Formalizing the Notion of Implementation, as Illustrated with Concurrent Garbage Collection

François-René Rideau ĐẶNG-Vũ Bân

CNET DTL ASR (France Télécom)
38–40 rue du general Leclerc
92794 Issy Moulineaux Cedex 9, FRANCE
http://fare.tunes.org/

Abstract

We study the notion of implementation in a generic framework for computational semantics. This paper, the first in a series, gives the detailed formal presentation of a notion of implementation. We introduce the notion pointing out the issues raised in presence of concurrency, and illustrate our ideas with Concurrent Garbage Collection. We then give a definition for the notion of implementation in a generic meta-level framework for the semantics of computing systems. We formalize usual properties expected from implementations, and particularly find an original notion of "soundness". We show how this notion can be used to solve such problems as modular implementation of synchronized concurrent systems. We relate this formal notion of soundness to many hacker tricks that were naturally used during actual implementation of various systems.

1 Introduction

As computing systems get more complex, especially due to concurrency and distribution, and as they get more used in mission-critical devices, it becomes increasingly important to understand how the high-level services they render may be *correctly* implemented in terms of low-level software and hardware.

Formalizing the notion of implementation may help specify, verify and debug the correct behavior of implementations (open or not), or of metaprogramming tools that produce and/or analyze implementations (most notably static or dynamic compilers). It is thus of interest to people designing or developing such metaprogramming tools, and to everyone wondering how much they can trust the tools they use.

There have been many works on the formal specification and verification of interpreters and compilers [11, 25, 13, 6]. However, these works are restricted to single-threaded centralized programming models and do not generalize obviously to specification of arbitrary parallel or distributed programs. Araujo [2] discusses correctness of a parallel implementation of Prolog, but in a 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 [16, 19], but his single-universe formalism isn't immediately suitable to specify generic notions of metaprogramming or of compiling. Goerigk and Hoffmann [10] give a formal definition for compiler correctness that is essentially the same as we use, but do not systematically develop their meta-logic or study additional properties of implementations.

This paper aims at providing as generic a meta-level framework as possible to internally express the notion of an implementation of a higher-level "abstract" computing system by means of a lower-level "concrete" computing system. Yet, we try to provide a framework that allows to obtain in a reusable and composable way results that be useful to people specifying, implementing, and verifying compilers, especially in presence of concurrency. We do not use a particular logical meta-formalism (computerized or not), but think the presented framework should pose no problem being expressed in any existing formalism.

In section 2, we informally study the difficulty of specifying concurrent implementations, and examine as a sample problem the concurrent garbage collected implementation of a high-level language. In section 3, we propose a generic notion of implementation, in a generic framework for computational semantics. In section 4, we study various desirable properties of implementations, and their composability we particularly introduce an original notion of "soundness" that generalizes many stronger ad-hoc notions found in other works on implementation. In section 5, we show how this notion is a useful tool to establish in a modular way the correctness of the implementation of concurrent systems, such as in the previously considered garbage collector case. We discuss implementation techniques for soundness. We conclude in section 6, suggesting possible applications to the present work.

2 An Informal Approach

2.1 Specifying Concurrent Implementations

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. This correspondance ought to preserve the semantics of the abstract system; this means that computations done with the concrete computing system lead to observations that are consistent with computations that would have been done in the abstract system.

Typically, abstract systems of interest are modern high-level programming languages such as dialects of the LISP or ML family of languages, or more recently, the Java language. Less general cases include programs that are written in such languages. Concrete systems of interest are modern microprocessor-based computers, or pools thereof, as programmed in assembly language. Often, an implementation is split into layers, factoring the correspondance through intermediate computing systems such as virtual machines or widely available low-level programming languages such as C.

The semantics of such computing systems and their implementation is very well understood in the single-threaded case; there have even been proven-correct implementations of high-level languages with proven-correct microprocessors [6]. Sequential monoprocessor implementations of sequential or concurrent languages may be even extracted from formal specifications of their abstract semantics [**?**]. That is, when the implementation is serialized at the low-level, serial synchronization of concurrent high-level activities is a marginal benefit.

But formalizing the implementation of a sequential or concurrent high-level system with a concurrent low-level system is still a research topic. Concurrent low-level system can be a multiprocessor system, a distributed system, or any computer with multiprogramming operating environment. The difficulty is that implementating high-level computing systems involves system services such as automatic memory management (garbage-collection) checkpointing, persistence, transactions, dynamic compilation, code migration, support for source-level debugging, and that every such service depends on global invariants that must be preserved despite the concurrency of the low-level system.

A satisfying theory of implementation must thus be applicable to concurrent computing systems, and include the study the good properties of an implementation relative to the preservation of high-level invariants.

2.2 Concurrent Garbage Collection

We'll try to illustrate the properties we want to study about implementations using the more specific but typical case of the implementing garbage collection in concurrent programming language.

Garbage collection [5, 28], 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, named the garbage collector or also "GC", track down objects in the graph that are "dead", that is, unreachable from any currently user-accessible object, and collect them, that is, eliminate them from the graph, saving representation space for future new objects.

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 horrible death of implemented programs when they try to access an object that has been collected.

Garbage collection is a relatively high-level transformation on object graphs; however, computer memory hardware typically doesn't provide dynamically reconfigurable graphs as such, but statically configured arrays of dynamic data cells according to the Von Neumann architecture. Object graphs may be represented by mapping every node of the graph to chunks of consecutive data cells in memory, and with arcs of the graph being represented as a number in corresponding data cell of the father node, pointing to the first data cell of the son node. A same object graph may have multiple representations as a memory state, depending on the mapping of graph nodes to memory chunks; a GC may safely modify the mapping as long as it preserves the graph. A same object graph may also lack any representation as a memory state, if it contains more nodes that can be fit into the finite memory size. On the other hand, not all memory states are valid graph representation; the encoding of graphs may rely on complex invariants to be respected by well-formed memory states so that GC may happen. For instance, many GC techniques require that the GC algorithm should be able to precisely differentiate pointers from other numbers, so that it may update, swizzle, marshall, or otherwise specially process pointers when moving, mapping, or migrating objects around the implementation-controlled world. 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. The correspondance between abstraction and concrete computation states is thus a *partial one-to-many* relation in the general case, although in some cases, it might be useful to have the implementation be *total*, or to refine its exact behavior until it be *one-to-one*.

Now, GCs are not usually run alone for the sake of it; rather, they run as a system support service, concurrently with actual implementation of the abstract activities that modify the graph, "mutators". And for architectural reasons, it may be plainly impossible, or extremely inefficient for the underlying computing architecture to preserve the GC invariants at all moments; so that mutator operations that are conceptually atomic when manipulating abstract graphs may involve many instructions when concretely implemented. The concurrent mutators may introduce lots of transient states, that do not correspond to any valid abstract graph representation; but the GC activity must only be able to observe the mutated memory when it is in a "stable" state that respects invariants. We say that the mutator implementation is *sound* if it is always possible for the GC to observe it in a stable state when needed. Soundness thus appears as an essential property of implementations so as to safely combine multiple concurrent activities.

We may also study more classical properties of garbage collected implementations of high-level languages. A GC'ed implementation will be *complete*, if it enables the representation of all possible object graphs that may happen as valid states during the abstract computation; 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. A GC'ed implementation will be *live* if it never stops or hangs, if progress of concrete computations corre-

spond to observable progress of abstract computations; this means that GC activity will always give the hand back to mutator activities after a (hopefully short) while. A GC'ed implementation will be *real-time* if there can be a bound on the "while" 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.

Without going into the details of garbage collection, we have already isolated properties that we require from implementation of garbage-collected languages. There now remains to formalize this intuition.

3 A Simple Model for Implementation

3.1 Modelling Computing Systems

Before we can express the notion of implementing a computing system with another, we must first formalize the notion of a computing system. By "computing systems", we mean just anything, from the most universal computing system (such as a full-fledged installation of GNU/Linux) down to the most special-purpose finite device (that may itself be given by a description within a universal system). Turing-machines (universal or not) or variants of λ- or π- calculi [1] are eligible, as are systems constituted by the states of execution of general- or special-purpose programming languages, or by just the reachable states of execution of a given program in some language.

We'll choose the simplest model we can think of that has some characteristic significant structure, so as to get the intuition of what we mean without having to overcome too many details. We feel that the essence of computation is that an operator sets some initial state in a system, and lets the system evolve in a sequence of state transitions; at times, the operator will observe the current state of computations, so as to see possible progress or success of the system.

To formalize such a model, we will consider what set theorists call a preorder: a set with a internal binary relation $\leq$ that is both reflexive and transitive. $\leq$ is sometimes obtained as the reflexive transitive closure $\longrightarrow^*$ of a "step" relation $\longrightarrow$; however, it needs not be, and it is often confusing to try to express everything in terms of atomic steps.

Model theorists who think in terms of stateful abstract machines will like to understand an element of the set as a possible state of the machine, and of the (step) relation as one of (atomic) transition from one state to another. The elements that are $\leq$ to a given state constitute the possible future states into which it may naturally evolve. The computing system can thus be viewed as a transition system.

Logicians may rather consider the states as states of relevant knowledge about the system [12], and the relation as a monotonous refinement or non-monotonous evolution of this knowledge. The elements that are $\leq$ to a given state are possible future states of knowledge about the system in presence of internal though process only. The computing system can thus be viewed as a logical reasoning system.

Semanticians who study programming languages, will like to understand an element of the set as the expression that constitute a program in context, and of the (step) relation as one of (atomic) reduction from one expression to another. The elements that are $\leq$ to a given expression are refined expressions into which it may be rewritten, that have more specialized denotational semantics than the considered expression. The computing system can thus be viewed as a term rewrite system.

Finally, category theorists [20] will consider a preorder as just a category whose objects are the elements of the set, and whose arrows are couples of objects that satisfy the binary relation (i.e. a category where you decide not to distinguish arrows that have same domain and codomain). The objects of the category are the semantic interpretations of the programs of a language, endowed with a preorder structure.

This same model seems to be a common root to most widely-used formal frameworks for defining and analyzing the semantics of computing systems: denotational semantics, operational semantics, rewrite logic, abstract state machines. It may thus be used to model implementations of a system whose semantics is described in one of these frameworks with another lower-level system whose semantics is described in another of these frameworks. Note that this model conspicuously takes into account only *internal* behavior of computing systems, the spontaneous intrinsic relations, transitions, reductions or morphisms that exist between elements, states, expressions or objects of the system. This model could be extended with more structure, but this is not essential to the current study.

In the rest of this paper, we will indifferently use the vocabulary of a set theorist, a model theorist, a logician, a semantician, a category theorist, in an attempt to raise universal mutual understanding. In diagrams, we'll also draw

$$x \xrightarrow{*} x'$$

the fact that $x \leq x'$ and

$$x \xrightarrow{+} x'$$

the fact that $x \leq x'$ and $x \neq x'$. Assuming that $\leq$ is indeed defined from a step relation $\longrightarrow$, we'll also draw

$$x \longrightarrow x'$$

the fact that x is rewritten in x' in one step.

3.2 Implementation

Informally, we want an implementation to be a correspondance between two computing systems, an "abstract" system A and a "concrete" system C, whereby states in a (hopefully large enough) fragment of A are each "implemented" by a choice of one or more states in C. Implementation means that an implementing concrete state should somehow have the "same" observable behavior as the implemented abstract state, although it may do as it pleases when abstract behavior is only partially specified. An implementation may be partial: sometimes, not all of the abstract system is implemented (particularly if said abstract system is infinite, while the concrete system is finite); we will be satisfied with a fragment of the abstract system that be big enough to successfully complete considered computations.

To formalize the notion, consider systems A and C endowed with preorders as above. An implementation will be

[1] There must be some relation between the notion of implementation and that of (bi)simulation, as known in asynchronous process calculi, but cannot say which at the time being.

some kind of partially-defined non-deterministic but injective function from A to C; its inverse function, a partial deterministic function from C to A will be a "partial interpretation" of C in A.

Partial and non-deterministic functions have not been as widely used for formal reasoning as have been their total deterministic counterparts, but have nonetheless been studied and formalized in logical, mathematical or computational frameworks [8, 22, 27]. Set theorists might just consider them as just binary relations; applicative functional programmers may take them as a primitive concept; but category theorists as well as many people will prefer to express them in terms of total deterministic functions.

A partial function f from a set X to a set Y can be easily macro-expressed from within common frameworks of total deterministic functions, as the data of an intermediate set Z of individual associations, and two (total deterministic) functions $\mathtt{an}f : Z \to X$ and $\mathtt{im}f : Z \to Y$ respectively giving the antecedent and image of every individual association. Thus, y is a possible value of $f(x)$, which we write $y \lhd f(x)$, iff there is a z such that $\mathtt{an}f(z) = x$ and $\mathtt{im}f(z) = y$. f is deterministic iff $\mathtt{an}f$ is injective; it is total iff $\mathtt{an}f$ is surjective; it is surjective iff $\mathtt{im}f$ is surjective; it is injective iff $\mathtt{im}f$ is injective. In particular f is total deterministic iff $\mathtt{an}f$ is bijective.

To stay within a framework of total deterministic functions, we'll thus have an implementation be the data of an intermediate set O of "observable abstract states" (or equivalently of "stable concrete states"), with an injection j from O into C (equivalently, we will consider O as a subset of C), and a function ϕ from O to A. We can then say that a concrete state c is stable and corresponds to observable abstract state a iff $a \xleftarrow{\phi} o \xrightarrow{j} c$ which we'll also write more simply as $a \xleftarrow{\Phi} c$. We will use the following diagram

$$A \xleftarrow{\Phi} C$$

to denote the fact that $\Phi^{-1} = j \circ \phi^{-1}$ be an implementation of A with C. Note that we privilege the inverse $\Phi = \phi \circ j^{-1}$ of the implementation Φ^{-1}; in fact, as we'll see later, it is the "interpretation" Φ, not the implementation Φ^{-1}, that preserves structure and deserves to be an arrow.

There remains to give a computing system (preorder) structure on O. Since O is conceptually a subset of C (through embedding j), we may quite naturally define $\leq$ in O such that $o \leq o'$ in O iff $j(o) \leq j(o')$ in C. Set theorists say that we consider on O the preorder structure induced by C, or the reciprocal image through j of C's structure. Category theorists say that O is a full subcategory of C, or that j is an embedding of O in C.

To sum up, when we have such a partial function $\Phi = \phi \circ j^{-1}$ with $\phi : O \longrightarrow A$ and $j : O \hookrightarrow C$, then we say that Φ^{-1} is an implementation of A with C through O.

4 Properties of an Implementation

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 section, we'll consider as defined above an implementation Φ^{-1} of an abstract computing system A with a concrete system C through the intermediate system O, given by the canonical injection $j : O \hookrightarrow C$, and an interpretation function $\phi : O \longrightarrow A$.

4.1 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.

Formally, for any diagram

$$\begin{array}{ccc} a & & a' \\ \Phi\big\uparrow & & \Phi\big\uparrow \\ c & \xrightarrow[C]{*} & c' \end{array}$$

we must be able to complete the diagram into

$$\begin{array}{ccc} a & \xrightarrow[A]{*} & a' \\ \Phi\big\uparrow & & \Phi\big\uparrow \\ c & \xrightarrow[C]{*} & c' \end{array}$$

That is, if by computing from an 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.

In other words, the basic reduction structure of the computing systems must be preserved by ϕ. Set theorists will say that ϕ is a non-decreasing function; category theorists will say that ϕ is a (covariant) functor[2]. Note that by the very construction of the structure on O as induced from C, 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, 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

[2] 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. This justifies our privileging Φ as the object of logical interest.

observation, the implementation yield a correct result. It is always safe 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 not to come with a design, but unsafe to come with a design that might erroneously kill people under intended use conditions.

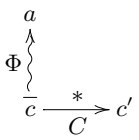
This notion of safety is the same as found in works by Lamport [17]. It is what Goerigk calls partial correctness [9], "partial" corresponding to the fact that Φ be a partial function rather than a total function.

Since safety is such an essential property, from now on, we'll only consider safe implementations. Besides, the categorical formalism we chose to be equally used as others demands structure-preservation from every considered function, anyway.

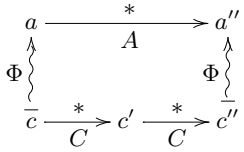
4.2 Soundness

Another important property of implementations is that of *soundness*: an implementation is sound iff any possible concrete transition path is somehow "meaningful", and corresponds to part of an abstract transition path[3].

This can be formalized by requiring that any diagram



may be completed into a diagram



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''.

In other words, the concrete system mustn't have meaningless transitions, whereby it spontaneously go wild or enter a deadlock; instead, starting from a stable state, it must always keep the possibility of evolving into a stable state, of "stabilizing".

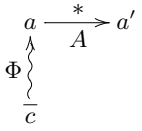
Soundness as such is not a composable property, since given sound implementations Φ^{-1} of A with C and Ψ^{-1} of C with E, we cannot know for sure without an additional hypothesis that the observable state in C that we obtain from invoking the soundness of Ψ^{-1} is itself a stable state with respect 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 (see below). 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.

Variant: We can have various notions of "real-time" and/or bounded-resource soundness, assuming we have a notion of "duration" or "size" of transition paths between two states, by requiring that can be exhibited transition path from c' to c'' be of duration or size less than some maximum admissible response time or more generally than some maximum pre-allocated amount of resources.

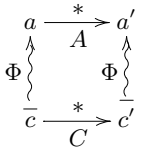
4.3 Completeness

An interesting property that we may require from an implementation, is that of *completeness*, whereby all transitions from a given implemented abstract state are themselves implemented.

Formally, from diagram

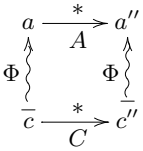


deduce diagram



Note that we do not require every abstract state to be implemented, which is a property that we call *totality* (from a deduce $a \underset{\Phi}{\leftsquigarrow} c$). An implementation that has both completeness and totality will be said to be *faithful*. Completeness and totality are composable properties. Completeness and soundness combined yield a composable property, too. If we look at temporal logic statements on transitions, completeness ensures preservation of increasing "may" statements by Φ.

Variant: Sometimes, we don't require that all possible transitions be implemented, but only that there be at least one implemented transition whenever there is a transition possible from a given observable abstract state. Formally, from same diagram as for completeness above, deduce diagram



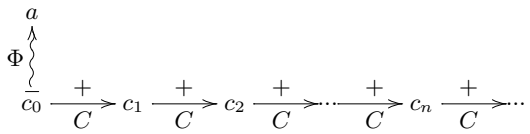
with a'' not a priori related to a'. We call this property *advance-preservation*. Advance-preservation is not composable.

4.4 Liveness

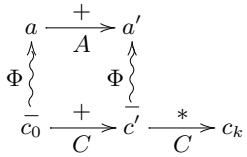
A useful property to require from an implementation is that of *liveness*: any concrete evaluation must spontaneously ad-

[3] It so happens that logicians tend to call "soundness" the property that we above called "safety" [13]. To avoid confusion, it might be better to use another name for the present property. The name "metastability", albeit long and hard-sounding, has 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.

vance. Formally, given a diagram



with $c_0 \dots c_n \dots$ an infinite sequence of strict transitions in C we deduce

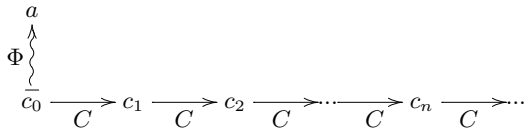


That is, there is an integer k such that abstract computation has strictly advanced before concrete computation reaches k-th transition.

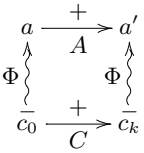
Liveness is a composable property. If we look at temporal logic statements on transitions, liveness ensures preservation of increasing "must" statements by Φ.

Variant: For "real-time" or otherwise bounded resource variants of liveness, we strengthen the property by weakening the hypothesis that the sequence (c_n) be infinite, and introducing instead some weaker criterion of being "long enough" for sequence (c_n) (for instance being of length greater than N, for a given integer N). Such variants may compose or not, depending on expected criteria on the length of transition paths for the implementations being composed and for their result.

Variant: With *strong liveness*, we consider individual steps instead of just strict transition, and we demand that $c' = c_k$. Hence, from



with $c_0 \dots c_n \dots$ an infinite sequence of steps in C, we deduce



Strong liveness combines both weak liveness and soundness simultaneously in the same property, and is composable, all of which make it a very pleasing property. It is a useful property for a single-threaded monoprocessor implementation of a computing system, and allows for design of straightforward such implementations. However, it is so strong that it costs too much to achieve in a concurrent or distributed implementation, since it requires spontaneous and simultaneously synchronization of all the concurrent or distributed processes. Strong liveness like the previous, "weak", liveness, has "real-time" and bounded-resource variants, with similar remarks as in the above cases.

An even stronger (and still composable) variant, *strong step preservation* has same hypotheses as strong liveness, but requires in its conclusion that

$$a \xrightarrow[A]{} a'$$

instead of merely that

$$a \xrightarrow[A]{+} a'$$

which means that you can retrieve individual abstract steps of computation from a concrete trace of execution; i.e. stable states are never optimized away as could otherwise be done.

5 Synchronizing Concurrent Implementations

5.1 Concurrent Implementations

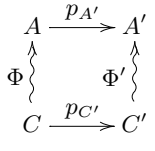
A concurrent computation system is a system made of mostly independent components. Most of the time, components behave like independent computing systems of their own, with local transitions that do not affect other components, and may happen in parallel. Sometimes, the components interact with each other, and the global system state is modified in a non-local synchronized way.

To formalize that, consider a system S such that the set of states of S is the product $\Pi_{i\in I} S_i$ of the set of states of various subsystems $(S_i)_{i\in I}$. Each subsystem S_i is called a component of the system, and the each projection $p_i : S \longrightarrow S_i$ is called a partial view on the system. Each S_i is also endowed with a "local" computational structure $\longrightarrow_{S_i}$. The global system S has a computational structure $\longrightarrow$ such that the product structure $\Pi_{i\in I} \longrightarrow_{S_i}$ is fully included in the structure $\longrightarrow$; that is, any tuple $(t_i)_{i\in I}$ of (possibly null) local transitions is a valid global transition, so that local transitions may happen "in parallel". S may also have additional "synchronized" transitions, that are no products of local transitions, and represent coherent communication or interaction between two or more components of S[4].

A concurrent implementation is simply an implementation the concrete system of which is a concurrent system. When implementing a concurrent abstract system with a concurrent concrete system we often want to establish a correspondance between components of the concrete system and components of the abstract system. This can be formalized as having partial views of each system that make

[4] Note that with the above definition, just any system could be considered concurrent by butchering it into components, trying to define proper sets of local transitions, and falling back to having only null transitions on every component, with every transition not expressed as a product of local transitions being considered a "synchronized" transition. Concurrency is not an intrinsic property of a computing system, but a property of its formalization; a same computing system may have several distinct concurrent or sequential formalizations, whose interest depend on the problem the user is trying to solve when studying the system. For instance, in the case of GC as considered before, most monoprocessor GC'ed implementations statically interleave GC with mutator threads at allocation points; there is no explicit GC thread visible from the system thread scheduler, only a special system mode for GC; conceptually though, we can still consider GC to happen as a concurrent service, one important enough that it has a specific scheduler, optimized for cheap synchronization and inlined around the whole program. As says Lamport, processes are in the eye of the beholder [18].

the following diagram commute:



Note that the concrete system will typically have more components than the abstract system, with additional components representing system services such as message-passing kernels, schedulers, garbage collectors, etc.

Assuming we know how to separately implement each of the components of a given abstract system (as considered with its local structure), as well as how to implement the "synchronized" transitions (possibly in a sequential way), the question is how to achieve a "reasonably concurrent" implementation of the whole system, that would allow local transitions to happen concurrently while still performing global synchronized transitions. That is, we want a technique to correctly combine concurrent implementations.

5.2 Soundness as a Synchronization Primitive

The solution we formalize is to require each component's implementation to be not just *safe* but also *sound* (and possibly complete, live, etc), and to use one or several additional system components to monitor synchronization of "user" components. Actually, this solution is being used universally, in a large variety of special cases.

Consider an abstract system A with components A_i, such that we have a safe and sound implementation of each A_i with C_i. Then we can construct a safe and sound implementation C of A with "user" components C'_i and "system" components C_s, defined as follows: the states of C'_i are the product of those of C_i with the state space of additional control flags and data that will be shared with the monitor and mostly controlled by the monitor, but accessible to no other C'_j. When control flags indicate C'_i is running, then C'_i has local transitions corresponding to each of C_i's transitions. When control flags indicate C'_i must stop, it will by soundness converge to a stable state, set a control flag indicating stability, and stop (i.e. have no more transition). When a synchronized transition is to happen, because a user component wants to communicate, or because some system service (clock, GC, I/O driver, etc) triggers one, the monitor will set "stop" flags on all relevant components, wait for their all being stabilized, and execute a sequential implementation of the abstract synchronized transitions from stable abstract component states; it may reactivate the components when done, or begin another abstract synchronized transition.

Particular instances of this technique have been in use for decades, and informally constitute the very core of operating system and language implementation design. Operating systems for time-sharing systems are thus typically implemented with a "kernel" serves the role of monitor for the implementation of a collection of "user processes" and "system services" that constitute the abstract computing system. Note that the monitor may itself be split into concurrent components, each corresponding to special kinds of synchronized transitions that may be implemented without global synchronization; this is all the truer since the concurrent system can be distributed rather than centralized. On the other hand, as with GC, part of the monitor may be optimized and inlined all around "user" code to achieve better efficiency, rather than having a kernel. Finding an optimized version of a monitor for the implementation of a given abstract system with a given concrete system is left as an exercise to the reader.

The above implementation technique thus allows to *modularly* achieve a safe and sound concurrent implementation of a concurrent system from safe and sound implementations of each of its components, as combined with a suitable monitor; the monitor ensures that interactions between concurrent activities may only happen when all relevant activities are synchronously in a suitably stable state. What remains to be acknowledged about it is that the key to this modular design is in the notions of stable states and of sound implementation. And the key has to match the lock: the latter notion depends on the former, and the former depends directly on the abstract system being implemented, its concurrency and atomicity model; when implementing a different application, the abstract system changes, and so does the notion of stable state, so does the concurrency and atomicity model. This means that a monitor for a given low-level notion of stability isn't immediately suitable as a monitor for a higher-level notion of stability that would have additional invariants; given such a notion, it requires the rework of a complete new layer of a monitor and of wrappers around user processes so as to implement the abstract system, and then, wrapped user processes cannot interact seamlessly with unwrapped user processes. This gives a theoretical account of the abstraction gap that prevents seamless mixing of components written in high-level languages such as Java with components written in low-level languages such as C, least the low-level component follows stringent conventions such as JNI.

All in all, it appears that it is useful to define several notions of state stability, depending on the kind of synchronization needed, on the invariants that the system is able to express. Unability for a high-level programming language to directly express the user application's arbitrary invariants leads to expensive cost in terms or program complexity and inefficiency due to abstraction inversions [3] involved in reimplementing the application's conceptually low-level invariants in terms of the programming language's high-level primitives.

Use of various mechanisms to achieve transparent preservation by the system of a non-trivial notion of stability can be traced back at least to ITS [7], in the late 1960's, with its PCLSRing feature [4], whereby interrupted system calls would always leave a process in valid user mode state. "Reflective" systems attempt to provide a way for the programmer to define monitors within his usual application programming language, using "metaobjects" that run concurrently with normal objects and are able to control them [21, 14]. However, the only language we know that allows the programmer to directly define his own notion of stability is C--, with its notion of safe-points, and with soundness being specified in terms of an `ExecuteToNextSafePoint` primitive [24]. Not surprisingly, the need for such notion was a direct consequence of designing the C-- language as a generic portable back-end to arbitrary high-level languages, each with its own abstract notion of stable state. Our study suggests that all languages, even high-level languages, should have the capability of defining one or several notions of stable states, for they are always used to implement applications that are higher-level computing systems than they are.

5.3 Techniques to Achieve Soundness

We have formalized soundness and seen how to use it, but haven't yet explained how it could be implemented. Herein, we'll try to give a general view of well-known techniques that have already been used in the past to achieve soundness whenever it has been needed. Mosberger et al. [23] propose use of various stabilization techniques for achieving atomic operations in presence of interrupts.

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, 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 so its jump out into the monitor instead of continueing execution; Ogesen used such technique for GC in the EVM implementation of Java [1].

Now, whatever nifty mechanisms the system uses to achieve soundness with respect to its own invariants, lack of user-accessibility of such mechanisms forces the user to pay a high cost when enforcing the high-level invariants he cares about. In absence of efficient system support for soundness with respect to user-defined invariants, the user must indeed fall back to the extremely costly methods of manually acquiring locks or polling, or invest in unportable reimplementation of system mechanisms. User definability of varieties of stability invariants and corresponding safe points can thus be viewed as a declarative dual to manually acquiring and releasing locks around critical sections; like all declarative approaches, it requires an additional bit of support from metaprogramming tools (compilers), but in exchange allows the latter to do all kinds of system-dependent optimizations that the programmer couldn't otherwise achieve in a way at the same time a simple, portable, and efficient.

6 Conclusion

We have defined a generic notion of implementation, applicable to a wide variety of existing systems and formalisms. We have formalized various desirable properties of implementations, and identified one, soundness, that we think lacked deserved recognition. We have shown how to use our formalized concept of soundness to achieve concurrent implementations of concurrent systems in a modular way. We have stressed the fact that at the center of all the properties we studied lies the intrinsically application-dependent notion of a stable system state. We have accordingly suggested that language design should include an explicit user-definable notion of stable system state.

We think that the above contribution is of interest to language designers as well as to people trying to formally establish the correctness of program implementations and of implementing metaprograms.

This formal notion of implementation could be used to specify failsafe implementations, by considering various "enrichments" of the concrete system of an implementation with additional state transitions for every kind of failure, and studying properties that are preserved in presence of failures (in increasing order of preference, liveness, soundness, safety).

A further research area we intend to explore, is to use this formal notion of an implementation, together with a formal notion of metaprogramming [26] to better understand and specify the semantics of reflective systems. Reflective systems are an attempt to allow direct expression (as opposed to mere manual implementation) of user-defined domain specific languages, by dynamically interning an implementation of these languages; formalizing the notion of implementation is thus necessary to specify the correctness of reflective programs.

We may even try to deduce "meta-object protocols" from the above-mentionned formalized logical properties: stabilization primitives can be considered a computational counterpart to the logical property of soundness, and we may try to similarly find computational counterparts to other logical properties.

A related research topic is Aspect Oriented Programming [15], whereby a program is defined by multiple distinct aspects that an implementation of the program should simultaneously implemenent. For instance, from our analysis of the notion of implementation, we may interpret crosscutting problems in AOP as the expression of implementation being a decompositional notion rather than a compositional one. A particular class of systems that must thus simultaneously implement multiple aspects are systems described with the ODP models, used in the industry and telecommunications.

References

[1] Ole Agesen. GC Points in a Threaded Environment. Technical report, Sun Microsystems, 1998. `http://www.sunlabs.com/research/java-topics/`.

[2] Lourdes Araujo. Correctness proof of a Distributed Implementation of Prolog by means of Abstract State Machines. *Journal of Universal Computer Science*, 3(5):416–442, 1997. `http://www.eecs.umich.edu/gasm/verif.html`.

[3] Henry G. Baker. Critique of DIN Kernel Lisp Definition Version 1.2. *Lisp and Symbolic Computation*, 4, 4:371–398, March 1992.

[4] Alan Bawden. PCLSRing: Keeping Process State Modular. Technical report, MIT, 1989. `http://fare.tunes.org/tmp/emergent/pclsr.htm`.

[5] David Chase and al. Garbage Collection FAQ, 1996. `http://www.iecc.com/gclist/GC-faq.html`.

[6] Paul Curzon. A verified Vista implementation. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993. `http://www.cl.cam.ac.uk/users/pc/fintr93.html`.

[7] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. ITS 1.5 Reference Manual. Memo 161a, MIT AI Lab, July 1969.

[8] Solomon Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995. http://math.stanford.edu/~feferman/.

[9] Wolfgang Goerigk. On the Correctness of Compilers and Compiler Implementations. Technical report, 1995. http://i44s11.info.uni-karlsruhe.de/~verifix/.

[10] Wolfgang Goerigk and Ulrich Hoffmann. Rigorous compiler implementation correctness: How to prove the real thing correct. In *Proc. FMTrends'98*. Springer, 1998. http://i44s11.info.uni-karlsruhe.de/~verifix/.

[11] Joshua D. Guttman and Mitchell Wand, editors. *VLISP: A Verified Implementation of Scheme*. Kluwer, Boston, 1995. Originally published as a special double issue of the journal *Lisp and Symbolic Computation* (Volume 8, Issue 1/2). ftp://ftp.ccs.neu.edu/pub/people/wand/vlisp/lasc/.

[12] J. Y. Halpern, R. Fagin, Y. Moses, and M. Y Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[13] John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press. http://www.cs.cmu.edu/~fp/papers/.

[14] R. Kaer and Th. Mahler. Introducing and Modeling Polycontextural Logics. Technical report, Institut für Kybernetik und Systemtheorie, 1998. http://www.techno.net/pcl/tm/plisp/.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, June 1997. http://www.parc.xerox.com/spl/projects/aop/.

[16] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(2):77–101, 1986. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-008.html.

[17] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. Technical Report SRC-015, Digital, 1988. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-015.html.

[18] Leslie Lamport. Processes are in the Eye of the Beholder. Technical report, Digital, 1994. http://www.research.compaq.com/SRC/personal/lamport/tla/.

[19] Leslie Lamport. Refinement in State-Based Formalisms. Technical Report SRC-1996-001, Digital, 1996. http://www.research.compaq.com/SRC/personal/lamport/tla/.

[20] Saunders Mac Lane. *Categories for the Working Mathematician, Second Edition*. Springer, 1998.

[21] Satoshi Matsuoka, Takuo Watanabe, Yuuji Ichisugi, and Akinori Yonezawa. Object-oriented concurrent reflective architectures. *Lecture Notes in Computer Science*, 612:211–??, 1992.

[22] Sigurd Meldal and Michal Antonin Walicki. Nondeterministic Operators in Algebraic Frameworks. Technical report, Stanford CS Laboratory, 1995. http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs/CSL-TR-95-664.

[23] David Mosberger, Peter Drushel, and Larry L. Peterson. Implementing Atomic Sequences on Uniprocessors Using Rollforward. *Software – Practice and Experiences*, 26(1), January 1996.

[24] Simon Peyton-Jones and Norman Ramsey. Machine-Independent Support for Garbage Collection, Debugging, Exception Handling, and Concurrency. Technical Report CS-98-19, Department of Computer Science, University of Virginia, August 1998. http://www.cs.virginia.edu/~nr/pubs/c--rti-abstract.html.

[25] Cornelia Pusch. Verification of Compiler Correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, pages 347–362. Springer-Verlag, 1996. http://isabelle.in.tum.de/~pusch/pubs/Pusch_TPiH1996.html.

[26] François-René Rideau. Metaprogramming and free availability of sources, January 1999. Translated from the french article "Métaprogrammation et libre disponibilité des sources" published in conference "Autour du Libre 1999". http://fare.tunes.org/articles/ll99/index.en.html.

[27] François-René Rideau. Reflection, non-determinism, and the lambda-calculus. Rapport de recherche non publié, CNET, 1999. http://fare.tunes.org/tmp/rndlc/.

[28] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, 1992. http://www.cs.utexas.edu/users/oops/papers.html.

A Acknowledgements

I would like to thank Elie Najm, my PhD advisor, for helping me to clarify some implementation concepts. I am particularly indebted to Fabien Delpiano, my fellow PhD student, for his precious help while writing this paper. Also, I wish to thank members of the Tunes project for their constant help and support. Last but not least, I must express my gratitude to Jean-Bernard Stefani, head of the marvelous department DTL/ASR of CNET, who made this research possible.