It's Only Illegal if You Get Caught:

Breaking Invariants and Getting Away with it

Raphaël Proust
University of Cambridge
Raphael.Proust@cl.cam.ac.uk

Alan Mycroft
University of Cambridge
Alan.Mycroft@cl.cam.ac.uk

Abstract

Programming languages and coding standards provide invariants to ease reasoning about the correctness of code. Although useful, invariants are often intentionally broken by programmers for performance or compatibility purposes. An operation that consists of multiple steps can preserve an invariant overall even though it breaks it temporarily during the process—e.g., inserting a node into a doubly linked list takes two operations between which the list is ill-formed. It is important that intermediate states of these operations are not observable by the rest of the program. We explore various devices that are used to bundle together the different steps of such an operation in a way that hides intermediate states—bringing some form of atomicity. However, while all these constructs might work in a certain context there is no way to ensure they still work for extensions and new versions of the programming language, the underlying operating system, the linked libraries, or even the processor architecture. We propose a new construct, `opaque`, to overcome these problems—decoupling code correctness and execution context—and future-proof invariant-breaking code by insisting that both current and future versions of the compiler treat the enclosed block as having no observable intermediate states.

Categories and Subject Descriptors D.3.m [*PROGRAMMING LANGUAGES*]: misc

Keywords Invariants; Abstraction; Programming language design; Feature interaction; Programming language evolution; Negative features; Close world assumption

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Onward! '14, October 20–24, 2014, Portland, Oregon, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3210-1/14/10…$15.00.
http://dx.doi.org/10.1145/2661136.2661142

1. Introduction

Invariants are essential to writing and reading programs. When a codebase grows too big, it becomes impossible for humans to fully understand the whole program. For this reason, large programs are (or should be) split into components—functions, procedures, modules, etc.—with simple interfaces. By restricting the possible interactions with their associated components, interfaces guarantee certain behaviours in the form of invariants—properties that, whatever happens in the execution of the program, remain true. These invariants help programmers to modularise the reasoning about the program.

Similarly to humans, analysis tools rely on invariants for asserting the correctness of the code they are fed. Programming languages such as Why3 and Dafny require their users to annotate programs so the system can statically guarantee some invariants.

The way programmers—and by extension analysis tools, because they are built by programmers—reason about programs is based on programming language documentation and specification. These explain, in a more or less formal language, how programs are evaluated, how code behaves. There are explanations about how such feature works and what such construct does and how the evaluation of such element unfolds. For example programmers might glance at the code presented in Figure 1 and assess its correctness based on the programming language documentation. Programmers read the documentation of the different constructs used in the program and assess the correctness of the behaviour and the absence of crashes. If the documentation does not mention, say, concurrency at all, programmers will not consider concurrency at all. More importantly, programmers will assume there is no concurrency and deem the program to behave as expected.

That is, programmers follow a *closed world assumption*: the rules for evaluating a program are exactly those of the programming language specification. Unfortunately, this assumption is wrong because programming languages evolve—and linked libraries are updated, operating systems are changed and new processor architecture are produced. New versions of programming languages can add new con-

```
(*invariant: there are [count] items in [stack]*)
type 'a t = {
        mutable stack: 'a list;
        mutable count: int
}
let push t x =
        t.count <- t.count + 1;
        t.stack <- x :: t.stack
let pop t =
        if t.count = 0 then
                None
        else
                (result = List.head t.stack;
                t.count <- t.count - 1;
                t.stack <- List.tail t.stack;
                Some result)
```

Figure 1: A simple stack implementation.

structs and thus new ways to evaluate programs. This means that code should be reasoned about—and analysed—in an ever changing context. In particular, the code in Figure 1 becomes buggy the day the programming language acquires concurrency or parallelism.

We take a close look at this issue. Through a series of examples, Section 2 exposes the necessity of breaking invariants and the different recovery devices that are commonly deployed. Section 3 details the issues the existing devices have. We suggest, in Section 4, making invariant breakability a central concern of programming language design and, to such effect, we sketch plans for a new construct called `opaque`.

2. Invariants

We now take a closer look at invariants: how and why they are enforced and their limits.

2.1 Different kinds of invariants

Programming languages have their own invariants. These are negative features, things that programmers *cannot* do. For example, it is not possible to mutate values in Haskell, a state monad may be used instead. This helps every Haskell programmer to reason about their and others' programs: by merely inspecting the type of a function they know whether it can have effects within the state monad or not.

There are also invariants that, although not guaranteed by a programming language, are widespread amongst the whole user community. As an illustration consider that Go users are encouraged not to leak the `panic` mechanism outside package boundaries in order to keep the control-flow simple: "The convention in the Go libraries is that even when a package uses panic internally, its external API still presents explicit error return values."[1] This is not enforced by the programming language, it is merely a style guide, a more or less informal coding standard, shared by all users of the language. It is not a grammar or semantic rule, simply an idiom that the whole community adopts.

Additionally, users maintain their own invariants over the whole or parts of projects. These are specific to the data handled by the program. A programmers might, say, handle a collection as an ordered list. The programmer is responsible for maintaining the ordering through every operation that manipulates such a list.

Invariants can be maintained through a range of different means: language specifications and implementations, code reviews, type checkers, program verification, `lint`-like tools, integrated and continuous test suites... In the case of the `0install` project[2], the immutability of certain objects was originally maintained by the programmer. Comments in the documentation warned library users to beware of modifying in place XML fragments returned by certain functions. After some refactoring, the burden of enforcing this invariant was shifted to the compiler. It involved removing a `mutable` annotation and rewriting the parts of the code the compiler pointed out as breaking immutability. The author of `0install`, Thomas Leonard, reported on the process on his blog [3]. This illustrates the fact that the same invariant can be maintained either by comments and carefulness or with type system constraints.

2.2 Breaking invariants: why?

While invariants have their use, it is sometimes necessary to break them. Even very strict programming languages provide mechanisms to break their invariants. Invariant-wrecking features of Haskell (including `coerce: a -> b`) are grouped in the `Unsafe` modules in different parts of the standard library [1]. The OCaml programming language provides an `Obj` module with the tersest and most alarming documentation [2]: "Operations on internal representations of values. Not for the casual user." The Rust programming language allows programmers to tag functions and blocks with the keyword `unsafe` [4] to indicate to the compiler that safety checks are to be temporarily reduced to a minimum.

Common reasons for breaking invariants include the following.

- Executing performance-critical tasks. For example, in a programming language with immutability, the inner loop of a function performing hashing might be implemented with in-place mutation.
- Interacting with other programming languages: calling into a different language voids the guarantees of the caller language.

[1] From `http://blog.golang.org/defer-panic-and-recover`

[2] Official website for the `0install` project: `http://0install.net/`

- Implementing specific categories of programs. As an example, consider that a just-in-time (JIT) compiler needs to mix writable data and executable code. It means that a JIT compiler needs to cast an array of bytes (code as a sequence of instructions) into a function (code as executable, callable procedure)[3]. A strong type system with no support for unsafe casting forbids such a program[4].

Consider the compelling case of Rust where the type system prevents users from distributing to distinct threads pointers to the same value. The original aim of this restriction is to allow safe and efficient in-place mutation. The `arc` module allows programmer to freely distribute pointers to read-only values. It is safe because read-only values are never mutated and thus can be accessed concurrently by multiple threads. Thus, to implement `arc` it is either necessary to make it a programming language primitive (i.e., a special case in the type system), or to break type-system invariants. The `arc` module is implemented as a library and relies on `unsafe` blocks and functions to break programming language invariants. The invariant breaking is restricted to specific parts of the implementation and is not apparent at all in the interface of the module. This approach simplifies the programming language implementation and documentation and modularises development—the library can evolve independently from the compiler and vice-versa.

For reasons such as these—that is, out of *necessity* and because it makes code better—programming languages such as Haskell, OCaml and Rust provide their users the means to break even the strongest invariants of their runtimes. Programming language support for invariant-breaking mechanisms is driven by necessity. Their pervasiveness is evidence that invariants are considered too restrictive by programming language designers and programmers alike: their breakability is recognised as vital to programming.

2.3 Breaking invariants: how?

Programming language designers let programmers break carefully designed invariants, because *it's ok*. Breaking an invariant is only bad if it is observed. As long as the invariant is observably maintained, its breaking hidden, no one cares. This is illustrated by the common idiom in ML: providing a purely functional interface to a library that relies on mutable state internally. The (imperative) implementation of the library is hidden and the library user is content with the illusion of immutability.

In fact, programming language designers provide features to hide invariant breaking. These devices are curtains behind which invariants are disregarded. We discuss them now.

[3] From an operating-system point of view, this bears resemblance with the breaking of the W^X invariant, designed to prevent virus mutation.

[4] Very rich type systems (such as with dependent types) could express the safety of—and thus allow—coercions of this sort.

2.3.1 Recovery

Programmers surround (or should surround) their dangerous code with comments. This is a notice to other maintainers: “here be dragons”. Although essential, comments are not sufficient. The dangerous parts of the code are (or should be) also surrounded by devices that will isolate the invariant breaking. These devices are prologues and epilogues surrounding the badly behaved code in a way that ensures the invariants are consistently restored, the safe state recovered. The effects of the badly behaved code is thus contained and prevented from leaking into other parts of the program.

Consider the examples in Figure 2: distinct ways to deal with the expression $evil$ that breaks some invariant on the value held by `x`. For example, $evil$ might traverse `x`, looking for a particular element, and mark the nodes it visits in order to detect cycles. (While the particular examples in the Figure 2 are written in a functional programming language, similar devices exist for other paradigms.) In each of the examples, the original value held by `x` is duplicated (through the deep-copy primitive `copy`) and distinct copies are passed to the invariant-breaking code $evil$ and the rest of the program $code$. This effectively saves the original value and restores it afterwards. The global α-renaming (Figure 2b) approach differs notably from the two other solutions in that the original value is passed to $evil$ and the clean copy to the rest of the $code$. The other differences are mostly stylistic with shadowing (Figure 2c) being arguably the most localised: the scope of the variable holding the copy does not extend past the $evil$ expression.

Also consider the case of optimisations and analyses in the LLVM framework. The framework enforces the invariant that, for each unit of the source code, the information provided by analyses continues to correspond to the code even after it is modified by optimisations. This correspondence is broken when the code is significantly transformed by some optimisations. The framework deals with this breaking in the following way: the programmer declares what set of analysis results is preserved by the optimisations they write [5], all the analyses not in this set are rerun by the framework. Thus, the LLVM framework automatically re-computes the data associated with the code. When the next optimisation in the pipe-line is run, all the analyses' results match the code being compiled.

Another solution to deal with invariant-breaking code is to append clean-up code: a piece of code that restores a data-structure's invariant by traversing it and mending it. Consider the case of an ordered list that is passed to a procedure dealing with lists in general (not necessarily ordered): the ordering might be lost. Running a sorting procedure afterwards restores the invariant.

These are different ways by which a programmer can deal with invariant breaking. Saving and restoring the data, recomputing part of the data, or reshaping it.

```
let x' = copy x in
let z =

        evil[x']
in
code[x, z]
```

(a) Local α-renaming

```
let x' = copy x in
let z =

        evil[x]
in
code[x', z]
```

(b) Global α-renaming

```
let z =
        let x = copy x in
        evil[x]
in
code[x, z]
```

(c) Shadowing

Figure 2: Save-and-restore devices.

2.3.2 Self-recovery

Some transformations do not need explicit recovery even though they traverse unsafe intermediary states. Consider the case of a `struct` with several fields, bound together by some invariants. Such a `struct` might be used to represent a node in a tree, with a field for each of the children, a field for the depth, a field for the size, etc. When performing an insertion in this tree, several field might need to be updated. Until all the fields have been updated, the node is in an inconsistent state that breaks its invariants—e.g., the tree is deeper than the depth field indicates. Once all the fields have been updated, the node has recovered its consistent state and does not need further mending.

2.3.3 Bundling and the notion of opacity

Programming languages come with a number of features meant to *bundle* different bits of code together. Functions, methods, procedures, blocks, modules, objects, classes… The function abstraction (amongst others) provides programmers with a means to place clear boundaries around parts of the code. Bundling the badly behaved code and the necessary recovery device—or all the steps of a self-recovering transformation—together in a function effectively hides the actual behaviour of the code.

When a function f calls another, the control flows to instructions that are not in the body of f. We still consider these instructions to be executed inside of f. Thus being *inside* a function does not only depend on the value of the program counter but also on the path the program counter followed there, the call string. In general bundles can be nested and thus inside and outside is relative.

One of the purposes of functions—and other bundling construct—is to provide a boundary for distinguishing local and global reasoning. It separates an interior (commonly known as the *body*) from an exterior and prevents details from the former (implementation) to be observed by the latter. (Another purpose is code re-use, but we are not presently concerned with that aspect.) Inside the function body, invariants might not hold. It is not problematic because the code fragment is small and reasoning does not rely on invariants as much. On the other hand, for the rest of the program the function behaves nicely as an unsplittable, uninterruptible[5] unit of code. The caller cannot know whether the invariant was broken or not: for all reasoning purposes, the execution of the function call has a “before” and an “after”, but no “during”. This allows global reasoning to rely on invariants. As an illustration, consider the `push` and `pop` functions in Figure 1. Values of the type `'a t` are meant to keep their fields in sync: `count` describes the number of elements in `stack`. From their callers' perspective, both functions `push` and `pop` keep this property true. Inside the body of both functions however, the fields are temporarily out of sync.

We call functions, and similar bundling devices that make their internal behaviour unobservable, *opaque*. From the exterior, they make it appears as if the interior is—for the purpose of reasoning about correctness—a single instruction.

2.3.4 Non-syntactic devices

Functions—like their variants: blocks, methods and procedures—provide mostly syntactic isolation: there is code inside the function borders and code outside. Nested function calls make the distinction between inside and outside not entirely syntactic as mentioned in Section 2.3.3.

Alternatively, programmers can use non-syntactic devices such as monads. In Haskell, all values whose evaluation triggers side effects or raise and handle exceptions need to be wrapped inside the appropriate monad. These are not syntactic constructs: whether a given value is inside or outside a certain monad depends on its type. (Just like functions, monads have other uses that we are not presently interested in.) Monads focus on isolating values rather than fragments of code. Thus, invariants can be encoded in the type system rather than the source code layout.

Another, non-syntactic way to enforce invariants is to use objects. The maintenance of invariants in object-oriented code has been studied by Leino and Muller [9] amongst other. (Objects provide other features that we are not concerned with.)

The current work only focuses on syntactic constructions: functions, methods, procedures, blocks, etc.

[5] Although a synonym in English, the word *atomic* has a different meaning in the programming language lexicon. The relation between the two concepts is explored in Section 3.

3. Feature interaction

Note that the opacity of functions is an assumption that does not hold in every execution model. Therefore it may not be preserved by compiler updates. We now look at specific examples of features reducing the opacity of functions.

When considering a concurrent program, functions lose their opacity as the execution can interleave code that may observe the state of values handled in the function. Consider the functions `void f(){foo();bar();}` and `void g(){bar();foo();}`, where `bar` and `foo` are two functions with side effects (both writing and reading) on disjoint sets of global variables. The functions `f` and `g` are not (observably) equivalent in concurrent execution contexts. For the same reason, the functions `push` and `pop` from the Figure 1 are not thread-safe. The concurrency feature *interacts* with the function feature, modifying the latter's semantics in non-trivial ways: it gives functions a "during" along with their "before" and their "after". The opaque nature of functions has become defeated by the interaction with concurrency. In order to restore the opacity (and the atomicity) of the function it is necessary to introduce synchronisation devices such as locks or semaphores.

Note that synchronisation devices can introduce deadlocks in the program. Thus, it is not trivial for the compiler to automatically wrap function bodies in appropriate synchronisation devices. Gudka's work on lock inference [6] provides automation of this kind for the Java programming language.

Some programming languages feature constructs that provide opacity under concurrency: synchronisation primitives. The `synchronized` keyword in Java is one such construct: it partially[6] restores the opacity of the method it is applied to. Semaphores, locks, and other devices can be used to surround the parts of the code that need to become opaque. If not provided as primitives of the programming language, they can be coded manually by the programmer and used as a library.

Similarly to concurrency, mutability can reduce function opacity. An impure function that leaks some state during execution—by writing it to a mutable value—will have its internal behaviour partly exposed.

Another feature that can defeat function opacity is exceptions. A raised exception will flow across function boundaries to the closest exception handler. This handler might observe the state the program was in when the exception was raised—e.g., the handler might be given data that has only been partially updated. Thus, exceptions can degrade function opacity.

[6] A `synchronized` method is never interleaved with another `synchronized` method of the same object, however, interleaving is possible with non-`synchronized` methods of the same object and any methods of another object. For this reason, `synchronized` makes methods partially opaque only.

4. A new look at programming language design

We now look at invariant-breaking features and opacity constructs from a programming language designer perspective.

4.1 Invariant breaking as a core concern of programming language design

Considering the pervasiveness of invariant breaking, and all the effort put by programming language designers to allow it, why do we even bother with invariants? The important point of invariants is that—even though they may not hold locally—they do hold globally, which is vital for understanding and maintaining the program. As detailed in Section 2.2, there are valid reasons to break invariants.

It is necessary to be pragmatic while developing a programming languages for the real world: invariants are vital but so is their breaking. A good programming language is one that provides appropriate means for programmers to misbehave. While this might not be ideal when teaching beginners to print an infinite stream of Fibonacci numbers, it is a necessary tool for seasoned practitioners interested in writing useful programs that interact with the system on which they run. Just like public health programmes encourage the distribution of clean needles to drug addicts in order to minimise risks of infection, we encourage programming language creators to provide all the apparatus that allows programmers to misbehave in a safe way. In a good programming language, it should be possible to go in and out of safety, compartmentalise "bad" behaviour, and escape any of the harmful consequences of invariant breaking.

4.2 The need for future-proofing constructs

Consider the following example: a collection being mutated in-place one element at a time. We consider, more specifically, the case of an array of integer mutated into an array of floats—thus temporarily mixing values of different types in a single collection and breaking invariants of the programming language runtime. (Another example is re-encoding text stored in ropes, say, from latin1 into utf8. When only part of the rope has been re-encoded, attempts to compute its length are bound to fail and attempts to print it would most likely result in gibberish.) In our chosen example—as in any similar situations—it is important that the intermediary states that the transformation goes through are not observed by other parts of the code: a partially updated array should not be traversed because it is heterogeneous. The operation as a whole maintains the programming language runtime invariant, but the individual updates of each element does not. Thus the whole update needs to behave as if executed in one single step in all the mental models the programmer uses when thinking about their code.

4.2.1 Bundles and sections

A first attempt towards making the update look, feel and, more importantly, behave as a single step is to bundle all of the intermediary operations together. As detailed in Section 2.3.3, there are different constructs for that purpose: functions, methods, procedures, macros—the example in Figure 3a illustrate the use of a function. All these help the programmer reason about the update as a single operation. However, they all fall short if the context—whether through evolutions of the programming language compiler, execution platform, linked library, or source code—changes and, say, concurrency is introduced.

In addition to bundling the updates together, the programmer then must surround the bundled code with a synchronisation device: locks, semaphores, or atomic sections marker—in the example in Figure 3b an atomic block is used. All these, when handled correctly, can make the intermediary states of the transformation invisible to the outside. However, they all fall short if the context—for similar reasons—changes further and exceptions are introduced.

On top of all the changes to the original code, the programmer now needs to enclose everything in an exception handler—as illustrated in Figure 3c.

This example exposes the shortcomings of the existing programming language constructs for the purpose of isolation. They tie code correctness with the execution context very tightly: each change to the feature set or the execution model of the programming language or execution environment of the program can create[7] bugs.

4.2.2 Inconsistent constructs

The reason for this state of affairs is that programming language constructs offer partial features: atomic blocks effectively prevent another thread jumping in and witnessing any of the temporary states traversed by the computation in the block, but they do nothing to prevent the current thread from jumping out—e.g., by raising and exception—and exposing the same temporary states to the rest of the world. There are no constructs that hide intermediate states consistently across contexts.

The different contexts appear due to the evolution of different elements of the tool-chain, surrounding libraries, etc. Because this evolution generally occurs over time, consistent constructs are more commonly known as *future-proof*.

Additionally, most programming language constructs have multiple uses: functions and methods provide code reuse, classes provide inheritance and initialisers, etc. These additional uses can be sources of further feature interactions.

[7] Bugs are not merely *exposed* by the changes in context: bugs that might have not existed at all can appear.

4.3 Future-proofing through negative features

We classify programming language features into two categories: positive and negative, and show how the latter can be used to future-proof code.

4.3.1 Positive and negative features

We call *positive* features those that bring more ways to unfold the evaluation of a program. Consider exceptions: they offer new ways for control to flow, new ways for programs to behave. In operational semantics, positive features are expressed as rules that can be applied to make the evaluation progress.

Conversely, we call *negative* features those that reduce the set of possible evaluation steps that can be taken. Consider Java's final: a keyword that makes certain behaviours invalid. When formalised in operational semantics, negative features are expressed as guards and patterns that limit the set of rules that can apply to a specific program.

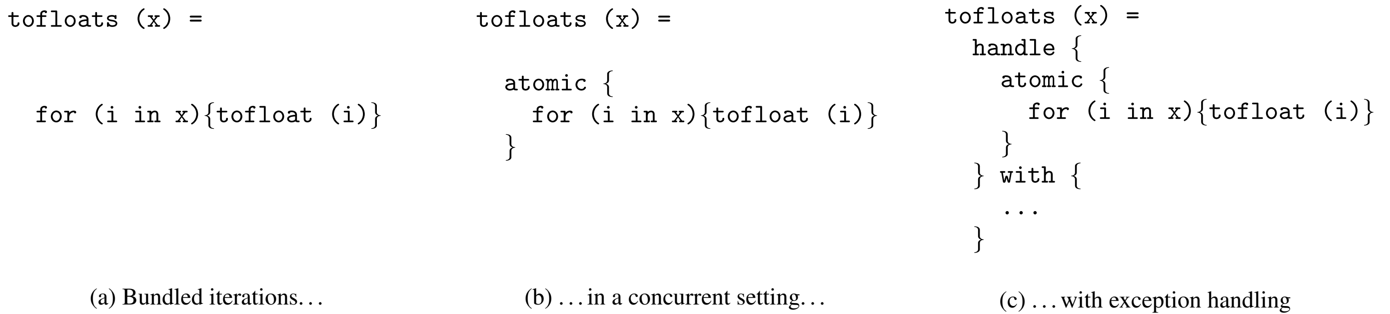
Note that some features can simultaneously have positive and negative aspects. For example, cooperative concurrency makes concurrency possible, but restricts the points at which the scheduler can make another thread run.

4.3.2 Closed world assumption

Consider a formal proof of, say, the absence of crashes in a specific program written in a programming language with only positive features. (Interestingly, the absence of crashes is a negative property: it is concerns with things *not* happening.) The proof shows that, in the states visited by the program, the operations it performs are safe. Such a proof uses an often unspoken assumption: there are no other evaluation rules than the rules given in the specification. Without this additional closed world assumption, there are no limits to the behaviour of programs and thus no way to guarantee that something cannot happen[8]. Note for example that the Haskell 2010 Language Report [10]—the specifications for the Haskell programming language—only mention mutability or immutability in the title of a section "Immutable non-strict arrays". Programmers reasoning about the correctness of Haskell programs will assume values are immutable because there are no construct to mutate them.

The set of rules used to evaluate a program evolves with the successive releases of the compiler. Note that, for reasons of backwards compatibility, constructs are very rarely removed. Whenever a (positive) feature—with associated new constructs and evaluation rules—is added to the programming language, the closed world assumption is invalidated, and so are the proofs of correctness. For each state that the

[8] This bears resemblance to inductive reasoning on integers. When defining the natural numbers inductively, after (a) $0 \in \mathbb{N}$ and (b) $x \in \mathbb{N} \implies S(x) \in \mathbb{N}$, a third axiom is necessary: (c) $\mathbb{N}$ is the smallest set satisfying (a) and (b). Axioms (a) and (b) are positive and allow integers to be constructed; axiom (c) is negative and prevents additional integers to be included. Without this last negative axiom, it is not possible to show $\forall$-properties by induction.



(a) Bundled iterations... (b) ... in a concurrent setting... (c) ... with exception handling

Figure 3

program traverses, it is necessary to show that either the new evaluation rules cannot apply or that applying them is harmless—and keeps the program in a safe state with respect to previous invariants.

4.3.3 Resilient proofs of correctness

A negative feature, one that restricts the set of possible evaluation steps rather than enlarging it, is fundamentally different. By restricting the way programs are evaluated, negative features offers guarantees that positive features cannot give. In a programming language with negative features, programmers can make deductions that continue to hold after other features have been added. Negative features allow programmers to consider only a subset of the evaluation rules on specific parts of the program. If this subset is unaffected by an update of the compiler—i.e., if the introduced positive features are forbidden by the protecting negative feature—then the proof remains valid across the update. More specifically, code that is protected by a negative construct (e.g., atomic) that makes a specific behaviour impossible (e.g., parallelism) will not need to be checked again when adding features that have the same specific behaviour (e.g., parallel execution constructs).

Note that negative features are not magic bullets: *some* proofs need to be redone. Only those concerned with parts of the programs that are protected by an appropriate negative features preserve their validity.

4.4 A future-proof device: `opaque`

We look into a specific proposition for protecting invariant breaking code with a negative feature: the `opaque` construct.

4.4.1 The `opaque` keyword

We propose that programming language designers and maintainers have an explicit agreement with their users about feature interaction and future-proof opacity. This treaty can take the form of the introduction of the special negative construct `opaque` with the following specification: the intermediate states traversed by the code inside of the `opaque` construct is not observable by the code outside. This property must hold in the first release of the programming language and must continue to hold in further versions, rendering the opacity future-proof. The details of how the responsibility of ensuring opacity is shared are given in Section 4.4.2.

Blocks can be qualified (`opaque {...}`), in which case the intermediary states the code traverses are not observable. Because blocks are the simplest syntactic construct `opaque` can also be used with functions (`void f () { opaque {...} }`) or be combined with macros (`#define SWAP(x,y) opaque {...}`) or other syntactic constructs. The `opaque` constructs can protect all the invariant breaking parts of the code that in principle expose dangerous states.

4.4.2 The `opaque` treaty

The `opaque` treaty is bipartite, binding both the programming language users and the programming language developers/maintainers.

For code that does not use unsafe features (such as unchecked type cast or pointer arithmetic), the treaty binds the programming language maintainers in the following way: the compiler tries to make state unobservable. The compiler might analyse the code and realise it is already opaque, or insert some runtime checks, or add code to copy data to avoid sharing. If the compiler fails to render the code opaque, it does not emit code and returns an error to the programmer. This gives the programmers the total assurance that, during execution, `opaque` blocks are indeed opaque. When changing the execution context—e.g., by introducing a new feature—the programming language maintainers ensure that the opacity of the block is preserved. Two different such cases are detailed in Sections 4.4.3 and 4.4.4.

The treaty binds the programming language users, the programmers, in the following way: programmers using unsafe features of the programming language (such as Haskell's `coerce` or Rust's `unsafe`) in an opaque block are responsible for guaranteeing the opacity of the whole block. The original aim of unsafe features is for programmers to be able to take full responsibility for the correctness of their code with respect to the current programming language runtime safety. In an `opaque` block, opacity is a criterion of correctness. Thus programmers using unsafe features take

```
let rec inplace_map f xs =
  opaque {
    match xs with
    | [] -> []
    | x::xs -> inplace_map x (f x); map f xs
  }
```

Figure 4: Higher-order opaque function.

full responsibility for guaranteeing opacity. A discussion on the utility of `opaque` in such a context is given in Section 4.4.5.

4.4.3 Opacity through static analysis

Consider the case of a programming language that evolves and acquires support for exceptions. (Or similarly a project that used to rely on an error monad and switches to native exceptions.) Raising an exception exposes the intermediate state of a computation to the closest handler. Thus exceptions must not be allowed to cross the borders of an `opaque` construct. This restriction prevents opaque blocks from having multiple exit points, preventing code executing outside of the opaque construct to observe partially updated data.

Exception raising and catching can be tracked through effect analysis [8]. Specific patterns of usage of exceptions can be rejected by the compiler based on the results of this analysis. In particular, it is possible to detect whether a piece of code has exceptions flowing out and to refuse to compile the corresponding program. (Java programmers are familiar with the concept: the compiler rejects code that lets exceptions leave methods if they are not annotated with the `raises` keyword.) Whenever the programming language maintainers add exceptions to a new version of the programming language, the `opaque` treaty requires them to implement something similar to the effect analysis and to make the compiler reject code which has exceptions flowing out of `opaque` blocks.

These restrictions are not as trivial as the previous statements make them appear. Indeed, higher-order functions—as well as function pointers, dynamic dispatch and other such mechanisms—complicate effect analysis. In particular, consider the function `inplace_map` of Figure 4 when the programming language evolves and gets support for exceptions. Initially (before the exceptions are available in the programming language), the type of the function might be as simple as `('a -> 'b) -> 'a list -> 'b list`. However, with the new feature, the type system needs to track exception raising and catching in the body of `inplace_map` and, thus, needs to impose restrictions on the argument `f`. We use the notation `-[`e_1`,`e_2`,...]->` for the arrows of functions possibly raising the exceptions $e_1, e_2, \ldots$ The type of `inplace_map` once the exceptions are supported in the programming language is `('a -[]-> 'b) -> 'a list -[]-> 'b list`.

Thus, adding a feature to the programming language—here exceptions—may require carrying out important modifications to the rest of the programming language—here the type system. This work, carried out by the programming language maintainers, ensures the consistency of the `opaque` construct. Existing programs will still work with the new version of the programming language because, in the absence of exceptions, every function of the old codebase will have empty effects—i.e., arrows of the form `-[]->` only—in the new system.

4.4.4 Opacity through runtime checks

Consider now a programming language acquiring concurrency. As pointed out in Section 3, opacity under concurrent contexts requires synchronisation which, if applied carelessly, can cause deadlocks or livelocks. This means that adding concurrency to the programming language causes the behaviour of `opaque` to evolve in potentially harmful ways. When the compiler is unsure about whether it can render the code opaque without introducing deadlocks and livelocks, it will interrupt compilation and explain the error to the user.

This means that, with the update, some projects will stop compiling. This is an argument that could deter adoption of the `opaque` treaty, and to some extent of negative features in general. However, note that without the `opaque` construct the problem also exists. Indeed, blindly adding concurrency to a programming language effectively breaks backwards compatibility in subtle and important ways: code that is correct and bug free in a sequential setting (e.g., such as code in Figure 1) can be incorrect in a concurrent setting. The `opaque` treaty does not create new issues in programming language evolution, it merely makes them explicit. It exposes the problem at compile-time with an error message rather than creating subtle bugs at run-time.

Note that there are other ways to provide concurrency. If the programming language maintainers provide cooperative threading concurrency, the compiler will instead check that `opaque` blocks do not contain primitives that cause yielding. If the programming language maintainers choose to use STM [7] instead of locks, races are handled by a transaction manager. In a way, these two proposals are less backwards incompatible.

4.4.5 Opacity with unsafe features

Unsafe features are included because programming language designers recognise the limits of their compiler's safety checks. It is meant for use when the programmer does know better and can ensure safety themselves. Within an opaque block, the use of unsafe features means the programmer, knowing better than the compiler, takes responsibility for correctness, and thus for opacity.

In the case where the programmer uses an (intrinsically) unsafe feature of the programming language—such as a type coercion, assertion check, unchecked upcast, etc.—it is still beneficial to use the `opaque` construct. For each new release,

the programmers can ask the compiler to point out all uses of unsafe features within an `opaque` block. After having read all these occurrences and made the necessary changes to the code, the programmer can be certain again of the opacity of every `opaque` block.

5. Conclusion

The pervasiveness of both the practice of and the support for invariant-breaking code is a clear indication that invariant breaking is seen as a necessity by programmers and programming language designers alike. The current abstractions that programming languages offer for bundling sets of instructions together are not resilient to feature interaction and, thus, to programming language evolution. It burdens programmers by placing responsibility on them to keep track of their whole code and of the evolution of their programming language of choice. Programming languages need future-proofing devices that allow their users to hide the real behaviour of their code. The `opaque` construct is our ("clean needles") attempt to improve programming languages concerning this particular issue. It is a showcase for negative features that apply consistently across different versions of a programming language.

References

[1] Haskell standard library modules. `http://www.haskell.org/ghc/docs/latest/html/libraries/base/`.

[2] Ocaml `Obj` documentation. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Obj.html`.

[3] Ocaml: What you gain. `http://roscidus.com/blog/blog/2014/02/13/ocaml-what-you-gain/#immutability`.

[4] Rust `unsafe` constructs. `http://static.rust-lang.org/doc/0.9/rust.html#unsafety`.

[5] Writing an LLVM Pass. `http://llvm.org/docs/WritingAnLLVMPass.html#specifying-interactions-between-passes`.

[6] GUDKA, K. *Lock Inference for Java*. PhD thesis, Imperial College London, December 2012.

[7] HARRIS, T., MARLOW, S., JONES, S. P., AND HERLIHY, M. Composable memory transactions. *Commun. ACM 51*, 8 (Aug. 2008), 91–100.

[8] JOUVELOT, P., AND GIFFORD, D. K. Algebraic reconstruction of types and effects, 1991.

[9] LEINO, K., AND MÜLLER, P. Object invariants in dynamic contexts. *ECOOP 2004–Object-Oriented Programming* (2004), 95–108.

[10] MARLOW, S. Haskell 2010 language report. `http://www.haskell.org/definition/haskell2010.pdf`.