Constraints versus Structures in Type Systems

A fledgling cognitive investigation

Hemant Gouni [1], **Will Crichton** [2] **and Jonathan Aldrich** [1]

[1]*Carnegie Mellon University, Pittsburgh, USA*
[2]*Brown University, Providence, USA*

Abstract

The tug-of-war between *constraints* and *structures* sits at the heart of many languages with sophisticated type systems. The main concern is whether it is preferable to *implicitly encode* correctness properties using systems of constraints, or *explicitly represent them* them as algebraic structures that directly capture the properties in question. In this proposal, we argue that the latter approach promises to make complex type system features easier to use. We propose a series of mostly-qualitative slightly-quantitative experiments to investigate these observations in the context of (1) Rust lifetimes and (2) information flow, taking advantage of recent work in reformulating both across the constraints-structures boundary. These experiments carry a focus on developing general and reusable insights about design principles for type systems beyond our setting.

Keywords: Type system design. Human factors. Usability. Cognitive models.

1 Introduction

Systems of constraints are a popular strategy for encoding correctness properties deployed by type system implementers, or occasionally by users of advanced statically typed languages. Some instances that readers may have previously encountered:

1. The Rust programming language [1] employs a flow-sensitive analysis in its borrow-checking pass order to generate subtyping constraints over lifetimes which, for instance, rule out dangling pointers.
2. Information flow type systems [2]–[5] work by generating and checking lattice constraints produced from the flow of data through computation.
3. Typeclass metaprogramming encodes properties—like whether certain functionality is available at a given state—into an organization of typeclass instances constrained to capture the property at hand. Successful instance resolution corresponds to these properties being discharged.

For languages that sport these features, it is not uncommon to see beginner programmers struggling with learning the associated systems of constraints and their connection to the semantics of programs. Indeed, in the case of Rust, lifetimes are often saddled with the majority of the blame for its unusually steep learning curve. A recent proposal [6] changes this to talk directly about sets of aliased variables, departing from lifetime variables with subtyping relationships that relate term variables. This potentially makes it easier to answer questions like 'What parts of the heap are affected by a computation?'.

Similarly, information flow type systems are both long-studied and under-used. A wide range of common security problems facing modern software lie squarely in the domain of information flow, yet it has seen little usage. Information flow is, like Rust's lifetimes, phrased in terms of constraints on program data. Our recent work [7] on information flow has identified an alternative formulation which transforms labels into sets of dependencies that reify the information represented by those labels. As with the sets-of-permissions approach to lifetimes, we hypothesize that our technique enables users to speed up common perceptual tasks, for instance 'Can I call some function with this piece of data I'm holding?' or 'What input types does an output type depend on?'. Within lattice-based information flow, by contrast, these questions require more thought and potentially interrupt concurrent cognition.

In a word, the constraint-based approach enforces program invariants by encoding restrictions as sets of inequalities, while the structural approach reifies these sets of inequalities as domain-specific (algebraic) structures embedded directly into the type system. To study this tension and test our hypothesis that structural models of type systems are often easier to use, we propose a series of experiments to investigate the cognitive concerns that lie along the divide. In particular, easing perceptual reasoning and shortening inferential steps in common tasks feature prominently. We'll first look at constraints as they occur in each of Rust and information flow, explaining in detail the

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

structural approaches in each case. We'll then outline our experiments, highlighting the cognitive elements elicited by each.

2 Constraints in the Wild

2.1 Rust Lifetimes

Rust uses a mechanism called *lifetimes* to track whether two variables (more generally, "places") can alias. Here is a contrived[1] example:

```
1 fn first<'a, 'b, 'c, T>(x: &'a T, y: &'b T) -> &'c T
2 where 'a: 'c {
3   let z = &*x; // z : &'d T, implicitly
4   z
5 }
```

Here, the symbols `'a`, `'b`, and `'c` are the *abstract* lifetimes of the types of `x`, `y`, and the return value. The symbol `'d` is the *concrete* lifetime of the type of `z`. The explicit constraint `'a: 'c` means "`'a` must outlive `'c`." Implicitly, Rust also requires that `'a: 'd` (due to line 3) and `'d: 'c` (due to line 4). Rust uses these outlives-constraints to check for valid aliasing relationships. The implicit constraints generate the requirement that `'a: 'c` by transitivity over `'d`. This requirement is satisfied by the explicit constraint in the `where` clause. However, this function would not type-check if the `where` clause were removed, or if `y` were returned instead.

Broadly speaking, Rust users find lifetimes intimidating [8]. Lifetimes are generally rated as the most difficult part of ownership. Our hypothesis is that a core challenge with lifetimes is the level of indirection between the syntax and semantics of lifetimes. For example, to deduce that `x` can flow into the return value of `first`, one must identify the returned lifetime `'c`, identify the constraints on `'c` (`'a: 'c`), recall the directionality of the constraint ("does `'a` outlive `'c` or is it the other way around?"), then find the uses of `'a` in other types.

Niko Matsakis, a lead developer of Rust, has proposed a structural approach to alias analysis in Rust [6]. Under his proposed system, the example above could be written as something like:

```
fn first<T>(x: &T, y: &T) -> &{x} T {
  let z = &*x; // z : &{x} T, implicitly
  z
}
```

We further hypothesize that a structural approach like Matsakis's would make lifetime-based alias analysis more legible to developers. More precisely, developers should be able to better make low-level inferences about the actions permitted by a given type signature more efficiently, and developers should be able to better make high-level inferences about the semantics and design of a type signature.

2.2 Information Flow

We have observed the constraints-structures distinction rear its head in another domain: information flow. Classically, information flow is done by organizing program data into a lattice and reasoning about computation via the direction of the flows it causes within the lattice. For instance, we may have the lattice `!⊥ ⊏ !password` (read as "bottom is ordered less than `!password`") and the following program typed under it:

```
let p : !password string = "katya"
let check : !a string -> !a bool with !password ⊑ !a =
    function attempt -> p == attempt
```

1 This example is also more verbose than necessary to demonstrate the syntax. The maximally concise signature is `fn first<'a, T>(x: &'a T, y: &T) -> &'a T`. A more complex scenario would be needed to demonstrate when explicit outlives-constraints are required.

We first declare a `string` with label `!password`, followed by a function `check` which compares its input with that string. Note that the label `!a` in `check` is *polymorphic*. It can be instantiated to any label within the stated constraints— so any greater than `!password` in the case of `check`. In the interest of simplicity, we'll be a bit cavalier about which labels are polymorphic in the following– it should be clear from context. So the following program type checks, since it faithfully tracks the dependency on `!password`.

```
let okay : !password bool = check "hello, world!"
```

If we had some `!very_secret` and declared that `!password` $\sqsubset$ `!very_secret`, then `!very_secret bool` would also work as a type for `okay`, too, since `!password` $\sqsubseteq$ `!very_secret`. But the following isn't well-typed, since it would leak a bit of information about the password— namely, whether it is equivalent to `"hello, world!"`. In particular, `!password` $\not\sqsubseteq$ `!⊥`.

```
let bad : !⊥ bool = check "hello, world!" ⚠
```

Even in this simple setting, tasks like 'Which expressions can I call `check` with?' or 'Which of a function's inputs flow to its output at label `!very_secret`?' require contextualizing labels and solving constraints between them within the declared lattice. These are examples of common tasks which should ideally be a matter of applying rote perceptual reasoning, requiring little active thought. In the course of prior work on finding metatheoretically elegant foundations for information flow, an alternative[2] syntax has emerged.

```
let p : [ password ] string = "katya"
let check : [ a ] string -> [ a password ] bool =
    function attempt -> p == attempt
```

We have a nearly identical-looking program, but each `!label` has been turned into a set of *dependencies* `[ d1 d2 d3 ]`. The advantage of this is that users need never contextualize labels within declared constraints. Dependency sets are ordered by subset inclusion, working uniformly for any dependency set (generating the so-called *free* lattice). It is possible to mechanically transform label constraints into dependency sets roughly as follows.

`!password` $\sqsubseteq$ `!a` $\mapsto$ `[ password ]` $\sqsubseteq$ `[ a ]` $\mapsto$ `[ password a ]`

This new setting seems to lose some expressiveness: we can no longer explicitly order labels ourselves, because the ordering is now given structurally. For instance, it seems we cannot control flows into the standard output by requiring that the argument to the `print` function is of some label declared to be less than `!stdout`.

```
let print : !a string -> unit with !a ⊑ !stdout = ...
```

In reality, we can represent this in our setting through the ability to write functions of impossible-looking type `[ alice ] string -> [ stdout ] string`[3] which act as explicit orderings. Here we have `[ alice ]` $\sqsubseteq$ `[ stdout ]`. The explicit ordering can be witnessed by calling our 'impossible' function on the argument to `print`, whose type is given below. Such functions may not always be available— they may only be provided by, for instance, the module or library that introduced the dependency under question. This can be thought of analogously to the way that methods of ML modules can provide functionality which is impossible without knowing the implementation type.

```
let print : [ stdout ] string -> unit = ...
```

2 You may notice that the difference between the two syntaxes parallels the difference between the relational and world-path Kripke semantics of modal logics. This is no accident!

3 The trick is that `alice` here— the label to be transformed— is existentially quantified. If `alice` is universally quantified, this type is inhabited only by the constant function. Also, the explicit orderings we work with here are *proof relevant*.

This nuance aside, the shift to dependency sets streamlines the common activities mentioned above— for instance, determining which of a function's arguments flow to its output— and causes them to become largely perceptual. Users need only scan the function type for symbols (dependencies) to quickly find answers to these questions without much interruption to any deeper logical reasoning processes producing these queries about the program. As a final illustration, here is a simple program typed under lattice-based information flow (in particular, Flow Caml [3])— we encourage readers to pause here and time how long it takes them to interpret the type for `inv_suc` before moving on. What's the relationship between `!c` and `!alice`?

```
let inv : !alice int -> !alice int = fun x -> -x
let suc : !bob int -> !bob int = fun x -> x + 1

val inv_suc : !a int -> !b int * !c int
    with !bob < !c
    and  !alice < !b
    and  !a < !b, !c
    and  !a < !alice, !bob
let inv_suc x = (inv x, suc x)
```

It's a trick question! To give the answer, here is the corresponding typing in our system: we see that the second element of the product has no dependency on `alice`. The primary difference here is that our ordering on dependency sets works fully structurally, so the input must have no dependencies. Specifically, the intersection of `[ alice ]` and `[ bob ]` is `[ ]`. For the same reason as with `print` above, these are practically as general as each other.

```
let inv : [ alice ] int -> [ alice ] int = fun x -> -x
let suc : [ bob ] int -> [ bob ] int = fun x -> x + 1

val inv_suc : [ ] int -> [ alice ] int * [ bob ] int
let inv_suc x = (inv x, suc x)
```

It may be an interesting exercise to have a friend read the second typing first and see if it helps them parse the first faster— we have informally found this to be the case. We would like to investigate this more than just informally, however.

3 Experimenting with Structures: First Steps

In the preceding two examples, we can find researchers creating alternative type systems that they believe better reflect how people want to think about some form of static analysis (alias analysis, IFC). Moreover, these alternatives seem to share something in common— a shift from constraints to structures. This leads us to a more general hypothesis: that structure-style type systems better map onto human reasoning than constraint-style type systems. Our goal is to design experiments which test this hypothesis from several angles.

There are two areas of cognition within which the constraints-structures distinction seems most relevant. The first is perceptual reasoning, which we have hinted at already. We would like the most routine information gathering tasks to impose the lowest possible cognitive overhead, in the ideal case being reduced to making low-latency pattern matching queries about symbols and text within the program. The other is logical reasoning, or broader, holistic thinking done when, for instance, weighing design considerations. These two areas roughly correspond to programming-in-the-small and programming-in-the-large.

Lifetimes in Rust and information flow act as our chosen surrogates here since we have points of comparison on both sides of the constraints-structures boundary. Extending our experiments to other langauge features for which only a constraint-based representation currently exists requires novel work to devise a reasonable structural alternative. That said, if readers of this paper are aware of other instances of the constraints-structures distinction amenable to experimentation, send them our way!

3.1 Natural Typing

Question: When people invent type system designs, do they tend towards constraints or structures?

We begin with a formative qualitative study that explores what typing notations might best match people's mental models of information flow and alias analysis. This is intended primarily to address the logical—not perceptual—axis because natural programming-style studies are known to rarely correspond syntactically to any particular system under consideration or to the ultimate end product.

At a high level, we plan to give each participant a part of a program that has interesting (but undocumented) properties, and talk about the ways it could be misused. Their job is to write type-like annotations that prevent future clients of the program from making mistakes. They can convey any information they wish, outside of modifying the executable code, to future clients by inserting annotations into the program in any notational system they devise.

In the information flow setting, we might give them one or more procedures that handle a secret, where the job of the participant is to insert information that prevents clients from using those procedures in ways that leak the secret. A (very) simple example might include the following declarations:

```
val pass : string = "katya"

fn check(pass_attempt : string) : bool {
    return pass_attempt == pass;
}

fn to_upper(text : string) : string { ... }

fn print(text : string) { ... }
```

We'd talk about what properties this code should have, for example that the password should not be printed. Functions should be documented to help clients use them correctly: for example, `to_upper` returns information about a password if one is passed in. `check` is of course the most interesting because it intentionally leaks information about its input, but in general we do not mind leaking a single bit.

After making sure they understand these properties, we will give participants some time to annotate the code in ways that express the ideas. We will use a thinkaloud protocol so that we can use not just the resulting annotations but also have some idea of what participants were thinking as they wrote them. We will supplement thinkaloud information by having a conversation with participants after the study to learn more about their motivations, to the extent anything is unclear.

We will experiment with a more active protocol where experimenters ask questions about annotations that points out something the participant may have missed, giving them a chance to try to encode it. This risks biasing participants but may result in more meaningful annotations, and more insight into their mental models due to the more complete design.

A point which invites caution: participants may be familiar with information flow policies adopted by large organizations (like security clearances within the US government). Prior experiences here may prime participants to default to lattices when working in the context of secret leakage, because organizational policies often map closely to them. Another potential issue: the open-ended nature of the experiment may mean some participants won't come up with anything resembling an information flow system. To counteract this, we might prompt participants to draw arrows over the program sketch specifying where data is and isn't allowed to go, and develop a syntax by guiding them to generalize patterns from the arrows. We hope this pushes participants to think along paths that will be more relevant to a qualitative analysis of the invented specification systems—flow sources and destinations are a universal idea underlying information flow—without pushing them towards either the constraints or structures sides.

For Rust lifetimes, the code examples would instead be primarily concerned with preventing various memory safety errors. For example, we might give participants the following code:

```
fn store_into(data : * int, data_holder : ** int) {
    *data_holder == data;
}
```

and then ask them to write annotations that would prevent dangling references if the variables pointed to by `data` and `data_holder` are allocated in different stack frames.

Rust is vastly more popular than information flow, or any other system for static memory safety, so we must filter out participants with pre-existing Rust knowledge. The same caveat about the open-ended nature of our experimental design applies, but is perhaps less severe in this case. Software engineers are relatively likely to be personally familiar with bugs resulting from memory unsafety, so have pre-existing programmatic knowledge about the problem domain that may be absent for information flow.

3.2 Synthesizing Types

Question: What is the influence of the constraints-structures distinction on how people induct mental models of type systems and apply them to examples?

In another exploratory qualitative study, we plan to give people guided tours of each of our type systems. The setup here is to give people partially-typed programs under either a constraint-based or structural system and elicit from them types for un-annotated elements of the program, if possible. In the context of information flow, continuing from the `inv_suc` program as given previously, a good initial question to ask may be 'What is the type of this call to `inv_suc`?'

```
let j : !jane int = 5          let j : [ jane ] int = 5
inv_suc j ⚠                    inv_suc j ⚠
```

The call will fail here because `[ jane ]` is not a subset of `[ alice ]` or `[ bob ]` (or equivalently `!jane` $\not\sqsubseteq$ `!alice` or `!bob`). A follow up question could be 'What are types for `inv`, `suc`, and `inv_suc` that would cause that call to succeed?' In the structural system, a possible solution might be:

```
val inv : [ jane ] int -> [ alice jane ] int
val suc : [ jane ] int -> [ bob jane ] int
val inv_suc : [ jane ] int -> [ alice jane ] int * [ bob jane ] int
```

If people's performance on this question differs significantly between the constraint-based and structural variants of the same program, it is a strong indication in favor of one or the other. Importantly, by holding people's hand through the type system and giving them bite-sized tasks along the way, we can observe the nature of the difficulties people are experiencing (or any mistakes they make) during these logical reasoning-heavy tasks. Inspired by the significant successes in linguistics from systematizing mistakes made by language learners [9], [10], we hope to develop similarly rigorous cognitive models for programming languages as have been made for natural languages.

In particular, the end goal of this experiment is to develop *semi-formal* (potentially *unsound*) natural deduction-style presentations of the cognitive machinery backing participants' mental models of each type system. These sorts of models have begun to be explored in pedagogical work on the Rust borrow checker [11], and we believe we can push it further in this setting. An example of a particularly strong result here would be showing that people's semi-formal models of constraint-based systems take on structural characteristics, which would provide evidence for the soundness of structures-over-constraints as a general design principle.

3.3 Program Slicing

Question: How does the constraints-structures divide affect quick information extraction tasks?

The core task here is to give people a program of moderate complexity, replete with type information, and ask them to highlight parts of the program that are relevant to certain type-driven queries.

For instance, in the realm of information flow, we might give people a simple password checker and some client code that uses it. The task is then to, say, highlight all lines of code affected by password data. A slightly more complex task may involve highlighting all the parts of a program needed to compute some expression. In the realm of Rust lifetimes, we could ask people to highlight all lines of code representing the "heap footprint" of some data— all the places that work with references to it. In both cases, our example programs will make heavy use of library functions whose implementation is not available, only their signature. This forces participants to contend with type information rather than attempting to make sense of programs without it. We also plan to set a sub-10-second time limit per task. This is done to ensure that the task is executed under similar constraints to those in a real-world setting, where this kind of information might be be queried in service of higher-level goals.

This experiment focuses heavily on perceptual reasoning: the subtasks mentioned above are routine activities people frequently engage in as part of longer-running problem-solving. An advantage of this experiment is that it's easy to automate, which is convenient because it's the most quantitative of all our proposed studies. That is, we can automatically compute program slices from information flow annotations algorithms or leverage the Rust compiler's alias analysis in order to mechanically determine whether people have given satisfactory answers. We need only provide example programs. A potential concrete implementation for this study is as a continuously-running experiment via a website with some gamification, to entice people to participate.

One potential soundness threat lies in carefully balancing the size and complexity of the programs we ask people to slice. Giving people programs that are too small or simple runs the risk of having them largely ignore type annotations and fall back on their existing programming knowledge to find the right slice. Large programs induce the opposite issue, leading people to flounder with the complexities of such programs that may have little to do with the type system under study. Striking the right balance between simplicity and interpretability for the presented programs will be a key design issue.

3.4 Attack/Defense Game

Question: Which style of type system is more effective *end-to-end* at helping people prevent bugs when writing small programs and at scaling to handle larger programs?

Finally, we have a coarser-grained experiment which addresses the bigger picture: does the design guidance provided by the constraints-structures distinction translate into holistic effectiveness? This final study is set up as a game: there's a *defender* and an *attacker*. The defender's job is to use one of the four type systems at hand. The attacker's job, depending on which of Rust or information flow we're considering, is to either extract a secret from the program or cause it to exhibit memory errors.

The game proceeds in turns, where the defender starts. The defender is given a program with security or memory safety issues and a few minutes to find and fix them. Notably, they are only permitted to run the type checker in a limited capacity, with a maximum number of usages permitted per turn. In wizard-of-oz style, type checking and inference queries are forwarded to a human experiment runner in order to sidestep wasting usages on trivial syntactic errors and enable participants to focus on higher-level issues. This also enables us to iterate on and test a wide range of type system designs inside the game in quick succession, without having to implement each one.

During the attacker's turn, which lasts for the same amount of time, they are permitted to attempt to cause the program to misbehave by running it on a number of example inputs. If the secret is leaked or memory corrupted, the attacker wins that round. If time runs out, the defender wins. The game then progresses to the next round. The defender can fix any failing inputs, and is given new buggy machinery to fix in addition. This exchange continues for 3-5 rounds, and the winner is determined by their success in the majority of them.

One point which deserves further explanation: why the limit on the number of type checking queries? (1) We can tune this limit to give the attacker and defender fair chances at winning the game, which is necessary to keep people engaged in the study. (2) By looking at when defenders are invoking the type checker or inference, we can draw inferences about the difficulty of the reasoning tasks at the point of the query, giving us rich information about the intrinsic difficulties induced across the constraints-structures distinction. (3) Most importantly, it allows us to get a sense of how the constraints-structures distinction plays into *programming-in-the-large*. That is, we suspect it is key to scalable type systems that their cognitive machinery is itself simple to work with. When performing high-level planning and design tasks that characterize programming-in-the-large, it is rare that the type system itself is available to help: often it is the case that we must rely on our fuzzy semi-formal understanding of the language's invariants to put together scaffolding that will eventually turn into a working program. This is felt especially sorely in languages with advanced type systems such as the ones we are considering: often a bad design is not merely difficult to maintain or inefficient, but nearly *impossible* to implement (as can sometimes be seen with zero-copy programming in Rust). We cannot feasibly directly contrast effectiveness between our candidate type systems in a programming-in-the-large setting (not the least of which because there are no large-scale deployments of the majority of them), so we instead attempt to approximate it by emphasizing users' ability to 'run the type checker in their head'. The extent to which this truly generalizes to programming-in-the-large is an open question. We are very interested in lightweight techniques for evaluating the cognitive scalability of type systems (that is, without having to run in-situ longitudinal studies)— if you have any thoughts here, get in touch!

4 Conclusion

We have observed a fundamental divide in type systems design. Correctness properties within the same domain may be expressed in a constraint-based fashion as a system of inequalities, or as an algebraic structure integrated into the core syntax of types. We have illuminated two particular typing constructs that exhibit this distinction, Rust lifetimes and information flow labels. Additionally, we have outlined the beginnings of a collection of studies intended to investigate the cognitive underpinnings of the constraints-structures distinction within both contexts. A central thrust of this work is that our results should ideally produce *general* and *re-usable* insights about type system design that can be readily applied whenever the constraints-structures distinction rears its head in the wild.

More broadly, we wish to pursue a program of building a modular human-centered theory of programming languages, placing human interface considerations firmly beside type theory at the foundations of programming languages. It is the *expectation* that, for instance, formal results from type theory can be modularly slotted in across new language design projects without having to repeat the proof work. Our hope is that, eventually, the same reusability and theory-building successes witnessed in the mathematical analysis of programming languages can be replicated in their analysis as human interfaces.

Finally, we want feedback! If you're not sure whether the experiments proposed here will be sufficiently insightful, or they have sparked ideas for more studies, or you have another instance of the constraints-structures distinction to add to the mix (particularly if it fits the mold of any of our current experiments)— or something else entirely, don't hesitate to reach out!

References

[1] N. D. Matsakis and F. S. Klock, "The rust language," *Ada Lett.*, vol. 34, no. 3, pp. 103–104, Oct. 2014, ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. [Online]. Available: https://doi.org/10.1145/2692956.2663188.

[2] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of computer security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[3] F. Pottier and V. Simonet, "Information Flow Inference for ML," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA: Association for Computing Machinery, 2002, pp. 319–330. DOI: 10.1145/503272.503302.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 147–160.

[5] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[6] N. Matsakis, *The borrow checker within*, https://smallcultfollowing.com/babysteps/blog/2024/06/02/the-borrow-checker-within/, 2024.

[7] H. Gouni, *Lecture 20: Functional and higher-order information flow*, https://15316-cmu.github.io/2024/lectures/20-functional.pdf, 2024.

[8] W. Crichton and S. Krishnamurthi, "Profiling programming language learning," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. DOI: 10.1145/3649812. [Online]. Available: https://doi.org/10.1145/3649812.

[9] D. Lebeaux, "Language acquisition and the form of the grammar," 2000.

[10] K. Demuth, "The acquisition of phonology," *The handbook of phonological theory*, pp. 571–595, 2011.

[11] W. Crichton, G. Gray, and S. Krishnamurthi, "A grounded conceptual model for ownership types in rust," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, Oct. 2023. DOI: 10.1145/3622841. [Online]. Available: https://doi.org/10.1145/3622841.