DOI:10.1145/3708981

The rational-programmer method is the first reasonably general approach for assessing whether linguistic features and tools can deliver helpful information with software development tasks.

BY CHRISTOS DIMOULAS AND MATTHIAS FELLEISEN

The Rational Programmer: Investigating Programming Language Pragmatics

EVERY SO OFTEN, someone creates or changes a programming language. In the process, these language creators make a number of design choices. They may wonder whether to check certain conditions at compile time or at run-time, they may choose between a simple error system or a provenance-tracking value system for sophisticated error reporting, or they may consider an alternative set of integrated development environment (IDE) tools. Their choices directly affect the software developers who will end up using these languages and tools; therefore, creators try to make these choices with developers in mind.

The Pragmatics of Programming Languages

To make this concrete, consider the design of TypeScript,[25] a typed sibling of JavaScript.[6] Its design explicitly aims to foster interactions between the type-checked code written in TypeScript and the untyped code in JavaScript. Hence, its designers had to make a choice concerning the integrity of type annotations; for example, whether a callback from JavaScript may apply a function of argument type `number` to a string. While the answer of TypeScript's creators is "yes," academic researchers who work on similar programming languages tend to loudly assert, "No, run-time checks should prevent such misapplications."

Making such choices should rest on solid information from a systematic evaluation. In turn, this calls for an evaluation method that can gather information about the use of a particular linguistic feature or tool in a specific situation. One concrete question could be whether run-time checks for a TypeScript-like language would help developers with locating the logical source of misapplications.

Designers should address such questions to programming language researchers, but those just study the theory and practice of languages. Concretely, researchers have studied the *semantics*[35,30] of *mixed-typed* languages and their *performance*.[34] The former shows that run-time checks are needed to establish a generalized form of type safety;[39] the latter says that run-time checks are often expensive. Neither investigation answers the question of whether and how information from failing run-time checks helps developers locate such misapplications. What the area lacks is a method for assessing the pragmatics of language features.

Linguists describe *pragmatics* as

IMAGE BY CYBERMAGICIAN

the study of natural-language use in context. By interpreting "context" as "work situation," the definition directly applies to the study of programming-language use. The above question is a concrete instance: Types are the novel feature of TypeScript, and finding the source of a mismatch between a program's checked type annotations and its run-time behavior is the work situation. An evaluation method should determine whether run-time checks provide information that assists with locating the source of the mismatch.

Over the past decade, the authors have developed such a method, dubbed the *rational programmer*. Their first, specific goal was to investigate whether run-time checks provide helpful information, because of their own involvement in a TypeScript-like language. To their surprise, the results of their rational-programmer experiments were highly nuanced: When a correct type annotation describes buggy untyped code, the information produced by the run-time checks is *not* all that helpful with finding the source of mismatches; when the problem is due to mistaken type annotations, though, the checks help a lot, and the aspect of checking that theory research often ignores—called blame assignment[9]—produces the most relevant information. The authors' general goal is to understand pragmatics information—using the rational programmer as their instrument. The next section addresses what the rational programmer delivers, how it works, and what it is not—a human being.

The rational-programmer method is the first reasonably general approach for assessing whether linguistic features and tools can deliver helpful information with software development tasks.

The Rational Programmer

As Morris[28] stated in his seminal 1968 dissertation, an investigation of programming languages must investigate syntax, semantics, and pragmatics. Syntax is a problem whose nature lends itself to precise mathematical and engineering investigations, and so is semantics. Researchers have therefore focused on these aspects. By contrast, pragmatics has been considered a nebulous concept, because it is about the concrete tasks developers face when they use a language. Investigating pragmatics thus seems to call for human studies, observing how people extract information from syntax and semantics plus how people use it in different situations.

A close look at this description suggests that jumping to human studies means taking several steps at once; that is, (a) checking whether syntax and semantics produce relevant information, (b) programmers understand this information, and (c) programmers act on this information. While human-subject studies are needed to deal with (b) and (c), it should be possible to investigate (a) without involving people as subjects. Indeed, this separation of concerns suggests that it makes sense to study whether human programmers understand the information and act on it only if an investigation of question (a) confirms its existence, its accessibility, and its actionable nature.

Questions about the information-content of language features resemble the questions economists face when they began to think about the effectiveness of interventions in the economy—the pragmatics of economic policy. In response, Mill[26] decided to construct and investigate an artificial economic actor: *homo economicus*. His idea was that *homo economicus* acts rationally, using all available information to make beneficial decisions in the realm of economics. While Mill's idea at first suggests striving for benefit means maximizing profit or minimizing cost, many economists have revisited and refined his idea since then; Simon's[31] ideas of *bounded rationality* and of *satisficing* profit goals stand out.[a]

The rational programmer method is the authors' response to the question on programming-language pragmatics. A rational programmer is a software actor that mechanically uses a linguistic feature to solve a specific problem. Like *homo economicus*, a rational programmer is an idealization—an abstraction that does not exist in the real world. No developer acts rationally in the sense of this abstracted programmer or even in a bounded-rational manner. But, assuming bounded rationality with respect to

a Although *homo economicus* is the foundation of *classic* economics, models resting on it explain only some macroeconomic phenomena and miss many others. *Behavioral economics* starts from the alternative assumption, namely, that economics must study the non-rational decision-making processes of human beings.

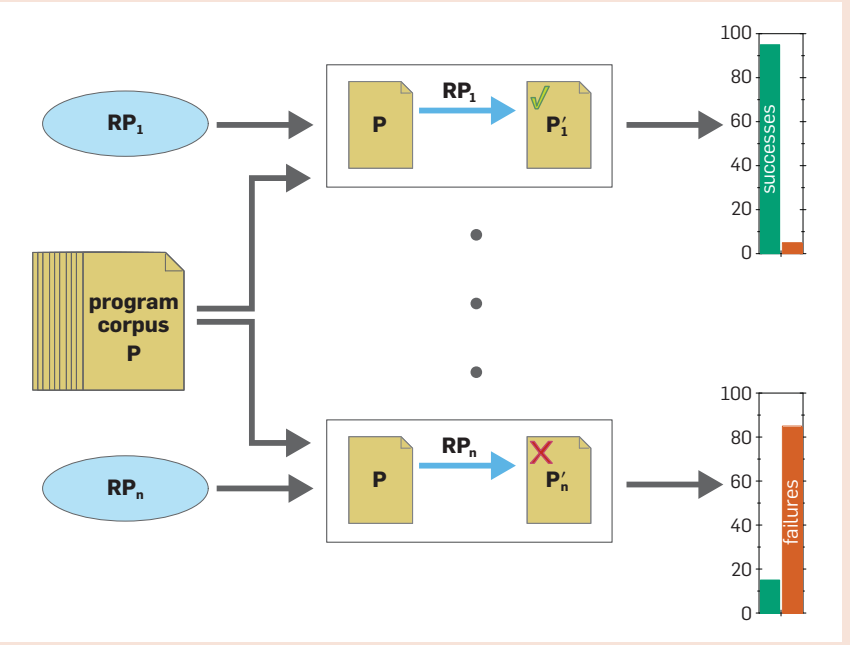
a chosen linguistic feature or tool enables a way of investigating pragmatics information.

Technically speaking, a rational programmer is an algorithm that, with a bounded effort, exploits information from *one specific language feature* to solve *a specific problem*. Concretely, it starts from a program P that suffers from a problem and acts on information to solve the problem; in the process, it is likely to edit P to obtain P', a program variant that represents a solution. In fact, a rational-programmer experiment may involve a number of rational programmers; each algorithm corresponds to a different hypothesis of the language designers about a language feature. Applying all algorithms to a large representative corpus of problematic programs may then yield insight into the value of the information that the investigated feature provides for this problem. Creating the rational-programmer algorithms as well as the representative scenario corpus requires problem-specific research; the experimental setup, though, remains the same. See Figure 1 for an overview.

In sum, the rational-programmer method employs large-scale experimentation with idealized behavior to check hypotheses about the information content of language features. This article first illustrates the idea with a concrete example. After sketching some more uses of the rational programmer method, the article presents a general schema. Following this generalization, it examines the labor involved in rational-programmer experiments. The final sections of the article relate the rational programmer to human programmers in two different ways. The article concludes with a call to arms.

Figure 1. A schematic overview of the rational-programmer experiment.

A rational programmer (RP) experiment consumes two inputs: a corpus of problematic programs P (in yellow) and a number of distinct RPs (in blue). Each program P exhibits a specific kind of problem. For example, all programs P in the given corpus exhibit the same kind of bug or suffer from similar performance bottlenecks. Each RP is an algorithm that extracts information from a language feature to solve the problem in a given program P. Each RP attempts to solve the problem by transforming P into a variant P' (blue arrow). Two different RPs may act differently on the same extracted information. Or, two RPs may use different flavors of the feature; doing so affects the information the RPs observe. In this way, each RP reifies a hypothesis about how the feature can produce helpful information in the given work situation. The RP experiment applies all RPs to all problematic programs in the corpus and records the outcomes. Given a problematic problem P and a RP, each outcome is either a success or a failure. An analysis of the collected outcomes explains how well different RPs perform. In short, the experiment puts to test and compares a number of hypotheses about programming-language pragmatics in an automated fashion.



Pragmatics by Experiment

The rapid development of mixed-typed[b] languages over the past decade sets up a perfect example of how the rational-programmer method can yield surprising insights. A mixed-typed language allows programmers to mix typed and untyped pieces of code in a single program. This mixing can happen in many different ways, but most frequently a programmer may link a typed piece of code to an existing untyped library in the same language or a programmer may write an untyped script that imports a typed module.

Microsoft's realization of this idea in the form of TypeScript has taken the developer world by storm. Many Web developers reach for TypeScript instead of JavaScript because they like types and can easily continue to link to the many useful, preexisting, and untyped libraries. On the academic side, Typed Racket[36] is the most robust realization of the idea. It has found some use in the real world, has extensive applications in academia, and provides a solid platform for programming-language investigations.

The designs of TypeScript and Typed Racket share similarities and yet differ in ways that inspire a rational-programmer experiment. Their type systems resemble each other closely. Both use occurrence typing,[37] and both come with sophisticated types for object-oriented programming.[33] Concerning their semantics, they differ in that they deal with type mismatches rather differently. A *type mismatch* occurs when untyped code and typed code exchange values that do not conform to the specified types.

A reader may wonder how a well-typed program can possibly go wrong.[27] It is of course not the typed code alone that causes type mismatches but the mixing of typed and untyped code. When such a mixture runs, untyped code can send a value into typed code that does not match the expected type. In the TypeScript world, a first, well-known cause is that the types imposed on untyped code are flawed. For example, the DefinitelyTyped re-

b The term "mixed-typed" covers optional type systems,[32] plug-in type systems,[3] gradual type systems,[30] and migratory type systems.[35]

Figure 2. A simple type-mismatch problem between JavaScript and TypeScript.

This simplistic program illustrates a type-mismatch problem. The code in the box on the left is a TypeScript "module" that implements a simple bank account module. The code in the box on the right is a JavaScript client "module" that imports the bank account functionality. The first call to `deposit` supplies a number as an argument; the follow-up call to `printbalance` correctly prints `'balance: 100'`. The second call to `deposit` supplies the string "`pennies`". Neither the code generated by the TypeScript compiler nor the JavaScript VM signal an error, even though the type specification explicitly requests a number. The final request to see the balance prints `'balance: 100 pennies!'`—a wrong answer with which no customer would be happy.

Bank.ts

```
var balance = 0;
export function deposit( amt: number ) {
  balance += amt;
}

export function printBalance() {
  console.log("balance: " + balance);
}
```

Client.js

```
const Bank = require('./Bank.js');

// logically correct interaction
Bank.deposit( 100 );
Bank.printBalance();

// logically incorrect interaction
Bank.deposit( "pennies!" );
Bank.printBalance();
```

pository[c] collects modules that import untyped libraries and re-export them with type specifications so TypeScript can type-check the importing module. In most cases, these adapter modules are programmed by developers other than those who created the libraries. Unsurprisingly, this results in flawed type specifications. Researchers (for example, Christiani and Thiemann,[4] Feldthaus and Møller,[7] Hoeflich et al.,[15] and Kristensen and Møller[16]) have investigated this problem and have found numerous such flaws. A second cause is dual to the first; the untyped code suffers from bugs. That is, the untyped code is supposed to live up to some type specification, but a bug occasionally causes a type mismatch at run-time. See Figure 2 for a TypeScript example.

Given the possibility of type mismatches, a language designer can choose one of a few alternative checking regimes:

1. **Ignore them**. The compiler checks and then erases types as it translates a program. The resulting code performs no run-time checks to enforce type integrity. If, for example, some untyped library calls an integer function with `"42"`, the mismatch may never be discovered during execution. The literature dubs this approach *erasure semantics*. TypeScript is the most prominent design using an erasure semantics.

2. **Notice them as early as possible**. The compiler translates types into run-time checks that enforce their integrity. When these checks fail, they raise an exception. Consider an untyped library that accidentally calls back a string-typed function with the number `42`. The explicit run-time checks of this second alternative are going to notice this problem as soon as it happens, and the exception-associated stack trace is going to be close to the problem-discovery point.

3. **Notice them and try to pinpoint a source**. The Typed Racket compiler can go even further and associate a message with these exceptions that assigns blame to a specific piece of untyped code, warning developers that this blame is useful only if the corresponding type specification is correct.

Given these alternative checking regimes, choosing from them should be understood as a prototypical question of language feature pragmatics:

> *which checking regimes deliver helpful information for locating the source of type-mismatch problems?*

A rational-programmer investigation can answer such questions to some extent. The remainder of this section explains how; readers interested in details should consult the work of Lazarek et al.[19,20]

Setting up a truly scientific experiment requires that everything except for the run-time checking regime of the investigated language remains the same. At this point, Typed Racket[10] is the only language that satisfies this desiderata because it implements all three alternative checking regimes.

Equipped with a suitable experimental environment, preparing a rational-programmer experiment is a two-step process. Step 1 calls for the identification of a large, representative corpus of problematic programs. To mechanize the experiment properly, a problematic program should be one with a *single*, *known* type-mismatch problem so that the experimental framework can automatically check the success or failure of a rational programmer. Furthermore, the problem should be a mis-specification of a type or a bug in an untyped piece of the program. No such ready-made corpus exists, but it is possible to create such a corpus from a representative collection of correct programs.[11] Starting from this collection, applying appropriate mutation operators[5] yields millions of suitable problematic programs; selecting a representative sample of tens of thousands supplies the corpus. For the statistical analysis of the selection, the reader may wish to consult the already-mentioned papers.

Step 2 demands the translation of hypotheses into rational programmers. Since the goal is to find out which checking regimes deliver helpful information for locating the source of type-mismatch problems, a rational programmer should try to strategically squeeze as much information from such checks as available.

Each rational programmer imple-

c See https://tinyurl.com/omsgny7

ments the same strategy, parameterized over the checking regime. The strategy is to run program P until execution stops due to an exception and to then inspect the available information from this failure. In one way or another, these exceptions point to an untyped piece of code. By equipping this piece with types, a rational programmer obtains P', which it tries to compile and run. If type checking P' fails, the experiment is a success because the type-mismatch problem has been discovered statically. Otherwise, P' type-checks, and running it again restarts the process. A rational programmer declares failure when it cannot act on the available information. See Figure 3 for a diagrammatic summary.

A key detail omitted from the diagram is how the rational programmers equip untyped pieces of code with types. As it turns out, each of the programs in the chosen collection[11] comes in two forms: typed and untyped. Moreover, all typed and untyped pieces can be mixed seamlessly—a property that the problematic programs in the corpus inherit by construction. Thus, the rational programmers can easily annotate untyped pieces of code with types by replacing it with its corresponding typed version.

The three alternative compiler designs suggest three rational programmers:

▸ **Erasing**. The erasure semantics may assign a program with a type-mismatch a behavior that is seemingly normal or that triggers an exception from the underlying virtual machine. Since such exceptions come with stack traces, the **Erasing** rational programmer can inspect this trace and replace the untyped piece of code closest to its top.

▸ **Exceptions**. When Typed Racket's run-time checks fail, they also display a stack trace. Like the Erasing rational programmer, the **Exceptions** one replaces the top-most untyped piece of code with its typed counterpart.

▸ **Blame**. The **Blame** programmer exploits the blame assignments that come with Typed Racket's failing run-time checks. It replaces the blamed piece of code with its typed version.

All three rational programmers proceed in the same manner, and thus the experimental setup may count (**S1**) how often the algorithm finds the single, planted bug, and if it does find it, (**S2**) how many replacements are needed.

An experiment needs a control:

▸ **Null**. The null-hypothesis programmer **randomly** chooses an untyped piece of code. This **Null** rational programmer always finds the problem (**S1**: 100%), because it eventually replaces all pieces of code with their typed versions. But, to get there, it may have to replace many untyped code pieces (**S2**: usually a large count).

Both theoretical investigations and developer anecdotes suggest that substantial benefits flow from run-time checks for locating type mismatches. Checks should discover mismatches early to avoid the uncontrolled and misleading propagation of faulty values. Furthermore, their stack traces are closer to the discovery of the problem, and the blame assignments in their exception messages seem to represent particularly useful information.

Concerning the search for the source of type mismatches (**S1**), the results of the rational-programmer experiment are somewhat surprising, however:

▸ When the bug is located in the type specification imposed on untyped code, the conjectured benefits are confirmed.

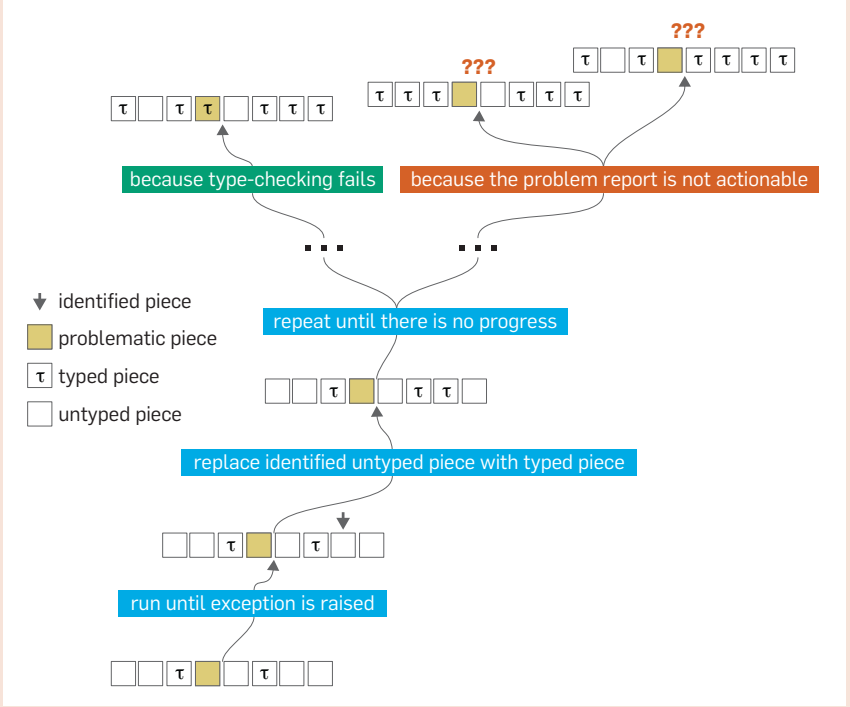
▸ When the bug is located in the untyped code, the expected benefits disappear. While **Blame** supplies information that is somewhat better than **Exceptions** and **Erasing**, the three differ only a little.

Concerning the effort (**S2**), all strategies need significantly fewer replacements than **Null**. Its existence thus confirms that the other three algorithms deliver useful information. Unsurprisingly, **Blame** fares the best; it needs the smallest number of replacements.

In sum, **Blame** provides the most helpful information for locating the source of problematic type specifications for untyped pieces of code. **Excep-**

Figure 3. How a rational programmer searches for the source of a mismatch.

The experiment confronts a rational programmer (RP) with a problematic program (bottom left), that is, a program with one known type mismatch (in yellow). The RP runs the program using its chosen run-time type-checking regime until it raises an exception, whose report points to an untyped piece of code as the source of the problem. The RP then replaces the identified untyped piece with its typed counterpart and runs the program again. This process is repeated until the untyped piece of code that causes the type mismatch becomes typed (top left), forcing the type checker to signal a type error, or until the problem report does not come with actionable information for the RP. In the first case, the type error exposes the problem, and the search succeeds; in the second case, it fails.



tions is essentially only as helpful as **Erasing**. For bugs in untyped code that cause type mismatches, the advantage of **Blame** over the others shrinks substantially.

The rational programmer versus theory. The results are particularly surprising when compared to the predictions of programming languages theory. Theoretical investigations predict that the run-time checking semantics finds every type-mismatch problem that the erasure semantics discovers—and finds it earlier than the erasure semantics.[13] The results of the rational-programmer experiment point out that this theoretical prediction does not directly translate into practice. Indeed, there are two problems:

1. Theoretical publications on mixed-typed languages focus on run-time checking. But, the investigation indicates that, for a broad variety of type mismatches and benchmarks, a language using an erasure semantics discovers and locates most of the type-mismatch problems anyways.

2. Many theoretical papers ignore the blame assignment part of run-time checking. But, the investigation shows that it is the key element to making run-time checks informative. Readers should interpret these two observations as blind spots of theoretical investigations in this research area.

Caveats. While these large-scale simulations look impressive, their interpretation must take into account that they apply only to Typed Racket. Its underlying untyped language, Racket, enforces strict preconditions for each of its primitives, which has significant implications for when and how the erasure semantics delivers information. By contrast, JavaScipt, the language underlying TypeScript, enforces lax preconditions. In all likelihood, lax enforcement in the underlying language will make a run-time checking regime more effective than in a Racket-like setting. The only way to check this conjecture is to reproduce the experiment with TypeScript.

As for every empirical investigation, the rational experiment described in this section comes with technical caveats, such as whether the corpus is truly representative, whether the statistical sampling is appropriate, or whether the presence of more than one bug affects the search and how. To mitigate them, the design of the experiment comes with an array of checks and controls. For instance, the experiment's corpus originates from correct programs written by different developers for different purposes and exhibit a variety of programming styles. They also vary in terms of language features, complexity, and size. Moreover, the mutants of these correct programs that form the experiment's corpus have been mechanically analyzed to confirm their suitability; they contain a wide range of non-straightforward type mismatches that structurally resemble issues reported by Typed Racket programmers. For an account of how the experimental design mitigates the technical caveats, the reader may wish to consult the papers by Lazarek et al.[18-20]

Concerns of Pragmatics Are Ubiquitous

The preceding section illustrates the rational-programmer method through the investigation of one concern: the various semantics of mixed-typed programs and the problem of finding the source of type-mismatch problems. Its results partially contradict and partially confirm hypotheses based on theoretical research.

This section reports some of the authors' experience with related pragmatics concerns and sketches how to go beyond. Concerns of pragmatics exhibit tremendous variability: the linguistic features considered, the work situations, and the chosen programming language. Given how developers tend to consider the available software tools as part of the chosen language, it is even natural to investigate the pragmatics information in alternative tool sets.

Here is a variant of the concern from the preceding section:

> *do assertions and contracts deliver helpful information for locating the source of different classes of logical bugs?*

The point of assertions and contracts[24] is to detect problems as early as possible. Once detected, the practical question is how to fix the problem, and the question of pragmatics is whether the violation of the specification provides developers with helpful information.

Again, the results of running a rational-programmer experiment are somewhat surprising. While an early variant of the RP experiment seemingly validated the folklore wisdom behind software contracts,[21] it lacked a NULL hypothesis. A subsequent reproduction run of the experiment with a NULL-hypothesis rational programmer appears to weaken the evidence.[18]

Mixed-typed languages also suggest investigations into the pragmatics of tools. Concretely, a mixed-type language with run-time checking tends to create performance problems, frictions between the typed and untyped parts of the program. The problem was discovered in the context of Typed Racket almost a decade ago[33] and confirmed in other settings.[13] Since profiling tools are one way to debug performance problems, and since Racket comes with two of them[2]—each of which supplies different feedback information—tool designers are confronted with a concern of pragmatics. When tools are concerned, the question is not just *which one to pick* but also *how to use it*. In other words, exploring the information delivery of a tool involves usage strategies, meaning the question becomes

> *Which combination of profiling tool and strategy delivers useful information for debugging performance problems in mixed-type languages?*

Running appropriate rational-programmer experiments confirms that one of the two profiling tools produces more valuable information in general; the surprise lurks in which strategies are successful and which ones are not. The interested reader can find the details of these results in a forthcoming paper by Hejduk et al.[14]

While the first two examples of experiments confirm the usefulness of the rational programmer within the linguistic environment of Racket, pragmatics questions arise whenever a language evolves and can be addressed with rational-programmer experiments. Consider the evolution of the Rust programming language and specifically the borrow-checking part of its type checker. This algorithm has

changed in significant ways over the past decade. Hence, a question to be investigated is

> *whether the choice of borrow-checking algorithm affects the expressive power of the Rust language.*

A plausible rational-programmer experiment tailored to this question could turn hypotheses about how borrow-checking algorithms affect expressive power into strategies of semantics-preserving transformations. The rational programmers would apply such transformations to a corpus of Rust programs that differ in whether two borrow-checking algorithms accept or reject them. A rational programmer would succeed if its transformation convinces the rejecting borrow-checking algorithm to admit the programs. Moreover, if the proportion of simple local transformations over global ones is high for a successful rational programmer, then the two borrow-checking algorithms may not affect the expressive power[8] of the language in a significant manner. Evidently, the details of the rational programmers and the corpus of programs are the two key challenges for turning this sketch into an actual experiment.

The structure of rational-programmer experiments remains similar across the presented spectrum of pragmatics concerns. For each of them, the experimenter must answer the following questions:

- Do variants of the same feature or tool exist?
- Is it possible to create rational programmers for each of the hypotheses about the information that the investigated feature or tool produces?
- Is the success for these rational programmers decidable?
- Does a representative corpus of problematic programs exist or can it be generated? (Each member of this corpus should exhibit one known relevant problem.)

This common structure also suggests the adaptation of the presented experiments to other language contexts: The experiment from the preceding section clearly applies to TypeScript; an experiment with executable specifications in Java should shed light on the pragmatics information resulting from violations, and exploring the expressiveness of type system variants may apply beyond Rust.

Successful RP experiments do not replace human studies. In fact, rational-programmer experiments and human studies are complementary.

Rational Pricing of Pragmatics

Rational-programmer experiments are costly in terms of human labor. These costs come in three different flavors: alternative implementations of features and tools, the experimental platform, and intellectual innovations in the investigated domain.

First, an experiment usually requires the implementation of two or more variants of a language feature or tool. When such implementations do not exist, new implementations for the sake of the experiment rarely need to meet production-level standards; prototypes tend to suffice. When multiple production implementations already exist, as is often the case with tools, this cost reduces to prototyping the space of usage strategies. In other words, this cost is analogous to the labor expended by designers and engineers in other fields when they create prototypes to test design hypotheses.

Second, the construction of the experimental framework requires labor. The size of the experimental corpus, the number of the rational programmers, and the complexity of the usage strategies call for a sophisticated infrastructure. Specifically, the infrastructure should assist with breaking the experiment into piecemeal tasks so that a run of an experiment can take advantage of clusters to execute tasks in parallel. As a run produces information, it should be possible to automatically perform validity checks so problems with an experiment are discovered as early as possible.

Although the workflows and tasks of rational-programmer experiments vary, the authors' experience indicates that different classes of experiments can share large pieces of the infrastructure—as long as it is carefully grown and properly organized. In this regard, the design of an optimized software framework for rational-programmer experiments seems like a promising way of mitigating these infrastructure-related costs and effectively managing the resources needed for running a rational-programmer experiment.

Third, each experiment poses two intellectual challenges: turning hypotheses into rational programmers

and constructing the experimental corpus. For the first challenge, if two experiments share an aspect of their combination of feature and work situation—such as the authors' investigations into mixed-typed languages and contracts—it is possible to reuse some ideas. For instance, the authors reused the idea of strengthening boundaries between pieces of code for the two investigations. For the second challenge, the authors were also able to reuse a carefully curated starter collection of programs for multiple experiments. Moreover, they reused the idea of mutation to generate a corpus of problematic programs from this collection, albeit the operators significantly differed between experiments. Since languages nowadays come with such representative starter collections of programs, running rational-programmer experiments in alternative language contexts should benefit from those.

Ultimately though, these intellectual challenges and their solutions are tied to the domain of pragmatics concerns at hand. Even for the experiment concerning mixed-typed languages, two different sets of mutation operators were needed: one for injecting bugs while respecting the type discipline and another for modifying type specifications while preserving the ability to run the program. In the end, rational-programmer experiments do ask for ingenuity and creativity.

From Pragmatics to Action

Preceding sections sketch how rational-programmer experiments can validate that particular uses of language features deliver intrinsic, task-specific information. Once this validation is available, the question arises as to what can be done with it. Two obvious ideas come to mind: Language designers can use this information as one factor in making decisions, and university instructors can leverage the information for course designs.

Language designers tend to weigh design alternatives against each other. The creators of TypeScript in all likelihood considered the most basic choice, namely, whether the integrity of type annotations should be enforced at run-time. They chose not to add run-time checks because they imagined a work situation in which developers are focused on performance. If they consider the work situation of finding the source of type-mismatch problems in DefinitelyTyped libraries instead, they might wish to reproduce the previously discussed rational-programmer experiment. Assuming this reproduction were to yield similar results, it would suggest making run-time checks available as an optional debugging aid.[d]

While the original design rationale is justified by performance considerations, the implications of a RP experiment will help students understand and contrast alternative design choices in light of other work situations.

In general, rational-programmer experiments can become an integral part of the *feedback loop* governing language design and implementation. When designers and implementers face a dilemma concerning syntactic or semantic choices, the rational-programmer offers a new instrument for evaluating the alternatives. They can:

- Prototype the variants of the corresponding feature or tool.
- Turn their ideas about task-specific information of the variants into rational programmers to run an experiment.
- Use positive results to enrich the documentation or to construct tools that support proper usage strategies.
- Feed negative results into a redesign step.

Concisely put, rational-programmer experiments can help designers avoid premature commitments to design alternatives.

University instructors tend to present syntactic and semantic concepts in typical courses on the principles of programming languages—sometimes informally, other times via implementations or formal models. But, they know they should also teach about pragmatics concerns, which is what the typical lectures on lexical versus dynamic scope for variable declarations illustrate: It is easy to explain how lexical scope enables modular reasoning about variable references and dynamic scope interferes with it.

When students return from internships or co-ops, at least some will have experienced type-mismatch problems in the context of TypeScript. An instructor can take this experience as a motivation to contrast the official design rationale of TypeScript—it is JavaScript once types are checked and

d Offering both may be necessary because the run-time checking implementation occasionally imposes a large performance penalty.[33]

erased—with the results of rational-programmer experiments. While the original design rationale is justified by performance considerations, the implications of a rational-programmer experiment will help students understand and contrast alternative design choices in light of other work situations, in particular, the benefits of runtime checks when developers wish to locate the source of mistakes in type annotations. More generally, presenting the results of rational-programmer experiments may help students understand design alternatives and design decisions, plus the rationales behind them, in concrete terms.

From Rational to Human Programmers

The authors know that human studies may be needed to understand how results from rational-programmer experiments relate to human actions or entail concrete suggestions for human programmers. Such studies might start with training one set of participants in the systematic application of successful rational-programmer strategies. Based on this training, observations of a group of trained programmers and a control group could determine how well programmers can apply their training and whether doing so makes them more effective at the particular task than untrained programmers.

The general point is that successful rational-programmer experiments do *not* replace human studies. In fact, rational-programmer experiments and human studies are complementary as they investigate related but distinct facets of how programming language ideas can benefit developers. While the rational programmer is concerned with the presence of potentially useful information in features and tools in a given work situation, human studies examine whether human developers can extract, interpret, and effectively use that information. In a sense, the relationship between the two can be viewed as analogous to the relationship between classic and behavioral economics:[38] Human studies can contradict some of the predictions based on rational-programmer experiments and thus help researchers identify weaknesses in classic models. Strictly speaking, rational-programmer experiments directly suggest human studies by refining hypotheses, corresponding usage strategies, and a corpus of programs to examine from a human-factors perspective.

In some cases, researchers do not need rational-programmer experiments. They can intuit that language features deliver pragmatics information that entails an obvious use and can evaluate their intuitions with simple experiments. Key is that such intuitions can be translated into a tool designed around highly structured, limited dialogues with the developer. Consider the interactive fault localization tool of Li et al.[22] The developer asks the tool to help find a bug in a program, and the tool responds with facts about the most suspicious parts of the code. The developer reacts to the facts by marking them as expected or unexpected. The tool uses this feedback to refine its fact generation until, after a number of dialogue steps, it produces a single fact that directly identifies the bug. The limited, structured way developers interact with such tools points to the way for evaluating them via simulation. Specifically, Li et al. simulate the usage of their tool with an oracle that provides always-perfect feedback as a substitute for user reactions. Similarly, to evaluate their tool for locating faults in spreadsheets, Lawrence et al.[17] construct a stochastic model of user reactions based on data collected from human users.

In other cases, the existence of pragmatics information is clear, and human-subject studies can directly help understand how developers can beneficially react to the pragmatics information. The work of Marceau et al.[23] is a good example. It exposes a direct relationship between the quality of error messages of some instructor-chosen teaching language and the process of eliminating errors by novice programmers. Concretely put, they report how subjects fix mistakes in programs much more quickly when the error messages use the same terminology as the text book and explain themselves (via a color scheme) in terms of pieces of the students' code. Similarly, Alimadady et al.[1] study the value of a new debugger for understanding asynchronous JavaScript code via the observation of professional developers. Their work shows that developers fix code much more quickly with the help of the novel tool when compared to a control group without access.

Pragmatics, the Neglected Question

Returning to the point of a scientific investigation of pragmatics, searching for pragmatic information in a feature or tool means focusing on one feature, observing its role in one task, and extracting as much information as possible from this combination. The rational-programmer method fits this specification: It replaces the human programmer with an algorithmic approximation that uses a feature as systematically as possible, it runs this algorithm on as many task-specific problems as feasible, and it measures progress toward the goal of the specific task.

From this angle, the rational programmer is a model. Language researchers know that despite their simplified nature, models have an illuminating power, in both theory and practice. When the typical paper at a Principles of Programming Languages (POPL) conference states a theorem about, say, the soundness of a type system, it does not claim that it applies to a language implementation and its vast set of libraries. Instead, the paper simplifies this system down to a small mathematical model, and the theorem applies to just this model. Yet, despite this simplification, theory has provided valuable guidance to language designers. Similarly, when the typical paper at a Programming Language Design and Implementation (PLDI) conference reports run-time measurements for a new compiler optimization, the authors have highly simplified models of program execution in mind. As Mytkowicz et al.[29] report, ignorance of these simplifications can produce wrong data—and did so for decades. Despite this problem, the simplistic performance model acted as a compass that helped compiler writers improve their product substantially over the same time period.

In the same way, rational-programmer experiments of pragmatics can confirm the *presence* of potentially useful information in language features and tools. They do yield results of different qualities depending on the specifics of their rational program-

mers. In some experiments, a rational programmer acts radically differently from a human programmer. While the first exclusively exploits the addition of types to the program to gain information about the type-mismatch location, the second is in all likelihood going to use many different sources, including plain hunches. The experiment does indicate that human programmers might benefit from adding types if they are willing to spend the effort of formulating them, and if the bug is located in type specifications. By contrast, for other experiments, both the rational and the human programmer are certain to take some similar steps reacting to a problem—for instance, when facing a performance problem both rational and human programmers are likely to use a profiling tool to understand the problem. In such cases, as indicated by this article's previous brief discussion on the pragmatics of profiling, the experiment can suggest which tool human developers should use and *how* they should use it to benefit from the pragmatics information.

The rational-programmer method cannot confirm the *absence* of useful information. By its very definition, a pragmatics experiment is about the use of features and tools in specific situations. Hence, the data gathered concerns a specific use case. While generalizing from this use case would violate basic principles of science, such a lack of pragmatics information in an experiment still enables language designers and instructors to draw valuable lessons about use strategies and to check into the improvement of features and the construction of supporting tools.

For now, the rational-programmer method is the first reasonably general approach for assessing whether linguistic features and tools can deliver helpful information with software development tasks. The authors' hope is that others will be inspired to conduct similar experiments, to reflect on the question of pragmatics, and to develop additional evaluation methods for this central concern of developers and language creators.

Acknowledgments

The authors thank Robby Findler, Ben Greenman, Nathaniel Hejduk, Alexis King, Caspar Popova, and especially Lukas Lazarek for their collaboration on early rational-programmer projects. Stephen Chang contributed the example of a type mismatch in TypeScript. The National Science Foundation has partially supported this research with several grants (SHF 2007686, 2116372, 2315884, 2412400 and 2237984). C

References

1. Alimadadi, S., Mesbah, A., and Pattabiraman, K. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the Intern. Conf. on Software Engineering*. ACM (2016), 1169–1180.
2. Andersen, L., St-Amour, V., Vitek, J., and Felleisen, M. Feature-specific profiling. In *Trans. on Programming Languages and Systems 41*, 1, Article 4 (2019), 34.
3. Bracha, G. and Griswold, D. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications*. ACM (1993), 215–230.
4. Cristiani, F. and Thiemann, P. Generation of TypeScript declaration files from JavaScript code. In *Proceedings of the Intern. Conf. on Managed Programming Languages and Runtimes*. ACM (2021), 97–112.
5. DeMillo, R.A., Lipton, R.J., and Sayward, F.G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.
6. ECMA International. *ECMA-262: ECMAScript Language Specification* (16th ed.). European Association for Standardizing Information and Communication Systems (2025); `https://tc39.es/ecma262/`.
7. Feldthaus, A. and Møller, A. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications*. (2014), 1–16.
8. Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming 17*, 1 (1991), 35–75.
9. Findler, R.B. and Felleisen, M. Contracts for higher-order functions. *Proceedings of the ACM SIGPLAN Intern. Conf. on Functional Programming*. ACM (2002), 48–59.
10. Greenman, B. Deep and shallow types for gradual languages. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM (2022), 580–593.
11. Greenman, B. GTP benchmarks for gradual typing performance. In *Proceedings of the ACM Conf. on Reproducibility and Replicability*. ACM (2023), 102–114.
12. Greenman, B., Felleisen, M., and Dimoulas, C. Complete monitors for gradual types. *Proceedings of the ACM on Programming Languages 3, Object-Oriented Programming Systems, Languages and Applications* (2019), 122:1–122:29.
13. Greenman, B. and Migeed, Z. On the cost of type-tag soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM (2018), 30–39.
14. Hejduk, N., Greenman, B., Felleisen, M., and Dimoulas, C. How profilers can help navigate type migration. (2024). unpublished manuscript; submitted for review.
15. Hoeflich, J., Findler, R.B., and Serrano, M. Highly illogical, Kirk: Spotting type mismatches in the large despite broken contracts, unsound types, and too many linters. *Proceedings of the ACM on Programming Languages 6, Object-Oriented Programming Systems, Languages and Applications* (2022), 142:1–142:26.
16. Kristensen, E.K. and Møller, A. Type test scripts for TypeScript testing. *Proceedings of the ACM on Programming Languages 1, Object-Oriented Programming Systems, Languages and Applications* (2017), 90:1–90:25.
17. Lawrance, J., Abraham, R., Burnett, M., and Erwig, M. Sharing reasoning about faults in spreadsheets: An empirical study. In *Proceedings of the IEEE Symp. on Visual Languages and Human Centric Computing*. IEEE (2006), 35–42.
18. Lazarek, L. *An Investigation of the Pragmatics of Debugging With Contracts and Gradual Types*. Ph.D. Dissertation. Northwestern University, 2024.
19. Lazarek, L., Greenman, B., Felleisen, M., and Dimoulas, C. How to evaluate blame for gradual types. *Proceedings of the ACM on Programming Languages 5, Intern. Conf. on Functional Programming* (2021), 68:1–68:29.
20. Lazarek, L., Greenman, B., Felleisen, M., and Dimoulas, C. How to evaluate blame for gradual types, part 2. *Proceedings of the ACM on Programming Languages 7, Intern. Conf. on Functional Programming* (2023), 194:1–194:28.
21. Lazarek, L. et al. Does blame shifting work? *Proceedings of the ACM on Programming Languages 4, Symp. on Principles of Programming Languages* (2020), 65:1–65:29.
22. Li, X., Zhu, S., d'Amorim, M., and Orso, A. Enlightened debugging. In *Proceedings of the Intern. Conf. on Software Engineering*. ACM (2018), 82–92.
23. Marceau, G., Fisler, K., and Krishnamurthi, S. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM Technical Symp. on Computer Science Education*. ACM (2011), 499–504.
24. Meyer, B. Design by contract. *Advances in Object-Oriented Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, (1991), 1–50.
25. Microsoft Corporation. TypeScript: JavaScript with Syntax for Types. 2025; `https://tinyurl.com/pcmuvnc`.
26. Mill, J.S. *Essays on Some Unsettled Questions of Political Economy*. Longmans, Green, Reader, and Dyer, London, U.K. (1874).
27. Milner, R. A theory of type polymorphism in programming. *J. of Computer and System Sciences 17*, 3 (1978), 348–375.
28. Morris, J.H. *Lambda-Calculus Models of Programming Languages*. Ph.D. dissertation. Massachusetts Institute of Technology (1968).
29. Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P.F. Producing wrong data without doing anything obviously wrong! In *Proceedings of the ACM Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM (2009), 265–276.
30. Siek, J.G. and Taha, W. Gradual typing for functional languages. In *Workshop on Scheme and Functional Programming*. University of Chicago (2006), 81–92. TR-2006-06.
31. Simon, H.A. *Administrative Behavior*. MacMillan Publishers, New York, NY (1947).
32. Steele, G.L. Jr. *Common Lisp* (2nd ed.). Digital Press, Woburn, Mass, (1990).
33. Takikawa, A. et al. Is sound gradual typing dead? In *Proceddings of the ACM SIGPLAN Symp. on Principles of Programming Languages*. ACM (2016), 456–468.
34. Takikawa, A. et al. Gradual typing for first-class classes. In *ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications*. ACM (2012), 793–810.
35. Tobin-Hochstadt, S. and Felleisen, M. Interlanguage migration: From scripts to programs. In *Proceedings of the Dynamic Languages Symp*. ACM (2006), 964–974.
36. Tobin-Hochstadt, S. and Felleisen, M. The design and implementation of typed scheme. In *Proceedings of the ACM SIGPLAN Symp. on Principles of Programming Languages*. ACM (2008), 395–406.
37. Tobin-Hochstadt, S. and Felleisen, M. Logical types for untyped languages. *Proceedings of the ACM SIGPLAN Intern. Conf. on Functional Programming*. ACM (2010), 117–128.
38. Tversky, A. and Kahneman, D. Advances in prospect theory: Cumulative representation of uncertainty. *J. of Risk Uncertainty 5*, (1992), 297–323.
39. Wright, A.K. and Felleisen, M. A syntactic approach to type soundness. *Information and Computation 115*, 1 (1994), 38–94.

Christos Dimoulas is an assistant professor, Department of Computer Science, McCormick School of Engineering, Northwestern University, Evanston, IL, USA.

Matthias Felleisen is a trustee professor, College of Computer and Information Science, Northeastern University, Boston, MA, USA.