Safely Extending Typed Programming Systems From Within (Thesis Proposal)

Cyrus Omar
Computer Science Department
Carnegie Mellon University
comar@cs.cmu.edu

Abstract

We propose a thesis defending the following statement:

> By organizing programming systems around *active type constructors*, library providers can be given the ability to introduce new concrete syntax, type system fragments and editor services. These extensions do not weaken the system's metatheory, cannot interfere and can be reasoned about modularly.

1 Motivation

> *The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem.* – John Reynolds, 1970 [53]

Achieving both simplicity and generality in a programming language and its associated tools – collectively, a *programming system* – requires organizing around a small number of mechanisms that support the realization of many others as libraries. Much progress has been made in the decades since Reynolds' statement above, but a programming system that comprehensively achieves these goals remains elusive, as evidenced by the fact that across all language lineages, new language and tool dialects remain common vehicles for new features and abstractions, both in research and practice.

We observe that these new dialects are often motivated by the desire for new concrete syntax, new type system fragments, or new editor services, all features of programming systems that library providers usually do not have substantial control over. Introducing mechanisms that decentralize control over these features could increase generality, but they must constructed with care, because giving away too much control could weaken the metatheory of the system. Moreover, they must come equipped with *modular reasoning principles* for it to be practical for library clients to rely upon them. Having to manually resolve conflicts between libraries or reestablish important properties of a program any time a new library is imported introduces undesirable complexity.

In this thesis, we aim to design mechanisms that handle these issues by embracing the already widely accepted notion that *type constructors* (or more concisely, *tycons*) can serve as the fundamental organizational unit of a programming system. We show how to delegate control over certain aspects of the concrete syntax, external type system and editor services to user-defined functions associated with type constructors, together forming what we call *active type constructors*. This permits library-based realizations of features that today would need to be built into a system by its central designers. By layering these mechanisms over a fixed typed internal language and constraining these functions appropriately, we retain key metatheoretic properties and arrive at powerful modular reasoning principles.

1.1 Motivating Example: Regular Expressions

To make the use cases we seek to address more concrete, let us begin with a simple example that we will return to throughout this proposal. *Regular expressions* are commonly used to capture patterns in strings (e.g. motifs in DNA sequences) [64]. Programmers who work with regular expressions would benefit from features like these:

1. **Concrete syntax for pattern literals**. An ideal syntax would permit programmers to express patterns in the concise, conventional manner. For example, consider a bioinformatics application: the BisI restriction enzyme cuts DNA when it sees the motif `GCNGC`, where `N` is any base. We would want to express such patterns with syntax like this, using curly braces to splice one pattern into another and parentheses to delimit captured groups:

   ```
   let N : Pattern = <A|T|G|C>
   let BisI : Pattern = <GC({N})GC>
   ```

   Malformed patterns would result in intelligible parser errors.

2. A **type system** that ensures that key invariants related to regular expressions are statically maintained. It might statically ensure that:

   (a) only patterns, or properly escaped strings, are spliced into a pattern literal, to avoid splicing errors and injection attacks [5, 10]

   (b) out-of-bounds backreferences to a captured group do not occur [58]

   (c) string processing operations do not lead to a string that is malformed, when well-formedness can be captured as membership in a regular language [26, 31]

3. **Editor services** to support syntax highlighting, consulting relevant documentation, interactive testing and pattern extraction from example strings.

No system today provides built-in support for all of the features enumerated above in their strongest form, so library providers must leverage general-purpose mechanisms.

The most common strategy is to ask clients to introduce patterns as strings, deferring their parsing, compilation and use to run-time. This provides only a partial approximation to feature 1 due to syntactic clashes between string escape sequences and regular expressions. Parsing errors due to these clashes have been observed to be difficult for even experienced programmers to diagnose [49]. Moreover, none of the static guarantees can be provided, leading to run-time exceptions (common even in well-tested code [58]), and mistakes that are not caught even at run-time, some of which can lead to security vulnerabilities due to injection attacks [5]. It also introduces performance overhead due to run-time parsing and compilation of patterns, and redundant run-time checks. Staging these computations can shift some of the errors and cost to compile-time, albeit with some additional semantic and syntactic complexity, but the issues related to the fact that a regular expression is not a string (e.g. syntactic clashes and injection attacks) persist.

A more semantically justifiable approach is to dispense with string representations entirely and directly work with an encoding of regular expression patterns using general-purpose constructs like sums and products (e.g. as exposed by functional datatypes or objects), or using an abstract type. This provides feature 2a while sacrificing feature 1 (such an encoding affords only an approximation of the verbose abstract syntax of patterns, not its concrete syntax). Advanced invariants like those described in features 2b and 2c are not directly tracked by such encodings and require solutions beyond those commonly found in modern simply-typed languages (we will return to existing approaches in Sec. **??**).

None of these approaches address the important issue of tooling (feature 3). A number of tools are available online that provide services ranging from simple syntax highlighting to interactive testing and explanations of complex patterns [3] and pattern extraction from examples (e.g. [4]). We have found that programmers benefit substantially from their

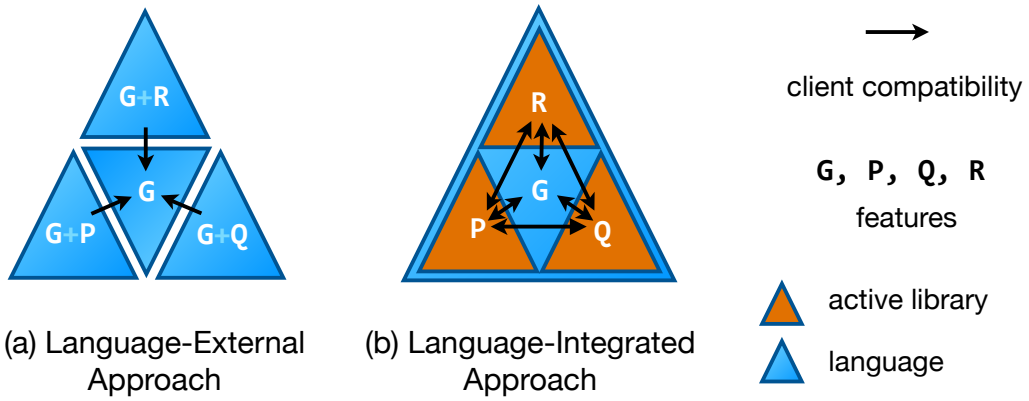

Figure 1: (a) When taking a language-external approach, new features are packaged together into separate languages and tools. (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time logic governing new constructs is expressed within "active libraries".

use [49]. Unfortunately, evidence suggests that tools that must be accessed externally are difficult to discover and are thus used infrequently [44, 13, 49]. We have found that they are also less usable than editor-integrated tools because they require the programmer to switch contexts and cannot make use of the code context [49]. Working with high-level abstractions like these in a "low-level" editor is thus awkward and error-prone, just like attempting to work with high-level abstractions directly in terms of an encoding. For these reasons, we will consider editor services as within the scope of an abstraction specification.

1.2 Language-External Approaches

In situations, like these, where a library-based approach is not satisfying, providers must take a *language-external approach*, either by developing a new system dialect (supported by *compiler generators* [11], *language workbenches* [24], *DSL frameworks* [25], or simply by forking an existing codebase), or by using an extension mechanism for a particular compiler[1], editor or other tool. For example, a researcher interested in providing the regular expression related features just described (let us refer to these collectively as `R`) might design a new system with built-in support for them, perhaps basing it on an existing system containing some general-purpose features (`G`). A different researcher developing a new language-integrated parallel programming abstraction (`P`) might take the same approach. A third researcher, developing a type system for reasoning about units of measure (`Q`) might again do the same. This results in a collection of distinct systems, as diagrammed in Figure 1a. Although often justified as "proofs of concept", this is a rather weak notion because no abstraction will be used entirely in isolation and such a construction gives us no rigorously justifiable reason to believe that it can safely and naturally coexist with others specified or implemented in the same way. That is, one must either use the system containing features `G+R`, `G+P` or `G+Q`. There is no system containing `G`, `R`, `P` and `Q` in other combinations, and merging the systems containing each separately can be non-trivial because there can be serious interference and safety issues, as we will discuss at length throughout this thesis.

Recent evidence indicates that this is one of the major barriers preventing research on programming languages from being driven into practice. For example, developers

[1]Compilers that modify, or allow modification of, the semantics of their base language, rather than simply permitting semantics-preserving optimizations, should be considered a pernicious means for creating new language dialects. That is, some programs that purport to be written in C, Haskell or Standard ML are actually written in compiler-specific dialects of these languages.

prefer high-level language-integrated parallel programming abstractions that provide strong semantic guarantees when all else is equal [18], but library-based approximations are far more widely adopted because "parallel programming languages" privilege only a few chosen abstractions at the language level. This is problematic because different abstractions are seen as more appropriate in different situations [63]. Moreover, parallel programming is rarely the only concern relevant to clients outside of a classroom or research setting. Support for regular expressions, for example, would be simultaneously desirable for processing large amounts of genomic data in parallel, but using these features together in the same compilation unit would be difficult or impossible if implemented using language-external means. Indeed, switching to a "parallel programming language" would likely make it *more* difficult to use regular expressions, as these are likely to be less well-developed in a specialized language than in an established general-purpose language.

Interoperability Even in cases where, for each component of a software system, a programming system considered entirely satisfactory by its developers is available (e.g. a team goes through the trouble of implementing `G+R+P` by reading papers about `G+R` and `G+P` and disentangling orthogonality-related issues), there remains a problem at any interface between components written using a different combination of features. An interface that externally exposes a specialized construct particular to one language (e.g. a function that requires a quantity having a particular unit of measure) cannot necessarily be safely and naturally consumed from another language (e.g. a parallel programming language). Tool support is also lost when calling into different languages.

One strategy taken by proponents of a language-oriented approach [68] to partially address the interoperability problem is to target an established intermediate language and use its constructs as a common language for communication between components written in different languages. Scala [46] and F# [50] are examples of prominent general-purpose languages that have taken this approach, and most DSL frameworks also rely on this strategy. As indicated in Figure 1a, this only enables interoperability in one direction. Calling into the common language becomes straightforward and safe, but calling in the other direction, or between the languages sharing the common target, does not, unless these languages are only trivially different from the common language.

As a simple example with significant contemporary implications, F#'s type system does not admit `null` as a value for any type both defined and used within F# code, but maintaining this sensible internal invariant still requires dynamic checks because the stricter typing rules of F# do not apply when F# data structures are constructed by other languages on the Common Language Infrastructure (CLI) like C# or SML.NET. This is not an issue exclusive to intermediate languages that make regrettable choices regarding `null`, however. The F# type system also includes support for checking that units of measure are used correctly [61, 33], but this more specialized static invariant is left entirely unchecked at language boundaries. Indeed, guidelines for F# suggest that exposing functions that operate over values having units of measure, datatypes or tuples is not recommended when a component "might be used" from another language [61] because it is awkward to construct and consume these from other languages without the convenient primitive operations (e.g. pattern matching) and syntax that F# includes. SML.NET prohibits exposing such types at component boundaries altogether. Moreover, it also cannot naturally consume F# data structures, despite having a rather similar syntax and semantics in most ways (both languages directly descend from ML).

1.3 Language-Integrated Approaches

We argue that, due to these problems with orthogonality and interoperability, taking a language-external approach to realizing a new feature should be considered harmful and avoided whenever possible. The goal of the research being proposed here is to design *language-integrated extension mechanisms* that give providers the ability to define, within

libraries, new features that, like those above, have previously required central planning, so that language-external approaches are less frequently necessary, as illustrated in Figure 1b. Such libraries have been called *active libraries* [67] in that they are not passive clients of features already available in the system. Features implemented within active libraries can be imported individually, unlike features implemented by external means, giving us a potential means to avoid the problems of orthogonality and interoperability.

We must proceed with caution, however: critical issues having to do with safety must be overcome before language-integrated extension mechanisms can be introduced into a system. If too much control over these core features of the system is given to developers, the system may become quite unreliable. Type safety, for example, may not hold if the static and dynamic semantics of the language can be modified or extended arbitrarily from within libraries. Furthermore, even if extensions can be shown not to cause such problems in isolation, there may still be conflicts between extensions that could weaken their semantics, leading to subtle problems that only appear when two extensions are used together, thwarting any attempts to reason modularly about particular programs and the system as a whole. These issues have plagued previous attempts to design language-integrated extensibility mechanisms. We will briefly review some of these attempts below, then return to our approach.

1.3.1 Background

The term *active libraries* was first introduced by Veldhuizen et al. [67, 66] to describe "libraries that take an active role in compilation, rather than being passive collections of subroutines". The authors suggested a number of reasons libraries might benefit from being able to influence the programming system directly, including high-level program optimization, checking programs for correctness against specialized criteria, reporting domain-specific errors and warnings, and "rendering domain-specific textual and non-textual program representations and for interacting with such representations" (anticipating interactions between libraries and tools other than just the compiler).

The first concrete realizations of active libraries in statically typed settings, prompting the introduction of the term, were libraries that performed domain-specific program optimization at compile-time by exploiting language mechanisms that allow for limited compile-time computation. A prominent example in the literature is Blitz++, a library that uses C++ template metaprogramming to optimize compound operations on vectors and matrices by eliminating intermediate allocations [65]. Although this and several other interesting optimizations are possible by this technique, its expressiveness is fundamentally limited because template expansion allows for only the substitution of compile-time constants into pre-written code, and template metaprograms are notoriously difficult to read, write, debug and reason about (see discussion in [54]).

More powerful and direct compile-time *term rewriting mechanisms* available in some languages can also be used for optimization, as well as for introducing specialized error checking logic and implementing new abstractions. For example, typed macros, such as those in MetaML [56], Template Haskell [57] and Scala [12], take full control over all of the code that they enclose. This can be problematic, however, as outer macros can interfere with inner macros. Moreover, once a value escapes a macro's scope, there is no way to rely on the guarantees and features that were available within its scope, because the output of a macro is simply a term in the underlying language (a problem fundamentally related to the problem of relying on a common intermediate language, described in Section 1.2). Thus, macros represent at best a partial solution: they can be used to automate code generation, but not to globally extend the syntax or static guarantees of a language. It can also be difficult to reason about the semantics of code when any number of enclosing macros may be manipulating it (particularly if they can change its type, as in some static macro systems), and to build tools that operate robustly in their presence.

Some term rewriting systems replace the delimited scoping of macros with global

pattern-based dispatch. Xroma (pronounced "Chroma"), for example, allows users to insert custom rewriting passes into the compiler from within libraries [67]. Similarly, the Glasgow Haskell Compiler (GHC) allows providers to introduce custom compile-time term rewriting logic if an appropriate flag is passed in [32]. In both cases, the user-defined logic can dispatch on arbitrary patterns of code throughout the component or program the extension is activated within. But libraries containing such global rewriting logic cannot be safely composed because two different libraries may attempt to rewrite the same piece of code differently. It is also difficult to guarantee that such logic is correct and difficult to reason about code when simply importing a library can change the semantics of the program in a highly non-local manner.

Another example of an active library approach to extensibility with non-local scope is SugarJ [21] and other languages generated by Sugar* [22], like SugarHaskell [23]. These languages permit libraries to extend the base syntax of the core language in a nearly arbitrary manner, and these extensions are imported transitively throughout a program. Unfortunately, this flexibility again means that extensions are not safely composable. For example, a library that defines a literal syntax for HTML would conflict with another that defines a literal syntax for XML because they define differing semantics for some of the same syntactic forms. If SugarJ was used by two different regular expression engines to provide literal syntax for regular expression patterns, there could easily be conflicts at link-time because both will introduce many of the same syntax but back them with differing implementations. And again, it is difficult to predict what an unfamiliar piece of syntax desugars into, leading to difficulties reading and reasoning about code.

2 Our Approach: Active Type Constructors

To motivate our approach, let us return to our example of regular expressions. We observe that every feature described in Sec. 1.1 relates specifically to how terms classified by a type for regular expression patterns, e.g. `Pattern`, should behave. Feature 1 calls for specialized syntax for its introductory form. Features 2a and 2b relate to which operations on patterns exist and how they should be typechecked. Feature 3 discusses editor services only relevant when editing a term of such a type. Feature 2c relates to the semantics of a related type constructor classifying strings known to be in a regular language.

Indeed, this is a common pattern: the semantics of programming languages (and logics) have long been organized around their types (equivalently, their propositions). For example, Carnap in his 1935 book *Philosophy and Logical Syntax* stated [15]:

> One of the principal tasks of the logical analysis of a given proposition is to find out the method of verification for that proposition.

In two modern textbooks about programming languages, *TAPL* [51] and *PFPL* [29], most chapters describe the syntax, semantics and metatheory of a new type constructor and its associated operators (collectively, a *fragment*) in isolation. Combining the fragments from different chapters into complete languages is, however, a language-external (that is, metamathematical) operation. In *PFPL*, for example, the notation $\mathcal{L}\{\rightarrow\ \texttt{nat}\ \texttt{dyn}\}$ represents a language that combines the arrow ($\rightarrow$), `nat` and `dyn` type constructors[2] and their associated operators. Each of these are defined in separate chapters, and it is generally left unstated that the semantics developed separately can be combined conservatively, i.e. with all the fundamental metatheory left intact, a notion we will refine later. This is intuitively justified by a sense that the rules in each chapter seem "well-behaved" in that they avoid violating the autonomy of other fragments. For example, they do not introduce a new value of a type defined in another fragment, because this would render invalid any conclusions arrived at by induction over value forms for that type.

[2] We can consider base types like `nat` and `dyn` as being trivially indexed type constructors, see Sec. **??**.

If we can somehow formalize this notion of a "fragment" and "autonomy" and internalize it into the language itself, rather than leaving it as a "design pattern" that only informally guides the work of a central language (or textbook) designer, we might achieve a safely extensible language, i.e. one where separately defined embeddings of fragments as libraries can be safely combined. Doing so without limiting expressiveness or weakening the system's metatheory is precisely the topic of this thesis.

2.1 Proposed Contributions

This thesis will introduce several language-integrated extensibility mechanisms, each organized around type constructors, that decentralize control over a different feature of the system. In each case, we will show that the system retains important metatheoretic properties and that extensions cannot violate one another's autonomy, in ways that we will make more precise as we go on. We will also discuss various points related to extension correctness (as distinct from safety, which will be guaranteed even if an incorrect extension is imported). To justify the expressiveness of each approach, we will give a number of examples of non-trivial features that are, or would need to be, built into conventional systems, but that can be expressed within libraries using the mechanisms we introduce. To help us gather a broad, unbiased collection of examples and demonstrate the scope and applicability of our approaches in practice, we will also conduct small empirical studies when appropriate (though the primary contributions of this work are technical).

We begin in Sec. 3 by considering **concrete syntax**. The availability of specialized syntax can bring numerous cognitive benefits [27], and discourage the use of problematic techniques like using strings to represent structured data [10]. But allowing library providers to add arbitrary new syntactic forms to a language's grammar can lead to ambiguities, as described above. We observe that many syntax extensions are motivated by the desire to add alternative introductory forms (a.k.a. *literal forms*) for a particular type constructor. For example, regular expression pattern literals as described in Sec. 1.1 are an introductory form for the `Pattern` type constructor. In the mechanism we introduce, literal syntax is associated directly with a user-defined type constructor and can be used only where an expression of a type it constructs is expected (shifting part of the burden of parsing into the typechecker). This avoids the problem of an extension interfering with the base language or another extension because these grammars are never modified directly. We begin by introducing these *tycon-specific languages (TSLs)* in the context of a simplified variant of a language we are developing called Wyvern. Next, we show how interference issues in the other direction – the base language interfering with the TSL syntax – can be avoided by using a novel layout-delimited literal form. We then develop a formal semantics, basing it on work in bidirectional type systems and typed elaboration semantics (which we together call a *bidirectionally typed elaboration semantics*). Using this semantics, we introduce a mechanism that statically prevents a third form of interference: unsafe variable capture and shadowing (a form of *hygiene*).

Wyvern has an extensible concrete syntax but a fixed semantics based around a simple variation on sum and product types. These are of course simple and highly general, and implementation techniques for them are well-developed, but there remain situations where providers may wish to extend the type system of a language directly by introducing more specialized static semantics. Examples of language dialects motivated by a need for this level of control abound in the research literature. For example, to implement the features in Sec. 1.1, new logic must be added to the type system to statically track information related to backreferences (feature 2b, see [58]) or to execute a decision procedure for language inclusion when determining whether a coercion requires a run-time check (feature 2c, see [26, 31]). We discuss more examples from the literature in Sec. **??**. To support these more advanced use cases in a decentralized manner, we next develop mechanisms for implementing **type system** extensions. We begin with a core calculus, $@\lambda$, in Sec. 5, then design a "practical" implementation, `@lang`, as a library

inside Python in Sec. 6 (which serves as an interesting challenge because Python begins with an initially quite impoverished static semantics). Both are organized around a bidirectionally typed translation semantics (which differs from an elaboration semantics like that used for Wyvern in that the external and internal languages have different type systems). We discuss issues including type safety (using techniques borrowed from the typed compilation literature), decidability and conservativity – that embeddings of a type system can be shown correct modularly, guaranteed using a form of "internal" type abstraction controlled by a simple static effect system.

Finally, in Sec. 7, we show an example of a novel class of **editor services** that can be implemented within libraries, introducing a technique we call *active code completion*. Code completion is a common editor service (found in editors like Vim and Emacs as well as in the editor components of development environments like Eclipse) that helps programmers by providing a menu of code snippets based on the surrounding code context. This is a useful but rather generic user interface. There are a variety of tools in the literature and online that help programmers generate code snippets using alternative, more specialized user interfaces. For example, a regular expression workbench helps programmers write regular expression patterns more easily (e.g. [2, 4]). A color chooser can be considered a specialized user interface for creating an expression of type `Color`. Active code completion brings these kinds of type-specific code generation interfaces, which we call *palettes*, into the editor, and allows library providers to associate them with their own type constructors. Clients discover and invoke palettes from the standard code completion menu, populated according to the expected type at the cursor (a protocol similar to the one we use for syntax extensions in Wyvern). When the interaction between the client and the palette is complete, the palette produces an elaboration of the type it is associated with based on the information received from the user (the reader may wish to skip ahead to Fig. 20 in Sec. 7 for an example). Using several empirical methods, including a large developer survey, we examine the expressive power of this approach and develop design criteria. We then develop an active code completion system called Graphite. Using Graphite, we implement a palette for working with regular expressions and conduct a small study that demonstrates the usefulness of type-specific editor services.

Taken together, this work aims to demonstrate that "actively typed" mechanisms can be introduced into many different kinds of programming systems to increase generality without the complexities of previous approaches. We approach the problem both by building up from first principles with type-theoretic models, and by developing practical designs and providing realistic contemporary examples. In the future, we anticipate that this work will be unified to provide foundations for an actively typed programming system organized around a minimal, well-specified and mechanically verified core, where nearly every feature is specified and implemented in a decentralized manner.

3 Tycon-Specific Languages (TSLs)

Many abstractions can be seen, semantically, as modes of use of general-purpose constructs like sums and products. For example, lists can be defined using a more general mechanism for defining polymorphic recursive sum types by observing that a list can either be empty, or be broken down into a product of the *head* element and the *tail*, another list. In an ML-like language, this would be written as a datatype:

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

By defining type constructors like `list` in terms of these general purpose semantic mechanisms, we immediately know how to reason about them (here, by structural induction) and examine them (by pattern matching). They are already well optimized by compilers and benefit from general-purpose editor support as well. While this is all quite useful, the associated general-purpose concrete syntax is often less than ideal. For example, few would claim that writing a list of numbers as a sequence of `Cons` cells is convenient:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))))
```

Because lists are a common data structure, many languages build in *literal syntax* for introducing them, e.g. `[1, 2, 3, 4]`. This syntax is semantically equivalent to the general-purpose syntax shown above, but brings cognitive benefits by drawing attention to the content of the list, rather than the nature of the encoding. Using terminology from Green's cognitive dimensions of notations [27], it is more *terse*, *visible* and *maps more closely* to the intuitive notion of a list. Stoy, in discussing the value of good notation, writes [59]:

> *A good notation thus conceals much of the inner workings behind suitable abbreviations, while allowing us to consider it in more detail if we require: matrix and tensor notations provide further good examples of this. It may be summed up in the saying: "A notation is important for what it leaves out."*

List, number and string literals are nearly ubiquitous features of modern languages. Some languages additionally build in specialized notation for other common data structures (like maps, sets, vectors and matrices), data formats (like XML and JSON), query languages (like regular expressions and SQL), markup languages (like HTML and Markdown) and many other types of data. For example, a language with built-in notation for HTML and SQL, supporting type-safe *splicing* via curly braces, might define:

```
let webpage : HTML = <html><body><h1>Results for {keyword}</h1>
  <ul id="results">{to_list_items(query(db,
    SELECT title, snippet FROM products WHERE {keyword} in title)}
  </ul></body></html>
```

as more concise and natural shorthand for:

```
let webpage : HTML = HTMLElement(Dict.empty(), [BodyElement(Dict.empty(),
  [H1Element(Dict.empty(), [TextNode("Results for " + keyword)]),
  ULElement((Dict.add Dict.empty() ("id", "results")), to_list_items(query(db,
    SelectStmt(["title", "snippet"], "products",
      [WhereClause(InPredicate(StringLit(keyword), "title"))]))))]]]
```

When a specialized notation like this is not available, but this equivalent general-purpose notation, which in essence captures only the abstract syntax of languages like HTML and SQL, is unsatisfying, developers typically turn to run-time mechanisms to make constructing data structures more convenient. Among the most common strategies across language paradigms in these situations is to simply use a string representation that is parsed at run-time.

```
let webpage : HTML = parse_html("<html><body><h1>Results for " + keyword + "</h1>
  <ul id=\"results\">" + to_string(to_list_items(query(db, parse_sql(
    "SELECT title, snippet FROM products WHERE '" + keyword + "' in title")))) +
  "</ul></body></html>")
```

Though recovering some of the notational convenience of the literal version, it is still more awkward, requiring explicit conversions to and from structured representations (`parse_html` and `to_string`, respectively) and escaping when the syntax of the language interferes with the syntax of string literals (line 2). Code like this also causes a number of problems beyond cognitive load. Because parsing occurs at run-time, syntax errors will not be discovered statically, causing potential problems in production scenarios. Run-time parsing also incurs performance overhead, particularly relevant when code like this is executed often (as on a heavily-trafficked website). But the most serious issue with this code is that it is fundamentally insecure: it is vulnerable to cross-site scripting attacks (line 1) and SQL injection attacks (line 3). For example, if a user entered the keyword `'; DROP TABLE products --`, the entire product database could be erased. These attack vectors are considered to be two of the most serious security threats on the web today [5]. Although developers are cautioned to sanitize their input, it can be difficult to verify that this was done correctly throughout a codebase. The best way to avoid these problems today is to avoid strings and insist on structured representations, despite the inconvenient notation.

Unfortunately, situations like this, where maintaining strong correctness, performance and security guarantees entails significant syntactic overhead, causing developers to turn to worse solutions that are more convenient, are quite common. To emphasize this, let us return to our running example of pattern literals. A small regular expression like `(\d\d):(\d\d)\w*((am)|(pm))` might be written using general-purpose notation as:

```
Seq(Group(Seq(Digit, Digit), Seq(Char(":"), Seq(Group(Seq(Digit, Digit)),
  Seq(ZeroOrMore(Whitespace), Group(Or(Group(Seq(Char("a"), Char("m"))),
  Group(Seq(Char("p"), Char("m"))))))))))
```

This is clearly more cognitively demanding, both when writing the regular expression and when reading it. Among the most common strategies in these situations, for users of any kind of language, is again to simply use a string representation that is parsed at run-time:

```
rx_from_str("(\\d\\d):(\\d\\d)\\w*((am)|(pm))")
```

This is problematic, for all of the reasons described above: excessive conversions between representations, interference issues with string syntax, correctness problems, performance overhead and security issues.

Today, supporting new literal syntax within an existing programming language requires the cooperation of the language designer. This is primarily because, with conventional parsing strategies, not all notations can unambiguously coexist, so a designer is needed to make choices about which syntactic forms are available and what their semantics should be. For example, conventional notations for sets and maps are both delimited by curly braces. When Python introduced set literals, it chose to distinguish them based on whether the literal contained only elements (e.g. `{3}`), or key-element pairs (`{"x": 3}`). But this causes an ambiguity with the syntactic form `{ }` – should it mean an empty set or an empty map (called a dictionary in Python)? The designers of Python chose the latter interpretation (for backwards compatibility reasons).

So although languages that allow providers to introduce new syntax from within libraries appear to hold promise for the reasons described above, enabling this form of extensibility is non-trivial because there is no longer a central designer making decisions about such ambiguities. In most existing related work, the burden of resolving ambiguities falls to clients who have the misfortune of importing conflicting extensions. For example, SugarJ [21] and other extensible languages generated by Sugar* [22] allow providers to nearly arbitrarily extend the base syntax of the host language with new forms, like set and map literals. These new forms are imported transitively throughout a program. To resolve syntactic ambiguities that arise, clients must manually augment the composed grammar with new rules that allow them to choose the correct interpretation explicitly. This is both difficult to do, requiring a reasonably thorough understanding of the underlying parser technology (in Sugar*, generalized LR parsing) and increases the cognitive load of using the conflicting notations (e.g. both sets and dictionaries) in the same file because disambiguation tokens must be used. These kinds of conflicts occur in a variety of circumstances: HTML and XML, different variants of SQL, JSON literals and dictionaries, or simply different implementations ("desugarings") of the same specialized syntax (e.g. two regular expression engines). Techniques that limit the kinds of syntax extensions that can be expressed, to guarantee that ambiguities cannot occur, simply cannot express these kinds of examples as-is (e.g. [55]).

In this work, we will describe an alternative parsing strategy that avoids these problems by shifting responsibility for parsing certain *generic literal forms* into the typechecker. The typechecker, in turn, defers responsibility to user-defined types, by treating the body of the literal as a term of the *type- or type-constructor-specific language (TSL)* associated with the type it is being checked against. The TSL is responsible for rewriting this term to ultimately use only general-purpose notation. This strategy avoids the problem of conflicting syntax, because neither the base language nor TSLs are ever extended directly. It also permits semantic flexibility – the meaning of a form like `{ }` can differ depending on its type, so it is safe to use it for empty sets, maps and other data structures, like JSON literals. This

```
let imageBase : URL = <images.example.com>
let bgImage : URL = <%imageBase%/background.png>
new : SearchServer
  def resultsFor(searchQuery, page)
    serve(~) (* serve : HTML -> Unit *)
      >html
        >head
          >title Search Results
          >style ~
            body { background-image: url(%bgImage%) }
            #search { background-color: %darken(`#aabbcc`, 10pct)% }
        >body
          >h1 Results for < HTML.Text(searchQuery)
          >div[id="search"]
            Search again: < SearchBox("Go!")
          < (* fmt_results : DB * SQLQuery * Nat * Nat -> HTML *)
             fmt_results(db, ~, 10, page)
               SELECT * FROM products WHERE {searchQuery} in title
```

Figure 2: Wyvern Example with Multiple TSLs

```
<literal body here, <inner angle brackets> must be balanced>
{literal body here, {inner braces} must be balanced}
[literal body here, [inner brackets] must be balanced]
`literal body here, ``inner backticks`` must be doubled`
'literal body here, ''inner single quotes'' must be doubled'
"literal body here, ""inner double quotes"" must be doubled"
12xyz (* no delimiters necessary for number literals; suffix optional *)
```

Figure 3: Inline Generic Literal Forms

frees these common notations from being tied to the variant of a data structure built into a language's standard library, which sometimes does not provide the exact semantics that a programmer needs (for example, Python dictionaries do not preserve key insertion order).

3.1 Wyvern

We specify our work as a variant of a new programming language being developed by our group called Wyvern [45]. To allow us to focus on the essence of our proposal, the variant of Wyvern we will describe in this thesis is simpler than the variant previously described: it is purely functional (there are no effects other than non-termination) and it does not enforce a uniform access principle for objects (fields can be accessed directly). Objects can thus be thought of as recursive labeled products with support for simple methods (functions that are automatically given a self-reference) for convenience. We also add recursive labeled sum types, which we call *case types*, that are quite similar to datatypes in ML. One can refer to the version of the language described in this thesis as *TSL Wyvern*. TSL Wyvern has a layout-sensitive concrete syntax, for reasons we will discuss.

3.2 Example: Web Search

We begin in Fig. 2 with an example showing several different TSLs being used to define a fragment of a web application showing search results from a database. Note that for clarity of presentation, we color each character according to the TSL it is governed by. Black represents the base language and comments are in italics.

3.3 Inline Literals

Our first TSL appears on the right-hand side of the variable binding on line 1. The variable `imageBase` is annotated with its type, `URL`. This is a named object type[3] declaring several fields representing the components of a URL: its protocol, domain name, port, path and so on (not shown). We could have created a value of type `URL` using general-purpose notation:

```
let imageBase : URL = new
  val protocol = "http"
  val subdomain = "images"
  (* ... *)
```

This is tedious. Because the `URL` type has a TSL associated with it, we can instead introduce precisely this value using conventional notation for URLs by placing it in the *body* of a *generic literal*, `<images.example.com>`. Any other delimited form in Fig. 3 could equivalently be used if the constraints shown are obeyed. The type annotation on `imageBase` implies that this literal's *expected type* is `URL`, so the body of the literal (the characters between the angle brackets, in blue) will be governed by the `URL` TSL during the typechecking phase. This TSL specifies how to parse the body to produce a Wyvern abstract syntax tree (AST) that explicitly instantiates a new object of type `URL` using general-purpose notation as if the above had been written directly. We will return to how this works shortly.

In addition to supporting conventional notation for URLs, this TSL supports *splicing* another Wyvern expression of type `URL` to form a larger URL. The spliced term is delimited by percent signs, as seen on line 2 of Fig. 2. The TSL parses code between percent signs as a Wyvern expression, using its abstract syntax tree (AST) to construct an AST for the expression as a whole. A string-based representation of the URL is never constructed. Note that the delimiters used to go from Wyvern to a TSL are controlled by Wyvern while the TSL controls how to return to Wyvern.

3.4 Layout-Delimited Literals

On line 5 of Fig. 2, we see a call to a function `serve` (not shown) which has type `HTML -> Unit`. Here, `HTML` is a user-defined *case type*, having cases for each HTML tag as well as some other structures, like text nodes and sequencing. Declarations of some of these cases can be seen on lines 2-6 of Fig. 4 (note that TSL Wyvern also includes simple product types for convenience, written `T1 * T2`). We could again use Wyvern's general-purpose introductory form for case types, e.g. `BodyElement((attrs, child))`. But, as discussed above, using this syntax can be inconvenient and cognitively demanding. Thus, we associate a TSL with `HTML` that provides a simplified notation for writing HTML, shown being used on lines 6-18 of Fig. 2. This literal body is layout-delimited, rather than delimited by explicit tokens as in Fig. 3, and introduced by a form of *forward reference*, written ~ ("tilde"), on the previous line. Because the forward reference occurs in a position where the expected type is `HTML`, the literal body is governed by that type's TSL. The forward reference will be replaced by the general-purpose term, of type `HTML`, generated by the TSL during typechecking. Because layout was used as a delimiter, there are no syntactic constraints on the body, unlike with inline literals (Fig. 3). For HTML, this is quite useful, as all of the inline forms impose constrains that would cause conflict with some valid HTML.

3.5 Implementing a TSL

Portions of the implementation of the TSL for `HTML` are shown on lines 8-15 of Fig. 4. A TSL is associated with a named type, forming an *active type*, using a more general mechanism for associating a pure, static value with a named type, called its *metadata*. Metadata is introduced as shown on line 8 of Fig. 4. Type metadata, in this context,

[3] In our example we do not make use of parameterized types (i.e. `list`), so types and type constructors are indistinguishable, and we will simply use "types" for concision.

```
casetype HTML
  Empty
  Seq of HTML * HTML
  Text of String
  BodyElement of Attributes * HTML
  StyleElement of Attributes * CSS
  (* ... *)
  metadata : HasTSL = new
    val parser = ~
      start <- '>body'= attributes> start>
        fn attrs, child => `HTML.BodyElement((%attrs%, %child%))`
      start <- '>style'= attributes> EXP>
        fn attrs, e => `HTML.StyleElement((%attrs%, %e%))`
      start <- '<'= EXP>
        fn e => `%e%`
```

Figure 4: A Wyvern case type with an associated TSL.

```
objtype HasTSL
  val parser : Parser
objtype Parser
  def parse(ps : ParseStream) : ParseResult
  metadata : HasTSL = new
    val parser = (* parser generator *)
casetype ParseResult
  OK of Exp * ParseStream
  Error of String * Location
```

```
casetype Exp
  Var of ID
  Lam of ID * Type * Exp
  Ap of Exp * Exp
  Tuple of Exp * Exp
  ...
  Spliced of ParseStream
  metadata : HasTSL = new
    val parser = (* quasiquotes *)
```

Figure 5: Some of the types included in the Wyvern prelude. They are mutually defined.

is comparable to class annotations in Java or attributes in C#/F# and internalizes the practice of writing metadata using comments, so that it can be checked by the language and accessed programmatically more easily. This can be used for a variety of purposes – to associate documentation with a type, to mark types as being deprecated, and so on.

For the purposes of this work, metadata values will always be of type `HasTSL`, an object type that declares a single field, `parser`, of type `Parser`. The `Parser` type is an object type declaring a single method, `parse`, that transforms a `ParseStream` extracted from a literal body to a Wyvern AST. An AST is a value of type `Exp`, a case type that encodes the abstract syntax of Wyvern expressions. Fig. 5 shows portions of the declarations of these types, which live in the Wyvern *prelude* (a collection of types that are automatically loaded before any other).

Notice, however, that the TSL for `HTML` is not provided as an explicit `parse` method but instead as a declarative grammar. A grammar is a specialized notation for defining a parser, so we can implement a more convenient grammar-based parser generator as a TSL associated with the `Parser` type. We chose the layout-sensitive formalism developed by Adams [7] – Wyvern is itself layout-sensitive and has a grammar that can be written down using this formalism, so it is sensible to expose it to TSL providers as well. Most aspects of this formalism are completely conventional. Each non-terminal (e.g. `start`) is defined by a number of disjunctive productions, each introduced using `->`. Each production defines a sequence of terminals (e.g. `'>body'`) and non-terminals (e.g. `start`, or one of the built-in non-terminals `ID`, `EXP` or `TYPE`, representing Wyvern identifiers, expressions and types, respectively). Unique to Adams grammars is that each terminal and non-terminal in a production can also have an optional *layout constraint* associated with it. The layout constraints available are `=` (meaning that the leftmost column of the annotated term must be aligned with that of the parent term), `>` (the leftmost column must be indented further) and `>=` (the leftmost column *may* be indented further).

Each production is followed, in an indented block, by a Wyvern function that generates a value given the values recursively generated by each of the n non-terminals it contains, ordered left-to-right. For the starting non-terminal, always called `start`, this function

must return a value of type `Exp`. User-defined non-terminals might have a different type associated with them (not shown). Here, we show how to generate an AST using general-purpose notation for `Exp` (lines 13-15) as well as a more natural *quasiquote* style (lines 11 and 18). Quasiquotes are expressions that are not evaluated, but rather reified into syntax trees. We observe that quasiquotes too fall into the pattern of "specialized notation associated with a type" – quasiquotes for expressions, types and identifiers are simply TSLs associated with `Exp`, `Type` and `ID` (Fig. 5). They support the full Wyvern concrete syntax as well as an additional delimited form, written with `%s`, that supports "unquoting": splicing another AST into the one being generated. Again, splicing is safe and structural, rather than based on string interpolation.

3.6 Formalization

A formal and more detailed description can be found in our paper.[4] In particular:

1. We provide a complete layout-sensitive concrete syntax. We show how it can be written without the need for a context-sensitive lexer or parser using an Adams grammar and provide a full specification for the layout-delimited literal form as well as other forms of forward-referenced blocks (for the forms **`new`** and **`case`**`(e)`).

2. We formalize the static semantics, including the literal parsing logic, of TSL Wyvern by combining a bidirectional type system (in the style of Lovas and Pfenning [38]) with an elaboration semantics (in a style similar to Harper and Stone [30]). By distinguishing locations where an expression synthesizes a type from locations where an expression is being analyzed against a known type, we can precisely state where generic literals can and cannot appear and how parsing is delegated to a TSL. The key judgements are of the form:

   $$\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau \qquad \text{and} \qquad \Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau$$

   Expressions, e, elaborate to *internal expressions*, i (which do not contain literals). This occurs simultaneously with typechecking; the first judgement captures situations where type analysis is occurring, the second type synthesis. The typing context, Γ, is standard, and the named type context, Θ, associates named types with their declarations and metadata.

3. A naïve rewriting strategy would be *unhygienic* – it could allow for the inadvertent capture and shadowing of variables around a TSL literal. We show a novel mechanism that ensures hygiene by requiring that the generated AST is closed except for subtrees derived from portions of the user's parse stream that are interpreted as spliced Wyvern expressions or identifiers. We formalize this mechanism by splitting the context in our static semantics.

3.7 Remaining Tasks and Timeline

This work was published at ECOOP 2014 (and received a Distinguished Paper Award) [48]. We have not formally shown how parameterized types work (e.g. we can't yet implement list literal syntax uniformly as a library). I plan to do this as remaining work in the course of completing the dissertation writing. I am currently working with two undergraduates on other extensions to this work, but do not plan to include it in my dissertation.

[4] `https://github.com/wyvernlang/docs/blob/master/ecoop14/ecoop14.pdf?raw=true`

4 Active Typechecking and Translation

In this and the following two sections, we will turn our focus to language-integrated, type-oriented mechanisms for implementing extensions to the static semantics of a simply-typed programming language, to allow providers to express finer distinctions than allowed by the general-purpose constructs (like the recursive labeled sums and products that we included in Wyvern) by introducing new type constructors and, unlike systems that permit only *refinements* of existing types, new operator constructors. In the course of doing so, we will return to a fixed (but flexible) syntax, emphasizing that the contributions in the previous section are orthogonal (combining them directly is left as future work).

4.1 Background

Typed programming languages are often described in *fragments*, each defining separate contributions to a language's concrete syntax, abstract syntax, static semantics and dynamic semantics. In his textbook, Harper organizes fragments around type constructors, each introduced in a different chapter [29]. A language is then identified by a set of type constructors, e.g. $\mathcal{L}\{\rightharpoonup \forall\, \mu\, \mathbf{1}\, \times\, +\}$ is the language that builds in partial function types, polymorphic types, recursive types, nullary and binary product types and binary sum types (its syntax is shown in Figure 6, discussed further below). It is left implied that the metatheory developed separately is conserved when fragments like these are composed.

internal types

$$\tau \quad ::= \quad \tau \rightharpoonup \tau \mid \alpha \mid \forall(\alpha.\tau) \mid t \mid \mu(t.\tau) \mid \mathbf{1} \mid \tau \times \tau \mid \tau + \tau \mid \blacktriangleleft(\sigma)$$

internal terms

$$\begin{array}{rcl} \iota & ::= & x \mid \mathtt{fix}[\tau](x.\iota) \mid \lambda[\tau](x.\iota) \mid \mathtt{ap}(\iota;\iota) \mid \Lambda(\alpha.\iota) \mid \iota\mathtt{[}\tau\mathtt{]} \\ & \mid & \mathtt{fold}[t.\tau](\iota) \mid \mathtt{unfold}(\iota) \mid \mathtt{triv} \mid \mathtt{pair}(\iota;\iota) \mid \mathtt{letpair}(\iota;x,y.\iota) \\ & \mid & \mathtt{inl}[\tau](\iota) \mid \mathtt{inr}[\tau](\iota) \mid \mathtt{case}(\iota;x.\iota;x.\iota) \mid \triangleleft(\sigma) \end{array}$$

internal typing contexts $\Gamma ::= \emptyset \mid \Gamma, x : \tau$

internal type variable contexts $\Delta ::= \emptyset \mid \Delta, \alpha \quad \Theta ::= \emptyset \mid \Theta, t$

term substitutions $\gamma ::= \emptyset \mid \gamma, \iota/x$

type substitutions $\delta ::= \emptyset \mid \delta, \tau/\alpha$

Figure 6: Syntax of $\mathcal{L}\{\rightharpoonup \forall\, \mu\, \mathbf{1}\, \times\, +\}$, our internal language (IL). We write internal types and terms in blue when they may contain the indicated "unquote" forms, discussed later.

Luckily, fragment composition is not an everyday task because fragments like these are "general purpose" – they make it possible to construct *isomorphic embeddings* of many other fragments as "libraries". For example, lists are isomorphic to the polymorphic recursive sum of products $\forall(\alpha.\mu(t.\mathbf{1}+(\alpha\times t))))$. Languages providing datatypes in the style of ML are perhaps most directly oriented around embeddings like this (preserving type disequality requires adding some form of generativity, as ML datatypes also expose).

Unfortunately, situations do sometimes arise when using these fragments to establish an isomorphic embedding that preserves a desirable fragment's static and dynamic semantics (including bounds specified by its cost semantics) is not possible. Embeddings can also sometimes be unsatisfyingly *complex*, as measured by the cost of the extralinguistic computations that are needed to map in and out of the embedding and, if these must be performed mentally, considering cognitive metrics like error message comprehensibility.

If complexity is the issue, abstraction providers have a few options in some settings (discussed in Sec. 4 of our paper draft[5]). When an embedding is not possible, however, or when these options are insufficient, providers are often compelled to form a new *dialect*. To save effort, they may do so by forking existing artifacts or leverage tools like compiler generators or language frameworks (also reviewed in the paper draft). Within the ML lineage, dialects that go beyond $\mathcal{L}\{\rightharpoonup \forall\, \mu\, \mathbf{1}\, \times\, +\}$ abound:

[5] https://github.com/cyrus-/papers/blob/master/att-popl15/att-popl15.pdf

1. **General Purpose Fragments:** A number of variations on product types, for example, have been introduced in dialects: n-ary tuples, labeled tuples, records (identified up to reordering), structurally typed or row polymorphic records [14], records with update and extension operators [35], mutable fields [35], and "methods" (i.e. as in Wyvern [8, 48]).[6] Sum-like types are also exposed in various ways: finite datatypes, open datatypes [37], hierarchically open datatypes [41], polymorphic variants [35] and ML-style exception types. Other generally useful fragments are also built in, e.g. `sprintf` in the OCaml dialect statically distinguishes format strings from strings.

2. **Specialized Fragments:** Fragments that track specialized static invariants to provide stronger correctness guarantees, manage unwieldy lower-level abstractions or control cost are also often introduced in dialects, e.g. for data parallelism [19], distributed programming [43], reactive programming [39], authenticated data structures [40], databases [47] and units of measure [34].

3. **Foreign Fragments:** A safe and natural foreign function interface (FFI) can be a valuable feature (particularly given this proliferation of dialects). However, this requires enforcing the type system of the foreign language in the calling language. For example, MLj builds in a safe FFI to Java [9].

This dialect-oriented state of affairs is, as we have argued, unsatisfying: a programmer can choose either a dialect supporting a principled approach to distributed programming, or one that builds in support for statically reasoning about units of measure, but there may not be an available dialect supporting both. Combining dialects is non-trivial in general, as we will discuss. Using different dialects separately for different components of a program is also untenable: components written in different dialects cannot always interface safely with one another (i.e. an FFI, item 3 above, is needed).

These problems do not arise when a fragment can be exposed as an isomorphic embedding because modern *module systems* can enforce abstraction barriers that ensure that the isomorphism need only be established in the "closed world" of the module. This is useful because it does not impose proof obligations on clients in the "open world".

For situations where an embedding is not evident, however, mechanisms are still needed that make specifying, implementing and reasoning about fragments similarly modular. Such mechanisms could ultimately be integrated directly into a language, blurring the distinction between fragments and libraries and decreasing the need for new dialects. Importing fragments that introduce new syntax and semantics would be as easy as importing a new module is today. In the limit, the community could rely on modularly mechanized metatheory and compiler correctness results, rather than requiring heroic efforts on the part of individual research groups as the situation exists today.

In this portion of our work, we take substantial steps towards this goal by constructing a minimal but powerful core calculus, @λ (the "actively typed" lambda calculus), and then bootstrapping an implementation of it called `@lang` atop an already widely-used language, Python. Both are structured as an *external language* (EL) governed by a typed translation semantics targeting a fixed *internal language* (IL). Rather than building in a monolithic set of external type and operator constructors, however, the typechecking judgement is indexed by a *tycon context*. Each tycon defines the semantics of its associated operators via functions written in a *static language* (SL). Types are values in the SL.

We introduce @λ by example in Sec. **??**, demonstrating its surprising expressive power and outlining its semantics and key metatheoretic properties, including *type safety*, *unicity of typing*, *decidability of typechecking* and properties that relate to composition of tycon definitions: *hygiene*, *stability of typing* and a key modularity result, which we refer to as *conservativity*: any invariants that can be established about all values of a type under *some*

[6] The Haskell wiki notes that "No, extensible records are not implemented in GHC. The problem is that the record design space is large, and seems to lack local optima. And all reasonable variants break backward compatibility. As a result, nothing much happens." [1]

```
type V = LPROD {venue : RSTR /([A-Z]+) \d{4}/}
let v : V = {venue="EXMPL 2015"}
let prefix = "01.0001/" : RSTR /\d{2}\.\d{4}\//
fun paper (title : RSTR /.+/) (id : RSTR /\d+/) =
  v.with(title = title, doi = prefix.concat(id))
let pid : RSTR /\d{3}/ = "005"
let p = paper "M Theory" (pid.coerce[/\d+/])
let c = p#venue#0
c.case(λc : RSTR /EXMPL/.c.!rc; v.!ri)
```

```
letstatic[ty[LPROD](list1[Lbl × Ty] (⌊venue⌋, ty[RSTR](⌊/([A-Z]+) \d{4}/⌋)))](V.
let(asc[V](lit[list1[Lbl] ⌊venue⌋](lit[⌊"EXMPL 2015"⌋](·))); v.
let(asc[ty[RSTR](⌊/\d{2}\.\d{4}\//⌋)](lit[⌊"01.0001/"⌋](·)); prefix.
let(λ[ty[RSTR](⌊/.+/⌋)](title.λ[ty[RSTR](⌊/\d+/⌋)](id.
  targ[with; list2[Lbl] ⌊title⌋ ⌊doi⌋](v; title, targ[concat; ()](prefix; id)))); paper.
let(asc[ty[RSTR](⌊/\d{3}/⌋)](lit[⌊"005"⌋](·)); pid.
let(ap(ap(paper; lit[⌊"M Theory"⌋](·)); targ[coerce; ⌊/\d+/⌋](pid; ·)); p.
let(targ[#; ⌊0⌋](targ[#; ⌊venue⌋](p; ·); ·); c.
targ[case; ()](c; λ[ty[RSTR](⌊/EXMPL/⌋)](c.targ[!rc; ()](c; ·)), targ[!ri; ()](v; ·)))))))))
```

Figure 7: An example external term, e_{ex}, in concrete syntax (left), desugared to abstract syntax (right), with static terms shown in green (in examples only). The notation $\lfloor \ldots \rfloor$ stands for an embedding of the indicated `label`, `/regular expression/`, `"string"` or numeral into the SL, with derived kinds **Lbl**, **Rx**, **Str** and **Nat**, not shown. The definitions of helper functions, e.g. *nil* and *list*n, are omitted for concision.

tycon context (i.e. in some "closed world") are conserved in any further extended tycon context (i.e. in the "open world"). The calculus combines several interesting type theoretic techniques, applying them to novel ends: 1) a bidirectional type system permits flexible reuse of a fixed syntax, in a manner similar to the work described in the previous section on TSLs; 2) the SL serves both as an extension language and as the type-level language; we give it its own statics (i.e. a *kind system*); 3) we use a typed intermediate language and leverage corresponding *typed compilation* techniques, here lifted into the semantics of the EL; 4) we leverage internal type abstraction implicitly as an effect during normalization of the SL to enforce abstraction barriers between type constructors. As a result, conservativity follows from the same elegant parametricity results that underly abstraction theorems for module systems. Like modules, reasoning about these *modular type constructors* does not require mechanized specifications or proofs: correctness issues in the type constructor logic necessarily causes typechecking to fail, so even extensions that are not proven correct can be distributed and "tested" in the wild.

We then continue to `@lang` in Sec. 6. In `@lang`, the static language is Python and type constructors, which in the calculus were tracked by a context and operated as a distinguished "open" datatype, can be implemented as subclasses of a built-in class, `atlang.Type`, leveraging a simple form of inheritance for the same purpose. We support Python's syntax in a natural manner and have implemented a number of full-scale examples of statically-typed language extensions, extending the initially impoverished type system of Python substantially.

5 @λ

5.1 Overview

Programs in @λ are written as *external terms*, e. The concrete syntax of external terms is essentially conventional, as suggested by the left side of the example in Figure 1. It desugars (purely syntactically) to a subtantially more uniform abstract syntax, shown on the right of Figure 1 and specified in Figure 8.

The static and dynamic semantics are specified simultaneously by a *bidirectionally typed translation semantics* targeting an *internal language*, with terms ι. The two key judgements have the form:

$$\Upsilon \vdash_\Phi e \Rightarrow \sigma^+ \rightsquigarrow \iota^+ \quad \text{and} \quad \Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota^+$$

These are pronounced "e (synthesizes / analyzes against) type σ with translation ι under typing context Υ and tycon context Φ". We indicate "outputs" when introducing judgement forms by a *mode annotation*, shown; these are not part of the judgement's syntax. The rules, shown in Figure 16, will be discussed incrementally below. Bidirectional typechecking is also sometimes called *local type inference* [52].

external terms

$$
\begin{array}{rcl}
e & ::= & \mathtt{letstatic}[\sigma](x.e) \mid \mathtt{asc}[\sigma](e) \\
& \mid & x \mid \mathtt{let}(e; x.e) \mid \mathtt{fix}(x.e) \mid \lambda[\sigma](x.e) \mid \mathtt{ap}(e; e) \\
& \mid & \mathtt{lit}[\sigma](\overline{e}) \mid \mathtt{targ}[\mathtt{op}; \sigma](e; \overline{e}) \mid \mathtt{other}[\iota] \qquad \overline{e} ::= \cdot \mid \overline{e}, e
\end{array}
$$

external typing contexts

$$
\Upsilon \quad ::= \quad \emptyset \mid \Upsilon, x \Rightarrow \sigma
$$

Figure 8: Abstract syntax of the external language (EL). Terms in gray are technical devices with no corresponding concrete syntax.

Internal Language @λ relies on a *typed internal language* with support for type abstraction. We will use $\mathcal{L}\{\rightharpoonup \forall\, \mu\, \mathbf{1} \times +\}$ (Figure 6), though our results do not depend on this precise choice of internal type constructors, only on the general forms of the judgements. We assume the internal statics are specified in the standard way by judgements for internal type formation $\Delta\ \Theta \vdash \tau$, typing context formation $\Delta \vdash \Gamma$ and type assignment $\Delta\ \Gamma \vdash \iota : \tau^+$. The internal dynamics are also a standard structural operational semantics with judgements $\iota \mapsto \iota^+$ and ι `val` (cf. [29]).

We also define a syntax for simultaneous substitutions, of terms for variables, γ, and of types for polymorphic type variables, δ, and assume standard judgements ensuring that these are valid with respect to a valid context, $\Delta \vdash \gamma : \Gamma$ and $\vdash \delta : \Delta$. We apply these substitutions to terms, types and typing contexts using the syntax $[\gamma]\iota$, $[\delta]\iota$, $[\delta]\tau$ and $[\delta]\Gamma$ (in some prior work, application of a substitution like this is written using a hatted form, e.g. $\hat{\gamma}(\iota)$; we intend the same semantics but use a notation more consistent with standard substitution, e.g. $[\iota/x]\iota'$). We will omit leading $\emptyset$ and $\cdot$ in examples; in specifications, the former is used for metatheoretic finite mappings, the latter for metatheoretic ordered lists.

This style of specification has some similarities to the Harper-Stone *typed elaboration semantics* for Standard ML [30]. There, however, external and internal terms were governed by a common type system. In @λ, internal types, τ, classify only internal terms, ι. This style may thus also be compared to specifications for the first stage of a type-directed compiler, e.g. the TIL compiler for Standard ML [62], here lifted "one level up" into the language. We will see how the techniques developed to reason about typed compilation are lifted into our semantics to compositionally reason about type safety, which requires that the translation of every well-typed external term is a well-typed internal term.

Kinds and Static Terms The workhorse of our calculus is the *static language*, which itself forms a typed lambda calculus where *kinds*, κ, classify *static terms*, σ. The syntax of the SL is given in Figure 9. We note that the core of the SL is a total functional programming language based on several standard fragments: total functions, quantification over kinds, inductive kinds (constrained by a positivity condition to prevent non-termination), and products and sums (these closely follow [29] so we again omit some details).

The kinding judgement, specified in Figure 18, takes the form $\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_{\mathbf{\Xi}} \sigma : \kappa^+$, where $\mathbf{\Delta}$ and $\mathbf{\Gamma}$ are analagous to Δ and Γ (and analagous well-formedness judgements are defined, also omitted). The natural number n is used as a technical device in our metatheory to prevent terms of the form $\mathbf{ana}[n](\sigma)$ and $\mathbf{syn}[n]$ from arising if not permitted; these forms, discussed later, never need to be written directly (they would have no corresponding concrete syntax), so n can assumed 0 for all user code. $\mathbf{\Xi}$ is described below. The normalization judgement (specified in Figure 19) takes the form $\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma^+ \parallel \mathcal{D}^+\ \mathcal{G}^+$, where $\mathcal{D}$, $\mathcal{G}$ and $\mathcal{C}$ are also technical devices that will be described later; they too can be ignored from the perspective of user code. We write $\sigma \Downarrow \sigma'$ iff $\sigma \parallel \emptyset \cdot \Downarrow_{\rightharpoonup;\emptyset;\emptyset} \sigma' \parallel \emptyset \cdot$. *Static values* are normalized static terms. Normalization can also raise an error (to indicate a type error in an external term, or a problem in a tycon definition, as we will discuss), indicated by the judgement $\sigma \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}$. We omit error propagation rules.

kinds

$$\begin{array}{lcl} \kappa & ::= & \kappa \to \kappa \mid \boldsymbol{\alpha} \mid \forall(\boldsymbol{\alpha}.\kappa) \mid \boldsymbol{k} \mid \mu_{\text{ind}}(\boldsymbol{k}.\kappa) \mid \mathbf{1} \mid \kappa \times \kappa \mid \kappa + \kappa \\ & \mid & \mathbf{Ty} \mid \mathbf{ITy} \mid \mathbf{ITm} \end{array}$$

static terms

$$\begin{array}{lcl} \sigma & ::= & \boldsymbol{x} \mid \lambda \boldsymbol{x}{:}\kappa.\sigma \mid \sigma\ \sigma \mid \Lambda(\boldsymbol{\alpha}.\sigma) \mid \sigma\,[\kappa] \\ & \mid & \mathbf{fold}[\boldsymbol{k}.\kappa](\sigma) \mid \mathbf{rec}[\kappa](\sigma; \boldsymbol{x}.\sigma) \\ & \mid & () \mid (\sigma, \sigma) \mid \mathbf{letpair}(\sigma; \boldsymbol{x}, \boldsymbol{y}.\sigma) \\ & \mid & \mathbf{inl}[\kappa](\sigma) \mid \mathbf{inr}[\kappa](\sigma) \mid \mathbf{case}(\sigma; \boldsymbol{x}.\sigma; \boldsymbol{x}.\sigma) \\ & \mid & \mathbf{ty}[c](\sigma) \mid \mathbf{otherty}[m; \tau] \mid \mathbf{tycase}[c](\sigma; \boldsymbol{x}.\sigma; \sigma) \\ & \mid & \blacktriangleright(\tau) \mid \mathbf{rep}(\sigma) \mid \triangleright(\iota) \\ & \mid & \mathbf{ana}[n](\sigma) \mid \mathbf{syn}[n] \\ & \mid & \mathbf{raise}[\kappa] \\ m, n & ::= & 0 \mid n + 1 \end{array}$$

kinding contexts $\boldsymbol{\Gamma} ::= \emptyset \mid \boldsymbol{\Gamma}, \boldsymbol{x} : \kappa$

kind variable contexts $\boldsymbol{\Delta} ::= \emptyset \mid \boldsymbol{\Delta}, \boldsymbol{\alpha} \quad \boldsymbol{\Theta} ::= \emptyset \mid \boldsymbol{\Theta}, \boldsymbol{k}$

Figure 9: Syntax of the static language (SL).

5.2 External Types are Static Values

External types (or simply *types*) are static values of kind **Ty**. The form $\mathbf{ty}[c](\sigma)$ introduces a type by "applying" a *tycon*, c, to a static term, σ, called the *type index*. There is one built-in tycon, $\rightharpoonup$, classifying possibly partial external functions, discussed below. All other types are constructed by tycons defined in the tycon context and identified by name. We write tycon names in small caps, e.g. LPROD, and use TC as a metavariable ranging over these (Figure 10). We will return to the other introductory form for types, $\mathbf{otherty}[m; \tau]$, which also serves only as a technical device, later.

Two user-defined types are constructed on line 1 of Figure 1: a labeled product type declaring a single field labeled `venue` of type RSTR `/([A-Z]+) \d{4}/`, which classifies *regular strings*, known statically to be in the regular language of the indicated regular expression, which can contain nested parenthesized groups.

Index Kinds The rule (*k-ty*) in Figure 18 requires that the type index must be of the *index kind* of the tycon, given by the *index kind context*, $\boldsymbol{\Xi}$, a simple mapping from tycons to kinds (Figure 10). The initial index kind context, $\boldsymbol{\Xi}_0$, is always included, defining the index kind of $\rightharpoonup$ as $\mathbf{Ty} \times \mathbf{Ty}$. The example index kind context in Figure 14, $\boldsymbol{\Xi}_{\text{ex}}$, specifies that the index kind of LPROD is $\mathbf{List}[\mathbf{Lbl} \times \mathbf{Ty}]$, and the index kind of RSTR is $\mathbf{Rx}$. We assume $\mathbf{List}[\kappa]$, $\mathbf{Lbl}$ and $\mathbf{Rx}$ are embedded into the static language in the standard way. The concrete syntax used on line 1 desugars to static terms of these kinds (this sugar could be defined modularly as *kind-specific syntax* by adapting the technique described in Sec. 3 [48]). If a tycon context, Φ, is valid, it uniquely determines a valid index kind context, as specified in Figure 13 by the judgement $\vdash \Phi \sim \boldsymbol{\Xi}^+$, detailed below. In our example, $\vdash \Phi_{\text{ex}} \sim \boldsymbol{\Xi}_{\text{ex}}$.

Type-Level Computation The labeled product type on line 1 is bound to the static variable ***V*** in the remainder of the external term. This desugars to the more general external construct $\mathbf{letstatic}[\sigma](\boldsymbol{x}.e)$, on the right. The rules (*syn-slet*) and (*ana-slet*), shown in Figure 16 (here, the former applies) begin by assigning a kind to σ under the index kind context determined by Φ, then substitute its value into the external term e to continue with synthesis or analysis. Static variables, written in bold, are distinct from external and internal variables. The rule (*n-ty*) in Figure 19 shows that normalizing $\mathbf{ty}[c](\sigma)$ simply involves normalizing σ. Because types are static values but need not be written directly in normal form, this is a form of *type-level computation of higher kind*, though our SL has a more general role than a standard type-level language.

tycons $c ::= \rightharpoonup \mid \text{TC}$

tycon contexts

$\Phi ::= \cdot \mid \Phi, \phi$

tycon defs

$\phi ::= \texttt{tycon}\ \text{TC}\ \{\omega\} : \psi$

opcon structures

$\omega ::= \texttt{rep} = \sigma \mid \omega, \texttt{ana lit} = \sigma \mid \omega, \texttt{syn op} = \sigma$

index kind contexts

$\Xi_0 := \rightharpoonup[\texttt{Ty} \times \texttt{Ty}]$

$\Xi ::= \Xi_0 \mid \Xi, \text{TC}[\kappa]$

tycon sigs

$\psi ::= \texttt{tcsig}[\kappa]\ \{\Omega\}$

opcon sigs

$\Omega ::= \cdot \mid \Omega, \texttt{lit}[\kappa] \mid \Omega, \texttt{op}[\kappa]$

Figure 10: Syntax of tycons and opcons + sigs and contexts.

Type Constructor Contexts A tycon context, Φ, is a list of tycon definitions, ϕ (Figure 10). Each tycon definition is annotated with a *tycon signature*, ψ, which specifies the index kind of the tycon as well as an *opcon signature*, Ω, which relates to the *opcon structure*, ω, discussed below. The tycons used in our example are defined in Figures 11 and 12. In Figure 13, the judgement $\vdash_{\Xi} \phi \sim \text{TC}^{+}[\kappa^{+}_{\text{tyidx}}]$ ensures that their names are unique, their signatures are valid, and that the opcon structure respects the opcon signature.

Type Equivalence To simplify the handling of type equivalence, type index kinds must be *equality kinds*: those for which semantic equivalence coincides with syntactic equality at normal form. We define these by the judgement $\Delta \vdash \kappa\ \texttt{eq}$, appearing as a premise of (*tcsig-ok*) in Figure 13. Equality kinds are similar to equality types as found in Standard ML. The main implication of this choice is that type indices cannot contain static functions

Representations Each external type has a corresponding internal type called its *concrete representation*. Each tycon definition specifies the concrete representations of the types it constructs with a *representation schema*, specified as the first component of the opcon structure. Rule (*oc-rep*) specifies that this is simply a static function mapping type indices to static terms of kind **ITy**, which are introduced by the quotation form $\blacktriangleright(\tau)$. Unquote forms, written $\blacktriangleleft(\sigma)$, normalize away recursively (rules omitted here).

In Figure 11, the representation schema of RSTR simply ignores the type index, mapping all regular string types to the same internal type: a recursive type pairing a string with a list of other regular strings, corresponding to the "pre-extracted" parenthesized groups (internal types **str** and $\texttt{list}[\tau]$ are defined in the usual way, and are distinguished from analagous kinds **Str** and **List** by initial capitalization). This is of course not the only valid choice of representation – one could simply use **str** (increasing the cost of group extraction, discussed below), or use a tree of offsets, or inspect the index to extract the number of groups, permitting the use of nested tuples (in practice, the IL might provide fixed length vectors).

In Figure 9, the representation schema of LPROD constructs a simple nested tuple representation based on the type index (the labels are not needed in the translation). Because the type index itself contains types, the representation schema must refer to the representations of these types. The static language provides the operator $\texttt{rep}(\sigma)$ to extract knowledge about the representation of type σ. The kinding rule (*k-rep*) is simple, but the normalization semantics are more interesting. Rule (*n-rep-conc*) extracts the representation schema and passes it the type index, but it does not always apply. For types constructed by a tycon other than ones named in the *concrete rep whitelist*, $\boxdot$ (a simple list of tycons), an *abstract representation*, is generated. The first time a representation is requested, a fresh internal type variable, α, is generated and the correspondence is recorded in the *abstract rep store*, $\mathcal{D}$, as seen in rule (*n-rep-abs*). Later requests return this type variable. Requesting the representation is a simple effect. Given V from our example and abbreviating the index of the type of the field venue as σ_{rsidx}:

$$\texttt{rep}(V) \parallel \emptyset \cdot \Downarrow_{\rightharpoonup,\text{LPROD};\emptyset;\Phi_{\text{ex}}} \blacktriangleright(\mathbf{1} \times \alpha) \parallel \emptyset, \texttt{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \leftrightsquigarrow \alpha \cdot$$

```
tycon RSTR {
  rep = λtyidx:Rx.▶(μ(s.str × list[s]))
  ana lit = λtyidx:Rx.λtmidx:Str.λargs:List[κarg].
    let aok : 1 = arity0 args in
    letpair (chars, groups) = rmatch tyidx tmidx in
      ▷(fold[s.str × list[s]](
        (◁(str_of_Str chars), ◁(rstrs_of_Strs groups)))),
  syn concat = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    letpair (targ, a) = arity2 args in letpair (aty, atr) = syn a in
    tycase[RSTR](aty; atyidx.(ty[RSTR](rconcat ttyidx atyidx),
      ▷(rsconcat ◁(ttr) ◁(atr))); raise[Ty × ITm]),
  syn # = λttyidx:Rx.λttr:ITm.λtmidx:Nat.λargs:List[κarg].
    let aok : 1 = arity0 args in
    let rg : Rx = rgroupn ttyidx tmidx in
      (ty[RSTR](rg), ▷(rsgroupn ◁(nat_of_Nat tmidx) ◁(ttr))),
  syn coerce = λttyidx:Rx.λttr:ITm.λtmidx:Rx.λargs:List[κarg].
    let aok : 1 = arity0 args in
    let slok : 1 = rsublang tmidx ttyidx in
    let rtr : ITm = rx_of_Rx tmidx in
      (ty[RSTR](tmidx), ▷(rsregroup ◁(rtr) ◁(ttr))),
  syn case = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    letpair (a1, a2) = arity2 args in letpair (a1ty, a1tr) = syn a1 in
    tycase[⇀](a1ty; a1tyidx.letpair(a1tyin, a1tyout) = a1tyidx in
    tycase[RSTR](a1tyin; r.let rtr : ITm = rx_of_Rx r in
    let a2tr : ITm = ana a2 a1tyout in
      (a1tyout, ▷(rscheck[◀(rep(a1tyout))] ◁(rtr) ◁(ttr)
        ◁(a1tr) λ[1](_.◁(a2tr))))
    ; raise[Ty × ITm]); raise[Ty × ITm])
  syn !rc = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    (▷(ty[RSTR](⌊/\d+/⌋)), str_of_Str ⌊"13"⌋)} : ψrstr
```

Figure 11: The definition of the RSTR tycon, ϕ_{rstr}.

The judgement $\vdash_\Phi \mathcal{D} \rightsquigarrow \delta^+ : \Delta^+$ constructs a substitution and corresponding internal type variable context by generating the concrete representations for each stored type. For the above example:

$$\vdash_{\Phi_{\text{ex}}} \emptyset, \mathbf{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \leftrightsquigarrow \alpha \rightsquigarrow \emptyset, \mu(s.\mathbf{str} \times \mathbf{list}[s])/\alpha : \emptyset, \alpha$$

The judgement $\vdash_\Phi \sigma \rightsquigarrow \tau^+$ extracts the concrete representation by placing every tycon in Φ in the whitelist and checking that the representation is well-formed ($\rightharpoonup$ is always in the whitelist; $\mathcal{D}$ may still be non-empty because of $\mathbf{otherty}[m;\tau]$, also discussed later):

$$\mathbf{rep}(V) \parallel \emptyset \cdot \Downarrow_{\rightharpoonup,\text{RSTR},\text{LPROD};\emptyset;\Phi_{\text{ex}}} \blacktriangleright(\mathbf{1} \times \mu(s.\mathbf{str} \times \mathbf{list}[s])) \parallel \emptyset \cdot$$

5.3 External Terms

Variables We can now continue to line 2 of Figure 1, where we bind a term to a variable using the form $\mathbf{let}(e_1; x.e_2)$. The rules (*syn-let*) and (*ana-let*) give an essentially standard semantics. The translation is to lambda application. The concrete representation of the synthesized type for e_1 determines the type annotation. The (*syn-var*) rule is standard. The well-formedness judgement for external typing contexts, $\vdash_\Phi \Upsilon$, ensures that variables map to types (Figure 17). The judgement $\vdash_\Phi \Upsilon \rightsquigarrow \Gamma^+$ translates the types in Υ to their concrete representations (Figure 15).

Functions On lines 4-6, we see a function definition. The syntax and semantics of functions are fixed and built into the external language (so that the semantics can uniformly

```
tycon LPROD {
  rep = λtyidx:List[Lbl × Ty].listrec[Lbl × Ty][ITy] tyidx ►(1)
    (λh:Lbl × Ty.λr:ITy.letpair (_, hty) = h in
      ►(◄(r) × ◄(rep(hty)))),
  ana lit = λtyidx:List[Lbl × Ty].λtmidx:List[Lbl].λargs:List[κarg].
    let inhabited : 1 = uniqmap tyidx in
    listrec3[Lbl × Ty][Lbl][κarg][ITm] tyidx tmidx args ▷(triv)
      λh1:Lbl × Ty.λh2:Lbl.λh3:κarg.λr:ITm.
        letpair (rowlbl, rowty) = h1 in let lok : 1 = lbleq rowlbl h2 in
        let rowtm : ITm = ana h3 rowty in ▷((◁(r), ◁(rwtm))),
  syn # = λttyidx:List[Lbl × Ty].λttr:ITm.λtmidx:Lbl.λargs:List[κarg].
    let aok : 1 = arity0 args in
    letpair (rownum, rowty) = lookup ttyidx tmidx in
      (rowty, prjnth rownum ttr)),
  syn with =
  λttyidx:List[Lbl × Ty].λttr:ITm.λtmidx:List[Lbl].λargs:List[κarg].
    letpair (olist, otr) = listrec2[Lbl][κarg][List[Lbl × Ty] × ITm]
      tmidx args (ttyidx, ttr)
      (λh1:Lbl.λh2:κarg.λr:List[Lbl × Ty] × ITm.
        letpair (rowty, rowtr) = syn h2 in letpair (rlist, rtr) = r in
          (append[Lbl × Ty] rlist (h1, rowty),
          ▷((◁(rtr), ◁(rowtr))))) in
    let inhabited : 1 = uniqmap olist in
    (ty[LPROD](olist), otr)
  syn !ri = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    (▷(ty[RSTR](⌊/\d+/⌋)), ▷(fold[s.str × list[s]](
      (◁(str_of_Str ⌊"oops"⌋), ◁(rstrs_of_Strs nil[Str])))))} : ψlprod
```

Figure 12: The definition of the LPROD tycon, ϕ_{lprod}.

handle the tricky issue of variable binding). Only the argument types need to be provided explicitly; the return type is synthesized, as shown in the rules (*syn-lam*). Rule (*syn-ap*) is standard. Function types are written concretely as $\sigma_1 \rightharpoonup \sigma_2$ and abstractly as $\mathbf{ty}[\rightharpoonup]((\sigma_1, \sigma_2))$, i.e. the index kind of the built-in type constructor $\rightharpoonup$ is $\mathbf{Ty} \times \mathbf{Ty}$ (cf. Ξ_0 in Figure 14). By the semantics of the operators described below, the full type synthesized by *paper*, written concretely, will be:

```
RSTR /..*/ ⇀ RSTR /\d\d*/ ⇀ LPROD {
  venue : RSTR /([A-Z]+) \d+/, title : RSTR /..*/,
  doi : RSTR /\d\d\.\d\d\d\d\/\d\d*/}
```

Like internal functions (and unlike static functions), external functions may be partial, via a **fix**point operator. Rule (*ana-fix*) follows Plotkin's PCF, cf. [29], here defined analytically.

Ascriptions Type ascriptions allow the programmer to explicitly specify the type a term should be analyzed against. For analytic terms appearing in synthetic positions, an ascription is necessary. In the abstract syntax, type ascriptions are always placed directly on terms using the form $\mathbf{asc}[\sigma](e)$ (in the concrete syntax, they may appear on the variable being bound for convenience). The rule (*syn-asc*) in Figure 16 specifies that the ascription must be a closed static term of kind **Ty** under the index kind context determined by the tycon context. It is normalized to a type before proceeding with analysis. This is also an essentially standard rule.

Literals The variable v binds a term introduced *literally*, under an ascription specifying that it should be analyzed against type $\boldsymbol{V}$. Inside this literal, we also see a literal, though with no ascription (it will be analyzed against the type specified in $\boldsymbol{V}$, as discussed below). All literal forms in the concrete syntax desugar to the same abstract form, $\mathbf{lit}[\sigma_{\text{tmidx}}](\overline{e})$.

The *term index*, σ_{tmidx}, captures statically known portions of the literal as a single static term and the *argument list*, $\overline{e}$, captures all external sub-terms. For example, in the labeled product literal, desugaring constructs a list of field labels as the term index (using standard helper functions whose signatures are shown in Figure **??**) and passes in the corresponding field values as arguments. For regular string literals, the string is lifted into the SL as the term index. There are no sub-terms, so the argument list is empty.

Literal terms can only appear in analytic positions in our calculus. More specifically, they are given meaning by the type constructor of the type they are being analyzed against. This tycon's opcon signature, Ω, determines the *literal term index kind* governing σ_{tmidx}. In Figure 14, ψ_{lprod} specifies Ω_{lprod} which in turn specifies the literal term index kind $\mathtt{List}[\mathtt{Lbl}]$, and ψ_{rstr} similarly specifies Ω_{rstr} and $\mathtt{Str}$. The literal index kind must also be an equality kind, as specified by rule (*ocs-lit*) in Figure 13. The rule governing type analysis of literals is (*ana-lit*), shown in Figure 16. It begins by checking the provided term index against this kind (premises 1-4).

Literal Opcon Definitions The next premise of (*ana-lit*) extracts the *literal opcon definition*, σ_{def}, defined using the form $\mathtt{ana\ lit} = \sigma_{\text{def}}$ in the opcon structure, ω, of the tycon definition. This is responsible for deciding the translation of the literal as a function of the type index, the term index and the types and translations of the arguments.

The relevant rule in Figure 13 is (*oc-lit*), which ensurses that σ_{def} has kind $\kappa_{\text{tyidx}} \to \kappa_{\text{tmidx}} \to \mathtt{List}[\kappa_{\text{arg}}] \to \mathtt{ITm}$. The kinds κ_{tyidx} and κ_{tmidx} are determined by the signature as just described. The kind $\mathtt{ITm}$ is analagous to $\mathtt{ITy}$, and has one introductory "quotation" form, $\triangleright(\iota)$. Normalization produces an internal term without such forms (rules (*n-qitm-**) in Figure 19), i.e. a standard IL term. Note that these can only be opaquely composed, not inspected (there is no elim form) and $\mathtt{ITy}$ and $\mathtt{ITm}$ are not equality kinds.

Before discussing arguments and the remaining premises, let us turn to the literal opcon definition for RSTR in Figure 11, which will be called by the rule (*ana-lit*) to derive a translation for the field value on line 1, $e_{\text{venue}} := \mathtt{lit}[\lfloor \texttt{"EXMPL 2015"} \rfloor](\cdot)$:

$$\begin{array}{l}\emptyset \vdash_{\Phi_{\text{ex}}} e_{\text{venue}} \Leftarrow \mathtt{ty}[\text{RSTR}](\lfloor \texttt{/([A-Z]+) \textbackslash d+/} \rfloor) \rightsquigarrow \mathtt{fold}[s.\mathtt{str} \times \mathtt{list}[s]](\\ \quad \mathtt{pair}(\llbracket \texttt{"EXMPL 2015"} \rrbracket; list1\,\mathtt{[}\mu(s.\mathtt{str} \times \mathtt{list}[s])\mathtt{]}\ \mathtt{fold}[s.\mathtt{str} \times \mathtt{list}[s]](\\ \quad\quad \mathtt{pair}(\llbracket \texttt{"EXMPL"} \rrbracket; nil\,\mathtt{[}\mu(s.\mathtt{str} \times \mathtt{list}[s])\mathtt{]}))))\end{array}$$

We write $\llbracket ... \rrbracket$ for the embedding of the indicated string into the IL, and abbreviate the definitions of $\mathtt{str}$, $\mathtt{list}[s]$ and the helper functions for working with lists as above. We refer to the translation as ι_{venue}.

The logic in Figure 11 simply checks that the argument list is empty using the simple helper function ***arity0***, which raises an error if the list is non-empty (rules (*k-raise*) and (*n-raise*); in practice, the provider would here provide an explicit error message to the client). It then calls the helper function ***rmatch*** on the term index to make sure it matches the regular expression provided as the type index, also extracting its groups statically. Finally, it constructs the translation above by calling simple helper functions for lowering static strings and lists into the IL using quotations. We do not show the details; this is, by design, exactly the code one would write in a compiler written in a simply-typed functional language for the branch of the case analysis in the translation logic corresponding to the regular string literal term constructor. Here, we have pulled it into the tycon definition, pushed the case analysis into the semantics, and do not require a separate term constructor by relying on bidirectional typechecking to permit us to share syntax amongst many types.

Arguments Let us now return to the important issue of how the semantics handles arguments by discussing how the literal opcon definition for LPROD in Figure 12 is called

by (*ana-lit*) to derive the following translation (assuming the definition of ***list1***):

$$\sigma_{\text{tmidx}} := \textbf{\textit{list1}}\texttt{[Lbl]}\ \lfloor \text{venue} \rfloor$$
$$\sigma_{\text{tyidx}} := \textbf{\textit{list1}}\texttt{[Lbl} \times \texttt{Ty]}\ (\lfloor \text{venue} \rfloor, \texttt{ty}[\text{RSTR}](\sigma_{\text{rsidx}}))$$
$$\emptyset \vdash_{\Phi_{\text{ex}}} \texttt{lit}[\sigma_{\text{tmidx}}](e_{\text{venue}}) \Leftarrow \texttt{ty}[\text{LPROD}](\sigma_{\text{tyidx}}) \rightsquigarrow \texttt{pair}(\texttt{triv}; \iota_{\text{venue}})$$

Note that we did not here exploit the opportunity to optimize the representation of labeled products of length 1. Our interest here is not in this particular decision, but to ensure that such decisions can be freely explored and have only local implications, like changes in the representation of an abstract type in an ML-like module system.

To call the opcon definition, the semantics needs to construct the *argument interface*, σ_{args}, a list of type $\texttt{List}[\kappa_{\text{arg}}]$. The kind κ_{arg} is defined in Figure 13: it classifies a pair of static functions that serve as hooks into the semantics, permitting the opcon definition to request synthesis or analysis, respectively, for the argument these functions abstract. The judgement $\overline{e} \looparrowright \sigma^+ \looparrowright \mathcal{G}^+; n^+$, specified in Figure 17, constructs a list of such function pairs by wrapping the static operators $\texttt{syn}[n]$ and $\texttt{ana}[n](\sigma)$. The kinding rules (*k-syn*) and (*k-ana*) (and the signatures in Figure **??**) imply that $\emptyset\ \emptyset \vdash^n_{\Xi} \sigma_{\text{args}} : \texttt{List}[\kappa_{\text{arg}}]$, where n is the number of arguments. All user-defined static terms, as we have seen, are always checked with $n = 0$, ensuring that these operators cannot arise directly in well-kinded opcon definitions, avoiding the possibility of "out of bounds" problems. Passing the argument interface into opcon definitions, which were checked with $n = 0$, does not cause kind safety problems due to a simple lemma, provable inductively:

Lemma 1 *If* $\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_{\Xi} \sigma : \kappa$ *and* $n' > n$ *then* $\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^{n'}_{\Xi} \sigma : \kappa$.

The arguments are placed in an *argument store*, $\mathcal{G}$. Initially, $\mathcal{G}_0$ is simply a numbered sequence of the arguments themselves, here $\mathcal{G}_0 := \cdot, 0 \hookrightarrow e_{\text{venue}}$. The normalization semantics for $\texttt{syn}[n]$ and $\texttt{ana}[n](\sigma)$ can have an effect on the argument store.

The literal opcon constructor for LPROD in Figure 12 begins by ensuring that the type the literal is being analyzed against might actually be inhabited by ensuring that there are no duplicate labels in the type index.[7] It then recurses simultaneously over the term index, type index and argument list, checking corresponding labels for equality and requesting analysis of each argument against the corresponding type. The rule (*n-ana*) shows how the argument is extracted from the store and analyzed in the typing context and tycon context provided (by rule (*ana-lit*) here) in the *operation context*, $\mathcal{C}$. If it succeeds, the argument store is updated with the type provided for analysis, the translation and, crucially, a fresh internal variable, x, standing for its translation. The abstract representation of the type provided for analysis is also produced using the whitelist in $\mathcal{C}$, which contains only the tycon of the type the literal is being analyzed against (and the function tycon, which is never held abstract so functions can be used for operations with binding structure, e.g. **case**, below). We write $\mathcal{G} \otimes n \hookrightarrow \ldots$ to indicate that the previous value in the store for n is removed. Because RSTR is not in $\boxdot$, we will thus have:

$$\sigma_{\text{rep}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{args}} \parallel \emptyset\ \mathcal{G}_0 \Downarrow_{\rightharpoonup,\text{LPROD};\emptyset;\Phi_{\text{ex}}} \rhd(\texttt{pair}(\texttt{triv}; x)) \parallel \mathcal{D}\ \mathcal{G}$$
$$\mathcal{D} := \emptyset, \texttt{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \leftrightsquigarrow \alpha$$
$$\mathcal{G} := \cdot, 0 \hookrightarrow e_{\text{venue}} : \texttt{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \rightsquigarrow \iota_{\text{venue}}/x : \alpha$$

Abstract Translation Checking We must now check that this *abstract translation* is *representationally consistent*. The next two premises in (*ana-lit*) generate the abstract representation of the type provided for analysis, as well as its corresponding substitution, δ, and internal type variable context, Δ, as was discussed. The judgement $\mathcal{G} \rightsquigarrow \gamma^+ : \Gamma^+$ generates a term substitution and internal typing context based on $\mathcal{G}$ similarly (Figure

[7] Implementing a $\texttt{Map}[\kappa]$ kind that ensures this by construction, while remaining an equality kind, requires some simple additions to the SL, e.g. abstract equality kinds modeled also on SML, which we omit for concision and to demonstrate this more generally applicable technique.

17). Here, $\mathcal{G} \rightsquigarrow (\emptyset, \iota_{\text{venue}}/x) : (\emptyset, x : \alpha)$ is generated. Finally, the semantics checks the *abstract translation* above against the abstract representation using these contexts, e.g. here $\emptyset, \alpha\ \emptyset, x : \alpha \vdash \mathbf{pair}(\mathbf{triv}; x) : \mathbf{1} \times \alpha$. It is easy to see that this holds. Only after performing this check are the substitutions δ and γ applied. We will show an example where this check fails even when the check would have succeeded if we had applied the substitutions first below and return to the powerful metatheoretic implications of this check in Sec. **??**.

Hygiene Note that the variables in Υ are not made available in Γ when performing this check, guaranteeing *hygienic translation*: the translation must not have any free variables other than those generated implicitly via the argument store (which applying the substitution will eliminate) so inadvertent capture cannot occur. We assume also that substitution application is *capture-avoiding* (i.e. the binding structure of terms is maintained, as can be implemented straightforwardly using a locally nameless representation). Thus, terms can be reasoned about without conditions on the context that translation logic may place them under, i.e. *compositionally*.

Targeted Operations On line 5 of Figure 1, we invoke the *labeled product extension opcon*, **with**, on v to create a new labeled product with additional fields, `title` and `doi`, with types consistent with those shown in the return type of *paper* above. The `doi` field's value is computed by invoking the *regular string concatenation opcon*, **concat**. A number of other similar *targeted operations* are seen elsewhere. These all desugar to the same abstract form, $\mathbf{targ}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e})$, where metavariable **op** ranges over opcon names, σ_{tmidx} again captures all statically known portions of the operation (e.g. the field names on line 5), e_{targ} is the *target* expression, and $\overline{e}$ contains all other external subterms.

In our calculus, all targeted operations are synthetic. The type constructor of the type recursively synthesized for the target is delegated responsibility over the semantics of the targeted operation. It must define an *opcon term index kind* in its opcon signature and an *opcon definition* in its opcon structure matching the name provided, and satisfying the kinding conditions in rules (*ocs-targ*) and (*oc-targ*) of Figure 13. The rule (*syn-targ*) in Figure 16 shows that type synthesis and translation of targeted operations shares much in common with the logic for literals just described. The term index is first checked against the term index kind, then the opcon definition is extracted and an argument interface is constructed as before. We include the target itself as the notional first argument, though for convenience we don't require that each opcon re-synthesize the translation by passing the components in as shown. Whereas literal opcon definitions only had to decide a translation, targeted opcon definitions must decide both a type and a translation, returning a pair of kind $\mathbf{Ty} \times \mathbf{ITm}$. The abstract translation is checked against the abstract representation of the type produced, as above.

We will not describe each opcon in Figures 11 and 12 in detail. Once again, the code is essentially identical to the code one would write in a standard typechecker. The "magic" happens in the dynamics of the SL, only manifesting when there is a problem. We note the following interesting features in these examples:

- In **concat**, the definition uses the elimination form for types, $\mathbf{tycase}[c](\sigma; \boldsymbol{x}.\sigma; \sigma)$, to extract the type index, compute the concatenated regular expression and construct a translation that performs the concatenation. This implies that tycons in our calculus have features of both open sum types (a default case is required) and modules. Tycon indices are not held abstract. In the calculus as given, there is rarely a reason to inspect the index of a different tycon. If we added support for direct calls between opcons (e.g. so that the regular expression could expose an operator that returned a labeled product containing the groups directly), having this ability would become more useful.
- The **coerce** operation supports converting between regular string types known statically to obey a sublanguage relation (cf. [20]). Rearrangement of group

boundaries can also be performed in this way (this requires a regrouping operation in this choice of translation). We invoke it on line 7 in an analytic position, so subsumption is applied. An interesting direction for future work is to add support for describing implicit coercions between types of the same tycon by enriching the subsumption rule.

- The **case** operation supports checked conversions. The first argument must synthesize a function type, where the index of the input type determines the regular expression that the string will dynamically be checked against. The "else branch", which is necessary here but not for a coercion, is analyzed against the return type of this function. Note that on line 9, the else branch will never be taken, but the type system "forgot" the information that could be used to optimize it away.

- Both example tycons define a **#** opcon, though with different index kinds. They are seen being used on line 8 to extract a field and then a subgroup. Like literals, types are used to distinguish the semantics despite the shared syntax. This has similarities to operator overloading, but here the static and dynamic semantics themselves are being overloaded statically.

- It is enlightening to derive the abstract translation and representation produced when typechecking p**#**`venue` on line 8, recalling that p is a labeled product with three fields each of a different regular string type, with `venue` being the first. Assuming **rs** is the concrete representation of regular strings:
  $\iota_{\text{abs}} := \mathbf{letpair}(x; \_, y.\mathbf{letpair}(y; z, \_.z))$
  $\tau_{\text{abs}} := \alpha_1$
  $\delta := \emptyset, \mathbf{rs}/\alpha_1, \mathbf{rs}/\alpha_2, \mathbf{rs}/\alpha_3$
  $\Delta := \emptyset, \alpha_1, \alpha_2, \alpha_3$
  $\gamma := \emptyset, p/x$
  $\Gamma := \emptyset, x : \mathbf{1} \times (\alpha_1 \times (\alpha_2 \times \alpha_3))$

- The **!rc** opcon definition shows an example where the representational consistency check would fail simply because a term of a type other than the one determined by the representation schema of RSTR was generated. The **!ri** opcon definition is more interesting, however: the translation produced matches the concrete type of RSTR, so type safety would not be violated if it were permitted and it would pass the check described above if it were performed post-substitution. But it is quite problematic for reasons of modular reasoning: it claims that the translation has a regular string type, but the string itself does not obey the *value invariant* that RSTR locally maintains: that the string in the translation is actually in the language specified by the type index. Luckily, it does not pass the abstract translation check because it is not *representationally independent* (**fold** cannot have type α).

5.4 Remaining Tasks and Timeline

In our paper draft, which is currently in submission to POPL 2015 (footnote 5 above), we give additional details on the metatheory, showing more precisely how to reason modularly about these type construction definitions. The major remaining tasks are to complete our analysis of the metatheory by more rigorously proving these theorems (and related lemmas) in the form of a longer technical report. To do so, we must reformulate the normalization rules as a small-step semantics. This is the most time-consuming task remaining; I plan to do this over the Fall semester, in the course of revising or resubmitting the paper (to ESOP in mid-October if not accepted to POPL).

Listing 1 [`listing1.py`] An `@lang` compilation script.

```
1  from atlib import record, decimal, dyn, string,
2    string_in, proto, fn, printf
3
4  print "Hello, compile-time world!"
5
6  Details = record[
7    'amount' : decimal[2],
8    'memo' : dyn]
9  Account = record[
10   'name' : string,
11   'account_num' : string_in[r'\d{2}-\d{8}']]
12 Transfer = proto[Details, Account]
13
14 @fn[Transfer, ...]
15 def log_transfer(t):
16   """Logs a transfer to the console."""
17   printf("Transferring %s to %s.",
18     t.amount.to_string, t.name)
19
20 @fn[...]
21 def __main__():
22   account = {name: "Annie Ace",
23     account_num: "00-00000001"} [:Account]
24   log_transfer(({amount: 5.50, memo: None}, account))
25   log_transfer(({amount: 15.00, memo: "Rent payment"},
26     account))
27
28 print "Goodbye, compile-time world!"
```

6 @lang

Listing 1 shows an example of a well-typed `@lang` program for which strong correctness guarantees have been statically maintained using several type constructors defined in libraries. As suggested by line 4, the top level of every `@lang` file can be seen as a *compilation script* written directly in Python, the static language. `@lang` requires no modifications to the language (version 2.6+ or 3.0+) or features specific to the primary implementation, CPython (so `@lang` supports alternative implementations like Jython and PyPy). This choice pays off immediately on the first line: `@lang`'s import mechanism is Python's import mechanism, so Python's build tools (e.g. `pip`) and package repostories (e.g. `PyPI`) are directly available for distributing `@lang` libraries, including those defining type constructors. Here, `atlib` is the "standard library" but does not benefit from special support in `atlang` itself.

6.1 Active Typechecking

Types Types are constructed programmatically in the static language by applying a type constructor to an index: `tycon[idx]`. This is in contrast to many statically-typed languages where types (e.g. datatypes, interfaces) cannot be constructed computationally, but are only declared "literally". In this example, we see several useful types being constructed:

1. On line 6, we construct a functional record type with two (immutable) fields, assigning it to the identifier `Details`. The field `amount` has type `decimal[2]`, which classifies decimal numbers having two decimal places, and `memo` has type `dyn`, classifying dynamic Python values. Syntactically, the index for `record` is provided using borrowed Python syntax for array slices (e.g. `a[min:max]`) to approximate conventional notation for type annotations. The field names are simply static strings (and could be computed rather than written literally, e.g. `'a' + 'mount'`, to support features like *type providers* [60]).

2. On line 9, we construct another record type. The field `name` has type `string` and the field `account_num` has a type classifying strings guaranteed statically to be in the regular language specified by the regular expression pattern provided as the index (written here as a *raw string literal* [6]). This fragment, based on recent work, statically tracks the language an immutable string is in, providing operations like concatenation, demonstrating the expressive power of our approach [26].

3. On line 12, we construct a simple *prototypic object type* [36], `Transfer`, classifying values consisting of a *fore* of type `Details` and a *prototype* of type `Account`. If a field cannot be found in the fore, the type system will delegate (statically) to the type of the prototype. This makes it easy to share the values of the fields of a common prototype amongst many fores, here to allow us to describe multiple transfers to the same account.

Incomplete Types Incomplete types arise from the application of a type constructor to an index containing an ellipsis (a valid array slice form in Python). The static terms `fn[Transfer, ...]` and `fn[...]` seen on lines 14 and 20 are incomplete types. We will discuss how the elided portion of the index (e.g. the return type) is determined below.

Active Type Constructors Type constructors are implemented in `@lang` libraries as Python classes inheriting from `atlang.Type` (classes here serving as Python's form of open sums). We show portions of `atlib.fn` in Fig. 2. Types are instances of these classes. Incomplete types are instead instances of `atlang.IncompleteType`. The compiler controls instantiation by overloading the subscript operator on the `atlang.Type` class object (using Python's *metaclass* mechanism [6]).

Function Definitions To be typechecked by `@lang` and define run-time behavior, function definitions must have an *ascription*, provided by *decorating* them with either a type or, here, an incomplete type (see [6] for a discussion of Python decorators; we again use Python's metaclasses and operator overloading to support the use of a class as a decorator). Ascribed function definitions do not share Python's semantics and indeed the underlying Python function is discarded immediately after its abstract syntax and static environment (its closure and the global dictionary of the Python module it was defined in) have been extracted by the compiler using Python's built-in reflection capabilities and `inspect` and `ast` packages [6]. The `ast` package defines Python's term constructors. For function literals (term constructor `FunctionDef`), the `@lang` compiler delegates to the type constructor of the ascription in one of two ways, depending on the ascription.

If the ascription is an incomplete type, the compiler invokes the class method `syn_idx_FunctionDef`, seen on line 8, with the *compiler context* (an object that provides hooks into the compiler and a storage location for accumulating information during typechecking), the incomplete index and the syntax tree of the function definition. Here, this method 1) checks and normalizes the incomplete index (here, checking that the argument types are indeed types and turning empty and single arguments into a 0- or 1-tuples for uniformity; not shown); 2) adds the argument names and types to a field of the context that tracks the types of local assignments (initializing it first if it is a top-level function, as in our example); 3) asks the compiler, via the method `ctx.check`, to typecheck each statement in the body (discussed below); 4) if the term constructor of the last statement is `ast.Expr` (a top-level expression), then the type of this expression is the return type, otherwise it is `unit`; and finally, 5) a fully specified index (and thus a type) is synthesized for the function as a whole, here equivalent to `fn[Transfer, atlib.unit]`. Note that this choice of using the last expression as the implicit return value (and considering any statement-level term constructor other than `Expr` to have a trivial value) is made in a library, not by the language itself.

Listing 2 A portion of the type constructor `atlib.fn`.

```
1  class fn(atlang.Type):
2    def __init__(self, idx):
3      # called by atlang.Type via the [] operator
4      atlang.Type.__init__(self,
5        self._check_and_norm(idx))
6
7    @classmethod
8    def syn_idx_FunctionDef(cls, ctx, inc_idx, node):
9      arg_types = cls._check_and_norm_inc(inc_idx)
10     if not hasattr(ctx, 'assn_ctx'):
11       ctx.assn_ctx = { }
12     ctx.assn_ctx.update(zip(node.args, arg_types))
13     for stmt in node.body: ctx.check(stmt)
14     last_stmt = node.body[-1]
15     if isinstance(last_stmt, ast.Expr):
16       rty = last_stmt.value.ty
17     else: rty = unit
18     return (arg_types, rty) # fully specified index
19
20   def ana_FunctionDef(self, ctx, node):
21     if not hasattr(ctx, 'assn_ctx'):
22       ctx.assn_ctx = { } # top-level functions
23     ctx.assn_ctx[node.name] = self # recursion
24     ctx.assn_ctx.update(zip(node.args, self.idx[0]))
25     # all but last
26     for stmt in node.body[0:-1]: ctx.check(stmt)
27     last_stmt = node.body[-1]
28     if isinstance(last_stmt, ast.Expr):
29       ctx.ana(last_stmt.value, self.idx[1])
30     elif self.idx[1] == unit or self.idx[1] == dyn:
31       ctx.check(last_stmt)
32     else: raise atlang.TypeError("...", last_stmt)
```

Were the ascription a (complete) type, the compiler would delegate to `fn` by calling an instance method on it, `ana_FunctionDef`, seen on line 20. This method proceeds similarly, but does not need to perform the first step and last steps (and does not need to be a class method) because the index was already determined when the type was constructed, so it can be accessed via `self.idx`. If the final statement is an expression, is analyzed against the provided return type (see below). Otherwise, only `unit` or `dyn` are valid return types (again, a choice made in a library)

Statements The `ctx.check` method is defined by the compiler. As is the pattern in this paper, the compiler delegates to an active type constructor based on the term constructor of the statement being checked. Most statement-level term constructors other than `Expr` simply delegate to the type constructor of the function they are defined within by calling class methods of the form `check_TermCon`, where `TermCon` is a term constructor or combination of term constructors in a few cases where a finer distinction than what Python itself made was necessary, as defined by `ast`.

For example, the class method `check_Assign_Name`, seen in Listing 3, is called for statements of the form `name = expr`, as on line 22 of Listing 1. In this examples, the assignables context (the `assn_ctx` field of the compiler context) is consulted to determine whether the name being assigned to already has a type due to a prior assignment, in which case the expression is analyzed against that type using `ctx.ana`. If not, the expression must synthesize a type, using `ctx.syn`, and a binding is added.

Bidirectional Typechecking of Expressions The methods `ctx.ana` and `ctx.syn` are also defined by the compiler and can be called by type constructors to request type analysis (when the expected type is known) and synthesis (when the type is an "output"),

Listing 3 Some forms in the body of a function delegate to the type constructor of the function they are defined within (via class methods during typechecking).

```
#class fn(atlang.Type): (cont'd)
  @classmethod
  def check_Assign_Name(cls, ctx, stmt):
    x, e = stmt.target.id, stmt.value # see ast
    if x in ctx.assn_ctx: ctx.ana(e, ctx.assn_ctx[x])
    else: ctx.assn_ctx[x] = ctx.syn(e)

  @classmethod
  def syn_Name(cls, ctx, e):
    try: return ctx.assn_ctx[e.id]
    except KeyError:
      try: return self._syn_lift(ctx.static_env[e.id])
      except KeyError: raise atlang.TypeError("...", e)

  @classmethod
  def syn_default_asc_Str(cls, ctx, e): return dyn
```

respectively, for an expression. Once again, the compiler delegates to a type constructor by a protocol that depends on the term constructor, invoking methods of the form `ana_TermCon` or `syn_TermCon`. This represents a form of bidirectional type system (sometimes also called a *local type inference system*) [52], and the standard subsumption principle applies: if analysis is requested and an `ana_TermCon` method is not defined but a `syn_TermCon` method is, then synthesis proceeds and then the synthesized type is checked for equality against the type provided for analysis. Type equality is defined by checking that the two type constructors are identical and that their indices are equal. Two types with different type constructors are never equal, to ensure that the burden of ensuring that typing respects type equality is local to a single type constructor. No form of subtyping or implicit coercion is supported for the purposes of this paper (though we have designed a mechanism that similarly localizes reasoning to a single type constructor, we do not have room to precisely describe it here.)

Names If the term constructor of an expression is `ast.Name`, as when referring to a bound variable or an assignable location, then the type constructor governing the function the term appears in is delegated to via the class method `syn_Name` (names, when used as expressions, must synthesize a type). We see the definition of this method for `fn` starting on line 9 in Listing 3. A name synthesizes a type if it is either in the assignables context or if it is in the static environment. In the former case, the type that was synthesized when the assignment occurred (by `syn_Assign_Name`) is returned. In the latter case, we must lift the static value to run-time. For the purposes of this paper, we support only other typed `@lang` functions, Python functions and classes (which are given type `dyn`, and can only be called with values of type `dyn`) and modules (which are given a *singleton type* – a type indexed by the module reference itself – from which @lang functions and Python functions and classes can be accessed as attributes in the usual way, see below).

Literal Expressions and Ascriptions Python designates literal forms for dictionaries, tuples, lists, strings, numbers and lambda functions. In `@lang`, the type constructor delegated control over terms arising from any of these forms is a function of how the typechecker encounters the term.

If the term appears in an analytic position, the type constructor of the type it is being analyzed against is delegated control. We see this on line 22 of Listing 1: the dictionary literal form appears in an analytic context, here because an explicit type ascription, `[:Account]`, was provided. Note that the ascription again repurposes Python's array slice syntax (the start is, conveniently, optional). The compiler invokes the `ana_Dict` method on the type the literal is being analyzed against, which is defined by its type constructor, here `atlib.record`, shown on line 17 of Listing 4. This method analyzes each field value

Listing 4 A portion of the `atlib.record` type constructor.

```
1  class record(atlang.Type):
2    def __init__(self, idx):
3      # Sig is an unordered mapping from fields to types
4      atlang.Type.__init__(self, Sig.from_slices(idx))
5
6    @classmethod
7    def syn_idx_Dict(self, ctx, partial_idx, e):
8      if partial_idx != Ellipsis:
9        raise atlang.TypeError("...bad index...", e)
10     idx = []
11     for f, v in zip(e.keys, e.values):
12       if isinstance(f, ast.Name):
13         idx.append(slice(f.id, ctx.syn(v)))
14       else: raise atlang.TypeError("...", f)
15     return idx
16
17   def ana_Dict(self, ctx, e):
18     for f, v in zip(e.keys, e.values):
19       if isinstance(f, ast.Name):
20         if f.id in self.idx.fields:
21           ctx.ana(v, self.idx[f.id])
22         else: raise atlang.TypeError("...", f)
23       else: raise atlang.TypeError("...", f)
24     if len(self.idx.fields) != len(e.keys):
25       raise atlang.TypeError("...missing field...", e)
26
27   def syn_Attribute(self, ctx, e):
28     if e.attr in self.idx: return self.idx[e.attr]
29     else: raise atlang.TypeError("...", e)
```

in the literal against the type of the field, extracted from the type index (an unordered mapping from field names to their types, so that type equality is up to reordering). The various literal forms inside the outermost form thus do not need an ascription because they appear in positions where the type they will be analyzed against is provided by `record`. For example, the value of the field `account_num` delegates to `string_in` via the `ana_Str` method (not shown).

An ascription directly on a literal can also be an incomplete type. For example, we can write `[x, y] [:matrix[...]]` or more concisely `[x, y] [:matrix]` instead of `[x, y] [:matrix[i32]]` when we know `x` and `y` synthesize the type `i32`, because the type constructor can use this information to synthesize the appropriate index. The decorator `@fn[Transfer, ...]`, discussed previously, can be seen as a partial type ascription on a statement-level function literal.[8] Lambda expressions support the same ascriptions:

```
(lambda x, y: x + y) [:fn[(i32, i32), ...]]
```

Records support partial type ascription as well, e.g.:

```
{name: "Deepak Dynamic", age: "27"} [:record]
```

The compiler delegates to the type constructor by a class method in these cases (just as we saw above with `fn`). In this case, the class method `syn_idx_Dict`, shown on line 6 of Listing 4, would be called. Because no record index was provided, an index must be synthesized from the literal itself, and thus the values are not analyzed against a type, but must each synthesize a type.

This brings us to the situation where a literal appears in a synthetic position, as the two string literals above do. In such a situation, the compiler delegates responsibility to the type constructor of the function that the literal appears in, asking it to provide a "default ascription" by calling the class method `syn_default_asc_Str`, shown on line 15

[8] In Python 3.0+, syntax for annotating function definitions directly with type-like annotations was introduced, so the entire index can be synthesized.

of Listing 3. In this case, we simply return `dyn`, so the type of the expression above has type `record['name': dyn, 'age' : dyn]`. A different function type constructor might make more precise choices. Indeed, a benefit of using Python as our static language is that it is relatively straightforward to provide a type constructor that allows us to provide different defaults, and control choices like the return semantics, as block-scoped "pragmas" in the static language using Python's `with` statement [6] (details omitted), e.g.

Listing 5 Block-scoped settings can be seen by type constructors.

```
with fn.default_asc(Num=i64, Str=string, Dict=record):
  @fn[...] # : fn[(), record["a": i64, "b": string]]
  def test(): {a: 1, b: "no ascriptions needed!"}
```

Targeted Expressions Expression forms having exactly one sub-expression, like `-e` (term constructor `UnaryOp_USub`) or `e.attr` (term constructor `Attribute`), or where there may be multiple sub-expressions but the left-most one is the only one required, like `e[slices]` (term constructor `Subscript`) or `e(args)` (term constructor `Call`), are called *targeted expressions*. For these, the compiler first synthesizes a type for the left-most sub-expression, then calls either the `ana_TermCon` or `syn_TermCon` methods on that type. For example, we see type synthesis for the field projection operation on records defined starting on line 27 of Listing 4.

Binary Expressions The major remaining expression forms are the binary operations, e.g. `e + e` or `e < e`. These are notionally symmetric, so it would not be appropriate to simply give the left-most one precedence. Instead, we begin by attempting to synthesize a type for both subexpressions. If both synthesize a type with the same type constructor, or only one synthesizes a type, that type constructor is delegated responsibility via a class method, e.g. `syn_BinOp_Add` on line 7 of Listing 6.

If both synthesize a type but with different type constructors (e.g. we want to concatenate a `string` and a `string_in[r]`), then we consult the *handle sets* associated as a class attribute with each type constructor, e.g. `handles_Add_with` on line 6 of Listing 6. This is a set of other type constructors that the type constructor defining the handle set may potentially support binary operations with. When a type constructor is defined, the language checks that if `tycon2` appears in `tycon1`'s handler set, then `tycon1` does *not* appear in `tycon2`'s handler set. This is a very simple modular analysis (rather than one that can only be performed at "link-time"), shown in Listing 7, that ensures that the type constructor delegated control over each binary expression is deterministic and unambiguous without arbitrarily preferring one subexpression position over another. This check is performed by using a metaclass hook (technically, this can be disabled; we assume that clients are not importing potentially adversarial extensions in this work, though lifting this assumption raises quite interesting questions that we hope will be addressed by future work).

6.2 Active Translation

Once typechecking is complete, the compiler enters the translation phase. This phase follows the same delegation protocol as the typechecking phase. Each `check_`/`syn_`/`ana_TermCon` method has a corresponding `trans_TermCon` method. These are all now instance methods, because all indices have been fully determined.

Examples of translation methods for the `fn` and `record` type constructors are shown in Listing 8. The output of translation on our example is discussed in the next subsection. Translation methods have access to the context and node, as during typechecking, and must return a translation, which for our purposes, is simply another Python syntax tree (in practice, we offer some additional conveniences beyond `ast`, such as fresh variable generation and lifting of imports and statements inside expressions to appropriate positions, in the module `astx` distributed with the language). Translation methods can

Listing 6 Binary operations in `atlib.string_in`.

```
class string_in(atlang.Type):
  def __init__(self, idx):
    atlang.Type.__init__(self, self._rlang_of_rx(idx))

  # treats string as string_in[".*"]
  handles_Add_with = set([string])
  @classmethod
  def syn_BinOp_Add(cls, ctx, e):
    rlang_left = self._rlang_from_ty(ctx.syn(e.left))
    rlang_right = self._rlang_from_ty(ctx.syn(e.right))
    return string_in[self._concatenate_langs(
      rlang_left, rlang_right)]
```

Listing 7 For each type constructor definition and binary operator, `atlang` runs a modular handle set check to preclude ambiguity.

```
def check_tycon(tycon):
  for other_tycon in tycon.handles_Add_with:
    if tycon in other_tycon.handles_Add_with:
      raise atlang.AmbiguousTyconError("...",
        tycon, other_tycon) # (other binops analagous)
```

assume that the term is well-typed and the typechecking phase saves the type that was delegated control, along with the type that was assigned, as attributes of each term processed by the compiler. Note that not all terms need to have been processed by the compiler if they were otherwise reinterpreted by a type constructor given control over a parent term (e.g. the field names in a record literal are never treated as expressions, while they would be for a dictionary literal).

6.3 Standalone Compilation

Listing 9 shows how to invoke the `@` compiler at the shell to typecheck and translate then execute `listing1.py`. The `@lang` compiler is itself a Python library and `@` is a simple Python script that invokes it in two steps:

1. It evaluates the compilation script to completion.
2. For any top-level bindings in the environment that are `@lang` functions, it initiates typehecking and translation as described above. If no type errors are discovered, the ordered set of translations are collected (obeying order dependencies) and emitted. If a type error is discovered, an error is displayed.

In our example, there are no type errors, so the file shown in Listing 10 is generated. This file is meant only to be executed, not edited or imported directly. The invariants necessary to ensure that execution does not "go wrong", assuming the extensions were implemented correctly, were checked statically and entities having no bearing on execution, like field names and types themselves, were erased. The dynamic semantics of the type constructors used in the program were implemented by translation to Python:

1. `fn` recognized the function name `__main__` as special, inserting the standard Python code to invoke it if the file is run directly.
2. Records translated to tuples (the field names were not needed).
3. Decimals translated to pairs of integers. String conversion is defined in a "runtime" package with a "fresh" name.

Listing 8 Translation methods for the types defined above.

```
#class fn(atlang.Type): (cont'd)
  def trans_FunctionDef(self, ctx, node):
    x_body = [ctx.trans(stmt) for stmt in node.body]
    x_fun = astx.copy_with(node, body=x_body)
    if node.name == "__main__":
      x_invoker = ast.parse(
        'if __name__ == "__main__": __main__()')
      return ast.Suite([x_fun, x_invoker])
    else: return x_fun

  def trans_Assign_Name(self, ctx, stmt):
    return astx.copy_with(stmt,
      target=ctx.trans(stmt.target),
      value=ctx.trans(stmt.value))

  def trans_Name(self, ctx, e):
    if e.id in ctx.assn_ctx: return astx.copy(e)
    else: return self._trans_lift(
      ctx.static_env[e.id])

#class record(atlang.Type): (cont'd)
  def trans_Dict(self, ctx, e):
    if len(self.idx.fields) == 1:
      return ctx.trans(e.values[0])
    ast_dict = dict(zip(e.keys, e.values))
    return target.Tuple(ctx.trans(target, ast_dict[f])
      for f, ty in self.idx)

  def trans_Attribute(self, ctx, e):
    if len(self.idx.fields) == 1: return ctx.trans(e)
    else: return ast.Subscript(ctx.trans(e.value),
      ast.Index(ast.Num(self.idx.position_of(e.attr))))
```

Listing 9 Compiling `listing1.py` using the `@` script.

```
> @ listing1.py
Hello, compile-time world!
Goodbye, compile-time world!
[@] _atout_listing1.py successfully generated.
> python _atout_listing1.py
Transferring 5.50 to Annie Ace.
Transferring 15.00 to Annie Ace.
```

4. Terms of type `string_in[r"..."]` translated to strings. Checks could here be performed statically.

5. Prototypic objects translated to pairs consisting of the fore and the prototype. Dispatch to the appropriate record based on the field name was static (line 5).

Type Errors Listing 11 shows an example of code containing several type errors. If analagous code were written in Python itself, these could only be found if the code was executed on the first day of the month (and not all of the issues would immediately be caught as run-time exceptions). In `@lang`, these can be found without executing the code (i.e. true static typechecking).

6.4 Interactive Invocation

`@lang` functions over values of type `dyn` can be invoked directly from Python. Typechecking and translation occurs upon first invocation (subsequent calls are cached;

Listing 10 [_atout_listing1.py] The file generated in Listing 9.

```
import atlib.runtime as _at_i0_

def log_transfer(t):
  _at_i0_.print("Transferring %s to %s." %
    (_at_i0_.dec_to_str(t[0][0], 2), t[1][0]))

def __main__():
  account = ("Annie Ace", "00-00000001")
  log_transfer((((5, 50), None), account))
  log_transfer((((15, 0), "Rent payment"), account))
if __name__ == "__main__": __main__()
```

Listing 11 [listing11.py] Lines 7-11 each have a type error.

```
from listing1 import fn, dyn, Account, Details,
  log_transfer
import datetime
@fn[dyn, dyn]
def pay_oopsie(memo):
  if datetime.date.today().day == 1: # @lang to Python
    account = {nome: "Oopsie Daisy",
      account_num: "0-00000000"} [:Account] # (format)
    details = {amount: None, memo: memo} [:Details]
    details.amount = 10 # (immutable)
    log_transfer((account, details)) # (backwards)
print "Today is day " + str(datetime.date.today())
pay_oopsie("Hope this works..") # Python to @lang
print "All done."
```

we assume that the static environment is immutable). We see this occurring when we execute the code in Listing 11 using the Python interpreter in Listing 12.

In future work, we plan to detail how to insert dynamic checks and wrapper objects, defined by active type constructors in a manner similar to how static checks are defined here, so that values that are not of type dyn can be passed in and out of untyped Python code. Because these can be seen as implicit coercions to and from dyn, and we do not aim to introduce this feature here, we omit discussion of this topic. Explicit coercions can be implemented using the mechanisms described above. For example, string_in provides an introductory form that checks a provided string or dyn value dynamically, raising an exception in the case of failure: [raw_input()] : string_in[r"\d+"].

Listing 12 Execution never proceeds into a function with a type error when using @lang for implicit compilation.

```
> python listing11.py
 Today is day 2
 Traceback (most recent call last):
   File "listing11", line 9, in <module>
 atlang.TypeError:
     File "listing11.py", line 7, col 12, in <module>:
       [record] Invalid field name: nome
```

6.5 Remaining Tasks and Timeline

The remaining tasks largely have to do with implementation of some of the mechanisms described above. I will be submitting the work above to some upcoming conference to be decided and will work on the implementation as well as remaining details of other case studies not mentioned here (e.g. the entirety of OpenCL) throughout the Fall semester.

7 Active Code Completion

An abstraction might benefit from support from a variety of tools beyond the compiler. For example, software developers today make heavy use of the code completion support found in modern source code editors [42]. Code completion typically takes the form of a floating menu containing contextually-relevant variables, assignables, fields, methods, types and other code snippets. When an item in the menu is selected, code is inserted immediately, without further input from the developer. By navigating and selecting from this menu, developers are able to avoid many common spelling and logic errors, eliminate unnecessary keystrokes and explore unfamiliar APIs without incurring the mental overhead associated with switching to an external documentation tool.

These systems are difficult to extend: a fixed strategy determines the completions that are available, so library providers cannot directly specify new domain-specific or contextually-relevant logic or provide assistance beyond that which a menu can provide. In this paper we propose a technique called *active code completion* that eliminates these restrictions using active types. This makes developing and integrating a broad array of highly-specialized code generation tools directly into the editor, via the familiar code completion command, significantly simpler.

In this work, we discuss active code completion in the context of object construction in Java because type-aware editors for Java are better developed than those for other languages, because we wish to do empirical studies, and because Java already provides a way to associate metadata with classes. The techniques in this section apply equally well to type-aware editors for any language with a similar mechanism.

For example, consider the following Java code fragment:

```
1  public Color getDefaultColor() {
2      return _
```

If the developer invokes the code completion command at the indicated cursor position (_), the editor looks for a *palette definition* associated with the *type* that the expression being entered is being analyzed against, which in this case is `Color` (due to the return type annotation on the method). If an associated palette is found, a menu item briefly describing this palette is added to the standard code completion menu. When selected, the corresponding palette is shown, replacing the standard code completion menu. Figure 20(a) gives an example of a simple palette that may be associated with the `Color` class[9].

The developer can interact with such palettes to provide parameters and other information related to their intent, and receive immediate feedback about the effect these choices will have on the behavior of the object being constructed. When this interaction is complete, the palette produces an elaboration for insertion at the cursor, of the type (here, class) that the palette was associated with (Figure 20(b)).

This is, in essence, an enriched edit-time variant of the mechanism used in Wyvern. As discussed in Sec. 3, we associate a palette by providing metadata in the form of a class annotation. For example, we might write:

```
1  @GraphitePalette(url="http://cs.cmu.edu/~comar/color-palette.html")
2  class Color { ... }
```

Were a type-aware editor for Wyvern written, it might instead use metadata:

```
1  objtype Color
2    ...
3    metadata : HasPalette = new
4      val palette_url = <http://cs.cmu.edu/~comar/color-palette.html>
```

[9] A video demonstrating this process is available at `http://www.cs.cmu.edu/~NatProg/graphite.html`.

7.1 Design Process

We sought to address the following questions before designing and implementing our active code completion system:

- What *specific* use cases exist for this form of active code completion in a professional development setting?
- What *general* criteria are common to types that would and would not benefit from an associated palette?
- What are some relevant usability and design criteria for palettes designed to address such use cases?
- What capabilities must the underlying active code completion system provide to enable these use cases and user interface designs?

To help us answer these questions, we conducted a survey of 473 professional developers. Their responses, along with information gathered from informal interviews and code corpus analyses, revealed a number of non-trivial functional requirements for palette interfaces as well as the underlying active code completion architecture. Participants also suggested a large number of use cases, demonstrating the broad applicability of this technique. We organized these into several broad categories. Next, we developed Graphite, an Eclipse plug-in that implements the active code completion architecture for the Java programming language, allowing Java library developers to associate custom palettes with their own classes. We describe several design choices that we made to satisfy the requirements discovered in our preliminary investigations and examine necessary trade-offs. Finally, we conducted a pilot lab study with a more complex palette, implemented using Graphite, that assists developers as they write regular expressions. The study provides specific evidence in support of the broader claim that highly-specialized tools that are integrated directly with the editing environment are useful. We conclude that active code completion systems like Graphite are useful because they make developing, deploying and discovering such tools fundamentally simpler.

The primary concerns relevant to this thesis are:

- The palette mechanism should not be tied to a specific editor implementation. We achieve this by using a URL-based scheme for referring to palettes, which are implemented as webpages, which can be embedded into any editor using standard techniques for embedding browsers into GUIs.
- The palette mechanism should not be able to arbitrary access the surrounding source code (for privacy reasons, as identified by survey participants). By using a browser, and only allowing access to highlighted strings, we avoid this problem.
- We provide a mechanism by which users can associate palettes with types externally. This could cause conflicts with palettes that the type defines itself. To resolve this, we give users a choice whenever such situations occur by inserting all relevant palettes into the code completion menu.

7.2 Remaining Tasks and Timeline

This work has been published at ICSE 2012 [49]. The main remaining tasks have to do with specifying it in terms of the bidirectional typing mechanisms in Sec. 3 more directly, rather than informally as in our paper. We plan to do this when writing the dissertation. There are also some pieces of related work that have been published since this paper was accepted that we need to review.

8 Conclusion

In arguing for language extensibility, we are accepting the philosophy of logical pluralism, explained by Carnap as follows [17]:

> Let us grant to those who work in any special fields of investigation the freedom to use any form of expression which seems useful to them. The work in the field will sooner or later lead to the elimination of those forms which have no useful function. Let us be cautious in making assertions and critical in examining them, but tolerant in permitting linguistic forms.

He explained his *principle of tolerance* further as follows [16]:

> In logic, there are no morals. Everyone is at liberty to build his own logic, i.e. his own form of language, as he wishes. All that is required of him is that, if he wishes to discuss it, he must state his methods clearly, and give syntactic rules instead of philosophical arguments.

Carnap formulated his principle of tolerance because he wished to freely explore the consequences of reasoning with different connectives and rules and emphasize that conducting philosophical reasoning by methodical logical analysis did not require one to accept any particular axioms. But he was also a prominent member of the Vienna circle, which sought a "unified science" the purpose of which was "to link and harmonise the achievements of individual investigators in their various fields of science" by the construction of a "constitutive system" [28]. These two programs were in conflict: were the Vienna circle's "constitutive system" to become naïvely tolerant of the inference rules (what Carnap called L-rules) developed by individual investigators, it would find that this would then deny them autonomy of reasoning.Thus, we propose a *principle of reciprocal tolerance*:

> In metalogic, there *are* morals. Everyone is at liberty to build their own logical fragment, i.e. their own language extensions, as long as these can be implemented within a framework that ensures, by formal means instead of philosophical argument, that autonomously derived reasoning principles are conserved when fragments are combined.

By following this principle, we believe that a research program that supports Carnap's logical pluralism within a "constitutive system" of the sort envisioned by the Vienna circle is both possible and practical. The work in this thesis gives first steps towards this goal.

References

[1] GHC/FAQ. http://www.haskell.org/haskellwiki/GHC:FAQ#Extensible_Records.

[2] How to check your RegExps in IntelliJ IDEA 11? http://blogs.jetbrains.com/idea/tag/regexp/.

[3] RegExr : Create & Test Regular Expressions. http://regexr.com/.

[4] txt2re: headache relief for programmers :: regular expression generator. http://txt2re.com/.

[5] OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013.

[6] The Python Language Reference. http://docs.python.org, 2013.

[7] M. D. Adams. Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule. In *POPL 2013*, pages 511–522, New York, NY, USA, 2013. ACM.

[8] J. Aldrich. The power of interoperability: why objects are inevitable. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 101–116. ACM, 2013.

[9] N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *ICFP '99*, pages 126–137, New York, NY, USA, 1999. ACM.

[10] M. Bravenboer, E. Dolstra, and E. Visser. Preventing Injection Attacks with Syntax Embeddings. In *GPCE '07*, pages 3–12, New York, NY, USA. ACM.

[11] R. Brooker, I. MacCallum, D. Morris, and J. Rohl. The compiler compiler. *Annual Review in Automatic Programming*, 3:229–275, 1963.

[12] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[13] D. Campbell and M. Miller. Designing Refactoring Tools for Developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 9:1–9:2, New York, NY, USA, 2008. ACM.

[14] L. Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[15] R. Carnap. *Philosophy and Logical Syntax*. 1935.

[16] R. Carnap. *The Logical Syntax of Language*. 1937.

[17] R. Carnap. Empiricism, semantics, and ontology. *Revue Internationale de Philosophie*, 4:20–40, 1950.

[18] V. Cavé, Z. Budimlić, and V. Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, 2010.

[19] M. Chakravarty, R. Leshchinskiy, S. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18. ACM, 2007.

[20] C. Doczkal, J.-O. Kaiser, and G. Smolka. A constructive theory of regular languages in coq. In G. Gonthier and M. Norrish, editors, *CPP*, volume 8307 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2013.

[21] S. Erdweg, T. Rendel, C. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*.

[22] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12. ACM, 2013.

[23] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of the 2012 Symposium on Haskell*, pages 149–160. ACM, 2012.

[24] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *Software Language Engineering*. 2013.

[25] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[26] N. Fulton. A typed lambda calculus for input sanitation. Senior thesis, Carthage College, 2013.

[27] T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[28] H. Hahn, O. Neurath, and R. Carnap. The scientific conception of the world: The Vienna Circle. 1929.

[29] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[30] R. Harper and C. Stone. A Type-Theoretic Interpretation of Standard ML. In *IN Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[31] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. In *ICFP '00*, 2000.

[32] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[33] A. Kennedy. Dimension types. In *ESOP '94*, pages 348–362. Springer, 1994.

[34] A. Kennedy. Types for units-of-measure: Theory and practice. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009.

[35] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2013.

[36] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, Oct. 1986.

[37] A. Löh and R. Hinze. Open data types and open functions. In *PPDP*, pages 133–144. ACM, 2006.

[38] W. Lovas and F. Pfenning. A bidirectional refinement type system for LF. In *Electronic Notes in Theoretical Computer Science, 196:113–128, January 2008.*, 2008.

[39] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP*, pages 82–93. ACM, 2005.

[40] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated Data Structures, Generically. In *POPL '14*, pages 411–423, New York, NY, USA, 2014. ACM.

[41] T. D. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst*, 26(5):836–889, 2004.

[42] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[43] T. Murphy, VII., K. Crary, and R. Harper. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC'07, pages 108–123, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] E. Murphy-Hill and G. C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 405–414, New York, NY, USA, 2011. ACM.

[45] L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, and J. Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '13, pages 9–16, New York, NY, USA, 2013. ACM.

[46] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. Technical report, EPFL, 2004.

[47] A. Ohori and K. Ueno. Making Standard ML a Practical Database Programming Language. In *ICFP*, ICFP '11, pages 307–319, New York, NY, USA, 2011. ACM.

[48] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.

[49] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active Code Completion. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

[50] R. Pickering. *Foundations of F#*. Apress, 2007.

[51] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[52] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[53] J. C. Reynolds. GEDANKEN - a simple typless language based on the principle of completeness and reference concept. *Comm. A.C.M.*, 13(5), May 1970.

[54] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, JGI '01, pages 1–10, New York, NY, USA, 2001. ACM.

[55] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In M. Hind and A. Diwan, editors, *PLDI*, pages 199–210. ACM, 2009.

[56] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608, 1999.

[57] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[58] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26. ACM, 2012.

[59] J. E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory*. MIT press, 1977.

[60] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Data Driven Functional Programming*, 2013.

[61] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Apress, 2012.

[62] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI '96*, Philadelphia, PA, May 1996.

[63] S. Tasharofi, P. Dinges, and R. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In G. Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer Berlin Heidelberg, 2013.

[64] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[65] T. L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In *Advances in Software tools for scientific computing*, pages 57–87. Springer, 2000.

[66] T. L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, 2004.

[67] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proc. 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[68] M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

$$\boxed{\vdash \Phi \sim \Xi^+}$$

$$\text{TCC-EMP}\ \frac{}{\vdash \cdot \sim \Xi_0} \qquad \text{TCC-EXT}\ \frac{\vdash \Phi \sim \Xi \qquad \vdash_\Xi \phi \sim \text{TC}[\kappa_{\text{tyidx}}]}{\vdash \Phi, \phi \sim \Xi, \text{TC}[\kappa_{\text{tyidx}}]}$$

$$\boxed{\vdash_\Xi \phi \sim \text{TC}^+[\kappa^+]}$$

$$\text{TCDEF-OK}\ \frac{\text{TC} \notin \text{dom}(\Xi) \quad \vdash \psi \sim \kappa_{\text{tyidx}} \quad \vdash_\Xi \text{TC}\ \{\omega\} : \psi}{\vdash_\Xi \textbf{tycon}\ \text{TC}\ \{\omega\} : \psi \sim \text{TC}[\kappa_{\text{tyidx}}]}$$

$$\boxed{\vdash \psi \sim \kappa^+}$$

$$\text{TCSIG-OK}\ \frac{\emptyset \vdash \kappa_{\text{tyidx}}\ \mathsf{eq} \qquad \vdash \Omega}{\vdash \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \sim \kappa_{\text{tyidx}}}$$

$$\boxed{\Theta \vdash \kappa\ \mathsf{eq}}$$

$$\text{KEQ-K}\ \frac{k \in \Theta}{\Theta \vdash k\ \mathsf{eq}} \qquad \text{KEQ-IND}\ \frac{\Theta, k \vdash \kappa\ \mathsf{eq}}{\Theta \vdash \mu_{\text{ind}}(k.\kappa)\ \mathsf{eq}} \qquad \text{KEQ-UNIT}\ \frac{}{\Theta \vdash \mathbf{1}\ \mathsf{eq}}$$

$$\text{KEQ-PROD}\ \frac{\Theta \vdash \kappa_1\ \mathsf{eq} \quad \Theta \vdash \kappa_2\ \mathsf{eq}}{\Theta \vdash \kappa_1 \times \kappa_2\ \mathsf{eq}} \qquad \text{KEQ-SUM}\ \frac{\Theta \vdash \kappa_1\ \mathsf{eq} \quad \Theta \vdash \kappa_2\ \mathsf{eq}}{\Theta \vdash \kappa_1 + \kappa_2\ \mathsf{eq}} \qquad \text{KEQ-TY}\ \frac{}{\Theta \vdash \mathbf{Ty}\ \mathsf{eq}}$$

$$\boxed{\vdash \Omega}$$

$$\text{OCS-REP}\ \frac{}{\vdash \cdot} \qquad \text{OCS-LIT}\ \frac{\emptyset \vdash \kappa\ \mathsf{eq}}{\vdash \textbf{lit}[\kappa]} \qquad \text{OCS-TARG}\ \frac{\vdash \Omega \quad \textbf{op} \notin \text{dom}(\Omega) \quad \emptyset \vdash \kappa\ \mathsf{eq}}{\vdash \Omega, \textbf{op}[\kappa]}$$

$$\kappa_{\text{arg}} := (\mathbf{1} \to (\mathbf{Ty} \times \mathbf{ITm})) \times (\mathbf{Ty} \to \mathbf{ITm})$$

$$\boxed{\vdash_\Xi \text{TC}\ \{\omega\} : \psi}$$

$$\text{OC-REP}\ \frac{\emptyset\ \emptyset \vdash^0_\Xi \sigma_{\text{rep}} : \kappa_{\text{tyidx}} \to \mathbf{ITy}}{\vdash_\Xi \text{TC}\ \{\mathbf{rep} = \sigma_{\text{rep}}\} : \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\cdot\}}$$

$$\text{OC-LIT}\ \frac{\vdash_\Xi \text{TC}\ \{\omega\} : \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \qquad \emptyset\ \emptyset \vdash^0_{\Xi, \text{TC}[\kappa_{\text{tyidx}}]} \sigma_{\text{def}} : \kappa_{\text{tyidx}} \to \kappa \to \textbf{List}[\kappa_{\text{arg}}] \to \mathbf{ITm}}{\vdash_\Xi \text{TC}\ \{\omega, \textbf{ana lit} = \sigma_{\text{def}}\} : \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega, \textbf{lit}[\kappa]\}}$$

$$\text{OC-TARG}\ \frac{\vdash_\Xi \text{TC}\ \{\omega\} : \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \qquad \emptyset\ \emptyset \vdash^0_{\Xi, \text{TC}[\kappa_{\text{tyidx}}]} \sigma_{\text{def}} : \kappa_{\text{tyidx}} \to \mathbf{ITm} \to \kappa \to \textbf{List}[\kappa_{\text{arg}}] \to (\mathbf{Ty} \times \mathbf{ITm})}{\vdash_\Xi \text{TC}\ \{\omega, \textbf{syn op} = \sigma_{\text{def}}\} : \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega, \textbf{op}[\kappa]\}}$$

Figure 13: Type constructor kinding.

$$\begin{array}{llll}
\Phi_{\text{ex}} & := \phi_{\text{rstr}}, \phi_{\text{lprod}} & \Xi_{\text{ex}} & := \Xi_0, \text{RSTR}[\mathbf{Rx}], \text{LPROD}[\textbf{List}[\textbf{Lbl} \times \mathbf{Ty}]] \\
\phi_{\text{rstr}} & \text{(Figure 11)} & \psi_{\text{rstr}} & := \textbf{tcsig}[\mathbf{Rx}]\ \{\Omega_{\text{rstr}}\} \\
\phi_{\text{lprod}} & \text{(Figure 12)} & \psi_{\text{lprod}} & := \textbf{tcsig}[\textbf{List}[\textbf{Lbl} \times \mathbf{Ty}]]\ \{\Omega_{\text{lprod}}\} \\
\Omega_{\text{rstr}} & := & \multicolumn{2}{l}{\textbf{lit}[\mathbf{Str}], \textbf{concat}[\mathbf{1}], \#[\mathbf{Nat}], \textbf{coerce}[\mathbf{Rx}], \textbf{case}[\mathbf{1}], !\textbf{rc}[\mathbf{1}]} \\
\Omega_{\text{lprod}} & := & \multicolumn{2}{l}{\textbf{lit}[\textbf{List}[\textbf{Lbl}]], \#[\textbf{Lbl}], \textbf{with}[\textbf{List}[\textbf{Lbl}]], !\textbf{ri}[\mathbf{1}]}
\end{array}$$

Figure 14: The contexts and signatures for Figures 11 and 12.

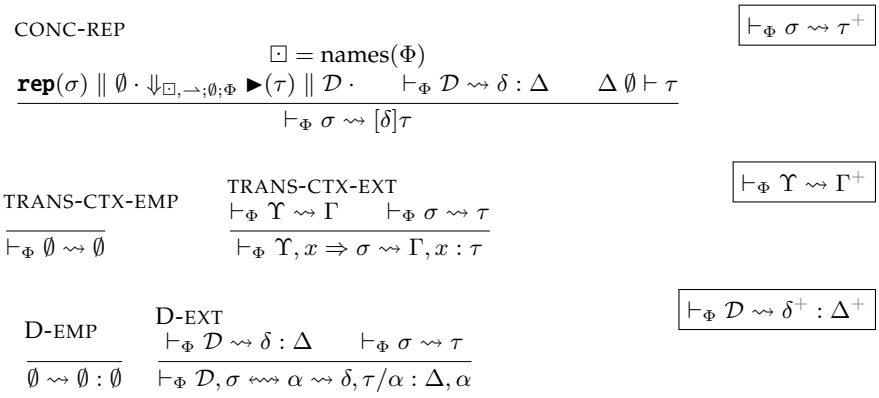

Figure 15: Concrete Representations

$\boxed{\Upsilon \vdash_\Phi e \Rightarrow \sigma^+ \rightsquigarrow \iota^+}$ $\boxed{\Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota^+}$

SUBSUME
$$\frac{\Upsilon \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota}{\Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota}$$

SYN-SLET
$$\frac{\vdash \Phi \sim \Xi \qquad \emptyset\,\emptyset \vdash^0_\Xi \sigma : \kappa \qquad \sigma \Downarrow \sigma' \qquad \Upsilon \vdash_\Phi [\sigma'/\boldsymbol{x}]e \Rightarrow \sigma_{\text{ty}} \rightsquigarrow \iota}{\Upsilon \vdash_\Phi \textbf{letstatic}[\sigma](\boldsymbol{x}.e) \Rightarrow \sigma_{\text{ty}} \rightsquigarrow \iota}$$

ANA-SLET
$$\frac{\vdash \Phi \sim \Xi \qquad \emptyset\,\emptyset \vdash^0_\Xi \sigma : \kappa \qquad \sigma \Downarrow \sigma' \qquad \Upsilon \vdash_\Phi [\sigma'/\boldsymbol{x}]e \Leftarrow \sigma_{\text{ty}} \rightsquigarrow \iota}{\Upsilon \vdash_\Phi \textbf{letstatic}[\sigma](\boldsymbol{x}.e) \Leftarrow \sigma_{\text{ty}} \rightsquigarrow \iota}$$

ASCRIBE
$$\frac{\vdash \Phi \sim \Xi \qquad \emptyset\,\emptyset \vdash^0_\Xi \sigma : \textbf{Ty} \qquad \sigma \Downarrow \sigma' \qquad \Upsilon \vdash_\Phi e \Leftarrow \sigma' \rightsquigarrow \iota}{\Upsilon \vdash_\Phi \textbf{asc}[\sigma](e) \Rightarrow \sigma' \rightsquigarrow \iota}$$

SYN-VAR
$$\frac{x \Rightarrow \sigma \in \Upsilon}{\Upsilon \vdash_\Phi x \Rightarrow \sigma \rightsquigarrow x}$$

SYN-LET
$$\frac{\Upsilon \vdash_\Phi e_1 \Rightarrow \sigma_1 \rightsquigarrow \iota_1 \qquad \vdash_\Phi \sigma_1 \rightsquigarrow \tau_1 \qquad \Upsilon, x \Rightarrow \sigma_1 \vdash_\Phi e_2 \Rightarrow \sigma_2 \rightsquigarrow \iota_2}{\Upsilon \vdash_\Phi \textbf{let}(e_1; x.e_2) \Rightarrow \sigma_2 \rightsquigarrow \textbf{ap}(\lambda[\tau_1](x.\iota_2); \iota_1)}$$

ANA-LET
$$\frac{\Upsilon \vdash_\Phi e_1 \Rightarrow \sigma_1 \rightsquigarrow \iota_1 \qquad \vdash_\Phi \sigma_1 \rightsquigarrow \tau_1 \qquad \Upsilon, x \Rightarrow \sigma_1 \vdash_\Phi e_2 \Leftarrow \sigma_2 \rightsquigarrow \iota_2}{\Upsilon \vdash_\Phi \textbf{let}(e_1; x.e_2) \Leftarrow \sigma_2 \rightsquigarrow \textbf{ap}(\lambda[\tau_1](x.\iota_2); \iota_1)}$$

ANA-FIX
$$\frac{\vdash_\Phi \sigma \rightsquigarrow \tau \qquad \Upsilon, x \Rightarrow \sigma \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota}{\Upsilon \vdash_\Phi \textbf{fix}(x.e) \Leftarrow \sigma \rightsquigarrow \textbf{fix}[\tau](x.\iota)}$$

SYN-LAM
$$\frac{\vdash \Phi \sim \Xi \qquad \emptyset\,\emptyset \vdash^0_\Xi \sigma_1 : \textbf{Ty} \qquad \sigma_1 \Downarrow \sigma_1' \qquad \vdash_\Phi \sigma_1' \rightsquigarrow \tau_1 \qquad \Upsilon, x \Rightarrow \sigma_1' \vdash_\Phi e \Rightarrow \sigma_2 \rightsquigarrow \iota}{\Upsilon \vdash_\Phi \lambda[\sigma_1](x.e) \Rightarrow \textbf{ty}[\rightharpoonup]((\sigma_1', \sigma_2)) \rightsquigarrow \lambda[\tau_1](x.\iota)}$$

SYN-AP
$$\frac{\Upsilon \vdash_\Phi e_1 \Rightarrow \textbf{ty}[\rightharpoonup]((\sigma_1, \sigma_2)) \rightsquigarrow \iota_1 \qquad \Upsilon \vdash_\Phi e_2 \Leftarrow \sigma_2 \rightsquigarrow \iota_2}{\Upsilon \vdash_\Phi \textbf{ap}(e_1; e_2) \Rightarrow \sigma_2 \rightsquigarrow \textbf{ap}(\iota_1; \iota_2)}$$

ANA-LIT
$$\frac{\begin{array}{c}\textbf{tycon}\ \text{TC}\ \{\omega\} : \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \in \Phi \qquad \textbf{lit}[\kappa_{\text{tmidx}}] \in \Omega \qquad \vdash \Phi \sim \Xi \qquad \emptyset\,\emptyset \vdash^0_\Xi \sigma_{\text{tmidx}} : \kappa_{\text{tmidx}} \\ \textbf{ana lit} = \sigma_{\text{def}} \in \omega \qquad \overline{e} \looparrowright \sigma_{\text{args}} \looparrowright \mathcal{G}_0; n \\ \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \parallel \emptyset\ \mathcal{G}_0 \Downarrow_{\rightharpoonup,\text{TC};\Upsilon;\Phi} \rhd(\iota_{\text{abs}}) \parallel \mathcal{D}\ \mathcal{G} \\ \textbf{rep}(\textbf{ty}[\text{TC}](\sigma_{\text{tyidx}})) \parallel \mathcal{D} \cdot \Downarrow_{\rightharpoonup,\text{TC};\emptyset;\Phi} \blacktriangleright(\tau_{\text{abs}}) \parallel \mathcal{D}' \cdot \\ \vdash_\Phi \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma \qquad \Delta\,\Gamma \vdash \iota_{\text{abs}} : \tau_{\text{abs}}\end{array}}{\Upsilon \vdash_\Phi \textbf{lit}[\sigma_{\text{tmidx}}](\overline{e}) \Leftarrow \textbf{ty}[\text{TC}](\sigma_{\text{tyidx}}) \rightsquigarrow [\delta][\gamma]\iota_{\text{abs}}}$$

SYN-TARG
$$\frac{\begin{array}{c}\Upsilon \vdash_\Phi e_{\text{targ}} \Rightarrow \textbf{ty}[\text{TC}](\sigma_{\text{tyidx}}) \rightsquigarrow \iota_{\text{targ}} \\ \textbf{tycon}\ \text{TC}\ \{\omega\} : \textbf{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \in \Phi \\ \textbf{op}[\kappa_{\text{tmidx}}] \in \Omega \qquad \vdash \Phi \sim \Xi \qquad \emptyset\,\emptyset \vdash^0_\Xi \sigma_{\text{tmidx}} : \kappa_{\text{tmidx}} \\ \textbf{syn op} = \sigma_{\text{def}} \in \omega \qquad e_{\text{targ}}; \overline{e} \looparrowright \sigma_{\text{args}} \looparrowright \mathcal{G}_0; n \\ \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ (\textbf{\textit{tr}}\ \textbf{syn}[0])\ \sigma_{\text{tmidx}}\ (\textbf{\textit{atl}}\ \sigma_{\text{args}}) \parallel \emptyset\ \mathcal{G}_0 \Downarrow_{\rightharpoonup,\text{TC};\Upsilon;\Phi} (\sigma, \rhd(\iota_{\text{abs}})) \parallel \mathcal{D}\ \mathcal{G} \\ \textbf{rep}(\sigma) \parallel \mathcal{D} \cdot \Downarrow_{\rightharpoonup,\text{TC};\emptyset;\Phi} \blacktriangleright(\tau_{\text{abs}}) \parallel \mathcal{D}' \cdot \\ \vdash_\Phi \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma \qquad \Delta\,\Gamma \vdash \iota_{\text{abs}} : \tau_{\text{abs}}\end{array}}{\Upsilon \vdash_\Phi \textbf{targ}[\textbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e}) \Rightarrow \sigma \rightsquigarrow [\delta][\gamma]\iota_{\text{abs}}}$$

ANA-OTHER
$$\frac{\vdash_\Phi \textbf{rep}(\Upsilon) \rightsquigarrow \Gamma \parallel \mathcal{D} \qquad \mathcal{D} \rightsquigarrow \delta; \Delta \qquad \Delta\,\Gamma \vdash \iota : \tau}{\Upsilon \vdash_\Phi \textbf{other}[\iota] \Leftarrow \textbf{otherty}[n; \tau] \rightsquigarrow [\delta]\iota}$$

Figure 16: Typing

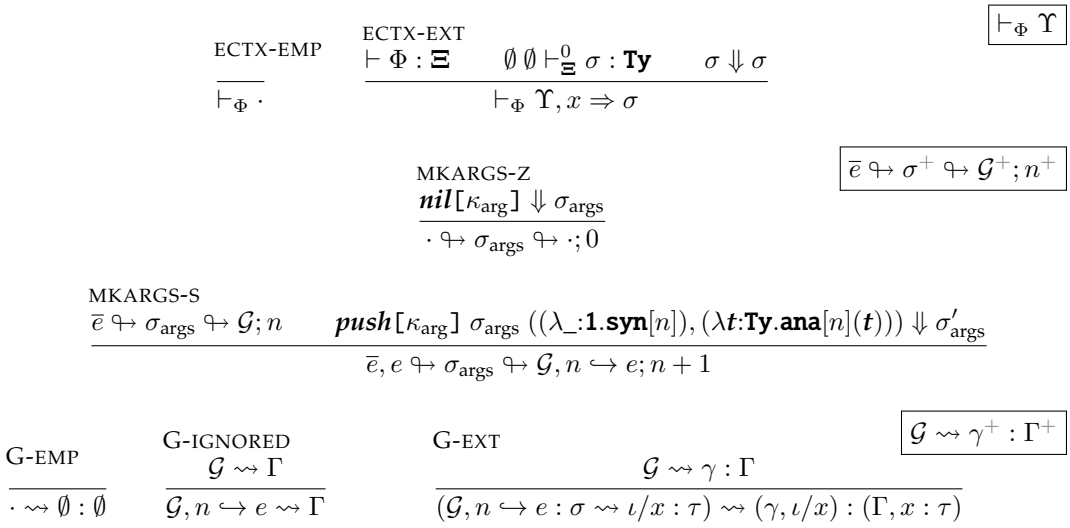

Figure 17: Auxiliary judgements for external statics.

(constructs lifted from $\mathcal{L}\{\rightarrow \forall\, \mu_{\text{ind}}\ \mathbf{1} \times +\}$ are standard, omitted [29])

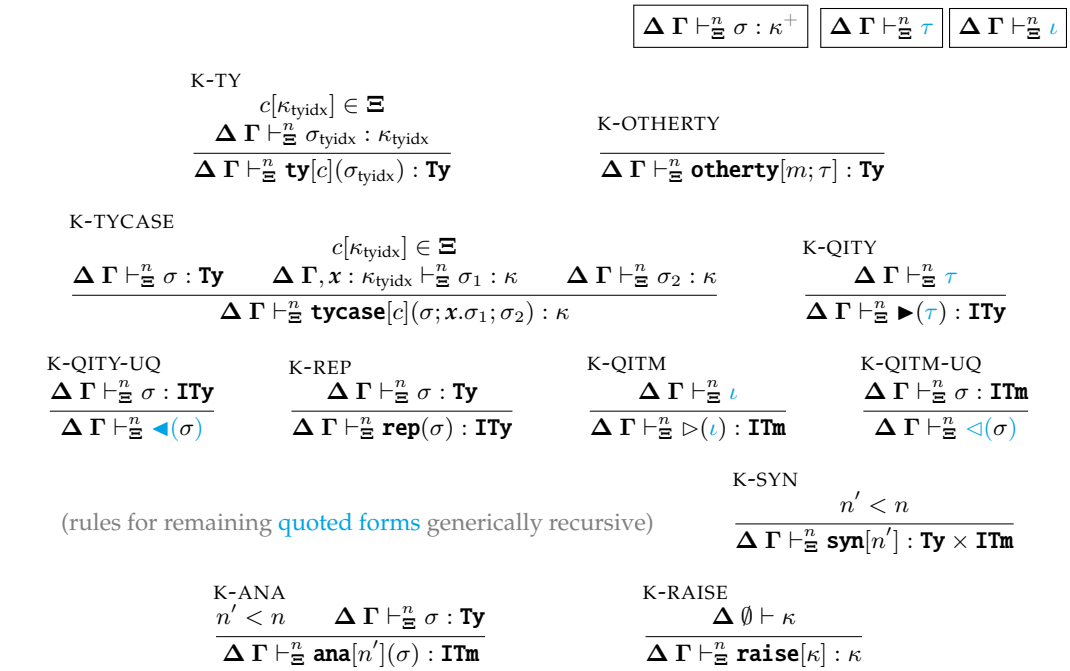

Figure 18: Kinding for static terms.

rep whitelist $\boxdot ::= \rightharpoonup \mid \boxdot, \mathrm{TC}$ **abstract rep store** $\mathcal{D} ::= \emptyset \mid \mathcal{D}, \sigma \leftrightsquigarrow \alpha$ **argument store** $\mathcal{G} ::= \cdot \mid \mathcal{G}, n \hookrightarrow e \mid \mathcal{G}, n \hookrightarrow e : \sigma \rightsquigarrow \iota / x : \tau$ **op context** $\mathcal{C} ::= \boxdot; \Upsilon; \Phi$

$\boxed{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma^{+} \parallel \mathcal{D}^{+}\ \mathcal{G}^{+}}$ $\boxed{\sigma \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}}$

N-TY
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}'}{\mathbf{ty}[c](\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathbf{ty}[c](\sigma') \parallel \mathcal{D}'\ \mathcal{G}'}$$

N-OTHERTY
$$\frac{}{\mathbf{otherty}[m;\tau] \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathbf{otherty}[m;\tau] \parallel \mathcal{D}\ \mathcal{G}}$$

N-TYCASE-1
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathbf{ty}[c](\sigma_{\text{tyidx}}) \parallel \mathcal{D}'\ \mathcal{G}' \quad [\sigma_{\text{tyidx}}/\boldsymbol{x}]\sigma_1 \parallel \mathcal{D}'\ \mathcal{G}' \Downarrow_{\mathcal{C}} \sigma_1' \parallel \mathcal{D}''\ \mathcal{G}''}{\mathbf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma_1' \parallel \mathcal{D}''\ \mathcal{G}''}$$

N-TYCASE-2
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \quad \sigma' \neq \mathbf{ty}[c](\sigma_{\text{tyidx}}) \quad \sigma_2 \parallel \mathcal{D}'\ \mathcal{G}' \Downarrow_{\mathcal{C}} \sigma_2' \parallel \mathcal{D}''\ \mathcal{G}''}{\mathbf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma_2' \parallel \mathcal{D}''\ \mathcal{G}''}$$

N-REP-CONC-TC
$$\frac{\begin{array}{c}\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \mathbf{ty}[\mathrm{TC}](\sigma_{\text{tyidx}}) \parallel \mathcal{D}'\ \mathcal{G}' \\ \mathbf{ty}[\mathrm{TC}](\sigma_{\text{tyidx}}) \leftrightsquigarrow \_ \notin \mathcal{D}' \quad \mathrm{TC} \in \boxdot \quad \mathbf{tycon}\ \mathrm{TC}\ \{\omega\} : \psi \in \Phi \quad \mathbf{rep} = \sigma_{\text{rep}} \in \omega \\ \sigma_{\text{rep}}\ \sigma_{\text{tyidx}} \parallel \mathcal{D}' \cdot \Downarrow_{\boxdot;\emptyset;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}'' \cdot\end{array}}{\mathbf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}''\ \mathcal{G}'}$$

N-REP-CONC-PARR
$$\frac{\begin{array}{c}\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathbf{ty}[\rightharpoonup]((\sigma_1, \sigma_2)) \parallel \mathcal{D}'\ \mathcal{G}' \\ \mathbf{rep}(\sigma_1) \parallel \mathcal{D}' \cdot \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1) \parallel \mathcal{D}'' \cdot \\ \mathbf{rep}(\sigma_2) \parallel \mathcal{D}'' \cdot \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1) \parallel \mathcal{D}''' \cdot\end{array}}{\mathbf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1 \rightharpoonup \tau_2) \parallel \mathcal{D}'''\ \mathcal{G}'}$$

N-REP-ABS
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \mathbf{ty}[c](\sigma_{\text{tyidx}}) \parallel \mathcal{D}'\ \mathcal{G}' \quad \mathbf{ty}[c](\sigma_{\text{tyidx}}) \leftrightsquigarrow \_ \notin \mathcal{D}' \quad c \notin \boxdot \quad (\alpha \text{ fresh})}{\mathbf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\alpha) \parallel \mathcal{D}', \mathbf{ty}[c](\sigma_{\text{tyidx}}) \leftrightsquigarrow \alpha\ \mathcal{G}'}$$

N-REP-ABS-OTHER
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathbf{otherty}[m;\tau] \parallel \mathcal{D}'\ \mathcal{G}' \quad \mathbf{otherty}[m;\tau] \leftrightsquigarrow \_ \notin \mathcal{D}' \quad (\alpha \text{ fresh})}{\mathbf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\alpha) \parallel \mathcal{D}', \mathbf{otherty}[m;\tau] \leftrightsquigarrow \alpha\ \mathcal{G}'}$$

N-REP-SEEN
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \quad \sigma' \leftrightsquigarrow \alpha \in \mathcal{D}'}{\mathbf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\alpha) \parallel \mathcal{D}'\ \mathcal{G}'}$$

N-SYN
$$\frac{n \hookrightarrow e \in \mathcal{G} \quad \Upsilon \vdash_{\Phi} e \Rightarrow \sigma \rightsquigarrow \iota \quad (x \text{ fresh}) \quad \mathbf{rep}(\sigma) \parallel \mathcal{D} \cdot \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}' \cdot}{\mathbf{syn}[n] \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} (\sigma, \triangleright(x)) \parallel \mathcal{D}'\ \mathcal{G} \otimes n \hookrightarrow e : \sigma \rightsquigarrow \iota / x : \tau}$$

N-SYN-SEEN
$$\frac{n \hookrightarrow e : \sigma \rightsquigarrow \iota / x : \tau \in \mathcal{G}}{\mathbf{syn}[n] \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} (\sigma, \triangleright(x)) \parallel \mathcal{D}\ \mathcal{G}}$$

N-SYN-FAIL
$$\frac{n \hookrightarrow e \in \mathcal{G} \quad [\Upsilon \vdash_{\Phi} e \not\Rightarrow]}{\mathbf{syn}[n] \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\boxdot;\Upsilon;\Phi}}$$

N-ANA
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \quad n \hookrightarrow e \in \mathcal{G}' \quad \Upsilon \vdash_{\Phi} e \Leftarrow \sigma' \rightsquigarrow \iota \quad (x \text{ fresh}) \quad \mathbf{rep}(\sigma') \parallel \mathcal{D}' \cdot \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}'' \cdot}{\mathbf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \triangleright(x) \parallel \mathcal{D}''\ \mathcal{G}' \otimes n \hookrightarrow e : \sigma' \rightsquigarrow \iota / x : \tau}$$

N-ANA-SEEN-OK
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \quad n \hookrightarrow e : \sigma' \rightsquigarrow \iota / x : \tau \in \mathcal{G}'}{\mathbf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \triangleright(x) \parallel \mathcal{D}'\ \mathcal{G}'}$$

N-ANA-SEEN-ERR
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \quad n \hookrightarrow e : \sigma'' \rightsquigarrow \iota / x : \tau \in \mathcal{G}' \quad \sigma' \neq \sigma''}{\mathbf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}}$$

N-ANA-FAIL
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \quad n \hookrightarrow e \in \mathcal{G}' \quad [\Upsilon \vdash_{\Phi} e \not\Leftarrow \sigma']}{\mathbf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\boxdot;\Upsilon;\Phi}}$$

N-RAISE
$$\frac{}{\mathbf{raise}[\kappa] \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}}$$

Figure 19: Static Normalization (selected)

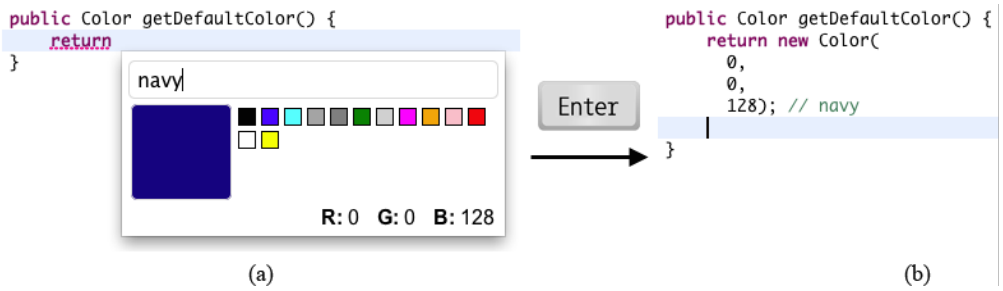

Figure 20: (a) An example code completion palette associated with the `Color` class. (b) The elaboration generated by this palette.