Type-Oriented Foundations for Extensible Programming Systems

PhD Thesis Proposal

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

January 16, 2014

Abstract

We propose a thesis defending the following statement:

Active types allow providers to extend a programming system with new compile-time and edit-time features 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 has 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. Due to the increasing diversity and complexity of modern problem domains and hardware platforms, it has become clear that no small collection of general-purpose constructs backed by a simple, all-purpose implementation can fully satisfy these criteria in all cases. Instead, researchers and domain experts (collectively, *providers*) continue to develop specialized notations, static and dynamic semantics, implementation strategies, optimizations, run-time systems and tools (collectively, *features*) designed to address these modern challenges in different circumstances.

Ideally, a provider would develop and distribute such new features orthogonally, as libraries, so that client developers could granularly choose those that best satisfy their needs. Unfortunately this is often infeasible because libraries are vehicles for specifying a program in terms of a monolithic collection of existing features. From the perspective of a library, the language's syntax, type system and dynamic semantics are fixed in advance, the compiler and run-time system are "black box" implementations of this fixed language, and the other tools, like code editors and debuggers, operate according to domain-agnostic protocols also based only on a fixed language specification. As a result, providers of new system features must often take *language-external approaches*, sometimes designing a new system altogether. We will argue that such approaches are problematic, and that taking them has led to an unnecessary gap between research and practice. In place of these approaches, we will advocate for *language-integrated extensibility mechanisms*, and show how, by organizing new features around types and enforcing certain constraints, these mechanisms can be made both safe and highly expressive. We call a type that introduces new features an *active type* and programming systems organized around active types *actively-typed programming systems*.

1.1 Motivating Example: Regular Expressions

To make the issue concrete, let us begin with a simple example. *Regular expressions* are a widely-used mechanism for finding patterns in semi-structured strings (e.g. DNA sequences) [44]. A programming system that included full compile-time and edit-time support for regular expressions might simultaneously provide features like these:

1. **Built-in syntax for pattern literals** (e.g. [2]) so that malformed patterns result in intelligible *compile-time* parsing errors. In languages lacking such literals, run-time errors relating to pattern syntax can occur, even in well-tested, deployed code [40].
2. **Typechecking logic** that ensures that key constraints related to regular expressions are not violated at compile-time:
   (a) only appropriate values are spliced into regular expressions, to avoid splicing errors and injection attacks [4]
   (b) out-of-bounds backreferences are not used [40]
   (c) strings known to be in the language of a regular expression are given a type that tracks this information and ensures that string manipulation operations do not inadvertently lead to a string that is not in the assumed language [18].

   When a type error is found, an intelligible error message is provided.
3. **Translation logic** that partially or fully compiles known regular expressions into the efficient internal representation that will be used by the regular expression matching engine (e.g., a finite automata [44]) ahead of time. In most languages, this compilation step occurs at run-time, even if the pattern is fully known at compile-time, thereby introducing performance overhead into programs. If the developer is not careful to cache compiled representations, regular expressions used repeatedly in a program might be needlessly re-compiled on each use.
4. **Editor services** for quickly testing regular expression patterns against test strings, referring to documentation or searching databases of common patterns (e.g. [1]).

No system today has built-in support for all of the features enumerated above. Instead, libraries generally provide support for regular expressions by leveraging general-purpose constructs. 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 like objects or inductive datatypes. Library providers have thus needed to compromise, typically by asking clients to provide regular expressions as strings, thereby deferring parsing, typechecking and translation to run-time, which introduces performance overhead and can lead to unanticipated run-time errors (as shown in [40]) and security vulnerabilities (due to injection attacks when user inputs are spliced into patterns, for example). Similarly, 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 less frequently and can be more awkward than necessary [11, 32], leading to lower productivity [32].

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* [16], *DSL frameworks* [17] or *compiler generators*), or by extending an existing system by some mechanism that is not part of the language itself, such as an extension

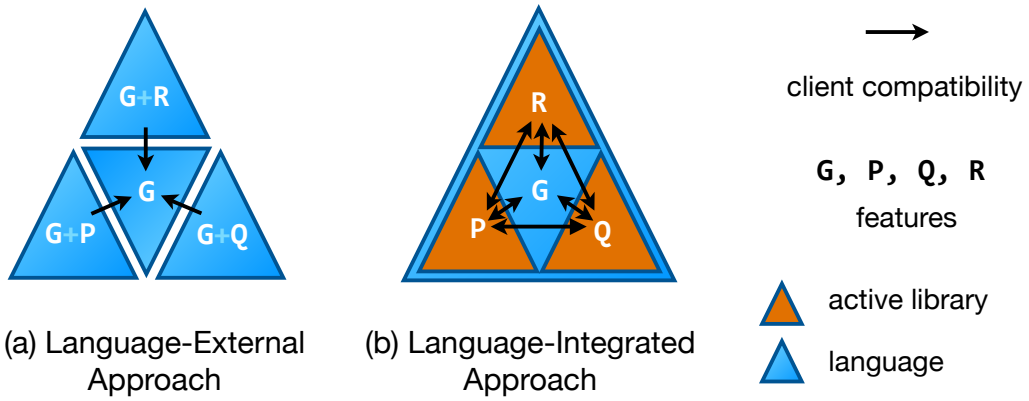

Figure 1: (a) When taking a language-external approach, new features are packaged together into separate languages and tools, causing problems with compatibility (described in the text). (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time 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.

mechanism for a particular compiler[1] or other tool. For example, a researcher interested in providing regexp-related features (let us refer to these collectively as `R`) might design a new language with built-in support for regular expressions, perhaps basing it on an existing language (containing some general-purpose constructs, `G`). A different researcher developing a new language-based parallel programming abstraction or implementation strategy (collectively, `P`) might take the same strategy. A third researcher, developing an alternative parallel programming abstraction (collectively, `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. Features cannot be adopted individually, but instead are only available coupled to a collection of other unrelated features (e.g. the incidental features of the newly-designed language and its associated tools). 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. Evidence supports the prominence of this problem: developers prefer language-integrated parallel programming abstractions to library-based implementations if all else is equal [12], but library-based implementations are more widely adopted because "parallel programming languages" privilege only a few chosen abstractions and implementation strategies. This is problematic because different parallel abstractions or implementation strategies are more appropriate in different situations [43] and 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. One must either use the system containing features `G+R` or `G+P`. There is no system containing both.

[1]Compilers that modify, or allow modification of, the semantics of their base language, rather than simply permitting meaning-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 sin, meaning that some programs that seem to be written in C, Haskell or Standard ML are actually written in tool-specific derivatives of those languages. Language-internal mechanisms do not lead to such fragmentation.

Different application concerns can sometimes be separated into different components, each written using a suitable language and tooling. This has been called the *language-oriented approach* to software development [48]. Unfortunately, this is insufficient: the interface between components written in different languages remains an issue. The specialized constructs particular to one language (e.g. futures in a parallel programming language) cannot always be safely and naturally expressed in terms of those available in another (e.g. a general-purpose language), so building a program out of components written in a variety of different languages is difficult whenever these are exposed at interface boundaries. Tool support is also lost when calling into a different language. We call this fundamental issue the *client compatibility problem*: clients of a certain collection of features cannot always interface with clients of a different collection of features in a safe, performant and natural manner.

One strategy often taken by proponents of the language-oriented approach to partially address the compatibility problem is to target an established intermediate language, such as the Java Virtual Machine (JVM) bytecode, and support a superset of its constructs. Scala [30] and F# [33] 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 true client compatibility in one direction. While calling into the common language becomes straightforward, calls in the other direction, or between the languages sharing the common target, are still restricted by the semantics of the common language.

If new languages only include constructs that can already be expressed safely and reasonably naturally in the common language, then this approach can work well. But many of the most innovative constructs found in modern languages (often, the features 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 [42, 24], but this 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, is based on Ocaml, but due to the need for bidirectional interoperability with CLI languages, it cannot support features like polymorphic variants, modules or functors [26] because they have no straightforward analogs in the object-oriented intermediate language of the CLI.

1.3 Language-Integrated Approaches

We argue that, due to these problems, 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 developers the ability to define in libraries new edit-time and compile-time features of the system that have previously been defined centrally[2] so that such language-external approaches are less frequently necessary. More specifically, we will show how control over aspects of **syntax**, **typechecking**, **translation** and **editor services** can be delegated to user-defined logic distributed in libraries, as illustrated in Figure 1b. Such libraries are called *active libraries* 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 [47]. Features implemented within active libraries can be imported as needed by the clients of libraries that rely on them, unlike features implemented by language-external means, seemingly avoiding the client compatibility problem.

[2]One might compare today's monolithic programming systems to centrally-planned economies, whereas extensible systems more closely resemble modern market economies.

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, because if too much control over such 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 (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.

1.3.1 Background

The term *active libraries* was first introduced by Veldhuizen et al. [47, 46] 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, 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 [45]. Although this and several other interesting optimizations have been shown 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 [37]).

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 and reporting 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 [38], Template Haskell [39] and Scala [10], take full control over all 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 compatibility problems described in Section 1). 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 block scoping of macros with 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 [47]. Similarly, the derivative of Haskell implemented by the Glasgow Haskell Compiler (GHC) allows libraries to define custom compile-time term rewriting logic if an appropriate flag is passed in [23]. 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 block-scoped 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 importing a library can change the semantics of the program in a non-local manner.

Another example of an active library approach to extensibility is SugarJ [13] and other generated by Sugar* [14], like SugarHaskell [15]. 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. These 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 standard notations but back them with differing implementations. And again, it can be difficult for users to predict what type of data 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.

To motivate the approach we will take in achieving these goals, let us return to our example of regular expressions. Observe that every feature described in Section 1.1 relates specifically to how terms classified by a single user-defined type[3] should behave. In fact, nearly all the features relate to the type representing regular expression patterns (let us call it `Pattern`[4]). Feature 1 calls for specialized syntax for the introductory form of this type. Features 2a and 2b relate to its static semantics. Feature 3 relate to its translation semantics. Feature 4 is about its edit-time behavior. The only remaining feature, 2c, relates to the static semantics of a different type family, `StringIn[r]` which classifies strings known to be in the language of the regular expression, `r`. It is exclusively when editing or compiling expressions of the associated type that the logic in Section 1.1 needs to be considered.

Indeed, this is not a property unique to our chosen example, but a very commonly-seen pattern in programming language design. Types are already known to be natural entities around which the semantics of a programming language or logic can be organized. In two major textbooks about programming languages, TAPL [34] and PFPL [21], 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 operators from different chapters into languages is a metatheoretic (in other words, language-external) notion. For example, in PFPL, the notation $\mathcal{L}\{\rightarrow \texttt{nat dyn}\}$ represents a language composed of the arrow ($\rightarrow$), `nat` and `dyn` types and their associated operators. These are defined in separate chapters, and it is 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 straightforwardly combined to form a language).

[3] We will generalize this to cover several different types within an indexed type family in later sections.

[4] We should note at the outset that to fully prevent conflicts between libraries, naming conflicts must also be avoided. Suitable namespacing mechanisms (e.g. URL-based schemes, as Java uses) are already well-developed in practice and will be assumed.

This 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 to a single type (or type family) as it is defined and scoping it only around expressions classified by that type (or by a type in that type family). We call types with such logic associated with them *active types* and systems that support them *actively-typed programming systems*. By constraining the extension logic itself by various means we will show that the system as a whole can maintain many important safety and non-interference properties that have not previously been achieved.

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 factors related to extension correctness. 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. We will also conduct empirical studies and case studies that demonstrate the scope and applicability of our approaches in practice.

We begin in Sec. 3 by considering **syntax**. The availability of specialized syntax can bring numerous cognitive benefits [19], and discourage the use of problematic techniques like using strings to represent structured data [8]. 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 syntax for the introductory forms (i.e. *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 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 specifying a layout-sensitive base syntax for Wyvern (like Python's and Haskell's) containing a form of *syntactic forward reference* that makes entering multi-line expressions simpler, and when used with TSLs, sidesteps interference issues that can arise between the TSL's syntax and the enclosing language's syntax. In addition to this novel syntactic form, we also show how typical literal forms (e.g. string literals, array literals, etc.) can be repurposed to operate more generically as TSLs. We then develop a formal semantics, basing it on work on bidirectional type systems, and introduce a novel mechanism that statically prevents unsafe variable capture. Finally, we conduct a corpus analysis to examine this technique's applicability, finding that a substantial fraction of string literals in existing code could be replaced by TSL literals. In addition to being more natural, this shifts parsing errors to compile-time, improves performance and can help prevent security vulnerabilities.

Wyvern has an extensible syntax but a fixed static and dynamic semantics. While the general-purpose abstraction mechanisms we have included in Wyvern – structurally-typed objects and inductive datatypes – are powerful, and implementation techniques for these have been well-studied, there remain situations where providers may wish to extend the **semantics** of a language directly by introducing new primitive types and operators and providing **specialized implementations** of them directly. Examples of type system extensions and new implementation strategies that require changing how the compiler elaborates expressions into terms in an intermediate language 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 [40]) or to execute a decision procedure for language inclusion when determining subtyping relationships (feature 2c, see [18]). New logic must also be added to the elaborator to implement feature 3. Having control over translation is particularly important in areas like parallel programming, where performance is of particular concern. To support these sorts of use cases, we next consider mechanisms for safe semantic extensions, while allowing the syntax to remain fixed. We begin in Sec. 4 with a type theoretic treatment, by specifying an "actively-typed" lambda calculus called @λ. We prove key safety and non-interference theorems and examine the connections between active types and several prior notions, including type-level computation, type abstraction and typed compilation. We then go on in Sec. 5.1 to demonstrate the expressiveness of this mechanism by designing and implementing a full-scale language, Ace. We show how it relates to, and extends, the core calculus and implement a number of powerful primitives from existing languages as libraries. We give examples from a variety of paradigms, including low-level parallel programming, functional programming, object-oriented programming and domain-specific languages to demonstrate the expressive power of this approach.

Finally, in Sec. 6, we will show how **editor services** that help developers correctly and efficiently enter complex expressions can be introduced from within active libraries, by a technique we call *active code completion*. Providers associate domain-specific 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 survey the expressive power of this approach and describe the design and safety constraints governing the mechanism. Based on these initial studies, we then develop an active code completion system for Java called Graphite. Using Graphite, we implement a palette for working with regular expressions in order to conduct a small study that provides evidence for 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 metatheoretic guarantees that the system provides. They also further demonstrate that types are a natural organizational unit for defining programming system features, because a great variety of features can be expressed in an actively-typed manner, and doing so can make it easier to guarantee that the features will be safely composable in any combination. In this thesis, each mechanism is implemented within a different programming system, showing that actively-typed mechanisms are relevant across traditional paradigms, including functional languages (@λ), class-based OO languages (Graphite), structurally-typed languages (Wyvern) and scripting languages (Ace). In the future, we anticipate that mechanisms like those in this thesis will be brought together to form a single highly-extensible system with a minimal, well-specified core, where nearly every feature is distributed within a library.

3 Type-Specific Languages

By using a general-purpose abstraction mechanism to encode a data structure, one immediately benefits from a body of established reasoning principles and primitive operations. For example, inductive datatypes can be used to express data structures like lists: intuitively, a list can either be empty, or be broken down into a value (its *head*) and another list (its *tail*). In an ML-like language, this concept is conventionally written:

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

By encoding lists in this way, we can reason about them by structural induction, construct them by choosing the appropriate case and inspect them by pattern matching. The programmer only needs to provide an encoding of the structure of lists; the semantics are filled in by the general-purpose abstraction mechanism.

While inheriting semantics can be quite useful, inheriting 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 provide specialized notation for constructing 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. More specifically, it is more *terse*, *visible* and *maps more closely* to the intuitive notion of a list, to use terminology from the literature on the cognitive dimensions of notations [19].

Although number, string and list 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, and markup languages, like HTML. For example, a language with built-in syntax for HTML and SQL, with type-safe interpolation of host language terms via 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 mean:

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

When a specialized notation is not available, and 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 are frequently tempted to write the example 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 and escaping when the syntax of the language clashes with the syntax of string literals (line 2). But code like this also causes a number of more serious 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, or in a loop). 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 provided 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 [4]. 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 is to use structured representations throughout the codebase, aided by specialized notation like that above [8].

To emphasize that this is a common problem, let us return to considering regular expression literals. It is quite tedious to write out a regular expression in a structured manner. A simple regular expression like `(\d\d):(\d\d)\w*((am)|(pm))` representing times might be written:

```
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 the same reasons as described above.

As we will examine further in our corpus analysis, situations like this, where specialized notation is necessary to maintain strong correctness, performance and security guarantees while avoiding unacceptable cognitive overhead, are quite common. Today, implementing new notations within an existing language requires the cooperation of the language designer. A primary reason for this is that, with conventional parsing strategies, not all notations can safely coexist, so a designer is needed to make choices about which syntactic forms are available and what their semantics are. 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, or key-value pairs. 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, in this case).

Languages that allow users to introduce new syntax from within libraries hold promise, but because there is no longer a designer making decisions about such ambiguities, the burden of resolving them falls to the clients of extensions. For example, SugarJ [13] and other extensible languages generated by Sugar* [14] allow providers to extend the base syntax of the host language (e.g. Java) with new forms, like set and map literals. 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 an understanding of the underlying parser technology (in Sugar*, GLR parsing using SDF) 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 frees these common notations from 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 order, while JSON does).

```
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 *)
```

Figure 3: Inline Generic Literal Forms

3.1 Wyvern

We develop our work as a variant of an emerging programming language being developed by our group called Wyvern [29]. 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). We also add recursive sum types, which we call *case types*, that operate similarly to datatypes in ML. One can refer to the version of the language described in this thesis as *TSL Wyvern* when the variant being discussed is not clear.

We begin with an example in Fig. 2 showing several different TSLs being used to define a fragment of a web application showing search results from a database. We will review this example below to develop intuitions about TSLs in Wyvern.

describe colors

3.2 Inline Literals

Our first TSL appears on line 1. The type of `imageBase` is `URL`, a *structural type* containing several fields representing the components of a URL: its protocol, domain name, port, path and so on. We could create a value of type `URL` using general-purpose notation:

```
1  let imageBase : URL = new
2    val protocol : String = "http"
3    val subdomain : String = "images"
4    val domain : String = "example"
5    (* ... *)
```

This is tedious. If the `URL` type has a TSL associated with it, we can instead instantiate precisely this value using conventional notation for URLs by placing it in a *generic literal*, `<images.example.com>`. 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)

```
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 : HasParser
9     val parser : Parser = ~
10      start -> ':body'= child::start>
11        `HTML.BodyElement(([], %child%))`
12      start -> ':style'= e::EXP[NEWLINE]>
13        let empty_attrs : Attributes = []
14        Exp.CaseIntro(Type.Var("HTML"), "StyleElement",
15          Exp.ProdIntro(valAST(empty_attr), e))
16      start -> '{'= e::EXP['}']>
17        `%e% : HTML`
```

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

is governed by the `URL` TSL. This TSL will parse the body at compile-time to produce a Wyvern AST that explicitly instantiates a new object of type `URL` (see below). Any other delimited form in Fig. 3 could equivalently be used if the constraints shown are obeyed.

In addition to supporting conventional notation for URLs, this TSL supports *typed interpolation* of a `URL` expression 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`. The TSL then assumes the interpolated expression is of this type to construct an AST. Untyped—and thus unsafe—interpolation of strings does not occur. Note that the delimiters that can be used to go from Wyvern to a TSL are determined by Wyvern (those in Fig. 3) while the delimiters that can be used to go from a TSL back to Wyvern are chosen by the TSL (see Sec. 3.5).

3.3 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. Declarations of some of these cases can be seen on lines 2-6 of Fig. 4 (TSL Wyvern also includes simple product types for convenience, written `T1 * T2`). The general-purpose introductory form for a case type like `HTML` is, e.g., `HTML.BodyElement((attrs, child))`. But, as discussed above, using this syntax can be 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 where the expected type is `HTML`, the literal body is governed by that type's TSL. The forward reference will be replaced by a general-purpose term of type `HTML` generated from the literal body by the TSL during compilation.

3.4 Implementing a TSL

Portions of the implementation of the TSL for `HTML` are shown on lines 8-17 of Fig. 4, written as a declarative grammar. A `Parser` is associated with a type using **`metadata`**, as shown. The **`metadata`** of a type `t` is simply a value that is associated with the type at both compile-time and run-time. It can be accessed using the syntactic form `t.`**`metadata`**. In this case, it is an object with a single field, called `parser` of type `Parser` (defined in Wyvern's standard library). A type equipped with a parser in this way is an example of an *active type*.

A grammar can be thought of as specialized notation that generates a parser. Since we have a type for parsers, `Parser`, we can implement a grammar-based parser generator itself as a TSL. In this case, we are basing our grammar formalism on the layout-sensitive formalism developed by Adams [6] – Wyvern is itself layout-sensitive and has a grammar that can be written down using this formalism, so it makes sense to support it for layout-sensitive TSLs as well. Most aspects of this formalism are completely conventional. Each non-terminal (e.g. `start`) is defined by a number of disjunctive productions, each introduced using `->`. Each production defines a sequence of terminals (e.g. `':body'`) and non-terminals (e.g. `start`), which can have names (e.g `child`) associated with them using `::`. Unique to Adams' formalism is that each terminal and non-terminal in a production can also have a *layout constraint* following it. Here, the layout constraints can be either `=` (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), `>=` (the leftmost column *may* be indented further). Layout constraints are optional in TSLs. We will discuss this further when we formally describe Wyvern's layout-sensitive concrete syntax.

Each rule is followed by a Wyvern expression, in an indented block, that implements the rewriting logic for the associated rule. This expression should have type `Exp`, a case type built into the standard library representing Wyvern expression ASTs. The ASTs generated by named non-terminals in the rule (e.g. `child`) are bound to the corresponding variables within this logic. Here, we show how to generate an AST using general-purpose notation for `Exp` (lines 13-15) as well as the more natural *quasiquote* style (lines 11 and 18). Quasiquotes are simply the TSL associated with `Exp` and support the full Wyvern concrete syntax as well as an additional delimiter form, written with `%s`, that allows "unquoting": interpolating another AST into the one being generated. Again, interpolation is typesafe, structured and occurs at compile time.

3.5 Implementing Interpolation

We have now seen several examples of interpolation. Within the TSL for `HTML`, for example, we see it used in several ways:

3.5.1 HTML Interpolation

At any point where a tag should appear, we can also interpolate a Wyvern expression of type `HTML` by enclosing it within curly braces (e.g. on line 13, 15 and 16-19 of Fig. 2). This is implemented on lines 17 and 18 of Fig. 4. The special non-terminal `EXP[token]` signals a switch into the Wyvern grammar. The tokenstream will be parsed as a Wyvern expression until the token `token` is encountered *where it would otherwise trigger a parse error*. In other words, the Wyvern grammar binds more tightly to itself than to any surrounding TSL, avoiding potential ambiguities. The AST for the parsed Wyvern expression is given an expected type, `HTML`, by simply surrounding it with a type ascription (line 18). Because interpolation must be structured (a string cannot be interpolated directly), injection and cross-site scripting attacks cannot occur. Safe interpolation of expressions of type `String` (where any inner angle brackets and other special characters are turned into HTML entities, e.g. `<` for `<`) could similarly be implemented using another delimiter.

3.5.2 CSS Interpolation

After the `:style` tag appears (e.g. on line 9 of Fig. 2), instead of hard-coding CSS syntax into the HTML DSL, we instead wish to use the TSL associated with a type representing a CSS stylesheet: `CSS`. We do this by again interpolating a Wyvern expression (lines 12-15 of Fig. 4), making sure that it appears in a position where the expected type is `CSS` (the second piece of data associated with the `StyleElement` constructor, in this case). Wyvern is given control until a full expression has been read and an unexpected newline appears (that is, a newline

that does not introduce a layout-delimited block). Here we see a use of a layout-delimited TSL within another layout-delimited TSL.

3.5.3 Interpolation within the CSS TSL

The TSL for `CSS` itself has support for interpolation in a similar manner, choosing `%` as the delimiter. It chooses the type based on the semantics of the surrounding CSS form. For example, when a Wyvern expression appears inside `url`, as on line 10 of Fig. 2, it must be of type `URL`. When a Wyvern expression appears where a color is needed, the `Color` type is used. This type itself has a TSL associated with it that interprets CSS color strings, showing again that TSLs can be used within TSLs by simply escaping out to Wyvern, the host language, and then back in. In this case, we emphasize that TSLs produced structured values by calling the `darken` method on it to produce a new color. This method itself takes a `Percentage` as an argument. The TSL for this type accepts literal bodies containing numbers followed by `pct`, or simply a real number without a suffix. Numeric literals, because they begin with a number (and no other form in Wyvern can), do not require delimiters (Fig. 3).

3.5.4 Interpolation within the SQLQuery TSL

The TSL used for SQL queries on line 18 of Fig. 2 follows an identical pattern, allowing strings to be interpolated into portions of a query in a safe manner. This prevents SQL injection attacks from occurring while maintaining standard SQL syntax.

3.6 Formalization

A formal and more detailed description can be found in our paper draft[5]. In particular:

should I include some/all of this in the proposal?

1. We provide a more complete layout-sensitive concrete syntax. We show how it can be written without the need for a context-sensitive lexer or parser and give a full specification for the layout-delimited literal form introduced by a forward reference, `~`, as well as other forms of forward-referenced blocks.

2. We detail the general mechanism for associating metadata with a type. A TSL is then implemented by associating a parser (of type `Parser`) with a type. The parser is responsible for rewriting tokenstreams (of type `Tokenstream`) into Wyvern ASTs (of type `Exp`). These types are defined in the standard library.

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.

4. We formalize the static semantics and literal parsing rules of TSL Wyvern as a bidirectional type system. By distinguishing locations where an expression synthesizes a type from locations where an expression is being analyzed against a previously synthesized type, we can precisely state where generic literals can appear. This system also provides a formal specification of our hygiene mechanism.

5. 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).

[5] `https://github.com/wyvernlang/docs/tree/master/ecoop14`

3.7 Remaining Tasks

The following tasks remain to be completed:

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 rigorous proofs of the metatheory.
2. We must further consider aspects of hygiene. In particular, we do not 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 (so that function application is the only way to pass data from a TSL to Wyvern code).
3. The corpus analysis we conducted was preliminary. We must perform this in a more complete and rigorous manner (collaborative work with Darya Kurilova).

4 Active Type Synthesis and Translation

In this section, we will restrict our focus to actively-typed mechanisms for implementing extensions to the static and dynamic semantics of programming languages. Programming languages are typically designed around a monolithic collection of primitive type and operator families. Consider, as a simple example, Gödel's T [21], a typed lambda calculus with recursion on primitive natural numbers (Figure 5). Although a researcher may casually speak of "extending Gödel's T with primitive product types" (Figure 6), modularly adding this new primitive type family and its corresponding operators to this language from within is impossible. That is, Gödel's T is not *internally extensible*.

$$\textsc{var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Arrow-I}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \textsc{Arrow-E}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \textsc{Nat-I1}\ \frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}$$

$$\textsc{Nat-I2}\ \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathsf{s}(e) : \mathtt{nat}} \qquad \textsc{Nat-E}\ \frac{\Gamma \vdash e_1 : \mathtt{nat} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_3 : \tau}{\Gamma \vdash \mathsf{natrec}(e_1; e_2; x, y.e_3) : \tau}$$

Figure 5: Static semantics of Gödel's T

The only recourse researchers have in such situations is to attempt to define new constructs in terms of existing constructs. Such encodings, collections of which are often called *embedded domain-specific languages (DSLs)* [17], must creatively combine the constructs available in the host "general-purpose" language. Unfortunately, such encodings can be difficult to conceive of and impractical or impossible in some cases. For our example of adding products to Gödel's T, a full encoding is impossible. Limited forms of Church encodings are possible, where products are represented by lambda terms,

$$\textsc{Prod-I}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \textsc{Prod-E1}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}(e) : \tau_1} \qquad \textsc{Prod-E2}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}(e) : \tau_2}$$

Figure 6: Static semantics of products

but they require a reasonable level of creativity[6] and in doing so, the type system cannot enforce a distinction between product types and the function types they are encoded using. It is also more difficult to reason about products represented in this way (for example, they will not have unique canonical forms). This strategy will also likely incur a performance penalty because it uses closures rather than a more direct and better optimized internal representation.

An internally-extensible programming language could address these problems by providing a language mechanism for extending, directly, its static and dynamic semantics, so as to support domain-specific type systems and implementation strategies in libraries. But, as mentioned in Section 1, some significant challenges must be addressed: balancing expressiveness with concerns about maintaining various safety properties in the presence of arbitrary combinations of user-defined extensions. The mechanism must ensure that desirable metatheoretic properties and global safety guarantees of the language cannot be weakened by extensions. And with multiple independently developed extensions used within one program, the mechanism must further guarantee that they cannot interfere with one another. Ideally, the correctness of an extension itself should be modularly verifiable, so that issues are detected before an extension is used.

5 From Extensible Compilers to Extensible Languages

To understand the genesis of our internal extension mechanism, it is helpful to begin by considering why most implementations of programming languages cannot even be externally extended. Let us consider, as a simple example, an implementation of Gödel's T, a typed lambda calculus with recursion on primitive natural numbers (see Appendix). A compiler for this language written using a functional language will invariably represent the primitive type families and operators using closed inductive datatypes. For example, a simple implementation in Standard ML may be based around these datatypes:

```
1  datatype Type = Nat | Arrow of Type * Type
2  datatype Exp = Var of var
3               | Lam of var * Type * Exp | Ap of Exp * Exp
4               | Z | S of Exp | Natrec of Exp * Exp * Exp
```

The logic governing typechecking and translation to a suitable intermediate language (for subsequent optimization and compilation by some back-end) will proceed by exhaustive case analysis over the constructors of `Exp`.

In an object-oriented implementation of Godel's T, we might instead encode types and operators as subclasses of abstract classes `Type` and `Exp`. Typechecking and translation will proceed by the ubiquitous *visitor pattern* by dispatching against a fixed collection of known subclasses of `Exp`.

In either case, we encounter the same basic issue: there is no way to modularly add new primitive type families and operators and implement their associated typechecking and translation logic.

A number of language mechanisms have been proposed that allow new cases to be added to datatypes and the functions that operate over them in a modular manner. In functional languages, we might use *open datatypes*. For example, if we wish to extend Gödel's T with product types and we have written our compiler in a language supporting open inductive datatypes, it might be possible to add new cases like this:

```
1  newcase Prod of Type * Type extends Type
2  newcase Pair of Exp * Exp extends Exp (* Intro *)
3  newcase PrL of Exp extends Exp     (* Elim Left  *)
4  newcase PrR of Exp extends Exp     (* Elim Right *)
```

[6] Anecdotally, Church encodings in System F were among the more challenging topics for students in our undergraduate programming languages course, 15-312. Note that System F, because it includes type abstraction (that is, parametric polymorphism), supports stronger encodings of types like products than System T does. But products are still only weakly definable in System F and the same fundamental problems discussed above occur.

The logic for functionality like typechecking and translation could then be implemented for only these new cases. For example, the `typeof` function that assigns a type to an expression could be extended like so:

```
1   typeof PrL(e) = case typeof e of
2       Prod(t1, _) => t1
3     | _ => raise TypeError("<appropriate error message>")
```

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 is not guaranteed (we will return to this point shortly). We have not, however, created an extensible programming language, for two reasons. First, compiler extensions are distributed and activated separately from libraries, so dependencies become more difficult to manage. Second, 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 using different compilers face the same problems with interoperability that those using different languages face. That is, extending a language by extending a single compiler for it is morally equivalent to creating a new language. Several prominent language ecosystems today are in a state where a prominent compiler has introduced or enabled the introduction of extensions that many libraries have come to rely on, including the Glasgow Haskell Compiler, SML/NJ and the GNU compilers for C and C++.

A more appropriate and useful place for extensions like this is directly within libraries, alongside abstractions that can be adequately implemented in terms of existing primitive abstractions. To enable this, the language must allow for the introduction new primitive type families, like `Prod`, operators, like `Pair`, `PrL` and `PrR`, and associated typechecking and translation logic. When encountering these new operators in expressions, the compiler must effectively hand control over typechecking and translation to the appropriate user-defined logic. Because this mechanism is language-internal, all compilers must support it to satisfy the language specification.

Statically-typed languages typically make a distinction between *expressions*, which describe run-time computations, and type-level constructs like types, type aliases and datatype declarations. The design described above suggests we may now need to add another layer to our language, an extension language, where extensions can be declared and implemented. In fact, we will show that **the most 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 (see Sec. **??** for examples). The type-level language is often constrained by its own type system (where the types of type-level values are called *kinds* for clarity) that prevents type-level functions from causing problems during compilation. This is precisely the structure that a distinct extension layer would have, and so it is quite natural to unify the two, as we will show in this work. We will proceed by developing an "actively-typed" lambda calculus, @λ, where type-level computation, typed compilation and type abstraction are all used to allow us to prove type safety and non-interference theorems. A draft of this calculus is available[7].

5.1 Ace

The syntax of @λ is abstract – there is a single uniform operator application form. To be practical, a wider variety of syntactic constructs must be used. A number of other practical considerations are also important. This section describes the design and implementation of Ace, an internally-extensible language designed considering both extrinsic and intrinsic criteria. To solve the bootstrapping problem, Ace is implemented entirely as a library

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

within the popular Python programming language. Ace and Python share a common syntax and package system, allowing Ace to leverage its well-established tools and infrastructure directly. Python serves as the compile-time metalanguage for Ace, but Ace functions themselves do not operate according to Python's fixed dynamically-typed semantics (cf. [35, 5]). Instead, Ace has a statically-typed semantics that can be extended by users from within libraries.

More specifically, each Ace function can be annotated with a base semantics that determines the meaning of simple expressions like literals and certain statements. The semantics of the remaining expressions and statements are governed by logic associated with the type of a designated subexpression. We call the user-defined base semantics *active bases* and the types in Ace *active types*, borrowing terminology from *active libraries* ([47], see Sec. **??**). Both are objects that can be defined and manipulated at compile-time using Python. An important consequence of this mechanism is that it permits *compositional* reasoning – active bases and active types govern only specific non-overlapping portions of a program. As a result, clients are able to import any combination of extensions with the confidence that link-time ambiguities cannot occur (unlike many previous approaches, as we discuss in Sec. **??**).

The *target* of compilation is also user-defined. We will show examples of Ace targeting Python as well as OpenCL, CUDA and C99, lower-level languages often used to program hardware. An active base or type can support multiple *active targets*, which mediate translation of Ace code to code in a target language. Ace functions targeting a language with Python bindings can be called directly from Python scripts, with compilation occurring implicitly. For some data structures, types can propagate from Python into Ace. We show how this can be used to streamline the kinds of interactive workflows that Python is often used for. Ace can also be used non-interactively from the shell, producing source files that can be further compiled and executed by external means.

The remainder of the section is organized as follows: in Sec. **??**, we describe the basic structure and usage of Ace with an example library that internalizes and extends the OpenCL language. Then in Sec. 4, we show how this and other libraries are implemented by detailing the extension mechanisms within Ace. To explain and demonstrate the expressiveness of these mechanisms (in particular, active types) further, we continue in Sec. **??** by showing a diverse collection of abstractions drawn from different language paradigms that can be implemented as orthogonal libraries in Ace. We include functional datatypes, objects, and several other abstractions. A full paper draft is available[8].

NOTE: Having an issue with tex macros but pretend sections 2-4 of the Ace paper are here basically as-is.

5.2 Remaining Tasks

- We need to fix a minor issue raised by a reviewer in the semantics of @λ about variables in internal terms when they are substituted.
- We need to write full proofs of the theorems stated in the @λ paper.
- We need to develop an elaboration of a subset of the Python grammar to @λ, to connect the two.
- We need to detail the examples in Sec. 4 of the Ace paper more substantially.

6 Active Code Completion

Software developers today make heavy use of the code completion support found in modern source code editors [28]. Most editors provide code completion in the form of a

[8] `https://github.com/cyrus-/papers/tree/master/ace-pldi14`

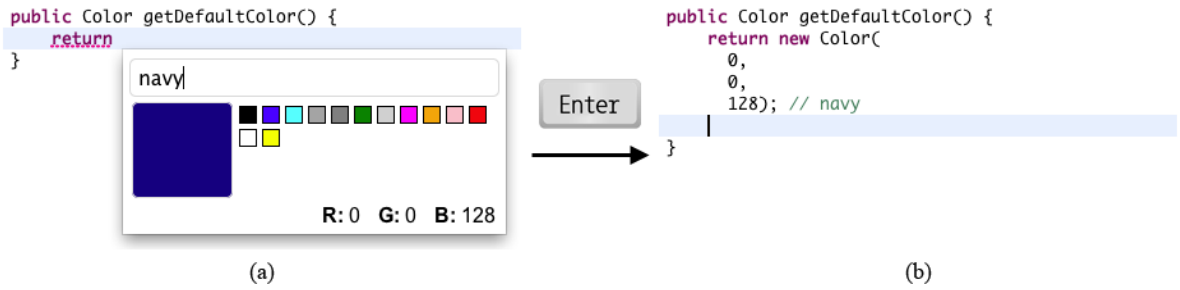

Figure 7: (a) An example code completion palette associated with the `Color` class. (b) The source code generated by this palette.

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 or API browser.

Several refinements and additions to the code completion menu have previously been suggested in the literature. These have focused on leveraging additional sources of information, such as databases of usage history [36][22], inheritance information [22], API-specific information [22][25], partial abbreviations [20], examples extracted from code repositories [9][7] and crowdsourced information [27][3], to increase the relevance and sophistication of the featured menu items. As with the standard form of code completion, many of these sources of data can also be utilized via external tools (e.g. Calcite [27] uses information that could already be accessed using the Jadeite [41] tool). Empirical evidence presented in these studies, however, suggests that directly integrating these kinds of tools into the editor is particularly effective. For example, users of the Calcite tool completed 40% more tasks in a lab study (unfortunately, a Jadeite control group was not included.)

In all of these systems, the code completion interface has remained primarily menu-based. When an item is selected, code is inserted immediately, without further input from the developer. These systems are also 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. In this paper we propose a technique called *active code completion* that eliminates these restrictions[9]. This makes developing and integrating a broad array of highly-specialized developer tools directly into the editor, via the familiar code completion command, significantly simpler. This technique is motivated by the evidence discussed above and further evidence provided in this paper that developers prefer, and make more effective use of, tools that do not require leaving the immediate editing environment.

In this paper, we discuss active code completion in the context of object construction. For example, consider the following Java code fragment:

```
1   public Color getDefaultColor() \{
2       return \textvisiblespace
```

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 1a gives an example of a simple palette that may be associated with the `Color` class[10].

[9] Portions of this work previously appeared in a poster abstract [31].

[10] A video demonstrating this process is available at

The developer can interact with such palettes to provide parameters and other information related to her 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 1b shows the inserted code after the user presses ENTER.

In accordance with best practices, 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 (Section II). 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 (Section III). Participants also suggested a large number of use cases, demonstrating the broad applicability of this technique. We organize these into several broad categories (Section IV).

Next, we describe Graphite, an Eclipse plug-in that implements the active code completion architecture for the Java programming language (Section V), allowing Java library developers to associate custom palettes with their own classes. We describe several design choices that we made to satisfy the requirements discovered in our preliminary investigations and briefly 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 (Section VI). The study provides specific evidence in support of the broader claim that highly-specialized tools that are integrated directly with the editing environment are particularly useful. We conclude that active code completion systems like Graphite are useful because they make developing, deploying and discovering such tools fundamentally simpler.

6.1 Remaining Tasks

- Related work: Murphy-Hill and Murphy, CSCW paper on peer interaction to enable discovery of tools; Improving program navigation with an active help system. CASCON 2010.; Improving software engineer's fluency by recommending dev environment commands. FSE 2012.
- Discuss safety more specifically.

7 Timeline

- @λ - ICFP - Mar. 1
- Wyvern - ECOOP or OOPSLA - late March

`http://www.cs.cmu.edu/~NatProg/graphite.html`.

- Ace - PLDI or OOPSLA - late March
- Then write everything up.

8 Conclusion

Todo list

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] Snipmatch.

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

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

[6] 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.

[7] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proc. 28th ACM Conference on Human Factors in Computing Systems (CHI'10)*, pages 513–522, 2010.

[8] 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.

[9] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proc. 7th European Software Engineering Conference & ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 213–222, 2009.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *Proc. 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, pages 332–343, 2009.

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

[22] D. Hou and D. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *Proc. 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 233 –242, 2011.

[23] 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.

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

[25] H. M. Lee, M. Antkiewicz, and K. Czarnecki. Towards a generic infrastructure for framework-specific integrated development environment extensions. In *Proc. 2nd International Workshop on Domain-Specific Program Development (DSPD'08), co-located with OOPSLA'08*, 2008.

[26] 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.

[27] M. Mooty, A. Faulring, J. Stylos, and B. Myers. Calcite: Completing code completion for constructors using crowds. In *Proc. 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'10)*, pages 15–22, 2010.

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

[29] 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.

[30] 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.

[31] C. Omar, Y. Yoon, T. LaToza, and B. Myers. Active code completion. In *Proc. 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, pages 261–262, 2011.

[32] 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.

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

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

[35] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, OOPSLA '13, pages 217–232, New York, NY, USA, 2013. ACM.

[36] R. Robbes and M. Lanza. How program history can improve code completion. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 317–326, 2008.

[37] 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.

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

[39] 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.

[40] 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.

[41] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers. Improving API documentation using API usage information. In *Proc. 2009 IEEE Symposium on Visual Language and Human-Centric Computing (VL/HCC'09)*, pages 119–126, 2009.

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

[43] 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.

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

[45] 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.

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

[47] 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.

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