An Approach for Type Inference of Dynamic Properties

By Sawyer Bergeron

I. SYNOPSIS

This project intends to demonstrate that it is possible to substantially improve the run-time performance of languages where object properties may be used (inserted, modified, read, or zeroed/removed) without being declared within the type declaration for that object's type.

The core novel approach that we propose here is the use of a stepped dual-state inference engine for transitively inferring the types of late-declared fields on objects.

Through a suite of benchmarking tests we show that this approach yields output binaries with comparable performance to contemporary statically-typed languages (between -79.17% regression and 28.13% improvement), and between 70% and 119,893.95% run-time performance improvement over contemporary dynamically-typed languages. The treated case demonstrates a maximum 405,460% improvement over the control case.

II. INTRODUCTION

A common trait in multiple well-known dynamically-typed languages (such as Python, JavaScript, and Lua) is dynamic property addition. This feature allows for fields, methods, and other functionality to be inserted into objects without requiring that the programmer pre-declare any such properties in any schema or class definition for that object. This ad-hoc approach to defining object behavior is commonly cited as a productivity enhancement for rapid experimentation and prototyping [1] and can allow for user-extensible data-flow through "upstream" code in ways that original creators may not have planned for or supported. Unfortunately, this flexibility (among others) is often considered to be a core reason why such languages are often slower to execute semantically similar operations at run-time.

In this paper, we propose and implement an approach whereby typical usage of these dynamic properties can, at compile time, be transformed into fully static properties (fields) of objects with no run-time costs related to field associativity or type validation. A three program benchmark suite is used to evaluate Luma's performance relative to Python, JavaScript, Rust, and a baseline Control deoptimization of Luma. Success for our implementation is determined by whether a statistically significant overall decrease in per-test running time is observed for Luma

relative to the Control and the two other included languages with dynamic property support (JavaScript and Python).

III. BACKGROUND

Ideally, every prototype or proof-of-concept project would have the time and resources to be rewritten using ever more robust frameworks and languages for optimal efficiencies and performance. Realistically, there are trade-offs -- temporal, monetary, and strategic constraints -- and projects are often finalized in the same language they were prototyped in, despite the potential for improvements and efficiency gains. Dynamically-typed languages (e.g. JavaScript, Python) are easy to use but yield inefficiencies and bugs down the road; whereas, statically-typed languages (e.g. Rust, C) improve performance but enjoy limited use for prototyping due to the conceptual and temporal overhead of up-front annotation requirements.

ML-style type inference [2] represented one major step toward reducing the cognitive and boilerplate overhead of statically-typed languages. Such systems allow the programmer to, in most if not all cases, omit type annotations for variables within code. Unfortunately, variable shape must still be specified in such systems. Row typing can solve similar (but ultimately orthogonal) problems, but does not allow for implicit composition and aggregation of disparately shaped objects in the ways that dynamically-typed languages often do.

This project seeks to explore whether we can deliver the best of both worlds: code that is both syntactically and semantically similar to leading dynamically-typed languages and that connotes the performance advantages of full static typing and fully monomorphic run-time code.

A familiarity with industry-standard approaches to lowering type representations for object-oriented languages may be necessary for an intuitive understanding of the approaches used, although every effort will be made to approach the subject matter from first principles. Readers who have prior experience with contemporary dynamically-typed, object-oriented languages such as Python, JavaScript, Lua, Julia, or Ruby, where dynamic properties are a commonly used and understood feature, will feel most comfortable with the programming paradigms discussed here.

Limited background is expected for topics related to type inference, as significant effort is put into a first-principles exploration of those topics. For a more in-depth understanding of the theory that forms the foundations for type inference as used here, the original work by Robin Milner [2] is very helpful for forming an intuitive understanding.

Literature evaluating analogous approaches to those to language design and static analysis tools is itself decidedly sparse. Many approaches attempt to attain performance improvements through improved analysis of existing languages. As showcased in [3], some approaches use substantially more powerful SAT solvers to attempt to compute expression types. Still others [4]

approach existing languages with partial solutions that are effective given type assumptions, but must establish type-checking barriers with fallback runtimes for when any such assertions are violated. A more directly comparable approach arises from row polymorphism as applied to ML family languages. This approach is highlighted in [5]. These approaches are contrasted in section VIII.

IV. MOTIVATION

Contemporary dynamically-typed programming languages, such as JavaScript or Python, are often significantly slower to execute semantically similar code than contemporary "statically-typed" languages such as C, Java, or Rust when comparing their reference compilers/interpreters/runtimes. This is particularly true for dynamic languages when the set of properties initialized on objects changes significantly and frequently at run-time, as this behavior can impair the efficacy of techniques such as type specialization [6], which are used by many JIT compilers [7] such as Ion (an optimizing compiler used by SpiderMonkey, Firefox's JS JIT), and TurboFan (an optimizing compiler used by V8, Chrome's JS JIT), and which are ripe for inclusion in others such as CPython [8]. Despite their productivity advantages during prototyping stages, performance limitations of these languages can have far reaching consequences for Quality-of-Service (QoS) metrics such as latency, energy consumption on edge/client machines, hardware and energy requirements for service providers, and on the environment insofar as such impacts affect resource consumption (power generation, precious metal mining/refinement, and manufacturing). Effective, performant alternatives to contemporary dynamically-typed programming languages can directly mitigate these shortcomings.

V. APPROACH

Luma uses existing approaches to type inference to determine the types of expressions and bindings local to each callable context (each roughly mapping to a function, procedure, or similar construct). This inference uses type annotations that are syntactically required in function signatures, as well as other constraints provided by literals, literal constructors, annotated field accesses, and function invocations to constrain the types of said expressions and bindings. In any case where a field is accessed on a type that does not directly describe its type, inference dependent on that field is paused and set aside until such time that the field's type is resolved and committed by later mechanisms. At any point that a fully confining constraint is established for such a field, that constraint is sent to an actor for the involved type, and is stored for later. If at any point all inferencing actors have no work to do that is not dependent on the type of such a field being known, an algorithm is used to select a single such field from the set of all such fields to commit to a type. The provided constraints for that field are combined, a unified type is constructed from them (and any type errors are emitted), and the selected type is broadcast to all inferencing actors that are pending resolution of that field's type. Inference then continues from that point until cycle repeat or until a stall/complete state is reached.

Explained in more detail, our implementation capitalizes on a very specific generalization and assumption around how programmers treat objects. This assumption is that the total set of statically-undeclared properties U that are inserted into an object instantiated from prototype P is small, and that there are very rarely any two sets $S \subseteq U$, $T \subseteq U$ where S is the set of properties on object P'_1 and T is the set of properties on object P'_2 such that $S \cap T$ is small while S and T individually are of significant size in relation to U. Less formally, it is rare that programmers will use objects of a given original type in wildly different ways, with wildly different dynamically-inserted fields. The second assumption is that when dynamic fields are used on a given base type, they are populated a substantial proportion of the time. That is, if space is reserved for a given dynamic field on every instance of a base type, then that space will be populated a substantial proportion of the time.

If we assume these two properties, and can obtain sufficient type annotation from the programmer, we can do two transformations. The first transformation is to reserve space for every dynamic field that is used (read from, assigned into) for a given base type. The second transformation is to take the type(s) of values that are inserted into/read from those fields and only emit code supporting those type(s).

For this paper, a reduced version of the second transformation is implemented, such that only one type may be used per individual dynamic field on any given base type.

These transformations alleviate two sources of performance loss on contemporary dynamically-typed languages. The first loss is that fields must be stored in associative maps per-object. Less formally, each object is comprised of potentially some metadata as well as a hashmap (or similar structure) that associates field names with their value. The second loss is that after loading from such a structure, field types must be sufficiently determined for their value to be useful. Either the operations available to apply to the value must be determined ("what does the '+' operator on a value of shape `('123', { '-': fn(_, _) -> _ @ 0x456, '+': fn(_, _) -> _ @ 0x478 })` do?") or the actual type of the value must be determined and the functionality determined based on that. The first approach is slow on account of the fact that it implies another lookup in an associative mapping; the second approach is slow on account of the fact that the total number of types in a program is often large, and specialized code would need to be emitted for each such type, and/or another associative structure would be required to map types to what code path should be jumped to.

Instead, the two assumptions above allow reducing each dynamic field to the equivalent of a `Maybe t`/`Option<T>` or nullable pointer, and then as the type of each field is inferred to a set of one, only a single specialization/monomorphization of the code depending on the value needs to be emitted. Less formally, since the type of the value in the field is known at compile time, the generated code does not need to account for it being "any" type; it only needs to account for the one type that the value could possibly be. This reduces code size (and is more friendly to instruction cache), and it reduces branching based on type data.

Future iterations of this approach may consider using “superpositional type variables”. Such an evolution would relax the restriction that only one type may be inserted into a given dynamic field and instead allow multiple types to be inserted. The resulting output representation of the field would then be a tagged union of the possible types, with a comparatively cheap branch dependent on the tag value. Performance benefits of this approach form the basis for libraries such as `enum-dispatch` for transforming dynamic dispatch into static dispatch in specific cases. This results in objects with layouts and semantics very similar to those seen in typical statically-typed languages such as C or C++. This also allows the same kinds of optimization passes that are used for those languages to be applied here, such that inferred properties represent a zero cost abstraction over nullable (checked) fields.

Now that we’ve seen what optimizations these assumptions and transformations allow, we need to survey what transformations and analyses need to be performed in order to not only determine the set of dynamic fields that may exist for a base type, but to statically (at compile time) determine their exact type.

Parsing

An increasingly expected feature in modern programming language tooling is the ability to produce more than one (valid) error at a time and to continue to provide diagnostics even once an error has occurred. Here, we use a parser combinator library developed for this project to accomplish resilient parsing even when the source inputs contain numerous syntax errors.

Our method is primarily modeled after contemporary recursive-descent parsers. One parser instance is created for each input file. When a rule is chosen for parsing based on lookahead, the function matching the parse rule stores a frame into a stack corresponding to the parser instance. This frame contains a vector of tokens that may be consumed by this rule. The tokens are ordered within the vector in the order they are most likely to be consumed. Each token is mapped to a numeric weight, with the weight indicating roughly how specific that token is to the current rule. For example, the `struct` token is quite closely related to the struct definition rule, and so has a much higher weight than is assigned for the `}` token elsewhere, as the `}` token is consumed by many different rules, so encountering one where it is not immediately expected is not a strong indication that any particular rule should be bailed out to.

When parsing a file without syntax errors, this metadata is unused, and is designed to be computationally cheap to maintain. When a syntax error is encountered, however, an algorithm is invoked that has access to threshold-bounded lookahead from the current location in the file, as well as the metadata constructed up to that point during parsing. The given algorithm then performs a spatial search for the lowest cost solution according to an internal ranking equation. The chosen solution is selected by optimizing the following metrics:

1. Token drop count -- a solution has a higher cost if it discards more of the input stream than another otherwise similar solution, as the user is unlikely to make long strings of syntax errors consecutively.

2. Locality of the matching solution rule -- a solution has a higher cost if a greater number of parsing rules must be escaped, as syntax errors are typically local to a construct, and it is rare that truly arbitrary segments of code are omitted.
3. Weight of the matching token -- a token that is less ambiguously matched to the expecting rule is a stronger candidate, as more specific tokens are often more intentionally inserted than more ambiguous tokens.
4. Distance to the matched token within the matched rule -- a solution with a smaller generated completion is preferred, as syntax errors themselves are expected to be small omissions more often than significant, structural omissions.

Once a solution is selected, all calls into the combinator library to extract tokens by any rule more local than the matched rule return a bubbling (similar to unwinding upon exception throw) token mismatch error. The input stream realignment handle and associated thrown syntax errors themselves contain a reference to the solution, such that when the rule that was selected to handle the syntax error is reached, it is realigned to the input stream and provided with the expected realigning token and input stream location. From this point, parsing continues as normal until the next syntax error is encountered.

Node Structure and Service Architecture

Our reference compiler is designed to facilitate intra-compilation-unit compiler parallelism. To this end, once parsing is completed (itself a file-level parallel process) every symbol within the constructed file/parse tree is assigned a node in a symbol tree. These nodes each correspond to a function, type definition, or modules/namespace within the compilation unit. Nodes are aware of their parent, the root of the tree, and any child nodes.

Once this tree is constructed, all following compilation steps are done using a service architecture. For every node, a series of services are started. These services include a name resolution service (Resolver), a local type inference service (Quark), a dynamic property type inference service (Transponster), a code generation service (Scribe), and a router service (Router). All services for any given node run within a single-threaded async executor, itself acting as an async task within a multi-threaded async executor. Services within a node can communicate over a low-overhead local interconnect, filtering through the Router service for the node. If a message is outbound (sent to a destination on another node) the Router service bridges the message out from the local message bus to a higher-overhead node-to-node message bus that allows cross-thread communication. All communication between nodes for name resolution, type inference, and monomorphization instructions occur over this message bus. This allows services to consume zero compute resources when they are not actively processing or woken by a message. Services have access to the executor that they are spawned within, and are themselves able to spawn new async tasks within the local executor. This forms the foundation on which the Hindley Milner-style type inference algorithm within Quark is built, as type unification can itself use async-await to trigger unifications and other rules that depend on prior unifications discovering new information.

Middle Intermediate Representation

After the parallel AST is built, a rewrite transformation is performed for every node corresponding to a function or callable. This transformation produces a representation termed the “Middle Intermediate Representation” (MIR), which contains a reduced set of primitives compared to the full syntax tree. This representation combines the representations for all types of loop, the representations for all branching constructs are combined, and rewrite rules are performed to transform method calls into field-access/function-invocation chains. This MIR is designed to be more easily reasoned about when performing type inference and reachability analysis, and is ultimately the input to the code generation phase after various transformations have been performed and type/reachability annotations have been added.

Local Type Inference

Type inference itself is far from a new or novel technique. For the purposes of this project, type inference is best understood as a mechanism by which the types of variables, expressions, and arguments (though not parameters) may be programmatically determined through applying a set of rules and a process of logical induction. A very popular method of performing type inference is through the use of the Hindley Milner (HM) type system and associated inference rules. Intuitively, HM uses the idea that types can be propagated through a program, and if we know that for two knowns A, B, and two unknowns C, D, that if some type (A, C) is equivalent to some other type (D, B), then A must be the same type as D, and C must be the same type as B. What follows this is that every instance of D may be replaced with A, and every instance of C may be replaced by B. As an applied example, assume we have two people “John” and “Sally”. John is currently riding in a red car. Sally is currently riding in a sedan. Both John and Sally are riding in the same car. By extension, John and Sally are riding in a red sedan. We can also infer from this that Sally is riding in a red car, and that John is riding in a sedan. Through this process, the types of most expressions can be inferred. In an analogous language without dynamic fields, this system is sufficient to infer the types of all expressions in the program.

Dynamic Property Type Inference

For intra-function type inference, Luma uses a variation of Hindley Milner type inference. Type variables are unified when they are explicitly set as equivalent (through let binding, function calls, or field/static property accesses) and when they appear at identical indices in parametric types that have been unified by application of the above properties. Luma does not support typeclasses or higher kinded types at this time. Luma requires type annotations for function parameter types and return type. This restriction not only allows for type inference to be done in parallel across the full set of functions in a given compilation unit, but it allows for property inference to know the complete base type that a property is on in more cases than it would otherwise.

Our approach to property inference, however, is a new approach introduced with this paper. Luma uses a binary-metastate converging system in order to resolve property types. This algorithm follows the following process:

1. A Hindley Milner-derived type inference solver is started for each function, lamda, and closure (every “callable” kind) that is defined within the compilation unit. These solvers follow the algorithm described in the prior paragraph. Within Luma, each of these solvers corresponds to an instance of the `Quark` struct.
2. An analogous service called `Transponster` is started for every module node of the “type” kind within the compilation unit. This service is initially dormant and is woken later.
3. Each solver instance steps the aforementioned type inference algorithm forward using the constraints provided by function parameter types, function return type, the known type of literals (integers, floating point, string literals), the type of structural data types that are constructed using literal syntax (`struct Foo { bar: 1, baz: "2" }`), and any additional type annotations or function calls that occur within the code.
4. Whenever a solver comes across a property access on a type where the base type is known (whether or not any type parameters are known), a query is sent out over the inter-module network to the module corresponding to the base type.
    a. If the target module has a static property (field or method) corresponding to the requested property key, that module returns a type variable defining the type of that property value in terms of type parameters to the base type and absolute references to other modules for any other types. As an example, if the base type of the original access is defined as such:

    ```
    struct S<T> {
            var f: Either<i32, T>
    }
    ```

    And there exists an instance `s` of `S`, and a field access of `s.f` occurs, then the response will state that the type of `f` is an absolute reference to the module node for `Either`, parameterized by an absolute reference to the module node for `i32` and the original type variable assigned to generic `T` that parameterizes `s`.
    b. If the target module does not have a static property mapping to the provided property key, then a different kind of type variable is instantiated called a `DynPropInstance`. If the value arising from the instance is itself unified with any type constraints, then those constraints are unified into the `DynPropInstance`. No constraints or information can be taken *out* of a `DynPropInstance` typevar, and it can not be used to transitively unify types yet, but any information about what type the `DynPropInstance` should be is sent along to the module that it is instantiated on. The later usage of this kind of typevar is discussed below.
5. Once all *available* type unifications have been performed, `Quark` will eventually “stall”. In a typical implementation of Hindley Milner, this state directly maps across to a program for which some type variables are not constrained, whether directly (on introduction) or transitively (through unification with other typevars). Once `Quark` reaches a stalled *or* solved state, it checks in with a global semaphore for the compilation unit.
6. Once all instances of `Quark` have reached this barrier, the compiler switches from the “local type inference” metastate to a temporary “dynamic property inference” metastate. This phase change sends a notification to all instances of the `Transponster` service.

From here, every instance of Transponster looks at all instances of DynPropInstance that have been opened against it.

7. For each instance of DynPropInstance, Transponster categorizes them into one of two categories:
    a. For instances where the entire parameterized type of the property has been constrained by Quark applying unification rules, the instance is assigned the "direct" category.
    b. For instances where only part of the parameterized type (or none at all) is known, the instance is assigned the "indirect" category.
8. Then, for all dynamic *properties* that instances of DynPropInstance have mentioned, the property is assigned a "commit score".
    a. If the property has *at least one* "direct" usage and has no "indirect" usages, then it is assigned a score of positive infinity.
    b. If the property has no direct usages, it is assigned a score of negative infinity.
    c. If the property has *both* direct and indirect usages, the score is calculated as the number of direct usages associated with the most common resolved type minus the total number of indirect usages plus the number of all other direct usages of other types. Intuitively, this is trying to do signal amplification and noise reduction. If the type of direct usages disagree, the program must already have type unification errors, so any further steps are to provide as helpful errors as possible. If the type of a property is very uncertain, we should defer committing to any particular type until other "higher signal strength" properties (potentially on completely different nodes!) have been committed first to avoid injecting more noise into the system. We also do not want nodes with many indirect usages and very few direct usages to be committed early, as that can easily fall into the same pitfall as with disagreeing direct usages (indirect usages are "weaker" forms of noise than disagreeing direct usages, but both are fundamentally forms of uncertainty).
9. For all properties with a score of positive infinity, the property's type is "committed". This fuses the type of the property to a definite value. If no such properties existed, then an election is run whereby the highest scored property across the entire compilation unit (tie-broken alphabetically) is committed in the same fashion. For every usage that existed on the property, the corresponding DynPropInstance is notified of the new, committed type of the property. That DynPropInstance then unblocks any unifications depending on the value, and raises type errors for any direct usages that disagree with the committed type. This allows any Quark waiting on the instance to proceed with downstream unifications that depend on any indirects.
10. Once all commits for this round are complete, all Transponsters communicate with a global barrier in the same fashion as when the first metastate transition occurred. This causes the system to go from the "dynamic property inference" metastate back to the "local type inference" metastate, and another round is started. The system goes back to step 3 of the algorithm, and continues with local Hindley Milner style type inference.
11. The system terminates if, for any round, the following conditions are met:
    a. No type unifications are performed by any Quark,

AND

b. No dynamic properties are committed by any Transponster.

If these conditions are met during any individual round, then the system has stalled, so the system exits.

Once the algorithm has completed, if there are any dynamic properties in any Transponster that are not in the committed state, or if there are any typevars in any Quark that are unconstrained, a type error is raised as the system does not have any decidable type for the input program.

If no type errors have been raised by any prior rule, then the program has been typechecked and all variables and expressions are of known type.

Intuitively, this program must always terminate. This comes from the following factors:

a. HM in this form is decidable and terminating without accounting for dynamic properties,
b. The number of expressions in the program must be finite, as the length of any program is itself finite,
c. Every time the system commits a type for a dynamic property, at least one expression goes from unknown type to known type,
d. HM in this form is responsible for deciding the type of every expression that is not a dynamic property access, so if we assume that the type of every dynamic property is eventually known, by (a) HM must terminate.
e. No dynamic property may ever un-commit its type, and
f. If no progress is made on any round by Quark nor by Transponster, the program terminates.

Output Code Generation

Once the typechecking phase of compilation has terminated, the type of every expression is precisely known. At this stage, all monomorphizations of functions can also be accounted for through a pass over the MIR for the full program (by nodes in parallel) to evaluate all function calls and their supplied generics. All monomorphizations that must be emitted are reported to the Scribe instance for that function, and each `Scribe` may transitively trigger monomorphizations from other `Scribe`s through the same mechanism.

Now, for every node where there exists a non-dormant Scribe, that instance emits any monomorphization of the MIR for that node using type information generated locally by Quark. This monomorphization takes the form of textual Rust code, with function names being generated by a deterministic munging process based on hashing node IDs for the function node and the nodes for all parameter and return types. All function calls use the same munging process to determine the monomorphic function that they are to call.

The output code uses different strategies for each of the three treatment cases to evaluate the effect on performance of type inference on dynamic properties. Up until this point in compilation, all treated cases are compiled in the same manner.

VII. FUTURE ABSTRACTIONS AND IMPROVEMENTS

The core mechanism present in Luma for property inference and other traits of the reference compiler serve as fundamental primitives on which several later abstractions may build. Within this section, we enumerate a subset of those abstractions and how they are enabled by property inference.

Dynamic Trait Installation

This abstraction formed the basis for Luma's original inception. In this abstraction, the values behind dynamic fields are not directly manipulated. Rather, they act as groupings of co-initialized fields and methods. This allows for "upgrading" an object `b` by inserting an implementation of some typeclass `C` at run-time. Such an object can then be passed around as the original base type, and then doing a checked downcast (effectively a null-checked dynamic field access) of `b` to typeclass `C`, where any access of functionality guaranteed by `C` can be done without any existential checks, mitigating much of what cost inferred properties still have (nullability and null-checks), even though it does not directly allow devirtualization.

Devirtualization of Callables/Const Values in Inferred Fields

Since all write locations for inferred fields are known at compile time, if every write location is of the same value, then two optimizations can be performed that are unavailable for languages with dynamic fields. The first optimization is that since the value is known, calls to functions that meet these criteria can be devirtualized. This means that the function pointer can be directly emitted into the output IR for the compiler. This is on its own more efficient for CPUs to execute (branch prediction and instruction prefetch can look "through" the reference and do not have to wait for or speculate on the value of a load from the field), but it also enables additional compiler optimizations to be significantly more helpful, such as inlining. These optimizations by extension allow for dead code elimination, instruction fusion, reachability analysis, and various peephole optimizations that collectively substantially improve the quality of the output binary. This also means that the representation of the dynamic field itself can be made smaller, as the only data that must be stored is whether the property is populated or not. This representation can be made as small as a single bit in a bitfield on the object in certain cases.

Better Incremental Compilation for Non-Optimized Builds

The reference Luma compiler is built around the concept of a communication "network" between nodes, with nodes each mapping to an individual namespace, function, or type definition. This network is used for all cross-module communication after the initial parse stage. If messages and responses are memoized, and local nodes can hash their prior states, nodes can announce any changes by propagating rebuild-required notifications to nodes that they answered before about changed state. This can trigger other nodes to send similar messages, and can build a footprint within the tree of what nodes need to rebuild. Space for additional future fields can be pre-reserved in monomorphized type schemas, such that regions of the compilation unit not interacting with any additional fields would not require a rebuild and would implicitly copy any

such additional values as a byproduct of working with any pre-reserved padding as opaque (but necessary) metadata.

Distributed Compilation Support

Since every node communicates using a common wire protocol, compilation of Luma can be performed in a distributed manner with no algorithmic changes. As type inference is largely performed local to each node (with cross-node communication only for non-cached name resolution, typevar instantiation, and dynamic property resolution) parallelism can be achieved down to individual function level in most cases. Clustering based on file/library boundaries may be performed to capitalize on tendencies for programs to be most sparsely linked at those boundaries compared to arbitrary intra-library and intra-file cuts. This can enable the compiler to be scaled across multiple machines to compile large programs more quickly for Continuous Integration (CI), shorter write-compile-test loops, and for lower latency in providing in-editor type-checking feedback for developers.

Dynamic Properties using Const-Computable Indexing

A common related paradigm in languages such as Python or JavaScript is the use of facilities such as `setattr()` and indexing of the `__attributes__` property. Under the proposed system, emulation of these capabilities is possible. This would require support for evaluation of const-computable values. Such values could be used as a selector or “tag” across the underlying dynamic property system for a given base type. This emulation would allow supporting use cases such as JSON parsing where property keys can be statically computed (static strings, simple arithmetic, build-time injection, statically-computable concatenated strings), and can be used to enforce schemas or discard unread data. Fallback approaches may be implemented for cases with non-const-computable property keys where simple associative maps may be used in conjunction with global type IDs to downcast extracted values. This presents a comparatively ergonomic encapsulation of the “perfect hashing” concept present in other systems, where key and value types may themselves be fully heterogeneous.

Cooperative Dynamic Field Inference

With the current design, type parameterization must be performed by individual solver nodes. Although parametric types may be used for inferred types of object properties, partial solutions for dynamic properties are not combined to improve inference power. Future iterations may introduce approaches to combining `Instance`s from multiple Directs or partial Directs to combine non-overlapping type parameters and propagate these partial (or in combination, complete) solutions back into the referencing `Quark` instances.

VIII. DISCUSSION

Comparison with Existing Approaches

One alternative approach that solves an adjacent problem is row polymorphism. Row polymorphism allows, using specific primitives and operations, objects to be treated similarly to an associative mapping from field identifiers to values. Such an approach, although syntactically dissimilar from common type annotation approaches in languages such as Python and JavaScript, can allow individual objects (and collections of homogeneously shaped objects) to be passed around and used in similar fashion to what this approach allows. An inherent limitation of fully type-erased row-polymorphic mechanisms, however, is that rediscovery of fields after heterogeneously shaped objects are grouped is largely impossible. The mechanisms that allow for type inference on row-polymorphic languages mean that not only is it impossible to do such field rediscovery, but that any grouping of heterogeneously shaped objects acts as a barrier for further type inference on fields not common (and assured existing by the type system) to all objects within such a grouping or collection. For example, the following program is impossible to typecheck using pure row polymorphism or derived approaches:

```
struct Foo {}

fn bar() -> ({ a: int, ... } + Foo) {
  let inst = Foo {};
  { a: 42, ..inst }
}

fn baz(p: { a: int, ... }) -> () {
   print(p.a + 1)
}

fn main() -> () {
   let a = [Foo {}, bar()]; # this is of type [Foo],
                            # not [{ a: int } + Foo]
   baz(a[1]) # this behavior is dynamically sound,
             # but not easily statically provable, as not
             # all items in `a` are { a: int } + Foo
}
```

While such a program is sound in dynamically-typed languages (when trivially translated to JavaScript or Python), aggregating heterogeneously shaped objects in an array here coerces them to some homogeneous type that is, at best, the most general type for all contained objects. Because on its own, row polymorphism does not allow for property/field rediscovery--it is primarily an approach for type-erased languages, and does not trivially expand to allow inference to cross these grouping barriers.

Limitations of Approach

Most limitations arise because of the stepped approach to field inference where a unified SMT based solver may be able to produce more desirable results. The first limitation introduced by this approach is that inferred fields are of a type and existence that can not be parameterized by the parameters of the parent type.

Make this a new section: A Proof Sketch on Eventual Convergence

This system can be intuitively shown to converge within finite, polynomial time.

Consider that Hindley Milner type inference may itself be often computed in O(n) time, but has a guaranteed worst case within EXPTIME [**cite]**. This is derived from the algorithm itself taking time proportional to the number of type unifications that occur across a program. Each type unification is of a complexity (and, consequently, time taken) proportional to the recursive sum of the number of parameters within the type variable on each side.

Unification of two individual non-parameterized type variables under this system is known to take Log(n) time due to the utilized tree-based, replacing, finite set implementation.

According to the original rules set forth by [Milner], unification order is not a consequential factor, as set iteration is performed in the proposed algorithms and does not have any defined order. Thus, a reordering of type variable unifications may not alter the worst case running time of the algorithm--the worst case must already account for the least efficient ordering of unifications.

By these tenets, a program not utilizing inferred fields must terminate within EXPTIME.

For any individual usage of an inferred field, a unification is scheduled dependent on resolution of the field's type with identical semantics to if the field were a typical record field.

No inferred field usage is revisited by local inference engines between initial visit and provided resolution. In addition, no unifications outside of the held unification are delayed by the held unification (all unifications occur eagerly unless they act directly on an inferred field).
[**include reasoning that we can extend this to reasoning as global inference, simply multiplying HM time by N**]

IX. TREATED CASES/METHODOLOGY

The compiler for this project will produce three primary output variants, each with differing levels of optimization beyond a baseline.

In the first variant, field inference is fully utilized, fields occupy discrete slots in objects, and function calls are inlineable by the optimizer applied to the output code. This case is expected to perform best of the Luma-derived variants.

In the second variant (conceptually the “control” treatment or baseline), fields do not occupy distinct slots and are instead stored as entries in a per-object hash table. Type inference metadata for dynamic fields is ignored, and all method calls and field accesses on these objects are fully dynamic and type-checked. This seeks to emulate the intuitively trivial implementation of a dynamically-typed language, and it is expected to provide significantly reduced performance over the first variant, as not only must every method call and field access index into the object’s hash table, but all objects must be passed by owning reference, and platform-native types (such as numeric types) must be boxed/wrapped and use either type-id matching or virtual calls to functions that do very simple intrinsic operations on their operands. This is expected to be slower than the compared languages and runtimes, as almost all contemporary dynamic programming languages have at least some distinction between reference types and primitive types that allows for faster operations on primitive types.

The additional cases are comprised of semantically similar programs written in Rust, JavaScript, and Python. Each program is written using standard practices for their respective languages and aim to present points of comparison with contemporary languages. Rust is chosen to represent a typical statically compiled language, and uses the LLVM framework for middle- and back-end optimization and code generation. LLVM underpins the Clang C and C++ compilers, as well as many other statically typed languages. Thus, Rust should provide a good point of reference for the performance that can be expected from contemporary statically-typed languages. JavaScript, executed within the NodeJS runtime (based on the V8 Javascript runtime) is presented as a comparatively highly performant, dynamically-typed, language. V8 performs state-of-the-art transformations and dynamic analyses to interpret, compile, and re-compile JavaScript at run-time to extract performance that is directly comparable to contemporary statically-typed, ahead-of-time compiled languages. Finally, Python (here executed within the reference CPython runtime and interpreter/pseudo-JIT) is selected as a largely *interpreted* dynamically-typed language. Although CPython does perform lowering to bytecode/Intermediate Representation (IR), it does not perform object shape specialization as seen in V8, and does not perform native code JIT compilation. It is selected for comparison both on account of its relative popularity, and as a representation of a relatively optimized typical interpreter for a dynamically-typed language.

The success of the treatment will be determined by whether inlining dynamic fields (as done to varying levels in the first and second variants) imparts a statistically significant improvement in program run-time performance. This improvement is determined by the percentage decrease in total execution time for individual test programs from the associated test battery, as programs that execute in less time allow more work to be done within a given period in the same execution environment by improving how many programs can be run in a given time period.

X. TEST INPUTS

Nbody Bench

This test input is designed to exercise the arithmetic and by-value semantics performance of the target languages. Where possible, objects are passed by-value rather than by-reference or by owning pointer, so this represents a prime target for hot-loop optimization where the Luma LLVM backend can do aggressive function inlining, instruction fusion, loop unrolling, and constant folding. JIT'd languages such as JavaScript should fare well here, as well, once the optimizing compiler takes over, but this will likely widen the gap with languages such as Python (under CPython) unable to take such optimizing steps.

LinkedList Bench

This test exercises pointer and reference counting semantics. Without major inferential leaps (that from analysis do not happen), this test reflects heap performance and pointer-target-type specialization. Pointer-target-type specialization represents how well the language implementation can optimize treatment of pointers based on the type they point to. In implementations such as CPython, no such specialization occurs so every pointer must be dereferenced and the object it points to analyzed to determine its type before any operation can be done to it. V8 may assign a shape to the object and operate on some (checked) assumptions of what is behind the pointer. Luma, because it does full type erasure, allows the optimizer to elide any checks for what type is behind a pointer, and only do a null check before dereferencing the type. JavaScript has an advantage in this case because of lower amortized cost to track object lifetimes as compared to reference counting as used in Luma.

Trie Bench

This test input is designed to test string manipulation performance. As Luma does not currently support by-ref slicing, this represents a good point of comparison with similar languages such as JavaScript and Python where more advanced approaches to owning references may allow straightforward string operations to be transformed into more performant substitutes than other "systems level" programming languages (such as Rust, C, or C++) may allow or feasibly automatically perform.

XI. EXPERIMENTAL EVALUATION AND RESULTS

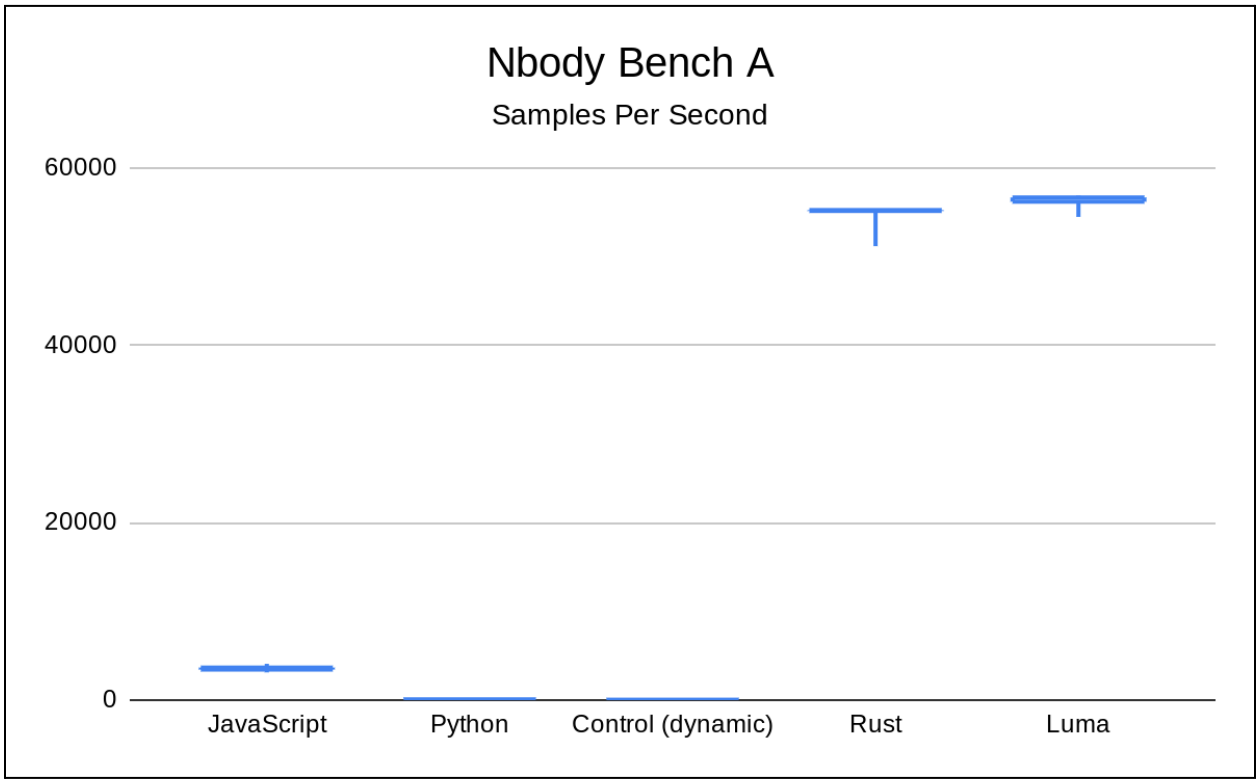
Each section is presented as two charts. The first reflects the projected performance in test operation-units per second, and the second shows the measured metric of time taken to complete a single run of the given test. For each language within each chart, there is an arm for the lowest and highest measurements out of 100 runs, and a box that spans the 25th percentile through 75th percentile (first through third quartile).

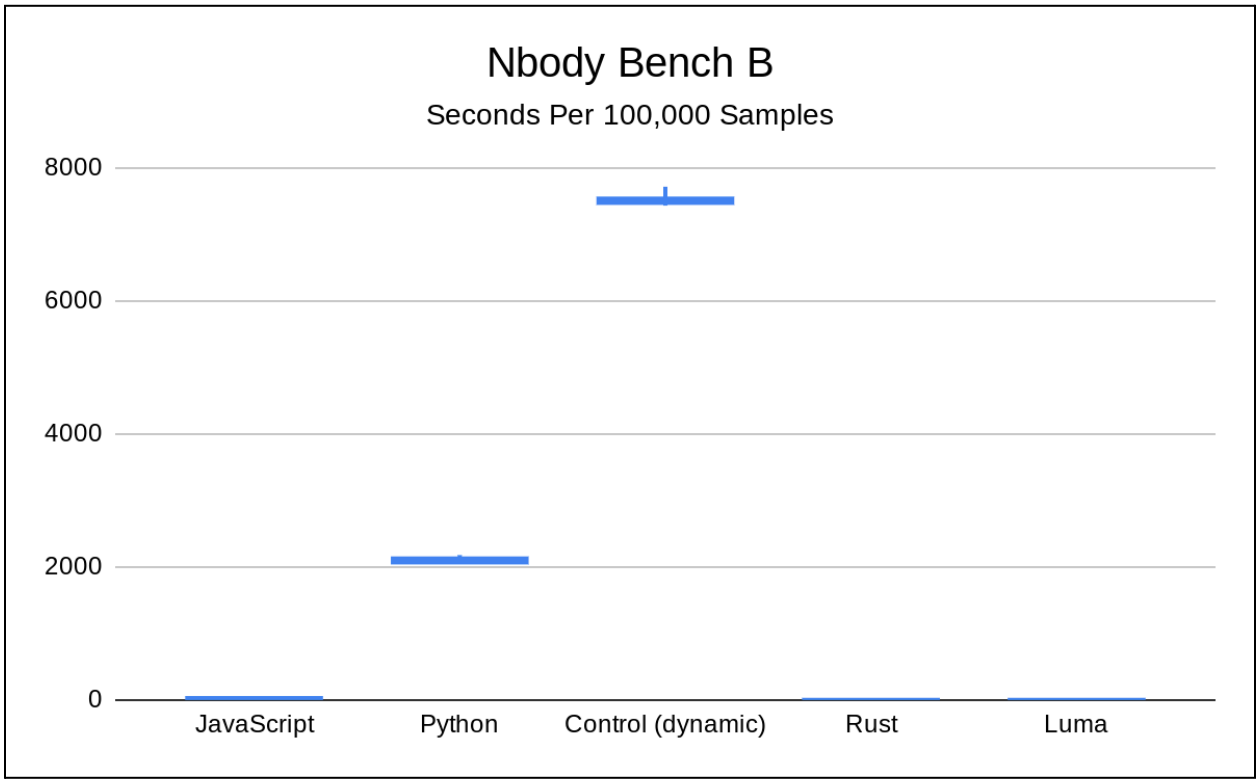
All tests were performed on an Apple M1 Pro CPU with default scheduling. All tests were observed to be scheduled on the P-cores of the CPU for every run. Tests were conducted using MacOS Ventura 13.2.1. Each Rust test was performed using artifacts compiled by

rustc-1.70.0-nightly, with `lto="fat"` set, `-C target-cpu=native` enabled, and `--release` mode active. Debug symbols were disabled. Each Python test was performed using CPython 3.11.3.

Nbody Bench

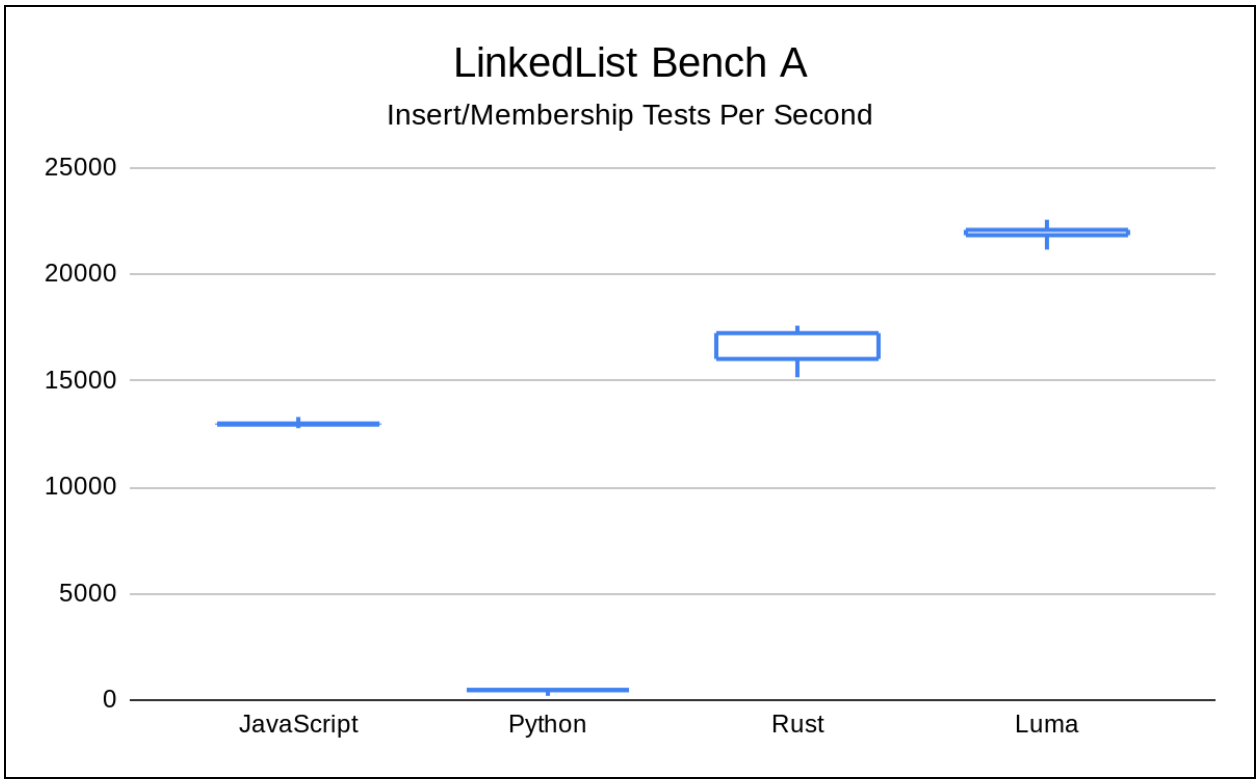
Nbody is presented as a product of samples per second. A sample in this context is a single time stepping of the simulation, with a round of velocity and position updates for every object within the simulation. Here, we observe a 2.57% mean improvement over Rust, a 1448.68% mean improvement over JavaScript, a 119,893.95% mean improvement over Python, and a 425087.97% improvement over the Control variant.

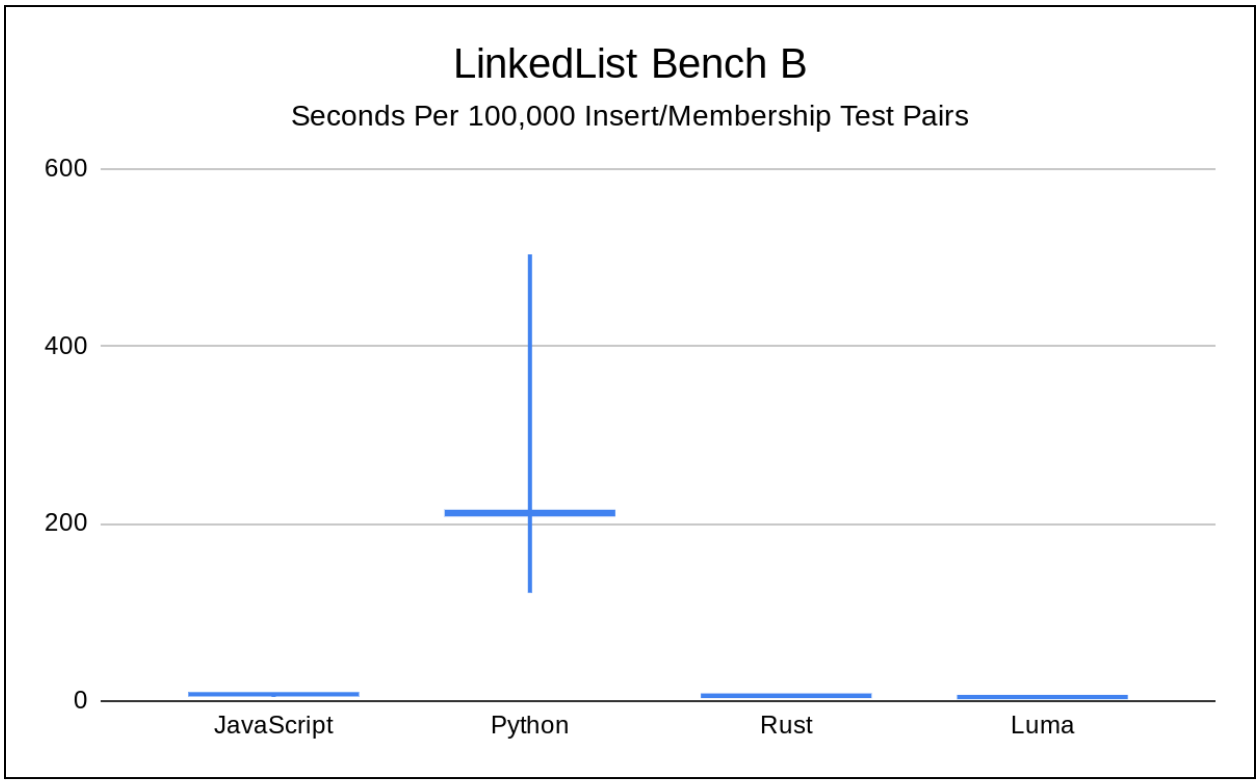




LinkedList Bench

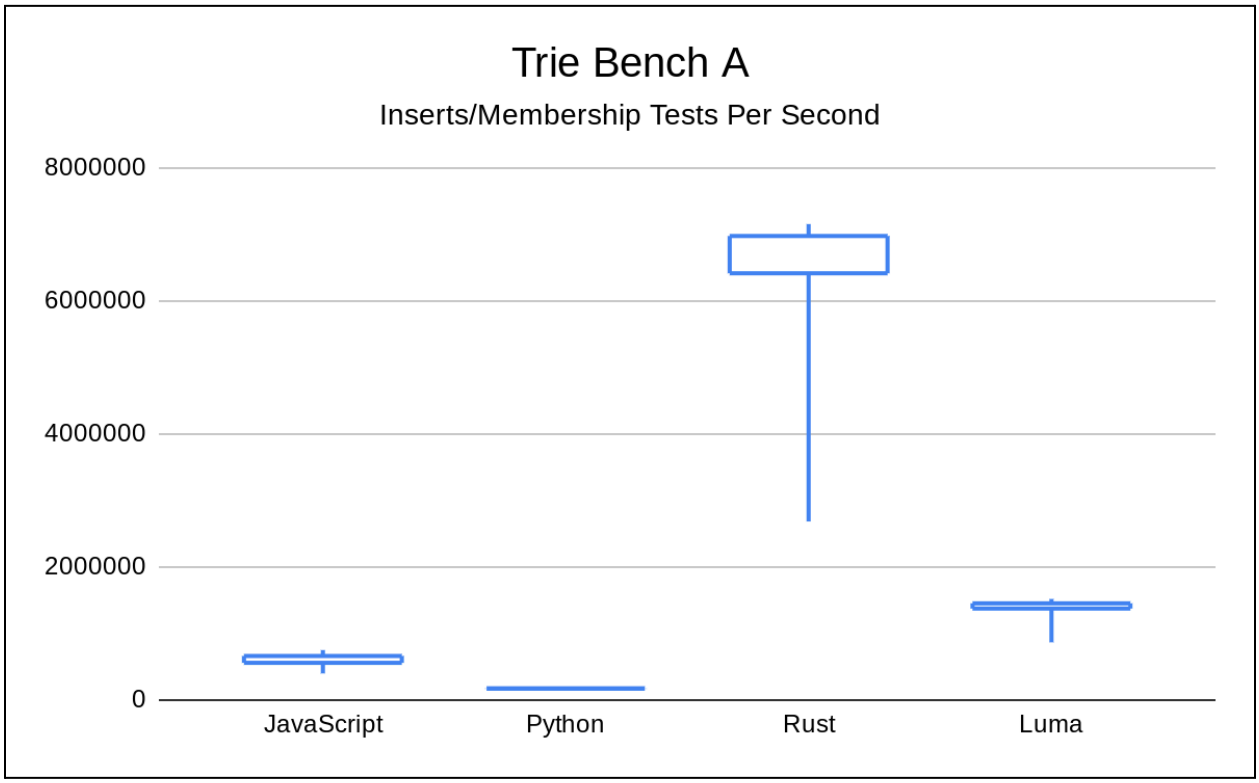
LinkedList is presented as time taken to compute a round of membership tests (whether the list contains a given key) and tail insertions. This test does not include results from the Control variant, as disqualifying soundness flaws were found with that variant when applied to this test input. Small-sample tests with manual soundness fixes reflect performance significantly lower than that displayed by Python for this test, but result confidence for that scenario is low. This test shows improved performance for Luma over Rust, likely due to the fact that the Luma implementation does not use the `prev` member of nodes of a conceptually doubly linked list, and so that field is neither emitted nor updated. Luma displays 28.13% mean performance improvement over Rust, 70.00% mean improvement over JavaScript, and 4580.05% mean improvement over Python.

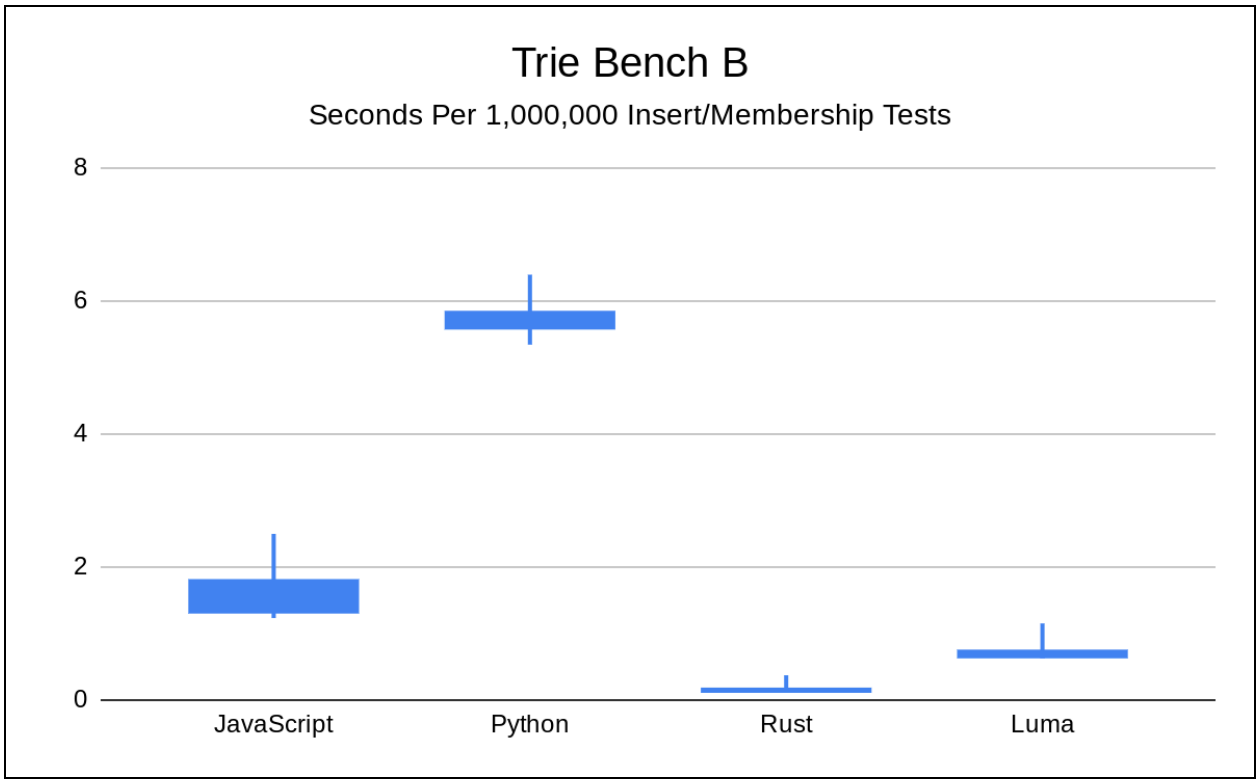

LinkedList Bench A
Insert/Membership Tests Per Second
25000
20000
15000
10000
5000
0
JavaScript
Python
Rust
Luma


LinkedList Bench B
Seconds Per 100,000 Insert/Membership Test Pairs
600
400
200
0
JavaScript
Python
Rust
Luma

Trie Bench

The Trie benchmark is used to analyze recursive call cost and how effectively basic string operations can be optimized. For this test we see substantially better performance from Rust, since the straightforward implementation trivially allows non-owning string slices to be used during recursive calls, so only a single string copy needs to occur for each insert. Luma also does quite well on this benchmark, reflecting the improved optimization opportunities from having fully monomorphic key types and comparatively compact nodes. Python improves here in comparison to other tests, likely reflecting CPython support for string slicing and owning slices. Luma displays a 79.17% mean performance regression in comparison to Rust (Rust taking 0.208 times the amount of time of Luma to complete the average sample). Luma displays 119.40% mean improvement over JavaScript and 729.14% mean improvement over Python. This test again omits data collected from the Control variant due to result tainting from the identified soundness issues.





XII. CONCLUSIONS

At the outset, we sought to develop a compiler which would allow for users to write code using the same paradigms and approaches that they would in contemporary dynamically-typed languages, with a minimal set of ergonomic tradeoffs to allow for the transformations and optimizations (and accompanying predictability and performance) seen in contemporary statically-typed languages. Luma presents a significant first step toward this goal, providing robust field type inference as a zero cost abstraction over traditional static typing. In every measured test, Luma outperformed the control and both tested dynamically-typed languages and displayed negligible loss over a hand-written analog in a contemporary statically-typed language. It provides a compelling alternative to contemporary dynamically-typed languages for the prototype and proof-of-concept phases of project development.

XIII. BIBLIOGRAPHY

[1] Chen, Z., Ma, W., Lin, W. *et al.* A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of Python dynamic features. *Sci. China Inf. Sci.* **61**, 012107 (2018). https://doi.org/10.1007/s11432-017-9153-3 https://link.springer.com/article/10.1007/s11432-017-9153-3.

[2] Milner, R. 1978. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375. http://doi.org/10.1016/0022-0000(78)90014-4.

[3] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type inference for static compilation of JavaScript. SIGPLAN Not. 51, 10 (October 2016), 410–429. https://doi.org/10.1145/3022671.2984017

[4] Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. SIGPLAN Not. 47, 6 (June 2012), 239–250. https://doi.org/10.1145/2345156.2254094.

[5] J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. Proc. ACM Program. Lang. 3, POPL, Article 12 (January 2019), 28 pages. https://doi.org/10.1145/3290325.

[6] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. SIGPLAN Not. 44, 6 (June 2009), 465–478. https://doi.org/10.1145/1543135.1542528.

[7] Mike Pall. 2008. LuaJIT roadmap 2008. (February 2008). http://lua-users.org/lists/lua-l/2008-02/msg00051.html.

[8] Mark Shannon. 2021. PEP 659-Specializing Adaptive Interpreter. (May 2021). https://peps.python.org/pep-0659.

XIV. ADDITIONAL LITERATURE

Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. An Empirical Study on Dynamic Typing Related Practices in Python Systems. In Proceedings of the 28th International Conference on Program Comprehension (ICPC '20). Association for Computing Machinery, New York, NY, USA, 83–93. https://doi.org/10.1145/3387904.3389253.

Jacques Garrigue. n.a. Simple Type Inference for Structural Polymorphism. (n.d.). Retrieved April 26, 2023 from https://caml.inria.fr/pub/papers/garrigue-structural_poly-fool02.pdf

Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In 34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs)), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1-17:29. DOI:https://doi.org/10.4230/LIPIcs.ECOOP.2020.17.

Shiyi Wei, Franceska Xhakaj, and Barbara G. Ryder. 2015. Empirical study of the dynamic behavior of JavaScript objects. Journal of Software: Practice and Experience 46, 7 (2016), 867-889. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2334

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). Association for Computing Machinery, New York, NY, USA, 607–618. https://doi.org/10.1145/2950290.2950343.

XV. APPENDIX

Due to space constraints the source code for Luma may be viewed at:
github.com/szbergeron/luma