Fast and Extensible Hybrid Embeddings with Micros

Sean Bocirnea
University of British Columbia
Vancouver, Canada
seanboc@cs.ubc.ca

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

Abstract

Macro embedding is a popular approach to defining extensible shallow embeddings of object languages in Scheme-like host languages. While macro embedding has even been shown to enable implementing extensible typed languages in systems like Racket, it comes at a cost: compile-time performance. In this paper, we revisit *micros*—syntax to intermediate representation (IR) transformers, rather than source syntax to source syntax transformers (macros). Micro embedding enables stopping at an IR, producing a deep embedding and enabling high performance compile-time functions over an efficient IR, before shallowly embedding the IR back into source syntax. Combining micros with several design patterns to enable the IR and functions over it to be extensible, we achieve extensible hybrid embedding of statically typed languages with significantly improved compile-time compared to macro-embedding approaches. We describe our design patterns and propose new abstractions packaging these patterns.

CCS Concepts: • **Software and its engineering → Extensible languages**; **Software performance**; *Frameworks*.

Keywords: Domain-specific languages, Compilers, Optimization, Shallow Embedding, Deep Embedding, Hybrid Embedding, Language-oriented Programming

ACM Reference Format:
Sean Bocirnea and William J. Bowman. 2025. Fast and Extensible Hybrid Embeddings with Micros. In *Proceedings of the 26th ACM SIGPLAN International Workshop on Scheme and Functional Programming (Scheme '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759537.3762696

1 Introduction

Contemporary macro systems have features for linguistic reuse and interposition, enabling macro embedding whole general-purpose languages with sophisticated static type systems as libraries [21]. Macro embedding is even expressive enough to enable user-extensible dependent type systems [6, 7].

However, macro embeddings can suffer serious performance problems. Shallowly embedding object languages through macros provides an elaboration into a host language, but means there is never a direct representation of the whole embedded language program. Exposing such a representation would require stopping macro expansion at some intermediate point and inverting the usual outside-in order of macro expansion, or decompiling fully expanded programs back into an intermediate representation. This makes implementing optimizations, particularly non-local transformations, difficult or impossible. Worse, it can dramatically affect compile time. By inverting expansion order, stopping expansion, or decompiling, the eventually emitted syntax objects will be re-expanded, resulting in worst-case quadratic expansion cost, and thus quadratic time algorithms for what would otherwise be a linear time AST traversal. As we will show, this is particularly noticeable when embedding typed languages with sophisticated type systems.

An alternative to shallow embedding, deep embedding, would provide a full AST that could be traversed and manipulated efficiently. Deep embeddings can even be integrated with a macro system, so that the macro system essentially provides the frontend, generating an full AST that can then be manipulated efficiently, before finally shallowly embedding back into the host language [1, 2]. Unfortunately, this approach gives up on the extensibility of the IR (although the source language remains macro-extensible).

In this paper, we present an approach to hybrid embedding languages with good compile-time performance and extensibility of both the surface language and internal representation. We develop a taxonomy for extensibility, and use it to compare and contrast our approach with the state of the art. We revisit *micros* as a key abstraction for creating an extensible deeply embedded IR [14]. Our micros target compile-time structs in Racket for an efficient AST representation. For extensible manipulation of the object language AST, we use generics to attach each compile-time function (such as the type checker) to its AST node. We use an extensibility pattern atop structs and generics to navigate the expression problem [23], and propose as future work two new abstractions packaging these patterns. This enables users to extend the IR both with new nodes and their compile-time functions, and override previous behaviour,

Scheme '25, Singapore, Singapore

ACM ISBN 979-8-4007-2162-5/25/10
https://doi.org/10.1145/3759537.3762696

similar to Racket objects but with better compile-time performance. We present a brief case-study of a dependently typed language micro embedded in Racket, extended with gradual typing, and a brief performance analysis of the type checker.

2 Related Work

To understand the language embedding design space, we first survey existing tools for creating language embeddings, which we refer to as *embedding language frameworks* (*ELFs*).

Turnstile. `turnstile+` [5], a dependent-type-supporting extension of the `turnstile` ELF described by Chang et al. [8], uses Racket macros to provide object language extensibility. `turnstile+` encodes object language forms as macros and performs type-checking operations as object languages are macro-expanded into core Racket forms. Adding new forms is as simple as importing a base language and defining new macros for new language forms. Since Racket's macros expander is open recursive, redefinition of old language forms is almost as easy: elide old terms in the import specification and define new terms with the same syntax. Interoperability of object languages with Racket is also easy, since everything expands into a shallow Racket embedding.

While `turnstile+` offers good extensibility characteristics, it fails to offer good compile time performance (see Section 5.2). Since `turnstile+` performs type checking during macro expansion, the competing expansion orders of macro expansion and type checking cause a worst-case quadratic expansion time. Typically, macros are expanded in order from outside in, meaning that in a run-of-the-mill macro invocation a term is walked only once by Racket's macro expander. Type checking requires sub-terms be checked, and thus expanded, prior to making a judgement on the type of parent terms. Since sub-terms must be fully expanded into Racket core syntax to be type checked, each term must first be completely expanded, with each sub-term recursively type checked, prior to its shallow embedding being walked by Racket's macro expander *again* after expansion.

In dependent type systems, like those targeted by `turnstile+`, type checking is significantly more complex than for simply typed languages, worsening performance. Furthermore, type checking in `turnstile+` is performed on syntax objects, requiring the traversal and pattern matching of an inefficient linked-list representation of the object language AST. This incurs poor compile time memory allocation and access performance when compared to a struct-based AST implementation, and makes common optimizations like in-place mutation impossible.

syntax-spec and ee-lib. `syntax-spec` [1] is an ELF which provides a DSL for defining macro-extensible object language syntax, and generates its expansion into an object-language-specific base language. `syntax-spec` is implemented in Racket, and is built over `ee-lib` [2], a compatibility layer between object languages and a host language's macro and binding system. Languages defined with `syntax-spec` expand into a base object language without performing type checking or fully elaborating into core Racket syntax. `syntax-spec` does not allow for extension of this base language, instead opting for macro-extensibility of the object language. Once elaborated into the base language, a program written in the object language can then be compiled by any method of the implementer wishes, even one which is non-extensible. `syntax-spec` also allows an object language to define boundary macros, with which object language users can mix object and host language code.

Given the flexibility it affords in compiler design for the base language, `syntax-spec` offers good compile time performance; an object language implementer can use whatever performant compiler design they wish for the base language, unconstrained by the `syntax-spec` parsing and macro frontend. If one were to type-check an object language defined using `syntax-spec`, a type-checker over the base syntax could be specified as part of the base language compiler, but the `syntax-spec` framework does not provide any means to extend that type-checker. This is in contrast to `turnstile+`, which exposes type-checking semantics as an extensible part of the object language.

LMS. Lightweight Modular Staging [19] (*LMS* for short) is an ELF that directly exposes an object language as a host language data structure, leveraging object-oriented design to achieve extensibility. LMS is implemented as a Scala library, providing the `Rep[_]` type for wrapping staged object language forms, allowing composition and manipulation of object language programs using whatever means the language implementer finds appropriate. LMS does not separately define concrete syntax; instead the object language syntax is that of the host language structure—a language user directly writes the abstract syntax representation of their object language program.

Consequentially, LMS encoded languages have no concrete syntax specification. A user of LMS can separately specify concrete syntax using Scala's macro system, which in the context of LMS would function like `syntax-spec`'s macro frontend, requiring elaboration into an LMS-encoded object language. Unlike `syntax-spec`, this core language could be extended via LMS.

3 Extensibility and Flexibility

While intuition reveals some differences in the ELFs we've described, we must develop a taxonomy for describing extensibility to specify our goals for our hybrid micro embedding. We begin by defining notions of syntax and semantics, consider embedding styles for ELFs, and then classify the ELFs we've discussed by their support for the extension of object languages in each of these three dimensions.

3.1 A Taxonomy for Extensibility

To describe classes of extensions over languages, we must first be able to describe languages.

Concrete syntax specifications dictate which strings are valid program syntax, declaring what users must write to interact with the features, binding forms, and lexical structures of a programming language. Abstract syntax specifications then declare the structure and organization of program syntax. We think of syntax as a first filter: semantic specifications can be used to reason about any well-formed piece of abstract syntax, but not every well-formed piece of syntax is a valid program.

Our notions of extensibility apply to any means of defining abstract syntax specifications, but in our examples we use the abstractions of *syntax classes* and *meta-variables*. Take for example the abstract syntax definition for the λ-calculus in Figure 1. We group syntax using the meta-variables x and t, allowing us to refer to classes of valid abstract syntax representing variables and expressions. Meta-variables allow syntactic forms to be parameterized on all possible instances of syntax of a particular class; for example the t in the syntax specification $t\ t$ denotes any possible expression. We use *term* to refer to any piece of abstract syntax which abides by a given object language's syntax specification.

$$\begin{array}{lll} x & \in & \mathit{Variable} \\ t & ::= & x \mid \lambda x.t \mid t\ t \end{array}$$

$\boxed{t \longrightarrow t'}$

$$\beta\text{-RED} \quad (\lambda x.t)\ t' \longrightarrow t[x \mapsto t']$$

Figure 1. The λ-calculus with (small-step) reduction semantics, omitting α, η, and substitution.

Semantic specifications are imposed on terms, encoding properties of a language like which pieces of syntax are well-formed programs, and how well-formed programs behave when they're executed. Models of language semantics may be abstractly thought of as *judgements* over terms, where the inclusion of syntax in a judgement encodes some meaning. When discussing extensibility, we use *judgement* to refer to all statements one could make over about abstract syntax, including specifications of both static and run-time semantics. For example, consider Figure 1, with the judgement $t \longrightarrow t'$ expressing the reduction of terms in the λ-calculus. $t \longrightarrow t'$ relates (possibly distinct) syntaxes representing terms, where t reduces to the term t' iff $t \longrightarrow t'$ holds for the pair of syntaxes (t, t'). By defining $t \longrightarrow t'$ for all terms, we encode the semantics of "reducing a term;" a term t reduces to t' because $t \longrightarrow t'$ holds for the pair of terms, and we check if an arbitrary term reduces to another by proving that the pair of terms satisfy the judgement.

Syntactic extension is the modification of the specifications for the syntactic forms and classes that could potentially make up a program. Syntactic extension is broadly useful: in order to add language features, we would like to be able to add new syntactic forms exposing them. But, *how* might we want to change base language syntax? Is an extension strictly additive, adding classes and cases but not modifying existing syntax, or might an extension have the need to modify existing syntax?

We call extensions which only add syntax *additive* and extensions which change existing syntax *strong*. Note that strong syntactic extensibility need not be strictly more desirable than additive syntactic extensibility, as depending on the goals of an ELF, it may not be practical to provide desired performance or usability characteristics while supporting strong extension. We perform an additive syntactic extension in Figure 2 to extend the λ-calculus into a Scheme-like well-scoped calculus we'll call λ-s. The syntactic portion of this extension is the addition of a binding context Γ, which may either be the empty set $\emptyset$ or a pair of some context and a variable name.

$$\begin{array}{lll} x & \in & \mathit{Variable} \\ t & ::= & x \mid t\ t \end{array} \qquad \Gamma \quad ::= \emptyset \mid \Gamma, x$$

$\boxed{t \longrightarrow t'}$

$$\beta\text{-RED} \quad (\lambda x.t)\ t' \longrightarrow t[x \mapsto t']$$

$\boxed{\Gamma \vdash t}$

$$\textsc{Gamma}\ \frac{x \in \Gamma}{\Gamma \vdash x} \qquad \textsc{Lambda}\ \frac{\Gamma, x \vdash t}{\Gamma \vdash \lambda x.t} \qquad \textsc{App}\ \frac{\Gamma \vdash t \qquad \Gamma \vdash t'}{\Gamma \vdash t\ t'}$$

Figure 2. Extension of the λ-calculus with a well-scopedness judgement, forming λ-s. Additive extensions in blue.

Semantic extension provides new meaning to terms. Consider again the λ-s extension in Figure 2. To complete the new language feature, we need to add the judgement $\Gamma \vdash t$ to encode the semantics of a "term being well-scoped." We call the introduction of $\Gamma \vdash t$ a *judgement-level semantic extension*, which is the means by which we add new judgements or modify the dependencies between existing judgements. Analogous to syntactic extension, we introduce the notions of additive and strong judgement-level semantic extensibility. Here, we need only additive judgement-level semantic extensibility since we did not need to change any existing judgements, namely $t \longrightarrow t'$.

Suppose we want to introduce types to this well-scopedness judgement, producing a simply-typed λ-calculus (*STLC*), with Church-style intrinsic typing. We perform both kinds of syntactic extension in Figure 3, using additive syntactic extension to add a meta-variable for types, τ, and syntax for

function types, constants, and base types. We then strongly extend the existing syntax for terms t, modifying the syntax for functions $\lambda x.t$ to include a type annotation, and do similarly for contexts Γ:

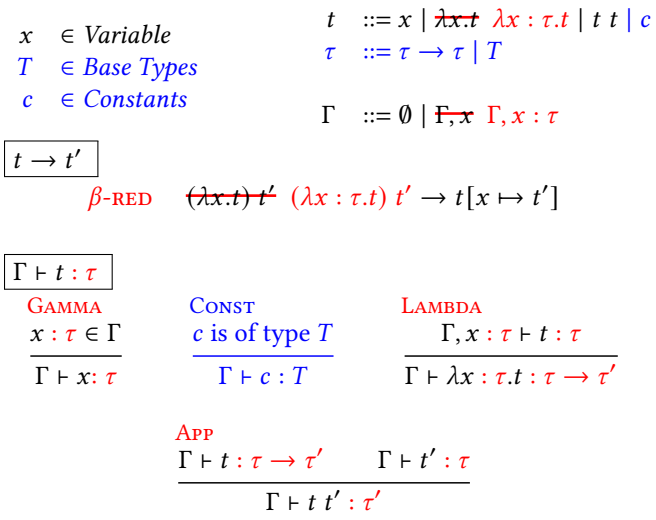

Figure 3. An extension of λ-s into STLC. Additive extensions in blue and strong in red.

In our STLC example, we use strong judgement-level extension to add the $\Gamma \vdash t : \tau$ judgement, which modifies the scopedness judgement $\Gamma \vdash t$. Judgement-level extension complete, we have a new class of statements we can make about terms, but have not yet described any such statement. To do so, we must also perform a *rule-level semantic extension,* where we extend meaning to new terms under existing judgements. Rule-level extensions can also be either additive or strong. Here, we use additive rule-level semantic extensibility to add Rule CONST, which types constants. We then use strong rule-level extension to modify the former $\Gamma \vdash t$ rules, resulting in typing rules which respect our new $\Gamma \vdash t : \tau$ judgement. As it stands, no other judgements in our STLC specification make use of the $\Gamma \vdash t : \tau$ judgement, but it remains exposed to a user of the ELF as a judgement one can use to describe STLC programs.

Note that additive syntactic extensions may require strong semantic extensions in order to achieve desired behavior. For example, sound gradual typing is a typing discipline which allows for static imprecision in typing that induces run-time constraint checks [20]. Often presented in literature as a modification of some base type system, the run-time semantics of a gradually typed language differ from those of the base language, changing how existing terms are interpreted. To add gradual types to a simply typed system, we would need to add the unknown type, an additive syntactic extension. Semantic extensions supporting this new syntax must be made to existing judgements, strongly extending rule-level semantics. We summarize our extensibility classes in Table 1.

The classes of extensibility we define map well to the *expression problem*, which as described by Wadler [23], is the problem of achieving extensibility of both data and addition of functions which operate on that data. In the context of programming languages, terms are data, and language semantics are functions over terms. Designing ELFs that support both rule-level and judgement-level semantic extension requires one to solve the expression problem. We require extensibility of data to achieve syntactic and rule-level semantic extension, and extensibility of functions on that data for judgement-level semantic extension. To support *either* rule-level semantic extension or judgement-level semantic extension, one does not need to solve the expression problem. Rule-level extension is modelled with object-oriented programming (*OOP*) structures, where syntax maps to classes, and judgements to methods called by a visitor, but adding new methods is hard. In contrast, judgement-level extension maps to typical functional patterns, where writing new functions which operate on an existing datatype is easy, but adding new cases to union types requires modification of existing functions.

Lastly, we define *extension* and *modification*. In Wadler's presentation, a solution to the expression problem should allow for extension without modification and re-compilation of (in our context) a base language's compiler; if the original language no longer exists as a distinct entity—a distinct unit of compilation in the host language—the object language has not been extended, it has been modified.

3.2 A Taxonomy for Flexibility

An embedding ELF's choice of embedding strategy greatly impacts the flexibility of the object language, be it *deep*, *shallow*, *hybrid* [11], or something less common like *compositional* [24] or *polymorphic* [12] embeddings. The related works discussed in this paper implement some variation of the first three, summarized below:

- **Deep embeddings** encode object languages as an AST which can be traversed and manipulated in the host language. Whole-program transformations can traverse the entire AST of object language programs, and a final step in compilation transforms the object language AST into host language code. Since the encoding of object language terms is different from those of equivalent host language terms, deep embeddings leak implementation details which prevent naive interoperability with equivalent host language features.
- **Shallow embeddings** encode object languages directly in terms of equivalent host language functionality. A shallowly embedding ELF can never reason about an AST as a whole, and as a consequence cannot perform whole-program transformations. However,

Table 1. Extensibility classes.

Strength \ Domain	Syntactic	Semantic	
		Rule-level	Judgement-level
Additive	Addition of syntactic forms	Addition of syntax to existing judgements	Addition of new judgements
Strong	+ modification of existing syntactic forms	+ modification of judgements over existing syntax	+ modification of existing judgements

shallow embeddings benefit from much better host language interoperability, as host language features can be used directly on encoded object language terms, as their encoding is idiomatic in the host language.

- **Hybrid embeddings** construct an AST as a deep embedding would, upon which whole-program transformations can be defined, but allows host language features to interact with object language terms like a shallow embedding. Hybrid embedded object language terms have both an internal representation (used to construct a whole-program AST) and a wrapper allowing for direct host language interoperability.

We note that with shallow embeddings, host and object language interoperability generally is possible at term boundaries; one can pass object language terms to host language functions and vice versa. With hybrid embeddings, an object language author must choose at which granularity to force the construction of a complete object language AST. In order to perform meaningful optimizations, it may be the case that interaction between host and object language can only occur at, for example, module boundaries.

3.3 Related Work, Taxonomized

Having established our taxonomy, we re-examine the ELFs discussed in Section 2.

Turnstile. Recall that `turnstile+` encodes language forms as host-language macros, which can be imported from a base language and shadowed by language extensions. Since each macro is responsible for the transformation and typechecking of its associated syntactic form in the object language, macro shadowing allows `turnstile+` to support strong syntactic and strong rule-level semantic extension. `turnstile+` also supports strong judgement-level semantic extension, as new terms can implement new judgements, and by shadowing old terms, old judgements can be modified for existing terms. Given `turnstile+` language forms are Racket macros, which individually expand fully to Racket core forms, `turnstile+` is a shallow embedding ELF. `turnstile+` object language terms can be individually imported and used in (host) language code, and a user does not have access to arbitrary AST transformations.

syntax-spec. A Racket-embedded DSL framework, `syntax-spec` provides a syntax specification language and macro system for object languages, with a conventional compiler design downstream. This approach requires that all extensions must live within an *object language* macro system. This is in contrast to `turnstile+`, where extensions are written using host-language level macros. Consequentially, one cannot *extend* a `syntax-spec` encoded language if it turns out that new constructs need to be introduced to the core in order to support a desired feature, but must instead modify the `syntax-spec` definition for the DSL core in order to express novel semantics.

As an example, in STLC as we defined in Figure 3, we could not add types previously unknown to the compiler. In `syntax-spec`, object-language extensions are limited to using macros to translate new syntax into existing core syntax. That is, the object language is macro extensible [10], but neither truly syntactically extensible nor truly semantically extensible in our taxonomy, since the core language cannot be extended. This is in contrast to `turnstile+`, where the *compiler* is specified extensibly, and can be iterated upon without clobbering the compiler for a base language. `syntax-spec` also allows an object language specification to define macros which introduce an interaction point between host and object language terms, making `syntax-spec` a hybrid embedding ELF.

LMS. LMS supports strong syntactic extension, and both strong rule-level and strong judgement-level semantic extension. Since LMS object language constructs are encoded as Scala classes, LMS is a deeply embedding ELF. Scala's OOP and trait system allows LMS to achieve these extensibility properties, at the expense of poor host language interoperability. LMS's `Rep[_]` type, used to stage object language programs, leaks into instances of object language syntax and typing, making object language representations incompatible with equivalent Scala features. LMS thus deeply embeds its object language.

4 A Micro Embedded Framework

We want to determine if it is possible for a hybrid embedding ELF to support strong syntactic, strong rule-level and strong judgement-level semantic extensibility, while compiling a dependently typed object language faster than `turnstile+`. Ideally, we want compilation performance on par with an object language implemented with an unrestricted choice

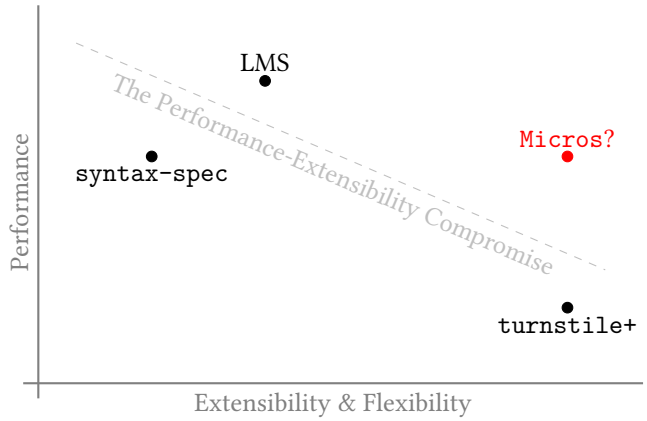

Figure 4. The ELF design space, optimistically.

of compiler architecture. To evaluate this research question, we begin developing a Racket-hosted ELF, using micros as a syntactic frontend which elaborates into a struct-based IR, made extensible using Racket generics. Given our analysis of related work, we can crudely draw the ELF design space in Figure 4. Note the gap between `turnstile+` and `syntax-spec`/LMS; this is the gap between the extremes of extensibility and performance we want to target with our micro embedding framework.

We present our micro embedding strategy as a collection of programming patterns, which we realize in Racket. To evaluate the extensibility of micro embeddings, we implement `fowl-base`, a bidirectional Martin-Löf type theory with natural, boolean, equality and Π types, which we extend independently with vector and Σ types. We then inherit from both extensions to form `fowl`, which we use as a base to further interrogate the extensibility properties of micro embeddings. We extend the `fowl` type checker with gradual dependent types [9] to create `fowl-geq`, and attempt to add exceptional types [17] to create `fowl-rett`. Our IR is unable to encode the `fowl-rett` extension; we discuss future work required to support `fowl-rett` in Section 6. We conclude in Section 5.1 that `fowl` strongly supports syntactic and judgement-level semantic extension, but only additive rule-level extensibility. In Section 5.2, we conclude that compilation of `fowl` performs two orders of magnitude better than a similar language written in `turnstile+` when evaluated on a dependently typed benchmark suite.

4.1 The Problem with Macros

Consider the following Racket pseudocode, shallowly embedding an object language `my-if` construct in terms of Racket's native `if`:

```
(define-syntax (my-if stx)
  (syntax-parse stx
    [(_  pred con alt) #'(if pred con alt)]))
```

Easy, breezy, beautiful. But, what happens when we want to transform a program that uses `my-if`? Suppose we'd like to perform a static optimization on `my-if`, compiling `my-if` to `con` if `pred` is a tautology. A naïve option is to pattern match the syntax of `pred`, using some internal logic to search for syntactic patterns that will always yield true a run time:

```
(define-syntax (my-if stx)
  (syntax-parse stx
    [(_  my-true con alt) #'con]
    [(_  (my-not my-false) con alt) #'con]
    ;; ... and many more
    [(_  pred con alt) #'(if pred con alt)]))
```

In general, our macro could thread some context through expansion to aid in symbolic execution, making this approach effective. However, it is also fragile: any change to program syntax, such as macro extensions of the source language, defeats this optimization. If we add new core syntax, we need to *modify* `my-if` to include those cases as well. This fails to meet our criterion for extensibility.

Suppose then that we require all forms to expand to their optimized core representation, and use `local-expand` to invert expansion order, expanding sub-expressions first into core forms. All macro extensions will be elaborated away, and core forms will optimize themselves if possible. Then, we can implement `my-if` as follows.

```
(define-syntax (my-if stx)
  (syntax-parse (local-expand stx)
    [(_  #t con alt) #'con]
    ;; ... and many more
    [(_  pred con alt) #'(if pred con alt)]))
```

Now, we've rediscovered the implementation strategy used by `turnstile+`! In `turnstile+`, sub-terms are type checked and manipulated by invoking `local-expand`, allowing extensions to be transparent to existing macros. For tasks like normalization, `turnstile+` can parse Racket core forms as the core IR.

Unfortunately, while extensible, this approach causes performance problems. Using `local-expand` and then including the emitted syntax object in our macro's output can cause quadratic expansion times, since the emitted syntax object (and all of its sub-expressions) will be re-expanded. For computationally intensive passes, such as dependent type checking algorithms, relying on syntax objects (an immutable linked list data structure) hampers performance optimizations we might want to perform.

All of these problems stem from macros expanding into syntax objects, and staying in the macro expander. So why not elaborate into something else, and interrupt macro expansion?

To implement this approach, we can use *micros* [14], which are transformers from syntax to a core IR, rather than syntax to syntax. Each micro expands into a core IR term, represented as an arbitrary data structure, and no more macro expansion happens on it. For example, we might define a `my-if` *micro* using structs as our IR representation.

```
(begin-for-syntax
  (struct if-ast (pred con alt)))
(define-syntax (my-if stx)
  (syntax-parse stx
    [(_  pred con alt)
     (if-ast (local-expand #'pred)
             (local-expand #'con)
             (local-expand #'alt))]))
```

This yields a deep embedding. If we use (compile-time) structs with generic methods attached, we can expand into an extensible core IR with many additional extensible compiler passes, such as type checking, before eventually shallow embedding back into syntax objects. This micro cannot be used by Racket's macro expander directly, since the macro expander expects transformers to produce syntax objects. We will need to trick the expander into letting us produce other data structures.

In the rest of this section, we present using this approach to implement `fowl-base`, an extensible dependently typed language hybrid embedded into Racket.

4.2 fowl-base-syn

`fowl-base` is two Racket modules. The first, `fowl-base-syn`, uses micros to define the syntax of `fowl-base` and its elaboration into an IR defined in the module `fowl-base-sem`.

Like `turnstile+` and `syntax-spec`, `fowl-base`'s micros are written using Racket macros. `fowl-base-syn` exports all defined micros, which can be imported either individually for term-level interoperability, or as a Racket hashlang in which a user can write `fowl-base` language programs. Figure 5 includes the micro definition for `if`, which expands into an abstract syntax representation.

Racket's module system makes it easy to extend our micro embeddings: an extension imports the syntax and semantics modules from a base language, and exports its own language forms, shadowing the base language. To perform a weak syntactic extension to `fowl-base`, we define a new micro in our extension. If we desire strong extension, this new micro can shadow the name of the old micro we wish to override. Modules also allow for separate compilation of a base and extended language.

We want micros to expand into structs, but Racket macros are syntax to syntax transformers; we need a way to return data instead. We implement micros on top of macros using the *mule pattern* demonstrated in Figure 5. The *mule pattern* creates a dummy piece of syntax, `#'eeyore`, and attaches to it the micro-expansion of the term as metadata (using syntax properties). The dummy syntax requires no further expansion, avoiding re-expansion costs. Our micros all follow this pattern, using a macro to define surface syntax, and the mule pattern to compose and return our IR representation, carrying it through macro expansion. `elab-to-structs` is used to recursively expand micros, returning the struct representations of child terms.

```
(define eeyore void)

(begin-for-syntax
  (define mule #'eeyore)
  (define (burden-mule expansion)
    (syntax-property mule 'expansion expansion))
  (define (unburden-mule m)
    (syntax-property m 'expansion))

  (define (elab-to-structs e)
    (define idx (syntax-local-make-definition-context))
    (syntax-parse (local-expand e 'expression '() idx)
      [e:expanded-term (unburden-mule #'e.body)]))

(define-syntax (if stx)
  (syntax-parse stx
    [(_  pred con alt)
     (burden-mule (fi:if-term (elab-to-structs #'pred)
                              (elab-to-structs #'con)
                          (elab-to-structs #'alt)))]))
```

Figure 5. An excerpt of `fowl-base-syn` demonstrating the `mule` micro pattern.

4.3 fowl-base-sem

`fowl-base-sem` defines mutable *structures*, one for each `fowl-base` language term. Racket structures are named records, which can optionally support *generic methods* (or in OOP nomenclature, interface methods) that can be called on any instance of a structure implementing the generic. Using structures as our IR representation improves performance over a syntax object representation, as structure fields have constant time access, and mutability allows the IR to be manipulated in place by language transformations.

`fowl-base-sem` also defines rules and judgments over the struct IR. To maintain extensibility, each `fowl-base` judgement is defined as a generic interface. To add a judgement rule for a `fowl-base` term, we attach an implementation for the judgement's generic interface to the structure in `fowl-base-sem` representing the `fowl-base` term.

Structs in an extension can implement a generic method for a judgement related to their AST node, giving us additive rule-level semantic extension. Note that this requires that judgement rules are syntax-directed. Additive judgement-level extensibility is also easy: extensions can define a new generic interface for new judgements, using `#:defaults` to implement rules for base language structs.

We see generics as judgements in practice in Figure 6. `fowl-base` is a bi-directionally typed language, so terms may implement either a type synthesis judgement, a type checking judgement, or both. In Figure 6 we define the synthesis and check judgements of `fowl-base` and a portion of the internal representation of `suc`, the successor constructor for Peano numerals. Here, we implement the inference rule for the synthesis judgement. The statement `#:methods gen:ir-elaborable` denotes that `suc` implements the generic `ir-elaborable` interface, and thus the

`elab-ir-synth!` and `elab-ir-check!` judgements. The choice to define both `synth!` and `check!` under the same interface is one of convenience, since in `fowl-base` all terms happen to implement both judgements. Unlike `turnstile+`, the synthesis judgement for `fowl-base` is able to mutate structures in place, meaning that when `fowl-base-syn` expands `fowl-base` surface syntax into `fowl-base-sem` structures, the synthesis and check judgements do not allocate and return a new syntax tree, but instead manipulate the input abstract syntax in place.

```
(define-generics ir-elaborable
  [elab-ir-synth! ir-elaborable t-env v-env r-env]
  [elab-ir-check! ir-elaborable t t-env v-env r-env])

(parameterize-judgement elab-ir-synth!)
(parameterize-judgement elab-ir-check!)

(serializable-struct
  suc term (nat) #:mutable #:transparent
  #:methods gen:ir-elaborable
  [(define (elab-ir-synth! e t-env v-env r-env)
     (match-define (suc n) e)
     (let ([n~ (elab-ir-check!$ n Nat
                                t-env v-env r-env)])
       (when n~ (set-suc-nat! e n~))
       (values #f Nat)))])
```

Figure 6. An excerpt of `fowl-base-sem` demonstrating a judgement and term definition.

While the patterns we've covered thusfar suffice to provide additive extensibility, they do not provide *strong* semantic extensibility. For strong extensibility, we need some way to dynamically bind rules and judgements, so that we can override base language behavior with new behavior defined in an extension. To achieve this, we introduce interposition points for judgements using Racket parameters, which can be dynamically rebound. We call this the *extensible generics* pattern.

The extensible generics pattern introduces a parameter for each generic, which acts as an interposition point for the judgement. All extensible generics call their implementation through this parameter, so by modifying the parameter, the judgement can be strongly extended.

In Figure 6 we call `parameterize-judgement`, a helper macro that uses inserts a parameter for `fowl-base` judgements. We present the full definition in Figure 7. Given the name of a judgement's generic function `A`, the macro wraps `A` in a parameter using `make-parameter`, and binds it to `A-prm`. `parameterize-judgement` also creates a memoized accessor procedure for this parameter, named `A$`. We then use `A$` in the rest of `fowl-base` to invoke the judgement encoded by our original function `A`. Memoization is used to improve compile time performance, as parameter bindings do not change as a micro embedded language compiler is running, but are accessed regularly.

```
(define-syntax (parameterize-judgement stx)
(syntax-parse stx
  [(_ gen)
   (let ([name (format-id #'gen "~a-prm" #'gen)]
         [acc  (format-id #'gen "~a$"    #'gen)])
     #`(begin (define #,name (make-parameter gen))
              (define #,acc (memo-prm #,name))))]))
```

Figure 7. `parameterize-judgement` macro for parametric binding of object language judgements.

Introducing an interposition point for judgement forms gives us strong judgement-level semantic extensibility. Thanks to these interposition points, languages extending `fowl-base` can modify judgements over `fowl-base` terms without requiring the recompilation of `fowl-base-sem`. Since `A-prm` is dynamically bound, we can redefine it in a language extension, resulting in all instances of `A$` within scope of that redefinition (including those of the base language) changing. We give an example of this with `fowl-geq` in Section 5.1.2.

Not shown is a pattern for obtaining strong *rule*-level semantic extensibility. To do so, we would require interposition points for structs, allowing us to rebind constructors for base language structs. Unfortunately, this was only realized during the implementation of `fowl-rett`, and thus `fowl-base` achieves only weak rule-level semantic extensibility. Section 5.1.3 expands on the consequences of this omission, and we consider approaches for fixing this in Section 6.

5 Evaluation

To evaluate the extensibility of our micro embedding approach, we extend `fowl`'s type system. To quantify the performance of micro embeddings, we compile a suite of dependently typed programs in both `fowl` and `Cur`, a dependently typed calculi implemented in `turnstile+`.

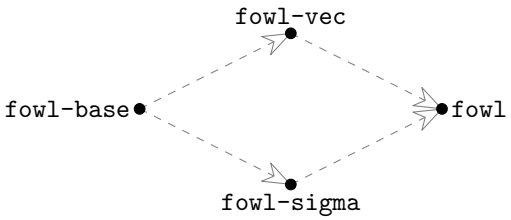

Figure 8. `fowl`'s diamond inheritance structure.

5.1 Extensibility

5.1.1 Additive Extension. To test additive extensibility, we define two extensions to `fowl-base`, `fowl-vec` and `fowl-sigma`, and combine them to create `fowl`; Figure 8 summarizes this inheritance pattern. In Figure 9 we include an excerpt of `fowl-vec`'s implementation. Here, `fowl-vec`

```
;; fowl-vec-sem
(require fowl-base-sem)
(provide (all-defined-out)
         (all-from-out fowl-base-sem))

(serializable-struct vec-term ground (len type) ...)
```

```
;; fowl-vec-syn
(require fowl-base-syn (for-syntax fowl-vec-sem))
(provide (all-from-out fowl-base-syn)
         Vec nil :: ind-Vec)

(define-syntax (Vec stx)
  (syntax-parse stx
    [(_ len type)
     (burden-mule (vec-term (elab-to-structs #'len)
                            (elab-to-structs
                            ↪  #'type)))]))
```

Figure 9. An excerpt of `fowl-vec-syn` and `-sem`.

imports `fowl-base` and defines new macros and structures for each of `fowl-vec`'s new terms, exporting both the new definitions and those original to `fowl-base`. `fowl-sigma` is written similarly, and is omitted here. Since we never needed to modify `fowl-base`, it remains its own language, and can be used as before in `fowl-base` language programs.

`fowl` can then be defined by the code in Figure 10. Since both `fowl-vec` and `fowl-sigma` make additive, nonconflicting extensions, all that is required is a simple dependency merge. We conclude that `fowl` supports at least additive syntactic extension and additive rule-level semantic extension.

```
;; fowl-syn
(require fowl-vec-sem fowl-sigma-sem)
(provide (all-from-out fowl-vec-sem fowl-sigma-sem))
```

```
;; fowl-sem
(require fowl-vec-syn fowl-sigma-syn)
(provide (all-from-out fowl-vec-syn fowl-sigma-syn))
```

Figure 10. The entirety of `fowl-syn` and `fowl-sem`.

5.1.2 Judgement-level Semantic Extension.

To test judgement-level semantic extensibility, we extend `fowl` with gradual types. *GEq* [9] is a dependently typed language with gradual types and sound equality over gradually typed terms. Relative to Bidirectional-CIC, GEq modifies the semantics and syntax of `J`, `Equal` and `refl`, and introduces a new judgement we name `elab-geq-coerce-unk!`, that coerces the unknown type to a term with a type requested by a check judgement.

In order for `fowl-geq` to coerce unknown terms in its check judgement, we use strong judgement-level extensibility to shadow `fowl`'s old check judgement with the modified one we define in `fowl-geq-sem`. Figure 11 shows how we achieve this with Racket's parameters: When calling into the entry point of `fowl`'s type checker, we `parameterize` the call with the function implementing `fowl-geq`'s instance of the check judgement, rebinding all uses of check in `fowl`. We also use strong syntactic extension in order to shadow `fowl`'s definitions for `J`, `Equal` and `refl`, extending their syntax with that required by GEq. The new equality terms elaborate into new `J-geq`, `equal-geq` and `refl-geq` structures in `fowl-geq-sem`. While normally we'd need to use strong rule-level extensibility to revise the check judgement for the old `J`, `Equal` and `refl` terms, they are never constructed by any other term in `fowl`, and therefore the old check implementation for these terms will never be invoked by a `fowl-geq` program.

```
(define (elab-geq-constr-synth! e h args
  t-env v-env r-env) #| Implementation ... |# )

(define (elab-geq e)
  (parameterize
    ([elab-ir-constr-synth!-prm elab-geq-constr-synth!]
     [type-consistent?-prm type-consistent-geq?])
    (define-values (e^ e^-t) (elab-ir-synth!$ e
    ↪  (empty-env) (empty-env) (empty-env)))
    (values (if e^ e^ e) e^-t)))
```

Figure 11. A snippet of `fowl-geq-sem` rebinding a judgement form.

5.1.3 Rule-level Semantic Extension.

To test rule-level semantic extensibility, we extend `fowl` with exceptions. Reasonably Exceptional Type Theory [17], or *RETT*, is an extension of CIC that adds exceptions that can be thrown and caught. Unlike CIC, which has a single hierarchy of type universes (`(Type 0)` being the smallest of such universes in `fowl-base`, represented by the `sort-term` structure type in `fowl-base-sem`), RETT has three separate universe hierarchies. As a consequence, `sort-term` now needs to store an additional field: not only the universe level, but also which hierarchy that universe belongs to; let's call its replacement `sort-term-rett`. In this case, only some of `fowl`'s terms need to interact with `sort-term-rett` any differently than they would with `sort-term`.

Unfortunately, the evaluated version of `fowl` does not have interposition points for abstract syntax constructors. Consequentially, we cannot replace all instances of the `sort-term` constructor in `fowl` with `sort-term-rett`, defined in `fowl-rett`, without replacing the judgements that refer to `sort-term`. To deal with this, we have two options:

1. Replace all terms that interact with `sort-term` with new `fowl-rett` terms, as we did with `fowl-geq`.
2. Create new check and synthesis judgements, taking advantage of strong judgement-level extensibility, and replace the judgements in `fowl` with these new judgements, which support `sort-term-rett`.

While both options would *work*, we hesitate to legitimize either as an extension; we'd be replacing a large portion of

either `fowl` terms, or the synthesis and check judgements in `fowl`, that have nothing to do with the change we actually want to make. Besides being annoying, it's bad programming practice—changes and fixes made to `fowl` will not propagate to `fowl-rett`. To fix this, we need interposition points on struct constructors, but this is nontrivial to support with generics; we discuss this further in Section 6.

5.1.4 Micro Embeddings, Taxonomized.

We conclude the following:

1. Micro embeddings can support strong syntactic extension, exemplified by `fowl-geq`, where we are able to redefine the syntax for `J`, `Equal` and `refl` without modifying `fowl`.
2. Micro embeddings can support additive rule-level semantic extension, since we are able to additively extend `fowl-base` with semantics for vectors and Σ types in `fowl-vec` and `fowl-sigma`, implementing existing `fowl-base` judgements. However, when implementing `fowl-rett`, our implementation oversight means we are unable to replace `sort-term` without *also* modifying judgements. Micro embeddings are therefore not shown to support strong rule-level semantic extension, but introducing interposition points for struct methods would resolve this issue.
3. Micro embeddings can support strong judgement-level semantic extension, since we are able to add gradual types and judgements in `fowl-geq`, and also modify the original check judgements in `fowl-base`.

5.2 Performance

To evaluate the performance of our micro embedding approach, we benchmark `fowl` against `cur` [5] (a dependently typed proof assistant written in the `turnstile+` ELF) and against `smalltt` [13] (a small non-extensible high-performance dependently typed language implementation) on a suite of dependently typed programs.

`cur` is a Racket hashlang, meaning that a `cur` program can be compiled to a Racket binary, performing all macro expansion into Racket core forms and type checking as specified in `cur`'s `turnstile+` implementation, resulting in an executable file. `fowl` behaves similarly, as it too is a Racket hashlang, compilation of which performs type checking and elaboration into Racket. Thus, `fowl` and `cur` benchmarking times are the complete compilation time including all parsing, expansion and compilation steps, including those of the Racket compiler.

In contrast, `smalltt` does not output compiled binaries, but does parse and type check programs. We use `smalltt` to represent the *absolute best-case scenario* for performant ELFs like LMS and `syntax-spec`. `smalltt` benchmarks favorably against all mainstream dependently typed languages (being one to two orders of magnitude faster than `Agda` [15], `Coq` [16], `Lean` [22][1], and `Idris 2` [4], as seen in Table 3) and uses both efficient term representation and optimizations to its normalization algorithm. Recall, `cur` has the additional work of re-walking macro expansions, and also has no algorithmic optimizations. `fowl` elaborates into an efficient IR representation before performing type checking (not needing to re-walk the AST), and performs elaboration in place, but otherwise does not make any efforts to avoid repeated work during normalization, and has no algorithmic optimizations. Any performance gains or losses relative to `cur` are thus predominantly due to `fowl`'s internal representation of language forms and hybrid embedding style. Since both `cur` and `fowl` share the same reader, runtime and compiler, their performance can be directly compared. This is not the case for `smalltt`, which we include only as a goalpost.

We use `hyperfine` [18] to estimate mean compilation time for `cur` and `fowl` across each benchmark. As configured, `hyperfine` reports the average duration of ten executions of `raco make`—which invokes the Racket compiler—on each benchmark program. Before these ten timed runs, three warm-up runs are performed and discarded to reduce the impact of cold processor caches. All benchmarks are executed in the same directory on the same 16-core Intel Skylake server machine with 16384 KB of L1 cache per core and 128 GB of total RAM. Each benchmark is executed sequentially, with a single core active at any one time. All benchmarking runs were completed within a 2-day period, with some `cur` runs not completing because the machine ran out of memory before type checking could complete. These runs are marked **DNF** in Table 2, with compilation times before failure all exceeding half an hour. For benchmarks of `smalltt`, we follow the procedure described in Kovács [13] and manually reload the benchmark in the `smalltt` repl thirteen times, discarding three warm-up runs. The mean and standard deviation in seconds for all runs is reported in Table 2.

In Figure 12 we plot mean compilation times for each language and benchmark. Note a clear trend across all benchmarks: `fowl` is about 1000x slower than `smalltt` across the board, and `cur` is 100x slower that `fowl`, when it could even compile the benchmark program. We discuss each benchmark in more detail below:

asymp-small. This benchmark defines an inductive vector type and constructs a 100-element vector, with each element being the base universe type, (`Type 0`). The `fowl` version of this benchmark uses the primitive vector defined in the `fowl-vec` extension; both the `cur` and `smalltt` versions

[1] `Lean` deserves note, as `Lean-4` has an extensible grammar which elaborates to a fixed core type theory referred to as the `Lean-4` *kernel*. `Lean-4`'s extensions must all be transformed into a corresponding kernel representation, via an extensible elaborator from the surface grammar to the kernel language. `Lean-4` is thus akin to `syntax-spec` with a dependently typed back-end.

Table 2. Compilation times for `cur` and `fowl`, and load times for `smalltt` on a subset of the `smalltt` suite.

Benchmark	`cur` μ	`cur` σ	`fowl` μ	`fowl` σ	`smalltt` μ	`smalltt` σ
`asymp-small`	29.1286 s	0.0501 s	0.7839 s	0.0057 s	0.0024 s	0.0005 s
`asymptotics`	**DNF**	**DNF**	0.9665 s	0.0073 s	0.0251 s	0.0030 s
`conv-eval`	1590.8327 s	2.7064 s	4.8969 s	0.0228 s	0.0017 s	0.0009 s
`stlc-small`	362.1032 s	0.9374 s	1.8499 s	0.0084 s	0.0017 s	0.0002 s
`stlc-small5k`	**DNF**	**DNF**	89.0571 s	0.9266 s	0.0525 s	0.0015 s
`stlc-small10k`	**DNF**	**DNF**	177.9284 s	1.2992 s	0.1031 s	0.0028 s

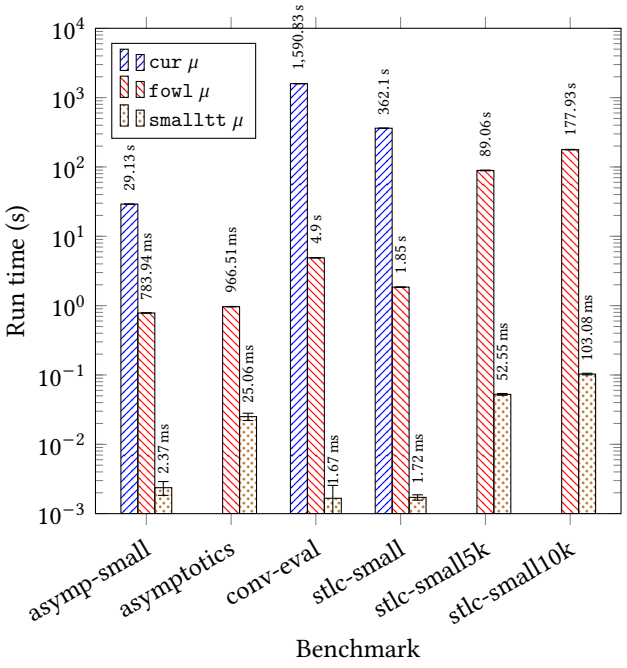

Figure 12. Compilation times for `cur` and `fowl`, and load times for `smalltt` on a subset of the `smalltt` benchmark suite. Omitted bars signify test execution failure.

use an inductively defined vector data type. `asymp-small` does not exist in the `smalltt` benchmark suite, and is a pared down version of the `asymptotics` benchmark. We include `asymp-small` since `cur` terminates without running out of memory on our benchmarking machine.

asymptotics. This benchmark is similar to `asymp-small`, but constructs a 1000-element vector. `smalltt` displays a nearly 10x increase in type checking time compared to `asymp-small`, implying a roughly linear increase in algorithm run time with term depth. `fowl` took barely longer to compile the 1000 element vector than it did to compile `asymp-small`, and thus type checking did not constitute the majority of `asymp-small`'s compilation time. `cur` failed to compile the 1000 element vector with 128GB of available system memory. `asymptotics` is a subset of the `asymptotics` benchmark present in the `smalltt` repository, as it sufficed to demonstrate the divide in `cur`, `fowl` and `smalltt`'s performance.

conv-eval. This benchmark defines inductive naturals (using built-in types in `fowl`), addition and multiplication (using standard library functions in `cur`), then uses these functions to construct larger and larger natural numbers. The version of `conv-eval` in the `smalltt` repository constructs both larger numbers and has additional tree constructions, but `fowl` lacks the ability for us to define inductive data types and both `fowl` and `cur` failed to type check the larger numbers present in the original benchmark. `conv-eval` highlights the importance of *glued evaluation*, a mechanism by which `smalltt` avoids normalization for syntactically equivalent terms. Neither `cur` nor `fowl` implement glued evaluation, and therefore `conv-eval` shows the largest performance divide between `fowl` and `smalltt`.

stlc-small. This benchmark first church-encodes a simply typed lambda calculus (*STLC*) and type checks the function $\lambda x : (\bot \to \bot).\lambda y : \bot.x(x(x(x(x(xy)))))$. The benchmark file has a length on the order of 50-100 lines of code (*loc*). All of `cur`, `fowl`, and `smalltt` perform as expected: `fowl` suffers compared to `smalltt`, since the complexity of church-encoded STLC terms is significant and `fowl` does not avoid the re-computation of normal forms. `cur` performs much worse, as the high term depth requires significant computational effort to reify into `cur` syntax after expansion. Both the `fowl` and `cur` benchmarks are written with explicit context types, as neither perform unification and type inference, unlike `smalltt`. `smalltt` thus performs additional work during type checking. `stlc-small` is unmodified from the `smalltt` benchmark suite.

stlc-small5k. This benchmark executes 96 copies of `stlc-small`, totaling on the order of 5000-10000 loc. `fowl` and `smalltt` both take 50x longer on `stlc-small5k` than on `stlc-small`. `cur` runs out of memory and fails to compile the benchmark. `stlc-small5k` is unmodified from the `smalltt` benchmark suite.

stlc-small10k. This benchmark executes 192 copies of `stlc-small`. `fowl` and `smalltt` both take 2x longer on `stlc-small10k` than on `stlc-small5k`. This is ideal, as

Table 3. Excerpt of reported type checking time of benchmarks in the `smalltt` benchmark suite in Kovács [13], with scaled `fowl` and `cur` compilation times for comparison.

Benchmark	`smalltt`	Agda	Coq	Lean	Idris 2	`fowl` (x0.7)	`cur` (x0.7)
`stlc-small`	0.0030 s	0.1060 s	0.1280 s	0.0730 s	0.5420 s	1.2950 s	217.2620 s
`stlc-small5k`	0.0370 s	4.4450 s	0.7620 s	2.6490 s	6.3970 s	62.3530 s	**DNF**
`stlc-small10k`	0.0720 s	22.8000 s	1.3880 s	5.2440 s	13.4960 s	124.5500 s	**DNF**

we see a linear increase in runtime over `stlc-small5k`. `cur` again runs out of memory and fails to compile the benchmark. `stlc-small10k` is unmodified from the `smalltt` suite.

We conclude that `fowl` performs about 10x slower than most modern dependently typed languages, far better than the 1000x slower performance of `cur`. We do so by comparing our benchmark runs for `smalltt` with the values reported in Kovács [13], which can be seen in Table 3. Our runs for `smalltt` are 30% slower than those reported by Kovács [13], so we include our measured compilation times for `cur` and `fowl` scaled by a factor of 0.7 for comparison. This performance is, however, given *no effort* put towards optimizing `fowl`'s type checking algorithm, only by micro embedding `fowl` into an efficient struct based IR that allows us to perform IR elaboration in place. We expect implementing algorithmic optimizations will get the performance of `fowl` significantly closer to the state of the art, but doing so is left to future work. Crucially, `fowl` has roughly linear, not quadratic, increases in compilation time with increased term complexity, meaning that `fowl`'s term representation is not limiting object language performance, unlike `turnstile+`.

6 Future Work

Currently, our micro embedding approach is a series of design patterns. This is, of course, antithetical to the Scheme ethos. Instead, these design pattern should be wrapped up in some abstractions, ideally implemented using macros (what else?). We conjecture we need three new abstractions, and have sketched an implementation of one.

micros. We can wrap up our mule pattern for implementing micros into its own `define-micro` form for defining micros. This is easily done: `define-micro` should bind an identifiers in the transformer environment to a function from syntax to an arbitrary IR representation, such as structs. It would automatically wrap the output in the mule pattern to smuggle the IR through the macro expander. The micro abstraction should also provide a `expand-micros` form that locally expands until a fully expanded mule, like our `elab-to-structs` implementation in Figure 5 or like the custom expand function provided by `syntax-spec`. The difficulty for us is in generating an extensible syntax class, called `expanded-term` in Figure 5, but this is easily done with another pattern using a parameter that holds an open recursive syntax parser that is extended by each micro. A prototype implementation is available as `syntax/micros` in the software artifact [3].

extensible generics. Our pattern for extending judgements might be abstracted into extensible generics for structs. We would define a form `extensible-generic` for defining new extensible generics, where the generic function indirects through an interposition point (a parameter). Later, the generic could be updated with new functionality, even changing its arity. Our experiments so far suggest this more general abstraction is more difficult to implement than our more limited pattern for extensible judgements. Generics introduce a struct property that is attached to the struct, and this property enables accessing the generic function directly, not through our interposition. The struct property itself cannot be easily interposed upon. It might be that we instead need to introduce a more limited abstraction for extensible judgements.

extensible structs. The next abstraction we need is an extensible struct: structs that can be updated later to include new fields. After updating, old users of the struct should implicitly create the new struct, and matching or projecting from the struct should work seamlessly, so this is not merely struct subtyping. This would be necessary to support the RETT extension properly. We would extend our design pattern for extensible judgements to structs: we would introduce an interposition point for each struct constructor, accessor, and match expander using a parameter. Calls to the "constructors", *e.g.*, would actually dereference the current constructor from the parameter. A prototype is available in the software artifact [3], although it does not support all necessary features, such as extending the matching expander.

Acknowledgments

We gratefully acknowledge the feedback of the anonymous reviewers of this work, and the feedback of Ronald Garcia and Yuanhao Wei.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04207 and CGS-M Award #6563. We also acknowledge the support of the province of British Columbia, BC Graduate Scholarship #6768. Cette recherche a été financée par le Conseil de recherches en

sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-2019-04207 et le prix CGS-M #6563.

Data Availability Statement

The software artifacts, containing the implementation of `fowl`, the test suite, and a prototype implementation of extensible micros and structs, have been made publically available [3].

References

[1] Michael Ballantyne, Mitch Gamburg, and Jason Hemann. 2024. Compiled, Extensible, Multi-language DSLs (Functional Pearl). *Proceedings of the ACM on Programming Languages* 8, ICFP (Aug. 2024), 64–87. doi:10.1145/3674627

[2] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for domain-specific languages. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–29. doi:10.1145/3428297

[3] Sean Bocirnea and William Bowman. 2025. *Fast and Extensible Hybrid Embeddings with Micros Artifact.* doi:10.5281/zenodo.16938371

[4] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (Sept. 2013). doi:10.1017/s095679681300018x

[5] Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2019. Dependent type systems as macros. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 1–29. doi:10.1145/3371071

[6] Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2020. Dependent type systems as macros. *Proceedings of the ACM on Programming Languages (PACMPL)* 4, POPL (2020), 3:1–3:29. doi:10.1145/3371071

[7] Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type Systems As Macros. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3009837.3009886

[8] Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type systems as macros. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3009837.3009886

[9] Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional equality for gradual dependently typed programming. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 165–193. doi:10.1145/3547627

[10] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17, 1-3 (Dec. 1991). doi:10.1016/0167-6423(91)90036-W

[11] Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. 339–347. doi:10.1145/2628136.2628138

[12] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *International Conference on Generative Programming and Component Engineering (GPCE) (GPCE '08)*. 137–148. doi:10.1145/1449913.1449935

[13] András Kovács. 2024. smalltt. https://github.com/AndrasKovacs/smalltt Accessed July 24, 2025.

[14] Shriram Krishnamurthi. 2001. *Linguistic reuse.* Ph. D. Dissertation. Rice University. https://hdl.handle.net/1911/17993

[15] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph. D. Dissertation. Chalmers University of Technology. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

[16] Christine Paulin-Mohring. 1993. Inductive definitions in the system Coq rules and properties. In *Typed Lambda Calculi and Applications*. Springer-Verlag, 328–345. doi:10.1007/bfb0037116

[17] Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. 2019. A reasonably exceptional type theory. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 1–29. doi:10.1145/3341712

[18] David Peter. 2024. hyperfine. https://github.com/sharkdp/hyperfine Accessed July 24, 2025.

[19] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming and Component Engineering (GPCE)*. Association for Computing Machinery, 127–136. doi:10.1145/1868294.1868314

[20] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. 81–92. http://scheme2006.cs.uchicago.edu/13-siek.pdf

[21] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages As Libraries. In *International Conference on Programming Language Design and Implementation (PLDI)*. ACM. doi:10.1145/1993498.1993514

[22] Sebastian Andreas Ullrich. 2023. *An Extensible Theorem Proving Frontend.* Ph. D. Dissertation. Karlsruher Institut für Technologie. doi:10.5445/IR/1000161074

[23] Philip Wadler. 1998. The Expression Problem. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt Accessed July 24, 2025.

[24] Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. 2021. Compositional Programming. *ACM Transactions on Programming Languages and Systems* 43, 3 (Sept. 2021), 1–61. doi:10.1145/3460228

Received 2025-07-24; accepted 2025-08-14