Demystifying Dependence

James Koppel
MIT
Cambridge, MA, USA
jkoppel@mit.edu

Daniel Jackson
MIT
Cambridge, MA, USA
dnj@csail.mit.edu

Abstract

Programmers are told “depend on interfaces, not implementations." But, given a program, is it possible even to assess objectively whether such advice has been followed?

Programmers frequently talk in ways like this about dependence, but the very term, like many used in software engineering, has hitherto eluded precise definition. In this work, we resolve a variety of confusions about dependence, and present a formal definition unifying multiple varieties of software dependence, grounded in Halpern and Pearl’s theory of actual causation. This definition is parameterized by the formal system characterizing the property of interest, and by constraints on “reasonable changes" to the program. By picking different choices of formal system, one can specialize the definition to characterize several notions of dependence, including build, correctness, and performance dependences. Overall, our work provides a path to making conversations about software dependence fully objective, and might serve as a basis for future work that automatically checks forms of dependence that were previously too abstract or high-level to be candidates for tools.

CCS Concepts: • **Theory of computation** → *Program specifications*; • **Mathematics of computing** → **Causal networks**.

Keywords: dependence, modularity

ACM Reference Format:
James Koppel and Daniel Jackson. 2020. Demystifying Dependence. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! ’20), November 18–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3426428.3426916

1 Introduction

The greatest gift research can give to industry—perhaps even more than new technology—is clarity, by offering simple and clear formulations of troubling challenges. Thanks to Floyd and Hoare, we can say what it means for a program to be incorrect, down to a single line. Thanks to researchers such as Abadi, Cardelli, Cook, and Aldrich, we can definitively explain why every plugin system must use something akin to objects [2, 3, 6]. Yet it seems that as soon as the topic of code quality comes up, discussion becomes murky, and the scene shifts from a scientist analyzing an artifact to an artist giving feedback in a studio.

Enter any corporate code review, and you’ll find decisions justified by reference to “coupling," “modularity," “knowing about," and, the subject of this paper, “dependence." These terms all evade meaning, enough that, 40 years after Parnas’s pioneering papers on encapsulation [35] and dependences [36], Richard Gabriel could not coax experts to give a decisive definition of “modularity" [10]. This lack of precise definitions yields what the first author has named “citrus advice" [26], advice that is potentially useful but can backfire in the absence of a deeper understanding.[1]

With that in mind, consider these mentions of dependence from two popular writers:

> If something logical depends on the implementation, then something physical should too.
> —Robert C. Martin, *Clean Code*[31]

> I can remove this dependency by placing a simple delegating method on the server that hides the delegate.
> —Martin Fowler, *Refactoring (2nd edition)* [9]

In a town of software engineering sages, Fowler and Martin—co-authors of *The Agile Manifesto*—would be sitting on the same porch. Yet it is easy to read the two quotes as giving opposite advice about interposing a delegating method. Perhaps Martin would counter that doing so does not actually remove the dependence, in which case it becomes clear that **they are talking about different things.**

The confusion continues as we consider comments by other authors:

> Objects that depend on an algorithm will have to change when the algorithm changes.
> —Erich Gamma et al, *Design Patterns* [11]

Onward! ’20, November 18–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8178-9/20/11.
https://doi.org/10.1145/3426428.3426916

[1] The term “citrus advice" comes from the story of how the Royal Navy saved thousands of sailors from scurvy by feeding them citrus fruits, but, by virtue of not understanding Vitamin C, they then failed to notice that supply-chain changes had caused the citrus to lose its scurvy protection qualities, resulting in a resurgence during 20th century Arctic expeditions.

> In the old Web site with the background specified separately on each page, all of the Web pages were dependent on each other.
> —John Ousterhout, *A Philosophy of Software Design* [33]

From the popular press, then, we gather that depending on something that changes is problematic because it breaks code—but it can also involve webpage backgrounds, which are not even executable. Interpreting all these writers in context requires a *deeper* understanding of dependence. Without it, their pronouncements risk being citrus advice.

Our aim in this paper is to provide an objective definition of *dependence* for software engineering that captures and clarifies programmers' intuitions in the multiple contexts in which the notion appears, and is sufficiently formal to be mechanically checkable. Our definition builds on the old idea that "*A* depends on *B* if changing *B* can change *A*," but refines it using recent developments in the theory of causality and its application to programming [14, 42]. Further, it gives a classification of the *different varieties* of dependence discussed by software engineers by varying the *language of properties* and the *space of allowed changes*. For example, when applied to a language's dynamic (execution) semantics, and specialized to checking specific values of variables and allowing intervention on any variable value, the definition becomes the familiar notion of dependence used in dynamic program-slicing. When applied to a language's static (compilation) semantics, and specialized to checking the property "Does it compile?" the definition becomes the notion of dependence used in discussions of package management.

Mechanical does not mean easy, nor even automatic. Several instantiations will require information typically not present in code, but only in proofs. Answers may differ depending on some arbitrary modeling assumptions about reasonable counterfactuals, and showing dependence may require finding a unique witness in an infinite space of counterfactuals.

Nonetheless, despite some unresolved difficulties and the challenges of full automation, we hope that our work will contribute to a clearer understanding of an idea that plays a central and fundamental role in programming and software design; that by demystifying dependence we will encourage more precise and effective usage of the idea, and that our framework will prove to be a fruitful basis for subsequent research.

2 Nine Dependency Puzzles

If you open a popular software engineering book, you'll be sure to find lots of advice about avoiding and reducing dependencies—in the context of design patterns, package management, use of third-party software, and so on. Yet, viewed under a microscope, contradictions emerge. Drawing on this folk understanding of dependence, we've identified a variety of puzzles:

1. If a call from procedure *A* to procedure *B* generally implies that *A* depends on *B*, then does a round-robin scheduler—which invokes a collection of tasks—in turn depend on each of the tasks?
2. If *A* writes a file and *B* reads it, there is clearly a coupling between the two induced by the assumption of a shared format. But which depends on which? Similarly: a client serializes messages received by a server. If either the serializer or the deserializer changes, the other will break. Do they both depend on each other?
3. If the lack of dependence of *A* on *B* means that a change to *B* cannot affect *A*, what about a change to *B* that introduces a new dependence, for example by modifying a previously unreferenced global variable used by *A*?
4. Furthermore: If *A* depends on *B* when a failure of *B* can lead to a failure of *A*, but any module may crash the program (e.g.: by stack overflow), then does every module depend on every other module?
5. On the other hand: If *A* depends on a module *B*, but checks the result and uses a slower and more reliable service *C* if *B* fails, then failure of *B* no longer implies failure of *A*. Does that mean that *A* does not depend on *B*?
6. Dependency inversion supposedly eliminates a dependence of *A* on *B* by passing *B* to *A* at runtime. But *A* will still fail if *B* fails. Then so why doesn't *A* still depend on *B*?
7. The dependency relation is usually treated as transitive, with cycles evidence of poor design. But if dependence of *A* on *B* means that *A* cannot function without *B*, what does a self-dependence mean?
8. Libraries are built independently and thus should not depend on application code. But a hash table implementation will give incorrect output if keys have inconsistent implementations of `equals()` and `hashcode()` methods. Does that mean that such library classes depend on their callers?
9. A robot controller is written using n-version programming. At each timestep, the robot turns left or right based on a majority vote of 5 different controller implementations. Suppose all 5 implementations implement different algorithms which somehow always give the same answer. Then no change to any single implementation can alter the robot. Does this mean the robot does not depend on any of them.

The simple solution to most of these puzzles is that questions like "Does module *A* depend on module *B*?" are malformed. Rather, each module has a variety of properties that can be

impacted and changes that can be made, leading to different kinds of dependence. The next section introduces the main concepts needed to frame dependence properly. Later, after developing our new definition, nwe resolve all these puzzles in §8.

3 What's in a Dependence Query?

3.1 Dependence Is about Properties, Not Programs

A company wants to add analytics to their mobile app to track user engagement. But they don't want to be tied to any one vendor, so they build a wrapper around the analytics library, so as not to depend on it. This pays off because, one day, they decide this analytics library is using too much bandwidth, so they switch to a different one that promises to be more lightweight. Only the wrapper code needs to be changed.

But, if the application didn't depend on the analytics library, how could changing the library make a difference?

We have previously argued that the concept of dependence only makes sense relative to some correctness property [24, 25]. For example, while the company may have successfully shielded the *functional correctness* of the app from the functional properties of the analytics framework, it did not block the app's *resource requirements* from those of the library. We extend this insight: there is no dependence of a code fragment A (on some other code fragment B), but rather a dependence of some *property* φ of A.

As a corollary, questions of dependence cannot be resolved without determining the property of interest, and so in general cannot be answered from the source code alone. For example, imagine an app that runs `showPopup("No connection")` when it tries to download an update and finds no Internet connection. If the correctness of the system merely requires that it show a popup with this message in this event, then there is no dependence on code that sets the default formatting of popups. But if there are specific requirements on the visual design of this particular popup, then it does.

Observing software writing in the wild, the most common translation of "A depends on B" seems to be "some relevant correctness property of A depends on the code of B", followed by a substantial minority of cases in which A's property is instead "successful compilation." We explore the landscape of such static vs. dynamic properties in the next section. When people disagree about whether a module depends on something, we hypothesize that the most common cause is this ambiguity over which property is being discussed.

3.2 Dependence Is Relative to a Semantics

According to many writers on software, the moment you import a new package into your project, it becomes a "dependency," yet the project does not "depend" on it until it's used. If the dependency in question is PostgreSQL, but the app would work with MySQL without any changes, then the code "does not depend" on PostgreSQL.

Software writers effortlessly flit between different varieties of dependence. The differences between these can be exposed by asking the question "What change in the dependency is relevant to the dependence?" For the first implied definition, that would be introducing any kind of build error. But for the second, it would be any change that alters the behavior of the dependee, and for the third, it would only be changes that cause it to differ from the SQL standard.

These varieties of dependence differ not just in the property being queried, but in the semantic relation being considered. Thus dependence in dynamic slicing may be formulated in terms of some execution relation ($\rightsquigarrow$), yet the question "Does it compile?" is not even askable in terms of ($\rightsquigarrow$). Yet that question *would* make sense in terms of some relation that models compilation, perhaps a composition of systems for parsing, typing, and linking.

Let us give rough definitions of the three varieties of dependence implied by the paragraph above. We see that each corresponds to a different formal system and property:

1. Package A depends on package B if A would not compile without B. This is **dependence in the static semantics** of the language.
2. Package A depends on package B if the execution of code in B is required for A to obtain its result. This is **dependence in the dynamic semantics** of the language.
3. Package A depends on package B if some special property of package B, not guaranteed by some broader spec (e.g.: the SQL standard), is needed for A to meet its requirements. This is **dependence in a correctness logic**.

In the example from the start of this section, a program that uses PostgreSQL, but only through queries that would work equivalently in any other database engine, depends on PostgreSQL in sense #2 but not sense #3. And returning to the Martin Fowler and Robert Martin quotes from §1, Fowler's "removing a dependency by adding a delegation method" appears to be referring to a dependence in the static semantics, as does Martin's "physical dependence." Martin's "logical dependence," on the other hand, appears to refer to dependence in either the dynamic semantics or correctness logic. (Additional context makes clear that the latter is intended.)

These parameters—of the language of properties and the relevant semantics—promise, less straightforwardly, to enable precise questions about the dependence of non-functional properties. For example, in the three systems above, questions like "Does this web page depend on the background color of another web page," from Ousterhout's example in §1, could not even be meaningfully asked. But now one can imagine a semantics encoding properties such as "visual consistency," or "usability" that might be evaluated in user studies or perhaps in the context of probabilistic models. Now questions such as "Does the usability requirement φ

of `Page1.html` depend on the `background-color` attribute in `Page2.html`" become well-posed. And there are already examples of using formal semantics to check similar properties; in particular, *visual logic* [34] has been used to check properties such as "good contrast" between text and background.

A more tractable kind of non-functional requirement is performance. Imagine a program using one of these two versions of the `max` function over an array:

```
public int max1(int[] arr) {
  assert(arr.length > 0);
  int result = arr[0];
  for (int i = 0; i < arr.length; i++) {
    if (arr[i] > result) result = arr[i];
  }
  return result
}

public int max2(int[] arr) {
  assert(arr.length > 0);
  int[] arr2 = copy(arr);
  while (!isSorted(arr2))
    arr2 = randomPermutation(arr2);
  return getLast(arr2);
}
```

Under most logics, there would be no way to distinguish `max1` from `max2`, even though the latter's unbounded running time is sure to upset many programmers. However, in a semantics that considers timing, such as Resource-Aware ML [21] or Timed ML [43], a question like "Does this application's performance spec rely on this other module's performance?" becomes well-posed.

3.3 Dependence is Relative to Permitted Changes

Depending on something means relying on a thing being what it is and not something else. So a dependence of A on B must be witnessed by a potential change to B that would impact A.

When a change may alter the dependence structure, or have global effects such as crashing the program, then the graph of susceptibility is the standard hairball: everything depends on everything else. This is the question raised by our third and fourth puzzles. For a dependence analysis to be useful, there must be some limits on what changes may be considered.

And it is not enough to have only broad restrictions such as "no changes that add new dependences." Consider the *Observer* pattern, where an object maintains a dynamic list of observer callbacks and invokes them upon certain events. The common justification is that doing so removes the dependence of the object on its observers. Yet there are countless ways an observer may interfere with an observee, from competing for shared resources such as locks and files, to raising an uncaught exception, to manipulating the runtime stack at the binary level. The intuitive justification—of non-dependence, and hence non-interference—can hold weight, but only under assumptions about permissible behavior which are not universally applicable.

Thus, dependence is defined relative to what changes are under consideration. Dependence queries about compilation consider changes in well-formedness; dependence queries about execution consider changes in runtime state; and dependence queries about modular correctness consider changes to the specs and guarantees of components. Within each of these categories, there must be some additional constraint on what changes are permitted, which we deem a *super-spec*.

Super-specs are discussed more in §5.2.

3.4 Dependence Is Causality

The semantics of dependence must differ from traditional program semantics in at least one fundamental respect. Traditionally, formal semantics aims to explain what the result will be of taking a program in its given form, and executing it (in terms of the values returned and effects produced). Verification, for example, is then about whether these results comport with expectation, as expressed in a specification.

The semantics of dependence must be different, because the concern is not what the result of executing this given program will be, but rather what the result will be of executing *another* program—albeit one closely related to this one. This other program may be the program that this program will evolve to as the design changes; or perhaps it's the program after an unwanted perturbation (such as an attack by an adversary); or perhaps it's simply the program that this program might have been had it been incorrect.

Central to dependence, therefore, is the concept of a *counterfactual hypothetical*. This leads to a startling observation: **every one of the examples from §2 has an isomorphic example in the domain of causality in the physical world**. In fact, two of these examples were directly based on examples from the causality literature (more on this in §8). We give translations of Examples 1-4 below, and leave the rest as an exercise to the reader.

1. A worker is packing boxes into a shipping container. Can the contents of any particular box cause the worker to succeed or fail?
2. An engineer designs a socket. Do the engineer's decisions cause the design of the corresponding plug, or vice-versa?
3. If Beatrice did not cause Bob's death, does that mean Beatrice could have done nothing to save Billy?
4. If any neighbor could have cut electricity and gas to your house, does that mean that every neighbor's actions are a cause of your being able to cook dinner at home?

These examples suggest that any advances in the theory and definition of causality might be translated into a better

understanding of software dependence. And so, in the next section, we introduce some of the concepts that, in the past 30 years, have transformed causality from a field of philosophy to a field of computer science.

4 Background: Actual Causation

In this section, we explain the recent work on the definition of actual causation, before we discuss how to apply the underlying ideas to the question of software dependence.

Theorists categorize causality into two categories. "Type causality" (also called "general causality") deals with general statements about categories such as "A stray spark causes fire." "Actual causality" deals with statements about specific events, such as "A stray spark in Thomas Farriner's bakery on Pudding Lane caused the Great Fire of London on September 2nd, 1666." As dependence deals with questions about specific programs and specific scenarios, actual causation is the one relevant here.

There have been several proposed formal definitions of actual cause, and there is no standard for determining whether a definition is correct—only arguments that it is is useful and produces answers that match intuition. In the remainder of this section, we give a crash course in actual causation, in preparation for developing a definition suitable for software. All of these details appear in the first two chapters of Halpern's book [14].

4.1 Structural Causal Models

In this section, we explain **structural causal models**, also known as **causal graphs**, the main objects of study in the Pearl school of causal inference. More detailed introductions can be found in either Pearl's [38] or Halpern's [14] books, as well as in many self-contained papers on causality [4, 13, 15].

Structural causal models view the world as a **deterministic program** on **non-deterministic inputs**. For instance, consider the classic example of a system where grass may be wet or not ($W = 1$ or $W = 0$), based on whether the sprinkler is on (S) and whether it rained (R). The state of the sprinkler and rain are determined by unknown background factors, captured in the variables U_1 and U_2, but the sprinkler will never be on when it has rained. Following are a series of equations (or rather, imperative assignment statements) determining the state of the system; Fig. 1 depicts the structure as a causal graph.

$$\begin{aligned} R &= U_1 \\ S &= U_2 \wedge \neg R \\ W &= R \vee S \end{aligned}$$

U_1 and U_2 are *exogeneous variables*, and may be probabilistic. R, S, and W, in contrast, are *endogenous* variables, and are given by the deterministic equations above, which may be read as a straight-line imperative program. The distribution of U_1 and U_2 thus induces a distribution on R, S, and W.

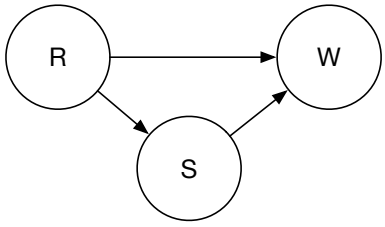

Figure 1. Causal graph for the sprinkler example. Exogeneous variables omitted.

Actual causation deals with specific scenarios rather than general models. Once a setting of the U_i has been picked, the system becomes completely deterministic, allowing queries about *why* a specific combination of variable values occurred.

An assignment of values $\vec{u} = \{u_1, \ldots, u_n\}$ to variables $\vec{U} = \{U_1, \ldots, U_n\}$ is denoted $\vec{U} = \vec{u}$, and is called a **context**. With some abuse of notation, it is frequently written as just $\vec{u}$. Similar notation is used for settings of other sets of variables.

More formally, a model M consists of a set of endogenous variables $\vec{X}$ and a set of exogenous variables $\vec{U}$, where each variable v takes values from a domain $\mathcal{D}_v$, and a set of equations $\{X_i = f_i(\vec{X}, \vec{U})\}$. Most settings limit their attention to *strongly recursive* models, in which the ith equation expression f_i ignores all variables $\{x_i, \ldots, x_n\}$ that are assigned later. Strongly recursive models may thus be interpreted as straight-line imperative programs.[2]

Given a model M and context $\vec{u}$, the judgment $(M, \vec{u}) \models X_i = x_i$ means that, when $\vec{U} = \vec{u}$, the equations of M entail the equality $X_i = x_i$. More generally, if φ is a logical formula in terms of the X_i (called an **event**), then $(M, \vec{u}) \models \varphi$ if φ is true under the unique assignment to the X_i entailed by the formulas of the model in the context $\vec{U} = \vec{u}$. Actual causation deals with the question of which values of variables may be considered to be "causes" of the event φ, i.e., which intermediate variables were relevant to φ being true.

Of importance is that, while the event φ may be an arbitrary formula, the causes may only be assignments to variables (or sets of variables). A formula like $R = 1 \vee S = 1$ is not permitted to be a cause, but a formula like $R = 1 \wedge S = 1$ is.

Intervention is the capability that distinguishes causal models from probabilistic models, which contain no more information than a count of how often each combination of variables occurs. It is the feature that justifies a model's

[2] Confusingly, while these programs may be "recursive" in the computability-theory sense of "computable," a programming languages theorist might be tempted to call them "not recursive," as they correspond to acyclic graphs!

interpretation as an imperative program rather than as a relation among variables. The intervened model $M_{X_i \leftarrow x_i}$ is a model that is identical to M, except that $f_i(\vec{X}, \vec{U})$ is replaced with the constant x_i. For example, if M is the sprinkler model defined above, then $M_{S \leftarrow 1}$ ("M where the sprinkler has been forced on") is defined by the following equations:

$$R = U_1$$
$$S = 1$$
$$W = R \vee S$$

Intervention is fundamentally outside the ability of pure statistical methods such as conditioning (discussed extensively in Pearl [38]). Note that, with this intervention, it becomes possible to speak about what happens when it rains *and* the sprinkler is on, which could never occur when passively observing the original model.

Intervention extends naturally to sets of assignments, $M_{\vec{X} \leftarrow \vec{x}}$, and the judgment $(M_{\vec{Z} \leftarrow \vec{z}}, \vec{u}) \models \varphi$ can also be written as $(M, \vec{u}) \models [\vec{Z} \leftarrow \vec{z}]\varphi$.

We have now defined all the tools needed to express actual causation.

4.2 But-for Causation

As a build-up to our preferred definition, we first present a simpler definition of actual causation called "but-for" causation, erroneously called "scientific causality" in law schools worldwide. In but-for causation, A is a cause of B if, but for A happening, B would not have happened. More formally:

Definition 4.1. $\vec{X} = \vec{x}$ is a **but-for cause** of φ in $(M, \vec{u})$ if there is an $\vec{x}' \neq \vec{x}$ such that:

1. The setting $\vec{X} = \vec{x}$ actually occurred: $(M, \vec{u}) \models \vec{X} = \vec{x}$
2. The event φ actually occurred: $(M, \vec{u}) \models \varphi$,
3. If $\vec{X} = \vec{x}'$ instead, φ would be false: $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}']\neg\varphi$

In the sprinkler/grass model, with $U_1 = 1$ and $U_2 = 0$, rain ($R = 1$) is a but-for cause of wetness ($W = 1$), because, without the rain (and since the sprinkler would not be on), the grass would not be wet. However, with $U_1 = U_2 = 1$ (and hence $R = 1, S = 0$), rain ($R = 1$) is not a but-for cause, because, were it not raining, the sprinkler would come on, and the grass would be wet anyway.

For an example in a program context, a program starting is a but-for cause of it printing any output: if it had it not started, there would be no output. However, the second sprinkler example above shows that, for over-determined events, but-for causation does not align with intuition: the rain should be the cause of the grass being wet—it's what made the grass wet, after all! This situation is isomorphic to Example 5 in §2, where an app uses some service (say, one payment service), and, if it fails, uses a different one. But-for causation would say—implausibly—that, if the first payment service succeeded, its success would not have been a cause of the app successfully processing a payment. Causality researchers call this *preemption*. More generally, whenever there is a backup in case of failure, no single happening can be the cause of success.

Such undesirable answers in but-for causation have led many to seek alternate definitions. At the very least, but-for causation is unsuitable as a basis for the theory of dependence, as it would indicate that the app does not depend on the first payment service, or at least no more than it depends on any unused payment service. Using such a definition of dependence in dynamic program slicing would construct a slice without including code from any payment service at all. We therefore turn to a more modern definition of causation.

4.3 The Halpern-Pearl Definition

After witnessing problems with but-for causation and other attempted definitions, Halpern and Pearl began working in the early 2000's on a new definition that was both rigorous and yet still captured our intuitions about the causes of events. In 2015, after discovering problems with two earlier attempts, Halpern published his current definition of actual causation [13], which has an imperfect but excellent track record of aligning with intuition.

The key idea that differentiates the three Halpern-Pearl definitions from but-for causation is the idea of a *contingency*. Because of over-determination, it may not be possible to change a variable X_i to make φ false, even if X_i was clearly used in the process that made φ true—because changing it may have a secondary effect that counteracts the change. However, by *blocking this secondary effect*, it becomes possible to observe that X_i's value was indeed involved in the chain of events leading to φ. Halpern's 2015 definition achieves this by allowing the change to the causing variables $\vec{X}$ to be observed in the context of some intervention (a "contingency") which forces a witness set of variables $\vec{W}$ to keep the values they would have taken had $\vec{X}$ not been modified.

Definition 4.2 (Actual cause (Halpern 2015))**.** $\vec{X} = \vec{x}$ is an actual cause of φ in $(M, \vec{u})$ if the following three conditions hold:

AC1. $(M, \vec{u}) \models (\vec{X} = \vec{x})$ and $(M, \vec{u}) \models \varphi$

AC2. There is a set $\vec{W}$ of variables and a setting $\vec{x}'$ of the variables in $\vec{X}$ such that if $(M, \vec{u}) \models \vec{W} = \vec{w}$ then

$$(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}]\neg\varphi$$

AC3. $\vec{X}$ is minimal; no proper subset of $\vec{X}$ satisfies conditions AC1 and AC2.

It's easy to show that every (minimal) but-for cause is also a Halpern 2015 cause. But the converse is not true. For example, in the sprinkler example with $U_1 = U_2 = 1$, $R = 1$ is a cause of $W = 1$, witnessed by the contingency $S \leftarrow 0$. In other words, the rain was a cause of the grass being wet, because, were there no rain, *and the sprinkler were kept in the off state*,

then the grass would not be wet. This judgment aligns with intuition. But is the mechanism of contingency the reason why?

4.4 Which Definition? Neither. Or Both.

In the writing of this paper, we switched between several definitions of causation, beginning with a direct analogue of the Halpern 2015 definition, and then one more similar to but-for, and then the current one, which is superficially more similar to the Halpern 2015 definition[3], but spiritually closer to but-for, as we found few examples of contingencies in a software context, and those we did find could be better justified without them.

All proposed definitions are evaluated by humans' highly-developed causal intuitions. And intuition states that, when it rains, the rain causes the grass's wetness. But compare: suppose instead there is no rain, but you activate the sprinkler 5 minutes before it was scheduled to turn on anyway. Though this graph is isomorphic, many would say your actions were not an actual cause of the grass's wetness an hour later. Perhaps then, the causal judgment about rain stems not from the weather, but from the differences between tapwater and rainwater; not from past actions, but from a more precise model of the end-result.

Similarly, for the example of a fallback payment processor, there is a material difference depending on which one is used (seen on the bill from each vendor), which supports the intuition that a successful transaction did depend on whichever processor was used. [4]

Throughout the remainder of this paper, we shall frequently point to undesirable results from our definition, and to alternate answers given by other definitions. Causality is still an immature field, despite its growing popularity in AI and other areas. And so, as long as there is debate over the proper definition of actual causation, there shall be debate over the exact logical formula defining dependence.

We now proceed to construct a definition of dependence based directly on actual causation. But first we must determine how to lift interventions from the setting of causal models to the setting of programs, and the evaluation of properties from boolean combinations of assignments to arbitrary facts about a program's static semantics, dynamic semantics, and correctness.

[3]And even more similar to the Halpern-Pearl 2005 definition [18], not described here.

[4]As a real example of this: the multi-language Cubix framework [28] uses an outdated-yet-robust Java parser, but falls back to a newer-but-buggy parser when it encounters newer syntax. It makes sense to say that the execution of Cubix depends on which parser was used in part because they produce different output; it would makea less sense if they had the exact same behavior save intermittent failure.

5 Causality in Programs

5.1 Intervention as Program Transformation

Intervention in a causal model is setting a variable to a constant. As structural causal models are straight-line programs, the naive extension to general programs is to modify the program by inserting assignment statements setting some variables to constants; this is the approach taken in early work by Icard [23].

This, however, is not sufficiently general for our purposes. The shift from straight-line to arbitrary programs is not merely about admitting more complex code *between* assignments to variables, but a fundamental change in the nature of assignment itself: that a variable can take on an entire family of values (or perhaps not exist at all). The power of an intervention must be upgraded accordingly.

From another lens, if the functions f_i defining each endogenous variable are the "lines of code" of a causal model, then **interventions may be seen as program transformations**. Indeed, researchers in pure causal inference have recently adopted a view very much like this, with a formalism in which interventions may set a variable not just to a constant, but to an arbitrary (perhaps stochastic) function of its predecessors [7].

When generalizing models to arbitrary programs, therefore, this suggests taking general program transformations as the analog to interventions, an idea introduced in our previous work formalizing a probabilistic programming language with counterfactuals [42].

If the interventions are program transformations, what are the variables? An easy answer is: fragments of the program state at arbitrary points in the execution. This raises the challenge of actually specifying such an execution point to perform an intervention, with identifiers such as "the value of `i` in the 5th iteration of the loop in function `foo` when called from a signal handler" being quite unstable across intervention. This is closely related to execution-point identification in dynamic analysis [41], and to mutation of execution traces in probabilistic programming [44]. Finding a universally satisfactory treatment is still an unsolved problem, though both cited papers offer effective defaults in their respective contexts, as does our work on counterfactual probabilistic programming [42]. In our example formalizations for specific systems in §7, we will artfully dodge these issues.

With this generalization, the definition of actual cause is already starting to look like a more refined version of "A depends on B if a change to B can change A." But, before we dive into the details, there is still one more aspect of structural causal models to generalize, one that was subtly sneaked into the definition.

5.2 The Concept of Valid Intervention

In the mid-2000s, Liblit and Aiken developed the Cooperative Bug Isolation (CBI) project [30]. In this project, their system would instrument programs to record at various points whether certain predicates such as "$x > y$" were true, in the style of Daikon [8]. They would then release the programs to users, and have the systems send statistics back home, so that they could detect which predicates were correlated with program failures.

But a problem emerged: the strongest correlates were often not ones useful for pinpointing a bug. Instead, they would often be ones correlated with a large input, as these were more likely to contain edge-cases. Aiken later told us in conversation that the inability to identify useful predicates was the greatest unsolved problem of the project.

The essence of the difficulty is that users of the CBI system were most interested in predicates that showed the cause of bugs—not just those that were correlates. And this is a much harder problem, even just to define. But below, we do define it—and find that it requires information not in the program.

Imagine CBI discovered that the predicate `strlen(s) > 1000` was correlated with a program crash, and an agent with limitless computational power was trying to determine whether this relationship was causal. To apply either of the definitions from §4, the agent would have to intervene in the program to make `s` be some string of length greater than 1000 and then look for the crash—and this is underdefined, as it clearly matters which string is used. In general, there is no good way to intervene on a predicate; this is the reason why the Halpern-Pearl definition only allows sets of assignments as causes, even as events may be arbitrary boolean formulas. But now suppose that the string were a structure with a separate length field, and the predicate was `s.length > 1000`. Now it is quite trivial to intervene and set `s.length` to something else—and then the program would crash irrespective of the pursued bug, **because this is not a valid intervention**.

In structural causal models, it is permissible to intervene in a variable to set it to any value in its domain. And that "in its domain" restriction hides a lot of work.

It has been noted [22] that structural causal models are ontologically similar to propositional logic, where each variable stands alone, and there are no composite structures. Programs are not like this, of course. The reason, for instance, that setting `s.length` to an arbitrary value is invalid is that it likely violates `s`'s *representation invariant* relating the length field to the string buffer itself.

Meanwhile, in structural causal models, restrictions on valid interventions are encoded into the choice of variables and their domains, so that causality questions may have different answers depending on modeling choices (they are *model-relevant*). For a classic example where adding variables changes an answer, in the Halpern-Pearl definition of §4.3, if two people throw rocks at the same bottle and it shatters, it cannot be shown that one person's throw but not the other's is a cause of the bottle shattering, unless there are variables added representing which one hit first. And for an example where changing a variable's domain changes causality, consider Example 4 from §3.4: it cannot be shown that your neighbor's action NA is a cause of you cooking dinner unless the domain includes a setting $NA = a_{\text{cut}}$ representing them cutting your power and gas lines (and the equations defining the model are augmented accordingly).

There are several ways to counteract this, such as adding a parameter for how "abnormal" an intervention may be, but, no matter what extensions are made to the formalism, the first line of defense is to **carefully construct the model** to permit exactly the relevant counterfactuals. So far, there is no analogue to this when checking causality in programs. By default, interventions as we have described them may insert arbitrary code and set variables to arbitrary values of their type, though there is no reason to believe that the set of reasonable values a variable may take should coincide with the domain of the standard types in a language.

As the analogue of the carefully-chosen variable domain in structural causal models, we stipulate that there must be some additional restrictions on what interventions are valid. As it is perfectly reasonable to perform an intervention that modifies a module's spec (to, say, determine whether another module's correctness relies on a specific property), for lack of a better term, we deem these extra restrictions a **super-specification**. We first described the need for such restrictions in our work on counterfactual probabilistic programming [42], but we swept it under the rug by assuming some unspecified static analysis to prevent invalid interventions. In this paper, we develop an explicit notion of super-specifications. Without such super-specs, there may be multiple possible answers to questions of dependence. Thus, even after choosing a specific property, it may be impossible to decisively determine dependence from inspecting the code alone. [5]

When changing modules, a common super-spec is some kind of frame condition stipulating which variables may be modified or what effects may result. As a spoiler, this is the resolution to two of the examples in §2, where dependence on a module varies with whether it's permissible to alter that module to crash or mess with another module's variables. When considering dependence of variable values on other variable values, a useful super-spec is to only permit interventions that respect a data structure's representation invariants, as in the string example. A more basic super-spec is to rule out interventions that result in an ill-formed

[5] But, at least for dependence on program state subject to a super-spec of data structure invariants, there is empirical hope for useful answers in the absence of user-provided restrictions: Zeller's experience with the HOWCOME tool [45] suggests that randomly mutating a valid program state is moderately likely to produce another valid program state.

program—but when considering build dependencies, such interventions are of great interest.

The super-spec assigned to a program element may differ with the nature of the query. One strand of research in actual causation allows considering not a single model but a range of models with a *normality ordering* [14, 16], where, for example, a more "abnormal" model for fire may be allowed to consider the case where there is a stray spark but no oxygen. Such judgments have analogues in software as well: a software engineer may conclude that one module's failure cannot affect another's, while a security engineer, unwilling to rule out a code-injection vulnerability, might not.

The need for a super-spec implies that the program text will generally not contain enough information to check dependence. This might seem undesirable, but we argue that this is necessary, and any purported definition that lacks this property will be flawed. If dependence is witnessed by possible changes, then, as the programmer's intentions influence what kinds of changes are possible, said intentions must also influence whether a dependence exists.

And now, with a fuller framework for extending interventions to programs, we are ready to translate notions of causality in the physical world into a notion of dependence in the world of software.

6 A General Definition of Dependence

The last section identified challenges defining causality in programs. This section answers these challenges by creating a general, formal definition of causality in program semantics, built atop state machines as a common language representing many kinds of semantics.

Causality is built on notions of components, properties, allowable interventions, and deterministic execution. It can thence be developed in many settings, from Halpern and Pearl's causal graphs, to databases and relational algebra [32], to continuous-time models of biochemical reactions [29]. We chose the setting of state machines as one which is expressive enough to model most forms of program semantics, while still permitting a simple, visual explanation. In this section, we both present the generalized definition of dependence in deterministic state machines, and **simultaneously give an example application** to dependence in dynamic semantics (which roughly corresponds to dynamic slicing), which requires little transformation to express as a state machine. §7 then applies it to develop several other forms of dependence.

Programs are substantially more expressive than structural causal models, and so any model of program semantics will exhibit phenomena with no correspondence in causal graphs. The two primary problem addressed by this section is how to generalize the notion of a variable.

An easy first thought when generalizing structural causal models to programs is to build a graph whose variables are program variables, where intervention is forcing a variable to a value, as done by one early attempt [23]. This is clearly not suitable for applications where the objects of dependence are high-level properties or otherwise not program variables. And it is not even suitable for expressing dependence in execution semantics, as it provides no way to, e.g., intervene on a variable but only in the 5th iteration of a loop. A state machine state, on the other hand, may represent an entity much finer or much coarser than a program location. And, in many cases, the state machine will correspond to a *flattening* of some hierarchical structure such as a typechecking derivation or a correctness proof, meaning it can provide arbitrary amounts of context.

As our running example, we use Imp, an imperative language with variables (integer only), assignments, arithmetic, boolean expressions, conditionals, and loops, all defined the usual way.

Actions, States, and Properties. In our setting, a *system* is a state machine (S, I, L, A, M) where S is a set of states, $I \subseteq S$ a set of initial states, L a set of action labels, and $A \subseteq S \times L \times S$ is a set of labeled transitions ("actions") on S. The available mutations, $M : S \to \mathbb{P}(S)$, are described below. The machine is required to be deterministic, so that, from a given initial state, there is exactly one (possibly empty) maximal path.

For example, to define a state machine for Imp which has[[] statement-level granularity: States are values of the current environment μ together with the current program counter pc, where pc indicates either the beginning of an assignment, or the condition of an if-statement or loop. Transitions correspond to executions of single statements, and their labels are unique identifiers for those statements. Fig. 2 shows a portion of the state machine `x := x + 1; y := x + 1; halt`, with the three statements numbered 1, 2, and 3, respectively; the full state machine has infinitely many initial states, corresponding to the infinitely many possible starting values of x.[6]

A *trace* is a maximal sequence of actions $(a_0, l_0, b_0), \ldots$, such that a_0 is an initial state ($a_0 \in I$) and $b_j = a_{j+1}$ for all $0 \leq j < n$. A finite trace corresponds to a terminating execution, and an infinite trace to a nonterminating one. For example, each of the columns in Fig. 2 is a trace.

We assume there is some language of *properties* on traces. We write $t \models \varphi$ if property φ is true for trace t. A property can equivalently be defined as a set of satisfying traces.

[6] This example shows how the choice of many initial states allows these state machines to model an initial input, in this case the initial value of x. It is similarly possible to model other aspects of the outside world, such as interactive input, as an exogeneous variable in the initial state containing an answer to all possible queries to the outside world. There are, after all, only countably infinitely many of them, and it is permissible to have similarly infinitely many initial states.

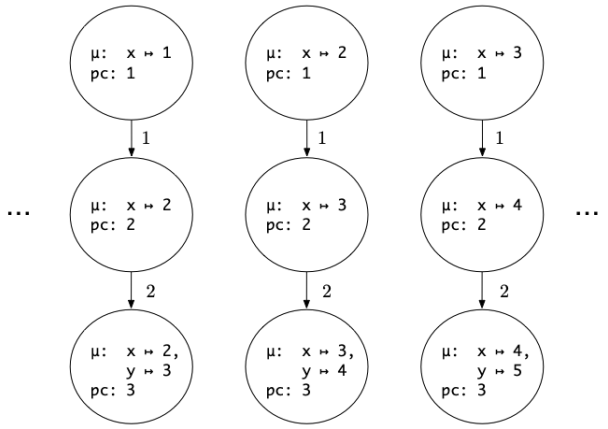

Figure 2. Fragment of state machine for program `x := x + 1; y := x + 1`

Permitted Mutations. Associated with each state a and its unique outgoing transition (a, l, b) is a set of available *mutated actions* or *available interventions* $M(a)$. Each available intervention is an alternate outgoing transition with the same variable, i.e.: (a, l, b') for some b'. This set is typically given by some super-spec ρ, where any transition (a, l, b') that satisfies $\rho(a, l, b, b')$ is an available mutation.

For Imp, if a is a state corresponding to an assignment $x := e$ and (a, l, b) its unique outgoing transition, then the super-spec $\rho(a, l, b, b')$ is: in the successor state b', no variable other than x may be modified, and the program counter of b' must equal that of b. If a is a state corresponding to the program point immediately before executing a conditional **if** e or **while** e, then the super-spec is that the new program counter must correspond to either the next statement to be executed if e is true, or the next statement if e is false. In summary: any assignment may be modified, and any branch taken may be flipped to the other alternative. However, neither assignments to new variables nor loops may be inserted.

Causality. With this setup, we can adapt the Halpern-Pearl definition of causality to the setting of state machines, and use it as a formal definition of dependency.

Defining the first intervention in a trace is straightforward: replace one edge a in a trace with a permitted mutant a', and continue execution. But including any additional, compensating changes is trickier. Unlike the static setting of causal graphs, in programs or the state machines modeling them, a small intervention may cause execution to take a completely different path. For this, we offer a broad, perhaps too broad, remedy: allow any permitted mutation on the new states as a "witness" to the causality of a'. [7]

[7]This renders our definition of causation more similar to Halpern and Pearl's 2005 definition [18], with a more liberal allowance on compensating changes, than the 2015 definition [13]. (These are referred to, somewhat confusingly, as the "updated" and "modified" definitions respectively in Halpern's book [14].)

Definition 6.1. Given a trace t, property φ, and an action a, we say that a *causes* φ (also: is a *it* cause of φ) in t iff:

1. a occurs in t
2. $t \models \varphi$
3. There exists a' an available intervention for a, and $\vec{w}'$ a set of available interventions to some set of actions $\vec{W}$, such that, if t' is the unique trace resulting from replacing a with a' in t and continuing execution with any encountered transition in $\vec{W}$ replaced with the corresponding transition in $\vec{w}'$, then $t' \not\models \varphi$. In this case, a' and $\vec{w}'$ are *witnesses* to the dependence on a. (Alternatively, the dependence *relies on* interventions a' and $\vec{w}'$.)
4. There is no trace in which any element of $\vec{w}'$ is itself a cause of φ.

For example, in the trace obtained by execution `x := x + 1; y := x + 1; halt` with start state $[x \mapsto 1]$, the first statement (i.e.: the outgoing edge of the top-left state of Fig. 2) is a cause, and hence a dependency, of the proposition $x == 2$ being true in the end state, as witnessed by the intervention replacing $x := x + 1$ with $x := 3$ (equivalently: mutating the top-left edge of Fig. 2 to target the center node). The statement $y := x + 1$ is not a cause, as no mutation is permitted to this action which affects x.

For a more complicated example which uses compensating changes, consider the code:

```
1: a := 1
2: b := 1
3: if (a == 1) then
4:     x := a
   else
5:     x := b
```

In the unique trace of this code from an empty start state, the action corresponding to line 1 is a cause of the proposition $x == 1$ being true in the final state. This is witnessed by mutating line 1 to `a := 2`, *and* by the compensating change of forcing the conditional to enter the "true" branch (i.e.: mutating the outgoing edge from the state $(\mu : [a \mapsto 2, b \mapsto 1], \text{pc} : 3)$ to target $(\mu : [a \mapsto 2, b \mapsto 1], \text{pc} : 4)$).

Awkwardly, by similar reasoning, line 2 is also a cause! The authors disagree over whether this is reasonable behavior, with the argument in favor that it interferes with the ability to change x by changing a and would be included in any static slice, and the argument against being that there is no dataflow from this line to x in the actual trace.

If a developer wishes to build a tool that reports line 1 but not line 2 as a dependence, they have many options for tweaking our definition. An easy choice is to restrict compensating changes so that they may restore the control flow of the original trace, but not introduce a new control

flow. However, we found such a distinction difficult to justify: it nonetheless enters different states, and it is quite possible for two traces that follow the same control path to have radically different behavior. [8] While such heuristics may be useful in applications, our conclusion is to question the soundness of the idea of dynamic slicing itself, for it relies on a distinction between control- and data-dependencies which is semantically untenable.

Lifting Actual to Type Causality. We defined dependence on a state, or, equivalently, on the outgoing transition from a state. But if a specific state is an actual cause of an event, then it can be said that a category containing this state is a general cause of an event. And thus, dependence on a state induces a dependence on any coarsening of the state. For example, "dependence on a label" can be defined as dependence on some transition with that label. In our running example on Imp, labels are effectively line numbers, so the development above also gives a definition for dynamic dependence on a line. And similarly, module dependences might be defined by coarsening to all the transitions executed by a given module.

This section has lifted dependence from vagueness into an objectively-checkable formal definition, but its edges are still blurred by the same imperfections that caused Halpern and Pearl to go through four definitions of actual causation over 20 years [13, 17, 18, 37].

7 More Example Formalizations

7.1 Linking Dependence

Module M has a build dependency on module N if M cannot be built in the absence of N, which typically means that M has a syntactic reference to N. We should hope to find that this is a special case of the definition of §6—and indeed it is, applying the definition to the static semantics of a language.

This section uses a simple model for whether a program successfully builds: whether a term in the simply-typed lambda calculus (STLC) is well-formed. This takes the form of a state machine which takes in a queue of bindings, and subsequently attempts to add each to its typing context. If a binding is in the typing context upon termination, then it is well-typed, and hence "built" successfully[9].

[8] Indeed, electronics are constructed this way. While there are many different things a CPU can do, the effect of each cycle is expressed as straight-line code: different components compute many possible results in parallel, and multiplexers select among them. Such a style is also important when implementing constant-time cryptography.

[9] This is superficially different from the typical presentation of the STLC, which gives a hierarchical typing derivation rather than a linear machine. In fact, there are mechanical techniques for converting any hierarchical proof system into a linear abstract machine [19, 27]; Sergey [40] gives a full worked-example of doing so for a type system.

Below, we present the full definition of the state machine. This machine evaluates each binding in a single step, although it could be modified to have individual steps checking each subterm, likely following Sergey [40]. Fig. 3 shows the execution of this machine on the program P, containing "modules" $F = \lambda f.\lambda x.f(fx)$ and $G = \lambda x.F(\lambda y.y)x$.

States A state is a triple (Γ, cur, q) of a typing context Γ, a focused binding cur, and a queue q. Γ takes the form $F : \tau_1, G : \tau_2, \ldots$, containing type bindings for all successfully evaluated prior definitions. A *binding* b is either an assignment $F = f$, or the special token `end`. cur is a binding containing the term currently being typechecked. The queue q, written in either the notation $[F = f, G = g, \ldots]$ or $F = f :: G = g :: \ldots$, is a list of bindings to be checked after the current one is completed.

Actions The state $(\Gamma, F = f, b :: q)$ steps to $((\Gamma, F : \tau), b, q)$ with label F if there is some τ such that $\Gamma \vdash f : \tau$ under the normal rules of STLC. It steps to (Γ, b, q) with label F if there is no such τ. Any state with cur $=$ `end` is terminal.

Properties Properties take the form $F \in \Gamma$, where Γ is the typing context of the terminal state. This means that the binding for F typechecked successfully.

Allowed Mutations From the state $(\Gamma, \text{cur}, F = f :: q)$, a mutated action may step to $(\Gamma, F = f, q)$. This is equivalent to forcing the binding $F = f$ to fail to typecheck.

Applying the definition of §6 to program P, we find that the well-formedness of G, $G \in \Gamma$, does depend on the well-formedness of F, as reified in the transition labeled F.

7.2 "Depend on Interfaces, Not Implementations"

A common slogan in software engineering is "depend on interfaces, not implementations." In this section, we formalize this advice inside the simplest setting that allows for reasoning about logical interfaces: Imp with Hoare-style verification. In this setting, the dependee "module" is the first portion of a sequential program, and the "interface" is a postcondition which is weaker than its actual behavior. The lack of dependence on implementation means that no change to the module which preserves its postcondition may alter the overall guarantees of the program.

In this domain, showing that an altered program meets a spec means finding a Hoare logic proof of that spec. More broadly, the "execution" of the Hoare logic semantics is a highly nondeterministic proof search which may involve conjuring new propositions (e.g.: loop invariants) out of thin air. It is overall a poor fit for our formalism in terms of deterministic state machines. While an alternate formalism of causality would be more appropriate for this problem—and we have preliminary work on one, based on derivation trees—we find that, for pedagogical purposes, we can encode

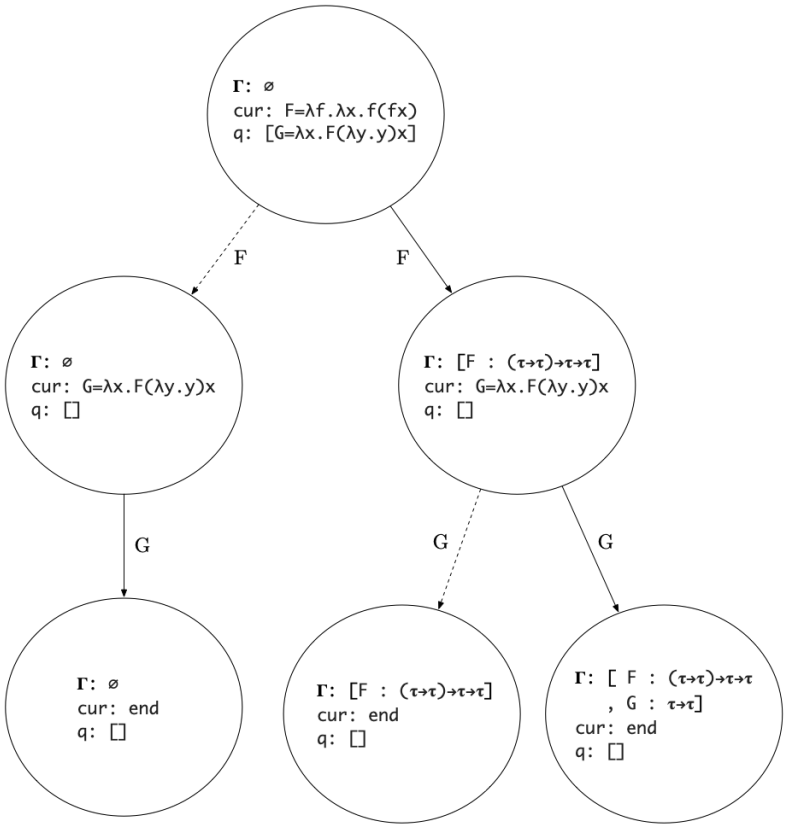

Figure 3. Execution of typing machine on the program P. Dashed edges represent possible interventions.

it quite adequately into a state machine by removing the nondeterminism from the language via these adjustments:

1. All loops are annotated with their loop invariant. They take the form while$_I$ E do S, where I is the loop invariant.
2. There is no arbitrary strengthening or weakening of assertions (i.e.: the CONSEQUENCE rule is removed). All postcondition weakening / preconditioning strengthening is done by the command weaken(Q), where the Hoare triple $\{P\}$ weaken(Q) $\{Q\}$ is derivable for any P with $P \Rightarrow Q$. This command also doubles as a way to give an explicit "interface" for the preceding code.

With these changes, computing the strongest postcondition of a command becomes a deterministic function, and we can now linearize the system into a state machine.

States We first define an *assertion*. At assertion P for a program point pc is a predicate a predicate which must be true of state at pc in all concrete program executions. A state is stack of program counter/assertion pairs $(pc_1, P_1) :: \cdots :: \emptyset$, with P_i an assertion for pc_i for all i. pc_1 is the "current" program point. Each successive pc_{i+1} is the pc prior to executing the parent loop or conditional pc_i, where relevant. At times we will speak loosely and only refer refer to the head of the state, (pc_1, P_1).

Actions Let C be the command at pc_1, and pc_1' its successor command. Then state $(pc_1, P_1) :: S$ transitions to a new state depending on C.

1. C is an assignment: Then the state steps to the target state $(pc_1', \text{sp}(C, P)) :: S$, where $\text{sp}(C, P)$ is the strongest postcondition of P across C.
2. C = weaken(Q). If $P_1 \Rightarrow Q$, then the state steps to $(pc_1', Q) :: S$. Else, the state is terminal. In the latter case, we say that the preceding segment of the program **fails** its immediate postcondition.
3. C = if E then C_1 else C_2. Let pc_0 be the pc for immediately before the execution of C_1. Then the state steps to $(pc_0, P_1 \wedge E) :: (pc_1, P_1) :: S$.
4. The previous case should communicate the use of the stack. The remaining rules are not necessary to communicate the intended point, and we leave them as an exercise to the reader. (This system is a linearization of the normal rules of Hoare logic, similarly to what can be mechanically derived via the transformation of §5.1 of [19].)

Properties The property in question is that a terminal state with a given pc and assertion Q is reached, i.e.: that Q is a valid postcondition for the entire program.

Allowed Mutations Depending on the kind of dependence query considered, this system uses one of two choices for the space of allowed mutation:

1. When checking for dependence on a spec, the following mutations are allowed: Consider some *pc* associated with a weaken command weaken(Q). Any state with this *pc* will have a transition to a new state with head (pc', Q). A valid mutation may instead target (pc', Q') for any proposition Q'. That is, the preceding code may be *ascribed* a new spec. (Note that the preceding code need not actually meet this spec!)
2. When checking for dependence on an implementation, the following mutations are allowed: Consider some *pc* associated with a command C, with outgoing transition targeting some pc'. A valid mutation is a transition targeting a state (pc', P) for an arbitrary assertion P (equivalent to replacing the code at *pc* with any code) **subject to the condition** that the intervened trace does not fail its immediate postcondition. More formally: consider the next state in the original trace whose *pc* is associated with a weaken command weaken(Q); then the intervened trace must reach another state with the same *pc*, and this state may not be terminal (meaning Q is a valid weakening of the postcondition). These mutations are used to check for dependence on an implementation.

We now apply this definition to a simple example:

```
// Module 1
choice := 1;
weaken(choice == 1
       || choice == 2
       || choice == 3);

// Module 2
result := choice * 2;
weaken(result < 10)
```

The only valid mutations to the body of module 1, the `choice := 1` line, are the equivalents of replacing it with code passing the postcondition, e.g.: `choice := 2` or `choice := 3`. Neither of these prevent module 2 from achieving the postcondition result < 10, so module 2 does not depend on the implementation of module 1. However, it does depend on the interface, as replacing module 1's postcondition with `choice == 5` will cause module 2 to violate its postcondition.

7.3 Trusted Bases

The idea of a trusted base is fundamental to high dependability systems. In short, the trusted base is that part of the system that must function correctly; failures in any other part of the system can be tolerated. If the system can be designed so that the trusted base is small, this implies a concomitant reduction in the effort required to guarantee correctness of the system as a whole, since only the trusted base need be checked. Of course the claim that the trusted base is indeed alone sufficient to establish correctness must be justified, and this may require in addition some kind of non-interference argument (to show that failures outside the trusted base cannot compromise it).

To define the concept of trusted base, we need to limit the interventions that can be considered in determining causality. The intuition is very simple: the very idea of a trusted base relies on the assumption that this part of the system will not break. So we consider causality and dependence only in the context in which interventions may not alter the behavior of the trusted base.

More formally, we partition the actions of the system as a whole into the actions of the trusted base B, which we shall refer to as the *trusted actions*, and the actions of the rest of the system, which are *untrusted*. Correctness of the system as a whole is with respect to some critical property P. It is rarely practical for this property to capture full correctness, and for most critical systems it will represent the non-occurrence of some catastrophe (such as loss of data, violation of security, physical accidents such as collision, etc). The base B is then sound for P—that is, it indeed forms a trusted base for that property—if P holds for all traces (that is, P is indeed a property of the system), and no action outside B causes P, in a context in which the only permitted interventions are on actions outside B. That is, so long as the actions in the trusted base execute faithfully, the criticality property depends only on the trusted base, and on no other parts of the system.

To illustrate this, consider the example of the *careful file transfer protocol* from the famous end-to-end paper [39]. In the simple (standard) file transfer protocol, blocks are read from disk, sent across the network, and written to disk on the other side. In the careful file transfer protocol, the sender computes a checksum of the file on disk; the receiver similarly computes a checksum after writing the file to disk on the other side, and sends it back to the sender; and if the two checksums do not match, the transfer is repeated until they do.

In the original paper, the example was used to explain the idea of end-to-end design: by designing the endpoints to perform the checksum computations, the protocol no longer depended on a reliable network, suggesting that the overall goals of a protocol can be established at the endpoints without burdening the network itself with requirements that might not even apply to other protocols. In the context of this paper, the key point is that the checksum mechanism becomes a trusted base for the protocol: so long as the checksums are computed correctly, the file transfer can be assumed to be correct. Below, we give a semi-formal description of

how to formalize this work into a state machine. Fig. 4 gives an example trace.

States The state components are the files on disk, the messages in the network, and various buffers for local storage. For convenience, we represent each category as a single variable, and represent the value of that variable as a relation (or equivalently as a predicate). Thus, for example, the `disk` variable holds the contents of the files on disk for both sender and receiver, and includes in its value the tuple `(S, C)` when, on the sender's disk, file has content *C*); likewise the network `net` contains `(S, C)` when the channel emanating from the sender (*S*) contains the value *C*; and the variable `checksum` contains `(S, i)` when the sender has computed the checksum *i* for the file.

Actions The actions are `disk.read(d,c)`, `disk.sum(d,i)` and `disk.write(d,c)` in which the file contents *c* are read, have checksum *i* computed, and written, at disk *d*; `net.send(p,c` and `net.recv(p,c)` in which the channel emanating from participant *p* has a message with content *c* sent on it or received from it; and an internal action `ftp.match` of the top-level protocol that checks that the checksum received by the sender matches the checksum it previously computed from the file on its own disk.

Property The correctness property is that, for any terminating trace, the data stored for the file in the final state is the same in the two disks (that is, the disk variable contains tuples of the form `(S,c)` and `(R,c)` where *c* is the shared value of the file.

Allowed Mutations Except for the trusted `disk.sum` and `ftp.match` actions, any action may be mutated to give either fail (resulting in a retry) or to give an arbitrary value[10].

The trusted base then comprises (a) the actions of the top-level program (including both the general order of actions and the specific `ftp.match` action), and (b) the `disk.sum` action that computes the checksum of files on disk. With this setup, applying the definition of §6, the correctness property does not depend on any non-trusted action.

[10]There is one additional subtlety that must be addressed, common to any formal analysis of a system of this sort. When considering the interventions that are permitted for the actions outside the trusted base, the formalism must disallow transitions that magically guess the correct value of the checksum. If such an intervention were allowed, the `net.send` action (for example) might cause the correctness property, because one could construct a trace in which an intervention breaks the `disk.write` action, causing the wrong data to be written to disk, and a second intervention breaks the `net.recv` action so that the sender happens to receive the correct checksum back, despite the fact that it does not match the file written to disk. Another solution to this problem would be to represent the checksum computation more abstractly in a way that disallows faking.

8 Puzzling No More!

We are now ready to revisit the examples of §2. With the new perspective of the proper framing and the multiple varieties of dependence, all of the apparent paradoxes disappear. Some of these examples also serve to illustrate some counterintuitive properties of dependence that it inherits from its basis in actual causation.

1. Assuming tasks are dynamically assigned, the round-robin scheduler's well-formedness does not depend on the tasks in static semantics. Specific results obtained and timing do depend in the dynamic semantics. Correctness does not depend in program logic, with a posssible exception for minor conditions on resource use.

2. For a pair of a serializer/deserializer, the round-trip property depends on both in the program logic. Alternatively, if the spec for each is defined with reference to a mathematical description of the format, defined as a relation between a data structure and its serialized form, then the correctness of each depends on said format in the program logic.

3. If it is reasonable for *B* to modify a global variable of *A*, then indeed the correctness of *A* *already* depends on *B* in the program logic not to do so, and any programmer checking the code would need to inspect *B* for whether it interferes with *A*. But if *B*'s super-spec includes a frame condition restricting what state is considered reasonable for it to modify, then such a programmer would not need to inspect *B*, and *A*'s results do not depend in the dynamic semantics on *B*.

4. Similarly, if it is considered valid to replace any module with one that crashes, then *A* not crashing depends in the dynamic semantics on every other module. But if each module has a super-spec prohibiting crashing, then these dependences are removed.

5. By the definition in §6, *A*'s success in the trace does indeed depend on *B*, witnessed by the contingency where *B* fails, and *C* also fails. (If *C* is never executed, then it cannot be a dependence.) Under a definition closer to but-for-causation, *A*'s success would not depend on *B*, as *B* cannot be modified to cause *A* to fail, but it would depend on the set $\{B, C\}$ rendering *B part of a dependence*. Meanwhile, *A*'s correctness does not depend on the correctness of *B* in the program logic according to either definition. This example is isomorphic to examples in causality on over-determined events and having a backup plan, such as Example 1 from Halpern [15], and Example 13 in Glymour [12]. This is the only realistic software-engineering example we have found that involves a contingency / compensating intervention; we discussed specific instantiations in §4.4.

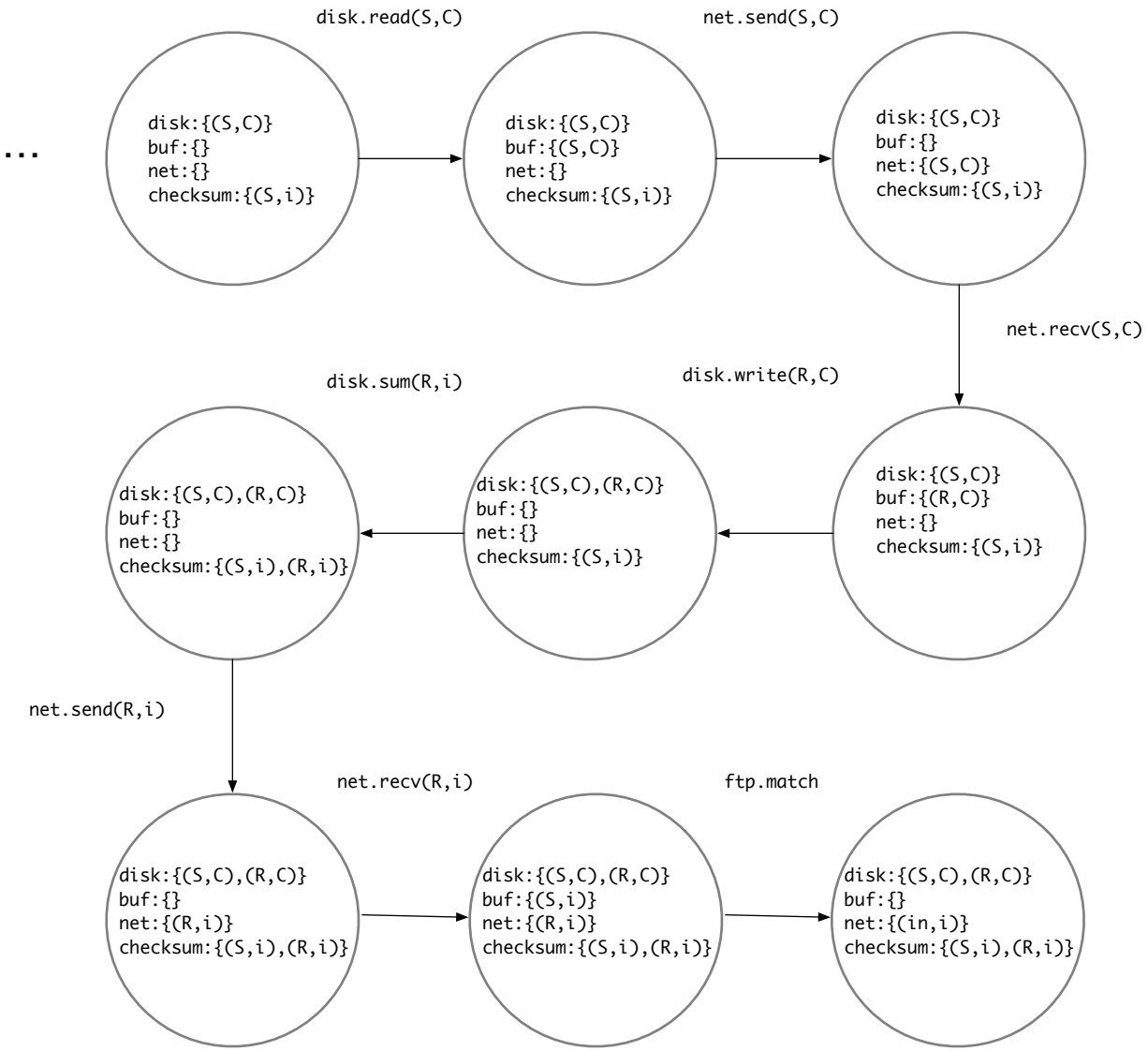

Figure 4. Sample trace of the careful file transfer protocol.

6. If A uses B through dependency inversion, then A's well-formedness does not depend on B in the static semantics, but its results do depend on B in the dynamic semantics, and its correctness depends on the spec of B in the program logic.

7. Even with dependence both ways between properties of A and properties of B, we need not consider self-dependence because, in general, dependence is *not* transitive. A may use B, and B may use C, but A need not exercise all behaviors of B, so it is possible for A's results or correctness to not depend in the dynamic semantics on C. Alternatively, see the structure in Example 5 above. Causation is generally not transitive for the same reasons; Halpern [15] discusses why, along with situations in which it is transitive.

8. Correctness of the hash table would likely be defined conditionally: *if* the inserted keys correctly implement `equals()` and `hashcode()`, *then* the hash table operations produce the correct results. Then a hash table's correctness does not depend in the program logic on the caller or keys. Its results do still depend in the dynamic semantics on both.

9. The robot's moves depend in the dynamic semantics on any subset of 3 programs, but each individual program is merely part of a dependence. This example is lifted directly from questions in causality about majority vote (Glymour [12] Example 14, Halpern [13] Example 3.8). According

to both the Halpern-Pearl definition and but-for causation with minimality, if 5 people vote on a proposal where majority vote wins, and all 5 vote in favor, then any subset of 3 is a cause, while each individual is "part of a cause." This example shows the use of the minimality condition, D3. [11]

As an extra insight, Example 2 shows that **coupling is different from dependence**. While the serializer and deserializer are clearly coupled for any reasonable definition, neither depends on the other: either the correctness of both is defined in terms of some common abstract format, or the correctness is defined in terms of a round-trip property that depends on both.

While we leave thorough investigation of the concept of coupling to future work, one possible working definition is: two code fragments are **coupled** if they ever must change in tandem to maintain some property. This would make the relation between dependence and coupling the same as between causation and correlation. According to Reichenbach's principle [20], if two variables are correlated, then either one causes the other or they have a common cause (or some common effect of both is being held constant). Correspondingly, if two code fragments are coupled, then either one depends on the other, they depend on some common thing (like the abstract description of a file format), or some property (such as the round-trip property) depends on both.

9 Conclusion

Programmers and programming-language researchers sometimes appear to work in parallel universes, each with its own concerns and priorities and seemingly little overlap. Advances that are recognized in both, and which combine the rigor and clarity of research with the subtle insights of practice, have the potential for great impact.

Type theory has achieved broad impact by providing a unified toolset, which has provided a foundation for both an abstract framework for computation and practical tools for structuring programs. In contrast, except for Parnas's pioneering exploration of the "uses" relation [36], and some work unifying several instantiations of noninterference [1], most work that might have contributed to a more general notion of dependence—such as theories of data abstraction (independence of representation) and polymorphism (independence of type)—has proceeded in more specialized contexts, resulting in theoretical ideas with more limited applicability.

A more generalized study of program dependences, we believe, might tie together these and many other existing threads of research (in compilation, program analysis, slicing, and so on), open room to explore it in new application areas, and provide a solid foundation for evaluation of program designs.

We hope that our paper will rekindle interest in the topic of dependences; that the promise that we see in studying these problems will inspire others to engage them; and that our ideas will encourage designers and tool developers to explore new forms of dependence analysis with new applications in many areas.

Acknowledgments

We warmly thank Hana Chockler, Zenna Tavares, Adam Lancaster, Eric Casteleijn, Benjamin Duron, and the anonymous reviewers for comments on earlier drafts of this paper. This research was funded in part by the Secure and Trustworthy Cyberspace (SaTC) program of the CISE division of the National Science Foundation under Award Number 1801399.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 147–160.

[2] Martín Abadi and Luca Cardelli. 1996. *A Theory of Objects*. Springer. https://doi.org/10.1007/978-1-4419-8598-9

[3] Jonathan Aldrich. 2013. The Power of Interoperability: Why Objects are Inevitable. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 101–116. https://doi.org/10.1145/2509578.2514738

[4] Gadi Aleksandrowicz, Hana Chockler, Joseph Y Halpern, and Alexander Ivrii. 2017. The Computational Complexity of Structure-Based Causality. *Journal of Artificial Intelligence Research* 58 (2017), 431–451.

[5] Hana Chockler and Joseph Y Halpern. 2004. Responsibility and Blame: A Structural-Model Approach. *Journal of Artificial Intelligence Research* 22 (2004), 93–115.

[6] William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. 557–572. https://doi.org/10.1145/1640089.1640133

[7] J. Correa and E. Bareinboim. 2020. A Calculus For Stochastic Interventions: Causal Effect Identification and Surrogate Experiments. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*. AAAI Press, New York, NY.

[8] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[9] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[10] Richard Gabriel. 2011. Definitions of Modularity. http://modularity.info/conference/2011/files/PerspectivesOnModularity/ModularityDefinitions.pdf. (2011).

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design.

[11] This is according to the Halpern 2015 definition [13] of actual causation. While we like the intuitive appeal of this answer, the formalism of §6 is expressive enough to permit sets to be causes. Instead, its answer is more similar to the Halpern-Pearl 2005 definition[18], which states: each of the 5 voters is a cause, but with *responsibility* [5, 14] $\frac{1}{3}$ (as 3 votes must be modified to change the outcome). Our preliminary (and more complicated) formalism based on tree-structured derivations does permit causes to be sets.

In *European Conference on Object-Oriented Programming*. Springer, 406–431.

[12] Clark Glymour, David Danks, Bruce Glymour, Frederick Eberhardt, Joseph Ramsey, Richard Scheines, Peter Spirtes, Choh Man Teng, and Jiji Zhang. 2010. Actual Causation: A Stone Soup Essay. *Synthese* 175, 2 (2010), 169–192.

[13] Joseph Halpern. 2015. A Modification of the Halpern-Pearl Definition of Causality. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

[14] Joseph Y Halpern. 2016. *Actual Causality*. MIT Press.

[15] Joseph Y Halpern. 2016. Sufficient Conditions for Causality to be Transitive. *Philosophy of Science* 83, 2 (2016), 213–226.

[16] Joseph Y Halpern and Christopher Hitchcock. 2015. Graded Causation and Defaults. *The British Journal for the Philosophy of Science* 66, 2 (2015), 413–457.

[17] Joseph Y Halpern and Judea Pearl. 2001. Causes and Explanations: A Structural-Model Approach: Part I: Causes. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*. 194–202.

[18] Joseph Y Halpern and Judea Pearl. 2005. Causes and explanations: A structural-model approach. Part I: Causes. *The British journal for the philosophy of science* 56, 4 (2005), 843–887.

[19] John Hannan and Dale Miller. 1992. From Operational Semantics to Abstract Machines. *Mathematical Structures in Computer Science* 2, 4 (1992), 415–459.

[20] Christopher Hitchcock and Miklós Rédei. 2020. Reichenbach's Common Cause Principle. In *The Stanford Encyclopedia of Philosophy* (spring 2020 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[21] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62. https://doi.org/10.1145/2362389.2362393

[22] Mark Hopkins and Judea Pearl. 2003. Clarifying the Usage of Structural Models for Commonsense Causal Reasoning. In *Proceedings of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*. AAAI Press Menlo Park, CA, 83–89.

[23] Thomas F Icard. 2017. From Programs to Causal Models. In *Proceedings of the 21st Amsterdam Colloquium*. 35–44.

[24] Daniel Jackson. 2002. Module Dependences in Software Design. In *International Workshop on Radical Innovations of Software and Systems Engineering in the Future*. Springer, 198–203.

[25] Daniel Jackson and Eunsuk Kang. 2009. Property-Part Diagrams: A Dependence Notation for Software Systems. In *ICSE Workshop: A Tribute to Michael Jackson, Vancouver*. Citeseer.

[26] James Koppel. 2018. You are a Program Synthesizer. http://www.pathsensitive.com/2018/12/my-strange-loop-talk-you-are-program.html. (Dec. 2018).

[27] James Koppel, Jackson Kearl, and Armando Solar-Lezama. 2020. Automatically Deriving Control-Flow Graph Generators from Operational Semantics. (2020). arXiv:cs.PL/2010.04918

[28] James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

[29] Jonathan Laurent, Jean Yang, and Walter Fontana. 2018. Counterfactual Resimulation for Causal Analysis of Rule-Based Models. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. 1882–1890. https://doi.org/10.24963/ijcai.2018/260

[30] Ben Liblit. 2007. *Cooperative Bug Isolation*. Vol. 4440. Springer.

[31] Robert C Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

[32] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Y Halpern, Christoph Koch, Katherine F Moore, and Dan Suciu. 2010. Causality in Databases. *IEEE Data Eng. Bull.* 33, ARTICLE (2010), 59–67.

[33] John Ousterhout. 2018. *A Philosophy of Software Design*. Yaknyam Press.

[34] Pavel Panchekha, Adam T Geller, Michael D Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying that Web Pages have Accessible Layout. *ACM SIGPLAN Notices* 53, 4 (2018), 1–14.

[35] David L Parnas. 1972. On the Criteria to be Used in Decomposing Systems into Modules. In *Pioneers and Their Contributions to Software Engineering*. Springer, 479–498.

[36] D. L. Parnas. 1979. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering* SE-5, 2 (1979), 128–138.

[37] Judea Pearl. 1998. On the Definition of Actual Cause. (1998).

[38] Judea Pearl. 2009. *Causality*. Cambridge University Press.

[39] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (Nov. 1984), 277–288. https://doi.org/10.1145/357401.357402

[40] Ilya Sergey and Dave Clarke. 2011. From Type Checking by Recursive Descent to Type Checking with an Abstract Machine. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. 1–7.

[41] William N Sumner and Xiangyu Zhang. 2013. Identifying Execution Points for Dynamic Analyses. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 81–91.

[42] Zenna Tavares, James Koppel, Xin Zhang, and Armando Solar-Lezama. 2019. A Language for Counterfactual Generative Models. http://www.zenna.org/publications/causal.pdf. (2019).

[43] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[44] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages via Transformational Compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 770–778.

[45] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. *ACM SIGSOFT Software Engineering Notes* 27, 6 (2002), 1–10.