The Ethical Compiler: Addressing the Is-Ought Gap in Compilation (Invited Talk)

William J. Bowman
University of British Columbia
Vancouver, Canada
wjb@williamjbowman.com

Abstract

The is-ought gap is a problem in moral philosophy observing that ethical judgments ("ought") cannot be grounded purely in truth judgments ("is"): that an *ought* cannot be derived from an *is*. This gap renders the following argument invalid: "It *is* true that type safe languages prevent bugs and that bugs cause harm, therefore you *ought* to write in type safe languages". To validate ethical claims, we must bridge the gap between is and ought with some ethical axiom, such as "I believe one *ought* not cause harm".

But what do ethics have to do with manipulating programs? A lot! Ethics are central to correctness! For example, suppose an algorithm infers the type of e is `Bool`, and e is in fact a `Bool`; the program type checks. Is the program correct—does it behave as it *ought*? We cannot answer this without some ethical axioms: what does the programmer believe *ought* to be?

I believe one ought to design and implement languages ethically. We must give the programmer the ability to express their ethics—their values and beliefs about a program—in addition to mere computational content, and build tools that respect the distinction between is and ought. This paper is a guide to ethical language design and implementation possibilities.

CCS Concepts: • **Theory of computation → Program specifications**; *Type structures*; • **Software and its engineering → General programming languages**; **Compilers**; **Formal software verification**; *Software performance*.

Keywords: Types, Compilers, Optimization, Security, Correctness, Ethics

ACM Reference Format:
William J. Bowman. 2025. The Ethical Compiler: Addressing the Is-Ought Gap in Compilation (Invited Talk). In *Proceedings of the 2025 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '25), January 21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3704253.3706135

PEPM '25, January 21, 2025, Denver, CO, USA

ACM ISBN 979-8-4007-1350-7/25/01
https://doi.org/10.1145/3704253.3706135

1 Introduction

Compilers *should* be correct [36]. But what *is* compiler correctness? What *should* compiler correctness mean?

Compiler correctness is one of the oldest questions in programming languages research [25], and what it means is still widely debated today [29]. Consider the following example definition: for all programs e, if e runs to the value v in an interpreter, then e runs to v after compilation. This is a formalization of a commonly used compiler correctness theorem *whole-program correctness*. A compiler may satisfy this specification, but if it does, *should* it be considered correct?

Our thesis is that this question is fundamentally impossible to answer.

"Should"—or *"ought"* to distinguish the technical from the colloquial use—is an ethical judgement. *"Ought"* is a judgement about what is *subjectively* true in the world: what one (a subject) values, believes, intends, considers to be moral; and what follows by logical deduction from those values, beliefs, intentions, and morals. By contrast, *"is"* expresses a truth judgement, a judgement about what is *objectively* true in the world: what is true about objects in the world, and what follows by logical deduction from those truths.

A question about what *ought* to be cannot be reduced to a question about what *is* [19]. This problem is called the *is-ought gap* in moral philosophy, and versions of it show up in many disciplines. There are many good arguments for why whole-program correctness *is* insufficient to capture the correctness of a compiler for realistic software, but that does not tell us whether it *ought* to be the definition of correctness. And, by the is-ought gap, arguments about why one definition of correctness *is* insufficient *cannot* tell us what the definition *ought* to be. We require some ethical axioms to bridge the gap. For example, if the developer only values monolithic whole programs without modularity, does not value separate compilation, and does not value security, then perhaps the compiler *ought* to be correct if it is whole-program correct. On the other hand, if the developer is writing cryptographically secure code, and believes the program *ought* to satisfy constant timeness, then the compiler

ought not be correct unless it preserves constant-timeness. Such ethical axioms must come from the programmer.

Much work in compilation ignores the is-ought gap, and we suffer as a result. In the best case, we argue about the merits of different compiler correctness theorems, from whole-program correctness to fully abstract compilation, which preserves and reflects all possible observational equivalences through compilation [29]. Frequently, we observe that "correctness" and "security" are different kinds of specifications [10, 13, 21]. In the worst case, we blame the programmer, telling them to just memorize what the compiler does [4].

This is not only about the compiler per se, but many aspects of programming language design and implementation, and how they interact with a programmer. Consider a type system, a common place for programmers to express specifications and beliefs about their program. What *ought* an expression's type be? When *ought* a program be considered well typed? Or consider an intermediate language, designed to guarantee safety but also admit optimizations. What programs *ought* to be equal? What dynamic checks *ought* to occur, and when? These question can only be answered by the programmer. Unfortunately, it is often *impossible* for the programmer to express their beliefs. Worse yet, much work is spent trying to *infer their beliefs* from what *is true* in the program.

Our original question, "what *ought* compiler correctness be?", is similarly flawed. It can only reduce correctness of the compiler to *truths* about the original program, but that cannot tell us what the compiled output *ought* to do. "Ought" cannot follow from "is"! There is no answer to what compiler correctness *ought* to mean independent of the program being compiled, and what the programmer believes and values in that program.

To address this gap, we must zoom out; cease the narrow focus on what is technically correct, stop thinking in purely pragmatic terms, and think instead about what *ought to be*: about ethics.

Dictum 1. *Compilers should be* ethical*, rather than* correct.

I believe compilers should, *ought*, be *ethical*, rather than *correct*. I present this and other dictums as meta-ethical axioms about language design and implementation. The ethical compiler preserves *intent* rather than *truth*[1]: what *ought* to be true of a program *ought* to be true of the compiled program. Designing and implementing an ethical compiler requires careful attention to distinctions often ignored in merely correct compilers. To have an ethical compiler, developers must be able to express their intent, and language implementations must be able to use and distinguish intent from the truths of the computation expressed in the program. So how do we design and implement ethical compilers?

[1]We use "intent" as the noun for an ethical judgement, and "truth" as the noun for a truth judgement.

2 Expressing Truth and Intent

We have two judgements: the subjective "ought" judgement, and the objective "is" judgement. In the context of ethical compilation, the object is a program and the subject is the programmer. The program is an expression of some computation, and certain truths hold of it. The programmer also has intents for that program, which may or may not be expressed.

It is common to focus on understanding truths about computations. Introductions to many languages and language implementations focus on how expressions evaluate, what optimizations will apply and when, or when some property will be *true* about a given program.

A key problem is in ethical compilation is expressing *intent*—what ought to be true. Many languages and tools restrict our ability to express intent. Such restrictions are often in the name of some guarantee or trade-off, such as safety, security, performance, or usability. Some such restrictions are mere accidents or historical artifacts; as a system or context changes, so too might our beliefs, values, and intents. There are practical and philosophical problems with these restrictions, though.

Philosophically, these restrictions enforce moral universalism: that a single system of ethics applies universally to everyone. They force the programmer to subscribe to the ethical framework of the compiler writer. Many programmers realize this and rebel, writing philosophical treatises on the morality and politics of their compilers:

> "Undefined behavior consists of exactly one proposition, to wit: There must be compiler developers whom the language standard protects but does not bind, alongside developers whom the language standard binds but does not protect." [16]

> "There is no ethical compilation under late capitalism." [17]

> "People need to drop the 'I know [what] the compiler does'-model and start using the 'The compiler is an evil djinn, secretly trying to corrupt your wishes with the moral compass of tobacco industry lawyers'-model of C semantics." [4]

If we subscribe to moral relativism, the meta-ethical belief that there is no one true system of ethics, then we must give the programmer the ability to express intent about their computation. Moreover, we must constantly improve our languages' systems of ethical expression to express new intents as they are discovered.

Dictum 2. *The programmer should be able to say whether something that* is ought *to be.*

Practically, any compiler that restricts the programmer's ability to express intent will create programs that do not behave as they ought, *i.e.*, unethical programs.

One of my favourite studies of unethical programs is the work of D'Silva et al. [13]. D'Silva et al. [13] name and discuss the so called "correctness-security gap", studying how compiler optimizations, even when "correct", can violate security properties. A simple example is a program that explicitly overwrites memory containing a password, but this overwrite is removed by dead store elimination. Overwriting *ought* to be preserved for security, but the programmer cannot express this. Another example is a cryptographic routine designed to be constant time, but common sub-expression elimination changes the timing of one branch. The timing behaviour *ought* to be preserved, but the programmer cannot express this. Both are ethical problems with the compiler; the resulting programs are unethical.

D'Silva et al. [13] propose adding keywords to C that would enable the programmer to express their intent. For example, the keyword `secure` would express that writes to memory that are never read are observable and should be preserved, and `lockstep` would indicate that timing is critical.

This is a great proposal, but it's incomplete. Bizarrely, despite being unethical, both optimizations are completely, formally, correct. So how does one reconcile the apparently correct behaviour of the programs with their obvious immorality?

The authors observe that the correctness-security gap arises from the C semantics—that is, from *truths* about C. For example, it states "GCC 3.2 only attempts to preserve the semantics of C ... [but] time and power consumption are often not specified by the language standard". They also observe this is a common problem in formal approaches to compiler correctness, stating "semantics accounts for the state of a program but not the state of the underlying machine". In C's semantics, *it is true that* any two writes to a memory cell *are* equivalent if that memory cell is never read. Unfortunately, the programmer believes it *ought not* be true—C's semantics is unethical.

It's not enough to express intent if what *ought* to be true can never be true.

Dictum 3. *Anything that* ought to be *should be possible.*

So the authors propose a second piece to the solution: ensure that the semantics of programs more closely model that state of the underlying machine. They work through an example of such a semantics and use it to show that dead store elimination does not preserve semantics.

Neither solution is enough on its own. Changing the semantics solves the ethical problem with the semantics—the semantics is now ethical, since it is now possible for what *ought* to be true to be in fact true. But we're left with the is-ought gap: given an arbitrary piece of C code with a dead store, ought the dead store be eliminated? The compiler *cannot* automatically decide this based on only the computation content of a C program. It requires an ethical axiom from the programmer: that this memory cell really *ought* to be overwritten, so the apparently dead store *ought not* be eliminated. The keywords are introduced for this purpose, to communicate this intent to the compiler, bridging the is-ought gap.

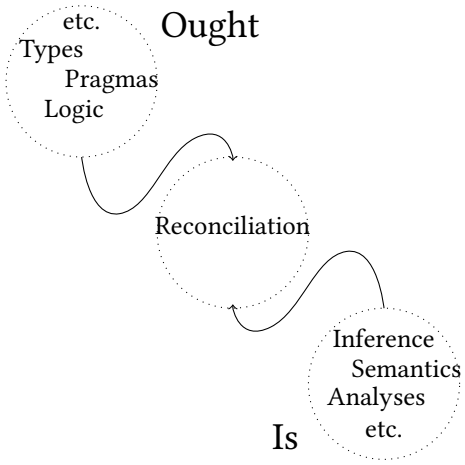

Figure 1. The Ethical Compilation Design Space

We can visualize the space of ethical compilation research as in Figure 1. The first fix, adding keywords, exists in the "Ought" space. The second fix, modifying the semantics, exists in the "Is" space. Both are necessary.

This is a great example, but adding keywords every time we need to express new intents is a little unsystematic for my taste.

My favourite approach to expressing intent is through type systems. Being positioned inside the language alongside the computation they describe, types often provide a local, compositional, lightweight place to hang expressions of intent. They can be extensible, enabling programmers and compiler developers to add new types to capture new intents.

However, simple type systems are often too limited to express intent in many domains. For example, while C's type system can express intent about how much memory a value will occupy, it cannot express the type of an array whose bounds ought to be statically known and, therefore, safe to access without dynamic bounds checks. Further, type systems often come with a guarantee—type safety—and as mentioned earlier, guarantees tend to restrict expressivity. For example, WebAssembly (Wasm) has a simple type system, but requires several dynamic checks to guarantee type and memory safety [18]. There is no way for the programmer to express that dynamic checks ought not be necessary.

Much work on type systems is about enabling more expressions of intent and addressing the is-ought gap. Consider the following example from my own work.

In Geller et al. [15], we study the aforementioned problem with Wasm: there is no way to express that a computation *ought* not require dynamic checks. In that work, we argue

from pragmatics: while Wasm is designed to allow implementations to mitigate these costs, in practice, they still add overhead [20], and in some contexts, those mitigations strategies cannot work. However, really, the work is fundamentally about ethical compilation: I *ought* to be able to express that dynamic checks are not necessary, particularly in a low-level language. This is impossible as Wasm violates Dictum 2 and Dictum 3—I cannot express that dynamic checks ought not be necessary, and the semantics requires dynamic checks to ensure safety.

Our work introduces Wasm-Prechk, which addresses both dictums. Past work designs type systems for low-level languages that enable expressing static bounds that can be used to eliminate array bound checks [35, 37]. We design such a type system for Wasm, extend it to all the dynamic checks in Wasm, and prove type safety. This work firmly falls into the "Ought" space of Figure 1, addressing Dictum 2: Wasm-Prechk enables the programmer to express dynamic checks ought not be necessary to ensure safety.

However, as we saw in our earlier example, it is not enough to express what ought to be if what ought to be is not possible. In Wasm, there is no way for an instruction to access memory without a dynamic bounds check; the semantics say no such thing is possible. So Wasm-Prechk introduces new instructions, a "pre-checked" counterpart to each instruction that requires a dynamic check. The pre-checked instructions require a stronger static check, to guarantee safety, but their semantics do not perform a dynamic check. This work falls into the "Is" space of Figure 1, addressing Dictum 3: Wasm-Prechk makes it possible for instructions that, in truth, do not perform dynamic checks.

Both pieces of work are necessary for an ethical compiler. With only the modified semantics, it's possible for a compiler to remove the dynamic checks. In fact, some Wasm implementations probably do this internally, using some static analysis to determine that a dynamic check *is* unnecessary, and never exposing those semantics to the programmer for safety. But that is not enough to be ethical: the programmer cannot express that the dynamic check *ought* to be unnecessary. Wasm-Prechk enables the programmer to express *ought*, and the language semantics to express *is*.

3 Preserving Intent

An ethical compiler requires additional expressivity for both truth and intent about computations, but it requires more than that. We've seen that a compiler may be technically correct but unethical, as it preserves the unintended truth of a computation. In our earlier examples, this resulted from unexpressed intent. When the intent is expressed, a merely correct compiler may still be unethical if it ignores what ought to be, and generates some code that does something else. To be ethical, the compiler must preserve intent.

Dictum 4. *What* ought *to have been, still* ought *to be.*

As types are my favourite way to express intent, *type preservation* is my favourite way to formalize ethical compilation.

Type preservation originated in some pragmatic concerns. Proof-carrying code was an exciting idea to pair a distributed program component with a specification and a certification of correctness [28]. This would completely eliminate trust in the program and the proof, reducing trust only to the specification (intent) and the proof checker—great for *security*! Unfortunately, certificates could be quite large. Type preservation presented a possible solution. By using typed intermediate languages, syntax could be reused as part of the proof of correctness for the specification in the type, a la the Curry-Howard correspondence. This could, in principle, reduce the size of certificates, and even give a clear way to generate and preserve some proofs through compilation [27].

That is still essentially the motivation found in the literature, but I prefer the philosophical argument: when types express intent, type preservation expresses ethical compilation. I don't need to argue that this is good; it is ethical by definition.

By systematically expressing intent in types, type preservation gives a systematic way of developing an ethical compiler. For each program transformation, an intermediate language must be designed as in Section 2 in which the required intent is expressible but for a new, transformed program. This could be a single typed intermediate language admitting various typed equivalences, or a series of typed intermediate languages each with different semantics but type systems capable of expressing the original intent. Then an ethical program transformation transforms types and syntax together, building a new well typed term.

Consider one recent and relatively simple example from my own work.

Some languages, such as dependently typed languages, or the simply typed λ-calculus (STLC) without other features, are terminating—all functions are intended to terminate. However, if we compile the language, performing closure conversion to compile first-order functions into second-class closed procedures and explicitly allocating data in memory (as any compiler targeting a realistic machine will do), the language becomes non-terminating. This is true even if the compiler is proven correct with respect to whole-program or separate compilation. The new truth can be observed when linking against handwritten target language components, which could cause unexpected non-termination, or when trying to statically analyze target language code, which can no longer be guaranteed to terminate. Suddenly, with access to explicit memory, functions can express recursion through the heap—we get unintended non-termination. How can we preserve the intended termination behaviour of all functions in the language?

We show that a simple type system that stratifies the kind of each heap allocated data type can preserve the intended

termination behaviour [22, 23]. This work uses stratified kinds based on predicative universe hierarchies from type theory [24], but adapts them to reason about regions of the heap that a heap-allocated type quantifies over, rather than the universe of propositions a proposition quantifies over. The idea is that in a pure functional language, when a function is created (allocated), it necessarily can only refer to data allocated in previous regions. By capturing this stratification of the heap after compilation, we make explicit in the type that a heap allocated structure (like a reference or a closure) can only refer to previous regions, as was the case in the source language. We can express the intent that there are no cycles in the heap that give rise to non-termination. Preserving typing into this language ensure the intended pattern of allocation and reference is preserved, and the intended termination behaviour of the language is preserved.

By first designing a typed intermediate language to express the desired intent (in this case, acyclicity in the heap), then proving type preservation, we have a recipe to design a compiler transformation that preserves that intent, *i.e.*, an ethical compiler.

While systematic, it's often not easy to preserve types. To express more intents requires richer type systems, and preserving that intent requires preserving types into low-level languages. Unfortunately, these requirements cause two major problems in type preservation. First, as type systems become more complex, they rely more and more on the syntax of programs, making the type system more sensitive to program transformations. Second, as programs become more low level, type systems can rely less on structure and expressions and instead must track state changes through low-level instructions.

While these are difficult, I find these problems easier to address when viewed through the lens of ethics, which helps me dispel invalid latent assumptions. Consider the following example difficulty from the literature on type preservation.

Dependent types are incredibly expressive, and very attractive for ethical compilation. With a type system sufficient to be a foundation of mathematics, surely I could express anything that ought to be true about my program. Then, by preserving that intent, I would have the *most ethical compiler*.

Unfortunately, this appeared to be impossible. A standard model compiler starts with a CPS transformation, making explicit the control flow of the program in the syntax [27]. Barthe and Uustalu [5] demonstrated that a CPS transformation of dependent types *could not* be type preserving to a sound dependent type system. After CPS translation, with continuations made explicit, a continuation could be invoked twice with different values, causing an inconsistency in the presence of dependent types. If the very first transformation could not preserve types, surely there was no hope in general.

This is related to the first problem with type preservation. The complexity of a dependent type system makes it sensitive to the interpretation of syntax. Dependent type systems typically interpret each expression that appears in types as a value to ensure decidability of type checking, interpret the equality type, *etc.* After their CPS translation, expressions encode control effects, and interpreting them as a single value wasn't possible.

Thankfully, that's *not* what their work shows. The real problem was the type system did not capture how continuations ought to behave.

Barthe and Uustalu [5] use a standard CPS translation, and a type translation corresponding to double negation. A term e of type A is CPS'd to a term of type $(A \to \bot) \to \bot$. This is completely standard, well understood, and ethically wrong. The CPS transformation in a compiler does not introduce arbitrary continuations that can be assigned the function type $(A \to \bot)$, invoked arbitrarily. These continuations are *intended* to be called exactly once, at the end of a computation, to jump to the next computation. The type $(A \to \bot)$ does not express that intent. So while this transformation is type preserving [27], it's not ethical.

To fix this, we need a type that guarantees a continuation is called exactly once at the end of its computation. In Bowman et al. [6], we design a dependently typed CPS'd intermediate language following the dictums from Section 2. We use a locally polymorphic answer type, assigning a CPS'd expression a type $\forall\alpha.(A \to \alpha) \to \alpha$. By parametricity, to produce a value of type α, any expression of this type must (extensionally) call its continuation exactly once as the last thing it does.

This isn't quite enough to prove type preservation; we also run into the second problem. Dependent types rely on the structure of programs to pass expressions into the type system. An application of a dependent function $e_1\ e_2$ produces a result type $B[x := e_2]$, where the result type B can depend on the argument e_2. As the control and data flow of the program has been turned inside out, now passed by invoking continuations, that expression e_2 is replaced by a continuation's parameter y. This can interfere with type checking; $B[x := e_2]$ is not the same as $B[x := y]$ when y is some arbitrary parameter rather than a specific expression.

The solution is to reflect in the typing rules the structure of the computation that must happen when executing on the machine, similar to the suggestion of D'Silva et al. [13] but at the level of *ought* rather than *is*. When a continuation is jumped to, its parameter *ought* only take on one value, unlike the parameter of a function. In our language, jumping to a continuation is a syntactic form separate from function calls, with a separate typing rule. It records an equation guaranteeing that the continuation parameter has a statically known value, similar to $y = e_2$, reestablishing the original intent. This also requires a representation of CPS'd computations that can be "cast" to their underlying value, which our locally polymorphic answer type allows by calling the computation

with the identity function. We then prove type preservation, ensuring ethical compilation, since continuations are restricted to their intended behaviour... in addition to some other less interesting compiler correctness theorems.

Our work is not unique in this observation that continuations must be restricted to rule out control effects [3, 33, 34], or that such restrictions and interpretations of effectful computations is necessary to combine effects in dependent type theory [1, 2, 7, 30, 31]. What is unique is the ethical approach we take to expressing and preserving what ought to be true.

4 Reconciliation

Truth and intent are not completely independent. Once we have ethical axioms to bridge the gap, we can check that what ought to be logically follows from what is true and those axioms. If we're not careful in how we reconcile is and ought, though, we can fall into the is-ought gap.

This is particularly true when we're trying to be *pragmatic*. When the programmer has written some computation, something is true of that computation. Asking them to separately write what ought to be true can feel redundant, and it is tempting to not bother. If I implement a function, why do I also need to write its type if the type is "obvious". Surely, *pragmatically*, we don't need to write it down if it's *true*.

This line of thinking is flawed. You might safely get away with it if the goal of a type system is to rule out some bugs [26], but not if your goal is ethical compilation. When types express intent, and there is no conflict in expressing what ought to be true separately from what is true. In fact, it is *vital*, since ought cannot follow from is. The values and beliefs of the programmer do not follow from the objective reality of the program.

We must therefore be careful in how we use inference in an ethical compiler.

Dictum 5. *One should never infer ought from is.*

Inference is an unfortunately common and tempting approach to many practical problems with expressing intent, but doomed to failure. Type inference is a common version of this, aiming to reduce the annotation burden on programmers by inferring what types are true in a program. However, relying on inferred types will only ever tell us whether a program is well typed in the sense of being consistent with itself, not whether it has the types it ought. Inference is also suggested as a solution to other is-ought problems. For example, D'Silva et al. [13] suggest inference for security and timing sensitive regions of code as a solution to security in the face of optimizations. Ethically, such a project is doomed: even if it could be done computationally (which would be surprising, given Rice's theorem), it could never tell us whether the code ought to be secure or sensitive to timing.

This does not mean inference has no place in ethical compilation. Consider the following two examples of ethical inference.

In our work Wasm-Prechk [15], we discuss inference. The annotations of the type system can be quite large and require a lot of developer effort. Standard static analysis (inference) techniques should be possible in many cases, including our benchmarks, and could probably be used to optimize our benchmarks as well as Wasm-Prechk.

However, we did not merely implement inference; that would not be ethical. Inference could never solve the problem we were trying to address: the ability for the programmer to express that dynamic checks ought not be necessary. Inference can only answer a different question: can we conclude, based on what is true, whether the dynamic checks are in fact necessary. To be ethical, we first need the ability express intent in "Ought" space, and then use inference only in "Is" space (Figure 1). Used this way, inference can help us be ethical. If inference tells us something is not true, but we have expressed that it ought to be true, then trying to reconcile the two helps us identify that our computation is not in keeping with our values.

In Chan et al. [8] we do implement inference, but ethically. This work designs an extension to Rocq with sized typing. Rocq relies on termination checking of recursive functions to decide type equivalence, and therefore to decide well typedness. Sized types expresses termination by annotating types with a representation of their size, and ensuring that the size decreases on recursive calls. The approach is more modular and expressive than the syntactic termination criteria used in Rocq. Sized typing is a nice solution to expressing intent, and by reducing termination arguments to types, yields a framework for preserving that intent. But there exists a lot of Rocq code written without sized typing. Past work suggested that complete inference was possible, which would enable adding sized typing in a completely backwards compatible way [32]. In this work, we show that is more or less true, although it turns out not to be practical.

Our approach to inference is careful though, and we resort to inference only to answer a question about *truth*. Inference tells us whether there exist some sized typing argument (within our size algebra) that proves the function terminates. This proof is arbitrary; it's not the reason the function ought to terminate, which only the programmer can express. To be ethical, inference would only be used to aid transition from the current termination checker to a new sized type system, with programmer involvement. Our approach makes inference secondary to an explicit system. We first design and study a sized type system where sized typing annotations are expressed; that is, where a programmer could express that a function *ought* to terminate through a sized typing argument. Then we use inference only to ask whether *it is true that* every Rocq program can be elaborated into our new, explicit, sized type system.

Ethics aside, the is-ought gap is also the source of pragmatic problems with inference. It is usually too expensive to

just consider all possible facts, when you could simply ask the programmer what they intend.

In the case of our sized types, while the inference algorithm can decide whether functions terminate, it's too slow to be practical without input from the programmer. Inference must churn through all the truths about the program, because it cannot know which functions ought to terminate by a sized typing argument, and which ought not be considered; or which parameters to a function ought to be relevant, and which ought to be ignored. We discuss that the only solution to this is giving the programmer the ability to express their intent: when do they want to use sized typing, which functions ought to be checked, which variables ought to be relevant, and which variable's size ought to decrease.

Another example of pragmatic problems with inference appears in secure compilation. Deng and Namjoshi [10] show that inferring whether dead store elimination causes a security problem is PSPACE-hard for finite-state programs, and undecidable in general. This shouldn't be a surprise with our ethics lens on: of course we cannot decide whether the programmer cares about a leak or not. So inference must answer a harder question: *is* there any leak anywhere that could, assuming the programmer cares, cause a problem in any other part of the code?

Treating inference as only about truth, and as secondary to expressing intent, is the only ethical way to use inference, and avoids computational feasibility issues.

When we have both truth and intent, we can ethically reconcile the two. Reconciliation is the third space in Figure 1. This is all the work that takes what is, and what ought to be, and does something with that information. Type checking, verification, testing, *etc.*, are all reconciliation. In the case of type inference, for example, we might infer that a term is of type `Bool`, and reconcile this against a claim that it ought to be a `Bool`. If it is, then the term is correctly well typed.

One of my favourite instances of ethical reconciliation is the system of blame as implemented in Racket's higher-order contract system [14]. Contracts are widely used in Racket to provide complex dynamic assertions, expressing intent. They're also used in gradual typing, to guard the interface between a statically typed component and a dynamically typed component. Tracking exactly where an error came from, and which component is to blame, is a decades long area of research [11, 12, 14]. The way Racket presents blame errors is delightfully explicit about truth, intent, and reconciliation. Each error message expresses three things: what was true (is), what was expected (ought), and who is to blame for this violation, *assuming the contract is correct*. This assumption that the contract is correct is an explicit part of the error message. Since the contract is written down by the programmer, it represents the programmer's expressed intent, and the compiler must trust it. But it's also a piece of code, and could contain a bug. The way blame is assigned and contract violations are reported makes all of this incredibly explicit, and ethical.

This is a great example particularly because of how difficult to understand error messages can often be, and I attribute much of that difficulty to the is-ought gap. For example, Crichton et al. [9] study the difficulty with understanding ownership errors in Rust.

Rust errors are great at reporting what is and is not true—inconsistencies—but they fail to express how this relates to what ought to be true, and Crichton et al. [9] note that Rust learners struggle with this. Through the lens of ethics, we can see where the problem arises. The Rust borrow checker infers, from ownership information, truths related to permissions on pointers. Implicit in Rust are some beliefs about what ought to be true about these permissions to ensure safety, but the programmer cannot directly express what permissions ought to be, and error messages do not mention these permissions. As a result, a programmer can be left in the dark about an ownership error, wondering why some assortment of inconsistent truths have anything to do with what ought to be true of their program. Crichton et al. [9] manage to improve learning outcomes by making explicit a permissions model of the borrow checker, and expressing what Rust believes permissions *ought* to be to the programmer as part of debugging. Their model and debugging tools enable reconciling inference against intent, rather than simply raising an inconsistency error resulting from inference.

Lacking this ability to reconcile is against ought, or relying on inference, is a recipe for practical and ethical problems.

Dictum 6. *One should reconcile is against ought.*

5 Conclusion

Programming language design and implementation focuses a lot on correctness, but it should not ignore ethics. Many practical and philosophical problems arise when we ignore or cannot express what ought to be true, or conflate what is true with what ought to be true.

We should focus on ethical compilation, rather than correct compilation. We should not resort to claims that something is technically correct when the programmer believes it ought not be correct. We should give the programmer the ability to express what ought to be true, and continually improve their ability to express intent. We should ensure that the programmer's intent is possible in their computations. We should preserve the programmer's intent. We should never attempt to infer what the programmer intended, and always reconcile truth and intent.

We should do all of this not because it is useful, or pragmatic, or secure, or correct (though it is all of those), but for no other reason than it is morally right.

Acknowledgments

I'm particularly grateful to Conor McBride, Jacques Carette, and others for many interesting discussions on intent in programming languages, and Kathi Fisler for some interesting discussion of ethics in computer science and programming languages. I'm also very grateful for the PEPM PC and chairs for their feedback, and the opportunity and forcing function to write these thoughts down.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04207. Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-2019-04207.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. NN66001-22-C-4027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NIWC Pacific.

References

[1] Danel Ahman. 2017. *Fibred Computational Effects*. Ph. D. Dissertation. University of Edinburgh. http://arxiv.org/abs/1710.02594

[2] Danel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, Vol. 9634. https://doi.org/10.1007/978-3-662-49630-5_3

[3] Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2034773.2034830

[4] Cornelius Aschermann. 2024. Tweet. https://x.com/is_eqv/status/1767939306520543697 Accessed Nov. 8, 2024.

[5] Gilles Barthe and Tarmo Uustalu. 2002. CPS Translating Inductive and Coinductive Types. In *Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*. https://doi.org/10.1145/509799.503043

[6] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-preserving CPS Translation of Σ and Π Types Is Not Not Possible. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, POPL (Jan. 2018). https://doi.org/10.1145/3158110

[7] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2535838.2535883

[8] Jonathan Chan, Yufeng Li, and William J. Bowman. 2023. Is Sized Typing for Coq Practical? *Journal of Functional Programming (JFP)* 33 (2023). https://doi.org/10.1017/s0956796822000120

[9] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, OOPSLA2 (2023), 1224–1252. https://doi.org/10.1145/3622841

[10] Chaoqiang Deng and Kedar S. Namjoshi. 2016. Securing a Compiler Transformation. In *International Static Analysis Symposium*. https://doi.org/10.1007/978-3-662-53413-7_9

[11] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct blame for contracts: no more scapegoating. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1926385.1926410

[12] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-28869-2_11

[13] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *IEEE Symposium on Security and Privacy Workshops, SPW*. 73–87. https://doi.org/10.1109/SPW.2015.33

[14] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*. ACM. https://doi.org/10.1145/581478.581484

[15] Adam T. Geller, Justin Frank, and William J. Bowman. 2024. Indexed Types for a Statically Safe WebAssembly. *Proceedings of the ACM on Programming Languages (PACMPL)* 8, POPL (Jan. 2024). https://doi.org/10.1145/3632922

[16] Joe Groff. 2024. Toot. https://f.duriansoftware.com/@joe/113364795231040676 Accessed Nov. 15, 2024.

[17] Joe Groff. 2024. Toot. https://f.duriansoftware.com/@joe/113365854022356161 Accessed Nov. 15, 2024.

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062363

[19] David Hume and Michael P. Levine. 1739. *A Treatise of Human Nature: On Understandings*. Vol. 3. Sterling Publishing, 624. Part 1, Section 1.

[20] Abhinav Jangda, Bobby Powers, Emery Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. (2019). https://doi.org/10.48550/arXiv.1901.09056

[21] Andrew Kennedy. 2006. Securing the .NET Programming Model. *Theoretical Computer Science* 364, 3 (Nov. 2006). https://doi.org/10.1016/j.tcs.2006.08.014

[22] Paulette Koronkevich and William J. Bowman. 2023. One Weird Trick to Untie Landin's Knot. In *Workshop on Higher-Order Programming with Effects (HOPE)*. https://www.williamjbowman.com/#hope2023-landins-knot

[23] Paulette Koronkevich and William J. Bowman. 2024. Type Universes as Allocation Effects. *CoRR* abs/2407.06473 (2024). https://doi.org/10.48550/ARXIV.2407.06473

[24] Per Martin-Löf. 1975. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics* 80 (1975), 73–118. https://doi.org/10.1016/s0049-237x(08)71945-1

[25] John McCarthy. 1961. A Basis for a Mathematical Theory of Computation, Preliminary Report. In *Western joint IRE-AIEE-ACM computer conference (Western) (IRE-AIEE-ACM '61 (Western))*. ACM Press, 225–238. https://doi.org/10.1145/1460690.1460715

[26] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[27] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (May 1999). https://doi.org/10.1145/319301.319345

[28] George C. Necula. 1997. Proof-Carrying Code. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/263699.263712

[29] Daniel Patterson and Amal Ahmed. 2019. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 85:1–85:29. https://doi.org/10.1145/3341689

[30] Pierre-Marie Pédrot. 2017. A Parametric CPS to Sprinkle CIC with Classical Reasoning. In *Workshop on Syntax and Semantics of Low-Level Languages*. https://web.archive.org/web/20220122222238/https://www.cs.bham.ac.uk/~zeilbern/lola2017/abstracts/LOLA_2017_paper_5.pdf

[31] Pierre-Marie Pédrot and Nicolas Tabareau. 2017. An Effectful Way to Eliminate Addiction to Dependence. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/lics.2017.8005113

[32] Jorge Sacchini. 2011. *On type-based termination and dependent pattern-matching in the calculus of inductive constructions*. Ph. D. Dissertation. École Nationale Supérieure des Mines de Paris. https://pastel.archives-ouvertes.fr/pastel-00622429/document

[33] Hayo Thielecke. 2003. From Control Effects to Typed Continuation Passing. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/640128.604144

[34] Hayo Thielecke. 2004. Answer Type Polymorphism in Call-by-name Continuation Passing. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-540-24725-8_20

[35] Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/507635.507657

[36] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1993498.1993532

[37] Christoph Zenger. 1997. Indexed Types. *Theor. Comput. Sci.* 187, 1-2 (1997), 147–165. https://doi.org/10.1016/S0304-3975(97)00062-5