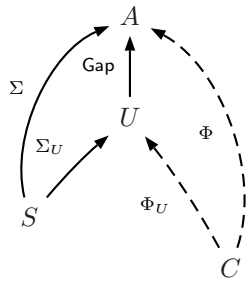
A simplified diagram to describe what compilation is about would be as follows:

take an abstract computation A and dump a concrete computation C that is correct according to an interpretation Φ.

However, you cannot usually directly manipulate the abstract computation A: it cannot manipulate programs up to operational equivalence, since operational equivalence of abstract computations is not computable. You can only manipulate data structures that encode computations. In practice, you start from a source representation S of A, with a semantics Σ:

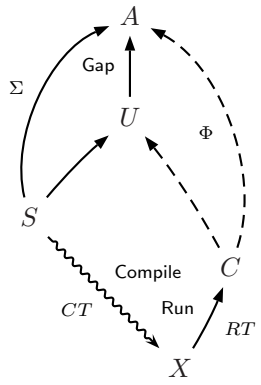


Now, the compilation process itself cannot fully fill the semantic gap between the source data structure and the abstract computation; it will manipulate programs up to some equivalence that is weaker than operational equivalence. The compiler thus manipulates an intermediate category U of the semantics actually understood by the compiler, that still has a semantic gap Gap, albeit a smaller one.



Finally, the compiler doesn't output a running computation, but a compile-time representation of a future running computation. Whereas the previous arrows were conceptual arrows in a category of transformations to reason about, the compiler is an arrow Compile in a category CT of effective computations, and there is an arrow Run to turn the result of compilation into

an actual runtime computation.



Formalization Potential

All in all, we have only scratched the surface of using our formalism to specify the behaviour of compilers and interpreters, and we already find the opportunity to illustrate how it can serve to express in a simple way a few crucial phenomena. We believe that our clarification of the notion of implementation offers great opportunity for the formal study of interpreters and compilers, and the building of staged compilers, the proof of compiler correctness, the modular development of compilers into components with well-defined interfaces, or the manipulation of program transformations as first-class objects.

4.2 Abstract Interpretation

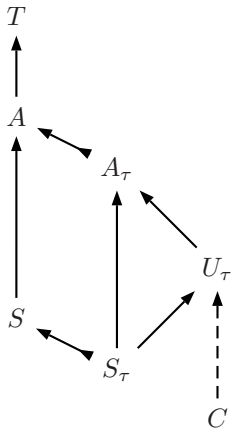
A topic related to implementations is Abstract Interpretation [**?**]. Given an implementation $\Phi^{-1} = j \circ \phi^{-1}$ of A with C through O, we called Φ a partial abstract interpretation, and ϕ an abstract interpretation. Indeed, ϕ is a functor from a concrete category to a more abstract one, that synthetizes information, losing irrelevant bits, while preserving essential structure of a program. However, our choice of this denomination is more of a tribute and a reminder of broad similarities than an equation in the approach taken.

The traditional field of Abstract Interpretation are usually interested in functors from some denotational semantics to other denotational semantics for the same syntactic algebra, whereas the functors we manipulate are from an operational semantics to an operational semantics for a different system. Also, most importantly, the field of abstract interpretation focuses on extracting information from given programs (going from concrete to abstract), whereas the field of implementation focuses on finding a program that fits the given specification (going from abstract to concrete).

Abstract Interpretation, or equivalently static typing [**?**], is usually used to prove some properties about programs, or to eliminate whole categories of bugs. In traditional settings, Abstract Interpretation is applied statically to programs: that if a type τ has been established for a given program, then it is guaranteed that all future executions of the program also have type τ.

The knowledge of the type τ of an expression can be used to compile it efficiently, because the category A_τ of computations of type τ is much more restricted that the category A of potentially untyped computations. Many optimizing program transformations are thus made available that are not valid within A type-directed compilation [**?**]. *find proper citation - Stalin.* This can be summarized by the following diagram, where S is the source computing system,

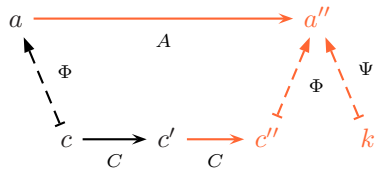
and A its abstract meaning, T the universe of types, S_τ and A_τ the respective restrictions of S and A to computations of type τ, U_τ is a space of computations of type τ up to a set of transformations known to be valid for such type, and C is the concrete computing system implementing the desired abstract computation A:



4.3 Migration

In distributed systems, migration of an activity consists in modifying the host processor on which the activity is running during its very execution; it might involve momentarily stopping the activity and waking it up after the operation has succeeded, but is otherwise unintrusive and painless for the activity and its programmer. The concept can be generalized into defining migration as any unintrusive and painless modification of the set of underlying resources involved in implementing an abstract program. Unintrusive and painless here means that the modification is invisible at the abstract level, and that activities written only with high-level primitives need not worry about it and cannot observe any visible side-effect, although activities that call low-level primitives might and could.

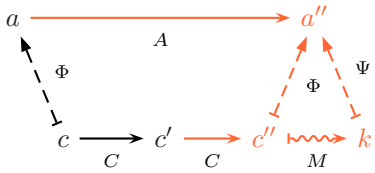
This generic concept of migration can be modeled with the following diagram:



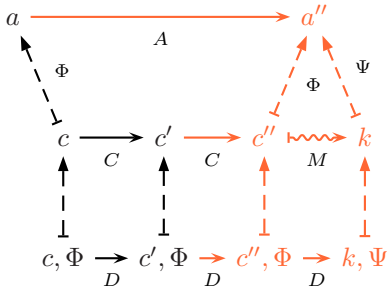
In other words, migration of an abstract computation A consists in interrupting the underlying computation C (corresponding to interpretation functor Φ) at current concrete state c', synchronizing it to a stable state c'', corresponding to some meaningful abstract state a'', and then reimplementing said abstract state a'' into a new computation K (corresponding to interpretation function Ψ).

Of course, sometimes the map ϕ or J might be very expensive to compute, or even uncomputable. In practice, what one wants is to instead use and compute a reasonably complete

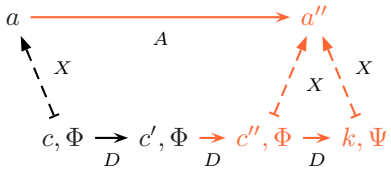
subset M of $\Psi^{-1} \circ \Phi$, as in following diagram:



Actually, the interesting cases are those when this migration happens internally to the computing system (whether explicitly triggered, or dynamically decided by some resource-optimizing scheduler). What usefully appears as a change in implementation at the abstraction level of C and K can thus be seen as simple computation in an underlying computation D at a lower abstraction level, whose state comprises both the current state of the intermediate abstraction level and data identifying the current implementation tactic for that level, as in following diagram:



This diagram can be simplified by omitting the intermediate abstraction level:



This generalized migration can thus be seen as the switch in an implementation built using the technique of selection (see section **??** above).

Many well-known phenomena can be thus formalized or refactored with this abstract kind of "migration"[1].

- Of course, in the case of distributed system, migration of mobile activities from one host to another, on a same or different architecture.
- Tracing garbage collection can be viewed as migrating the heap from an old space full of garbage to a new fresh space (see the section 4.6 on Garbage Collection below)
- Database schema update can be seen as migrating data from an old representation to a new extended (or restricted) one.
- Dynamic type-directed compilation can be seen as migrating code from some inefficient or invalidated setting to a new setting that allows speedier or type-correct setting[**?**].
- (Dynamic) code refactoring, change in data representation and upgrade of virtual machine code can likewise be seen as (dynamic) change in implementation.

[1] This point of view has been studied within the Tunes Project since 1996 ☞ `http://tunes.org/Migration/`.

A common pattern in all these uses of migration is to have an automatic mechanism detect that the current implementation is invalid or inefficient, and can invalidate and interrupt the current implementation without corrupting its state (using the soundness property in a computable way), then use some strategy to select a new implementation tactic, then the migration itself from the old implementation to the new one, and finally the activation of the new implementation.

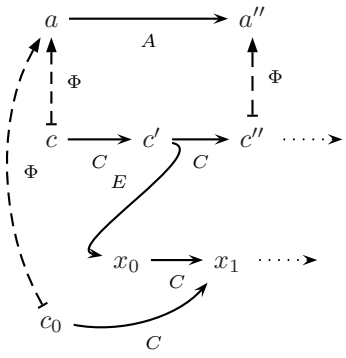
If we start from the above migration pattern, consider not just abstract interpretation functors, but natural transformations between these functors, we can also modelize incremental evolution of system representation: dynamic implementation techniques that depend on runtime values, dynamic balancing of code between several hosts, incremental garbage collection, live schema upgrade in a database, partial infrastructure update in a virtual-machine implementation, etc.

4.4 Fault Tolerance

Inclusion of implementations (see section 3.5 above) allows to model fault tolerance: given an implementation of an abstract system A with a concrete system C, we require that it remains a correct implementation even in presence of additional "fault transitions" extending C into a system E. I.e. Φ^{-1} is an implementation of A with C, and j being the embedding of C into E, $\Psi^{-1} = j \circ \Phi^{-1}$ is our implementation of A with E. These fault transitions might include packet loss, limited data corruption, requests for incorrect operations (e.g. by crackers attacking the system), system reset (e.g. consecutive to power failure or system crash), etc.

What is interesting is that our concept of implementation remains valid for both Φ^{-1} and Ψ^{-1}, and that the difference in reliability due to faults is expressed through discrepancy in the set of properties verified by the two implementations: some properties (liveness, real-time behaviour, etc.) will only hold in the restricted system C where faults don't happen, whereas more essential ones (correctness, soundness) will hold even in presence of faults. In more complex cases, statistic properties must hold in presence of erroneous behaviour with non null but low enough probability — we are convinced that our framework could be easily adapted to the expression of such properties.

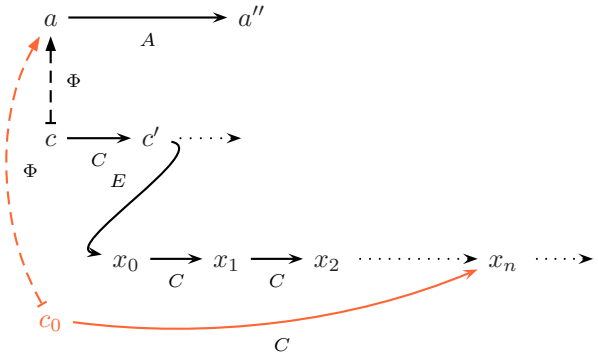
Here is a diagram that shows what could go on in such a setting:



Starting from a concrete state c that correctly implements an abstract state a, we may begin computations, that may ultimately lead to a further state a''. But until the evolution of the system has been properly committed, a failure transition may set us from a state of intermediate computation c' back into a state x_0 where information was lost. However, if computation continues from this set back state, it will eventually reach a state that x_1 that will be a valid future for a concrete state c_0 that also implemented a. Thus, the set back will have been

contained within from the point of view of external observers, all abstract observations will be coherent with correct computation from valid abstract states to valid abstract states, without ever invalidating a visible effect.

We can even combine the above case study with the liveness diagram to express a formal property of fault tolerance that states that E being an extension of C with additional arrows but no additional nodes, if we're implementing the future of a with C despite failures in E, starting with c, then any long enough sequence of C-future states of an E-future state x_0 of c must ultimately include a C-future x_n of a state c_0 implementing of a:



Fault tolerance is also an essential property of filesystems, databases, object stores, and other persistent systems: these systems must be able to keep data in a consistent state despite various kinds of software and hardware failures. After a failure, a consistent state for the data must be recovered, and even when loss of recent modifications is allowed (see Optimistic Evaluation below in section 4.5), the loss should be contained and not corrupt innocent other data. [**?**] Our framework for studying the properties of implementations in general and fault tolerance in particular allows to capture the essence of the ACID test of databases. *find proper citation*

- Atomicity with respect to the application's abstract model is captured by our Correctness criterion: the state of the system can only observably evolve by applying complete abstract transformations; when observing (possibly through Soundness), either a transformation was committed or wasn't.
- Consistency is also captured by the Correctness criterion, since Correctness is a global criterion, not just a local one. The difference is that Atomicity is conformance to operational invariants of the abstract model, whereas Consistency is typically enforcement of its declarative invariants.
- Isolation is yet again captured by the Correctness criterion. The difference being that Isolation specifically expresses part of the Correctness criterion that must be enforced in presence of concurrent transactions.
- Durability is guaranteed by the Fault Tolerance criterion as formalized above.

An additional requirement may be that no modification declared as "committed" might be lost — i.e. applications may synchronize on the event of some transaction being committed. The case of the synchronization between successive transactions in a single-threaded activity is also captured by our Correctness criterion. The case of synchronization between concurrent activities, this can be expressed the use of soundness or a variant thereof (see section 3.6

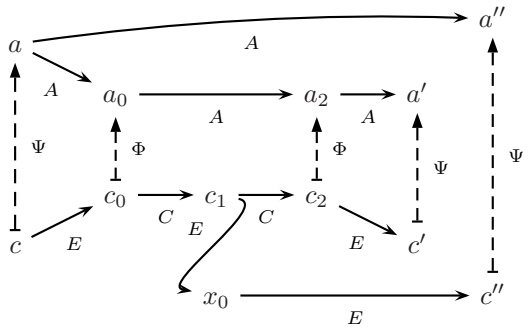
above on Concurrent Implementations). Considering the high latency of committing data in a database, we are also lead to see how our framework helps deal with optimistic evaluation, allowing synchronization within transactions.

4.5 Optimistic Evaluation

Optimistic evaluation consists in making computations whose validity depends on assumptions that may later prove incorrect [**?**]. *more citations ...* Using resources that would otherwise remain mostly unused to do such computations can greatly enhance performance, especially if the assumptions have a high-probability of being correct. This is the principle behind speculative execution in CPU implementations, but also behind database transactions that may be rolled-back. Actually, optimistic evaluation is essential whenever persistence is involved[2].

This can be modelled within our framework by refining the construction we used for fault tolerance: there again, we'll have a "same" implementation of an abstract system A with a system C "without fault" where some hypothesis is assumed to hold and a system E "with faults", where this hypothesis might or might not hold after all. The difference as compared to the setting used with fault tolerance is that although initial concrete states chosen for an abstract computation will be the same, and although the concrete system will also be the same (up to additional fault arrows), we allow for some discrepancy in the way the two implementations interpret their current state. Formally, suppose Φ^{-1} is an implementation of A with C, with $I \subset \Phi^{-1}$ being a restriction of Φ^{-1} associating some initial concrete states for some of the abstract computations; j being the embedding of C into E, $j \circ I$ will also be included in our implementation Ψ^{-1} of A with E; but whereas Φ^{-1} will be optimistic and advance much faster than Ψ^{-1} at the microscopic level, Ψ^{-1} won't validate its results until the hypothesis is checked.

An example diagram of things happening is as follows:



Where Ψ^{-1} implements A with E starting from concrete state c observable as abstract state a, we make an assumption, and start computing with implementation Φ^{-1} of A with a subsystem

[2] Considering the very high latency of committing anything to disk, operations would be prohibitively expensive if every modification to persistent data was to be systematically committed before a new operation could be performed on the same data. Things would be even worse considering that most computers don't have any mechanism to assess that data is indeed committed (the ATA interface used by most disks doesn't provide any such mechanism), not to talk about mission-critical applications where data is not to be considered safe until a remote backup is made. Optimistic evaluation is thus especially useful in these cases, since individual write operations have extremely high probability of success, yet the probability of failure is high enough over the lifespan of an application that it cannot be completely ignored. Indeed, desktop computers equipped with popular operating systems crash daily, and even servers may crash once in a while because of a power failure, a hardware malfunction, a low-probability software bug, or leakage of some software-managed resource. It has been remarked [**?**] that together with disk-based transaction protocol, the addition of a simple battery-backed "Transactional RAM" (TRAM) to hold transaction journals could considerably enhance the latency and reliability of persistent systems, including databases.

C of E. This assumption can lead to states c_0, c_1, c_2, some of which are observable through Φ as having a meaning of the computation having advanced beyond a. However, we can only commit the changes into a state c' observable through Ψ after having assessed that the assumption was valid. Indeed, while in the middle c_1 of the computation in C, the assumption may be invalidated, in what can appear to be an "error" transition as far as C is concerned, but is normal operation as far as E is concerned. The system would thus evolve in a state c'' observable through Ψ as an abstract state a'' for which the assumption doesn't hold. Note that the possibility of invalidation means that C may not include any externally visible effect (irreversible arrow in A) as observed by Ψ — although it could internally simulate such effects (as observed by Φ), that will only be make-believes until they are possibly committed.

Thus, for instance, after a file has been written to, all running processes will see it as modified, although the changes might take several seconds to be actually committed to disk. So if no fault occurs, the write operation can be thought as having succeeded immediately, and processes may start computing based on this modification. But if a fault occurs, we'll realize that the underlying implementation Ψ isn't really as advanced as the user may have thought it was[3] since he was interacting with what seemed to be Φ. If failure is frequent or catastrophic enough that user confusion is a problem, the user interface can use concurrent synchronization primitives (see below) to detect when modifications are committed or not, and use color codes or some other marker[4] to distinguish the status of the data he is manipulating as far as commitment goes at various abstraction levels. Such solutions can be viewed as making the user part of the computing system, the user having to do part of the failure-handling: in manually persistent systems, by explicitly saving files every now and then, and in both manually and orthogonally persistent systems, by accepting to consider as lost the work he did since the system last committed its modifications.

Note that although C is a subsystem of E, it is considered with an interpretation function Φ different from Ψ. Thus, for instance, invoking the respective soundness properties of Ψ^{-1} and Φ^{-1} will lead to very different results. Starting from a same state of intermediate computation c_1, using the soundness of Φ^{-1} will lead to a state like c_2 that is observable through Φ, while using the soundness of Ψ^{-1} will lead to a state like c' or c'' that is observable through Ψ, which might not have advanced significantly from a, and might even involve rolling back and invalidating current optimistic computations. This is indeed what happens in transactions with preemptible locks: if we want to force the system into a stable state for an operation that doesn't need to preempt the lock, then we synchronize the thread that holds the lock without invalidating it, whereas if we need to access the data protected by the preemptible lock, then we will invalidate the transaction. This leads us to how soundness can be considered a synchronization principle.

4.6 Garbage Collection

Garbage collection [**?**, **?**], or "GC", is a technique used for automatic memory management when implementing typed high-level languages whose programming model involves a dynamically evolving graph of objects. It consists in having a special system activity dynamically modify the way the graph is represented in memory, so as to minimize the use of memory

[3] Jochen Liedtke [**?**] tells that this has actually been used as a feature: on a computer running the persistent system Eumel, considering the five minutes of average latency for committing data to the persistent store. a developer would occasionally reset the system on purpose, to prevent a change he just made from being committed to disk, when that change had quickly turned out to be catastrophic.

[4] For instance, EMACS uses a double star marker ** in the status line to indicate that the current buffer has not been saved since last modified; meanwhile, it will periodically save modifications to an autosave file, but will only overwrite the original file upon explicit user interaction.

resources. It allows programs to run that would otherwise not fit in the available memory, or would have to be written, at a cost, in a completely different way that manages memory manually; in particular, it allows for many programming styles like functional programming, logic programming, declarative programming, that considerably simplify the task of the programmer, and make for faster, smaller, cheaper, and more reliable programs. GCs written by specialists in memory management also end up in software more efficient, more reliable, and in the end cheaper than software where memory is manually managed by non-specialists, and much cheaper than software where memory is manually managed by specialists.

GC at Multiple Abstraction Levels

In our paradigm, memory allocation in general is characterized by the interpretation function that maps the state of a memory zone called the heap into a graph or of high-level objects. Usually, the data in an individual node in the graph are represented by the contents of cells at consecutive addresses in memory, with arcs in the graph being represented by the address of the target node from the appropriate cell in the source node. The heap also contains "meta-data" that helps determine which addresses are used by which kind of nodes, which cells in a node are addresses and which are raw data, which addresses are free, how to handle further allocation, etc. A same object graph may have many representations as a heap in memory, depending on how abstract nodes map to concrete addresses. A given object graph may also lack any representation as a heap in memory, if it contains more nodes that can be fit into the finite memory size; hopefully, all the graphs involved in the particular program being run will have a valid representation as found by the implementation. Of course, not all memory states are valid graph representations, as the encoding of graphs may rely on complex invariants to be respected by well-formed heap. Lastly, a given memory state must represent only one valid object graph, or else we would be unable to use the concrete computations to unambiguously implement the abstract computations.

Garbage collection consists in changing in the way the graph is represented; in a way, it is thus a particular case of Migration (see section 4.3 above). Conceptually, this modification in graph representation is done by a set of system activities called the garbage collector, while modifications to the graph itself is done by a set of user activities called the mutator.

- At a very high level of abstraction, the semantics of the high-level programming language, the mutator has visible effects based on its finite explorations of a graph that may be conceptually infinite (at least potentially), and the garbage collector is invisible.
- At a high level of abstraction, the high-level programming language's virtual machine, the mutator atomically modifies a finite graph, and the garbage collector may prune "dead" parts of the graph that are unreachable from any mutator activity.
- At an intermediate level of abstraction, the virtual machine implementation, both the mutator and the collector modify a graph representation, that remains in a consistent state at every step, although it may change significantly between steps, with graph nodes being moved around. So as to trace live nodes in the graph, the GC will partition them into conceptual colors (sets), that may be implemented as address ranges [**?**], bitmaps [**?**], doubly-linked lists [**?**], or any appropriate means. Incremental modification of the heap metadata can be seen as a phenomenon of incremental migration.
- At an even lower level of abstraction, the processor instructions, both the mutator and collector modify memory in ways that introduce intermediate states where the heap has no coherent meaning in terms of an abstract graph. So that the lower level of abstraction correctly implement the intermediate level of abstraction, the low-level activities

that constitute collector and mutator must synchronize around the intermediate level's invariants.

Our framework proposes a way to express the implementation relationship between these abstraction levels, and to study the properties of these implementations; it promotes explicit formalization of of these abstraction levels and implementation relationships as a paradigm to achieve modular correctness proofs.

GC and Properties of Implementations

We may study how garbage collection interacts with the various desirable properties we listed for implementations.

Eliminating dead objects is a *safe* transformation of program state in as much as it preserves the semantics of the language: that is, it allows no abstract user observation that would distinguish the transformed program from the original one. A garbage collector needs not eliminate all unreachable objects; if actual computations run concurrently with the garbage collector, determining the exact set of objects that are reachable at a given moment would require stopping all other activities for the duration of the GC, which can be more expensive than is worth. So garbage collectors, concurrent or not, may have various degree of "conservativeness" whereby they eliminate sets of objects that are smaller than the set of unreachable objects. On the other hand they may not collect any reachable object; that would be unsafe, and would likely lead to erratic behaviour of implemented programs when they try to access an object that has been collected. Similarly, moving the data addresses for graph nodes so as to compact the heap is a safe transformation, because it is invisible at the abstraction level that only deals with finite graphs. There again, the GC may be precise or conservative about which objects in the heap may be moved, and which are locked by mutator threads.

An implementation with GC will be *complete*, if it enables the representation of all possible object graphs that may happen as valid states during the abstract computation being considered; this is only possible if we can give a bound on the total amount of live nodes at any moment in the computation state, so that they may all be encoded within the finite amount of available concrete memory.

An implementation with GC will be *sound*, at least at the virtual machine level, in as much as the GC and mutator must preserve the heap in a meaningful state. Indeed, so that the GC may run when any thread requests memory allocation, it must always be possible for the GC to observe the heap in a stable state and modify it when needed. In a concurrent system, this means that all threads may be stopped in a safe state, hence, soundness of mutator threads with respect to the heap invariants, as discussed in our section above (3.6) on Concurrent Implementations [**?**, **?**, **?**]. Soundness thus appears as an essential property of implementations so as to safely combine multiple concurrent activities. Note that if the GC invariants are weak enough, then about any point in a program will be safe, except for a few critical regions of code at the end of which polling may happen; however, such a weak GC invariant also means the GC will have to be very conservative (see above) in the way it determines what is unreachable or movable, and be accordingly limited in the transformations it can perform on the heap (graph pruning and heap compaction).

An implementation with GC will be *live* if it never stops or hangs, if progress of concrete computations correspond to observable progress of abstract computations; this means that GC activity will always give the hand back to mutator activities after a (hopefully short) lapse of time. An implementation with GC will be *real-time* (have real-time liveness) if there can be a bound on the lapse of time involved in making abstract computations progress; in presence of concurrent activities, this affects not only liveness of the implementation, but also soundness,

since the mutator activities must "stabilize" fast enough so that the GC can do its job and yield back execution to other mutators fast enough.

4.7 Systems with Multiple Abstraction Levels

Often, computing systems can be programmed at multiple abstraction levels. For instance, a system could be programmed through user interface gestures, through a configuration language, through an high-level embedded extension language such as LISP, through an API in a low-level language such as C, through inline assembly statements inside C code. To understand the semantics of any particular computation, one has to "descend" to the lowest abstraction level involved in such computation.

In a static reasoning meta-system, in a framework that can model only one global mapping of abstraction level, this means that reasoning is only possible in as much as the system would be completely described in terms of its lowest level of abstraction. Formalism thus becomes both very complex to handle conceptually, and very expensive to handle computationally, because all high-level operations must be dealt with as very complex combinations of the lowest-level of abstraction. Such an approach quickly becomes unmanageable as lower-level computations are considered: depending on the needs of the day, one may be satisfied with an abstract view of the world, or may need to address concrete resource management, or maybe deal with atomicity at the symbolic assembly-level, or consider operations involving binary code, or discrete signals and logic gates, or analog currents, or even quantum electronic effects. All these abstraction levels have their relevance, for their own purpose[5]. None is the "absolute" lowest level; there might always be something below.

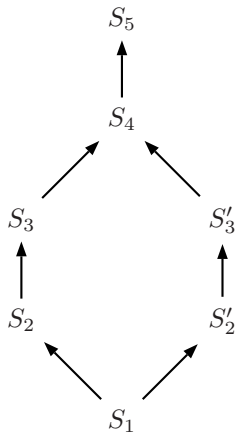
Since a "one world" view in both practically impossible and theoretically undesirable, people commonly use multiple abstraction levels to describe, implement and study computing systems. However, with logical formalisms and paradigms that are incapable of simultaneously dealing with several levels of abstraction, the coherency between the many representations of the many parts of the system at the many levels of abstraction must be asserted manually, which can cost a lot in terms of developer work and/or of unreliability of the resulting product. In other words, one-world formalisms in practice only allow for the study of partial correctness of a system.

The modelling paradigm we propose allows to simultaneously consider many abstraction levels from within the same conceptual framework, so that at any time, programming, reasoning, assessment of invariants, transformation of code, etc., will be done with the "right" level of abstraction, no more, no less. Computerized tools based on such a paradigm could automatically and robustly enforce the coherency between the many representations of a system being studied, from within the meta-system.

As a common case of systems being studied, we can deal with layered implementations, where the implementation of an abstract system A with a concrete system C is done with many conceptual intermediate systems $S_1 \dots S_n$, and a tower of interpretation functors $\Phi_{i,j}$ from S_i into S_j. Because different invariants require different points of view, the "tower" may

[5] For instance, to assess the correctness of a given program to be run on an embedded device, one may first check the algorithmic correctness of what it computes, using a very abstract view of the program, then one may assert resource boundedness using a resource-aware representation; one may then want to consider delays and cache misses to prove the real-time behaviour of the system. While debugging the high-level parts of the system, it is perfectly normal to add debugging side-effects (logging behaviour), that would be incorrect from a security point of view, or to disable optimizations that would be required from a performance point of view. Things that are irrelevant or worse, a nuisance, when studying certain aspects of a program, are the very essential stuff to consider when studying other aspects, and vice versa. The point is: what you consider a relevant effect or not is a matter of point of view, of level of abstraction, etc. Some effects are relevant when you see things with the right magnifying loop, or from the right angle; some are relevant only when you use a telescope and point it to the right place in the sky.

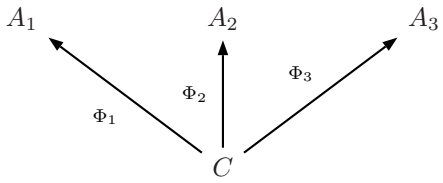
not even be linear. For instance, there may be several equivalent ways to decompose a same implementation in intermediate layers, as in the below diagram, where of S_1 implements S_5 in a way that can be decomposed in terms of intermediate implementations S_2, S_3 and S_4, or S'_2, S'_3 and S_4:



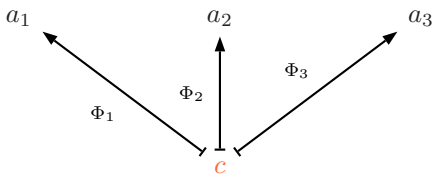
4.8 Aspect-Oriented Programming

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

Using our conceptual framework, Aspect-Oriented Programming can be formalized as follows. Programs in a concrete programming system C are specified through several "aspects" in abstract systems $A_1 \dots A_n$:



A metaprogram w named the "aspect weaver" is meant to take the aspect specifications a_1 to a_n and turn them into a concrete program c:



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 A_i. 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 A_i as to what are valid computations in the intended system. With such a formalization, it is obvious that cross-cutting (or absence thereof) is inherent in

the decomposition of a system into aspects, and that up to isomorphism, a system can be decomposed into aspects in various ways such that the aspects be orthogonal or cross-cut in a wholly different manner.

One way in which the constraint between aspects is typically specified is by describing the aspects as separate activities that interact through the supervision of a controller; This kind of specification is useful for the purpose of developing the aspect weaver, where the weaver is said controller, and most things happen at compile-time. However, aspects are useful only in as much as there is a simpler "declarative" way of specifing their interaction that is more abstract than the invocation of a specific weaving algorithm.

A useful tool for specifying the cross-cutting of aspects in a declarative way is join-points. Join points can be formalized with functors from some of the systems A_i to a common system J. The *arrows* of J are the join points, and represent abstract (parametrized) events that happen during a computation. Each system A_i then specifies separate sub-computations that are meant to happen in case of such an event. A controler still needs to be specified as to the way these sub-computations are scheduled and interact with each other, and might be specified as a activity that executes in parallel with aspects and dispatches messages to them according to the events that happen, with the aspects A_i themselves being able to raise new events.

Another way of seeing things is to consider the aspects $a_1 \ldots a_n$ as constraints on the specified program c, and the aspect weaver as as constraint solving metaprogram. Indeed, custom Aspect-Oriented Programming systems can be built quite effectively by writing metaprograms in a suitable logic programming framework [**?**].

Chapter 5

Metaprogramming

TO BE CONTINUED HERE...

> I'd rather write programs that write programs than write programs.
>
> — Dick Sites

5.1 Intent

Up to this point, we have studied implementations as given logical entities outside and above the considered computing systems; they were obscure input parameters to logical statements that assess correctness properties of these implementations and combinations thereof. Our next concern is to consider implementations themselves as computational objects that can be built as part of computing systems. In other words, we want to consider the "logic" with which we discuss computing systems and their interrelationships (implementation or otherwise) no more as a black box at a "superior" level outsidethe world of computing, but as a computing process of its own right. We will begin by justifying this desire with a quick taxonomy of possible applications.

To be able to express an implementation, we will need a framework where the computing system being implemented and the computing system implementing it will be data objects manipulated by a computing system at a level above them: a "meta-level" computing system, as distinguished from the "base" system the implementation of which being expressed. The distinction between the base level and the meta-level is formalized as the problem of *representation*, with concepts allowing discourse about the base level computing system being reified as *data* manipulated by the meta-level formal system.

Programming at that meta-level in general, and manipulating (base-level) programs in particular will be called "metaprogramming". Metaprogramming can be used for many tasks beyond implementation, including but not limited to all the intermediate tasks that may be useful to implement the base system, and any kind of programming tasks that may ever be of interest to programmers. In other words, the meta-system is a computing system of its own[1].

[1] This is why it is a better practice to design a metasystem as a full-fledged computing system with a particular extended functionality to manipulate code, than as a code representation extended with ad-hoc transformation tools. Indeed, *XXXX Compare C macros, C++ templates, Lisp macros, camlp4, Prolog or Lisp metacircular interpreters. Issues of runtime vs compile-time, load-time, coffee-time, etc.* And should there be specific strict security or termination issues with respect to metaprograms, they can be better solved by considering them as constraints about the design of a suitable metalanguage than by growing random features in which to write

Since a meta-program can be considered as a program in some computing system of its own, it may also be manipulated by a meta-program that is in an according meta-level — the meta-level of the meta-level is called the meta-meta-level, and its programs are meta-meta-programs; and so on for further elaboration in ever deeper meta^n-programming. Note that contrary to a common — and, we think, misleading — usage, we like to think conceptually of the meta-level as being *below* the base level. Indeed, it is arguably foundational to the base-level rather than elaborated on top of it[2].

Move this as final remark of the meta-programming chapter, or first remark of the next Sometimes, the meta-level computing system in which metaprogramming happens and the base-level computing system that is being metaprogrammed will be somehow "the same", for some notion of sameness to be specified on a case-by-case basis. In these cases, the system being its own metasystem is said to be "reflective".

To express metaprogramming in its full generality, we will require a universal framework in which any computing system can be represented as data and manipulated. This framework can itself be implemented as part of a computing system; by virtue of the universality of the framework, said computing system can itself be represented within the framework, and is thus reflective — possibly in many different ways. However, universality proves to be a more interesting property than merely as a tool to achieve reflection, and has more applications. [3]

Reflection is useful in conjunction with universality. It is always conceptually possible to have but it isn't constructible a posteriori internally within a system known to be universal. Indeed, unless

metaprograms until they don't fit the constraints anymore.

[2] The foundational nature of the meta-level with respect to the base-level can be argued with the following metaphor: the meta-level is used to specify the semantics of the base-level; the meta-system has to exist before the base-system; when implementing a system with a meta-system, the base-system sits on top of the meta-system. When implementing a system, it is always possible to dig deeper foundations that have more layers of meta-depths, but there is a quickly decreasing return on investment to dig deeper foundations to the system being built. It is always possible, though at a cost, to dig below an existing system so as to change and enhance its foundations without modifying the system, just like it is possible to rest on the existing foundations to modify the existing system or to build a new one.

With such a metaphor, one can also wonder about what could be a boat that floats on a sea of liquid changing semantics, *a semi-portable program that automagically adapts to a variety of small semantic discrepancies in the underlying target computing systems, such as a portable Scheme program.* or a submarine that preserves its integrity deep below the surface, *embedded programs written in the lambda-calculus or some other virtual machine, (the "hull"), that has implementations at each of many meta-depths; useful for an inspector or debugger that can dynamically adapt to the semantic depth required to provide the user with a proper view.* or a plane that takes off some solid ground area to land on another, *code migration and translation accross languages.* or a lighter-than-air *higher-order code in some abstract version of pure typed lambda-calculus.* or a hydroplane, etc. *portable migratable code such as the Tube* Of course, the metaphor has limits, too, especially since it only takes into account one dimension of relationship between computing systems.

We can trace the origins of this upside-down association of "meta" with "high" to metaphysics, metaprogramming, and other meta-level sciences looking like lofty abstract theological discussions about things that don't matter unless you're already high in a hierarchy relevant to that activity. Actually, quite the opposite is true. Just like metaphysics is about the foundational questions of existence, that matter for everyone at every moment. Of course, not everyone needs to become a specialist in metaprogramming, just like not everyone needs be a specialist in metaphysics, just like not everyone needs be a specialist in bakery, motorcycle maintenance, beer brewery or haircutting. It's a specialized task, like everything, and most people are best off knowing just enough of the general principles to not do anything obviously counterproductive when interacting with the work of specialists.

Thus what matters, economically and socially, is not the lofty "conceptually above" reputation of metaprogramming as much its being "implementationally below": infrastructure that serves developers,

[3] *TO MOVE SOMEWHERE DOWN THE REST OF THE MAIN TEXT:* the universality guarantees that it is always possible to solve problems each with a language that expresses exactly the abstractions required by the given problem, rather than having to suffer the impedance mismatch between problem and language that is inevitable in systems with only one abstraction level, whether they are reflective or not — thus avoiding both the risks of being desperately abstract and slow, and of being overly low-level and drowned in details.

you have some "magic" help from the outside, you can't determine internally which of the infinitely many systems that you can express is "the" current one, which representation adequately fits the current system. And indeed, the only to be sure that a system representation accurately describes the current system is to cheat, so as to make the system fit it description as much as the description is based on the system. This is especially true in dynamic (as opposed to merely static) systems that dynamically evolve or change or grow variants, and you want the representation you'll use for current system to evolve with the system, so as to track these changes.

Just like universality is useful beyond a tool to implement reflection, reflection is useful beyond a mere trick to illustrate universality.

5.2 Taxonomy of Metaprograms

Taxonomy is the death of science

— A. N. Whiteheadd

We can classify pure static metaprograms, by giving them a type in a variant of the simply typed lambda-calculus with suitable extensions: a type P for program representations, a type D for generic data. To simplify things, P is included in D (through a canonical injection j) and programs can be thought as being of type $D \to D$, through an evaluator $eval : P \to D \to D$.

Actually this setting is essentially the same as what is studied in recursion theory *(cite Kleene, Smullyan)*, except that recursion theorists go through the pains of systematically encoding functions as data through $eval$, and bookkeeping their actual type (or types, when there are many, with puns being made possible by j being implicit) aside.

Note that this setting exhibits the subtle difference between metaprograms and higher-order functions: because they operate on program representations instead of programs as black boxes that can only be invoked, metaprograms may look "inside" the functions being processed, whereas their functional arguments remain opaque to higher-order functions. To illustrate this point, here is a list of general ideas for metaprograms, each with a simple type. This list was taken from our earlier article [**?**]. Note that other taxonomies than the below have been proposed for metaprograms [**?**], based on XXX

XXX Examples XXX

5.3 The Necessity of Metaprogramming

Chassez le naturel, il revient au galop

— XXX

Note that the functions that go from data to algorithmic behavior can very well be achieved with higher-order functions (by writing an interpreter, or a "compiler" that composes proper higher-order combinators); what can't be done with higher-order function is to see inside a functional argument, to decompose an arbitrary algorithm merely by watching its results (though there exist subclasses of algorithms for which such reflection principles hold). Of course, by coding programs in a way that explicitly "declare" just the information needed about themselves for the purpose of intended metaprogramming effects, it is possible to achieve the same features as provide metaprograms without actually resorting to metaprogramming; moreover, with suitable higher-order combinators (if expressible in the programming language), it is possible to keep a "direct style" of programming while doing so. However, this technique of keeping "just

enough" information for given metaprograms to be written as normal functions is of course doomed to entail complete rewrite of the whole program if a new metaprogramming effect is ever to be invoked that requires more (or different) information than was previously encoded. To allow arbitrary metaprograms to be run, the developers will end up keeping "all the information" about their programs, which amounts to making the source code available as well as a precise mapping from source to object code. At this point, the system may as well be an interpreter for "scripts" written in a language with metaprogramming facilities. But this has a cost as compared to using a language with builtin metaprogramming facilities: a naïve implementation will incur quite a negative performance impact on the resulting code; an efficient implementation that be robust enough for deployment purposes will require to spend lots of expensive talents and resources; there remains a trick consisting in having developers redundantly provide separate versions of their code, one being oriented toward efficient execution and the other being oriented toward program manipulation; but in such a setting, ensuring the consistency between declared semantics and implicit programmed semantics becomes a maintenance nightmare. The conclusion is that while you may manually cope with the absence of a metaprogramming framework to achieve given metaprogramming features, you cannot escape a robust metaprogramming framework to achieve these features with any kind of performance, robustness and human cost effectiveness.

Arrows in the base system as nodes of in the meta-system. Cartesian Closed Categories with Internal Cartesian Closed Category...

Things like PCLOS, etc. : extend existing languages not only with new features, but also with new semi-automatically managed aspects.

5.4 Representation

semantic representation..... data vs code. Data is "all there". It has trivial semantics. It is passive. Recursively enumerable, decidably comparable.

[4].

Jean Goubault's lambda-eval-quote calculus. Write-only QUOTE (=boxing inside lambda) vs source-code pattern-matching (intermediate states are possible). Lisp SEXP and camlp4 grammars.

Language Extension

ASTs (Abstract Syntax Trees) are great for referentially transparent languages — or rather, for a referentialy transparent description of the language's semantics. But in any other settings, it is necessary to carry from the surrounding context so as to make sense of a parse tree. Even the semantics of Lisp macros depends on an "environment". Now, for arbitrary extensions in the language's semantics, a macro could potentially require an arbitrary extension to what is contained in said "environment".

i.e. when all relevant grammar attributes are synthetized rather than inherited. Of course, higher-order attributes that are ultimately called with the right top-level environment can make do for inherited attributes.

Extensible grammars. Inherited attributes: take expectations into account when macro-expanding.

[4] My school experience as math buff made me particularly frustrated with the fact that some particular things that had trivial cost in math cost so much in computer engineering: changes in representation.

5.5 Programming as Implementation

Programming consists in defining the semantics of an application in terms of the semantics of the programming language being used. A program is thus the implementation of an abstract system A, the application, with a concrete system C, the programming language's execution environment.

In as much as the behaviour of A consists in reacting to environmental input by enacting proper output action, then the program can be viewed as an interpreter for the language of input actions I to the language of output actions O. This is the "using as programming" paradigm, through which all uses of a computer are seen as programming. (The difference between a user and a programmer is that the programmer knows that there is no difference between using and programming.)

5.6 Building Domain-Specific Languages

Centaur is a dedicated system meta-language. Compiler-building tools.

5.7 Internal Extension

Incremental building of DSLs. [?]

Object systems built as internal extensions to LISP [?], FORTH [?].

Paul Graham's On Lisp [?]. CLOS, MEXP, Screamer, etc.

FORTH: Mops, OpenBoot; low-level and non-standard.

Poplog: pop-11, prolog, commonlisp, ml.

Chapter 6

Reflection

One person's data is another person's program
— Guy L. Steele, Jr.

6.1 Intent

"Loops" in Metaprogramming system. (As usual, formal mathematical approach: fix-points.)

Reflection is about a computer system maintaining an effective model of itself, within a its model of the real world.

If the model is effective, it can become an essential tool. cf. emergence of the notion of conscience, as described by Hofstadter; however, we do not claim that "reflection" is enough to ever reach the end of such a process; we only suggest that it may be necessary early on.

John Harrison ... in systems of formal logic [?]. Lev Balishev [?].

Linear logic as combinators for monadic computing systems.

Cite all the people in ☞ `http://tunes.org/Review/Reflection.html` *?*

6.2 Reification

The issue of source code reification.

KWOTE.

Internal semantic reification of functional objects [?]. Must loose one restriction among the following... Be actually lower-level than advertised, and/or include non-determinism.

6.3 Self-Reference Demystified

XXX this paragraph needs work: A bit more formally, given a "base" formal system and its formalization in some "meta" system, it is often possible, if the "base" system is expressive enough, to find internal data structures within the base system that "reflect" its formalism what is usually seen as programs with implicit semantics in a "base" system. Viewing code as data in the "meta" system allows to manipulate it in arbitrary ways for a great variety of purposes, whereas in the "base" system, you can only either execute it or ignore it[1].

[1] Reflection thus depends on code being open and available for inspection, manipulation and modification, rather than considered as black-box opaque systems. A semi-formal discussion of reflection and its relationship

Same reflective self-reference viewed at various meta levels: base, meta, meta-meta; internal, base, meta; internally internal, internal, base.

6.4 Meta-Circular Interpreters

Turing machines, LISP 1.0, about any language can do it. mind special I/O. eval in null-env.

Causal connection: the fact that the implementation at one level is also a correct implementation at another level. Going "down" indefinitely.

6.5 Introspection and Intercession

dynamic typing were there was static typing. eval in current (global) env. eval in current local env?

Self-Observation, Self-Modification incompatibility between correctness, completion, and determinism [?].

6.6 Procedural Reflection

Tower model: 3Lisp, etc. Graph model: PLISP [?]

Maintenance and partiality.

hen thinking about self-representations in reflective systems, it's important to bear in mind that they are just that — *representations*. The causal connection, and in particular the computational effectiveness which it supports, can lead to a confusion between the representation and the mechanism which is represented. Similarly, the metaphoric relationship drawn between reflective systems and mechanical ones — a story in which mechanism is "exposed to view", and in which users can "reach in" to effect changes — can also contribute to this confusion. When thinking of the design of a MOP-based system, there are at least two important aspects of the representation *qua representation* to be borne in mind; *maintenance* and *partiality*.

Maintenance refers to the way in which the representation is actively maintained by the reflective system. Elements of the representation are created as needed, and/or maintained in correspondence with elements of the system itself, rather than being continually present. The lazily-created reflective interpreter layers of the 3-Lisp implementation illustrate this. While the 3-Lisp model guarantees that the representation is always available when requested, it may not actually exist *until* its requested. At the point where it is created, the elements of the representation (or rather, of any instance of the representation) are a rationalisation of the system's state according to an idealised model. so, when designing the model, and considering the terms in which the meta-level interface is cast, it's important to remember the distinction between "exposed structure" and the *actual* implementation mechanisms; a distinction that the system maintains actively.

A second critical property, which follows from the maintenance of the representation, is its inherent *partiality*. The purpose of the representation is not to provide an absolute, decontextualized or impartial description of the system's activity. Rather, the representation describes some aspects of the system's behaviour for the purposes of some domain of expected behaviour. It reveals certain aspects of behaviour, and hides others; similarly, it supports certain forms of tailoring and modification, but not others. The representation is a *designed* artifact; and, in line with perceived needs and expectations, the designer sets the bounds on the flexibility which it embodies.

---

with source codpe availability can be found in our 1999 paper "Metaprogramming and Free Availability of Sources" [**?**].

— Paul Dourish [**?**]

6.7 Compile-Time Reflection

Macros. LISP 1.5, Common LISP, etc. Napier88, CRML, OpenC++, OpenJava, Pliant, camlp4...
Many systems with an open compiler are somehow compile-time reflective.
Static vs Dynamic compile-time reflection

6.8 Language-based Reflection

Does not exist yet.

Chapter 7

Expressive Power

7.1 Illative Computational Logic

Fcite Curry self-fulfilling completeness of reflective systems [**?**] *XXX contact specialist of computability about that*

7.2 Beyond Computability

[?]

Peter Wegner...

not everything is in the programmer's head

Even if the target must specifically not be reflective, if it is a static one-off piece of software, where all information is to be given before-hand (because it is to be statically checked with an external tool; because it is to run on an autonomous embedded system; etc.) reflection is useful in the metaprogramming environment for that target system. And even then, the reflective capabilities can be kept in the front-end, for debugging, upgrade, etc.

Game theory: interaction between system, user and external world. probabilistic errors in user interaction.

7.3 Expression vs implementation

The *expressive power* of a language is not the same as its *implementatory power*. [1] Matthias Felleisen [**?**]

The possibility of implementing Haskell in C, and conversely, does not the least imply that both languages be of equivalent expressive power.

The implementation power of a language enbodies the class of behaviours that can be described in the language as complete programs. Within the framework of classic proprietary software development, that may be the only thing that matters, to a first approximation, *and*

[1] A promising way to distinguish them is through probabilistic models for programmer code modification, including programmer mistake. Since expression takes into account statically or dynamically enforceable contracts with respect to incremental code modifications, it can considerably enhance narrow the set of checkably correct programs, allowing to make humanly probable useful programs that would be extremely improbable (infinitesimally probable) in presence of human errors. Such models could be checked, albeit at a some non-negligible cost, against real-life software development practices in various companies, from a software engineering point of view.

that could be why forty years of proprietary software have conditionned computer scientists into not caring about any kind of expressive power beyond that. .

The language could be "write-only", and "write-once", implementative power wouldn't be less. Expressive power is the information you can exchange using the language with other people with whom you work. When programming is an incremental undertaking, what you care about is expressive power. That's what people need in the world of free software computing. To be very expressive, the language must be "read-write" and "write-indefinitely". For instance, a language to describe finite state machines operating on an indefinite tape (Turing Machines) can implement any one-shot computations from integers to integers in asymptotically optimal space and time. But as a tool for interprogrammer communication, it is not nearly as expressive as Cayenne that allows to describe arbitrary functions from arbitrary higher-order types to any other, including the ability to define statically enforced logical invariants. In Cayenne (which compiles into Haskell, which can be compiled into machine code, bypassing C for everything but for system interface and runtime support), you can define a type whose elements are precisely all sort functions, excluding any buggy function that sometimes fail. The language can thus express much more than C or any other implemented language, for that matter.

That reflective systems have no more implementatory power than other "Turing-equivalent" systems is quite a good sign: it means that we can actually implement reflective systems with our usual kind of non-reflective universal programming languages.

7.4 Information Flow

When considering implementations, inputs correspond to universal predicates, and output to existential predicates. They must thus be explicitly distinguished by the formalism.

Stability flow: to transactions on some abstract objects correspond transactions on the concrete objects that implement them. In a concurrent system, it's important to master the "stability flow" so as to maintain a sound system.

7.5 Universal Systems

Does there exist a Universal System?

C isn't a universal language. When you formalize the concept of "Turing-equivalence", you can see many reasons why it isn't. John Tromp [?].

... are universal systems with respect to metaprogramming and metareasoning... Such systems are, in particular, computationally and logically reflective. But their power does not reside in this sole amusing side-effect (that may be and is often used to prove or disprove fixpoint theorems); their power resides in this universality, that allows any structure to be modelled, manipulated, and executed inside them, in a way as easy as physically possible (but not easier) with respect to intrinsic complexity...

Not a chaotic enumeration of capabilities; instead, a way to unify existing capabilities within a same framework. Whatever is "primitive" is a question of point of view — choosing between points of view is not a CS question but a SE question.

TO BE CONTINUED HERE...

non-determinism: Mitch Wand [?] showed that the (equational) theory of F-expr is trivial; that is, if you can completely access and deconstruct the source code of an expression at runtime, then (obviously), you can always distinguish expressions that do not have the exact same source code. Well, thanks to non-determinism, we can build a variant on F-expr, the R-expr, the equational theory of which is exactly as trivial or as elaborate as one wishes, within some interesting limits. Indeed, if

we only allow access to the deconstructed source code of a compiled expression up to replacement by a provably equivalent expression, *then we can have an equational theory of R-expr that be about as elaborate as we wish. The limits to constructing such R-expr is in the choice of the equivalence up to which the system can substitute an expression for another.*

non-determinism as incomplete self-information. Elaborate schemes to cheat at the meta^{n+1}-level, by maintaining the information that the meta^n-level believes is the most accurate while having the additional information that describes how things really *happen at the meta^{n+1}-level.*

To promise not to use additional information: wrap the program into an equational rewriter that removes the information while preserving the abstract semantics. Have a standard idiom with this declared semantics.

How to express side effects??? (in a concurrent/distributed system) reification principle: "if you can enclose it, you can reify it".

cite Articles de Reflection 96; Bawden: quasiquote; Bawden: macros?

Add "dynamic vs static point of view" in the introduction. Move personal rambling to foreword Split in parts.

Plan: - intro, plan. To actually discuss implementation internals, must open the box and discuss programming at the meta-level metaprogramming not just done within implementations. natural and artificial barriers. - static approach: ad hoc crippled transformation languages (macros, templates) compilers as black box. but still, compilers have internals. - representation (static pov: data structure; dynamic pov: computing system) abstraction level from text to semantics; inevitable gap - static vs dynamic pov "should we stick to a computational representation?" doh! dynPOV: we always have a computational system; the question is how much of the target system's semantics is implicit (and relieves us from thinking about it), and how much is explicit (and allows for expressing transformation). Making something explicit without allowing transformations == useless PAIN. - metaprogramming. staged computation. - dynamic approach: same generic language for metaprogramming (plus API) as for base-level programming. compile-time reflection (API may be higher or lower level). mixing stages = runtime-reflection

Expressiveness: input, output, control, state

Chapter 8

Architecture of a Reflective System

Foo!

— bar

Starting from the conclusions of our previous chapter on CS. Software engineering rather than CS point of view; or rather, a dynamic point of view of computer system programming.

Paradigm of interaction between human programmers and the computer system. Universal system for such a paradigm, to maximize expressiveness. [Though tuning parameters vary depending on relative knowledge and strength of various humans and computers, and help identify the shape of the universal system (defined up to transformation, with quite some liberty).

8.1 System Architecture in Concurrent Systems

Include architecture1.tex et architecture2.tex

8.1.1

Must consider...

Ultimately, this is software engineering.

Conclusion: No kernel.

8.1.2 Security

A programmer wants to program his application in as high a level of abstraction that suits his needs as possible. However, security is an overall property of a system, that can be defeated by people having access to effects that are hidden by the abstraction level for which the secure system was designed. A system that is "secure" for some abstraction level might not be secure anymore when attacked with lower-level effects that evade its abstract model of security.

Integrity Breach Through Invisible Effects

When security is specified at too high an abstraction level, invisible low-level effects can be used breach the integrity of data.

A well-known trick to access data that is meant to stay secret by using lower-level effects than considered by the security model is measure the power consumption of a smartcard while it is multiplying two numbers. Indeed, multiplication is typically done by repeating the loop of adding the multiplicand to the result if the multiplicator's low bit is set then shift multiplicand left and multiplicator right. Since adding consumes more current than not adding, one can thus easily identify the bits in the multiplicator on the power consumption curve. *cite*

Some password systems, safe locks, etc., are also known to have been cracked by measuring the time, number of cache misses, etc., needed to reject a login attempt. *cite*

In a purportedly secure system that accepted as extension arbitrary code that is "provably secure" according to some abstract model of security, it is similarly possible to have certified code actively reveal secret data and breaching the system's integrity. By using low-level effects such as delays and cache misses, the innocuous-looking program could communicate secret data on a subliminal channel while still respecting abstract security invariants.

This would be a non-trivial feat, and might slow things down enough to be noticeable. On the other hand, when spying cryptographic communications, it is often only needed transmit a 56- to 256- bit session key once in a while, so if one manages to interleave the subliminal message with normal use, there is no need for a high bit rate. A program with more access rights on the same system could observe this information and relay it on other channels, possibly with subliminal channels on other media.

Temporary Tampering

In a setting where security is specified at too high an abstraction level that ignores some hidden side-effects, "provably secure" code could also temporarily alter variables between stable states.

Malicious code that can communicate with external programs and interact with them as if it were in a stable state, can for all practical purpose alter sensitive variables. It might have to restore the safety invariants before next stable state is reached, but by postponing reaching such a stable state long enough (or indefinitely), it can take control of the system's sensitive data until it is too late.

And of course, linking to external code that is not proven correct at all, or removing safety checks for "efficiency" reasons can allow malicious code to enter an otherwise secure system.

Secure multi-level architecture

The possibility of these attacks does not mean that it is impossible to secure a system with multiple abstraction levels, or that such a system is intrinsically less secure than one with just one abstraction level. On the contrary, security is something intrinsically complex, and factoring the system into levels of abstractions allows to keep it as simple as possible, though no simpler.

Preventing Lower-Level Effects

Preventing lower-level effects: ensure that all code except some kernel is written at higher-levels of the system; audit the kernel. This prevents malicious software from running, but doesn't prevent loss of integrity through lower-level observation.

So that the compiler doesn't introduce insecurity, have a macro-expansion model of compilation from high-level to low-level, together with a set of allowed transformations (potential optimizations) that reduce the set of potentially problematic effects, never increases it.

8.2 Reflective Development Methodology

> The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.
> — Fred Brooks, Jr.

There is an intrinsic complexity in software development, and an extrinsic complexity. Reflection allows to define a process to minimize extrinsic complexity.

8.2.1

> Software development consists in incrementally debugging the empty program.
> — (inspired by) Ehud Shapiro, Algorithmic Program Debugging

find original citation by Ehud Shapiro

8.3 Expert Systems

8.3.1 AI vs IA

> The meta-Turing test counts a thing as intelligent if it seeks to apply Turing tests to objects of its own creation.
> — Lew Mammel, Jr.

When such thing as basing a computing system on an expert system is considered, many people are upset and start expressing their fear and disbelief with respect to the possibility of an Artificial Intelligence (AI). They contend that an AI is impossible and that therefore, such an "AI" approach of things is doomed to fail.

Now, these fears stem from a fundamental misunderstanding of what expert systems are and try to be. Actually, despite their legacy as systems intended to build an "AI", expert systems can be seen as tools that can incrementally enhance the behavior of systems facing complex situations: they needn't ever achieve a state of autonomous "intelligence" (hopefully, someday, they might, but they needn't); they may serve as a tool for Intelligence Amplification (IA), that extend the reach and power of the human developers' intelligence, by automating "trivial" tasks and "grunt work", and thus letting the human focus on the important things.

cf Kimbly vs Kennel debate on comp.lang.misc, january 2001. As an example, it has been suggested that Expert Systems be used to determine the proper low-level encoding for some high-level user-specified abstract data structure. Some compilers already do some of it using such techniques as deforestation [?], and other formalized techniques are known to enhance cache locality and decrease memory-to-disk swapping in programs "crunching" large bodies of data [?], etc. People who oppose this reality will invoke the uncomputability of Kolmogorov complexity to argue that optimization is impossible. But, the word "optimization" is a misnomer: the "optimizer" doesn't try find the "best" possible implementation, merely one that is allegedly much better than a naïve interpretation, and hopefully good enough for the task at hand.

As Peter Norvig puts it [?], even though known optimization algorithms have dismal worst-case behavior, in practice, there most often exists a good enough algorithm among known ones, particularly The problem is to find a good enough one with an affordable cost, as small as possible (in economic term, considering overall human work and capital).

If these people's argument was truly valid, "optimizing" compilers wouldn't be possible either; yet they are possible, and in daily use. These people are blinded and fail to see that "optimizer" is a marketing term that refers to a program that searches for a better result (as compared to what a naïve approach would yield, in practical situations), not for the "best" result (which would require a sensible framework to make such absolute statements, to begin with). Finding "good enough" program can be done as a routine task most of the time. Just like so called "optimizers" (who's the <censored/> who originally misapplied this term?) do not and cannot produce "optimal" programs, but only "fairly good" programs, by whatever dynamic standard emerges from competition. Just like Solomonoff induction is impossible, yet brains can pretty well induce properties from loads of samples.

Thus, "Expert System" is a matter of point of view on software: existing compilers already do with their hardwired algorithms and heuristics a lot of things that used to be to be done in expert systems only. The difference is that with expert systems, you consider expertise as dynamically, incrementally, enhanceable knowledge, whereas compilers are generally intended as static entities with fixed knowledge. Now, in our architecture, you can dynamically compile code that will thereupon remain static, so that you can get the best of both approaches: the dynamic evolvability of expert systems, and the efficiency of statically compiled compilers.

The specific data structures used in programs come as a result of an enormous history of deep human thought and progress. But this doesn't mean that any particular hacker really understands this history, or that any program really takes advantage of it all. Indeed, few people really master the wide diversity of algorithms relating to any particular subject, including their tradeoffs and optimization opportunities; and even fewer people master the algorithms relating to several subjects (exceptions like Donald Knuth receive Turing Awards). For each subject, there might be a handful of them at most worldwide, and even they will fail to be consistently good at programming more than very few projects at a time. Unless you want to restrict the number of programs written in the whole world to a handful, there is quite a room for expert systems that will automate the application of program-writing knowledge to the "optimization" of common programs.

The ability to combine formalized knowledge from many domains, if implemented, might thus enable the development of a large body of multi-domain software that was previously too complex or expensive, involving the cooperation of too many very specialized experts, even though this formalized knowledge in each domain is only a small part of what experts know.

AI-completeness of the task? Some might contend that for the expert-system approach to actually yield the best possible results in all cases is an AI-complete task (i.e. a satisfying expert system existed, one could extract an Artificial Intelligence out of it, so that the system cannot exist before AI is realized, if ever). However, "best" results are hardly ever needed, so there is no requirement for a strong AI. "good enough" results is what is required for the approach to be useful, and pretty stupid systems are sufficient for that, as prove all the tools that have already been developed and used to help us develop programs, instead of having to switch binary code directly into the machine's memory, as seen in computers from the 1940's.

Good example target for metaprogramming expert system: Garbage Collection.

cggc (gaga): GC is typically a domain where people will manually propagate a lot of constraints all over the place. Moreover, efficient implementations rely on "puns", tightly matching the encoding of tags and the implementation of locks with the available instruction set and addressing modes, and on lots of tweaking with encoding, buffer sizes, etc. Finally, it has to be part of an integrated multi-layer design. All these just call *for automation.*

8.4 Software Deployment from a Reflective Point of View

Applying the above-described methodology to software deployment strategies.

With a reflective system, there is no more need for binary compatibility, except in very marginal fringe cases necessary for bootstrap and interchange: there needs be a set of base meta-protocols used to dynamically negociate a common protocol between parties.

This is about Communication Protocols in general, not just communication of executable code.

8.4.1 Analyzing existing strategies

Binary Compatibility

understood as: at the Hardware-Executable Machine-Code Level

> As long as there is proprietary software, there will be binary compatibility; as long as there is binary compatibility, there will be x86 and similar horrors.
>
> —

Programmers often complain about the inefficiency, complexity and/or lack of foresight of legacy hardware architectures (PC compatibility) or communication protocols (SMB, IIOP, XML-RPC), etc. Indeed, it is easy, with hindsight, to see the shortcomings of such legacy solutions as currently dominate the market; yet, by the mere inertia of the need to be binary-compatible with them, bit for bit, these legacy architectures and protocols persist, and prevent progress through the advent of new protocols. It is interesting to understand whence stems this inertia, and what positive counterparts make us accept to pay this price, so that we could devise a system architecture that would bring the same benefits without having the same problems.

Obviously, binary compatibility is a technical measure that is as much beyond the understanding of the most numerous customers who participate in establishing its supremacy, as it is despised and superfluous to the lead technicians who complain about the limitations it imposes. Binary compatibility is thus not a goal per se, but a means to a goal, a device by which customers and technicians alike fulfill a goal that actually matter to them. This important goal is *the perennity of digital information*, that is, the robustness of programs and data alike against changes in the execution environment due to diversity in space and time. Users, whether neophytes or specialists, want to not have to care about technical details of hardware and software compatibility. *Caring about compatibility, like caring about much anything, costs a lot of time that they prefer to spend solving their own problems instead.*

Binary compatibility is a solution to the problem of having to care about hardware and software compatibility. When a set of computers are binary compatible, programs written for one can run on another; when a set of programs are designed to conform to the application binary interface for an architecture, they will run on any computer that provides said architecture. binary compatibility architectures thus provide a reference standard such that hardware manufacturers can produce hardware without having to worry too much about software support, software publishers can write software without to having to worry too much about hardware support, and computer users can easily buy hardware and software, either as a combination or as an independent upgrade, without having to worry too much about compatibility. Out of competing solutions, the market will quickly select, by a network effect, the economically most attractive solution, which is a priori distinct from the solution that experts would deem "best" on technical merits alone.

if the architecture that supports the common binary compatibility is widely supported enough and perennial enough, programs written for one can be acquired once, and deployed

many times over, without loss of either processing capability or accumulated data. Programs are assured to run at a partner's or a customer's, and to still run after an upgrade; conversely, computers built to standard specifications are assured to run anyone's programs, as long as they respect the compatible application binary interface (ABI).

The principle of binary compatibility — impliedly "at the hardware executable level" —

Indeed defines a standard interface on both sides of which clearly defined competitive markets can evolve with the mutual insurance that the other side is strong.

a lower-level compatibility that leaves part of the architecture unspecified introduces friction in the market (CP/M, graphics in PC-DOS,);

reducing friction due to conflict of interest between providers and users of Interface... ABI...

In the legal context of proprietary software, where vendors deploy black-box binary programs to the mass consumer market, and users are never sure they will be able to or want to get or afford binary upgrades (or whether the "upgrade" would suit them), binary compatibility is just a fact of life — the only way to ensure perennity of software.

However binary compatibility has many drawbacks. Mainly, it induces great inertia in software infrastructure, which prevents progress. That is, it forces any new version of the system to support all interfaces exactly as is. *forces low-level interfaces, which means different programs either inefficient marshalling or insecure sharing; compatibility with low-level programs means that only trivial invariants hold, so you can trust noone to respect your high-level invariants; choice of ABIs is mostly an ignorant precocious commitment; ....*

Now, this binary compatibility is not the only technical solution to the problem of software perennity, and other techniques already exist that palliate all its shortcomings. The justification behind the extreme dominance of binary compatibility has to be sought in legal issues, not in technical issues. However, the legal context of proprietary software sold for the masses isn't the only context for software deployment, nor might it survive in the long run.

Source Compatibility

Source compatibility takes an opposite point of view. The source code, as worked on by the original developers, is made available for users so they can taylor them to their changing needs. Proficient users can modify code themselves, and other ones can hire experts to do that for them. This supposes a completely different legal context with respect to software development than mass-licensed proprietary software.

Technically, ...

Fix the bugs rather than work around them. When they really must be worked around, do it in a taylored specific way. Contain bugs.

The more standard the language, the more compatible the implementation, and the less work is required for porting. In the best cases, recompilation happens seamlessly.

The work of making source adaptions can be nicely packaged and shared at large.

However, source code without the right to modify it is mostly useless; it is even more useless than binary code that you can hack. Source code whose modifications you cannot share is prohibitively expensive, because you can't share the cost of modifying it. Indeed, with Free Software operating systems like GNU/Linux, FreeBSD, etc., source-code compatibility has attained a high-level of stability and ease-of- If the source code is Free Software *find proper citation* rather than simply "shared source", *find proper citation* then

Pre-compiled, pre-packaged binaries (debian, rpm) Compilation at installation-time (c-l-c).

Virtual Machines

....

Features moving from software to hardware, from hardware to software. Floating-Point calculations were typically done in software until FPU became the norm, and are now done in hardware. On the other hand, the RISC concept has been to move many complex instructions from hardware to software. The whole m68k-based architecture was done in hardware in early Macintosh computers, and was done in software and in latter PowerMacintosh machines; the same happened for the x86 architecture with PC emulators, and with the recent Transmeta series of processors, where software serves to translate the x86 instruction set into hidden native instructions. *find proper citation*

It takes some finite time (though not negligible) to rebuild executable "native" code from virtual machine code. This can be done on the fly, with dynamic translators, JIT, etc. Or this can be done in advance. It's all a matter of cacheing. Generic techniques can be applied to evaluate the right behaviour, considering expected tradeoffs.

Transparent recompilation by the system is way of achieving the speed of native code compilation with the portability advantage of Virtual Machines. ANDF made it an explicit install-time feature, so did the Windows NT/Alpha x86 compatibility layer (optionally, for speed); Java JIT compilers make it an implicit run-time feature.

So as to be really portable, virtual machines require a language with well-defined semantics, like Java, as opposed to C. The well-definedness requirements are even stricter if dynamic run-time mobility of data (or better, code), and not just static launch-time portability, is required. This is to be decoupled from the requirement of being high-level or low-level, that gives more or less freedom and responsibility for implementers to develop sophisticated efficient translation layer between the human specification and the machine implementation. x86 as a VM – CISC instructions hardware translated RISC core, or software translated to a VLIW core.

users care that their airline booking systems still work, that their payroll systems pay them the right amount every month, that their pacemakers beep at the appropriate intervals – Dan Barlow

this is a one-time thing recompiling is done once, when upgrading hardware and recompiling can be *fast* only tight loops need be superoptimized e.g. the ocaml compiler is relatively fast

theran notes that all major Lisp implementations have a facility to precompile *method dispatch* because users of production systems don't want to wait for cache misses.

people are ready to wait 4 hours for their windows to install on their upgraded hardware. Why wouldn't they wait 4 hours for their old games to recompile?

theran- gimme a break. Every time I update Linux to a version with yet another ABI, shit breaks left and right for weeks.

wli- It's fairly easy, in a large application, to chew too much CPU and too much RAM, and end up in a situation where no single thing is the bottleneck, esp. after a round of going and identifying "tight loops" and superoptimizing them.

wli- The thing will be big and slow and unwieldy and there is no single culprit. if you chew up too much RAM, then using interpreted bytecodes for most of your program is a big win dan: this is because they're other people. its actually quite normal for them to have other priorities

besides, nothing prevents you from superoptimizing. even less in the background, while you run, and/or while you're idle. high-level bytecodes have better code density than assembly, and can save cache misses. wli- Code density is generally not the problem. The problem is usually data. So what if you've got 20MB of code when the data set is 3GB? Fare- if you have bad datastructures, then superoptimizing won't help. In any case, you don't need binary compatibility. Actually, you better have source compatibility, so you can recompile your source with a compiler that knows about optimizing data structure representation.

Fare- binary compatibility is a low-level way of achieving upgrade-irrelevance to software deployment. what the users care is this irrelevance, not the means to it. JVM bytecodes achieve

the same hardware-irrelevance, and are accepted just as much. What users want is to copy around, archive and install packages, without having to worry about low-level compatibility details. binary-compatible hardware achieves that, at the cost of limiting hardware options. source-compatible software achieves that, at the cost of having to hack the source once in a while. the crux of compatibility is not hardware/software – it's about what was once software is now hardware – FPU operations.

JVM bytecodes sure won't bring you performance. But Oberon/Tube bytecodes could. instead of having a virtual-machine with byte instructions, they encoded compressed high-level parse trees that they compile on the fly. no semantic loss, no performance loss. if ever you can get so much performance with oberon, you can get as much with their portable bytecode. the same principle applies to whatever language you want. They call it "slim" binaries – only the essential performance gain as compared to what? it IS native code generation! Because I want to go the other direction. Not away from the machine, but toward it. wli- Sounds to me like you have the overhead of generating code at runtime in addition to running the program. the portable bytecode is just a portable ENCODING for high-level programs. the programs get compiled to native code. the oberon people did it at load time and showed it could be done faster than the disk or network could feed bytecodes. matju- the JIT stuff could also work "on install", or a cached-to-disk variant of "on load". besides, Java is giving a bad name to JIT, because efficient JITs must decompile their bytecode into high-level code before to recompile it down to native code with slim binary, no need to decompile. moreover, you can 1) cache precompiled packages 2) use "fit" binaries instead of slim binaries.

except it can be done in a declarative way that the system handles for you, with a fallback in slim binary form, should your platform not have special fat tactics wli- machine code generation operates on basic blocks, so you basically have an expression dag where the live-outs are the roots of the dag and the live-ins are the leaves.

The essence of software deployment

Three independent problems.

compatibility ultimately only means that ... the best-defined the semantics, the easier is compatibility

efficiency – can itself be divided into complementary (not opposite) run-time and compile-time problems.

run-time efficiency – problem of adaptation between declared semantics and underlying hardware. The highest-level the semantics, the most opportunity for the adapter to make things efficient. If the declared semantics are too low-level, any gap between it and that of the underlying hardware makes implementation slow (abstraction inversion).

compile-time efficiency – the compromise is opposite however, compile-time can be reduced by cacheing (worst case: recompile at every run – interpreter; next worst, recompile at every load – JIT; then recompile at install, on every computer – then recompile at distribution-making, on and for all).

compromise of defining the abstraction level of the declared semantics.

If you look at it carefully, just any compatibility is virtual machine, and virtual machine is always binary. Even source code has a binary ASCII representation with which you're compatible. Explicitly compiling it before execution is a matter of explicit or implicit cacheing infrastructure for efficient execution.

According to a different point of view, ultimately, everything in a computer is binary code, yet you want to interoperate in compatible ways. But this isn't "binary compatibility" as we know it anymore. This is just interoperability. The binary code isn't the rigid "don't touch,

don't even look inside" immutable chunk of vendor-supplied code it used to be – it's a trustable intentional object whose pattern of bits evolve with circumstances.

8.4.2 A Reflective Approach

Principles of Automation

- The only incompressible user action is to express the intent of using some piece of software.
- Anything detail beyond that expression of intent must be automated away from the user's care, in as much as it can be done in a way efficient enough so the user won't want to work around it.
- Advanced users who have more specific needs and know how to precisely enough describe them must be able to do so as efficiently as possible.
- Implementation decisions should be taken dynamically and automatically, after sufficient information is available to take them; not statically and manually when onl wild guesses are possible.

FIT binaries

FIT binaries are not FAT binaries; they are not Slim binaries. FIT is just a combination of both: have the software be generally Slim but add a tad of FAT in hot spots. That is, provide a portable high-level semantic description of the program, but also add architecture-dependent highly-optimized version of small parts of the program, so as to solve performance bottlenecks. These FAT portions can depend on fine-grained runtime configuration (precise processor model and stepping, availability of extension boards, various hardware or software options), and are typically precompiled with a expensive optimization settings or even hand-tuned in assembly.

Such techniques are already done in unsystematic, ad-hoc ways that provide little portability and are not automatically reusable, for such things as graphics libraries in PC games, with bummed *find proper citation* versions of the rendering code depending on available vendor-specific instruction set extensions (MMX, 3DNow!, etc) or graphics accelerator hardware.

The interface between Slim and FAT portions of code abide by some common ABI.

Protocol Negociation

With FIT binaries, we saw that the execution infrastructure had to have some sophistication in choosing how to implement such and such part of some software, rather than blindly jumping into a program, dispatching between several programs depending on the architecture, or using a black-box compiler. Once you accept the principle that the execution infrastructure can have any sophistication in selecting implementation strategies, you realize that there is no reason to stop here.

Reflective Architecture

What we're left at is a Meta-Protocol for protocol negociation, and a base protocol for bootstrapping the negociation (that needs be simple and straightforward rather than super-efficient).

That's automating the choice abstraction level

Obstacles to implementation

The Oberon Tube people have implemented their Slim binaries; *find proper citation* it could be argued that Microsoft's .NET infrastructure follows this principle. FAT binaries are implemented in MacOS 8 and 9. Precompilation of binaries is implemented in WindowsNT/Alpha. Negociation protocols are as old as telnet. *find proper citation* Efficient search of an implementation path is trivial in Prolog, and is the basis of many instruction selection algorithm within compilers. *find proper citation* All these techniques already exist. Combining them would be a mild engineering feat, but would by no means be considered conceptually difficult by anyone.

The obstacle to their deployment is not technical – all of these techniques if it ever was conceptual, it is no more, by this very chapter; the real obstacle is legal: proprietary software makes it binary compatibility almost a necessity, and biases the market strongly toward a dominant architecture.

The residual Free Software market seens both its objectives and means shifted away from concerns that would lead to reflective solutions:

- the dominance of binary-compatible hardware due to proprietary software provides an easy way-out for free software developers seeking compatibility, and makes it uneconomical to seek better long-term solutions.
- non-modifiable proprietary software forces ignorant early commitment.

We start from the specification of compilation of a programming language, and elaborate towards specification of reflective programming systems. We then generalize the approach towards specification of correct metaprograms, and revisit existing techniques used in the field of "reflective" and object-oriented programming.

8.4.3 A MOP based on Implementation Theory

Note that the above axioms do have an operational interpretation.

Safety corresponds to interpretation of synchronization points (reifying the abstract state of the computation).

Soundness corresponds to achieving synchronization (sending a signal to the implementation so it synchronizes).

Completeness corresponds to implementing evaluation strategies (useful for interactive debugging and existential program proofs alike).

Liveness corresponds to detecting advance of computation (useful for committing choices

For every *aspect A that our concrete system C is implementing, (including independent aspects, including synchronized aspects), we have an function $\phi_A : C \to A$ and the primitives corresponding to*

> Reality is that which, when you stop believing in it, doesn't go away.
>
> — Philip K. Dick

Chapter 9

Conclusion

9.1 Contributions

> 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*, is the interest of a theory you've tested to be valid.

Résumer clairement ce qui a été établi, et ce qui n'a pas été établi.

9.2 Comparison with other works

9.3 Perspectives for Further Research

Un système réflexif comme pré-IA...

9.4 Political Aspect

La question se pose: si les techniques réflexives ont tant d'avantage, et sont déjà largement connues, pourquoi ont-elles suscité si peu d'intérêt relativement aux autres techniques ? Dans un article précédemment publié [**?**], nous avons établi qu'un facteur politique jouait fortement contre l'investissement dans ces techniques: la P.I., "Propriété Intellectuelle", qui, en encourageant les entreprises à dresser des barrières légales empêchant l'accès à leurs logiciels, rendent largement inutiles des techniques consistant précisément en la manipulation dynamique de logiciels, et promeuvent un style de programmation "bête et brutal" peu propice à la manipulation subséquente des programmes par d'autres programmes. Cette tendance va sans doute disparaître dans les années qui viennent: avec le succès croissant des logiciels libres, publiés dans un esprit de liberté opposé au "Protectionnisme Informationnel", les techniques de métaprogrammation seront de plus en plus appréciées, et tôt ou tard, l'intérêt des systèmes réflexifs apparaîtra clairement. Si nos explications sont justes, nous prévoyans alors un "changement de phase" dans l'opinion publique, comparable à celui qui a lieu actuellement en faveur des logiciels libres contre les logiciels exclusifs; du jour au lendemain, le "réflexif" sera à la mode, et tout le monde en parlera, tandis que l'industrie migrera peu à peu, en commençant par les projets non excessivement alourdis par la compatibilité avec l'existant, et où la capacité des techniques réflexives

à gérer le changement ou la complexité sera plus directement utile: systèmes répartis, systèmes multi-aspects, développement incrémental à long-terme.

9.5 Last Words

Detailed Credits... in Appendix?

Appendix A

Formalisms used in this Thesis

In the body of this thesis, we used several formalisms that deserve clarification, not because they depend on any conceptual novelty, but because they rely on conventions that, though they may be accepted within specific communities, are not universally known and used.

A.1 Category Theory

cite some book of Category Theory

A.1.1 Category

A category is a set of nodes and a set of arrows between pairs of nodes, such that for any arrows $f : x \longrightarrow y$ and $g : y \longrightarrow z$ there is a composed arrow $g \circ f : x \longrightarrow z$ (also notated fg), and such that composition be associative and have an "identity" arrow $id_x : x \longrightarrow x$ for every node x, with which composition is idempotent (i.e. $f \circ id_x = id_y \circ f = f$).

A.1.2 Functor

A functor is a morphisms from a category to another category, that is, an application that preserves the structure of the source category.

A.2 Partial and Non-Deterministic Functions

A.2.1 Binary Relations viewed as Functions

Functions that are not forcibly total will be said to be partial, and functions that are not forcibly deterministic will be said to be non-deterministic. Herein, when we say "function", we'll mean "partial non-deterministic function". In our non-deterministic framework, traditional "functions" are called "deterministic functions"; traditional "mappings" are called "total deterministic functions".

An interesting approach is then to reuse and extend the classical paradigm of functions-as-relations, where functions are identified to binary relations of same graph. Actually, since we accept non-determinism and partiality in functions, we now have an exact isomorphism between functions and relations, whereas this was not the case when only deterministic functions were accepted:

Given a binary relation R from E to F,

R is *deterministic* iff $\forall x \in E\ \forall y \in F\ \forall z \in F \quad xRy \wedge xRz =: y = z$

R is *total* iff $\forall x{\in}E \quad \exists y{\in}F\ xRy$

R is *injective* iff $\forall x{\in}E\ \forall y{\in}E\ \forall z{\in}F \quad xRz \wedge yRz =: x = y$

R is *surjective* iff $\forall y{\in}F \quad \exists x{\in}E\ xRy$

R is *bijective* iff if it is deterministic total injective and surjective.

Moreover, the *inverse* of R is the binary relation R^{-1} from F to E such that

$$\forall x{\in}E\ \forall y{\in}F \quad yR^{-1}x \Leftrightarrow xRy$$

Taking the inverse of a binary relation is an involutive operation, and that a binary relation is deterministic if and only if its inverse is injective, while it is total if and only if its inverse is surjective.

We then identify a partial non-deterministic function with a binary relation of same graph: the graph Γ_T of a function or binary relation T is defined[1] such that:

$$\Gamma_T = \{\langle x, y\rangle \in X{\times}Y \mid T : x \mapsto y\}$$

That is, a function ϕ is identified to the relation $\overset{\phi}{\mapsto}$ that relates its inputs to their respective outputs (if any), and a binary relation R is identified to the function that maps elements of the first set to those of the second set to which it is related. In other words, a function ϕ and a relation R are identified if and only if

$$\forall x\ \forall y \quad y = (\phi\ x) \Leftrightarrow xRy.$$

Only, the scripture $y = (\phi\ x)$ above is misleading at best in a non-deterministic context, since the expression $(\phi\ x)$ (application of function ϕ to argument x) may take several values, unless ϕ is deterministic; thus we have to refine the concept of equality.

A.2.2 Background History

While partial functions are of common use in recursion theory [**?**, **?**], and have been cleanly expressed in formal systems [**?**], non-deterministic functions as such have been mostly ignored, to the point that the word "function" is generally understood only for deterministic functions.

We can trace back the explicit formal use of non-determinism in a programming language to John McCarthy's ambiguity operator [**?**], although it looks like people involved in Algorithmic Information Theory [**?**] have long studied non-deterministic Turing-Machines and suches. An interesting review of the use of non-determinism in algebraic frameworks was made by Meldal and Walicki [**?**].

As far as actual computational systems go, it looks like there is a taboo against breaking the foundational theorems of confluence in λ-calculi and most works deal with "don't care" non-determinism, that is clustered in ways such that it is not relevant to the result of the overall computation. Practical implementations generally focus on deterministic semantics, too, since traditional monoprocessors are intrinsically deterministic. However, non-determinism does appear in even the most formalized programming language standards, where for instance, the order of evaluation of arguments is not specified [**?**], [**?**]. However, languages challenging the usual imperative, functional, or "object-oriented" single-threaded programming paradigms have developed an approach to non-determinism as being a fact of life. So is it with the Logic Programming paradigm as found in languages such as Prolog, Oz, Mercury, or in the

[1] Actually, which of graphs, sets, functions, relations, or total mappings are "more primitive", and which are defined from the others, depends on the formalism in use, and is of little importance as long as the properties that interrelate them hold.

portable CommonLISP extension Screamer [?]. So is it with Rewrite Logic [?, ?]. So is it with asynchronous programming, in distributed systems [?, ?] or even just interrupt-based input/output. *Dijkstra's THE [?] ???*

In a previous paper [?], we studied how partial functions and non-determinism were essential concepts in the study of reflective systems.

A.3 A Graphical Formalism

In the relevant sections of our thesis, notably in chapter ??, we used a graphical formalism instead of classical formulas made of sequences of symbols. We considered that such a two-dimensional formalism was more appropriate to explain the concepts at hand than the usual one-dimensional formalism, and allowed not to burden the text with complex formulas, while preserving a mathematically precise formalism. Here, we explain how to interpret our graphical formalism in terms of sequential formulas.

A.3.1 Arrows and Labels

Names

The nodes and arrows in our diagrams will have names, that can be arbitrary mathematical expressions. If multiple arrows or nodes in a given diagram that have same name, then the multiple occurrences refer to the same arrow or node in the considered category.

Nodes

The most basic objects are nodes in a category, represented by a name, as follows:

$$X$$

$$Y$$

$$Z$$

The above diagram thus means that we consider three nodes X, Y and Z in some category implicitly obvious from the context.

The position of a node in a diagram does not have any meaning a priori, although we'll later define conventions so as to make diagrams more readable.

Arrows

We can then express relations between nodes with arrows. In the simplest case, a function from the set X to the set Y is represented, as follows:

$$X \longrightarrow Y$$

The shape of the arrow, as defined in section "Arrow Properties" A.3.2 below, will specify additional properties for arrows.

Arrow Labels

We can optionally specify a name for the arrow, with a label placed on the left hand side of the arrow when going from the base to the tip:

$$X \xrightarrow{\phi} Y$$

The above diagram represents two sets X and Y (or nodes in another category than the category of sets), and a function from X to Y named ϕ.

Here is the very same diagram, with X on the right, which is possible, since node placement doesn't matter. Notice how the label is now under the arrow, since this is now the left side of the arrow:

$$Y \xleftarrow[\phi]{} X$$

Category Labels

We can also optionally specify the name of a category (or subset of a category) in which an arrow is specified to exist, with a label placed on the right hand side of the arrow (opposite to the optional arrow name label):

$$X \xrightarrow[C]{} Y$$

Note that the category label may move as the arrow changes direction, symmetrically to the arrow label.

Associations

Given a function or relation ϕ from X to Y, it is possible to specify that ϕ associates element y in Y to element x in X, by an association arrow as follows:

$$x \xmapsto[\phi]{} y$$

Note that the name of ϕ is in the position of the category label. If there were a name in the name label position, it would indicate the pair (x, y), as member of the relation ϕ.

A.3.2 Arrow Properties

We can specify properties on arrows by using recognizable traits in their graphical representation. This traits can be combined, to specify multiple properties. Absence of a trait doesn't mean that the arrow can't actually have said property, only that the property isn't either required as part of the hypotheses or established as part of the conclusions.

Functoriality

When in a category C, we can specify that function ϕ is a structure-preserving functor rather than a mere function, by representing the arrow with a "functor tip" as follows:

$$X \xrightarrow[C]{} Y$$

Injectivity

Function ϕ can be specified to be injective by drawing an arrow at its base as follows:

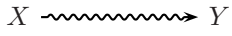
$$X \rightarrowtail Y$$

Surjectivity

Function ϕ can be specified to be surjective by doubling the arrow at its tip as follows:

Non-determinism

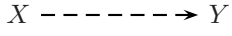
Function ϕ can be specified to be non-deterministic by giving it a waved shape as follows:



Actually, non-determinism is the weak property to assume by default, whereas determinism (i.e. every element in the from-set has at most one image in the to-set) is only to be assumed if the arrow is straight, concave or convex rather than waved. That is, a waved arrow may actually denote a deterministic function (although this property of determinism is not specified by the diagram), whereas a straight arrow may not denote a non-deterministic function (the diagram specifies determinism for the given arrow).

Partiality

Function ϕ can be specified to be partial, by drawing it with a dashed line, as follows:



As with non-determinism, partiality being a weaker property than totality, it is the actual default property, whereas functions are only specified to be total mappings if they are drawned with uninterrupted arrows.

Curved arrows

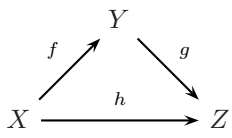
Some arrows will be drawn curved, as follows:

This doesn't imply any specific property for arrows. It is just done so as to make diagrams easier to read, particularly when there are many arrows from X to Y, or when arrows or their labels would otherwise be drawn at the same place.

A.3.3 Simple Assertions

Commutative Diagrams

Every diagram can be read as the assertion that said diagram is commutative. Hence, the following diagram means that $h = g \circ f$:



Simple Implications

In a given diagram, we will use color to distinguish between hypotheses and conclusions, so as to be able to express simple implication rules.

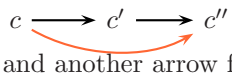
Nodes and arrows in black are the hypothesis or left hand side of the assertion. Arrows in color are the conclusion or right hand side of the assertion. Note that when printed or displayed on a black and white output device, color may be replaced by a light tone of grey, or with some kind of shading.

Free variables in black are universally quantified. Free variables in color are existentially quantified.

Thus, for instance, the following diagram means that between any nodes X and Y (in category C), there exists an arrow ϕ in category C from X to Y:

Note that X, Y and C are in black, whereas ϕ (both arrow and label) are in color. Note how this rule is consistent with function labels in an association arrow being in category position.

As a more elaborate example, transitivity of arrows in a category can be expressed by the following figure:



That is, given an arrow from c to c' and another arrow from c' to c'', there is an arrow from c to c'' that makes the diagram commutative.

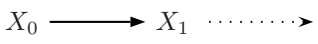
As a last remark, note that a black arrow can only link black nodes, and that conversely, a color node can only be at the base or tip of a color arrow. Hence, the following diagram is valid, but it would make sense if the arrow were changed in black:

It means that in considered (subset of) a category $<$, for any point x, you can find a point y such that there is an arrow in $<$ from x to y. I.e. with proper additional hypotheses, $<$ defines a strict order with no maximal element.

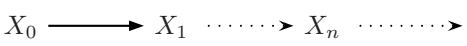
Ellipses

Sometimes, we will use dotted arrows to indicate that there we consider an sequence of arrows of indefinite length. These arrows are meant to be in the same category and have the same properties as the arrow just before them in a line, if applies, or else, are meant to have "usual" properties implicit from context.

Hence, the following diagram means that we consider a sequence $X_0, X_1 ... X_n ...$ of nodes with a functor from each to the next:



The following diagram means about the same thing, but gives a name to a particular element in the sequence with a sufficiently high index n, which might be used as part of the other side of the implication:



A.3.4 Diagram Conventions

A.3.5 Naming Conventions

Naming conventions for nodes... for arrows... for set elements...

A.3.6 Arrow Flow Conventions

Vertical bottom up direction as going from concrete to abstraction... Horizontal left to right direction as evolution in time from past to future... Multiple possible future as branches (might be thought of in a third dimension)... examples from actual diagrams with backlinks...

Appendix B

Bootstrapping a Reflective System

> You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program.
>
> — Alan Perlis

This appendix and the next one ought to have been chapters of this thesis. However, in absence of a working prototype, it is but mere speculation, and cannot be presented as part of a scientific thesis. Still, we felt it might be interesting to our readers were we want to go, so we included them as appendices.

Après avoir théorisé la possibilité, les modalités et les avantages d'un système réflexif, il nous reste à en exhiber un qui satisfasse à nos critères. Développer un système réflexif à partir d'outils externes ne participant pas de ce système (et n'ayant en général pas eux-mêmes d'affinités particulières avec le système à construire) s'appelle traditionnellement bootstrap du système[1]. C'est l'exercice auquel nous nous sommes prêté durant cette thèse.

Citer notamment les travaux de Jacques Pitrat [?] et Jean-Luc Dormoy [?]; le Dragon Book fait remonter aux milieu des années 50 la notion d'amorçage, avec les premiers compilateurs, et cite [?]; les diagrammes en T sont attribués à [?].

B.1 General Principle and its constraints

The static point of view is to consider the problem: "What will be the structure of the completed system?" Once this structure determined, the system is decomposed into a set of modules, each module is implemented with existing tools, and when they are ready, they are assembled into a complete system. This point of view considers metaprogramming tools as given things, that cannot or should not be substantially enhanced, but are only at best a choice to make early on

[1] Dans l'optique statique classique, l'amorçage d'un système s'arrête au moment où un compilateur du langage statiquement spécifié a été obtenu et réécrit dans ledit langage. Dans l'optique dynamique, le système étant en constante évolution, l'amorçage peut être considéré comme un processus continu d'enrichissement du système à partir de versions antérieures; pour un système réflexif, cet enrichissement permet de rendre superflus les outils méta externes jusqu'à ce qu'ils soient réduits à leur strict minimum, à savoir le matériel[2].

on peut toutefois distinguer une phases initiale d'avancement plus aigüe durant laquelle le système restreint considérablement l'étendue du méta-système externe sur lequel repose son implémentation.

(if not imposed). This point of view perfectly fits small projects, and isolated people a closed world where tools are black boxes purchased from third parties with whom it is very difficult to negociate the direction of tool development.

The reflective dynamic point of view considers that the problem is more of "what is the structure of the development process?". It considers the system not as a programming object to complete, but as a programming process to evolve. The development process, rather than the goal system, is divided into modules, with earlier ones having a significant influence on how latter ones can be developed. ... and et de partir de la décomposition obtenue de ce système en un ensemble de modules, pour écrire chacun de ces modules et les assembler en un système complet. Le point de vue dynamique est qu'il faut prendre en compte le processus de développement, car selon l'ordre de disponibilité des composants, le développement des composants suivants peut se trouver simplifié voire rendu possible; il est même possible envisageable que certaines décisions de développement ne soient prises qu'après qu'une certaine expérimentation aient montré le bon choix à faire, voire aient suggéré le type d'évolution souhaitable.

La problématique dynamique de développement d'un système consiste donc en la donnée d'un point de départ et d'un ensemble de contraintes sur le but à atteindre; à partir de cette problématique, on détermine dynamiquement le composant qui semble avoir le meilleur rendement, compte tenu de ses avantages amortis par rapport à son coût de développement étant donné l'état actuel du système.

Le point de départ de notre système est l'ensemble des outils de développement librement disponibles sur l'Internet. Les systèmes d'exploitation libres Linux ou BSD fournissent ainsi toute une infrastructure de développement qui est relativement standardisée, largement compatible avec de nombreux systèmes, bien documentée, soigneusement implémentée, corrigibles en cas d'erreur, et adaptable aux besoins spécifiques des utilisateurs. C'est donc naturellement de nombreux implémenteurs de langages de programmation et de systèmes de calcul ont choisi cette infrastructure pour développer leur système, et que de même nous l'avons choisie, enrichie de tous lesdits langages.

La direction vers laquelle nous voulons faire évoluer le système est celle d'un système réflexif qui serait universel du point de vue de la métaprogrammation de systèmes répartis (par delà ce qu'offrent Erlang ou JoCaml), et qui serait complètement amorcé au-dessus du noyau Linux (comme Pliant ou Oberon), voire du matériel PC (comme Retro, NativeOberon, ou l'intégrale de GNU/Linux ou d'un BSD).

According to the golden rule "eat your own dog food", the system will have to be actually used, as much as possible, during its own development. C'est de toute façon l'essence même du point de vue dynamique de développement logiciel que d'utiliser les composants précédemment développés pour créer les composants suivants.

B.2 Plan d'amorçage

> Greenspun's Tenth Rule of Programming: any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

Le coeur d'un système réflexif de métaprogrammation est son infrastructure de transformation de code, dont la première fonction sera de compiler des programmes d'un langage de haut niveau vers du code exécutable par la machine. Cette infrastructure sera utilisée de façon omniprésente dans tout le reste du système, et est donc à développer avant le reste. Le sous-but suivant sera de développer un système interactif capable de faire profiter au mieux l'utilisateur de

cette infrastructure pour développer des programmes, et tout particulièrement des programmes concurrents ou répartis.

En s'inspirant de la structure des compilateurs existants, on peut diviser a priori de manière grossière le compilateur en plusieurs passes, correspondant à la traduction entre des couches d'abstraction du système. Les interfaces entre ces passes du compilateur seront autant de langages de programmation de plus ou moins haut niveau. Distinguons notamment, du plus concret au plus abstrait, les langages suivants:

- L0: binaire directement exécutable
- L1: (macro)assembleur
- L2: langage de bas niveau à la C--
- L3: langage algébrique élémentaire (LISP non sûr)
- L4: langage algébrique de haut niveau (LISP + filtrage)
- L5: système expert basé sur des règles de réécriture

Notons que ces langages évolueront avec la structure de compilation, bien que le but même de leur définition est l'espoir qu'ils évolueront relativement lentement par rapport au reste. Si tant est que des motifs apparaissent au cours de cette évolution, ils pourront être explicités dans le système en factorisant ces langages en modules et foncteurs exprimant directement ces motifs.

Étant donné comme but intermédiaire l'établissement de cette infrastructure de compilation, reste à déterminer un plan pour le développement de cette infrastructure. Étant donné les interactions entre composants, voici le plan adopté:

1. *Amorcer un compilateur.*

   Durant cette étape, il faut choisir un métalangage M, qui soit adapté à l'écriture d'un compilateur, et qui soit lui-même assez une cible facile pour la métaprogrammation; les candidats naturels sont CommonLISP, Scheme (avec des extensions idoines), OCaml (avec camlp4)[3]. Il s'agit d'obtenir une flèche $L4 \xrightarrow[M]{} L0$. En fait, on amorce d'abord la partie haute du compilateur $L4 \xrightarrow[M]{} L3 \xrightarrow[M]{} M$; puis, on peut écrire la partie basse à l'aide du langage haut, par une flèche $L3 \xrightarrow[L4]{} L2 \xrightarrow[L4]{} L1 \xrightarrow[L4]{} L0$. À la fin de cette étape, le système, tournant au dessus du métasystème M, peut compiler du code à tous niveaux d'abstraction, de L4 à L0; accessoirement, du code de haut niveau L3 ou L4, peut tourner au-dessus de M s'il suit certaines restrictions.

[3] Un langage comme C ne convient pas, étant d'un niveau d'abstraction très bas qui rend malaisée l'écriture d'un compilateur, tout en ayant de nombreuses limitations le rendant malaisé comme cible d'un compilateur (voir avec C-- les efforts pour créer un langage qui conviendrait à ce dernier usage). C++, s'il permet entre autres un style de programmation de plus haut niveau, hérite toutefois la complexité d'aspects bas niveau de C (notamment concernant la gestion mémoire) qui lui conservent sa lourdeur comme métalangage pour un compilateur, et complexifient sa sémantique au point de rendre douteuse son utilisation comme cible pour la métaprogrammation. C et C++ ont tous deux un modèle mal défini et peu utilisable en ce qui concerne la concurrence, et sont donc très éloignés du modèle que l'on cherche à atteindre lors de cet amorçage. Le langage Java conviendrait beaucoup mieux que C et C++, n'ayant pas les défauts précédents; toutefois, son utilisation reste très lourde dans l'écriture d'un compilateur, en comparaison des langages considérés, et l'intérêt principal du langage, sa portabilité, n'est pas pertinente relativement au projet de développer une infrastructure réflexive qui lui serait étrangère.

2. *Amorcer l'exécutif*

   Le code produit par le compilateur précédent dépend de l'existence d'un exécutif pour fonctionner: fonctions d'entrée/sortie de base, code d'initialisation et autres interfaces avec le système, mais aussi gestion automatique de la mémoire, et toute une bibliothèque d'algorithmes variés. C'est à cette étape que cet exécutif sera développé, tandis que le compilateur fonctionne encore au-dessus de l'infrastructure d'exécution du métasystème M. À la fin de cette étape, le compilateur de la première étape pourra donc compiler du code écrit en L4 en un programme exécutable. Or, on s'est précisément arrangé dans la première étape pour que la majeure partie du code soit écrite en L4 ! Moyennant une petite traduction (semi-automatique) du reste du code de M en $L4$, il devient alors possible au système de se compiler lui-même, et d'obtenir un exécutable indépendant de M.

3. *Amorcer l'interacteur*

   Une fois la chaîne de compilation autonome, il devient possible de la compléter en un système de développement interactif. L'intérêt de l'exercice n'est pas dans le développement d'une n-ème interface utilisateur graphique ou textuelle, mais dans la définition et l'implémentation effective d'un ensemble cohérent d'abstractions pour la métaprogrammation interactive incrémentale dans le contexte dynamique d'un système concurrent en cours d'exécution. Ces abstractions comprendront toutes les briques de bases au-dessus desquelles

4. *Amorcer le système de développement*

   Au-dessus des primitives fournies par l'intéracteur,

   La paradigme de développement.

B.3 Le compilateur

B.3.1 Choix du métalangage initial

Notre choix pour le métalangage M s'est porté sur CommonLISP, pour une convergence de raisons.

Tout d'abord, la structure syntaxique flexible des dialectes LISP, permet d'économiser sur le coût de parseurs dans les phases initiales où la structure des langages traités change rapidement. Grâce à cette même simplicité syntaxique, CommonLISP dispose de facilités standardisées pour certaines formes de métaprogrammation dynamique à l'intérieur même du système, à l'aide de macros et de compilation dynamique. CommonLISP dispose aussi d'un large corps de fonctions prédéfinies, dans lesquelles puiser au besoin sans avoir à les implémenter avant. et qu'il ne sera donc pas nécessaire d'impl

OCaml

B.4 L'exécutif

Eat our own dogfood

Partie dure: la gestion automatique de la mémoire. Développement d'un GC: étant donné la fragilité des choix d'implémentation vis-à-vis du reste du compilateur, des applications, etc, le

développement d'un GC efficace passe par le développement d'une infrastructure flexible au méta-niveau, pour adapter les choix d'implémentation du GC au reste du système. On obtient alors un "Méta-GC".

Partie facile, mais lourde: bibliothèque de fonctions diverses. Point intéressant, la structuration modulaire.

Partie technique, l'interface avec le système sous-jacent. Choix Linux.

B.5 The interactor

> Nihil in intellectu nisi prius in sensu
> (Il n'est rien dans l'esprit qui ne passe d'abord par les sens)

B.6 The development system

> In the old days, an operating system was designed to optimize the utilization of the computer's resources. In the future, its main goal will be to optimize the user's time.
> — Jakob Nielsen ☞ `http://www.useit.com/`

The last step in bootstrapping the system consists in completely interning the development meta-system into the interactor.

B.7 Self-extensible languages

TRAC, BLISS, LISP, FORTH, Prolog, Rebol

Appendix C

Building Distributed Systems

Zut si c'est cassé!

This appendix and the previous one ought to have been chapters of this thesis. However, in absence of a working prototype, it is but mere speculation, and cannot be presented as part of a scientific thesis. Still, we felt it might be interesting to our readers were we want to go, so we included them as appendices.

C.1 Domains

C.2 Migration

General mechanism that combines reification, transformation, absorption. The three steps can be advantageously folded in one, by partial evaluation.

Particular cases: GC; Load-balancing; Secure access mecanisms (capabilities, checked contracts).

C.3 Synchronisation

Distributed hand-shake.

Dynamic elimination of useless synchronizations.

Optimistic evaluation until actual synchronization.

C.4 Replication

Basing on synchronization and migration, maintain coherent replicas of an object.

Achieve that at the meta-level so as to make it transparent.

Replay: memoize the interaction history (or a digest thereof). Can also serve for explanation, or for interactive debugging.

All depends on object demarcation (see domain above).

C.5 Interfacing

CORBA, XMI, etc.

C.6 Evolution

> Software entities are more complex for their size than perhaps any other human construct because no two parts are alike. If they are, we make the two similar parts into a subroutine – open or closed. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.
>
> — Fred Brooks, Jr.

Automatic update. Schema Evolution. Conflict Resolution, either interactive or batch (dynamic aspect of interaction vs atomicity).

See how Forman and Danforth use Metaclasses in the SOM model to handle class evolution [?].

C.7 State of the Art

- E ☞ `http://www.ERights.org/`, language for Secure Distributed Computing based on capabilities and "smart contracts".
- Erlang ☞ `http://www.Erlang.org/`, dynamic distributed programming language designed around pure applicative functional threads communicating structured objects through channels.
- Mozart ☞ `http://www.mozart-oz.org/`, a distributed system based on the logic programming language Oz.

Experimental systems (work, but never deployed much) include:

- JOCaml ☞ `http://pauillac.inria.fr/jocaml/`, an extension of the statically typed functional programming language Objective Caml with distributed computing based on the Join Calculus.
- David Alan Halls' Tube ☞ `ftp://Samaris.tunes.org/pub/papers/people/David.A.Halls/`, distributed system achieving transparent code migration by metaprogramming code (CPS transforms) with Scheme+TCP+threads considered as a virtual machine.
- Matthew Fuchs' Dreme
- Kali
- Christian Queinnec's ...
- ABCL/R
- Bawden's thesis

http://techreports.larc.nasa.gov/ltrs/PDF/aiaa-90-3435.pdf

Appendix D

Semantics of bootstrapped languages

Here are the details explaining the bootstrapped languages.

More information will be available on my web site as soon as available:
☞ `http://fare.tunes.org/phdthesis/`.

D.1 L4: LAMP

Level 4 language (L4 dans **??**), LAMP: LAnguage Meta-Programming. It is a functional language meant to manipulate other languages, whose initial primary purpose is to write the compilers in the system.

At its core is a functional language based on rewriting with abstract patterns. *These patterns can express objects in arbitrary computation categories; any pattern is statically tied to its defining category, and dynamically mapped into lower-level resources in a well-founded way.*

As an example of meta-programming features, the language will later be later extended with logic programming and declarative features.

[?]

D.2 L3: KLINK

A level 3 langage (L3 in the **??**), KLINK: Kernel LISP Implementation Named KLINK. A tail-recursive minimal LISP, with a set of core features on which to build the rest of the system.

KLINK features a notion of linear domain, that serves as the basis for a module system,

Minimal LISP dialect with domains.

Pivotal operational point of the system.

It's a high-level operational language (i.e. not very declarative; the operational semantics is clearly defined). Capable of expressing, through reflection, arbitrary invariants (very important point).

Essentially, a graph reduction language.

In the initial bootstrap compiler, all domains are managed by the metasystem's resource managers (GC, etc.).

D.3 L2: TAGLL

Level 2 language (L2 in the **??**): a low-level algebraic language.

Provides an abstraction of the processor.

Explicit linear variables.

Annotations to decompile or edit links with L3.

D.4 L1: LAP

Level 1 language (L1 in the **??**): LAP, LISP Assembly Programming. It is a (macro)assembler embedded in the LISP language.

Besides obligatory recognition of the architecture-dependent instruction set, it is based on a library that manages segments of binary data and symbol tables.

Partial evaluation of patterns by the rewrite system as a way to compile the symbolic assembly pattern matcher into an efficient binary assembler.

Constraint solver to optimize relative location of segments, their content (when multiple options are available), their encoding (when some can be uncompressed or compressed, swapped in or out, compiled in or out), etc. With respect to size constraints, processor mode available for decoding, cache conflict avoidance, etc.

```
   (+ 1 1)
(define (y f)
  (lambda (x)
    (f (lambda (x) ((y f) x))
       x)))
```