Type-Oriented Foundations for Safely Extensible Programming Systems

PhD Thesis Proposal

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

February 5, 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 also being 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) [44]. 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 [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 to ensure that string manipulation operations do not inadvertently lead to a string that is not in an assumed language [18].

   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 [40]) and security vulnerabilities (due to injection attacks when user inputs are spliced into patterns, for example) [**?**]. 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 [**?**] and can be more awkward than necessary [11, 32], leading to lower productivity [**?**, 32].

complete references

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 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.

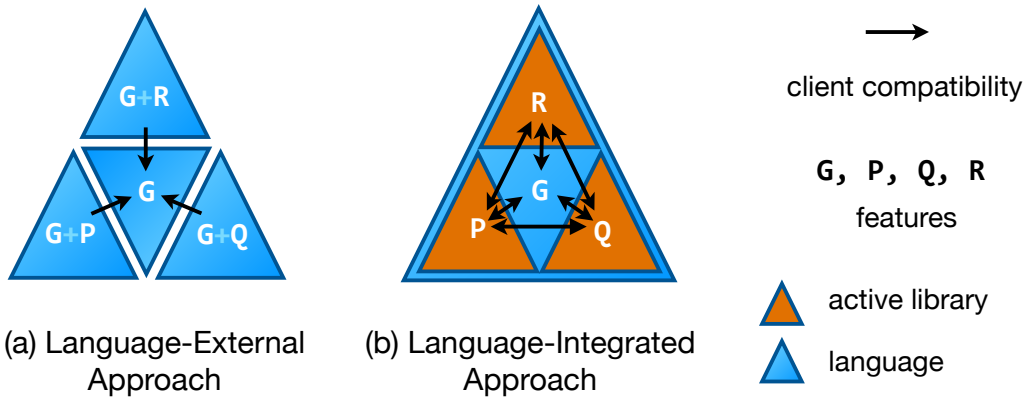

Figure 1: (a) When taking a language-external approach, new features are packaged together into separate languages and tools, causing problems with compatibility (described in the text). (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time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

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 emphasizes the importance of this problem for driving research into practice: 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 associated implementation strategies. This is problematic because different parallel programming abstractions or implementation strategies are more appropriate in different situations [43]. 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 has been constructed 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 [48] 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 [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 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 [42, 24], 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 [26] because they have no straightforward 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[2] 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* [47] 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.

[2]One might compare today's programming systems to centrally-planned economies, whereas extensible-systems more closely resemble modern market economies. Our safety constraints serve a role analagous to market regulation. We leave further development of this analogy to the reader.

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. [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 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 [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 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 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 [47]. 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 [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 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 [13] and other languages 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. 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`[3]). 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 [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 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

[3]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.

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).

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) as it is defined and scoping it only around 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 [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 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 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 [40]) or to execute a decision procedure for language inclusion when determining whether a coercion is possible (feature 2c, see [18]). 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. 4.1 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. 4.2 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. 5, we will show how **editor services** that interactively help developers correctly and productively introduce terms of a particular type can be introduced 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 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 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, introduce them by naming the appropriate case and providing the associated data, if any, 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 [19], 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 those 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 [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 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 [13] and other extensible languages generated by Sugar* [14] 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

```
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

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).

3.1 Wyvern

We develop our work as a variant of a new 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). Objects are thus essentially 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. We will go through this example below to develop intuitions about TSLs in Wyvern. 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:

fix comment formatting

new color

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

```
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

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 8-17 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

```
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.

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 [6] – Wyvern is itself layout-sensitive and has a grammar that can be written down using this formalism, so it is sensible to expose it to TSL providers as well. Most aspects of this formalism are completely conventional. Each non-terminal (e.g. `start`) is defined by a number of disjunctive productions, each introduced using `->`. Each production defines a sequence of terminals (e.g. `':body'`) and non-terminals (e.g. `start`, or one of the built-in non-terminals `ID`, `EXP` or `TYPE`, representing Wyvern identifiers, expressions and types, respectively). Unique to Adams' formalism is that each terminal and non-terminal in a production can also have an optional *layout constraint* associated with it. The layout constraints available are `=` (meaning that the leftmost column of the annotated term must be aligned with that of the parent term), `>` (the leftmost column must be indented further) and `>=` (the leftmost column *may* be indented further). We will discuss this formalism further when we formally specify Wyvern's layout-sensitive concrete syntax.

Each production is followed, in an indented block, by a Wyvern function that generates a value given n arguments, which will bound to 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 should return a value of type `Exp`. 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 their corresponding syntax tree. 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. This is accomplished 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.

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

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

3.6 Formalization

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

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 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 [**?**]) with an elaboration semantics (in the style of Harper and Stone [**?**]). 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). and Josh?

3.7 Remaining Tasks and Timeline

The following tasks remain to be completed. I have scheduled this for the period of March 2nd-25th, before the OOPSLA submission deadline.

1. In our current top-level grammar, expressions within parenthesis must still obey layout constraints. This is not strictly necessary for preventing ambiguities, and it is quite convenient to not require this (as in Python, for example). We will add these simple rules to the grammar.

2. 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.

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

3. 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.

4. We must further consider aspects of hygiene. In particular, 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 (so that function application is the only way to pass data from a TSL to Wyvern code, as in the `Parser` TSL we have shown).

5. 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

General-purpose abstraction mechanisms, like those available in Wyvern and other languages today, are powerful, and should be used when possible. But relying on a small fixed collection of them may not always be satisfactory, for several reasons:

1. General-purpose abstraction mechanisms themselves are considerably varied. Consider, for example, the many different ways languages expose product types: simple products, n-ary products, labeled products[5], structurally-typed labeled products, labeled products with mutable fields, labeled products with methods, labeled products with prototypic inheritance, labeled products with delegation, labeled products with class-based inheritance, and various combinations of these. New mechanisms and variants are constantly being developed. To allow these new general-purpose mechanisms to be evaluated in a controlled manner in practice, it must be possible to import them into existing codebases in a piecemeal fashion.

2. Specialized type systems that directly enforce stronger invariants than general-purpose abstraction mechanisms are capable of straightforwardly enforcing are often developed. In addition to the examples of type systems for regular expressions in Sec. 1.1 ([**?**, **?**]), there are also specialized type systems for parallel programming [**?**, **?**], concurrent programming [**?**, **?**, **?**], distributed programming [**?**], reactive programming [**?**], authenticated data structures [**?**], security policies [**?**], information flow [**?**], web programming [**?**], aliasing [**?**], network protocols [**?**], units of measure [**?**] and many others. These type systems are implemented by language-external means (e.g. by creating a new variant of ML, Java or C), leading to the problems of Sec. 1.2.

3. Interoperability layers with external and legacy systems are another major area where support for directly implementing a new type system (that of the external language) and implementing its primitives in terms of a low-level interoperability API would be useful (e.g. [**?**]).

In this section, 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 will begin by taking a first-principles type-theoretic approach in Sec. 4.1, focusing on safety issues, and continue on in Sec. 4.2 by designing a full 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, bootstrapping an extensible static type system using the quotation capabilities of uni-typed Python.

[5]We use the phrase *labeled product* to avoid the connotations of related terms like *record*, *struct* and *object*.

4.1 Foundations

Programming languages are typically organized around a monolithic collection of indexed type and operator constructors. Let us begin by considering the simply-typed lambda calculus (STLC). It provides a single type constructor, indexed by a pair of types, that we will call ARROW for uniformity. It also provides two operator constructors: λ, indexed by a type, and ap, indexed trivially. We can specify the static semantics of the STLC as follows, using a uniform abstract syntax where type and operator indices are written in braces to emphasize this way of thinking about its 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.

Although a researcher may casually speak of "extending the STLC" with a primitive natural number type and its corresponding operators (as in Gödel's T; Figure 7) or primitive product types and their corresponding operators (Figure 8), it is clearly not possible to directly introduce these from within the STLC.

$$\textsc{nat-I1}\ \frac{}{\Gamma \vdash \mathtt{z}[()]() : \textsc{Nat}[()]} \qquad \textsc{nat-I2}\ \frac{\Gamma \vdash e : \textsc{Nat}[()]}{\Gamma \vdash \mathtt{s}[()](e) : \textsc{Nat}[()]}$$

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

Figure 7: Static semantics of natural numbers.

$$\textsc{prod-I}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}[()](e_1; e_2) : \textsc{Prod}[(\tau_1, \tau_2)]} \qquad \textsc{prod-E1}\ \frac{\Gamma \vdash e : \textsc{Prod}[(\tau_1, \tau_2)]}{\Gamma \vdash \mathtt{prl}[()](e) : \tau_1} \qquad \textsc{prod-E2}\ \frac{\Gamma \vdash e : \textsc{Prod}[(\tau_1, \tau_2)]}{\Gamma \vdash \mathtt{prr}[()](e) : \tau_1}$$

Figure 8: Static semantics of products

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 sometimes called *embedded domain-specific languages (EDSLs)* [17], must creatively combine the constructs available in the host "general-purpose" language to encode a desired semantics. Unfortunately, finding an encoding that fully captures both the static and dynamic semantics of a desirable construct is not always possible. Even when it possible, it may be impractical to work with and optimize. For our example of adding natural numbers or products to the STLC, a reasonable encoding is impossible. In closely-related languages, like the polymorphic lambda calculus (a.k.a. System F), such constructs are weakly definable [**?**]. However, developing such an encoding requires a level of creativity beyond that needed to understand the intended semantics[6]. The type system also does not enforce a distinction between the type of natural numbers and the polymorphic function type used for the encoding. And in practice, working with numbers encoded as polymorphic functions directly can be awkward and can also incur a performance penalty, due to the use of closures rather than a more direct representation.

[6]Anecdotally, Church encodings in System F were among the more challenging topics for students in our undergraduate programming languages course, 15-312.

An extensible programming language could address these problems by providing a language-integrated mechanism for introducing new type and operator constructors and implementing their associated static and dynamic semantics. 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, determinism, decidability) of the language are maintained by extensions. Because multiple independently developed extensions might be used within one program, the mechanism must further guarantee their *non-interference*. These are the issues we seek to address in this work.

4.1.1 From Extensible Compilers to Extensible Languages

The monolithic character of most programming languages is reflected in, and perhaps influenced by, the most common techniques used for implementing programming languages. Programming languages are not implemented using open representations of types or expressions. 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 may be based around these datatypes, for example:

```
1  datatype Type = Arrow of Type * Type
2  datatype Exp = Var of var | Lam of var * Type * Exp | Ap of Exp * Exp
```

The compiler front-end, which consists of a typechecker and translator to a suitable intermediate language (for subsequent compilation by some suitable back-end), 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 `Type` and `Exp`. Typechecking and translation, however, will proceed by the ubiquitous *visitor pattern*, 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 and operator constructors to `Type` and `Exp`, respectively, and provide implementations of 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 our implementation of the STLC with product types and we have written our compiler in a language supporting open inductive datatypes, it might be possible to add new cases:

cite

```
1  newcase Prod of Type * Type extends Type
2  newcase Pair of Exp * Exp extends Exp
3  newcase PrL of Exp extends Exp
4  newcase PrR of Exp extends Exp
```

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

```
1  typeof (ctx, PrL(e)) = case typeof (ctx, 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 and non-interference 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). That is, extending a language by extending a single compiler for it is morally equivalent to creating a new language.

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, all compilers must support it to satisfy the language specification.

The design described above suggests we may now need to add another layer to our language, beyond the type-level language (τ) and expression language (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. This is often more constrained than the expression language is (e.g. it might guarantee termination, to ensure that typechecking is decidable). This is precisely the structure that a distinct extension layer would have, and so it is quite natural to unify the two.

4.1.2 @λ

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 abstract types to allow us to prove key type safety and non-interference theorems.

We specify the semantics of the external language as a Harper-Stone style elaboration semantics: all terms in the external language are simultaneously assigned a type and an elaboration to a term in a fixed typed internal language [**?**]. The mechanism will check that the elaboration is type preserving, following upon work done on the TIL compiler for Standard ML by Morrisett et al. [**?**], so that the semantics can be reasoned about compositionally and so that type safety can be proven.

Any global properties that the internal language maintains are guaranteed to hold no matter which extensions are used. In other words, extension providers cannot implement type systems weaker than the type system of the internal language. Providers are able to implement type systems stronger than the internal type system. We are interested in permitting only deterministic, decidable extensions, so semantic extensions are implemented as total functions in the type-level language (essentially, as a typechecker), rather than extracted from inductive specifications, like those in the Sec. 4.1[7]. This has the added benefit of giving providers finer-grained control over error messages – with a strictly inductive specification, it is difficult to provide different error messages for different failure points in a derivation.

To give an example, the fragment in Fig. 9 shows how to introduce a new type constructor, `Nat`, indexed trivially (i.e. by a value of kind unit). Each type constructor must have a *representation schema* associated with it that determines an associated internal type for every value the index might take. The representation schema of NAT is given on line 2 by a type-level function from the index to an internal type, $\mathbb{Z}$. Internal types, σ, are reflected into the type-level language, τ, via the form $\blacktriangledown(\sigma)$.

[7]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 unit with
2     schema λidx:unit.▼(ℤ)
3     opcon Z of unit (λidx:unit.λargs:list[Den].empty args [[▽(0) as Nat[()]]])
4     opcon S of unit (λidx:unit.λargs:list[Den].pop_final args λtrans:ITm.λty:⋆.
5       if ty = Nat[()] then [[▽(△(trans)+1) as Nat[()]]] else err("...S requires nat..."))
6     opcon Rec of unit (λidx:unit.λargs:list[Den].pop args λ.a.a.a..aa.a)
7  end
```

Figure 9: An implementation of primitive natural numbers as internal integers in @λ.

A key decision that we make is to associate operator constructors directly with type constructors. This provides a scope for information hiding, as we will discuss, and also forms the foundation for a higher-level elaboration mechanism that allows us to use a more conventional (though still fixed) concrete syntax for expressions, rather than the uniform abstract syntax that the language as specified uses. We will explore this in the design of Ace. The definitions of three new operator constructors are given on lines 3-XXX. finish this

A draft of the full definition of this calculus is available as a paper draft[8].

4.2 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 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

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

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[9].

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

4.3 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.

5 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 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. 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.

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

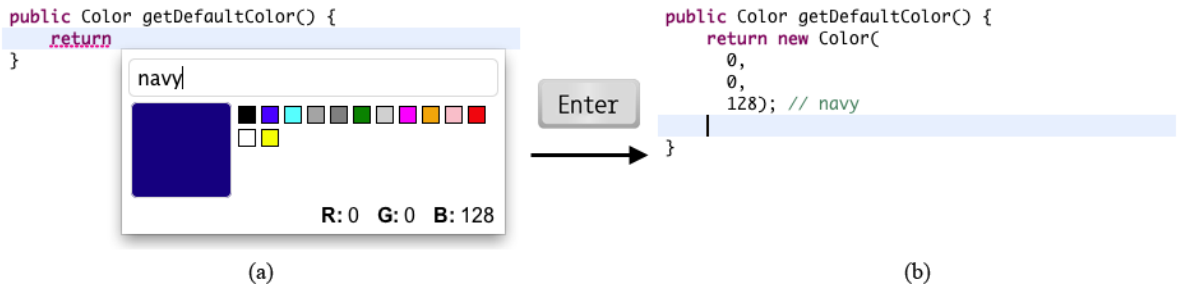

Figure 10: (a) An example code completion palette associated with the `Color` class. (b) The source code generated by this palette.

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].

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

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

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.

5.1 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.

6 Timeline

- @λ - ICFP - Mar. 1
- Wyvern - ECOOP or OOPSLA - late March
- Ace - PLDI or OOPSLA - late March
- Then write everything up.

7 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.