Type-Oriented Foundations for Safely Extensible Programming Systems

PhD Thesis Proposal

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

February 25, 2014

Abstract

We propose a thesis defending the following statement:

Active types allow providers to extend a programming system with new syntax, semantics and editor services from within libraries in a safe and expressive manner.

1 Motivation

Specifying and implementing a programming language together with its supporting tools (collectively, a *programming system*) that is built upon sound theoretical foundations, helps users identify and fix errors as early as possible, supports a natural programming style, and performs well across diverse problem domains and hardware platforms remains a grand challenge in computing. In view of this goal, researchers and domain experts (collectively, *providers*) continue to develop new special-purpose syntax, static and dynamic semantics, implementation strategies, optimizations, run-time systems and tools (collectively, *features*) designed to address these challenges in increasingly diverse contexts. Ideally, a provider would be able to develop and distribute these kinds of new features orthogonally, as libraries, so that client developers could granularly choose those that best satisfy their needs. Unfortunately this is often infeasible because from the perspective of a library, the language's syntax and semantics are fully specified in advance, the compiler and run-time system are "black box" implementations of this fixed specification, and the other tools, like code editors and debuggers, operate according to fixed protocols. Providers of new system features must, as a result, take *language-external approaches*, often by deriving a new programming system altogether. This approach has been encouraged, historically, by the availability of tools like compiler generators and language workbenches. We will argue that these approaches are technically problematic and that taking them has led to an unnecessary gap between research and practice. In their place, we will develop *language-integrated extensibility mechanisms* that decentralize control over several core aspects of the programming system. By organizing new features around types and constraining them appropriately, we aim to show that these mechanisms can guarantee safety and non-interference of extensions while remaining highly expressive.

1.1 Motivating Example: Regular Expressions

To make the problem we aim to address concrete, we begin with a simple example that we will return to throughout this work. *Regular expressions* are a widely-used abstraction for finding patterns in semi-structured strings (e.g. DNA sequences) [45]. If a programming system wished to fully support regular expressions, it might simultaneously provide features like these:

1. **Syntax for pattern literals** so that the cognitive load of reading and writing patterns is low, patterns are parsed once at compile-time and malformed patterns result in intelligible compile-time errors.
2. A **static semantics** that ensures that key invariants related to regular expressions are maintained:
   (a) only other patterns and properly escaped strings are interpolated into a pattern, to avoid splicing errors and injection attacks [3, 6]
   (b) out-of-bounds backreferences are not used [42]
   (c) string processing operations do not lead to a string that is malformed, when well-formedness can be captured by a regular expression [17]

   When an error is found, an intelligible error message is provided.
3. **Editor services** that allow clients to interactively test regular expression patterns against test strings, refer to documentation or search for common patterns (e.g. [1]).

No system today builds in support for all of the features enumerated above in their strongest form. Instead, libraries generally provide support for regular expressions by leveraging general-purpose abstraction mechanisms. Unfortunately, it is impossible to fully define the syntax and the specialized static semantics described in the references above in terms of general-purpose notations and abstractions. Library providers thus need to compromise. The most common strategy is to ask clients to enter patterns as strings deferring their parsing, typechecking, compilation and use to run-time. This proves only a weak approximation to feature 1 (due to clashes between string escape sequences and regular expression syntax). None of the static guarantees can be provided in this way, leading to run-time exceptions (even in well-tested code, as shown in [42]), and logic errors that are not caught even at run-time, some of which can lead to security vulnerabilities (due to injection attacks, for example [3]). It also introduces performance overhead (due to run-time parsing and compilation of patterns, and potentially redundant run-time string checks). Requiring that clients instead introduce patterns directly using general-purpose constructs like object or inductive datatype constructors or operations over an abstract type rather than via strings can only provide feature 2a while eliminating even the weak approximation to feature 1 that the use of string literals provides.

None of these approaches address the issue of tool support. Regular expressions are not trivial to work with, so tool support can be quite helpful, even for experts, and a number of tools are available online. Unfortunately, tools that must be discovered independently and accessed externally are used infrequently [30] and are less usable than editor-integrated tools [10, 33], leading to lower productivity.

1.2 Language-External Approaches

When the syntax and semantics of a system must be extended to fully realize a new feature, as in the example above and others we will give throughout this work, providers typically take a *language-external approach*, either by developing a new or derivative programming system (supported by *language workbenches* [15], *DSL frameworks* [16] or *compiler generators* [8]), or by extending an existing artifact, such as an extension mechanism for a particular

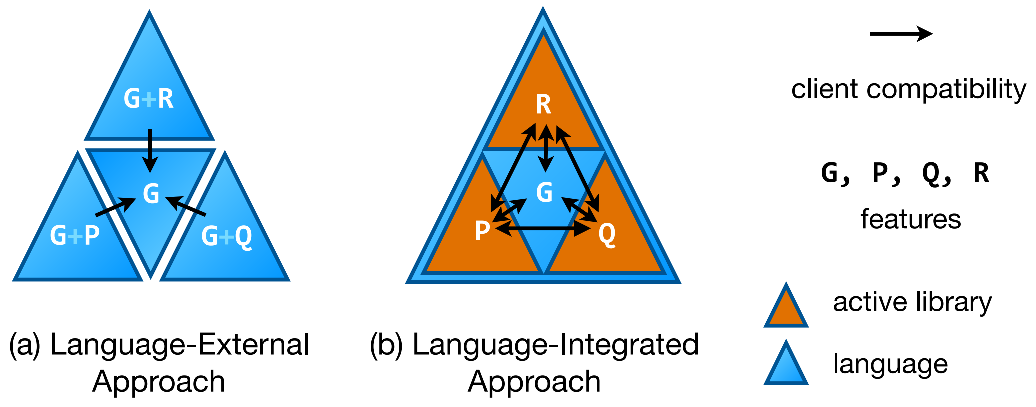


Figure 1: (a) When taking a language-external approach, new features are packaged together into separate languages and tools, causing problems with orthogonality and client compatibility (described in the text). (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time logic governing new constructs is expressed within "active" libraries. If the extension mechanism guarantees that extensions cannot interfere with one another or the base system, these problems are avoided.

compiler[1], editor or other tool. For example, a researcher interested in providing regular expression related features (let us refer to these collectively as `R`) might design a new system with built-in support for these, 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 an alternative parallel programming abstraction (`Q`) might again do the same. This results in a collection of distinct systems as diagrammed in Figure 1a. Unfortunately, when providers of new features take language-external approaches like this, it causes problems for clients related to **orthogonality** and **client compatibility**.

Orthogonality Features implemented by language-external means cannot be adopted individually, but instead are only available coupled to a fixed collection of other features. This makes adoption more costly when these incidental features are not desirable or insufficiently developed, or when the features bundled with a different language or tool are simultaneously desirable. 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 is no requirement that providers of new features use a common mechanism. Even in cases where a common mechanism has been used, there are 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 from being driven into practice: developers prefer language-integrated parallel programming abstractions with stronger safety guarantees and more natural syntax to library-based abstractions if all else is equal [11], but library-based implementations are more widely adopted because "parallel programming languages" privilege only a few chosen abstractions at the language level. This is problematic because different parallel

[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 languages. Many "practical" compilers, including `gcc`, `GHC` and `SML/NJ`, are guilty of this, meaning that some programs that seem to be written in C, Haskell or Standard ML are actually written in tool-specific derivatives of these languages. Language-integrated mechanisms do not lead to such fragmentation.

programming abstractions are seen as more appropriate in different situations [44]. Moreover, parallel programming support is rarely the only concern relevant to client developers outside of a classroom setting. Regular expression support, for example, may be simultaneously desirable for processing large amounts of textual data in parallel, but using these features together in the same compilation unit would be difficult or impossible.

Client Compatibility Even in cases where for each component of a software system there is a programming system that is completely satisfactory in isolation, there remain problems at the interface between components. An interface that exposes the specialized constructs particular to one language (e.g. futures in a parallel programming language) cannot necessarily be safely and naturally consumed from another language (e.g. a general-purpose language). Tool support is also lost when calling into a different language. We call this fundamental issue the *client compatibility problem*: code written by clients of a certain collection of features cannot always interface with code written by clients of a different collection in a safe, performant and natural manner.

One strategy often taken by proponents of a *language-oriented approach* to software development [50] to partially address the client compatibility problem is to target an established intermediate language, such as the Java Virtual Machine (JVM) bytecode, and use its constructs as a common language for communication between components written in different languages. Scala [32] and F# [35] 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 client compatibility 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.

This approach can work well when new languages consist of constructs that can also be expressed safely and almost as naturally in the common language. But many of the most innovative constructs found in modern languages (often, those that justify their creation) are difficult to define in terms of existing constructs in ways that guarantee all necessary invariants are statically maintained and that do not require large amounts boilerplate code and run-time overhead. As a trivial example with significant practical implications, F#'s type system does not admit `null` as a value for any type defined within F# code, but maintaining this important invariant still requires run-time checks because the 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 [43, 23], but this more specialized invariant is left unchecked at language boundaries. No alternative general-purpose intermediate language would change this situation.

1.3 Language-Integrated Approaches

We argue that, due to these problems with orthogonality and client compatibility, 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 have previously required central planning, so that language-external approaches are less frequently necessary. More specifically, we will show how control over aspects of the **syntax**, **static and dynamic semantics** and **editor services** can be delegated to user-defined logic distributed in libraries, as illustrated in Figure 1b. Such libraries have been called *active libraries* [48] because, rather than being passive clients of features already available in the system, they contain logic invoked by the system during development or compilation to provide new features. Features implemented within active libraries can be imported as needed, unlike features implemented by external means, seemingly avoiding the problems of orthogonality and client compatibility.

We must proceed with caution, however: critical issues having to do with safety must be overcome before language-integrated extension mechanisms can be integrated into a system. If too much control over these core aspects 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. For example, if two active libraries introduce the same syntactic form but back it with differing (but individually valid) semantics, the issue would only manifest itself when both libraries were imported somewhere within the same scope. These kinds of safety issues have plagued previous attempts to design language-integrated extensibility mechanisms. We will briefly review some of these attempts below.

1.3.1 Background

The term *active libraries* was first introduced by Veldhuizen et al. [48, 47] 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 by being able to influence the programming system at compile-time or edit-time, 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 [46]. 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 [39]).

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. These mechanisms suffer from problems of composability, safety and expressiveness. For example, compile-time macros, such as those in MetaML [40], Template Haskell [41] and Scala [9], take full control over all of the code that they enclose. This can be problematic, however, as outer macros can interfere with the functionality of 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 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, and to build tools that operate robustly in their presence.

Some term rewriting systems replace the explicitly delimited scoping of macros with global pattern-based dispatch. Xroma (pronounced "Chroma"), for example, is designed around active libraries and allows users to insert custom rewriting passes into the compiler from within libraries [48]. Similarly, the derivative of Haskell implemented by the Glasgow Haskell Compiler (GHC) allows providers to introduce custom compile-time term rewriting logic if an appropriate flag is passed in [22]. In both cases, the user-defined logic can dispatch on arbitrary patterns of code throughout the component or program the extension is activated within, so these mechanisms permit non-local extensions to the static and dynamic semantics. 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 [12] and other languages generated by Sugar* [13], like SugarHaskell [14]. 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 notations 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 Active Types

The language-integrated extension mechanisms that we will introduce in this thesis are designed to be highly expressive, permitting library-based implementations of features that compare to and go beyond the features found in modern programming systems, including those implemented via mechanisms like those described above. However, we seek to avoid the associated safety and composability issues and maintain the ability to understand and reason locally about code.

To motivate our approach, let us return to our example of regular expressions. Observe that every feature described in Sec. 1.1 relates specifically to how terms classified by a single user-defined type or parameterized family of types should behave. In fact, nearly all the features relate to types representing regular expression patterns. Feature 1 calls for specialized syntax for the introductory form for this type. Features 2a and 2b relate to its static semantics. Feature 3 is about its edit-time behavior. The remaining feature, 2c, relates to the semantics of a related parameterized family of types: `StringIn[r]`, which classifies strings known to be in the language of a statically-known regular expression `r`. It is exclusively when editing or compiling expressions of the associated type that the logic in Sec. 1.1 needs to be considered.

Indeed, this is not a property unique to our chosen example, but a commonly-seen pattern in programming language design. The semantics of a programming language or logic is often organized around its types (equivalently, its propositions). In two major textbooks about programming languages, *TAPL* [36] and *PFPL* [20], most chapters describe the semantics and metatheory of a few new types and their associated primitive operations without reference to other types. The composition of the types and associated operations from different chapters into complete languages is a language-external operation. For example, in PFPL, the notation $\mathcal{L}\{\rightarrow \texttt{nat}\ \texttt{dyn}\}$ represents a language composed of the arrow ($\rightarrow$), `nat` and `dyn` types and their associated operators. Each of these are defined in separate chapters, and it is generally left unstated that the semantics and metatheory developed separately will be conserved in composition (justified by the fact that, upon careful examination, it is indeed the case that almost any combination of types defined separately in PFPL can be combined to form a language with little trouble).

This ubiquitous type-oriented organization suggests a language-integrated alternative to the mechanisms described in Section 1.3.1 that preserves much of their expressiveness but eliminates the possibility of conflict and makes it easier to reason locally about a piece of code: associating extension logic directly with a single type (or parameterized family of types) and scoping it to operate globally, but only on expressions classified by that type (or by a type in that family). This guarantees that different extensions don't have overlapping scope, preventing a range of common conflicts. By constraining the extension logic itself by various means we will show that the system as a whole can also maintain many other important safety and non-interference properties that have not previously been achieved. We call types with such logic associated with them *active types* and systems that support them *actively-typed programming systems*.

2.1 Proposed Contributions

This thesis will introduce several language-integrated extensibility mechanisms, each based on active types, that give providers control over a different aspect of the system from within libraries (that is, in a decentralized manner). In each case, we will show that the system remains fundamentally safe and that extensions cannot interfere with one another. We will also discuss various points in the design space related to extension correctness (as distinct from safety, which must be maintained 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 our mechanisms. 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.

We begin in Sec. 3 by considering **syntax**. The availability of specialized syntax can bring numerous cognitive benefits [18], and discourage the use of problematic techniques like using strings to represent structured data [6]. 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. For example, regular expression pattern literals as described in Sec. 1.1 are an introductory form for the `Pattern` type. In the mechanism we introduce, literal syntax is associated directly with a type and can be used only where an expression of that type 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 the base grammar of the language is never extended directly. We begin by introducing such *type-specific languages (TSLs)* in the context of a new language we are developing, 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, combining work on bidirectional type systems and elaboration semantics, and introduce a novel mechanism that statically prevents another form of interference – unsafe variable capture and shadowing by extensions (providing a form of *hygiene*). Finally, we conduct a corpus analysis to examine this technique's expressiveness, finding that a substantial fraction of string literals in existing code could be replaced by TSL literals.

Wyvern has an extensible syntax but a fixed general-purpose static and dynamic semantics. The general-purpose abstraction mechanisms we have included in Wyvern are powerful, and implementation techniques for these are well-developed, but there remain situations where providers may wish to extend the **semantics** of a language directly, by introducing new primitive types and operators. Examples of type system extensions that require 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 [42]) or to execute a decision procedure for language inclusion when determining whether a coercion requires a run-time check (feature 2c, see [17]). We discuss more examples from the literature where general-purpose abstraction mechanisms proved inadequate and researchers had to turn to language-external approaches in Sec. 4. To support these more advanced use cases in a decentralized manner, we next develop language-integrated mechanisms for implementing semantic extensions, while leaving the syntax fixed. We begin in Sec. 5 with a type theoretic treatment, specifying an "actively typed" lambda calculus called @λ. By beginning from first principles, we are able to cleanly state and prove the key safety and non-interference theorems and examine the connections between active types and several prior notions, including type-level computation, typed compilation and abstract types. We then go on in Sec. 6 to demonstrate the expressiveness of this mechanism by designing and implementing a full-scale actively typed language, Ace. Interestingly, Ace is itself bootstrapped as a library within an existing language without a conventional static type system, Python. We discuss how we accomplish this, relate Ace to the core calculus, discuss how we can achieve safety properties given the weaker guarantees that Python itself provides, and implement a number of powerful primitives from existing languages as libraries, giving examples from a variety of paradigms, including low-level parallel and concurrent languages, functional languages, object-oriented languages and specialized domains, like the regular expression types discussed in the introduction. We also introduce a novel extensible form of staged compilation where the static class tags of Python values can propagate into Ace functions "just-in-time", and argue that this is particularly well-suited to contemporary scientific workflows.

Finally, in Sec. 7, we will show how **editor services** can be implemented from within active libraries, by a technique we call *active code completion*. To provide a new editor service, providers associate specialized user interfaces, called *palettes*, with types. Clients discover and invoke palettes from the code completion menu at edit-time, 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 generates a term of the type it is associated with based on the information received from the user. Using several empirical methods, including a large developer survey, we examined the expressive power of this approach and developed design criteria. Based on these criteria, we then developed an active code completion system for Java called Graphite. Using Graphite, we implemented a palette for working with regular expressions and conducted a small study that demonstrates the usefulness of type-specific editor services as compared to externally-available tools.

Taken together, this work aims to demonstrate that actively typed mechanisms can be introduced throughout a programming system to allow users to extend its syntax, semantics and edit-time behavior from within libraries, without weakening the safety guarantees that the system provides. We approach the problem both from first principles by developing type-theoretic models and from the perspective of contemporary software engineering practice by developing practical implementations, conducting corpus analyses and providing realistic examples throughout this work. This evidences that types are both a theoretically elegant and practical organizational unit for defining programming system features. It is precisely our type-oriented approach that makes it possible to guarantee that features introduced by extension providers will be safely composable in any combination. In the future, we anticipate developing a programming system that will bring together several actively-typed mechanisms, organized around a minimal, well-specified and formally verified core, where nearly every feature is specified, implemented and verified in a decentralized manner and distributed as a library.

3 Type-Specific Languages

General-purpose abstraction mechanisms like those built into modern languages are semantically quite flexible. By using a general-purpose abstraction mechanism to define an abstraction, one immediately benefits from a body of useful operations, established reasoning principles, well-optimized implementations and tool support. For example, lists can be encoded using a more general mechanism for defining inductive datatypes. Intuitively, a list can either be empty, or be broken down into a *head* element and a *tail*, another list. In an ML-like language, the polymorphic list type is declared:

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

By encoding lists in this way, we can immediately reason about them by structural induction and examine them by pattern matching.

While these operations and reasoning principles can be quite useful, the associated general-purpose syntax can sometimes be a liability. 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 include *literal syntax* for introducing them, e.g. `[1, 2, 3, 4]`. This notation is semantically equivalent to the general-purpose notation shown above, but brings cognitive benefits by drawing attention to the content of the list, rather than the nature of the encoding. To use terminology from the literature on the cognitive dimensions of notations [18], it is more *terse*, *visible* and *maps more closely* to the intuitive notion of a list.

Although list, number and string literals are nearly ubiquitous features of modern languages, some languages also provide specialized notation for other common data structures (like maps and sets), data formats (like XML and JSON), query languages (like regular expressions and SQL), markup languages (like HTML) and many other types of data. For example, a language with built-in notation for HTML and SQL, with type-safe *interpolation* of host language terms within 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 shorthand for:

```
let webpage : HTML = HTMLElement(Dict.empty(), [BodyElement(Dict.empty(),
  [H1Element(Dict.empty(), [TextNode(concat("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 the equivalent general-purpose notation is too cognitively demanding for comfort, developers typically turn to run-time mechanisms to make constructing data structures more convenient. Among the most common strategies in these situations is to simply use a string representation that is parsed at run-time. Developers across language paradigms frequently write examples like the above as:

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

Though recovering much of the notational convenience of the literal version, it is still more awkward to write, 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 [3]. Although developers are cautioned to sanitize their input, it can be difficult to verify that this was done correctly throughout a codebase. The most straightforward way to avoid these problems today is to insist on structured representations, despite their inconvenience.

Unfortunately, it does not appear that this is working. 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 authoring the regular expression and when reading it. Among the most common strategies in these situations, for users of both object-oriented and functional languages, is 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: it requires explicit conversions between representations, interference issues with string syntax, correctness problems, performance overhead and security issues.

Today, supporting new literal notations within an existing 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 users to introduce new syntax from within libraries appear to hold promise for the reasons described above, providing 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 the clients using conflicting extensions. For example, SugarJ [12] and other extensible languages generated by Sugar* [13] allow providers to 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. 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) can all cause problems that may be difficult to anticipate.

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-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

```
let imageBase : URL = <images.example.com>
let bgImage : URL = <%imageBase%/background.png>
new : SearchServer
  def resultsFor(searchQuery : String, page : Nat) : Unit =
    serve(~) (* serve : HTML -> Unit *)
      :html
        :head
          :title Search Results
          :style ~
            body { background-image: url(%bgImage%) }
            #search { background-color: %`#aabbcc`.darken(20pct)% }
        :body
          :h1 Results for {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

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, dictionaries and other data structures, like JSON literals. This 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 develop our work as a variant of a new programming language being developed by our group called Wyvern [31]. 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 are thus a simple variant of records equipped with 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*.

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.

```
1  casetype HTML
2     Empty of Unit
3   | Text of String
4   | Seq of HTML * HTML
5   | BodyElement of Attributes * HTML
6   | StyleElement of Attributes * CSS
7   | ...
8    metadata = new : HasTSL
9      val parser : Parser = ~
10       start -> ':body'= start>
11         fn child:Exp => `HTML.BodyElement(([], %child%))`
12       start -> ':style'= EXP>
13         fn e:Exp => `HTML.StyleElement(([], %e%))`
14       start -> '{'= EXP '}'=
15         fn e:Exp => `%e% : HTML`
```

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

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` (not shown). This is an object type declaring several fields representing the components of a URL: its protocol, domain name, port, path and so on. We could have created a value of type `URL` using general-purpose notation:

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

This is tedious. Instead, 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 will parse the body (at compile-time) to produce a Wyvern abstract syntax tree (AST) that explicitly instantiates a new object of type `URL` using general-purpose notation.

In addition to supporting conventional notation for URLs, this TSL supports *typed interpolation* of another Wyvern expression of type `URL` to form a larger URL. The interpolated 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 with expected type `URL`, using its AST to construct an AST for the expression as a whole. String interpolation never occurs. Note that the delimiters that are used to go from Wyvern to a TSL are controlled by Wyvern (those in Fig. 3) 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. `HTML.BodyElement((attrs, child))` (unlike in ML, in Wyvern we must explicitly qualify constructors with the case type they are part of when they are used. This is largely to make our formal semantics simpler and for clarity of presentation.) 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,

```
objtype HasTSL
  val parser : Parser
objtype Parser
  def parse(ts : TokenStream) : (Exp *
    TokenStream)
  metadata = new : HasTSL
    val parser : Parser = (* parser generator *)
casetype Type
    Var of ID
  | Arrow of Type * Type
  | Prod of Type * Type
  metadata = new : HasTSL
    val parser : Parser = (* type quasiquotes *)
```

```
casetype Exp
    Var of ID
  | Lam of ID * Type * Exp
  | Ap of Exp * Exp
  | ProdIntro of Exp * Exp
  | CaseIntro of Type * ID * Exp
  ...
  | FromTS of Tokenstream
  | Error of ErrorMessage
  metadata = new : HasTSL
    val parser : Parser = (* quasiquotes *)
```

Figure 5: Some of the types included in the Wyvern prelude.

shown being used on lines 6-20 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.

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 type, forming an *active type*, using a more general mechanism for associating a value with a named type. A value associated with a named type is called its *metadata*, and is introduced as shown on line 8 of Fig. 4. 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 `TokenStream` extracted from a literal body to a Wyvern AST, which is a value of type `Exp`. This is 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 ordinary Wyvern 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 [5] – 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' formalism 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). We will discuss this formalism further when we formally specify Wyvern's layout-sensitive concrete syntax.

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": interpolating another AST into the one being generated. Again, interpolation is safe and structural, rather than based on string interpolation.

We have now seen several examples of TSLs that support interpolation. The question then arises: what type should the interpolated Wyvern expression be expected to have? This is determined by placing the interpolated value in a place in the generated AST where its type is known – on line 11 of Fig. 4 it is known to be `HTML` and on line 13 it is known to be `CSS` by the declaration of `HTML`, and on line 15, it is known to be `HTML` by the use of an explicit ascription. When these generated ASTs are recursively typechecked during compilation, any use of a nested TSL at the top-level (e.g. the CSS TSL in Fig 2) will operate as intended.

3.6 Formalization

A formal and more detailed description can be found in our paper draft.[2] 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 Adams' formalism and provide a full specification for the layout-delimited literal form introduced by a forward reference, `~`, as well as other forms of forward-referenced blocks (for the forms **`new`** and **`case`**`(e)`, in particular).

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 [26]) with an elaboration semantics (in a style similar to Harper and Stone [21]). 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.

3. A naïve rewriting strategy would be *unhygienic* – it could allow for the inadvertent capture of local variables. 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 tokenstream that are interpreted as nested Wyvern expressions. We also show how to explicitly refer to local values available in the parser definition (e.g. helper functions) in a safe way. We formalize this hygiene mechanism by splitting the context in our static semantics.

4. We provide several examples of TSLs throughout the paper, but to examine how broadly applicable the technique is, we conduct a simple corpus analysis, finding that string languages are used ubiquitously in existing Java code (collaborative work with Darya Kurilova).

3.7 Remaining Tasks and Timeline

The paper we have submitted to ECOOP 2014 has received quite positive reviews, so we anticipate that it will be accepted (notifications are sent on Mar. 7th). If accepted, final revisions on the paper text are due on May 12th and so we will largely do these tasks after the OOPSLA deadline in late March. If not accepted, we will aim to resubmit to OOPSLA.

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

1. We must write down the full formal semantics (including rules that we omitted for concision in the paper draft) in a technical report, as well as provide more complete proofs of the metatheory. This might require slightly restricting the recursion in the rule for literals, so that induction is well-founded.

2. Our current static semantics does not support mutually recursive type declarations, but this is necessary for our prelude to typecheck, so we will add support for this. In the paper, the formalism is meant to be expository, so we will likely leave this additional complexity for the tech report.

3. We must further consider aspects of hygiene:

   (a) We do not yet have a clean mechanism for preventing unintentional variable *shadowing*, only unintentional variable *capture*. It may be possible to prevent shadowing by preventing variables from leaking into interpolated Wyvern expressions (so that function application is the only way to pass data from a TSL to Wyvern code, as in the `Parser` TSL above).

   (b) In macro systems like Scheme's, free variables in generated ASTs refer to their bindings within the macro definition, rather than where they are inserted. To support this, we have introduced the **`valAST`** operator. Using it is a bit verbose and awkward – every free variable `x` in a quasiquote must instead be replaced with an interpolated term of the form **`valAST`**`(x)` for the resulting `Exp` to meaningful as the return value of the `parse` function. We believe we can insert **`valAST`** automatically from within the definition of the TSL for `Exp` using single variable (rather than whole expression) interpolation.

4 Type-Oriented Mechanisms for Semantic Extensions

In this and the following two sections, we will focus on language-integrated, type-oriented mechanisms for implementing extensions to the static and dynamic semantics of programming languages, to allow developers to go beyond just the general-purpose constructs, like objects and inductive datatypes, that we have seen thus far. We will return to using a uniform introductory form, under the assumption that the methods in the previous section could also be applied to the languages we are considering (though we will not integrate them explicitly). We will also introduce a flexible (though not extensible) dispatch mechanism for common elimination forms.

4.1 Foundations

Typed programming languages can be considered in terms of the collection of indexed type and operator constructors they provide. The simply-typed lambda calculus (STLC), for example, provides a single type constructor, ARROW, indexed by a pair of types, and two operator constructors: ***lam***, indexed by a type, and ***ap***, which can be thought of as being indexed trivially. Their syntax and static semantics can be written abstractly like this, to emphasize this uniform way of thinking about the type and operator structure and introduce the typographic conventions we will use in Sec. 5:

$$\textsc{arrow-I}\quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \boldsymbol{lam}[\tau](x.e) : \textsc{Arrow}[(\tau, \tau')]} \qquad \textsc{arrow-E}\quad \frac{\Gamma \vdash e_1 : \textsc{Arrow}[(\tau, \tau')] \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \boldsymbol{ap}[()](e_1; e_2) : \tau'}$$

Figure 6: Static semantics of the simply typed lambda calculus.

We might extend this language with universal quantification over types by adding the type constructor FORALL, which is indexed by a type under a binder, and its associated operator constructors (see [20] for a review). If we are building a programming language rather than a theorem prover, we might also add a fixpoint operator. It is tempting to stop here: the language now supports general recursion and the dynamic semantics of sums, products and inductive and co-inductive types can be defined by a method analagous to Church encodings in the untyped lambda calculus [?]. This is of clear theoretical and pedagogical interest. But this calculus remains impractical as a programming language. Reynolds, striking a cautionary tone reminiscent of Perlis in his description of the "Turing tarpit" [34], summarized the issues succinctly [38]:

what to cite for this?

> To say that any reasonable function can be expressed by some program is not to say that it can be expressed by the most reasonable program. It is clear that the language requires a novel programming style. Moreover, it is likely that certain important functions cannot be expressed by their most efficient algorithms.

A simple calculus, equipped perhaps with some purely syntactic desugarings as given by a Church-style weak encoding, is quite flexible when only the dynamic semantics of a program are of interest. But continuing to build new type and operator constructors (collectively, *constructs*) into a language can continue to make statically reasoning about programs simpler and more precise, support a more natural programming style and endow the language with a more favorable cost semantics.

Consistent with this view, typed programming languages generally expose a richer set of constructs externally than their compilers work with internally. A functional language like ML, for example, might expose n-ary product types (tuples), record types, recursive datatypes and forms of type abstraction (polymorphism in the term language and abstract types via the module language). A compiler for the language might then, after typechecking a program according to the richer static semantics that these constructs are governed by, translate it to an internal language (IL) with only simple product and sum types, general recursive types and perhaps some machine-oriented constructs like arrays, for further compilation to machine code. Correctly implementing the dynamic semantics of the language becomes the primary concern during this phase, so a simple IL that decreases the number of cases that must be considered when implementing and verifying the compiler has proven to be a wise choice.

It is again tempting to stop here. Indeed, languages like ML are often referred to as "general-purpose languages" and have been used successfully for a range of computing tasks. The constructs they provide certainly occupy a "sweet spot" in the design space, and a variety of higher-level abstractions can be strongly defined in a satisfying manner. But claims that they are suitable for *all* purposes are not backed by any "universality" theorems stronger than those for the simple calculus described above. In fact, situations continue to arise in both research and practice where new constructs are clearly needed because a useful type system can only be weakly defined in terms of standard general-purpose constructs, or the definition is impractically verbose or inefficient:

1. General-purpose abstractions continue to evolve and admit variants. There are many variations on record types, for example: records with functional record update, prototypic delegation or specified field ordering (that is, labeled tuples) all have valid uses (and we plan to give all of these as examples). Similarly, sum types also admit variants (e.g. various forms of open sums, as we will discuss). Even something as seemingly simple as a type-safe variant of `printf` requires extending a language like Ocaml with a new operator constructor [?].

2. Perhaps more interestingly, specialized type systems that enforce stronger invariants than general-purpose constructs are capable of enforcing are often proposed by researchers. One need not take more than a cursory glance through the literature to discover specialized type systems for parallel programming [?, ?], concurrent programming [?, ?, ?], distributed programming [?], dataflow programming [?], authenticated data structures [?], information flow [?, ?], database queries [?], aliased

citations

references [?], effects [?], network protocols [?], regular expressions [?], units of measure [?] and many others. These examples are all implemented non-orthogonally (typically as extended versions of an existing language).

3. Interoperability layers that allow programmers to safely and naturally interact with libraries written in a different language require enforcing the type system of the foreign language within the calling language. Using an interoperability layer that does not do this can lead to safety issues, even when both languages are separately known to be safe. Safe interoperability layers generally require direct extensions to the language (e.g. in MLj [?]). Interoperability is a particularly important concern when transitioning away from commonly-used languages, so this is a significant barrier to adoption of modern languages in practice.

An extensible programming language could address these problems by providing a safe language-integrated mechanism for introducing new type and operator constructors and implementing their associated static and dynamic semantics directly, and we will explore this in the next two sections. Let us first take a detour into a typical compiler to better motivate the mechanisms that we will introduce into the language.

4.2 From Extensible Compilers to Extensible Languages

The monolithic character of most programming languages is reflected in, and perhaps influenced by, the most common constructs used for implementing programming languages. Let us consider an implementation of the STLC. A compiler written using a functional language will invariably represent the primitive type and operator constructors using closed recursive sums. For example, a simple implementation in Standard ML might be based around these datatypes (where variables, binders and operator application to arguments are handled separately, not shown)[3]:

```
datatype Ty = Arrow of Ty * Ty
datatype Op = Lam of Ty | Ap
```

In a class-based object-oriented implementation of the STLC, we might instead encode type and operator constructors as subclasses of abstract classes `Ty` and `Op`. If typechecking and translation proceed by the common *visitor pattern*, dispatching against a fixed set of known subclasses of `Op`, then we encounter a similar issue: everything is defined at once.

This issue was first discussed by Reynolds [37]. A number of mechanisms have since been proposed that allow new cases to be added to data types and the functions that operate over them in a modular manner. In functional languages, we might turn to *open datatypes and open functions* [25]. For example, we might add a new type constructor, `Tuple`, indexed by a list of types, and two new operator constructors: an introductory form, `NewTuple`, indexed trivially, and an elimination form, `Prj`, indexed by a natural number (hypothetical syntax):

```
newcon Tuple of Ty list extends Ty
newcon NewTuple extends Op
newcon Prj of nat extends Op
```

The logic for functionality like typechecking and translation, if organized around open functions, can then be implemented for only these new cases. For example, the `optype` function that synthesizes a type for an operator invocation given a a list of argument types might be extended to support the new constructor `Prj` like so:

```
optype (Prj(n), [t]) = case t of
   Tuple(ts) => (case List.nth n ts of
      Some t' => t'
     | _ => raise IndexError("<projection index out of bounds>"))
  | _ => raise TypeError("<tuple expected>")
optype (ctx, Prj(n), _) = raise ArityError("<expected 1 argument>")
```

[3]This is essentially the approach that we took in the infrastructure for the coding assignments in 15-312.

Using a class-based mechanism, we might dispense with the visitor pattern and instead invert control to abstract methods of `Op`.

```
class Prj extends Op {
  Prj(Nat n) { this.n = n; }
  Nat n;
  Ty optype(List<Ty> args) {
    if (args.length != 1) throw new ArityError("<expected 1 argument>")
    Ty t = args[0];
    if (t instanceof Tuple) {
      Tuple t = (Tuple) t;
      try {
        return t.idx[this.n]
      } catch OutOfBoundsException(e) {
        throw new IndexError("<projection index out of bounds>");
      }
    } else throw new TypeError("<tuple expected>");
  }
}
```

If we allowed programmers to load definitions like these into our compiler by, e.g., a pragma declaration, we will still have only succeeded in creating an extensible compiler. We have not created an extensible programming language because other compilers for the same language will not necessarily support the same extensions. These mechanisms are not truly *language-integrated*, so if our newly-introduced constructs are exposed at a library's interface boundary, clients of that library who are using different compilers face the same problems with client compatibility that those using different languages face (as described in Sec. 1.2). Worse still, these mechanisms allow the metatheoretic properties of the language to be weakened by extensions and orthogonality is not guaranteed (in several senses of the word, which we will return to). Nevertheless, these two strategies give us the conceptual seeds for the approaches that we will take in Secs. 5 and 6. In both cases, we introduce into the language a mechanism similar to, but more constrained and specialized than, the two above. Extensions become library imports, rather than compiler-specific pragmas or flags.[4]

We begin in Sec. 5 by developing a core calculus called @λ and focusing on theoretical issues, including type safety, decidability and conservativity (that extensions cannot weaken one another, in any combination). We will also sketch a flexible type-directed dispatch mechanism for standard syntactic forms that targets this core calculus, relying fundamentally on the idea of associating extension logic with types. We then give examples of simple but interesting extensions that are built into, or are only weakly definable, in modern functional languages.

This provides a bridge to allow us to continue on in Sec. 6 with the task of designing and implementing a full-scale programming language based on similar mechanisms called Ace. We will show more interesting examples of statically-typed general-purpose abstractions, as well as specialized abstractions (including the entirety of the OpenCL programming language for GPU programming), implemented as libraries in Ace. Ace uses Python as its type-level language and supports user-defined base and target languages, rather than picking a fixed base language ($\mathcal{L}\{\rightarrow\}$) and target language ($\mathcal{L}\{\rightharpoonup \mathbb{Z}\ \mathbf{1} \times +\}$) as in @$\lambda$. Ace also supports an extensible form of "just-in-time" staging: dynamic class tags extracted from Python values can propagate into the Ace compiler as static types at the point during execution where an Ace function is called from a Python script. We show that this is particularly well-suited for scientific workflows.

Ace is aimed squarely at achieving the goal of *expressiveness*. That is, it helps advance the goal of this work, which is to give programmers orthogonal access to a richer variety of features from within a single language. Python is at one extreme, initially providing effectively only a single type, `dyn`, so it poses an interesting challenge. Using a type-level

[4]For unfortunate languages where a canonical implementation functions as the specification (e.g. Haskell and Scala), the mechanisms we introduce can be left in the compiler. Our metatheoretic advances apply also to this design, but we do not consider it further in this thesis.

$$
\begin{array}{rrcl}
\textbf{programs} & \rho & ::= & \mathsf{tycon}\ \textsc{Tycon}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\mathsf{schema}\ \tau_{\text{rep}}; \theta\};\ \rho \mid e \\
 & \theta & ::= & \mathsf{opcon}\ \boldsymbol{op}\ \mathsf{of}\ \kappa_{\text{idx}}\ (\tau_{\text{def}}) \mid \theta; \theta \\
 & & & \\
\textbf{external terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid \textsc{Tycon}.\boldsymbol{op}[\tau_{\text{idx}}](e_1; \ldots; e_n) \\
 & & & \\
\textbf{internal terms} & \iota & ::= & x \mid \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \iota \mid \lambda x{:}\sigma.\iota \mid \iota_1\ \iota_2 \mid \bar{z} \mid \iota_1 \oplus \iota_2 \mid \mathsf{if}\ \iota_1 \equiv_{\mathbb{Z}} \iota_2\ \mathsf{then}\ \iota_3\ \mathsf{else}\ \iota_4 \\
 & & \mid & () \mid (\iota_1, \iota_2) \mid \mathsf{fst}(\iota) \mid \mathsf{snd}(\iota) \mid \mathsf{inl}[\sigma_2](\iota_1) \mid \mathsf{inr}[\sigma_1](\iota_2) \mid \mathsf{case}(e; x.e_1; x.e_2) \\
\textbf{internal types} & \sigma & ::= & \sigma_1 \rightharpoonup \sigma_2 \mid \mathbb{Z} \mid 1 \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \\
 & & & \\
\textbf{type-level terms} & \tau & ::= & \mathbf{t} \mid \lambda\mathbf{t}{:}\kappa.\tau \mid \tau_1\ \tau_2 \mid []_\kappa \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1; \tau_2; \mathbf{h}, \mathbf{t}, \mathbf{r}.\tau_3) \\
 & & \mid & \bar{z} \mid \tau_1 \oplus \tau_2 \mid \mathit{label} \mid () \mid (\tau_1, \tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau) \\
 & & \mid & \mathsf{inl}[\kappa_2](\tau_1) \mid \mathsf{inr}[\kappa_1](\tau_2) \mid \mathsf{case}(\tau; x.\tau_1; x.\tau_2) \\
\text{equality} & & \mid & \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \\
\text{types} & & \mid & \textsc{Tycon}[\tau] \mid \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{derivates} & & \mid & \llbracket \tau_{\text{itm}}\ \mathsf{as}\ \tau_{\text{ty}} \rrbracket \mid \mathsf{let}\ \llbracket \mathbf{x}\ \mathsf{as}\ \mathbf{t} \rrbracket = \tau\ \mathsf{in}\ \tau_1 \\
\text{reified IL} & & \mid & \triangleright(\bar{\iota}) \mid \blacktriangleright(\bar{\sigma}) \\
 & \bar{\iota} & ::= & x \mid \mathsf{fix}\ x{:}\bar{\sigma}\ \mathsf{is}\ \bar{\iota} \mid \cdots \mid \triangleleft(\tau) \mid \mathsf{trans}(\llbracket \tau_{\text{itm}}\ \mathsf{as}\ \tau_{\text{ty}} \rrbracket) \\
 & \bar{\sigma} & ::= & \bar{\sigma}_1 \rightharpoonup \bar{\sigma}_2 \mid \cdots \mid \blacktriangleleft(\tau) \mid \mathsf{rep}(\tau) \\
 & & & \\
\textbf{kinds} & \kappa & ::= & \kappa_1 \to \kappa_2 \mid \mathsf{list}[\kappa] \mid \mathbb{Z} \mid \mathsf{Lbl} \mid 1 \mid \kappa_1 \times \kappa_2 \mid \kappa_1 + \kappa_2 \mid \mathsf{Ty} \mid \mathsf{D} \mid \mathsf{ITm} \mid \mathsf{ITy}
\end{array}
$$

Figure 7: Syntax of Core @λ. Here, x ranges over external and internal language variables, $\mathbf{t}$ ranges over type-level variables, TYCON ranges over type constructor names, ***op*** ranges over operator constructor names, $\bar{z}$ ranges over integer literals, *label* ranges over label literals (see text) and $\oplus$ ranges over the standard total binary operations over integers (e.g. addition). The form $\mathsf{trans}(\llbracket \tau_{\text{itm}}\ \mathsf{as}\ \tau_{\text{ty}} \rrbracket)$ is used internally by the semantics (it need not be supported by a parser).

language that provides few strictly enforced semantic guarantees makes giving strong metatheoretic guarantees about Ace more difficult. Ace does include checks similar to those in @λ that catch most accidental problems but they can be intentionally subverted by untrusted extension providers using the dynamic metaprogramming features of Python. We conjecture, but do not plan to show in this thesis, that it would be possible to use Ace to bootstrap a statically-typed subset of Python that, if used to compile extension logic, would be sufficient to provide theoretical guarantees comparable to @λ (building, perhaps, on recent work that resulted in a mechanized semantics for Python [**?**]).

5 @λ

In this section, we will develop an "actively typed" version of the simply-typed lambda calculus with simply-kinded type-level computation called @λ. More specifically, the level of types, τ, will itself form a simply-typed lambda calculus. *Kinds* classify type-level terms in the same way that types conventionally classify expressions. Types become just one kind of type-level value (which we will write Ty, though it is also variously written $\star$, `T` and `Type` in various settings). Rather than there being a fixed set of type constructors, we allow the programmer to declare new type constructors, and give the static and dynamic semantics of their associated operators, by writing type-level functions. In the semantics for this calculus, our kind system combined with techniques borrowed from the typed compilation literature and a form of type abstraction allow us to prove strong type safety, decidability and conservativity theorems.

The syntax of @λ is given in Fig. 7. An example of a program containing an extension that implements the static and dynamic semantics of Gödel's **T** is given in Fig. 8. We do not, of course, deny that natural numbers can be strongly encoded with a similar usage

```
tycon $\textsc{Nat}$ of 1 {schema $\lambda$idx:1.$\blacktriangleright(\mathbb{Z})$;
  opcon *z* of 1 ($\lambda$idx:1.$\lambda$args:list[D].**if_empty args** $[\![\triangleright(0)$ as $\textsc{Nat}[()]]\!]$);
  opcon *s* of 1 ($\lambda$idx:1.$\lambda$args:list[D].
    **pop_final args** $\lambda$x:ITm.$\lambda$t:Ty.
    **check_type t** $\textsc{Nat}[()]$ $[\![\triangleright(\triangleleft(\mathbf{x})+1)$ as $\textsc{Nat}[()]]\!]$);
  opcon *rec* of 1 ($\lambda$idx:1.$\lambda$args:list[D].
    **pop args** $\lambda$x1:ITm.$\lambda$t1:Ty.$\lambda$args:list[D].
    **pop args** $\lambda$x2:ITm.$\lambda$t2:Ty.$\lambda$args:list[D].
    **pop_final args** $\lambda$x3:ITm.$\lambda$t3:Ty.
    **check_type t1** $\textsc{Nat}[()]$ (
    **check_type t3** $\textsc{Arrow}[(\textsc{Nat}[()], \textsc{Arrow}[(\mathbf{t2}, \mathbf{t2})])]$
    $[\![\triangleright(($fix $f{:}\mathbb{Z} \rightharpoonup$ rep(t2) is $\lambda x{:}\mathbb{Z}$.
      if $x \equiv_{\mathbb{Z}} 0$ then $\triangleleft(\mathbf{x2})$ else $\triangleleft(\mathbf{x3})\ (x-1)\ (f\ (x-1)))\ \triangleleft(\mathbf{x1}))$ as t2$]\!]$))
};
let **nat** : Ty = $\textsc{Nat}[()]$ in
let *plus* : $\textsc{Arrow}[(\mathbf{nat}, \textsc{Arrow}[(\mathbf{nat}, \mathbf{nat})])] = \lambda x{:}\mathbf{nat}.\lambda y{:}\mathbf{nat}.$
  $\textsc{Nat}.\textit{rec}[()](x; y; \lambda p{:}\mathbf{nat}.\lambda r{:}\mathbf{nat}.\textsc{Nat}.\textit{s}[()](r))$ in
let *two* : **nat** = $\textsc{Nat}.\textit{s}[()](\textsc{Nat}.\textit{s}[()](\textsc{Nat}.\textit{z}[()]()))$ in
$\textsc{Arrow}.\textit{ap}[()](\textit{plus}; \textsc{Arrow}.\textit{ap}[()](\textit{two}; \textit{two}))$
```

Figure 8: Gödel's **T** in @λ, used to calculate 2+2. The simple helper functions **if_empty**, **pop**, **pop_final**, **check_type** all have return kind D + 1 (see text). Their definitions can be found in the appendix. We use let to bind both external and type-level variables for clarity of presentation; these can be eliminated by manually performing the indicated substitutions (or added to the semantics).

and asymptotic performance profile (but with potentially relevant additional function call overhead) by existing means (e.g. by an abstract type). We will provide more sophisticated examples where this is not the case below.

5.1 Programs

A program, ρ, consists of a series of static declarations followed by an external term, e. The syntax for external terms contains three core forms: variables, functions, and a form for all other operator invocations, which we will discuss below. Compiling a program consists of first *kind checking* its static declarations and type-level terms then *type checking* the external term and, simultaneously, *translating* it to a term, ι, in a *typed internal language*. In our calculus, the internal language contains partial functions (via the generic fixpoint operator of Plotkin's PCF [?]), simple product and sum types and a base type of integers (to make our example interesting and give a nod to practicality on contemporary machines). In practice, the internal language could be any intermediate language for which type safety and decidability of typechecking are known, as @λ requires these to establish the corresponding theorems about the language as a whole.

The *active typing judgement* relates an external term to a *type* and an internal term, called its *translation*, under *typing context* Γ and *constructor context* Φ:

$$\Gamma \vdash_\Phi e : \tau \Longrightarrow \iota$$

This judgement follows standard conventions for the typing judgement of a simply-typed lambda calculus in most respects. The typing context Γ maps variables to types and obeys standard structural properties. The key novelties here are that the available type and operator constructors are not fixed by the language, but rather are tracked by the constructor context, Φ, and that a translation is simultaneously derived. The dynamic semantics of external terms are defined by their translation to the internal language. For this judgement, the two contexts and the external term can be thought of as input, while the type and translation are output (we do not consider a bidirectional approach in this work, though this may be a promising avenue for future research).

5.2 Types

User-defined type constructors are declared at the top-level of a program (or package, in a practical implementation) using tycon. Each type constructor in the program must have a unique name, written e.g. NAT. We do not intrinsically consider the issue of namespacing, under the assumption that globally unique names can be generated by some extrinsic mechanism (e.g. a URI-based scheme). A type constructor must also declare an *index kind*, κ_{idx}. Types themselves are type-level values of kind Ty and are introduced by applying a previously declared type constructor to a type-level term of its index kind, $\text{TYCON}[\tau_{\text{idx}}]$. The kind Ty also has an elimination form: a type can be case analyzed against a type constructor in scope to extract its index. Like open datatypes, there is no longer a notion of exhaustiveness. To maintain totality, we require the default case.

To permit the implementation of interesting type systems, the type-level language includes several other kinds of data alongside types. We lift several standard functional constructs to the type level: unit (1), binary sums ($\kappa_1 + \kappa_2$), binary products ($\kappa_1 \times \kappa_2$), lists ($\mathsf{list}[\kappa]$) and integers ($\mathbb{Z}$). We also include labels (Lbl), written in a slanted font, e.g. *myLabel*, which are atomic string-like values that can only be compared, here only for equality, and play a distinguished role in the expanded syntax, as we will later discuss. Our first example, NAT, is indexed trivially, i.e. by unit kind, 1, because there is only one natural number type, $\text{NAT}[()]$, but we will show examples of type constructors that are indexed in more interesting ways in later portions of this work. For example, TUPLE is indexed by a list of types and LABELEDTUPLE is indexed by a list pairing labels with types.

Two type-level terms of kind Ty are equivalent if they apply the same constructor, identified by name, to equivalent indices. Going further, we ensure that deciding type equivalence requires only checking for syntactic equality after evaluation to normal form by imposing the restriction that type constructors can only be indexed by kinds for which equivalence can be decided in this way. All combinations of kinds introduced thus far have this property and this is sufficient for many interesting examples. We leave introducing a richer notion of equivalence of types that extension providers can extend as future work, as the metatheoretic guarantee that typing respects type equivalence would be quite a bit more complex in such a setting.

Type constructors are not first-class; they do not themselves have arrow kind as in some kind systems (e.g. [?]; Ch. 22 of *PFPL* describes a related system [20]). The type-level language does, however, include total functions of conventional arrow kind, $\kappa_1 \to \kappa_2$. Type constructor application can be wrapped in a type-level function to emulate first-class type constructors (and indeed, such a wrapper could be generated automatically, though we avoid this for simplicity in our semantics). Equivalence at arrow kind does not coincide with β-equivalence, so type-level functions cannot appear in type indices. Our treatment of equivalence in the type-level language is thus quite similar to the treatment of term-level equality using "equality types" in a language like Standard ML. Indeed, one might imagine that a practical implementation of our calculus would start with a pure, total subset of the term language of ML to construct the type-level language (consistent with its early development as a "metalanguage" and, as we will see, the role of our type-level language as a DSL for implementing type systems and translators, which functional languages are widely considered well-suited for as-is). We leave the development of such an implementation (and of bootstrapping type-level and internal languages that are themselves user-defined or extensible) as areas for future work.

cite

5.3 Operators

User-defined operator constructors are declared using opcon. For reasons that we will discuss, our calculus associates every operator constructor with a type constructor. The *fully-qualified name* of the operator constructor, e.g. $\textsc{Nat}.\boldsymbol{z}$, must be unique. Operator constructors are indexed by type-level values, like type constructors, and so also declare an index kind κ_{idx}. In our first example, all the operator constructors are indexed trivially, but later examples will use more interesting indices. For example, the projection operators for tuples and labeled tuples use numbers and labels as indices, respectively. Note that neither operator constructors nor operators are first-class type-level values (we plan to investigate lifting the latter restriction) and we do not impose any restrictions on their index kinds.

In the external language, an operator is selected and invoked by applying an operator constructor to a type-level index and $n \geq 0$ *arguments*, $\textsc{Tycon}.\boldsymbol{op}[\tau_{\text{idx}}](e_1; \ldots; e_n)$. For example, on line 18 of Fig. 8, we see the operator constructors $\textsc{Nat}.\boldsymbol{z}$ and $\textsc{Nat}.\boldsymbol{s}$ being invoked to compute two.[5] To derive the active typing judgement for this form, the semantics invokes the operator constructor's *definition*: a type-level function that must examine the provided operator index and the recursively determined *derivates* of the arguments to determine a derivate for the operation as a whole, or indicate an error. A derivate is a type-level representation of the result of deriving the active typing judgement[6]: an internal term, called the translation, paired with a type. Derivates have kind D and introductory form $[\![\tau_{\text{trans}} \text{ as } \tau_{\text{ty}}]\!]$. The elimination form $\mathsf{let}\ [\![\mathbf{x} \text{ as } \mathbf{t}]\!] = \tau \text{ in } \tau'$ extracts the translation and type from the derivate τ, binding it to $\mathbf{x}$ and $\mathbf{t}$ respectively in τ'. Because constructing a derivate is not always possible (e.g. when there is a type error in the client's code, or an invalid index was provided), the return kind of the function is an "option kind", $\mathsf{D} + \mathsf{1}$, where the trivial case indicates a statically-detected error in the client's program. In practice, it would instead require providers to report information about the precise location of the error and an appropriate error message.

A translation, as we have said, is an internal term. To construct an internal term using a type-level function, as we must do to construct a derivate, we must expose a type-level representation of the internal language. A *quoted internal term* is a type-level value of kind ITm with introductory form $\rhd(\bar{\iota})$ and a *quoted internal type* is a type-level value of kind ITy with introductory form $\blacktriangleright(\bar{\sigma})$. Neither kind has an elimination form. Instead, the syntax for the quoted internal language includes complementary *unquote forms* $\lhd(\tau)$ and $\blacktriangleleft(\tau)$ that permit the interpolation of another quoted term or type, respectively, into the one being formed. Interpolation is capture-avoiding, as our semantics will clarify. We have now described all the kinds in our language.

The definition of $\textsc{Nat}.\boldsymbol{z}$ is quite simple: it returns the derivate $[\![\rhd(0) \text{ as } \textsc{Nat}[()]]\!]$ if no arguments were provided, and indicates an error (an *arity error*, though we do not distinguish this in our calculus) otherwise. This is done by calling a simple helper function, **is_empty** : $\mathsf{list}[\mathsf{D}] \rightarrow \mathsf{D} \rightarrow (\mathsf{D} + \mathsf{1})$, that selects the appropriate case of the sum given the derivate that should be produced if the list is indeed empty. The definition of $\textsc{Nat}.\boldsymbol{s}$ is only slightly more complex, because it requires inspecting a single argument. The helper function **pop_final** : $\mathsf{list}[\mathsf{D}] \rightarrow (\mathsf{Ty} \rightarrow \mathsf{ITm} \rightarrow (\mathsf{D} + \mathsf{1})) \rightarrow (\mathsf{D} + \mathsf{1})$ "pops" a derivate from the head of the list and, if no other arguments remain, passes its translation and type to the "continuation", returning the error case otherwise. The continuation checks if the argument type is equal to $\textsc{Nat}[()]$ using **check_type** : $\mathsf{Ty} \rightarrow \mathsf{Ty} \rightarrow \mathsf{D} \rightarrow (\mathsf{D} + \mathsf{1})$, which operates similarly. If these arity and type checks succeed, the resulting derivate is composed by adding one to the translation of the argument, passed into the continuation as $\mathbf{x}$ in our example, and pairing it with the natural number type: $[\![\rhd(\lhd(\mathbf{x}) + 1) \text{ as } \textsc{Nat}[()]]\!]$. We will return to the definition of $\textsc{Nat}.\boldsymbol{rec}$ after in the next subsection.

[5]Although our focus here is entirely on semantics, a brief note on syntax: in the expanded syntax, the trivial indices and empty argument lists can be omitted, so we could write `Nat.s(Nat.s(Nat.z))`. With the ability to "open" a type's operators into the context, we could shorten this still to `s(s(z))`. Alternatively, with the ability to define a TSL in a manner similar to that in Sec. 3, we might instead just write `2`.

[6]Technically, the abstract active typing judgement, which is similar in form and we will introduce shortly.

By writing the typechecking and translation logic in this way, as a total function, we are taking a rather "implementation-focused view" rather than attempting to extract such a function from a declarative specification. That is, we leave to the provider (or to a program generator or kind-specific language that transforms such a specification into an operator definition) the problem of finding a deterministic algorithm that adequately implements their intended semantics, allowing us to prove decidability of typechecking for the language as a whole by essentially just citing the termination theorem for the type-level language. We also avoid difficulties with error reporting in practice. Rob Simmons' undergraduate thesis has a good discussion of these issues [**?**].

Because the input to an operator definition is the recursively determined derivate of the argument in the same context as the operator appears in, our mechanism does not presently permit the definition of operator constructors that bind variables themselves or require a different form of typing judgement (e.g. additional forms of contexts). This is also why the λ operator constructor needs to be built in. It is the only operator in our language that can be used to bind variables. The type constructor $\textsc{Arrow}$ can be defined from within the language (and is included in the "prelude" constructor context), but only defines the operator constructor, ***ap***. These two operators translate to their corresponding forms in the internal language directly, as we will discuss below. We leave adding the ability to declare and manipulate new contexts as future work.

5.4 Representational Consistency Implies Type Safety

In our example, natural numbers are represented internally as integers. Were this not the case – if, for example, we added an operator constructor $\textsc{Nat}.\boldsymbol{z2}$ that produced the derivate $[\![\rhd(\mathsf{inl}[\mathbb{Z}](()))\ \mathsf{as}\ \textsc{Nat}[()]]\!]$, then there would be two different internal types, $\mathbb{Z}$ and $1 + \mathbb{Z}$, associated with a single external type, $\textsc{Nat}[()]$. This makes it impossible to operate compositionally on the translation of an external term of type $\textsc{Nat}[()]$, so our implementation of $\textsc{Nat}.\boldsymbol{s}$ would produce ill-typed translations in some cases but not others. Similarly, we wouldn't be able to write functions over all natural numbers because there would not be a well-typed translation to give to such a function.

To reason compositionally about the semantics of well-typed external terms when they are given meaning by translation to a typed internal language, we must have the following property: for every type, τ, there must exist an internal type, σ, called its *representation type*, such that the translation of every external term of type τ has internal type σ. This principle of *representational consistency* arises essentially as a strengthening of the inductive hypothesis necessary to prove that all well-typed external terms translate to well-typed internal terms, precisely because operators like λ and $\textsc{Nat}.\boldsymbol{s}$ are defined compositionally. It is closely related to the concept of *type-preserving compilation* developed by Morrisett et al. for the TIL compiler for Standard ML [**?**]. cite

It is easy to show by induction that, under a "closed-world assumption" where the only available operators are $\textsc{Nat}.\boldsymbol{z}$ and $\textsc{Nat}.\boldsymbol{s}$, the representation type of $\textsc{Nat}[()]$ is $\mathbb{Z}$. If we can maintain this under an "open-world assumption" (requiring that, for example, applications of operator constructors like $\textsc{Nat}.\boldsymbol{z2}$, above, are not well-typed), and we target a type safe internal language, then we will achieve type safety: well-typed external terms cannot go wrong, because they always translate to well-typed internal terms, which cannot go wrong. For the semantics to ensure that representational consistency is maintained by all operator definitions, we require that each type constructor must declare a *representation schema* with the keyword schema. This must be a type-level function of kind $\kappa_{\text{idx}} \to \mathsf{ITy}$, where κ_{idx} is the index kind of the type constructor. When the compiler needs to determine the representation type of the type $\textsc{Tycon}[\tau_{\text{idx}}]$ it simply applies the representation schema of $\textsc{Tycon}$ to τ_{idx}.

As described above, the kind ITy has introductory form $\blacktriangleright(\bar{\sigma})$ and no elimination form. There are two forms in $\bar{\sigma}$ that do not correspond to forms in σ that allow quoted internal types to be formed compositionally:

1. ◀(τ), already described, unquotes (or "interpolates") the quoted internal type τ
2. `rep`(τ) refers to the representation type of type τ (we will see in the next subsection why this needs to be in the syntax for $\bar{\sigma}$ and not directly in the type-level language)

These additional forms are not needed by the representation schema of Nat because it is trivially indexed. In Fig. **??**, we show an example of the type constructor Tuple, implementing the semantics of n-tuples by translation to nested binary products. Here, the representation schema requires referring to the representations of the tuple's constituent types, given in the type index.

add this example from appendix of ESOP

Operator constructor definitions might also need to refer to the representation of a type. We see this in the definition of the recursor on natural numbers, Nat.***rec***. After checking the arity and extracting the types and translations of its three arguments, it produces a derivate that implements the necessarily recursive dynamic semantics using a fixpoint computation in the internal language. This fixpoint computation produces a result of some arbitrary type, **t2**. We cannot know what the representation type of **t2** is, we refer to it abstractly using `rep`(**t2**). The translation is well-typed no matter what the representation type is. In other words, a proof of representational consistency of the derivate produced by the definition of Nat.***rec*** is parametric (in the metamathematics) over the representation type of **t2**. This leads us into the concept of conservativity.

5.5 Parametricity Implies Conservativity

In isolation, our definition of natural numbers can be shown to be a strong encoding of the semantics of Gödel's **T**. That is, there is a bijection between the terms and types of the two languages and the typing judgements preserve this mapping. Moreover, we can show by a bisimulation argument that the translation produced by @λ implements the dynamic semantics of Gödel's **T**. We will provide more details on this later. A necessary lemma in the proof, however, is that the value of every translation of an external term of type Nat$[()]$ is a non-negative integer. This is needed to show that the fixpoint computation in the definition of Nat.***rec*** is terminating, as the recursor always does in **T**. A very similar lemma is needed to show that the translation of every external term of type Nat$[()]$ is equivalent to the translation that would be produced by some combination of applications of Nat.***z*** and Nat.***s*** (the analog to a canonical forms lemma in our setting). Fortunately, the definitions we have provided thus far admit such lemmas.

However, these theorems are all quite precarious because they rely on exhaustive induction over the available operators. As soon as we load another library into the program, these theorems may no longer be *conserved*. For example, the following operator declaration in some other type that we have loaded would topple our house of cards:

It can also be shown to maintain the invariant that natural numbers elaborate into *non-negative* internal integers. Note that because we are beginning with a simply-typed, simply-kinded formulation, these kinds of statements must be proven metatheoretically (as with other sorts of complex invariants in simply-typed languages). With a naive formulation of this mechanism, these theorems would be quite a bit more precarious, however. If another provider, perhaps a malicious one, declares a new operator constructor that introduces natural numbers, these theorems may not be *conserved*. For example, the following operator declaration is quite problematic:

In progress below

```
opcon badnat of 1 (λidx:1.λargs:list[Elab].⟦▽(-1) as Nat[()]⟧)
```

With this declaration, there is now a new and unexpected introductory form for the natural number type, and even worse, it violates our previous implementation invariants. A naive check for type preservation would not catch this problem: `-1` does have type $\mathbb{Z}$, it is simply not an integer that could have been emitted by the original definition of `Nat`.

To prevent this problem, we *hold the representation of a type abstract* outside of the operators explicitly associated with it. If `badnat` cannot know that the representation of

`Nat[()]` is $\mathbb{Z}$, then the above operator implementation cannot be correct. Because we cannot prove relational correctness properties from within a simply-typed/kinded calculus, our semantics enforces this by using a form of value abstraction, similar to that described by Zdancewic et. al [**?**]. In brief: the representation of a type, written $\mathsf{rep}(\tau_{\text{ty}})$, does not reduce further to the actual representation type (e.g. to $\mathbb{Z}$) when not in an operator definition associated with τ_{ty}. The only internal form that can be checked against $\mathsf{rep}(\tau_{\text{ty}})$ is the special form $\mathsf{trans}([\![\tau_{\text{itm}} \text{ as } \tau_{\text{ty}}]\!])$, an abstracted internal term corresponding to the internal term reflected by τ_{itm}. Rather than exposing it explicitly, it is wrapped so that it can't be checked against any type other than $\mathsf{rep}(\tau_{\text{ty}})$. A full description of this mechanism (and the others, described above) requires us to introduce the semantics in detail. A draft of the semantics of $@\lambda$ is available as a paper draft[7].

5.6 Expressiveness

The use cases permitted by this mechanism are not simply subsumed by a module system with support for abstract types and functors. Indeed, such a module system is an orthogonal concern in that it must sit atop a core type system. More particularly, the distinction is the fundamental one between functions and operators. Functions cannot examine type indices at compile-time to determine a return type and implementation (or generate static error messages), as operators are able to do. It is encouraging that there are clear parallels between module systems with abstract types and type constructors with abstract schemas, and any full-scale language design based on this mechanism should certainly provide both mechanisms (with the former used in most cases). We plan to investigate this relationship more thoroughly, showing examples where abstract types are more clearly unsuitable, in the remainder of this work.

5.6.1 Remaining Tasks and Timeline

We are actively working on this mechanism and plan to submit a paper to ICFP 2014 on Mar. 1 based on a submission to ESOP 2014 that was not accepted. The following key tasks, other than clarifying the writing, remain to be completed:

1. Reviewers of the previous submission to ESOP found a problem with our treatment of internal type variables that needs to be corrected by threading a context through the type-level evaluation semantics.

2. We must more rigorously state and prove the key lemmas and theorems related to type preservation, type safety, decidability and conservativity.

3. We must develop examples that are more interesting than natural numbers. In particular, some interesting form of labeled product type that SML doesn't have (e.g. a record with prototypic dispatch) as well as sum types would be interesting.

4. We must consider whether properties related to termination can be conserved by this mechanism, because a fixpoint can be used to introduce a non-terminating expression of any type. This may be possible by being careful about how deabstraction interacts with fixpoints.

5. We must more clearly connect this work to the Harper-Stone semantics and other related work, and clarify the relationship to abstract types.

6. We must show how to add additional syntactic forms to the external language that defer to the generic operator invocation form in a type-directed manner. This connects the theory to the work on Ace, described below.

[7] `https://github.com/cyrus-/papers/tree/master/esop14`

6 Ace

The key choices that a language must make when supporting the mechanism described in the previous section are:

1. What is the semantics of the type-level language?
2. What is the syntax of the external language, and how should the syntactic forms dispatch to operator definitions?
3. What is the semantics of the internal language?

In @λ, the type-level language was a form of the simply-typed lambda calculus with a few common data types, and the internal language was a variant of PCF. The external language currently only includes direct dispatch by naming an operator explicitly. This is useful for the purposes of clarifying the foundations of our work.

In this section, we wish to explore a different point in this design space, with an eye toward practicality and expressiveness. In particular, we want to show how to implement the extension mechanism itself as a library within an existing, widely used language, solving a *bootstrapping problem* that has prevented other work on extensibility from being effective as a means to bring more research into practice. This language, called Ace, makes the following choices:

1. Python is used as the type-level language (and more generally, as a compile-time metalanguage).
2. Python's syntax is used for the external language. We will describe the dispatch protocol below.
3. The internal language can be user-defined, rather than being pre-defined as in @λ.

The choice of Python as the host language presents several challenges because we are attempting to embed an extensible static type system within a uni-typed (a.k.a. dynamically typed) language, without modifying its syntax in any way. We show that by leveraging Python's support for function quotations and by using its class-based object system to encode type constructors, we are able to accomplish this goal. Then, with a flexible statically-typed language under the control of libraries, we implement a variety of statically-typed abstractions, including common functional abstractions (e.g. inductive datatypes with pattern matching), low-level parallel abstractions (all of the OpenCL programming language), object systems and domain-specific abstractions (e.g. regular expression types, as described in the introduction). Ace can be used both as a standalone language and as a staged compilation environment from within Python.

6.0.2 Language Design and Usage

Listing 1 shows an example of an Ace file. As promised, the top level of an Ace file is written directly in Python, requiring no modifications to the language (versions 2.6+ or 3.3+) nor features specific to CPython (so Ace supports alternative implementations like Jython, IronPython and PyPy). This choice pays dividends on line 1: Ace's package system is Python's package system, so Python's build tools (e.g. `pip`) and package repostories (e.g. `PyPI`) are directly available for distributing Ace libraries.

The top-level statements in an Ace file, like the `print` statement on line 10, are executed at compile-time. That is, Python serves as the *compile-time metalanguage* of Ace. Functions containing run-time behavior, like `map`, are annotated as Ace functions and are then governed by a semantics that differs substantially from Python's (in ways that we will describe below). But Ace functions share Python's syntax. As a consequence, users of Ace benefit from an ecosystem of well-developed tools that work with Python syntax, including parsers, code highlighters, editor modes, style checkers and documentation generators.

Listing 1 [`listing1.py`] A generic imperative data-parallel higher-order map function targeting OpenCL.

```
1   import ace, examples.clx as clx
2
3   @ace.fn(clx.base, clx.opencl)
4   def map(input, output, f):
5      thread_idx = get_global_id()
6      output[thread_idx] = f(input[thread_idx])
7      if thread_idx == 0:
8         printf("Hello, run-time world!")
9
10  print "Hello, compile-time world!"
```

6.0.3 OpenCL as an Active Library

The code in this section uses `clx`, an example of an library that implements the semantics of the OpenCL programming language, and extends it with some additional useful types, using Ace. Ace itself has no built-in support for OpenCL.

To briefly review, OpenCL provides a data-parallel SPMD programming model where developers define functions, called *kernels*, for execution across thousands of threads on *compute devices* like GPUs or multi-core CPUs [19]. Each thread has access to a unique index, called its *global ID*. Kernel code is written in the OpenCL kernel language, a somewhat simplified variant of C99 extended with some new primitive types and operators, which we will describe as needed in our examples below.

6.0.4 Generic Functions

Lines 3-4 introduce `map`, an Ace function of three arguments that is governed by the *active base* referred to by `clx.base` and targets the *active target* referred to by `clx.opencl`. The active target determines which language the function will compile to (here, the OpenCL kernel language) and mediates code generation. The active target plays an analagous role to the internal language of @λ.

The body of this function, on lines 5-8, does not have Python's semantics. Instead, it will be governed by the active base together with any *active types* used within it. No types have yet been assigned, however. Because our type system is extensible, the code inside could be meaningful for many different assignments of types to the arguments (a form of *ad hoc polymorphism*). We call functions awaiting types, like `map`, *generic functions*. Once types have been assigned, they are called *concrete functions*.

Generic functions are represented at compile-time as instances of `ace.GenericFn` and consist of an abstract syntax tree, an active base, an active target and a read-only copy of the Python environment that they were defined within. The purpose of the *decorator* on line 3 is to replace the Python function on lines 4-8 with an Ace generic function having the same syntax tree and environment and the provided active base and active target. A decorator in Python is simply syntactic sugar that applies another function directly to the function being decorated [4]. In other words, line 3 could be replaced by the following statement on line 9: `map = ace.fn(clx.base, clx.opencl)(map)`. Ace extracts the abstract syntax tree for `map` using the Python standard library packages `inspect` (to retrieve its source code) and `ast` (to parse it into a syntax tree). The ability to extract a function's syntax tree and inspect its closure directly are the two key ingredients for implementing a mechanism like this as a library within another language.

6.0.5 Concrete Functions and Explicit Compilation

To compile a generic function to a particular *concrete function*, a type must be provided for each argument, and typechecking and elaboration must then succeed. Listing 2 shows how to explicitly provide type assignments to `map` using the subscript operator (implemented

Listing 2 [`listing2.py`] The generic `map` function compiled to map the `negate` function over two types of input.

```python
1  import listing1, ace, examples.clx as clx
2
3  @ace.fn(clx.base, clx.opencl)
4  def negate(x):
5    return -x
6
7  T1 = clx.Ptr(clx.global_, clx.float)
8  T2 = clx.Ptr(clx.global_, clx.Cplx(clx.int))
9  TF = negate.ace_type
10
11 map_neg_f32 = listing1.map[[T1, T1, TF]]
12 map_neg_ci32 = listing1.map[[T2, T2, TF]]
```

Listing 3 Compiling `listing2.py` using the `acec` compiler.

```
> acec listing2.py
Hello, compile-time world!
[acec] listing2.cl successfully generated.
```

using Python's operator overloading mechanism). We do so two times in Listing 2, on lines 11 and 12. Here, `T1`, `T2`, `TF`, `clx.float`, `clx.int` and `negate.ace_type` are types and `clx.Ptr` and `clx.Cplx` are type constructors. We will discuss these in the next section.

Concrete functions like `map_neg_f32` and `map_neg_ci32` are instances of `ace.ConcreteFn`. They consist of a *typed* abstract syntax tree, an elaboration into the target language and a reference to the originating generic function.

To produce an output file from an Ace "compilation script" like `listing2.py`, the command `acec` can be invoked from the shell, as shown in Listing 3. The result of compilation is the OpenCL file shown in Listing 4. The `acec` compiler (a simple Python script) operates in two stages:

1. Executes the provided Python file (`listing3.py`).

2. Extracts the elaborations from concrete functions and other top-level constructs that define elaborations (e.g. types requiring declarations) in the final Python environment. This may produce one or more files, depending on which active targets were used (here, just `listing3.cl`, but a web framework built upon Ace might produce separate HTML, CSS and JavaScript files).

We will show in the thesis, but omit in this proposal, that for targets with Python bindings, such as OpenCL, CUDA, C, Java or Python itself, generic functions can be executed directly, without any of the explicit compilation steps in Listings 2 and 3. This represents a form of staged compilation. In this setting, the dynamic type of a Python value determines, at the point of invocation, a static type assignment for the argument of an Ace generic function.

6.0.6 Types

Lines 7-9 of Listing 2 construct the types used to generate concrete functions from the generic function `map` on lines 11 and 12. In Ace, types are themselves values that can be manipulated at compile-time. Python is thus Ace's type-level language. More specifically, types are instances of a Python class that implements the `ace.ActiveType` interface. Implementing this interface is analagous to defining a new type constructor in @λ.

As Python values, types can be assigned to variables when convenient (removing the need for facilities like `typedef` in C or `type` in Haskell). Types, like all compile-time objects derived from Ace base classes, do not have visible state and operate in a referentially transparent manner (by constructor memoization, which we do not detail here).

Listing 4 [`listing2.cl`] The OpenCL file generated by Listing 3.

```
1  float negate__0_(float x) {
2    return x * -1;
3  }
4
5  kernel void map_neg_f32(global float* input,
6      global float* output) {
7    size_t thread_idx = get_global_id(0);
8    output[thread_idx] = negate__0_(input[thread_idx]);
9    if (thread_idx == 0) {
10     printf("Hello, run-time world!");
11   }
12 }
13
14 int2 negate__1_(int2 x) {
15   return (int2)(x.s0 * -1, x.s1);
16 }
17
18 kernel void map_neg_ci32(global int2* input,
19     global int2* output) {
20   size_t thread_idx = get_global_id(0);
21   output[thread_idx] = negate__1_(input[thread_idx]);
22   if (thread_idx == 0) {
23     printf("Hello, run-time world!");
24   }
25 }
```

The type named `T1` on line 7 directly implements the logic of the underlying OpenCL type `global float*`: a pointer to a 32-bit floating point number stored in the compute device's global memory (one of four address spaces defined by OpenCL [19]). It is constructed by applying `clx.Ptr`, which is an Ace type constructor corresponding to pointer types, to a value representing the address space, `clx.global_`, and the type being pointed to. That type, `clx.float`, is in turn the `clx` type corresponding to `float` in OpenCL (which, unlike in C99, is always 32 bits). The `clx` library contains a full implementation of the OpenCL type system (including complexities, like promotions, inherited from C99). Ace is *unopinionated* about issues like memory safety and the wisdom of such promotions. We will discuss how to implement, as libraries, abstractions that are higher-level than raw pointers, or simpler numeric types, but Ace does not prevent users from choosing a low level of abstraction or "interesting" semantics if the need arises (e.g. for compatibility with existing libraries). We also note that we are being more verbose than necessary for the sake of pedagogy. The `clx` library includes more concise shorthand for OpenCL's types: `T1` is equal to `clx.gp(clx.f32)`.

The type `T2` on line 8 is a pointer to a *complex integer* in global memory. It does not correspond directly to a type in OpenCL, because OpenCL does not include primitive support for complex numbers. Instead, the type constructor `clx.Cplx` defines the necessary logic for typechecking operations on complex numbers and elaborating them to OpenCL. This constructor is parameterized by the numeric type that should be used for the real and imaginary parts, here `clx.int`, which corresponds to 32-bit OpenCL integers. Arithmetic operations with other complex numbers, as well as with plain numeric types (treated as if their imaginary part was zero), are supported. When targeting OpenCL, Ace expressions assigned type `clx.Cplx(clx.int)` are compiled to OpenCL expressions of type `int2`, a *vector type* of two 32-bit integers (a type that itself is not inherited from C99). This can be observed in several places on lines 4.14-4.21. This choice is merely an implementation detail that can be kept private to `clx`. An Ace value of type `clx.int2` (that is, an actual OpenCL vector) *cannot* be used when a `clx.Cplx(clx.int)` is expected (and attempting to do so will result in a static type error).

The type `TF` on line 9 is extracted from the generic function `negate` constructed in Listing 2. Generic functions, according to Sec. 6.0.4, have not yet had a type assigned to them, so it may seem perplexing that we are nevertheless extracting a type from `negate`. Although a conventional arrow type cannot be assigned to `negate`, we can give it a *singleton type*: a type that simply means "this expression is the *particular* generic function `negate`". This type could also have been explicitly written as `ace.GenericFnType(listing2.negate)`. During typechecking and translation of `map_neg_f32` and `map_neg_ci32`, the call to `f` on line 6 of Listing 1 uses the type of the argument to generate a concrete function from the generic function that inhabits the singleton type of `f` (`negate` in both cases shown). This is why there are two versions of `negate` in the output in Listing 4. In other words, types *propagate* into generic functions – we didn't need to compile `negate` explicitly. In effect, this scheme enables higher-order functions even when targeting languages, like OpenCL, that have no support for higher-order functions (OpenCL, unlike C99, does not support function pointers). Interestingly, because they have a singleton type, they are higher-order but not first-class functions. That is, the type system would prevent you from creating a heterogeneous list of generic functions. Concrete functions, on the other hand, can be given both a singleton type and a true function type. For example, `listing2.negate[[clx.int]]` could be given type `ace.Arrow(clx.int, clx.int)`. The base determines how to convert the Ace arrow type to an arrow type in the target language (e.g. a function pointer for C99, or an integer that indexes into a jump table constructed from knowledge of available functions of the appropriate type in OpenCL).

6.0.7 Extensibility

The core of Ace consists of about 1500 lines of Python code implementing its primary concepts: generic functions, concrete functions, active types, active bases and active targets. The latter three comprise Ace's extension mechanism. Extensions provide semantics to, and govern the compilation of, Ace functions, rather than logic in Ace's core.

Active types are the primary means for extending Ace with new abstractions. An active type, as mentioned previously, is an instance of a class implementing the `ace.ActiveType` interface. Listing 5 shows an example of such a class: the `clx.Cplx` class used in Listing 2, which implements the logic of complex numbers. The constructor takes as a parameter any numeric type in `clx` (line 5.2).

6.0.8 Dispatch Protocol

In a compiler for a monolithic language, there would be a *syntax-directed* protocol governing typechecking and translation. In a compiler written in a functional language, for example, one would declare datatypes that captured all forms of types and expressions, and the typechecker would perform exhaustive case analysis over the expression forms. That is, all the semantics are implemented in one place. The visitor pattern typically used in object-oriented languages implements essentially the same protocol. This does not work for an extensible language because new cases and logic need to be added *modularly*, in a safely composable manner.

Instead of taking a syntax-directed approach, the Ace compiler's typechecking and translation phases take a *type-directed approach*. When encountering a compound term (e.g. `e[e1]`), the compiler defers control over typechecking and translation to the active type of a designated subexpression (e.g. `e`) determined by Ace's fixed *dispatch protocol*. Below are examples of the choices made in Ace.

- Responsibility over **attribute access** (`e.attr`), **subscripting** (`e[e1]`), **calls** (`e(e1, ..., en)`) and **unary operations** (e.g. `-e`) is handed to the type recursively assigned to `e`.
- Responsibility over **binary operations** (e.g. `e1 + e2`) is first handed to the type assigned to the left operand. If it indicates a type error, the type assigned to the right operand is handed responsibility, via a different method call. Note that this operates like the corresponding rule in Python's *dynamic* operator overloading mechanism.

Listing 5 [in `examples/clx.py`] The active type family `Ptr` implements the semantics of OpenCL pointer types.

```python
class Cplx(ace.ActiveType):
  def __init__(self, t):
    if not isinstance(t, Numeric):
      raise ace.InvalidTypeError("<error message>")
    self.t = t

  def type_Attribute(self, context, node):
    if node.attr == 'ni' or node.attr == 'i':
      return self.t
    raise ace.TypeError("<error message>", node)

  def trans_Attribute(self, context, target, node):
    value_x = context.trans(node.value)
    a = 's0' if node.attr == 'ni' else 's1'
    return target.Attribute(value_x, a)

  def type_BinOp_left(self, context, node):
    return self._type_BinOp(context, node.right)

  def type_BinOp_right(self, context, node):
    return self._type_BinOp(context, node.left)

  def _type_BinOp(self, context, other):
    other_t = context.type(other)
    if isinstance(other_t, Numeric):
      return Cplx(c99_binop_t(self.t, other_t))
    elif isinstance(other_t, Cplx):
      return Cplx(c99_binop_t(self.t, other.t))
    raise ace.TypeError("<error message>", other)

  def trans_BinOp(self, context, target, node):
    r_t = context.type(node.right)
    l_x = context.trans(node.left)
    r_x = context.trans(node.right)
    make = lambda a, b: target.VecLit(
      self.trans_type(self, target), a, b)
    binop = lambda a, b: target.BinOp(
      a, node.operator, b)
    si = lambda a, i: target.Attribute(a, 's'+str(i))
    if isinstance(r_t, Numeric):
      return make(binop(si(l_x, 0), r_x), si(r_x, 1))
    elif isinstance(r_t, Cplx):
      return make(binop(si(l_x, 0), si(r_x, 0)),
        binop(si(l_x, 1), si(r_x, 1)))

  @classmethod
  def type_New(cls, context, node):
    if len(node.args) == 2:
      t0 = context.type(node.args[0])
      t1 = context.type(node.args[1])
      return cls(c99_promoted_t(t0, t1))
    raise ace.TypeError("<error message>", node)

  @classmethod
  def trans_New(cls, context, target, node):
    cplx_t = context.type(node)
    x0 = context.trans(node.args[0])
    x1 = context.trans(node.args[1])
    return target.VecLit(cplx_t.trans_type(target),
      x0, x1)

  def trans_type(self, target):
    return target.VecType(self.t.trans_type(target),2)
```

- Responsibility over **constructor calls** (`[t](e1, ..., en)`), where `t` is a *compile-time Python expression* evaluating to an active type, is handed to that type, by using Python's eval functionality to evaluate t. If `t` evaluates to a type constructor, like `clx.Cplx`, the type is first generated via a class method, as discussed below.

6.0.9 Typechecking

When typechecking a compound expression or statement, the Ace compiler temporarily hands control to the object selected by the dispatch protocol by calling the method `type_`X, where X is the name of the syntactic form, taken from the Python grammar [4] (appended with a suffix in some cases).

For example, if `c` is a complex number, then the operations `c.ni` and `c.i` extract its non-imaginary and imaginary components, respectively. These expressions are of the form `Attribute`, so the typechecker calls `type_Attribute` (line 7). This method receives the compilation context, `context`, and the abstract syntax tree of the expression, `node`, and must return a type assignment for it, or raise an `ace.TypeError` if there is an error. In this case, a type assignment is possible if the attribute name is either `"ni"` or `"i"`, and an error is raised otherwise (lines 8-10). We note that error messages are an important and sometimes overlooked facet of ease-of-use [27]. A common frustration with using general-purpose abstraction mechanisms to encode an abstraction is that they can produce verbose and cryptic error messages that reflect the implementation details instead of the semantics. Ace supports custom error messages.

Complex numbers also support binary arithmetic operations partnered with both other complex numbers and with non-complex numbers, treating them as if their imaginary component is zero. The typechecking rules for this logic is implemented on lines 17-29. Arithmetic operations are usually symmetric, so the dispatch protocol checks the types of both subexpressions for support. To ensure that the semantics remain deterministic in the case that both types support the binary operation, Ace asks the left first (via `type_BinOp_left`), asking the right (via `type_BinOp_right`) only if the left indicates an error. In either position, our implementation begins by recursively assigning a type to the other operand in the current context via the `context.type` method (line 24). If supported, it applies the C99 rules for arithmetic operations to determine the resulting type (via `c99_binop_t`, not shown).

Finally, a complex number can be constructed inside an Ace function using Ace's special constructor form: `[clx.Cplx](3,4)` represents $3 + 4i$, for example. The term within the braces is evaluated at *compile-time*. Because `clx.Cplx` evaluates not to an active type, but to a class, this form is assigned a type by handing control to the class object via the *class method* `type_New`. It operates as expected, extracting the types of the two arguments to construct an appropriate complex number type (lines 50-57), raising a type error if the arguments cannot be promoted to a common type according to the rules of C99 or if two arguments were not provided.

6.0.10 Translation

Once typechecking a method is complete, the compiler enters the translation phase, where terms in the target language are generated from Ace terms. Terms in the target language are generated by calling methods of the *active target* governing the function being compiled. The translation phase operates similarly to typechecking, using the dispatch protocol to invoke methods named `trans_`X. These methods have access to the context and node just as during typechecking, as well as the active target (named `target` here).

As seen in Listing 4, we are implementing complex numbers internally using OpenCL vector types, like `int2`. Let us look first at `trans_New` on lines 54-60, where new complex numbers are translated to vector literals by invoking `target.VecLit`. This will ultimately generate the necessary OpenCL code, as a string, to complete compilation (these strings are not directly manipulated by extensions, however, to avoid problems with, e.g. precedence). For it to be possible to reason compositionally about the correctness of compilation, all complex numbers must translate to terms in the target language that have a consistent target type. The `trans_type` method of the `ace.ActiveType` associates a type in the target language, here a vector type like `int2`, with the active type. Ace supports a mode where this *representational consistency* is dynamically checked during compilation (requiring that the active target know how to assign types to terms in the target language, which can be done for our OpenCL target as of this writing).

The translation methods for attributes (line 12) and binary operations (line 31) proceed in a straightforward manner. The context provides a method, `trans`, for recursively determining the translation of subexpressions as needed. Of note is that the translation methods can assume that typechecking succeeded. For example, the implementation of `trans_Attribute` assumes that if `node.attr` is not `'ni'` then it must have been `'i'` on line 14, consistent with the implementation of `type_Attribute` above it. Typechecking and translation are separated into two methods to emphasize that typechecking is not target-dependent, and to allow for more advanced uses, like type refinements and hypothetical typing judgements, that we do not describe here.

6.0.11 Active Bases

Each generic function is associated with an active base, which is an object implementing the `ace.ActiveBase` interface. The active base specifies the *base semantics* of that function. It controls the semantics of statements and expressions that do not have a clear "primary" subexpression for the dispatch protocol to defer to. A base is handed control over typechecking of statements and expressions in the same way as active types: via `type_`X and `trans_`X methods. Each function can have a different base.

Literals are the most prominent form given their semantics by an active base. Our example active base, `clx.base`, assigns integer literals the type `clx.int32` while floating point literals have type `clx.double`, consistent with the semantics of OpenCL and C99. The `clx.base` object is an instance of `clx.Base` that is pre-constructed for convenience. Alternative bases can be generated via different constructor arguments. For example, `clx.Base(flit_t=clx.float)` is a base semantics where floating point literals have type `clx.float`. This is useful because some OpenCL devices do not support double-precision numbers, or impose a significant performance penalty for their use. Indeed, to even use the `double` type in OpenCL, a flag must be toggled by a `#pragma` (The OpenCL target we have implemented inserts this automatically when needed, along with some other flags, and annotations like `kernel`). Similarly, for some applications, avoiding accidental overflows is more important than performance. Infinite-precision integers can be implemented as an active type (not shown) and the base can be asked to use it for numeric literals. In all cases, this occurs by inserting a constructor call form, as described above.

The base also initializes the context and governs the semantics of variables and assignment. This can also permit it to allow for the use of "built-in" operators that were not explicitly imported. The semantics of `get_global_id` and `printf` are provided by `clx.base`. In fact, the base provides extended versions of them (the former permits multi-dimensional global IDs, the latter supports typechecking format strings).A base which does not provide them as built-ins (requiring that they be imported explicitly) can be generated by passing the `primitives=False` flag.

6.0.12 Expressiveness

Thus far, we have focused mainly on the OpenCL target and shown examples of fairly low-level active types: those that implement OpenCL's primitives (e.g. `clx.Ptr`) and extend them in simple but convenient ways (e.g. `clx.Cplx`).

Listing 6 [`datatypes_t.py`] An example using statically-typed functional datatypes.

```
1  import ace, ace.python as py, examples.fp as fp
2
3  Tree = lambda name, a: fp.Datatype(name,
4    lambda tree: {
5      'Empty': fp.unit,
6      'Leaf': a,
7      'Node': fp.tuple(tree, tree)
8    })
9
10 DT = Tree('DT', py.dyn)
11
12 @ace.fn(py.Base(trailing_return=True))[[DT]]
13 def depth_gt_2(x):
14   x.case({
15     DT.Node(DT.Node(_), _): True,
16     DT.Node(_, DT.Node(_)): True,
17     _: False
18   })
19
20 @ace.fn(py.Base(main=True))[[]]
21 def __main__():
22   my_lil_tree = DT.Node(DT.Empty, DT.Empty)
23   my_big_tree = DT.Node(my_lil_tree, my_lil_tree)
24   assert not depth_gt_2(my_lil_tree)
25   assert depth_gt_2(my_big_tree)
```

But Ace has proven useful for more than low-level tasks like programming a GPU with OpenCL. We now describe some interesting extensions that implement the semantics of primitives drawn from a range of different language paradigms, to justify our claim that these mechanisms are highly expressive.

6.0.13 Growing a Statically-Typed Python Inside Ace

Ace comes with a target, base and type implementing Python itself: `ace.python.python`, `ace.python.base` and `ace.python.dyn`. These can be supplemented by additional active types and used as the foundation for writing statically-typed Python functions. These functions can either be compiled ahead-of-time to an untyped Python file for execution, or be immediately executed with just-in-time compilation, just like the OpenCL examples (not shown in this proposal).

6.0.14 Recursive Labeled Sums

Listing 6 shows an example of the use of statically-typed polymorphic recursive labeled sum types together with the Python implementation just described. It shows two syntactic conveniences that were not mentioned previously: 1) if no target is provided to `ace.fn`, the base can provide a default target (here, `ace.python.python`); 2) a concrete function can be generated immediately by providing a type assignment immediately in braces (in Python 3, a more convenient syntax is available). Listing 6 generates Listing 7.

Lines 6.3-6.11 define a function that generates a recursive algebraic datatype representing a tree given a name and another Ace type for the data at the leaves. This type implemented by the active type family `fp.Datatype`. A name for the datatype and the case names and types are provided programmatically on lines 6.3-6.8. To support recursive datatypes, the case names are enclosed within a lambda term that will be passed a reference to the datatype itself. These lines also show two more active type families: units (the type containing just one value), and immutable pairs. Line 6.13 calls this function with the Ace type for dynamic Python values, `py.dyn`, to generate a type, aliased `DT`. This type is implemented using class inheritance when the target is Python, as seen on lines 7.1-7.13. For C-like targets, a `union` type can be used (not shown).

The generic function `depth_gt_2` demonstrates two features. First, the base has been setup to treat the final expression in a top-level branch as the return value of the function,

Listing 7 `[datatypes.py]` The dynamically-typed Python code generated by running `acec datatypes_t.py`.

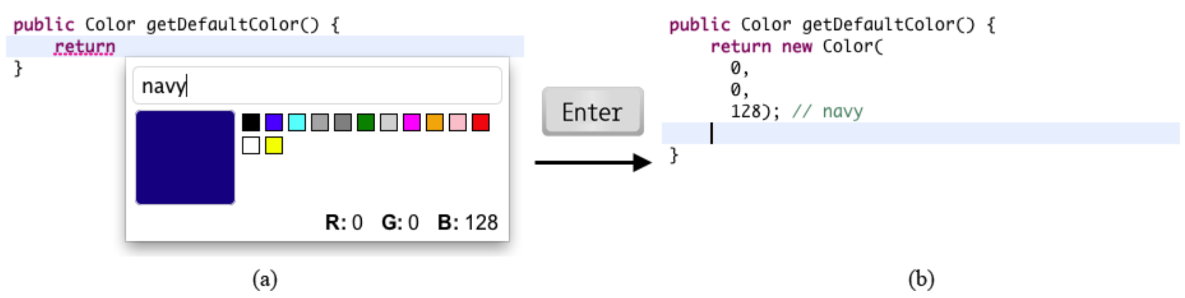
```python
class DT(object):
  pass

class DT_Empty(DT):
  pass

class DT_Leaf(DT):
  def __init__(self, data):
    self.data = data

class DT_Node(DT):
  def __init__(self, data):
    self.data = data

def depth_gt_2(x):
  if isinstance(x, DT_Node):
    if isinstance(x.data[0], DT_Node):
      r = True
    elif isinstance(x.data[1], DT_Node):
      r = True
  elif True:
    r = False
  return r

if __name__ == "__main__":
  my_lil_tree = DT_Node(DT_Empty(), DT_Empty())
  my_big_tree = DT_Node(my_lil_tree, my_lil_tree)
  assert not depth_gt_2(my_lil_tree)
  assert depth_gt_2(my_big_tree)
```

consistent with typical functional languages. Second, the `case` "method" on line 6.17 creatively reuses the syntax for dictionary literals to express a nested pattern matching construct. The patterns to the left of the colons are not treated as expressions (that is, a type and translation is never recursively assigned to them). Instead, the active type implements the standard algorithm for ensuring that the cases are exhaustive and not redundant [20]. If the final case were omitted, for example, the algorithm would statically indicate a warning (or optionally an error, not shown), just as in ML.

6.0.15 Remaining Tasks and Timeline

A paper on this work was recently rejected from PLDI 2014. We plan to resubmit this work to OOPSLA 2014, due on Mar. 25th, after completing the following tasks (largely after the Mar. 1 ICFP deadline):

- The reviewers asked for more details about the composition properties of extensions, the relationship to abstract data types, to work on staging and on how variables and contexts are treated. We will give more space to these issues in the new draft.
- We will provide more details, given the additional space available in an OOPSLA paper, of the examples other than OpenCL. We will consider using an example other than OpenCL to motivate the main body of the paper, so that the general distaste most PL researchers have for direct memory manipulation can be avoided.
- We will attempt to connect better to the theoretical work we have done on @λ, to emphasize that this work is about integrating that mechanism into an existing language (with the compromises this must necessarily entail).
- The reviewers made several more concrete suggestions at different points in the paper that we will integrate.



```
public Color getDefaultColor() {
    return
}
```

```
public Color getDefaultColor() {
    return new Color(
      0,
      0,
      128); // navy

}
```

Figure 9: (a) An example code completion palette associated with the `Color` class. (b) The source code generated by this palette.

7 Active Code Completion

A language's syntax and semantics influences the behavior of 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 [29]. Editors for object-oriented languages provide code completion in the form of a floating menu containing contextually-relevant variables, fields, methods, types and other code snippets. 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.

In all such systems, the code completion interface has remained primarily menu-based. When an item in the menu is selected, code is inserted immediately, without further input from the developer. 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:

```
public Color getDefaultColor() {
   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* of the expression being entered, which in this case is `Color`. 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 9(a) gives an example of a simple palette that may be associated with the `Color` class[8].

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 generates appropriate source code for insertion at the cursor. Figure 9(b) shows the inserted code after the user presses ENTER.

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

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.

7.1 Remaining Tasks and Timeline

This work has been published at ICSE 2012 [**?**], and we do not plan on extending it further in this thesis. The main tasks have to do with relating it to the mechanisms in the previous sections, so that it generalizes beyond Java. We plan to do this when writing the thesis. There are also some pieces of related work that have been published since this paper was accepted that we need to review.

8 Timeline

To summarize, we plan on completing the work in this thesis per the following schedule:

- The work on @λ will be submitted on Mar. 1 to ICFP.
- The work on Ace will be submitted on Mar. 25 to OOPSLA.
- The work on Wyvern will, pending likely final acceptance, be written up in final form by May 12 for ECOOP.
- The work on Graphite has been completed and published at ICSE.

References

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

[2] perlre - Perl regular expressions. `http://perldoc.perl.org/perlre.html`.

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

[4] The python language reference (http://docs.python.org/), 2013.

[5] M. D. Adams. Principled parsing for indentation-sensitive languages: Revisiting landin's offside rule. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 511–522, New York, NY, USA, 2013. ACM.

[6] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 3–12, New York, NY, USA, 2007. ACM.

[7] V. Breazu-Tannen and A. R. Meyer. Polymorphism is conservative over simple types. In *Logical foundations of functional programming*, pages 297–314. Addison-Wesley Longman Publishing Co., Inc., 1990.

[8] R. Brooker, I. MacCallum, D. Morris, and J. Rohl. The compiler compiler. *Annual review in automatic programming*, 3:229–275, 1963.

[9] 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.

[10] 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.

[11] 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*, page 9. ACM, 2010.

[12] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

[13] S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 3–12. ACM, 2013.

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

[15] 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*, pages 197–217. Springer, 2013.

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

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

[18] 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.

[19] K. O. W. Group et al. The opencl specification, version 1.1, 2010. *Document Revision*, 44.

[20] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[21] 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.

[22] 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.

[23] A. Kennedy. Dimension types. In *Programming Languages and Systems—ESOP’94*, pages 348–362. Springer, 1994.

[24] 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.

[25] A. Löh and R. Hinze. Open data types and open functions. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 133–144. ACM, 2006.

[26] W. Lovas and F. Pfenning. A bidirectional refinement type system for lf. In *Electronic Notes in Theoretical Computer Science, 196:113–128, January 2008. [NPP07] [Pfe92] [Pfe93] [Pfe01] Aleksandar Nanevski, Frank Pfenning, and Brigitte*, 2008.

[27] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 499–504. ACM, 2011.

[28] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP ’02, pages 110–122, New York, NY, USA, 2002. ACM.

[29] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] A. J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, Sept. 1982.

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

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

[37] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *Conference on New Directions on Algorithmic Languages*, Munich, Aug. 1975. IFIP WP 2.1.

[38] J. C. Reynolds. An introduction to polymorphic lambda calculus. In *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, 1994.

[39] 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.

[40] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–??, 1999.

[41] 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.

[42] 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.

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

[44] 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.

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

[46] 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.

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

[48] 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.

[49] P. Wadler. The expression problem. *java-genericity Mailing List*, 1998.

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