Type-Oriented Foundations for Safely Extensible Programming Systems

PhD Thesis Proposal

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

February 17, 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) [39]. If a programming system wished to fully support regular expressions, it might simultaneously provide features like these:

1. **Built-in syntax for pattern literals** (e.g. [2]) so that the cognitive load of using regular expressions is low and malformed patterns result in intelligible compile-time errors.
2. A **static and dynamic semantics** that ensures, at compile-time whenever possible, that key invariants related to regular expressions are maintained:
   (a) only appropriate values are spliced into regular expressions, to avoid splicing errors and injection attacks [3]
   (b) out-of-bounds backreferences are not used [36]
   (c) strings known to be in the language of a regular expression are given a type that tracks this information to ensure that string manipulation operations do not inadvertently lead to a string that is not in an assumed language [16].

   When a type 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. Instead, libraries generally provide support for regular expressions by leveraging general-purpose abstraction mechanisms. Unfortunately, it is impossible to fully encode the syntax and the specialized static and dynamic semantics described above in terms of general-purpose notations and abstractions. Library providers thus need to compromise, typically by asking clients to provide regular expressions as strings and deferring parsing, typechecking and compilation to run-time. This introduces performance overhead and can lead to unanticipated run-time errors (as shown in [36]) and security vulnerabilities (due to injection attacks when user inputs are spliced into patterns, for example) [6]. Useful tools for working with regular expressions are rarely integrated into editors, and even more rarely in a way that facilitates their discovery and use directly when the developer is manipulating regular expressions. Tools that must be discovered independently and accessed externally are used infrequently [27] and can be more awkward than necessary [9, 30], leading to lower productivity.

1.2 Language-External Approaches

When the semantics or implementation of a system must be extended to fully realize a new feature, as in the example above, providers typically take a *language-external approach*, either by developing a new or derivative programming system (supported by *language workbenches* [14], *DSL frameworks* [15] or *compiler generators* [7]), or by extending an existing system by some mechanism that is not part of the language itself, such as an extension mechanism for a particular compiler[1] 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

[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.

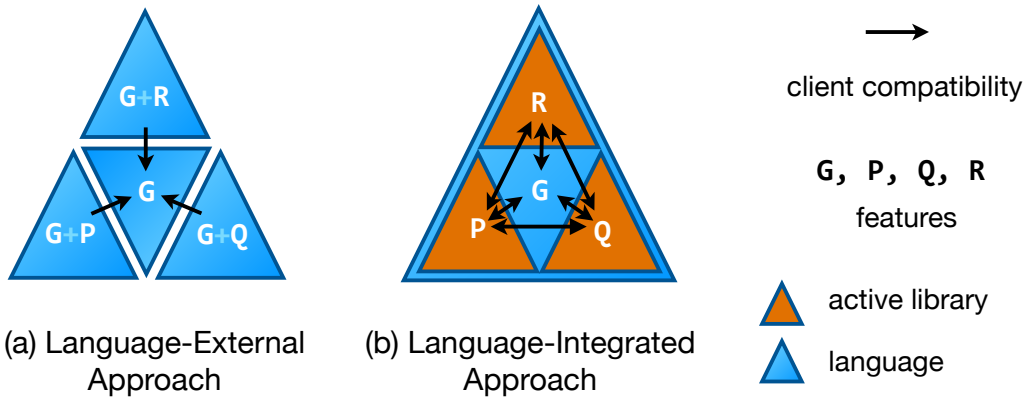

Figure 1: (a) When taking a language-external approach, new features are packaged together into separate languages and tools, causing problems with compatibility (described in the text). (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time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

system containing some general-purpose features (`G`). A different researcher developing a new language-based parallel programming abstraction or implementation strategy (`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 (cf. Sec. 1.3.1, below).

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 to library-based implementations if all else is equal [10], but library-based implementations are more widely adopted because "parallel programming languages" privilege only a few chosen abstractions and associated implementation strategies. This is problematic because different parallel programming abstractions or implementation strategies are seen as more appropriate in different situations [38]. 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 when they have been implemented by language-external means is difficult or impossible.

Client Compatibility Even in cases where for each component of a system there is a system that is completely satisfactory, 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 [43] 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 [29] and F# [31] 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, but calling in the other direction, or between the languages sharing the common target, is not supported.

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. For example, the type system of F# guarantees that null values cannot occur within F# data structures, but maintaining this important invariant still requires run-time checks because the typing rules of F# do not apply when F# code is called from other languages on the Common Language Infrastructure (CLI) like C#. The F# type system also includes support for checking that units of measure are used correctly [37, 22], but this highly-specialized invariant is left completely unchecked at language boundaries. In some cases, desirable features must be omitted entirely due to concerns about interoperability. F#, for example, aimed to retain source compatibility with Ocaml code, but due to the need for bidirectional interoperability with CLI languages, it does not support features like polymorphic variants, modules or functors [23] because they have no apparent analogs in the type system of the CLI.

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* [42] 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 introduced into a programming system. If too much control over these core aspects of the system is given to developers, the system may become 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 at link-time. 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 compilation unit. 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. [42, 41] 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 [40]. 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 [33]).

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 extending a language with new abstractions. These mechanisms are highly expressive because they allow users to programmatically manipulate syntax trees directly, but they suffer from problems of composability and safety. For example, compile-time macros, such as those in MetaML [34], Template Haskell [35] and Scala [8], 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 value 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 truly extend the semantics 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 [42]. 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 [21]. 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 are highly expressive and avoid some of the difficulties of explicitly invoked macros. 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 is SugarJ [11] and other languages generated by Sugar* [12], like SugarHaskell [13]. 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 means that extensions are also 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 comparable to the built-in features found in modern programming systems, but without the kinds of **safety** problems that have been an issue in previous mechanisms, as described above. We also aim to maintain the ability to understand and reason locally about code. This is accomplished by organizing extension logic around types and scoping it to or around expressions of the type it is associated with, rather than applying it globally or within an explicitly delimited scope as in previous mechanisms.

To motivate this 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 family of types should behave. In fact, nearly all the features relate to the type representing regular expression patterns (let us call it `Pattern`[2]). Feature 1 calls for specialized syntax for the introductory form for this type. Features 2a and 2b relate to its static and dynamic semantics. Feature 3 is about its edit-time behavior. The remaining feature, 2c, also relates to the semantics of a single family of types: `StringIn[r]`, which classifies strings known to be in the language of the 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 [32] and PFPL [19], 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 compose without trouble (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).

[2]We should note at the outset that to fully prevent conflicts between libraries, naming conflicts must also be avoided. Suitable namespacing mechanisms (e.g. URI-based schemes, as Java uses) are already widely used in practice and will be assumed implicitly whenever needed.

This ubiquitous type-oriented organization suggests a principled 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 family of types) and scoping it to affect only 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 is guaranteed even if an incorrect extension is used). 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 other 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 conduct empirical studies.

We begin in Sec. 3 by considering **syntax**. The availability of specialized syntax can bring numerous cognitive benefits [17], 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 interference issues, 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, syntax extensions are associated directly with a type and are active only where an expression of that type is expected (shifting part of the burden of parsing into the typechecker). This avoids extension interference problems because the base grammar of the language is never extended directly. We call such an extension a *type-specific language (TSL)* and introduce TSLs in the context of a new language, Wyvern. We begin by showing how interference issues between the base language and the TSL syntax can be avoided by either introducing minor syntactic constraints on the body of a literal or by using a novel layout-delimited literal form. We then develop a formal semantics, basing it on work on bidirectional type systems and elaboration semantics, and introduce a novel mechanism that statically prevents 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 operations. 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 [36]) or to execute a decision procedure for language inclusion when determining whether a

coercion is possible (feature 2c, see [16]). 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, Python. We discuss how we accomplish this, relate Ace to the core calculus and implement a number of powerful primitives from existing languages as libraries, giving examples from a variety of paradigms, including low-level parallel programming, functional programming, object-oriented programming and specialized domains, like regular expression types.

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, we surveyed the expressive power of this approach and developed design criteria. Based on these initial studies, we then developed an active code completion system for Java called Graphite. Using Graphite, we implemented a palette for working with regular expressions and conduct a small study that demonstrates the usefulness of this approach.

Taken together, these mechanisms demonstrate that actively-typed mechanisms can be introduced throughout a programming system to allow users to extend both its compile-time and edit-time semantics from within libraries, without weakening the safety guarantees that the system provides. They also further reinforce the idea that types are a natural organizational unit for defining programming system features and show how a type-oriented approach can make it easier to guarantee that the features will be safely composable in any combination. By implementing our techniques within a variety of different host systems, we demonstrate that actively-typed mechanisms are relevant across traditional paradigms. 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 are quite powerful. By using a general-purpose abstraction mechanism to encode a data structure, one immediately inherits a body of primitive operations, established reasoning principles, well-optimized implementations and tool support. For example, lists can be encoded using inductive datatypes, the general-purpose abstraction mechanism that most functional programming languages emphasize. 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 inheriting these operations and reasoning principles can be quite useful, inheriting 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 specialized 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 [17], 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 others. For example, a language with built-in notation for HTML and SQL, with type-safe interpolation of host language terms within curly braces, might allow:

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

to be shorthand for:

```
1 let webpage : HTML = HTMLElement(Dict.empty(), [BodyElement(Dict.empty(),
2   [H1Element(Dict.empty(), [TextNode(concat("Results for ", keyword))]),
3   ULElement((Dict.add Dict.empty() ("id", "results")), to_list_items(query(db,
4     SelectStmt(["title", "snippet"], "products",
5       [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 sufficient. 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 same reasons as described above: needing 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 values (e.g. `{3}`), or key-value 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 of extensions. For example, SugarJ [11] and other extensible languages generated by Sugar* [12] 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.

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 neither the base language nor TSLs are ever extended directly. It also avoids semantic conflicts – 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 also frees these common notations from being tied to the variant of a data structure built into the standard library, which sometimes does not provide the exact semantics that a programmer needs (for example, Python dictionaries do not preserve key insertion order).

```
1  let imageBase : URL = <images.example.com>
2  let bgImage : URL = <%imageBase%/background.png>
3  new : SearchServer
4    def resultsFor(searchQuery : String, page : Nat) : Unit =
5      serve(~) (* serve : HTML -> Unit *)
6        :html
7          :head
8            :title Search Results
9            :style ~
10             body { background-image: url(%bgImage%) }
11             #search { background-color: %‘#aabbcc‘.darken(20pct)% }
12         :body
13           :h1 Results for {searchQuery}
14           :div[id="search"]
15             Search again: {SearchBox("Go!")}
16           { (* fmt_results : DB * SQLQuery * Nat * Nat -> HTML *)
17             fmt_results(db, ~, 10, page)
18               SELECT * FROM products WHERE {searchQuery} in title
19           }
```

Figure 2: Wyvern Example with Multiple TSLs

```
1  <literal body here, <inner angle brackets> must be balanced>
2  {literal body here, {inner braces} must be balanced}
3  [literal body here, [inner brackets] must be balanced]
4  ‘literal body here, ‘‘inner backticks‘‘ must be doubled‘
5  ’literal body here, ’’inner single quotes’’ must be doubled’
6  "literal body here, ""inner double quotes"" must be doubled"
7  12xyz (* no delimiters necessary for number literals; suffix optional *)
```

Figure 3: Inline Generic Literal Forms

3.1 Wyvern

We develop our work as a variant of a new programming language being developed by our group called Wyvern [28]. 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 or mutable state) and it does not enforce a uniform access principle for objects (fields can be accessed directly). Objects are thus essentially recursive labeled product types with simple methods (functions that require a self-reference). 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* when the variant is not clear.

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.

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:

```
1  let imageBase : URL = new
2    val protocol : URLString = "http"
3    val subdomain : URLString = "images"
4    (* ... *)
```

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`.

In addition to supporting conventional notation for URLs, this TSL supports *typed interpolation* of a 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 delimiters that can be used to go from a TSL back to Wyvern are controlled by the TSL.

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, 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 – as the argument to the function `serve` – 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 type. A value associated with a type is called the type's 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` to a Wyvern AST, which is a value of type `Exp`. This is in turn a case type that encodes the syntax of Wyvern expressions. Fig. 5 shows these types.

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

```
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.

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 type safe and structured, 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.[3] 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

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

```
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
  | Literal of TokenStream
  | Error of ErrorMessage
  metadata = new : HasTSL
    val parser : Parser = (* quasiquotes *)
```

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

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 [24]) with an elaboration semantics (in a style similar to Harper and Stone [20]). 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 bifurcating 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.

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 making it impossible for variables to be introduced into interpolated Wyvern expressions directly (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 values 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 in a quasiquote must be inside **`valAST`**. We believe we can insert **`valAST`** automatically from within the definition of the TSL for `Exp` using a very simple extension to the interpolation mechanism that allows interpolation of single variables, rather than whole expressions, and plan to write up how this works.

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. We will leave the syntax fixed, inverting the setup in the previous section. We do not believe the two approaches to be incompatible with one another, but it is helpful to examine the issues separately. We leave combining the approaches introduced in this thesis into a single language as future work.

4.1 Foundations

Typed programming languages are typically organized around a collection of indexed type and operator constructors. The simply-typed lambda calculus (STLC), for example, provides a single type constructor, conventionally written $\rightarrow$, indexed by a pair of types, and two operator constructors, λ, indexed by a type, and ***ap***, which can be thought of as being indexed trivially. It's static semantics can be written like this, emphasizing this uniform manner of thinking about type structure:

$$\textsc{var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{arrow-i}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda[\tau](x.e) : \textsc{Arrow}[(\tau, \tau')]} \qquad \textsc{arrow-e}\ \frac{\Gamma \vdash e_1 : \textsc{Arrow}[(\tau, \tau')] \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{ap}[()](e_1; e_2) : \tau'}$$

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

We can extend this language with universally quantified and recursive types by adding the type constructors $\forall$ and μ, each indexed by a type binder (a type-level term that may refer to a free type variable), and their associated operator constructors (see [19]).

It is tempting to stop here: the language is now universal (i.e. Turing complete) and the dynamic semantics of operators associated with other seemingly useful data types, including sums and products, 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 is plainly impractical as a programming language. Reynolds, striking a cautionary tone reminiscent of Perlis in his description of the "Turing tarpit" [**?**], put it succinctly [**?**]:

right terminology?

> 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.

A simple language, equipped perhaps with some purely syntactic sugar of the sort that a Church-style encoding defines, very often suffices when only the dynamic semantics of abstractions are of importance. But continuing to add type and operator constructors (collectively, *constructs*) to a language can make statically reasoning about programs simpler and more precise and support a more natural programming style.

Consistent with this view, typed programming languages generally expose more constructs externally than their compilers work with internally. A functional language like ML, for example, might expose n-ary product types (tuples), labeled product types (records), recursive labeled sum types (datatypes) and forms of type abstraction (polymorphism and abstract data types). A compiler for the language might then, after typechecking these constructs according to their own rules, translate these to an intermediate language (IL) with only simple products and sums for further compilation to machine code. Correctly implementing the dynamic semantics of the language is 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 is 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 "sweet spots" in the design space, and many abstractions can be encoded using these constructs in a completely satisfactory manner. Claims about being suitable for all reasonable purposes are not backed by any "universality" theorems stronger than those for the simple calculus described above, however. Situations continue to arise in both research and practice where new constructs are desirable:

1. General-purpose abstractions are considerably varied. There are many variations on record types, for example: structurally-typed records, or records with forms of dynamic dispatch. We will show an example of prototypic dispatch, for example, and argue that we cannot safely define it by a simple syntactic desugaring to a language like ML. Similarly, sum types are awkward to work with in object-oriented languages, and also admit variants, as we will discuss in the next section.

2. Specialized type systems that enforce stronger invariants than general-purpose abstraction mechanisms are capable of straightforwardly 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 [**?**, **?**], databases [**?**], aliased references [**?**], effects [**?**], network protocols [**?**], regular expressions [**?**], units of measure [**?**] and many others.

3. Interoperability layers that allow programmers to safely and naturally interact with libraries written in a different language require enforcing the type system of that language within the calling language. Using library-based interoperability layers today can lead to safety issues when this is not possible.

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 mechanisms 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. A simple implementation in Standard ML could be based around these datatypes, for example:

```
1  datatype Ty = Arrow of Type * Type
2  datatype Op = Lam of Type | Ap
```

The compiler front-end, which consists of a typechecker and translator to a suitable intermediate language, as described above, will proceed by exhaustive case analysis over the constructors of `Exp`.

In a class-based object-oriented implementation of Godel's T, 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 collection of known subclasses of `Op`, then we encounter a similar issue: there is no way to modularly add new primitive type and operator constructors to `Ty` and `Op`, respectively, and provide implementations of their associated typechecking and translation logic. Everything is defined in one place.

A number of mechanisms have 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* [**?**]. For example, if we wish to extend our implementation of the STLC with product types and we have written our compiler in a language supporting open datatypes, it might be possible to add new constructors. For example, we might add a new type constructor, `Tuple`, indexed by a list of types, and two new operator constructors: `NewTuple`, indexed trivially, and `Prj`, indexed by a natural number:

cite

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

The logic for functionality like typechecking and translation can then be implemented for only these new cases. For example, the `optype` function that synthesizes a principle type for an operator invocation given a context and a list of arguments could be extended to support the new case `PrL` like so:

```
optype (ctx, Prj(n), [e]) = case typeof (ctx, e) 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>")
```

In a class-based object-oriented language like Java, we might dispense with the visitor pattern and instead invert control to methods of `Op`.

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

If we allowed users to define new modules containing definitions like these and link them into our compiler, we will have succeeded in creating an externally-extensible compiler, albeit one where safety and conservativity is not guaranteed (we will return to this point shortly). We have not, however, created an extensible programming language because other compilers for the same language will not necessarily support the same extensions. 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). Nevertheless, this gives us a conceptual seed for the approach that we wish to take.

We argue that a more appropriate and useful place for extensions like this is directly within libraries, alongside abstractions that can be implemented in terms of existing primitive abstractions. To enable this, the language must allow for the introduction of new type constructors, like `Prod`, operator constructors, like `Pair`, `PrL` and `PrR`, and their associated type synthesis and translation logic. Because this mechanism is integrated into the language specification, all compilers must support it.

The design described above suggests we may now need to add another layer to our language, above the type-level language (conventionally, τ) and expression language (conventionally, e), where extensions are implemented. In fact, we will show that **a natural place for type system extensions is within the type-level language**. The intuition is that extensions to a statically typed language's semantics will need to manipulate types as values at compile-time. Many languages already allow users to write type-level functions for various reasons, effectively supporting this notion of types as values at compile-time. The type-level language is often constrained by its own type system, where the types of type-level terms are called *kinds* for clarity, that prevents type-level functions from causing problems during compilation. The kind system is often more constraining than the type system is (e.g. it might permit only terminating functions, to ensure that typechecking is decidable). This is precisely the structure that a distinct extension layer would have, and so we will show that is quite natural to unify the two.

We will begin by taking a first-principles type-theoretic approach in Sec. 5, 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 type-directed desugaring mechanism for common syntactic forms that targets this core calculus. We then continue on in Sec. 6 by designing a full programming language called Ace. We will show how a range of statically-typed general-purpose abstraction mechanisms as well as more specialized abstractions can be expressed as safely-composable extensions within Ace. Notably, Ace is itself implemented as a library for Python. We discuss how to bootstrap a highly-expressive extensible static type system atop the quotation and object-oriented features available in Python, a "dynamically typed" language.

But, as mentioned in Section 1.3, some significant challenges must be addressed before such a mechanism can be relied upon. The desire for expressiveness must be balanced against concerns about maintaining various safety properties in the presence of arbitrary combinations of user-defined extensions to the language's core semantics. The mechanism must ensure that desirable *metatheoretic properties* (e.g. type safety, decidability) of the language are maintained by extensions. Because multiple independently developed extensions might be used within one program, the mechanism must further guarantee that extensions are *conservative*. These are the issues we seek to address in the next two sections.

5 @λ

5.0.1 @λ: Conservatively Extending a Type System From Within

We will begin by developing an "actively-typed" version of the simply-typed lambda calculus with type-level computation called @λ. In the semantics for this calculus, we will integrate typed compilation techniques and a form of type abstraction to allow us to prove key type safety and non-interference theorems. The abstract syntax of @λ is in Fig. 7.

The semantics of the language are in the style of the Harper-Stone elaboration semantics for Standard ML: all terms in an external language, e, are simultaneously assigned a type and an elaboration to a term, ι, in a fixed typed internal language [**?**, **?**]. The mechanism checks that any generated elaborations are *type preserving* – that all external terms of a

$$
\begin{array}{rrcl}
\textbf{programs} & \rho & ::= & \mathsf{tycon}\ \textsc{Tycon}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\mathsf{schema}\ \tau_{\text{rep}}; \theta\};\ \rho \mid \mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho \mid e \\
\text{operators} & \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}\langle\tau_{\text{idx}}\rangle(e_1; \ldots; e_n) \\
\\
\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) \\
\text{type-level data} & & | & \bar{\mathsf{z}} \mid \tau_1 \oplus \tau_2 \mid \textit{“str”} \mid () \mid (\tau_1, \tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau) \\
\text{types} & & | & \textsc{Tycon}\langle\tau\rangle \mid \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{structural equality} & & | & \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \\
\text{elaborations} & & | & [\![\tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}}]\!] \mid \mathsf{error} \mid \mathsf{case}\ \tau\ \mathsf{of}\ [\![\mathbf{x}\ \mathsf{as}\ \mathbf{t}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{reified IL} & & | & \triangledown(\iota) \mid \blacktriangledown(\sigma) \\
\\
\textbf{kinds} & \kappa & ::= & \kappa_1 \to \kappa_2 \mid \mathsf{list}[\kappa] \mid \mathbb{Z} \mid \mathsf{Str} \mid 1 \mid \kappa_1 \times \kappa_2 \mid \star \mid \mathsf{Elab} \mid \mathsf{ITm} \mid \mathsf{ITy} \\
\\
\textbf{internal terms} & \iota & ::= & x \mid \lambda x{:}\sigma.\iota \mid \iota_1\ \iota_2 \mid \mathsf{fix}\ f{:}\sigma\ \mathsf{is}\ \iota \mid (\iota_1, \iota_2) \mid \mathsf{fst}(\iota) \mid \mathsf{snd}(\iota) \\
& & | & \bar{\mathsf{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 \\
& & | & \mathsf{abs}([\![\tau_1\ \mathsf{as}\ \tau_2]\!]) \mid \triangle(\tau) \\
\textbf{internal types} & \sigma & ::= & \sigma_1 \to \sigma_2 \mid \mathbb{Z} \mid \sigma_1 \times \sigma_2 \mid \mathsf{rep}(\tau) \mid \blacktriangle(\tau)
\end{array}
$$

Figure 7: Abstract syntax of @λ. Here, x ranges over external and internal language variables, $\mathbf{t}$ ranges over type-level variables, TYCON ranges over type constructor names and ***op*** ranges over operator constructor names. The form *“str”* represents string literals, $\bar{\mathsf{z}}$ represents integer literals and $\oplus$ represents standard total binary operations over integers.

particular type, τ, elaborate to internal terms with a consistent internal type, σ – following the approach taken in the TIL compiler for Standard ML by Morrisett et al. [**?**]. This allows the semantics to be reasoned about compositionally and, when combined with type safety of the internal language, gives us type safety of the extensible external language. This design ensures that global properties that the internal language maintains are guaranteed to hold no matter which extensions are introduced. In other words, extension providers cannot implement type systems weaker than the type system of the internal language. Providers are, however, able to implement type systems that maintain invariants stronger than those the internal type system can maintain. The internal language of @λ is PCF with simple products and one base type, integers. Because of the availability of the fixpoint operator, the internal language is universal (i.e. Turing-complete).

It is reasonable, if we wish to actually implement this language, to permit only the addition of deterministic, decidable rules to the semantics. Extension logic is thus implemented in a functional style (essentially, lifting fragments of the typechecker into the type-level language), rather than extracted from inductive specifications, like those in the Sec. 5[4]. This has the added benefit of giving providers finer-grained control over error reporting. When extracting an implementation from a typical inductive specification, which only specifies “success conditions”, it is difficult to provide different error messages at different failure points within a single rule.

To give a first example, the code in Fig. 8 (written using concrete syntax) shows how to introduce a new type constructor, NAT, indexed trivially (i.e. by a type-level value of kind 1, also known as unit). Types themselves are type-level values of kind $\star$ and are introduced by naming a type constructor and providing an index of the appropriate kind: $\textsc{Nat}\langle()\rangle$. To ensure that type equality is decidable, type constructors can only be indexed by values of a kind for which equality is “trivially” decidable (i.e. where semantic equality coincides with syntactic equality). This means that types themselves, as well as type-level data structures containing values of equality kind, can be used, while type-level functions cannot.

[4]Though we will not explore this further in this thesis, a system for extracting such a function from a declarative specification could potentially be implemented using the techniques in Sec. 3 – a *kind-specific language*!

```
1  tycon Nat of 1 with
2     schema λidx:1.▼(ℤ)
3     opcon Z of 1 (λidx:1.λargs:list[Elab].is_empty args ⟦▽(0) as Nat[()]⟧)
4     opcon S of 1 (λidx:1.λargs:list[Elab].pop_final args λx:ITm.λty:⋆.
5       checktype ty Nat[()] ⟦▽(△(x)+1) as Nat[()]⟧)
6     opcon Rec of 1 (λidx:1.λargs:list[Elab].
7       pop args λx1:ITm.λty1:⋆.λargs':list[Elab].
8       pop args' λx2:ITm.λty2:⋆.λargs'':list[Elab].
9       pop_final args'' λx3:ITm.λty3:⋆.
10      check_type t1 Nat[()] (
11      check_type t3 Arrow[(Nat[()], Arrow[(t2, t2)])]
12        ⟦▽((fix f:ℤ → rep(t2) is λx:ℤ.
13          if x = 0 then △(x2) else △(x3) (x - 1) (f (x - 1))) △(x1)) as t2⟧))
14 end
```

Figure 8: An implementation of primitive natural numbers as internal integers in @λ.

To ensure that compilation is type-preserving, as described above, each type must have an internal type, called its *representation*, associated with it. The representation of a type might depend on its index, so each type constructor must declare a type-level function that returns an internal type given an index, called its *representation schema*. An internal type, σ, is reflected as a type-level term of kind ITy via the introductory form $\blacktriangledown(\sigma)$. Because NAT is indexed trivially, it simply returns the reflected form of the internal type $\mathbb{Z}$.

In the external language, e, all operators (other than λ, for reasons we will discuss) are invoked uniformly. The natural number two, for example, is written:

$$\text{NAT}.s\langle()\rangle(\text{NAT}.s\langle()\rangle(\text{NAT}.z\langle()\rangle()))$$

Operator constructors, like type constructors, are indexed by type-level values. In our example, all the operators are indexed trivially, but we will see examples later where non-trivial operator indices are useful. We will discuss why operator constructors are associated directly with type constructors, rather than existing as standalone entities, when we discuss the scoping mechanism for the representation schema, which is essential for guaranteeing non-interference. This coupling also forms the foundation for a higher-level elaboration mechanism that allows us to use more conventional and concise (though still fixed) syntactic idioms, which we intend to explore in this thesis, but have not yet formalized. For now, we simply note that intuitively, most operators are either introduction or elimination forms for a particular type, so we are simply encoding a common idiom.

The declarations of the three operator constructors related to natural numbers are given on lines 3-13 of Fig. 8. Operator constructors are declared by specifying the kind of type-level value they are indexed by and giving an *operator definition*, a type-level function of kind $\kappa \to \mathsf{list}[\mathsf{Elab}] \to \mathsf{Elab}$. The kind Elab classifies *elaborations*. There are two introductory forms for this kind: *valid elaborations*, $[\![\tau_{\text{iterm}} \text{ as } \tau_{\text{type}}]\!]$, consist of a reflected internal language term paired with a type assignment, while *error elaborations*, error, represent a type error (in a full implementation, the error elaboration would contain an error message, as discussed above, but for simplicity in our core calculus, we simply treat all type errors equivalently). Internal terms, ι, like internal types, are reflected as type-level terms of kind ITm via the introductory form $\triangledown(\iota)$. The form $\triangle(\tau)$ interpolates a type-level value of kind ITm into an internal term. Note that there are no elimination forms for the kinds ITy and ITm. Composition occurs entirely via interpolation – syntax trees are never examined directly by extensions in @λ.

Operator definitions are called by the compiler to generate an elaboration for operator invocations. In our example, the definition of NAT.z simply checks that there were no arguments (using a simple helper function for working with lists of elaborations, `is_empty`, not shown) and if so, provides an elaboration where the internal language integer `0` is associated with type $\text{NAT}\langle()\rangle$. If the list is not empty, `is_empty` simply returns error.

The other two operator constructors are more verbose, but follow a similar pattern – elaborations are popped off and broken down into their constituent internal terms and types by helper functions, then a functional program that implements the intended semantics is written to produce an elaboration as a whole. Note that the `Rec` operator does not itself bind new variables; it must use the built-in `Arrow` type constructor. There is no support in @λ for introducing new forms of binding; λ is the only available binding operator. Note that application is invoked like other operators: ARROW.$\boldsymbol{ap}\langle()\rangle(e_1; e_2)$.

This definition of natural numbers can be shown to be an adequate encoding of the semantics given in Sec. 5. 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:

```
1 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 $\mathsf{rep}(\tau_{\text{type}})$, does not reduce further to the actual representation type (e.g. to $\mathbb{Z}$) when not in an operator definition associated with τ_{type}. The only internal form that can be checked against $\mathsf{rep}(\tau_{\text{type}})$ is the special form $\mathsf{abs}(\llbracket \tau_{\text{iterm}} \text{ as } \tau_{\text{type}} \rrbracket)$, an abstracted internal term corresponding to the internal term reflected by τ_{iterm}. Rather than exposing it explicitly, it is wrapped so that it can't be checked against any type other than $\mathsf{rep}(\tau_{\text{type}})$. 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 @λ is available as a paper draft[5].

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.

5.0.2 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:

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

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.

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

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!"
```

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.

6.0.3 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.

6.0.4 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 [18]. 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.

6.0.5 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.

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

```
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]]
```

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.

6.0.6 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 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).

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

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

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

```
float negate__0_(float x) {
  return x * -1;
}

kernel void map_neg_f32(global float* input,
    global float* output) {
  size_t thread_idx = get_global_id(0);
  output[thread_idx] = negate__0_(input[thread_idx]);
  if (thread_idx == 0) {
    printf("Hello, run-time world!");
  }
}

int2 negate__1_(int2 x) {
  return (int2)(x.s0 * -1, x.s1);
}

kernel void map_neg_ci32(global int2* input,
    global int2* output) {
  size_t thread_idx = get_global_id(0);
  output[thread_idx] = negate__1_(input[thread_idx]);
  if (thread_idx == 0) {
    printf("Hello, run-time world!");
  }
}
```

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.

6.0.7 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).

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 [18]). 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. 6.0.5, 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).

6.0.8 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).

6.0.9 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.
- 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.

6.0.10 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 [25]. 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,

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

```
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)
```

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.

6.0.11 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.

6.0.12 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.

6.0.13 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`).

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.

6.0.14 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).

6.0.15 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).

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

```
import ace, ace.python as py, examples.fp as fp

Tree = lambda name, a: fp.Datatype(name,
  lambda tree: {
    'Empty': fp.unit,
    'Leaf': a,
    'Node': fp.tuple(tree, tree)
  })

DT = Tree('DT', py.dyn)

@ace.fn(py.Base(trailing_return=True))[[DT]]
def depth_gt_2(x):
  x.case({
    DT.Node(DT.Node(_), _): True,
    DT.Node(_, DT.Node(_)): True,
    _: False
  })

@ace.fn(py.Base(main=True))[[]]
def __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)
```

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

```
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)
```

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, 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 [19]. 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.

6.0.16 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.

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 [26]. 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

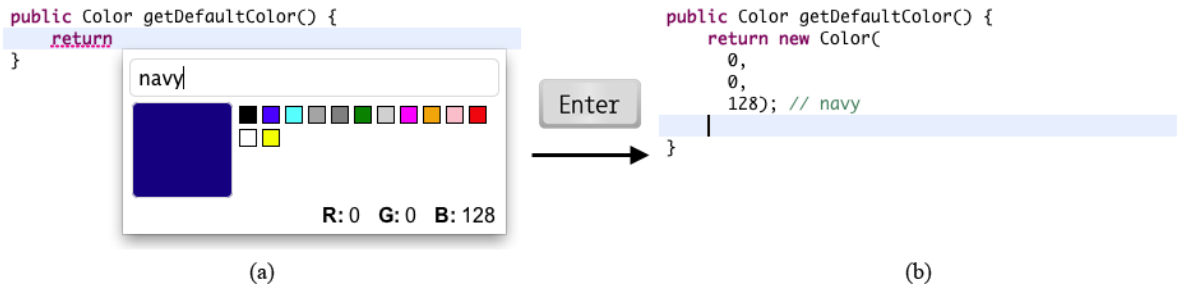

Figure 9: (a) An example code completion palette associated with the `Color` class. (b) The source code generated by this palette.

a way to associate metadata with classes. The techniques in this section apply equally well to type-aware editors for any language with a similar mechanism.

For example, consider the following Java code fragment:

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

If the developer invokes the code completion command at the indicated cursor position (_), the editor looks for a *palette definition* associated with the *type* 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[6].

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.

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

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

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

[35] 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.

[36] 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.

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

[38] 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.

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

[40] 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.

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

[42] 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.

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