Safely Extending Typed Programming Systems From Within

PhD Thesis Proposal

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

April 13, 2014

Abstract

We propose a thesis defending the following statement:

> ***Active types*** *allow abstraction providers to extend a programming system with new notations, strengthen its type system and implement specialized editor services from within libraries in a safe and expressive manner.*

1 Motivation

Specifying and implementing a "general purpose" programming language and its associated tools (collectively, a *programming system*) has long been a grand challenge in computing. Were a truly general system to emerge, we might expect that its users would rarely need to develop a new dialect of the system. Embeddings of new abstractions using mechanisms built into the language would be suitably reasonable, natural and performant. Alas, new language and tool dialects spanning all major language lineages continue to be widely used as vehicles for new abstractions (examples of which we will discuss throughout this work), suggesting that not all useful abstractions can be seen, when considered comprehensively, as mere modes of use of any contemporary "general purpose" programming system.

We observe that new dialects are often motivated by the desire to introduce new syntax, strengthen the language's static semantics or provide specialized editor services, all aspects of existing systems that appear fixed when considered from within (that is, from the perspective of a library provider). It follows then that language-integrated mechanisms that allow library providers to extend these features of the system could strengthen its claim to generality by decreasing the need for new language and tool dialects. However, such mechanisms must be introduced with care, because decentralizing control over such fundamental aspects of the system could weaken its metatheory and permit ambiguities and conflicts when two libraries are combined (collectively, *safety issues*). It could also make it more difficult to define new functions over expression forms in the language (the *expression problem*) and make it more difficult to understand the meaning of programs. In this thesis, we aim to show that by associating system extensions with types (forming what we call *active types*) rather than new expression forms, and using a *bidirectional type system* that enforces critical abstraction barriers between extensions, we can give abstraction providers the ability to orthogonally implement new syntax, type systems and editor services from within libraries, rather than as new language and tool dialects, while avoiding safety issues and retaining an essentially conventional typing discipline.

1.1 Motivating Example: Regular Expressions

To make the problems we aim to address more concrete, we begin with a simple example that we will return to throughout this work. *Regular expressions* are commonly used to express patterns in strings (e.g. DNA sequences) [68]. Programmers who work with regular expressions regularly might want a programming system supporting features like:

1. **Syntax for pattern literals**. An ideal syntax would permit us to express patterns in a concise, conventional manner. For example, The BisI restriction enzyme cuts DNA whenever it sees the pattern `GCNGC`, where `N` represents any base. We would want to express it, perhaps, as follows (using curly braces to splice one pattern into another):

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

   We would want malformed patterns to result in intelligible compile-time errors.

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

   (a) only other patterns and properly escaped strings are spliced into a pattern, to avoid splicing errors and injection attacks [5, 11]

   (b) out-of-bounds backreferences to a captured group are not used [62]

   (c) string processing operations do not lead to a string that is malformed, when well-formedness can be captured as membership in a regular language [25, 32]

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

No system today builds in support for all of the features enumerated above in their strongest form, so library providers must provide support for regular expressions by leveraging general-purpose mechanisms. Unfortunately, it is impossible to completely define the syntax and the specialized static semantics referenced above in terms of general-purpose notations and abstractions, so library providers need to compromise.

The most common strategy is to ask clients to enter patterns as strings, deferring their parsing, typechecking, compilation and use all to run-time. This provides only a partial approximation to feature 1 (due to clashes between string escape sequences and regular expression syntax). None of the static guarantees can be provided in this way, leading to run-time exceptions (even in well-tested code [62]), and logic errors that cannot even be caught at run-time, some of which can lead to security vulnerabilities (due to injection attacks [5]). It also introduces performance overhead due to run-time parsing and compilation of patterns, and redundant run-time checks.

Requiring that clients instead introduce patterns directly using general-purpose constructs like sums or products (as exposed by, e.g., functional datatypes or objects) or operations over an abstract type, rather than via strings, provides only feature 2a at the expense of the weak approximation to feature 1 that the use of string literals provided.

None of these approaches address the issue of tool support (feature 3). Regular expressions are not trivial to work with, and we have found that tool support can be quite helpful, even for professional programmers [50]. Indeed, a number of tools are available online that provide services like syntax highlighting, testing and explanations of patterns [2] and pattern extraction from examples (e.g. [3]). Unfortunately, tools that must be accessed externally are difficult to discover and are thus used infrequently [45, 14, 50]. We found that they are also less usable than editor-integrated tools because they require the programmer to switch contexts and cannot make use of the code context [50]. Working with high-level abstractions in a "low-level" editor is thus awkward and error-prone, just like attempting to work with high-level abstractions in a low-level language. For these reasons, we consider editor behavior as within the scope of an abstraction's specification.

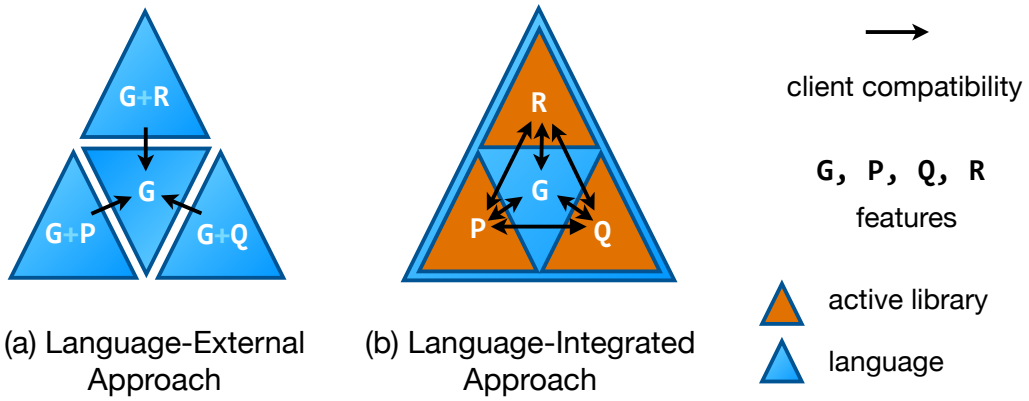

Figure 1: (a) When taking a language-external approach, new features are packaged together into separate languages and tools, causing problems with orthogonality and client compatibility (described in the text). (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time logic governing new constructs is expressed within "active libraries". If the extension mechanism precludes conflicts by construction, the problems of orthogonality and client compatibility are avoided.

1.2 Language-External Approaches

When the syntax and semantics of a system must be extended to fully realize a new feature providers typically take a *language-external approach*, either by developing a new system dialect (supported by *compiler generators* [12], *language workbenches* [23], *DSL frameworks* [24], or simply be forking an existing codebase), or by using an extension mechanism for a particular compiler[1], editor or other tool. For example, a researcher interested in providing regular expression related features (let us refer to these collectively as `R`) might design a new system with built-in support for these, perhaps basing it on an existing system containing some general-purpose features (`G`). A different researcher developing a new language-integrated parallel programming abstraction (`P`) might take the same approach. A third researcher, developing a type system for reasoning about units of measure (`Q`) might again do the same. This results in a collection of distinct systems, as diagrammed in Figure 1a. This might be entirely sufficient if developing a proof of concept is the only goal, but it makes it difficult to achieve adoption of these abstractions (even by other researchers) related to **orthogonality** and **client compatibility**.

Orthogonality Features implemented by language-external means cannot be adopted individually, but instead are only available coupled to a fixed collection of other features. This makes adoption more costly when these incidental features are not desirable or insufficiently developed, or when the features bundled with a different language or tool are simultaneously desirable. That is, one must either use the system containing features `G+R`, `G+P` or `G+Q`. There is no system containing `G`, `R`, `P` and `Q` in other combinations, and merging the systems containing each separately can be non-trivial because, even in cases where a common mechanism has been used, there are serious interference and safety issues, as we will discuss at length throughout this thesis.

Recent evidence indicates that this is one of the major barriers preventing research from being driven into practice. For example, developers prefer language-integrated

[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. That is, some programs that purport to be written in C, Haskell or Standard ML are actually written in compiler-specific dialects of these languages.

parallel programming abstractions with stronger semantic guarantees, more efficient implementation and more natural syntax to library-based approximations when all else is equal [18], but library-based approximations are more widely adopted because "parallel programming languages" privilege only a few chosen abstractions at the language level. This is problematic, because different abstractions are seen as more appropriate in different situations [66]. Moreover, parallel programming is rarely the only concern relevant to clients outside of a classroom setting. Support for regular expressions, for example, would be simultaneously desirable for processing large amounts of genomic data in parallel, but using these features together in the same compilation unit would be difficult or impossible. Indeed, switching to a "parallel programming language" would likely make it *more* difficult to use regular expressions, as these are likely to be less well-developed in a specialized language than in an established general-purpose language. The intuition was perhaps most succinctly expressed by a participant in a recent study by Basili et al. [8]: "I hate MPI, I hate C++. [But] if I had to choose again, I would probably choose the same."

Client Compatibility Even in cases where, for each component of a software system, there was a programming system considered entirely satisfactory by its developers, there would remain a problem at the interface between its components. An interface that exposes externally a specialized construct particular to one language (e.g. a function that requires a quantity having a particular unit of measure) cannot necessarily be safely and naturally consumed from another language (e.g. a parallel programming language). Tool support is also lost when calling into different languages. We call this 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 taken by proponents of the language-oriented approach to abstraction development [73] to partially address the client compatibility problem is to target an established intermediate language and use its constructs as a common language for communication between components written in different languages. Scala [48] and F# [52] are examples of prominent general-purpose languages that have taken this approach, and most DSL frameworks also rely on this strategy. As indicated in Figure 1a, this only enables client compatibility in one direction. Calling into the common language becomes straightforward and safe, but calling in the other direction, or between the languages sharing the common target, does not, unless these languages are only trivially different from the intermediate language.

As a simple example with significant contemporary implications, F#'s type system does not admit `null` as a value for any type both defined and used within F# code, but maintaining this sensible internal invariant still requires dynamic checks because the stricter typing rules of F# do not apply when F# data structures are constructed by other languages on the Common Language Infrastructure (CLI) like C# or SML.NET. This is not an issue exclusive to intermediate languages that make regrettable choices regarding `null`, however. The F# type system also includes support for checking that units of measure are used correctly [64, 34], but this more specialized static invariant is left entirely unchecked at language boundaries. Exposing functions that operate over datatypes, tuples and values having units of measure is not recommended when a component "might be used" from another language [64] because it is awkward to construct and consume these from other languages without the convenient primitive operations (e.g. pattern matching) and syntax that F# includes. SML.NET prohibits exposing such types at component boundaries altogether. It cannot naturally consume F# data structures, despite having a rather similar syntax and semantics in most ways (both languages directly descend from ML).

1.3 Language-Integrated Approaches

We argue that, due to these problems with orthogonality and client compatibility, taking a language-external approach to realizing a new feature should be considered harmful and

avoided whenever possible. The goal of the research being proposed here is to design *language-integrated extension mechanisms* that give providers the ability to define, within libraries, new features that have previously required central planning, so that language-external approaches are less frequently necessary. More specifically, we will show how control over aspects of the **syntax**, **type system** 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* [71] because, rather than being passive clients of features already available in the system, they contain logic invoked by the components of the programming system while a program is being edited or compiled to provide new features. Features implemented within active libraries can be imported as needed, unlike features implemented by external means, seemingly avoiding the problems of orthogonality and client compatibility.

We must proceed with caution, however: critical issues having to do with safety must be overcome before language-integrated extension mechanisms can be integrated into a system. If too much control over these core aspects of the system is given to developers, the system may become quite unreliable. Type safety, for example, may not hold if the static and dynamic semantics of the language can be modified or extended arbitrarily from within libraries. Furthermore, even if extensions can be shown not to cause such problems in isolation, there may still be conflicts between extensions that could weaken their semantics, leading to subtle problems that only appear when two extensions are used together. As a simple 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 within the same scope. Resolving these kinds of ambiguities requires significantly more expertise with parser technology than using the syntax itself does. These (and other more serious) safety and ambiguity 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. [71, 70] to describe "libraries that take an active role in compilation, rather than being passive collections of subroutines". The authors suggested a number of reasons libraries might benefit from being able to influence the programming system directly, including high-level program optimization, checking programs for correctness against specialized criteria, reporting domain-specific errors and warnings, and "rendering domain-specific textual and non-textual program representations and for interacting with such representations" (anticipating interactions between libraries and tools other than just the compiler).

The first concrete realizations of active libraries in statically typed settings, prompting the introduction of the term, were libraries that performed domain-specific program optimization at compile-time by exploiting language mechanisms that allow for limited compile-time computation. A prominent example in the literature is Blitz++, a library that uses C++ template metaprogramming to optimize compound operations on vectors and matrices by eliminating intermediate allocations [69]. Although this and several other interesting optimizations are possible by this technique, its expressiveness is fundamentally limited because template expansion allows for only the substitution of compile-time constants into pre-written code, and template metaprograms are notoriously difficult to read, write, debug and reason about (see discussion in [58]).

More powerful and direct compile-time *term rewriting mechanisms* available in some languages can also be used for optimization, as well as for introducing specialized error checking logic and implementing new abstractions. For example, typed macros, such as those in MetaML [60], Template Haskell [61] and Scala [13], take full control over all of the code that they enclose. This can be problematic, however, as outer macros can interfere with inner macros. Moreover, once a value escapes a macro's scope, there is no way to rely

on the guarantees and features that were available within its scope, because the output of a macro is simply a term in the underlying language (a problem fundamentally related to the problem of relying on a common intermediate language, described in Section 1.2). Thus, macros represent at best a partial solution: they can be used to automate code generation, but not to globally extend the syntax or static guarantees of a language. It can also be difficult to reason about the semantics of code when any number of enclosing macros may be manipulating it, and to build tools that operate robustly in their presence.

Some term rewriting systems replace the delimited scoping of macros with global pattern-based dispatch. Xroma (pronounced "Chroma"), for example, allows users to insert custom rewriting passes into the compiler from within libraries [71]. Similarly, the Glasgow Haskell Compiler (GHC) allows providers to introduce custom compile-time term rewriting logic if an appropriate flag is passed in [33]. In both cases, the user-defined logic can dispatch on arbitrary patterns of code throughout the component or program the extension is activated within. But libraries containing such global rewriting logic cannot be safely composed because two different libraries may attempt to rewrite the same piece of code differently. It is also difficult to guarantee that such logic is correct and difficult to reason about code when simply importing a library can change the semantics of the program in a highly non-local manner.

Another example of an active library approach to extensibility with non-local scope is SugarJ [20] and other languages generated by Sugar* [21], like SugarHaskell [22]. These languages permit libraries to extend the base syntax of the core language in a nearly arbitrary manner, and these extensions are imported transitively throughout a program. Unfortunately, this flexibility again means that extensions are not safely composable. For example, a library that defines a literal syntax for HTML would conflict with another that defines a literal syntax for XML because they define differing semantics for some of the same syntactic forms. If SugarJ was used by two different regular expression engines to provide literal syntax for regular expression patterns, there could easily be conflicts at link-time because both will introduce many of the same notations but back them with differing implementations. And again, it is difficult to predict what an unfamiliar piece of syntax desugars into, leading to difficulties reading and reasoning about code.

2 Active Types

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

To motivate our approach, let us return to our example of regular expressions. We observe that every feature described in Sec. 1.1 relates specifically to how terms classified by a single user-defined type (or indexed family of types) should behave. In fact, nearly all the features relate to types representing regular expression patterns. Feature 1 calls for specialized syntax for the introductory form. Features 2a and 2b relates to how operations on patterns should be typechecked. Feature 3 discuss services only relevant when editing a pattern. Feature 2c relates to the semantics of a related family of types: strings known to be in the language of a statically-known regular expression.

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

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

In two major textbooks about programming languages, *TAPL* [53] and *PFPL* [30], most chapters describe the syntax, semantics and metatheory of a new type constructor and its associated operators (collectively, a *fragment*) in isolation. Combining the fragments from different chapters into complete languages is, however, a language-external (that is, metamathematical) operation. In *PFPL*, for example, the notation $\mathcal{L}\{\rightarrow \texttt{nat dyn}\}$ represents a language that combines the arrow ($\rightarrow$), `nat` and `dyn` type constructors and their associated operators. Each of these are defined in separate chapters, and it is generally left unstated that the semantics and metatheory developed separately can be combined conservatively, i.e. with all the fundamental metatheory left intact, a notion we will refine later. This is justified by a sense that the rules in each chapter seem "well-behaved" in that they avoid referring excessively to fragments in other chapters, and thus preserve their autonomy.

A fragment that violates the autonomy of another fragment could easily be defined. For example, introducing a new value of a type defined elsewhere would render invalid any conclusions arrived at by induction over values of that type, clearly bad behavior. If we can somehow formalize this notion of a "fragment" and "autonomy" and internalize it into a language itself, rather than leaving it as a "design pattern" that only informally guides the work of a central language (or textbook) designer, we might achieve a safely extensible language, i.e. one where separately defined embeddings of fragments as libraries can be safely combined. Doing so without limiting expressiveness is precisely the topic of this thesis. We adopt the practice of organizing a fragment around the types it introduces, and call a type associated with new syntax, semantics or editor services an *active type*. This choice, of associating extensions with types rather than expression forms, avoids many issues collectively associated with the *expression problem*, as we will discuss later.

2.1 Proposed Contributions

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

We begin in Sec. 3 by considering **syntax**. The availability of specialized syntax can bring numerous cognitive benefits [27], and discourage the use of problematic techniques like using strings to represent structured data [11]. But allowing library providers to add arbitrary new syntactic forms to a language's grammar can lead to ambiguities, as described above. We observe that many syntax extensions are motivated by the desire to add alternative introductory forms (a.k.a. *literal forms*) for a particular type. For example, regular expression pattern literals as described in Sec. 1.1 are an introductory form for the `Pattern` type. In the mechanism we introduce, literal syntax is associated directly with a named type and can be used only where an expression of that type is expected (shifting part of the burden of parsing into the typechecker). This avoids the problem of an extension interfering with the base language or another extension because these grammars are never modified directly. We begin by introducing these *type-specific languages (TSLs)* in the context of a simplified variant of a language we are developing called Wyvern. Next, we show how interference issues in the other direction – the base language interfering with the TSL syntax – can be avoided by using a novel layout-delimited literal form. We then develop

a formal semantics, basing it on work in bidirectional type systems and typed elaboration semantics (which we together call a *bidirectionally typed elaboration semantics*). Using this semantics, we introduce a mechanism that statically prevents a third form of interference: unsafe variable capture and shadowing (a form of *hygiene*).

Wyvern has an extensible concrete syntax but a fixed "general-purpose" static and dynamic semantics. The constructs 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 type constructors and operators. Examples of language 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 [62]) or to execute a decision procedure for language inclusion when determining whether a coercion requires a run-time check (feature 2c, see [25, 32]). We discuss examples where researchers had to turn to language-external approaches for this sort of reason in Sec. 4.

To support these more advanced use cases in a decentralized manner, we next develop mechanisms for implementing **type system** extensions. We begin with a core calculus, @λ, in Sec. 5, then design a "practical" implementation, Ace, as a library inside Python in Sec. 6 (which serves as an interesting challenge because Python begins with an initially quite impoverished static semantics). Both are organized around a bidirectionally typed translation semantics (which differs from an elaboration semantics in that the target language has a different type system, as we will discuss) that leverages type-level computation to permit the implementation of new type system fragments. The core calculus supports our claims of safety – we discuss issues including type safety (using techniques borrowed from the typed compilation literature), decidability and conservativity (that embeddings of a type system can be shown correct modularly, guaranteed using a form of type abstraction). We use Ace to support our claims of expressiveness, showing a variety of type system fragments implemented as libraries, including the entirety of the OpenCL programming language, several functional abstractions, a simple object system and domain-specific abstractions like the regular expression types described previously.

Finally, in Sec. 7, we show an example of a novel class of **editor services** that can be implemented within libraries, introducing a technique we call *active code completion*. Code completion is a common editor service (found in editors like Vim and Emacs as well as in the editor components of development environments like Eclipse) that helps programmers avoid mistakes, minimize keystrokes and explore APIs quickly by providing a menu of code snippets based on the surrounding code context. This is a useful but rather general-purpose user interface. There are a variety of tools in the literature and online that help programmers generate code snippets using alternative, more specialized user interfaces. For example, a regular expression workbench helps programmers write regular expression patterns more easily (e.g. [1, 3]). A color chooser can be considered a specialized user interface for creating an expression of type `Color`. Active code completion brings these kinds of type-specific code generation interfaces, which we call *palettes*, into the editor, and allows library providers to associate them with their own types. Clients discover and invoke palettes from the standard code completion menu, populated according to the expected type at the cursor (a protocol similar to the one we use for syntax extensions in Wyvern). When the interaction between the client and the palette is complete, the palette produces an elaboration of the type it is associated with based on the information received from the user (the reader may wish to skip ahead to Fig. 16 in Sec. 7 for an example). Using several empirical methods, including a large developer survey, we examine the expressive power of this approach and develop design criteria. Based on these criteria, we then develop an active code completion system called Graphite. Palettes are implemented as sandboxed webpages, avoiding excessive reliance on any particular editor implementation (our initial implementation is as a very simple Eclipse extension). Using Graphite, we

implement a palette for working with regular expressions and conduct a small study that demonstrates the usefulness of type-specific editor services.

Taken together, this work aims to demonstrate that actively typed mechanisms can be introduced into many different kinds of programming systems to increase expressiveness without weakening safety guarantees. We approach the problem both by building up from first principles with type-theoretic models, and by developing practical designs and providing realistic contemporary examples. In the future, we anticipate that this work will provide foundations for an actively typed programming system organized around a minimal, well-specified and formally verified core, where nearly every feature is specified, implemented and verified in a decentralized manner. We conclude with a brief historic discussion in Sec. 8.

3 Type-Specific Languages

General-purpose constructs are, semantically, quite flexible. For example, lists can be defined using a more general mechanism for defining polymorphic recursive sum types by observing that a list can either be empty, or be broken down into a *head* element and a *tail*, another list. In an ML-like language, this would be written:

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

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

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

Because lists are a common data structure, many languages include *literal notation* 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. Using terminology from Green's cognitive dimensions of notations [27], it is more *terse*, *visible* and *maps more closely* to the intuitive notion of a list. Stoy, in discussing the value of good notation, writes [63]:

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

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, sets, vectors and matrices), data formats (like XML and JSON), query languages (like regular expressions and SQL), markup languages (like HTML and Markdown) and many other types of data. For example, a language with built-in notation for HTML and SQL, supporting type-safe *splicing* via curly braces, might define:

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

as shorthand for:

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

When a specialized notation like this is not available, but 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 across language paradigms in these situations is to simply use a string representation that is parsed at run-time.

```
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 [5]. Although developers are cautioned to sanitize their input, it can be difficult to verify that this was done correctly throughout a codebase. The best way to avoid these problems today is to avoid strings and insist on structured representations, despite the inconvenient notation.

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

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

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

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

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

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

So although languages that allow providers to introduce new syntax from within libraries appear to hold promise for the reasons described above, enabling this form of extensibility is non-trivial because there is no longer a central designer making decisions about such ambiguities. In most existing related work, the burden of resolving ambiguities

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

Figure 2: Wyvern Example with Multiple TSLs

falls to clients who have the misfortune of importing conflicting extensions. For example, SugarJ [20] and other extensible languages generated by Sugar* [21] 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 because disambiguation tokens must be used. These kinds of conflicts occur in a variety of circumstances: HTML and XML, different variants of SQL, JSON literals and dictionaries, or simply different implementations ("desugarings") of the same specialized syntax (e.g. two regular expression engines). Techniques that limit the kinds of syntax extensions that can be expressed, to guarantee that ambiguities cannot occur, simply cannot express these kinds of examples as-is (e.g. [59]).

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 permits semantic flexibility – the meaning of a form like { } can differ depending on its type, so it is safe to use it for empty sets, maps and other data structures, like JSON literals. This frees these common notations from being tied to the variant of a data structure built into a language's standard library, which sometimes does not provide the exact semantics that a programmer needs (for example, Python dictionaries do not preserve key insertion order).

3.1 Wyvern

We develop our work as a variant of a new programming language being developed by our group called Wyvern [47]. To allow us to focus on the essence of our proposal, the variant of Wyvern we will describe in this thesis is simpler than the variant previously described: it is purely functional (there are no effects other than non-termination) and it does not enforce a uniform access principle for objects (fields can be accessed directly). Objects can thus be thought of as recursive labeled products with support for simple methods (functions that

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

Figure 3: Inline Generic Literal Forms

are automatically given a self-reference) for convenience. We also add recursive labeled sum types, which we call *case types*, that are quite similar to datatypes in ML. One can refer to the version of the language described in this thesis as *TSL Wyvern*. TSL Wyvern has a layout-sensitive syntax, for reasons we will discuss.

3.2 Example: Web Search

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

3.3 Inline Literals

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

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

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

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

3.4 Layout-Delimited Literals

On line 5 of Fig. 2, we see a call to a function `serve` (not shown) which has type `HTML -> Unit`. Here, `HTML` is a user-defined *case type*, having cases for each HTML tag as well as some other structures, like text nodes and sequencing. Declarations of some of these cases can be seen on lines 2-6 of Fig. 4 (note that TSL Wyvern also includes simple product types for

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

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

```
1  objtype HasTSL
2    val parser : Parser
3  objtype Parser
4    def parse(ps : ParseStream) : ParseResult
5    metadata : HasTSL = new
6      val parser = (* parser generator *)
7  casetype ParseResult
8    OK of Exp * ParseStream
9    Error of String * Location
```

```
10 casetype Exp
11   Var of ID
12   Lam of ID * Type * Exp
13   Ap of Exp * Exp
14   Tuple of Exp * Exp
15   ...
16   Spliced of ParseStream
17   metadata : HasTSL = new
18     val parser = (* quasiquotes *)
```

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

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 6-18 of Fig. 2. This literal body is layout-delimited, rather than delimited by explicit tokens as in Fig. 3, and introduced by a form of *forward reference*, written `~` ("tilde"), on the previous line. Because the forward reference occurs in a position where the expected type is `HTML`, the literal body is governed by that type's TSL. The forward reference will be replaced by the general-purpose term, of type `HTML`, generated by the TSL during typechecking. Because layout was used as a delimiter, there are no syntactic constraints on the body, unlike with inline forms (Fig. 3). For HTML, this is quite useful, as all of the inline forms impose constrains that would cause conflict with some valid HTML.

3.5 Implementing a TSL

Portions of the implementation of the TSL for `HTML` are shown on lines 8-15 of Fig. 4. A TSL is associated with a named type, forming an *active type*, using a more general mechanism for associating a pure, static value with a named type, called its *metadata*. Metadata is introduced as shown on line 8 of Fig. 4. Type metadata, in this context, is comparable to class annotations in Java or attributes in C#/F# and internalizes the practice of writing metadata using comments, so that it can be checked by the language and accessed programmatically more easily. This can be used for a variety of purposes – to associate documentation with a type, to mark types as being deprecated, and so on.

For the purposes of this work, metadata values will always be of type `HasTSL`, an object type that declares a single field, `parser`, of type `Parser`. The `Parser` type is an object type declaring a single method, `parse`, that transforms a `ParseStream` extracted from a literal body

to a Wyvern AST. An AST is a value of type `Exp`, a case type that encodes the abstract syntax of Wyvern expressions. Fig. 5 shows portions of the declarations of these types, which live in the Wyvern *prelude* (a collection of types that are automatically loaded before any other).

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

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

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

3.6 Formalization

A formal and more detailed description can be found in our paper draft.[2] In particular:

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

2. We formalize the static semantics, including the literal parsing logic, of TSL Wyvern by combining a bidirectional type system (in the style of Lovas and Pfenning [40]) with an elaboration semantics (in a style similar to Harper and Stone [31]). By distinguishing locations where an expression synthesizes a type from locations where

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

an expression is being analyzed against a known type, we can precisely state where generic literals can and cannot appear and how parsing is delegated to a TSL. The key judgements are of the form:

$$\Gamma'; \Gamma \vdash_\Sigma e \rightsquigarrow \hat{e} \Leftarrow \tau \qquad \text{and} \qquad \Gamma'; \Gamma \vdash_\Sigma e \rightsquigarrow \hat{e} \Rightarrow \tau$$

Expressions, e, elaborate to *literal-free expressions*, $\hat{e}$. This occurs simultaneously with typechecking; the first judgement captures situations where type analysis is occurring, the second type synthesis. The typing context, Γ, is standard, and the named type context, Σ, associates named types (object types and case types, but not arrow or product types) with their declarations and metadata.

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

3.7 Remaining Tasks and Timeline

This work has been accepted to ECOOP 2014, having received quite positive reviews. Final revisions on the paper text are due on May 12th.

1. There are a few presentation issues with our current formalism as written that need to be fixed. We must also write down the rules that we omitted for concision in the paper draft in a technical report, as well as provide more complete proofs of the metatheory. This might require slightly restricting the recursion in the rule for literals, so that induction can be shown to be well-founded.

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

3. We must further consider aspects of hygiene:

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

   (b) In macro systems like Scheme's, free variables in generated ASTs refer to their bindings within the macro definition, rather than where they are inserted. To support this, we have introduced the **`toast`** operator. Using it is currently explicit, so free variables that were not inside **`toast`** result in errors. We believe we can insert **`toast`** automatically using single variable (rather than whole expression) splicing so that free variables are implicitly **`toast`**ed, from within a library. We will show this.

4 Active Typechecking and Translation

In this and the following two sections, we will turn our focus to language-integrated, type-oriented mechanisms for implementing extensions to the static semantics of a simply-typed programming language, to allow providers to express finer distinctions than allowed by the general-purpose constructs (like the recursive labeled sums and products that we included in Wyvern). In the course of doing so, we will also introduce a slightly more constrained variant of TSLs to support an implementation of our mechanisms atop a fixed syntax, emphasizing that the contributions are orthogonal (combining them directly is left as future work). We consider a type-directed mechanism for supporting common elimination forms as well.

4.1 Background

Simply-typed programming languages are often described in fragments, each consisting of an indexed type constructor and associated term-level operator constructors. The simply typed lambda calculus (STLC), for example, consists of a single fragment containing the type constructor `arrow`, indexed by a pair of types, and two operator constructors: `lam`, indexed by a type, and `ap`, which we may think of as being indexed trivially. Their syntax and static semantics are shown in Fig. 6, written abstractly in a way that emphasizes this way of thinking about the type and term structure. A fragment is typically identified by the type constructor it is organized around, so the STLC can more generically called $\mathcal{L}\{\rightarrow\}$, a notational convention used by Harper [30] and others.

$$\textsc{arrow-intro}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \texttt{lam}[\tau](x.e) : \texttt{arrow}[(\tau, \tau')]} \qquad \textsc{arrow-elim}\ \frac{\Gamma \vdash e_1 : \texttt{arrow}[(\tau, \tau')] \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{ap}[()](e_1; e_2) : \tau'}$$

Figure 6: Static semantics of the $\rightarrow$ fragment (cf. *PFPL* Ch. 8.2).

Gödel's **T**, or $\mathcal{L}\{\rightarrow\ \texttt{nat}\}$, adds the `nat` fragment, specifying natural numbers and a recursor that allows one to "fold" over a natural number. The static semantics of this fragment, also written abstractly with explicit type and operator indices, is shown in Fig. 7. The language $\mathcal{L}\{\rightarrow\ \texttt{nat}\}$ is more powerful than $\mathcal{L}\{\rightarrow\}$ because the STLC can be embedded into **T** but the reverse is not true.

$$\textsc{nat-intro-1}\ \frac{}{\Gamma \vdash \texttt{z}[()]() : \texttt{nat}[()]} \qquad \textsc{nat-intro-2}\ \frac{\Gamma \vdash e : \texttt{nat}[()]}{\Gamma \vdash \texttt{s}[()](e) : \texttt{nat}[()]}$$

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

Figure 7: Static semantics of the `nat` fragment (cf. *PFPL* Ch. 9.1).

Buoyed by this increase in power, we might go on by adding fragments that specify more general constructs. For example, we could specify a fragment for labeled products as shown in Fig. 8. Here, our indices are (metatheoretic) lists, and we use syntactic shorthand rather than writing auxiliary judgements for handling empty and cons cases for concision.

$$\textsc{lprod-intro}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{ltpl}[[\ell_1, \cdots, \ell_n]](e_1;\ \cdots ; e_n) : \texttt{lprod}[[(\ell_1, \tau_1), \cdots, (\ell_n, \tau_n)]]} \qquad \textsc{lprod-elim}\ \frac{\Gamma \vdash e : \texttt{lprod}[[\cdots, (\ell, \tau), \cdots]]}{\Gamma \vdash \texttt{pr}[\ell](e) : \tau}$$

Figure 8: Static semantics of the `lprod` fragment (cf. *PFPL* Ch. 11.2).

If we consider the $\forall$ fragment, however, defining universal quantification over types (i.e. parametric polymorphism), we might take a moment to refine our understanding of when adding a new fragment is wise. In $\mathcal{L}\{\to\ \forall\}$, studied by Girard as System **F** [26] and Reynolds as the polymorphic lambda calculus [57], it is known that sums, products, and inductive and co-inductive types can all be weakly defined by a technique analagous to Church encodings. This means that we can *translate* well-typed terms of, for example, $\mathcal{L}\{\to\ \forall\ \texttt{nat} + \texttt{ltpl}\}$ to well-typed terms in $\mathcal{L}\{\to\ \forall\}$ in a manner that preserves their dynamic semantics. But Reynolds, in a remark that recalls the "Turing tarpit" of Perlis [51], reminds us that discarding the source language and programming directly with the *embedding* corresponding to this encoding may be unwise [57]:

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

An *isomorphic embedding* of a typed language fragment, i.e. one that preserves static reasoning principles, in a sense that we will make more precise as we go on, can be far more difficult to establish than a weak embedding. For example, the aforementioned embedding into $\mathcal{L}\{\to\ \forall\}$ does not preserve type disequality and some other equational reasoning principles (and we will see less subtle issues with more complex fragments soon). Establishing an isomorphic embedding that is also natural, as measured to a first approximation by the amount of boilerplate code that must be manually generated (and how likely one is to make a mistake when writing it), and that has a reasonable cost semantics is harder still. But these are the criteria one must satisfy to claim that a new language is unnecessary, because an embedding is sufficient.

4.2 Motivation

Modern general-purpose languages provide constructs, like datatypes (case types in TSL Wyvern), records, abstract types and objects, that can be used to construct satisfying isomorphic embeddings of many language fragments as libraries, occupying what their designers and users see as "sweet spots" in the design space. However, "general-purpose" and "all-purpose" remain quite distinct, and situations continue to arise in both research and practice where desirable fragments can still only be weakly defined in terms of general-purpose constructs, or an isomorphic embedding, while possible, is widely seen as too verbose, brittle or inefficient:

1. General-purpose constructs continue to evolve. There are many variants of products: labeled tuples, records (like labeled tuples, but where field ordering does not matter), records with functional record update, and record-like data structures with field delegation (of various sorts). Wyvern contains records with methods. Similarly, sum types also admit many variants (e.g. various forms of open sums). These are all either awkward or impossible to isomorphically embed into general-purpose languages that do not build in support for them. Even something as seemingly simple as a type-safe `sprintf` operator requires special support from languages like Ocaml.

2. Perhaps more interestingly, specialized type systems that enforce stronger invariants than general-purpose constructs are capable of enforcing are often developed by researchers. One need take only a brief excursion through the literature to discover language extensions that support data parallel programming [10, 19], concurrency [56], distributed programming [44], dataflow programming [41], authenticated data structures [42], database queries [49], units of measure [35], regular expressions [25] and many others.

3. Safe and natural foreign function interfaces (FFIs) require enforcing the type system of the foreign language within the calling language. For example, MLj extended Standard ML with constructs for safely and naturally interfacing with Java [9]. For any other language, including others on the JVM like Scala and the many languages that might be accessible via a native FFI, there is no way to guarantee that language-specific invariants are statically maintained, and the interface is far from natural.

These sorts of innovations are, as these references suggest, generally disseminated as *dialects* of an existing general-purpose language, constructed either as a fork, using tools like compiler generators, DSL frameworks or language workbenches, or directly within a particular compiler for the language, sometimes activated by a flag or pragma. This, as we have argued, is quite unsatisfying: a programmer can choose either a dialect supporting, for example, an innovative approach to data parallelism or one that builds in support for statically reasoning about units of measure, but there may not be an available dialect supporting both. Forming such a dialect is alarmingly non-trivial, even in the rare situation where a common framework has been used or the dialects are implemented within the same compiler, as these mechanisms do not guarantee that different combinations of individually sound dialects remain sound when combined. Metatheoretic and compiler correctness results can only be derived for the dialect *resulting* from a language composition operation, so in even a moderately diverse abstraction ecosystem, dialect providers have little choice but to leave this task to clients (providing, perhaps, some informal guidelines or partial automation). Avoiding dialect composition altogether in any large project is also difficult because interactions between components written in different dialects can lead to precisely the problems of item 3 (the *client compatibility* problem previously discussed).

These are not the sorts of problems usually faced by library providers. Well-designed languages preclude the possibility of "link-time" conflicts between libraries and ensure that the semantics of one library cannot be weakened by another by strictly enforcing abstraction barriers. For example, a module declaring an abstract type in ML can rely on any representation invariants that it internally maintains no matter which other modules are in use, so clients can assume that they will operate robustly in combination without needing to attend to burdensome "link-time" proof obligations.

Inspired by this approach, we will now introduce a mechanism for defining new type system fragments within libraries, rather than building them into the language. The semantics of operators will be delegated to type-level functions associated with these type constructors. Using techniques from typed compilation and a form of type abstraction, we will be able to guarantee type safety and (for our theoretical work) *conservativity*: that isomorphic embeddings need only be established in a closed world to hold in the open world. We will approach this mechanism from two directions: we begin in Sec. 5 by constructing a core calculus, @λ, then continue in Sec. 6 by expanding it into a full-scale language design, Ace.

5 @λ

An example of an @λ program that uses a *fragment*, $\phi_{\mathtt{nat}}$, defining primitive natural numbers as in Fig. 7 is shown in Fig. 9. An equivalent program written in core @λ, without syntactic sugar and with all type-level terms normalized, is shown in Fig. 10. The syntax of core @λ is shown in Fig. 11 and the syntactic desugarings are shown in Fig. 12. The definition of $\phi_{\mathtt{nat}}$ is shown in Fig. 13. We will discuss these in the following sections.

5.1 Overview

A *program*, ρ, consists of a series of fragment declarations for use by an external term, e. In practice, these fragment declarations would be packaged separately and imported by some mechanism that we do not include in the core calculus for simplicity (see Sec. 6).

```
using φ_nat in
  let one = s⟨z⟨⟩⟩ in
  let plus = (λx.λy.natrec x ⟨y; λp.λr.s⟨r⟩⟩) : nat → nat → nat in
  plus one one
```

Figure 9: A program that uses the natural number fragment, $\phi_{\mathtt{nat}}$, defined in Figure 13, written with the syntactic sugar defined in Figure 12.

```
using φ_nat in
  let one = intro[inr[Unit, Unit](())](intro[inl[Unit, Unit](())]()) : NAT[()] in
  let plus = (λx.λy.x · elim[()](y; λp.λr.intro[inr[Unit, Unit](())](r))
                : ARROW[(NAT[()], ARROW[(NAT[()], NAT[()])])] in
  plus · elim[()](one) · elim[()](one)
```

Figure 10: An equivalent program written in core @λ without syntactic sugar and with type-level terms normalized.

Compiling a program consists of first *kind checking* it (see below), then typechecking the external term, e, and, simultaneously, *translating* it to a term, ι, in the *typed internal language*. The dynamic behavior of an external term is determined entirely by its translation. The key judgements are the *bidirectional active typechecking and translation judgements*, relating an external term, e, to a type, τ, and an internal term, ι, called its *translation*, under *typing context* Γ and *constructor context* Φ.

$$\Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota \qquad \text{and} \qquad \Gamma \vdash_\Phi e \Leftarrow \tau \rightsquigarrow \iota$$

The typing context, Γ, maps variables to types in essentially the conventional way ([30] contains the necessary background for this section). The constructor context, Φ, tracks user-defined type constructors introduced in the "imported" fragments. This form of semantics can be seen as lifting into the language specification the first stage of a type-directed compiler like the TIL compiler for Standard ML [65] and has some parallels to the Harper-Stone semantics for Standard ML [31]. There, as in Wyvern, external terms were given meaning by elaboration from the EL to an IL having the same type system. Here the two languages have different type systems, so we call it a translation rather than an elaboration (arranging the judgement form slightly differently to emphasize this distinction, cf. above).

In @λ, the internal language (IL) provides partial functions (via the generic fixpoint operator of Plotkin's PCF), simple product types and integers for the sake of our example (and as a nod toward practicality on contemporary machines). In practice, the internal language could be any typed language with a specification for which type safety and decidability of typechecking have been satisfyingly determined. In Sec. 6, we will see how the internal language can itself be made user-definable. The internal type system serves as a "floor": guarantees that must hold for terms of any type (e.g. that out-of-bounds access to memory never occurs) must be maintained by the internal type system. User-defined constructors can enforce invariants stronger than those the internal type system maintains at particular types, however. Performance is also ultimately limited by the internal language and downstream compilation stages that we do not here consider (safe compiler extension has been discussed in previous work, e.g. [67]).

The external language has a fixed syntax with six forms: variables, let-bindings, lambda terms, type ascription and generalized introductory and elimination forms. As we will see, the generalized introductory form is given meaning by the type it is analyzed against (similar to the protocol for TSLs in Wyvern), and the elimination form is given meaning by the type of the external term being eliminated (e). This represents an internalization into the language of Gentzen's inversion principle [**?**]. Fig. 12 shows how to recover more conventional introductory and elimination forms by a purely syntactic desugaring.

what to cite for this?

$$
\begin{array}{rrcl}
\text{programs} & \rho & ::= & \mathsf{using}\ \phi\ \mathsf{in}\ e \\
\text{fragments} & \phi & ::= & \mathsf{tycon}\ \textsc{tycon}\ \mathsf{of}\ \kappa_{\text{tyidx}}\ \{\mathsf{iana}\ \tau_1; \mathsf{esyn}\ \tau_2; \mathsf{rep}\ \tau_3\} \mid \mathsf{def}\ \mathbf{t} : \kappa = \tau \mid \phi; \phi \\
\text{external terms} & e & ::= & x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \lambda x.e \mid e : \tau \\
& & \mid & \mathsf{intro}[\tau_{\text{opidx}}](e_1; \ldots; e_n) \mid e \cdot \mathsf{elim}[\tau_{\text{opidx}}](e_1; \ldots; e_n) \\
\text{type-level terms} & \tau & ::= & \mathbf{t} \mid \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \mid \mathsf{error} \\
& & \mid & \lambda \mathbf{t}{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
& & \mid & [] \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1; \tau_2; \mathbf{t}_{hd}, \mathbf{t}_{tl}, \mathbf{t}_{rec}.\tau_3) \\
& & \mid & \ell \\
& & \mid & () \mid (\tau_1, \tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau) \\
& & \mid & \mathsf{inl}[\kappa_1, \kappa_2](\tau_1) \mid \mathsf{inr}[\kappa_1, \kappa_2](\tau_2) \mid \mathsf{case}\ \tau\ \mathsf{of}\ \mathsf{inl}(\mathbf{t}) \Rightarrow \tau_1 \mid \mathsf{inr}(\mathbf{t}) \Rightarrow \tau_2 \\
& & \mid & \textsc{tycon}[\tau_{\text{tyidx}}] \mid \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{tycon}\langle \mathbf{t} \rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
& & \mid & \triangleright(\iota) \mid \blacktriangleright(\sigma) \mid \mathsf{repof}(\tau) \\
& & \mid & (\Gamma; e)? \mid \mathsf{syn}(\tau_1; \mathbf{t}_{ty}, \mathbf{t}_{trans}.\tau_2) \mid \mathsf{ana}(\tau_1; \tau_2; \mathbf{t}_{trans}.\tau_3) \\
\text{kinds} & \kappa & ::= & \kappa_1 \to \kappa_2 \mid \mathsf{list}[\kappa] \mid \mathsf{L} \mid \mathsf{Unit} \mid \kappa_1 \times \kappa_2 \mid \kappa_1 + \kappa_2 \mid \mathsf{Ty} \mid \mathsf{ITm} \mid \mathsf{ITy} \mid \mathsf{Arg} \\
\text{typing contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x \Rightarrow \tau \\
\\
\text{internal terms} & \iota & ::= & x \mid \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \iota \mid \lambda x{:}\sigma.\iota \mid \iota_1\ \iota_2 \mid n \mid \iota_1 \pm \iota_2 \mid \mathsf{if0}(\iota; \iota_1; \iota_2) \\
& & \mid & () \mid (\iota_1, \iota_2) \mid \mathsf{fst}(\iota) \mid \mathsf{snd}(\iota) \mid \triangleleft(\tau) \\
\text{internal types} & \sigma & ::= & \sigma_1 \rightharpoonup \sigma_2 \mid \mathsf{int} \mid \mathsf{unit} \mid \sigma_1 \times \sigma_2 \mid \blacktriangleleft(\tau)
\end{array}
$$

Figure 11: Syntax of Core @λ. Here, x ranges over external and internal language variables, $\mathbf{t}$ ranges over type-level variables, TYCON ranges over type constructor names, ℓ ranges over labels, n ranges over integers and $\pm$ ranges over standard binary operations over integers. The introductory form for arguments, $(\Gamma; e)?$, should only be constructed by the compiler, not by type constructor providers.

5.2 Types and Type-Level Computation

@λ supports, and makes extensive use of, simply-kinded type-level computation. Specifically, type-level terms, τ, themselves form a typed lambda calculus. The classifiers of type-level terms are called *kinds*, κ, to distinguish them from *types*, which are type-level values of kind Ty. As in Sec. 4, types are formed by applying a *type constructor* to an *index*. User-defined type constructors are declared in fragment definitions using tycon. Each constructor in the program must have a unique name, written e.g. NAT or LPROD. A type constructor must also declare an *index kind*, κ_{tyidx}. A type is introduced by applying a type constructor to an index of this kind, written $\textsc{Tycon}[\tau_{\text{tyidx}}]$. For example, the type of natural numbers is indexed trivially (i.e. by kind Unit), so it is written $\textsc{Nat}[()]$.

To permit the embedding of interesting type systems, the type-level language includes several kinds other than Ty. We lift several functional data structures to the type level: here, only unit (Unit), binary products ($\kappa_1 \times \kappa_2$), binary sums ($\kappa_1 + \kappa_2$) and lists ($\mathsf{list}[\kappa]$), in addition to labels (introduced as ℓ, possibly with a subscript, having kind L). The type constructor NAT is indexed trivially because there is only one natural number type, but LPROD would be indexed by a list of pairs of labels and types. The type constructor ARROW is included in the initial constructor context, Φ_0, and has index kind Ty $\times$ Ty. As with the internal language, in practice, one could include a richer programming language and retain the spirit of the calculus, as long as it does not introduce general recursion at the type level. For example, our desugarings add support for number literals and string literals, which require adding numbers and strings to the type-level language (and providing a means for lifting them from the type-level language to the internal language, as we will discuss).

The kind Ty also has an elimination form, $\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle \mathbf{x} \rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2$ allowing the extraction of a type index by case analysis against a contextually-available type constructor. To a first approximation, one might think of type constructors as constructors of a built-in open datatype [39], Ty, at the type-level. Like open datatypes, there is no notion of exhaustiveness so the default case is required for totality.

$$
\begin{array}{rcll}
\tau\langle e_1;\ldots;e_n\rangle & := & \mathsf{intro}[\mathsf{fst}(\tau)](e_1;\ldots;e_n) : \mathsf{snd}(\tau) & \text{assisted intro} \\
\{\ell_1{=}\, e_1,\ldots,\ell_n{=}\, e_n\} & := & \mathsf{intro}[\ell_1 :: \ldots :: \ell_n :: []](e_1;\ldots;e_n) & \text{labeled literal} \\
(e_1,...,e_n) & := & \mathsf{intro}[()](e_1;\ldots;e_n) & \text{unlabeled literal} \\
n & := & \mathsf{intro}[n]() & \text{number literal} \\
s & := & \mathsf{intro}[s]() & \text{string literal} \\
e\ e_1 & := & e \cdot \mathsf{elim}[()](e_1) & \text{application} \\
\tau\ e\ \langle e_1;\ldots;e_n\rangle & := & e \cdot \mathsf{elim}[\tau](e_1;\ldots;e_n) & \text{assisted elim} \\
e.\ell & := & e \cdot \mathsf{elim}[\ell]() & \text{projection} \\
\mathsf{case}\ e\ \{\tau_1 \Rightarrow e_1 \mid \ldots \mid \tau_n \Rightarrow e_n\} & := & e \cdot \mathsf{elim}[\tau_1 :: \ldots :: \tau_n :: []](e_1;\ldots;e_n) & \text{case analysis} \\
\tau_1 \to \tau_2 & := & \textsc{arrow}[(\tau_1,\tau_2)] & \text{arrow types}
\end{array}
$$

Figure 12: Desugaring from conventional concrete syntax to core forms. The number and string literal forms assume type-level numbers, n, and strings, s (details not shown).

$$
\begin{array}{rcl}
\phi_{\mathtt{nat}} & := & \mathsf{tycon}\ \textsc{nat}\ \mathsf{of}\ \mathsf{Unit}\ \{ \\
& & \quad \mathsf{iana}\ \lambda\mathbf{opidx}{:}\mathsf{Unit} + \mathsf{Unit}.\lambda\mathbf{tyidx}{:}\mathsf{Unit}.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{Arg}].\mathsf{case}\ \mathbf{opidx} \\
& & \qquad \mathsf{of}\ \mathsf{inl}(\_) \Rightarrow \mathbf{arity0}\ \mathbf{args}\ \triangleright(0) \\
& & \qquad \mid \mathsf{inr}(\_) \Rightarrow \mathbf{arity1}\ \mathbf{args}\ \lambda\mathbf{a}{:}\mathsf{Arg}.\mathsf{ana}(\mathbf{a};\textsc{nat}[()];\mathbf{x}.\triangleright(\triangleleft(\mathbf{x})+1)) \\
& & \quad \mathsf{esyn}\ \lambda\mathbf{opidx}{:}\mathsf{Unit}.\lambda\mathbf{tyidx}{:}\mathsf{Unit}.\lambda\mathbf{x}{:}\mathsf{ITm}.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{Arg}].\mathbf{arity2}\ \mathbf{args}\ \lambda\mathbf{a1}{:}\mathsf{Arg}.\lambda\mathbf{a2}{:}\mathsf{Arg}. \\
& & \qquad \mathsf{syn}(\mathbf{a1};\mathbf{t1},\mathbf{x1}. \\
& & \qquad\quad \mathsf{let}\ \mathbf{t2} : \mathsf{Ty} = \textsc{arrow}[(\textsc{nat}[()],\textsc{arrow}[(\mathbf{t1},\mathbf{t1})])]\ \mathsf{in} \\
& & \qquad\quad \mathsf{ana}(\mathbf{a2};\mathbf{t2};\mathbf{x2}.(\mathbf{t1}, \\
& & \qquad\qquad \triangleright((\mathsf{fix}\ f{:}\mathsf{int} \rightharpoonup \blacktriangleleft(\mathsf{repof}(\mathbf{t2}))\ \mathsf{is}\ \lambda x{:}\mathsf{int}. \\
& & \qquad\qquad\quad \mathsf{if0}(x;\triangleleft(\mathbf{x1});\triangleleft(\mathbf{x2})\ (x-1)\ (f\ (x-1))) \\
& & \qquad\qquad)\ \triangleleft(\mathbf{x}))))) \\
& & \quad \mathsf{rep}\ \lambda\mathbf{tyidx}{:}\mathsf{Unit}.\blacktriangleright(\mathsf{int}) \\
& & \}; \\
& & \mathsf{def}\ \mathbf{nat} : \mathsf{Ty} = \textsc{nat}[()]; \\
& & \mathsf{def}\ \mathbf{z} : (\mathsf{Unit}+\mathsf{Unit}) \times \mathsf{Ty} = (\mathsf{inl}[\mathsf{Unit},\mathsf{Unit}](()),\mathbf{nat}); \\
& & \mathsf{def}\ \mathbf{s} : (\mathsf{Unit}+\mathsf{Unit}) \times \mathsf{Ty} = (\mathsf{inr}[\mathsf{Unit},\mathsf{Unit}](()),\mathbf{nat}); \\
& & \mathsf{def}\ \mathbf{natrec} : \mathsf{Unit} = ()
\end{array}
$$

Figure 13: The natural number fragment, including definitions used by the assisted intro and elim desugarings, defined and shown being used above.

Type constructors are not first-class; they do not themselves have arrow kind as in some kind systems (e.g. [74]; Ch. 22 of *PFPL* describes a related system [30]). The type-level language does, however, include total functions of arrow kind, written $\kappa_1 \to \kappa_2$. Type constructor application can be wrapped in a type-level function to emulate a first-class or uncurried version of a type constructor for convenience.

Two type-level terms of kind Ty are equivalent if they apply the same constructor, identified by name, to equivalent indices. Going further, we ensure that deciding type equivalence requires only checking for syntactic equality after normalization by imposing the restriction that equivalence at a type constructor's index kind must be decidable in this way. Our treatment of equivalence in the type-level language is thus quite similar to the treatment of term-level equality using "equality types" in a language like Standard ML. Conditional branching on the basis of equality at an equality kind can be performed in the type-level language. Equivalence at arrow kind is not decidable by our criteria, so type-level functions cannot appear within type indices. This also prevents general recursion from arising at the type level. Without this restriction, a type-level function taking a type as an argument could "smuggle in" a self reference as a type index, extracting it via case analysis (continuing our analogy to open datatypes, this is closely related to the positivity condition for inductive datatypes in total functional languages like Coq).

$\boxed{\Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota}$ $\boxed{\Gamma \vdash_\Phi e \Leftarrow \tau \rightsquigarrow \iota}$

$$\textsc{ATT-FLIP}\ \frac{\Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota}{\Gamma \vdash_\Phi e \Leftarrow \tau \rightsquigarrow \iota} \qquad \textsc{ATT-VAR}\ \frac{x \Rightarrow \tau \in \Gamma}{\Gamma \vdash_\Phi x \Rightarrow \tau \rightsquigarrow x} \qquad \textsc{ATT-ASC}\ \frac{\tau \Downarrow_\Phi \tau' \qquad \Gamma \vdash_\Phi e \Leftarrow \tau' \rightsquigarrow \iota}{\Gamma \vdash_\Phi e : \tau \Rightarrow \tau' \rightsquigarrow \iota}$$

$$\textsc{ATT-LET-SYN}\ \frac{\Gamma \vdash_\Phi e_1 \Rightarrow \tau_1 \rightsquigarrow \iota_1 \qquad \Gamma, x \Rightarrow \tau_1 \vdash_\Phi e_2 \Rightarrow \tau_2 \rightsquigarrow \iota_2 \qquad \mathsf{repof}(\tau_1) \Downarrow_\Phi \blacktriangleright(\sigma_1)}{\Gamma \vdash_\Phi \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow \tau_2 \rightsquigarrow (\lambda x{:}\sigma_1.\iota_2)\ \iota_1}$$

$$\textsc{ATT-LAM-ANA}\ \frac{\Gamma, x \Rightarrow \tau_1 \vdash_\Phi e \Leftarrow \tau_2 \rightsquigarrow \iota \qquad \mathsf{repof}(\tau_1) \Downarrow_\Phi \blacktriangleright(\sigma_1)}{\Gamma \vdash_\Phi \lambda x.e \Leftarrow \textsc{ARROW}[(\tau_1, \tau_2)] \rightsquigarrow \lambda x{:}\sigma_1.\iota}$$

$$\textsc{ATT-INTRO-ANA}\ \frac{\begin{array}{c}\vdash_\Phi \mathsf{iana}(\textsc{TYCON}) = \tau_{\text{def}} \qquad \tau_{\text{def}}\ \tau_{\text{opidx}}\ \tau_{\text{tyidx}}\ ((\Gamma; e_1)? :: \ldots :: (\Gamma; e_n)? :: []) \Downarrow_\Phi \triangleright(\iota) \\ \mathsf{repof}(\textsc{TYCON}[\tau_{\text{tyidx}}]) \Downarrow_\Phi \blacktriangleright(\sigma) \qquad \Gamma \vdash_\Phi \iota : \sigma\end{array}}{\Gamma \vdash_\Phi \mathsf{intro}[\tau_{\text{opidx}}](e_1; \ldots; e_n) \Leftarrow \textsc{TYCON}[\tau_{\text{tyidx}}] \rightsquigarrow \iota}$$

$$\textsc{ATT-ELIM-SYN}\ \frac{\begin{array}{c}\Gamma \vdash_\Phi e \Rightarrow \textsc{TYCON}[\tau_{\text{tyidx}}] \rightsquigarrow \iota \qquad \vdash_\Phi \mathsf{esyn}(\textsc{TYCON}) = \tau_{\text{def}} \\ \tau_{\text{def}}\ \tau_{\text{opidx}}\ \tau_{\text{tyidx}}\ \triangleright(\iota)\ ((\Gamma; e)? :: (\Gamma; e_1)? :: \ldots :: (\Gamma; e_n)? :: []) \Downarrow_\Phi (\tau, \triangleright(\iota')) \\ \mathsf{repof}(\tau) \Downarrow_\Phi \blacktriangleright(\sigma) \qquad \Gamma \vdash_\Phi \iota' : \sigma\end{array}}{\Gamma \vdash_\Phi e \cdot \mathsf{elim}[\tau_{\text{opidx}}](e_1; \ldots; e_n) \Rightarrow \tau \rightsquigarrow \iota'}$$

Figure 14: The bidirectional active typechecking and translation judgements.

5.3 Bidirectional Active Typechecking and Translation

The rules for bidirectional active typechecking and translation are shown in Fig. 14.

1. The rule ATT-FLIP is standard in bidirectional type systems (without subtyping): if a term synthesizes a type, it can be analyzed against that type; the translation is unaffected by this.

2. The rule ATT-VAR says that variables synthesize the type they have in the typing context and translate to variables in the internal language.

3. The rule ATT-ASC says that a term ascribed by a type synthesizes that type if the term can be analyzed against that type, after it has been normalized. The normalization judgement for type-level terms is written $\tau \Downarrow_\Phi \tau'$. The kinding rules (not shown here) guarantee that normalization of type-level terms cannot go wrong (we will refine what precisely this means later).

4. The rule ATT-LET-SYN first synthesizes a type for the bound value, then adds this binding to the context and synthesizes a type for the term the binding is scoped over. The translation is to a function application, in the conventional manner.

   To do so, however, we must also translate the type itself so that an appropriate type annotation for the function argument can be emitted. Every type has an internal type associated with it called its *representation type*. The type-level operator $\mathsf{repof}(\tau)$ evaluates to a *quoted internal type*, $\blacktriangleright(\sigma)$ (of kind ITy), where σ is the representation type of τ. Type constructors define the representation type for every possible index by providing a type-level function called the *representation schema*, written after the keyword rep. The normalization rule showing this is given in Fig. 15. In our example, the representation schema is simple: natural numbers translate to integers. We will see an example where this is less trivial later. Note that quoted internal types support splicing using the $\blacktriangleleft(\tau)$ form. These forms are eliminated during normalization.

$\boxed{\tau \Downarrow_\Phi \tau'}$

$$\frac{\tau \Downarrow_\Phi \text{TYCON}[\tau_{\text{idx}}] \qquad \vdash_\Phi \mathsf{rep}(\text{TYCON}) = \tau_{\text{rep}} \qquad \tau_{\text{rep}}\ \tau_{\text{idx}} \Downarrow_\Phi \blacktriangleright(\sigma)}{\mathsf{repof}(\tau) \Downarrow_\Phi \blacktriangleright(\sigma)}\ \text{REPOF}$$

$$\frac{\tau \Downarrow_\Phi (\Gamma; e)? \qquad \Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota \qquad [\tau/\mathbf{t}_{ty}, \triangleright(\iota)/\mathbf{t}_{trans}]\tau_2 \Downarrow_\Phi \tau_2'}{\mathsf{syn}(\tau_1; \mathbf{t}_{ty}, \mathbf{t}_{trans}.\tau_2) \Downarrow_\Phi \tau_2'}\ \text{SYN}$$

$$\frac{\tau_1 \Downarrow_\Phi (\Gamma; e)? \qquad \tau_2 \Downarrow_\Phi \tau_2' \qquad \Gamma \vdash_\Phi e \Leftarrow \tau_2' \rightsquigarrow \iota \qquad [\triangleright(\iota)/\mathbf{t}_{trans}]\tau_3 \Downarrow_\Phi \tau_3'}{\mathsf{ana}(\tau_1; \tau_2; \mathbf{t}_{trans}.\tau_3) \Downarrow_\Phi \tau_3'}\ \text{ANA}$$

Figure 15: Normalization semantics for the type-level language. Missing rules (including error propagation rules and normalization of quoted internal terms and types) are unsurprising and will be given later.

5. The rule ATT-LAM says that lambda terms can only be analyzed against arrow types and translate to lambda terms in the internal language.

6. The rule ATT-INTRO-ANA says that generalized introductory forms can only be analyzed against a type. That type constructor is consulted to extract its *introductory operator definition*, written (e.g. in Figure 13) after the iana keyword as a type-level function. This function is given the provided operator index, the type's type index and a list encapsulating the operator's *arguments*. An argument is reified as a type-level term of the form $(\Gamma, e)?$ and has kind Arg. Note that only the compiler should construct such a term (in practice, this could be enforced purely syntactically, so we do not enforce this judgmentally here). The operator definition is responsible for producing a translation, or evaluating to error if this is not possible. A translation is simply a quoted internal term, written $\triangleright(\iota)$, with kind ITm. Like quoted internal types, quoted internal terms support an unquote form, written $\triangleleft(\tau)$, which is normalized away (in a capture-avoiding manner, not shown).

   Natural numbers have two introductory forms, each indexed trivially. Because there is only one generalized introductory form, we instead index the operator by a simple sum kind with two trivial cases, Unit + Unit. The first case corresponds to the zero operator and the second to the successor. In the introductory operator definition, we see in the first case that the translation 0 is produced after checking that no arguments were provided (using a simple helper function, **arity0**, not shown). In the second case, we first extract the single argument (using the helper function **arity1**, not shown). We then analyze it against the type NAT[()] using one of the elimination forms for arguments, $\mathsf{ana}(\tau_1; \tau_2; \mathbf{t}_{trans}.\tau_3)$. If it succeeds, we construct the appropriate translation based on the translation of the argument. The normalization rule for successful analysis, ANA, is shown in Figure 15.

   Note that we do not show here the error propagation rules. We need two additional judgements for this: the judgement $\Gamma \vdash_\Phi e$ `error` says that e has a type error, and the judgement τ $\texttt{err}_\Phi$ says that normalizing τ produced an error term. Appropriate error propagation rules need to be written down.

   If analysis of the introductory form succeeds, the translation is checked for *representational consistency*: that the produced translation has the type indicated by the type constructor's representation schema. This is expressed by essentially a standard typechecking judgement for the internal language, written $\Gamma \vdash_\Phi \iota : \sigma$ (not shown). Note that the typing context will need to be translated to a corresponding typing context for the internal language. We will discuss this further below.

Notice that the assisted introductory form in Figure 12 allows us to define type-level variables $\mathbf{z}$ and $\mathbf{s}$ that simultaneously provide an operator index and type ascription, simulating introductory forms that synthesize a type without requiring support for such in the semantics (we will discuss a more expressive variant of the system in Sec. 6 where ascriptions can also be type constructors, not just types, permitting a form of synthetic introductory form).

Note also that lambda is technically the introductory form for the ARROW type constructor, but because it requires manipulating the context, it must be built in to @λ. We do not currently support type system fragments that require adding or manipulating existing contexts.

7. The rule ATT-ELIM-SYN says that elimination forms synthesize a type. This is done again by consulting a type-level function, called the *eliminatory operator definition*, associated with a type constructor, here the type constructor of the type recursively synthesized for the primary operand, e. This definition is invoked with the type index, the operator index, the translation of the primary operand and a list of arguments. Because elimination forms synthesize a type, this definition must produce both a type and a translation. Representational consistency is then checked.

   In our example, the elimination form for natural numbers is the recursor (shown in Fig. 7). Our definition of it first checks that exactly two arguments were provided (not including the primary operand), then synthesizes a type and extracts a translation from the first argument using the form $\mathsf{syn}(\tau_1; \mathbf{t}_{ty}, \mathbf{t}_{trans}.\tau_2)$ (Fig. 15). The second argument is analyzed against an appropriate arrow type to support variable binding. If successful, the type $\mathbf{t1}$ (the type of the base case) is synthesized and a translation implementing the dynamic semantics of the recursor by a fixpoint computation is produced.

   The assisted elimination form shown in Fig. 12 provides a means to avoid providing an operator index explicitly. Here, we simply define **natrec** as the trivial value to avoid needing to write it explicitly (and for clarity).

   Note that the elimination form for the ARROW type constructor can be defined in this manner (because it does not require binding variables), so application is simply syntactic sugar, rather than a built-in operator in the external language.

5.4 Representational Consistency Implies Type Safety

A key invariant that our operator definitions maintain is that well-typed external terms of type NAT[()] always translate to well-typed internal terms of internal type int, the representation type of NAT[()]. Verifying this for the zero case is simple. For the translation produced by the successor case to be of internal type int requires that $\triangleleft(\mathbf{x})$ be of internal type int. Because it is the result of analyzing an argument against NAT[()], and the only other introductory form is the zero case, this holds inductively. We will discuss the recursor later, but it also maintains this invariant inductively.

This invariant would not hold if, for example, we allowed the type and translation:

$$(\text{NAT}[()], \triangleright((0, ())))$$

In this case, there would be two different internal types, int and $\mathsf{int} \times \mathsf{unit}$, associated with a single external type, NAT[()]. This would make it impossible to reason *compositionally* about the translation of an external term of type NAT[()], so our implementation of the successor would produce ill-typed translations in some cases but not others. Similarly, we wouldn't be able to write functions over all natural numbers because there would not be a well-typed translation to give to such a function. This violates type safety: there are now well-typed external terms, according to the active typechecking and translation judgement, for which evaluating the corresponding translation would "go wrong".

To reason compositionally about the semantics of well-typed external terms when they are given meaning by translation to a typed internal language, the system must maintain the following property: for every type, τ, there must exist an internal type, σ, called its *representation type*, such that the translation of every external term of type τ has internal type σ. This principle of *representational consistency* arises essentially as a strengthening of the inductive hypothesis necessary to prove that all well-typed external terms translate to well-typed internal terms because λ and successor are defined compositionally. It is closely related to the concept of *type-preserving compilation* developed by Morrisett et al. for the TIL compiler for SML [65], here lifted into the language. Our judgements check this extension correctness property directly by typechecking each translation produced by a user extension (the translations of, for example, lambda terms will be inductively representationally consistent, so no additional check is needed).

Translational internal terms can contain translational internal types. We see this in the definition of the recursor on natural numbers. The type assignment is the arbitrary type, $\mathbf{t2}$. We cannot know what the representation type of $\mathbf{t2}$ is, so we refer to it abstractly using $\mathsf{repof}(\mathbf{t2})$. Luckily, the proof of representational consistency of this operator is parametric over the representation type of $\mathbf{t2}$.

5.5 Representation Independence Implies Conservativity

In fact, we will demand this sort of *representation independence* at *all* types constructed by a type constructor other than the one an operator constructor is associated with, as this will lead us to the concept of conservativity. If we closed the constructor context at this point (let us call this language $@\lambda[\Phi_{\mathtt{nat}}]$) our definition of natural numbers can be shown to admit an isomorphic embedding of the semantics of Gödel's $\mathbf{T}$, i.e. $\mathcal{L}\{\rightarrow\ \mathtt{nat}\}$. That is, there is an isomorphism (a mapping with an inverse) between every type in $\mathbf{T}$ and a type in $@\lambda[\Phi_{\mathtt{nat}}]$. Indeed, it is an entirely obvious one, given how directly we constructed $@\lambda[\Phi_{\mathtt{nat}}]$. The statics are preserved, and we can show by a bisimulation style argument that the translation produced by $@\lambda[\Phi_{\mathtt{nat}}]$ implements the dynamic semantics of Gödel's $\mathbf{T}$. We must leave full details to later, but the techniques are standard for reasoning about compiler correctness. All we have done is lifted them into the language.

An important lemma is that every external term of type $\textsc{Nat}[()]$ is equivalent to one of the introductory forms we defined for the natural number type (corresponding to the canonical forms lemma of $\mathbf{T}$, and necessary to construct a well-founded induction principle for natural numbers). The proof of this lemma is quite straightforward for $@\lambda[\Phi_{\mathbf{T}}]$, since the only other values in the language are functions and the elimination form does not introduce any values. However, it is also quite precarious because it relies on exhaustive induction over the available type constructors. As soon as we define and use another type constructor, this property may no longer be *conserved*. There are operator definitions that are representationally consistent, but still cause problems because they violate important invariants maintained by the existing introductory forms, and thus qualify as a new introductory form. For example, an elimination form for some other type that synthesized the type $\textsc{Nat}[()]$ paired with the translation $\triangleright(-1)$ violates the internal invariant that the translation of a natural number is *non-negative*.

If we assume an abstraction barrier between type constructors[3] *must exist, we do not need a fragment specification to see that our hypothetical operation is violating it*. We can prevent problems by using a technique closely related to the one used to enforce abstraction barriers between modules in ML – holding the representation type abstract. If no other type constructor can know that the representation type of $\textsc{Nat}[()]$ is int, then our hypothetical definition is incorrect independent of the specification of $\textsc{Nat}$.

Our calculus as presented does not maintain this invariant for simplicity. We have drafted a unidirectional (i.e. type assignment) version of this calculus where this invariant

[3] A fragment consisting of finitely many type constructors can be easily turned into a fragment containing just one by adding a sum type to the index and case analyzing against it whenever a distinction is necessary.

is maintained using a syntactic type abstraction technique similar to one developed by Grossman et. al [28] to check that the translation produced by an operator depends only on the details of the representation of the type constructor it is associated with, and not any other.[4] This then leads, as we will show in this thesis, to a *conservativity theorem*: that any properties of the value of a translation of an external term of some type that are maintained by the operators associated with that type's constructor are conserved in every larger constructor context (a mouthful to be sure, but quite a powerful theorem).

5.6 Extension Correctness

Because we are restricting ourselves to a simply-typed, simply-kinded calculus, problems with representational consistency and representation independence are detected by the system only when typechecking a program where an incorrect operator constructor like ***n1*** is actually used. This is sufficient to derive type safety and conservativity theorems, but clients of an extension might occasionally face compile-time errors originating not in their own code, not in an extension they are using. This would be analagous to a compiler-internal assertion failing or a translation validation technique, like that used in some compilers, finding a problem [54]. This cannot cause incorrect translations to be produced, but when validation fails, it is an annoyance because the client has to file a bug report to the provider of the compiler/extension. To avoid these situations from arising, providers can prove these properties extrinsically (made possible by the fact that we provide a detailed specification of the mechanism and state precisely the necessary extrinsic proof obligations) and/or extensively test their extensions. This is precisely the current situation for nearly every modern programming language implementation.

The only major exceptions to this are fully-verified compilers like CompCert (for a variant of C) [37] and CakeML (for a variant of Standard ML) [36], which are implemented in dependently-typed languages (Coq and HOL4, respectively). Adding dependent kinds to the type-level language thus represents an analagous approach – proofs of these properties, as well as proofs that the operator adequately implements a fragment, could be given directly alongside operator definitions, rather than extrinsically. To ensure that the typechecker is efficiently executable, we could take the approach CompCert takes, which is to rely on extraction to a simply-typed language (Ocaml) where proofs have been erased. An implementation of @λ would be a suitable extraction target. We plan to outline these approaches, but a full development of this line of research will be left for future work, as the scope of this thesis is limited to safety and expressiveness, rather than the broader issue of verifying extension correctness. An appropriate analogy would be methods that prevent array out-of-bounds issues by run-time checks, rather than by a static analysis. Here, run-time is "normalization time" of type-level terms, which occurs statically with respect to the external language.

5.7 Expressiveness

Although the example of a natural number type implemented as an integer could also have been shown using an abstract type, there are more interesting examples that cannot be implemented in such a way. For example, our paper draft[5] shows an example implementing n-ary tuples in a variant of @λ (in terms of nested binary tuples in the IL). Were we to program in a variant of the IL with support for abstract types, we could not introduce the logic of tuples or labeled tuples as a single module. At best, we can implement any *particular* tuple or labeled tuple as an abstract type (at the cost of substantial manual labor). More generally, modules can only expose finitely many operations, and these operations are functions, so one cannot introduce operators that work with indices statically to introduce more interesting typechecking logic. We will show an example of

[4] https://github.com/cyrus-/papers/blob/master/att-icfp14/att-icfp14.pdf
[5] https://github.com/cyrus-/papers/blob/master/ace-oopsla14/ace-oopsla14.pdf

a `sprintf`-like operator later that makes more obvious use of this, as well as examples of more interesting record and object types, sum types and a simplified variant of regular expression types starting in the next section.

It is encouraging that there are clear parallels between module systems with abstract types and type constructors with "abstract" representation schemas. Note also that a module system must sit atop a core type system, which is what we are defining here. There are some subtleties related to combining generative type abstraction and type-level computation that need to be considered to introduce both into the same language [75] (we will discuss this further later). As we suggested earlier, general-purpose constructs should be used when possible, as they are easier to reason about than arbitrary operators. The purpose of this mechanism is to make it possible to go beyond these when needed, and to make the built-in "conveniences" of modern languages less arbitrary (indeed, this theme threads through the entire thesis).

Not all type systems can be implemented in this way. In particular, only type systems for which the typing judgement looks essentially like the typing judgement of the simply-typed lambda calculus can be implemented in @λ, because we do not allow constructors to introduce new static contexts. We plan to discuss this and other limitations in more detail, and sketch specific directions for future work, later.

5.8 Remaining Tasks and Timeline

The core of the mechanism has been figured out, but the details remain to be written down. In particular, the following tasks remain to be completed in the next 2-3 months.

1. We currently have three slightly different variants of the formal system: the one presented here (a simplified bidirectional variant not submitted anywhere), the one presented in the submitted Ace paper (a bidirectional variant with some additional support for synthesizing types for introductory forms given a type constructor ascription) and the one presented in the aborted ICFP paper (a unidirectional version with support for guaranteeing representation independence, as described above). We need to unify these into a single system or, if the committee feels it makes sense, to present both unidirectional and bidirectional variants.

2. I need to more completely write down the full rules and metatheory and prove the type safety and conservativity theorems (which are the main pieces of metatheory) along with the key lemmas. I will also sketch out a proof of termination of the normalization judgement, which, together with a lemma for decidability of internal typing will will then imply decidability of typechecking.

3. I need to write down the proof obligations for showing extension correctness, based on the guidelines described above, and give proofs that the examples of natural numbers and records are correct.

4. I also need to write down the conditions for proving that an encoding is isomorphic to a fragment specification and show how to prove such a theorem for the natural number example. This will motivate our more detailed discussion on conservativity.

5. We have not yet described how to maintain hygiene. Based on the work in Sec. 3 on enforcing hygiene for type-specific language definitions, we plan to describe how hygiene can be maintained for operator definitions as well.

Listing 1 [`listing1.py`] An Ace compilation script.

```
from ace import ty
from examples.py import fn, string
from examples.fp import record
from examples.oo import proto
from examples.num import decimal
from examples.regex import string_in

print "Hello, compile-time world!"

@ty
def A(): record[amount : decimal[2]]

@ty
def C(): record[
  name : string,
  account_num : string_in[r'\d{10}'],
  routing_num : string_in[r'\d{2}-\d{4}/\d{4}']
]

@ty
def Transfer(): proto[A, C]

@fn
def log_transfer(t):
  """Logs a transfer to the console."""
  {t : Transfer}
  print "Transferring %s to %s." % (
    [string](t.amount), t.name)

@fn
def __toplevel__():
  print "Hello, run-time world!"
  common = {name: "Annie Ace",
            account_num: "0000000001",
            routing_num: "00-0000/0001"} is C
  t1 = ({amount: 5.50}, common) is Transfer
  t2 = ({amount: 15.00}, common) is Transfer
  log_transfer(t1)
  log_transfer(t2)

print "Goodbye, compile-time world!"
```

6 Ace

The work in this thesis is motivated by the intuition that it is better to ask programmers to import a library than to adopt a new programming system. Thus far, however, all of our mechanisms have been introduced in the context of a new programming system. While this is useful to describe the essence of our approach, it also creates a "chicken-and-egg" problem. In this and Sec. 7, we will discuss how we can implement mechanisms similar in essence to those described above in existing general-purpose languages. This section focuses on developing Ace, an extensible bidirectional type system based on the work in the previous section embedded inside Python, a dynamically-typed language. By doing so, we inherit many of the tools in Python's existing ecosystem, as we will describe below. Python serves as the type-level language, rather than a simply-typed lambda calculus as in @λ. Unlike in @λ, the internal language can be user defined, as can the semantics of functions, and the syntax trees of arguments can be inspected. Type constructors are identified with classes that implement the interface `ace.Type`. Other than these differences, the calculus and Ace contain essentially the same mechanism. Indeed, a calculus very similar to the one above was submitted alongside the paper on Ace (to OOPSLA 2014, see footnote above).

6.1 Ace: Language Design and Usage

Listing 1 shows that the top-level of every Ace file is a *compilation script* written directly in Python. Ace requires no modifications to the Python language (version 2.6+ or 3.0+) nor features specific to the primary implementation, CPython (so Ace supports alternative implementations like Jython and PyPy). This choice pays immediate dividends on the first six lines: Ace's import mechanism is Python's import mechanism, so Python's build tools (e.g. `pip`) and package repostories (e.g. `PyPI`) are directly available for distributing Ace-based libraries, including those defining new type systems.

6.2 Types

Types are constructed programmatically during execution of the compilation script. This stands in contrast to many contemporary statically-typed languages, where types (e.g. datatypes, classes, structs) can only be declared. Put another way, Ace supports *type-level computation* and Python is its type-level language. In our example, we see several types being constructed:

1. On lines 10-11, we construct a named functional record type with a single (immutable) field named `amount` with type `decimal[2]`, which classifies decimal numbers with two decimal places. `record` and `decimal` are *indexed type constructors*. We will provide more details in Sec. 6.7, but note that syntactically, `record` overloads subscripting and borrows Python's syntax for array slices (e.g. `a[min:max]`) to approximate conventional functional notation for type annotations.

2. On lines 13-18, we construct another named record type. The field `name` has type `string`, defined in `examples.py`, while the fields `account_num` and `routing_num` have more interesting types, classifying strings guaranteed statically to be in a regular language specified statically by a regular expression pattern (written, to avoid needing to escape backslashes, using Python's *raw string literals*). The typechecking rules are based on [25] (work that was done in collaboration with my summer student, Nathan Fulton). To our knowledge, this type system has not previously been implemented. We include it to emphasize that we aim to support specialized type systems, not just general purpose constructs like records and strings.

3. On lines 20-21, we construct a named *prototypic object type* [38]. The type `Transfer` classifies terms consisting of a *fore* of type `A` and a *prototype* of type `C`. If a field cannot be found in the fore, the type system will delegate (here, statically) to the prototype. This makes it easy to share the values of the fields of a common prototype amongst many fores, here to allow us to describe multiple transfers differing only in amount.

6.3 Typed Functions

Typed functions implement the run-time behavior and are distinguished by the presence of a decorator specifying their *base semantics* (here `py` from `examples.py`; Sec. 6.7). These functions are, however, still written using Python's syntax, a choice that is again valuable for extrinsic reasons: users of Ace can use a variety of tools designed to work with Python source code without modification, including code highlighters (like the one used in generating this paper), editor plugins, style checkers and documentation generators. We will see several examples of how, with a bit of cleverness, Python's syntax can be repurposed within these functions to support a variety of static and dynamic semantics that differ from Python's. Ace leverages the `inspect` and `ast` modules in the Python standard library to extract abstract syntax trees for functions annotated in this way [6].

On lines 23-28, we see the function `log_transfer`. We follow Python conventions by starting with a documentation string that informally specifies the behavior of the function. Before moving into the body, however, we also write a *type signature* (line 26) stating that

the type of `t` is `Transfer`, the prototypic object type described above. As a result, we can assume on line 28 that `t.amount` and `t.name` have types `decimal[2]` and `string`, respectively. We will return to the details of `print` and the form `[string](t.amount)`, which performs an explicit conversion, in Sec. 6.7.

Note that the argument signature repurposes Python's syntax for dictionary literals to approximate conventional notation for type annotations. In version 3.0 of Python, syntax for annotating arguments with arbitrary values directly was introduced [4]:

```
@fn
def log_transfer(t : Transfer):
  """Logs a transfer to the console."""
  print ...
```

These annotations were initially intended to serve only as documentation (suggesting that users of dynamically typed languages see potential in a typing discipline, if not a static one), but they were also made available as metadata for use by unspecified future libraries. Ace supports both notations when the compilation script is run using Python 3.*x*, but we use the more universally available notation here in view of our extrinsic goals: the Python 2.x series remains the most widely used by a large margin as of this writing.

6.4 Bidirectional Typechecking of Introductory Forms

On lines 30-39, we define the function `__toplevel__` (the base will consider this a special name for the purposes of compilation, discussed in Sec. 6.5). After printing a run-time greeting, we introduce a value of type `C` on lines 33-35. Recall that `C` is a record type, so we provide the names of the fields (here, without quotes because we are no longer in the type-level language) and values of the appropriate type using the syntactic form normally used for dictionary literals. We specify which record type the literal should be analyzed against by giving an *ascription*. This again repurposes existing syntax: here the `is` operator, which normally performs reference equality. If the right side of the operator is a static type, then it operates instead as an ascription form. We see the same form used with Python's tuple syntax to introduce terms of type `Transfer` by providing the fore and prototype.

The string, number and dictionary literal forms inside the outermost forms on lines 26-30 are also introductory, but do not have an ascription. This is because the outermost ascriptions completely determine which types they will need to have. As we will discuss in more detail shortly, Ace is built around a *bidirectional type system* that distinguishes locations where an expression must *synthesize* a type (e.g. to the right of bindings) from those where it can be *analyzed* against a known type (e.g. as an argument to a function with known type) [40]. Unascribed literal forms must ultimately be analyzed against a type.

If an unascribed literal is used in a synthetic location, the base can, however, specify a "default ascription". For example, if we had not specified the literal ascription, the base would cause the dictionary literal to be analyzed against the type `dyn`, covering dynamically classified Python values. This would still lead to a type error because keys are then treated as expressions, as in Python, rather than field names, and the identifiers shown have not been bound in the top-level context (emphasizing that "dynamically-typed languages" can still have a static semantics, even if it only involves checking that top-level variables are bound).

An ascription on a literal form can also be a type constructor instead of a type. For example, we might write `[1, 2] is matrix` instead of `[1, 2] is matrix[i32]` when using a base for which the default ascription for number literals is `i32`. Because the arguments synthesize the appropriate type, the type constructor can synthesize an appropriate index based on the types of the subexpressions. If we wanted a matrix of `dyn`s, then we would have to write either `[1, 2] is matrix[f32]`, ascribe each inner literal so that it synthesizes the `f32` type, `[1 is f32, 2 is f32] is matrix`, or construct a base with a different default ascription.

Listing 2 Compiling `listing1.py` using `acec`. Both steps can be performed at once by writing `ace listing1.py` (line 3 will not be printed with this command).

```
1  % acec listing1.py
2  Hello, compile-time world!
3  Goodbye, compile-time world!
4  [acec] _listing1.py successfully generated.
5  % python _listing1.py
6  Hello, run-time world!
7  Transferring 5.50 to Annie Ace.
8  Transferring 15.00 to Annie Ace.
```

Listing 3 [`_listing1.py`] The file generated in Listing 2.

```
1  import examples.num.runtime as __ace_0
2
3  def log_transfer(t):
4    print ("Transferring %s to %s." % (
5      __ace_0.decimal_to_str(t[0],2), t[1][0]))
6
7  print "Hello, run-time world!"
8  common = ("Annie Ace", "0000000001", "00-0000/0000")
9  t1 = ((5,50), common)
10 t2 = ((15,0), common)
11 log_transfer(t1)
12 log_transfer(t2)
```

6.5 External Compilation

To typecheck, translate and execute the code in Listing 1, we have two choices: do so externally at the shell, or perform compilation interactively (or implicitly) from within Python. We will begin with the former. Interactive and implicit compilation are implemented but we choose here to focus on the core mechanism.

Listing 2 shows how to invoke the `acec` compiler at the shell to typecheck and translate `listing1.py`, resulting in a file named `_listing1.py`. This is then sent to the Python interpreter for execution. Note that the `print` statements at the top-level of the compilation script were evaluated during compilation only. These two steps can be combined by running `ace listing1.py` (the intermediate file is not generated unless explicitly requested in this case).

The Ace compiler is itself a Python library, and `acec` is a simple Python script that invokes it, operating in two steps:

1. It evaluates the compilation script to completion.
2. For any top-level bindings that are Ace functions in the final environment (instances of `ace.TypedFn`, as we will discuss), it initiates active type-checking and translation (Sec. 6.7). If no type errors are discovered, the translations are collected (obeying order dependencies) and emitted, with file extension(s) determined by the target(s) in use, discussed further in Sec. 6.7. Here our target is the default target associated with the base `py`, which emits Python 2.6 files. If a type error is discovered, no file is emitted and the error is displayed on the console.

In our example, there are no type errors, so the file `_listing1.py`, shown in Listing 3, is generated. This file is meant only to be executed. The invariants necessary to ensure that execution does not "go wrong" were checked statically and entities having no bearing on execution, like field names and types themselves, were erased. Notice that:

1. The base recognized the function name `__toplevel__` as special, placing the translation of its body at the top level of the file. Ace does not have any special function names itself; this is a feature of the user-defined base.
2. Records with two or more fields translated to tuples of their values (e.g. `common` on line 8). If there was only a single field, like the terms of type `A` inside `t1` and `t2`, the value was passed around unadorned.

Listing 4 [`listing4.py`] Lines 8-12 each have type errors.

```
1  from listing1 import fn, A, C, log_transfer
2  from datetime import date
3  @fn
4  def pay_oopsie(a):
5    {a : A}
6    print "Checking date..."
7    if date.today().day == 1:
8      common = {nome: "Oopsie Daisy",
9                account_num: None,
10               routing_num: "0-0000-0002"} is C
11     log_transfer((common, a))
12     a.amount += 1000
13 print str(date.today().day)
```

Listing 5 Compiling `listing4.py` using `acec` catches the errors statically (compilation stops at first error).

```
1  % acec listing4.py
2  2
3  [acec] TypeError in listing4.py (line 8, col 15):
4    [record listing1.A]
5      Invalid field name: nome
6      Valid field names: name, account_num, routing_num
```

3. Decimals translated to pairs of integers. Conversion to a string happened via a helper function defined in a "runtime" package imported with an internal name, `__ace_0`, to avoid naming conflicts.

4. Terms of type `string_in[r"..."]` translated to strings. Checks for membership in the specified regular language were performed entirely statically by the type system.[6]

5. Prototypic objects are represented as pairs consisting of the fore and the prototype. Dispatch to the appropriate record based on the field name is static (line 5).

6.6 Type Errors

Listing 4 shows an example of code containing several type errors. Indeed, lines 8-12 each contain a type error. If analagous code were written in Python itself, these could only be found if the code was executed on the first day of the month (and, depending on the implementation, not all of the issues would immediately result in run-time exceptions, possibly leading to quite subtle problems; for example, the unexpected use of `None`[7]). Static type checking allows us to find these errors during compilation. Listing 5 shows the result of attempting to compile this code. The compilation script completes (so that functions can refer to each other mutually recursively), then the typed functions in the top-level environment are typechecked. The typechecker raises an exception statically at the first error, and `acec` prints it to the console as shown.

6.7 Active Typechecking and Translation

Enabling the addition of new forms to a language in an open tool ecosystem is difficult. Indeed, it is the canonical example of the long-studied *expression problem* [72]. Although a number of approaches have been developed, we argue that many of them are overly permissive, making it difficult to reason compositionally about metatheoretic issues (e.g. type safety) and avoid ambiguities when extensions are combined (see Sec. 1.3.1). Ace's term forms, as we have seen, are fixed by Python's grammar, so Ace sidesteps the

[6] Note that there are situations (e.g. if a string is read in from the console) that would necessitate an initial run-time check, but no further checks in downstream functions would be necessary.

[7] The notation `+T` constructs a `None`-able option type (not shown).

Listing 6 [`listing6.py`] The example detailed in Sec. 6.7.

```
1  from listing1 import py, A, record
2  @py
3  def example(a):
4    {a : A}
5    x = a.amount
6    x = x + x
7    return {
8      y: {amount: x} is A,
9      remark: "Creating an anonymous record"} is record
```

expression problem almost entirely (and thus inherits broad tool support, as described earlier). The key insight is that leaving the forms fixed does not mean the semantics must also be fixed. Instead of taking a syntax-directed view of extensibility, we take a type-directed view: users cannot add new forms but they can add new type constructors, which we empower with more control over the semantics of existing forms (e.g. literals, as discussed above, but also nearly every other form, as we will now discuss) in a controlled manner. The key features that characterize active typechecking and translation are:

- a *delegation protocol* that delegates responsibility for typechecking and translating a term to:
  - a type or type constructor extracted from a subterm
  - a per-function base semantics (or simply *base*) for forms for which this is not possible (variables, literals without ascriptions and most statements)
- a mechanism for allowing the user to define new types, type constructors and bases from within the language (in particular, *from within the type-level language*) rather than fixing them ahead of time

To see how this works, let us trace through how Ace processes the example in Listing 6.

6.8 Base Decorators

Decorators in Python (line 2) are syntactic sugar: we could equivalently have omitted line 3 and inserted the top-level statement `example = py(example)` on the line after the function definition. Note, however, that `py` is not a function but an instance of a class, `examples.py.PyBase`, that inherits from `ace.Base`. It can be used as a decorator because `ace.Base` overloads the call operator, shown in Listing 7. The `_process` function (not shown) operates as follows:

1. The Python standard library is used to extract an abstract syntax tree (`AST`) from the function.
2. The closure of the function, together with the globals in the module it is defined in, are extracted to reify its static environment.
3. The argument annotations are processed. If provided in the body, as in our example, `_process` checks that the argument names are spelled correctly and evaluates the type-level expressions (e.g. `A` above) in the static environment to produce a mapping of argument names to types, called the argument signature. If Python 3's annotations were used, this is not necessary.

By the end of Listing 6, `example` is thus an instance of `ace.TypedFn`. It could have been constructed directly by calling the constructor on line 2 of Listing 7 (this would be inconvenient, but could be useful for metaprogramming).

Listing 7 A portion of the `ace` core showing how a base can be used as a decorator to construct a typed function.

```
1  class TypedFn(object):
2    def __init__(self, base, ast, static_env, arg_sig):
3      # ...
4    # ...
5  class Base(object):
6    def __call__(self, f):
7      (ast, static_env, arg_sig) = _process(f)
8      return TypedFn(self, ast, static_env, arg_sig)
9    # ...
```

6.9 Checking Typed Functions

When the Ace compiler begins typechecking a `TypedFn` (when asked to by `acec`, or when it is invoked interactively, not discussed in this paper), it proceeds as follows:

1. An `ace.Context` is constructed with a reference to the function (Listing 9, lines 2-3).
2. The base is asked to initialize the context. This can be seen on lines 2-4 of Listing 8, where the `PyBase` base adds an attribute tracking local bindings (initially only the arguments) and the return type, which will be synthesized but initially is not known.
3. For each statement in the body (after the documentation string and type annotation), the compiler delegates to either the base, or of a type synthesized from a subterm, by calling a method named `check_`F, where F is the form of the statement (derived directly from Python's `ast` package [6]). These methods are responsible for checking that the statement is well-typed (raising an `ace.TypeError` if not) and having any needed effect on the context.
4. The base synthesizes a type for the function as a whole via the `syn_FunctionDef_outer` method. Here, an `arrow` type is synthesized (lines 26-29) based on the argument signature and the synthesized return type.

The assignment statements on lines 6.5[8] and 6.6 are checked by the base method `check_Assign_Name` shown on lines 8.6-8.14 and the return statement on line 6.7 is checked by the base's `check_Return` on lines 8.19-8.23. Both work similarly: if this is the first assignment to a particular name, or the first return statement seen, then the value must be able to synthesize a type in the current context. Otherwise, it is analyzed against the previously synthesized type.

6.10 Bidirectional Typechecking

Synthesis and analysis are mediated by the context via its `syn` and `ana` methods. To see how it works, let us begin at the first statement in our example, the assignment statement on line 6.5. As just stated, the method `check_Assign_Name` on lines 8.6-8.14 is called. Because the locals dictionary added to the context by the base does not yet have a binding for `x`, the base asks to synthesize a type for the value being assigned, `a.amount`, by calling `ctx.syn`.

This method is defined on lines 9.5-9.26. The relevant case in this method is the one on line 9.9, because `a.amount` is of the form `Attribute` according to Python's grammar. The context delegates to the type recursively synthesized for its value, `a`. We recurse back into `syn`, now taking the first branch for terms of the form `ast.Name`. Synthesizing a type for a name is delegated to the base by calling its `syn_Name` method. We can see its implementation for the base we are using on lines 8.37-8.44. The identifier `a` is an argument to the function, which the base included in the initial locals dictionary, so we hit a base case and the type `A` is synthesized.

[8] In this section, we will need to refer to code in Listings 6, 8, 9 and 10, so we adopt this syntactic convention to refer to line numbers for concision.

Listing 8 A portion of the base used in our examples thus far, defined in the `examples.py` package.

```
class PyBase(ace.Base):
  def init_ctx(self, ctx):
    ctx.locals = dict(ctx.fn.arg_sig)
    ctx.return_t = None

  def check_Assign_Name(self, ctx, s):
    x, e = s.target.id, s.value
    if x in ctx.locals:
      ctx.ana(e, ctx.locals[x])
    else:
      ty = ctx.syn(e)
      ctx.locals[x] = ty

  def trans_Assign_Name(self, ctx, target, s):
    return target.direct_translation(ctx, s)

  def check_Return(self, ctx, s):
    if ctx.return_t == None:
      ctx.return_t = ctx.syn(s.value)
    else:
      ctx.ana(s.value, ctx.return_t)

  def trans_Return(self, ctx, target, s):
    return target.direct_translation(ctx, s)

  def syn_FunctionDef_outer(self, ctx, f):
    if ctx.return_t == None:
      ctx.return_t = unit
    return arrow[ctx.fn.arg_sig, ctx.return_t]

  def trans_FunctionDef_outer(self, ctx, target, f):
    if f.name == "__toplevel__":
      return target.Suite(ctx.trans(f.body))
    else:
      return target.direct_translation(f, ctx)

  def syn_Name(self, ctx, e):
    x = e.id
    if x in ctx.locals:
      return ctx.locals[x]
    elif x in ctx.fn.static_env:
      return self.syn_lifted(x)
    else:
      raise ace.TypeError("...var not bound...", e)

  def trans_Name(self, ctx, target, e):
    if e.id in ctx.locals:
      return target.direct_translation(ctx, e)
    else:
      return self.trans_lifted(ctx, target, e)

  default_Dict_asc = default_Str_asc = dyn
  # ...
  def init_target(ctx): return PyTarget()
py = PyBase()
```

Listing 9 The `ace.Context` class delegates typechecking and translation of expressions, depending on their form and sub-terms, to a base, a type or a type constructor.

```
class Context(object):
  def __init__(self, fn):
    self.fn = fn

  def syn(self, e):
    if instanceof(e, ast.Name):
      delegate = self.fn.base
      ty = delegate.syn_Name(self, e)
    elif instanceof(e, ast.Attribute):
      delegate = self.syn(e.value)
      ty = delegate.syn_Attribute(self, e)
    # ... other compound forms similar (cf appendix)
    elif isinstance(e, ast.Str):
      return self.ana(self, e,
        self.fn.base.default_Str_asc)
    # ... other unascribed literal forms similar
    elif is_ascribed_Dict(e):
      lit, delegate = get_lit(e)
      if issubclass(delegate, Type): # tycon
        ty = delegate.syn_Dict(self, lit)
      else:
        return self.ana_Dict(lit, delegate)
    # ... other ascribed literal forms similar
    e.delegate = delegate
    e.ty = ty
    return ty

  def ana(self, e, ty):
    if instanceof(e, ast.Dict):
      ty.ana_Dict(self, e)
    # ... other literal forms similar
    else:
      syn = self.syn(e)
      if ty != syn:
        raise TypeError("...syn/ana mismatch...", e)
      return
    e.delegate = e.ty = ty

  def trans(self, target, e):
    d = e.delegate
    if instanceof(e, ast.Name):
      trans = d.trans_Name(self, target, e)
    elif instanceof(e, ast.Attribute):
      trans = d.trans_Attribute(self, target, e)
    # ... other forms similar
    e.trans = trans
    return trans
```

Listing 10 The `examples.fp.record` type constructor.

```
@slices_to_sig
class record(ace.Type):
  def __init__(self, sig, anon=False):
    self.sig, self.anon = sig, anon

  @classmethod
  def syn_Dict(cls, ctx, e):
    sig = Sig((f, ctx.syn_ty(v))
      for f, v in zip(e.keys, e.values))
    return cls(sig, anon=True)

  def ana_Dict(self, ctx, e):
    for f, v in zip(e.keys, e.values):
      if f.id in self.sig:
        ctx.ana(v, self.sig[f.id])
      else:
        raise ace.TypeError("...extra field...", f)
    if len(self.sig) != len(e.keys):
      raise ace.TypeError("...missing field...", e)

  def trans_Dict(self, ctx, target, e):
    if len(self.sig) == 1:
      return ctx.trans(e.values[0])
    else:
      value_dict = dict(zip(e.keys, e.values))
      return target.Tuple(
        ctx.trans(target, value_dict[field])
        for field, ty in self.sig)

  def syn_Attribute(self, ctx, e):
    if e.attr in self.sig:
      return self.sig[e.attr]
    else:
      raise ace.TypeError("...field not found...", e)

  def trans_Attribute(self, ctx, target, e):
    if len(self.sig) == 1:
      return ctx.trans(target, e)
    else:
      idx = idx_of(self.sig, e.attr)
      return target.Subscript(
        ctx.trans(target, e.value), target.Num(idx))

  def __eq__(self, other):
    if isinstance(other, record):
      if self.anon or other.anon: return self is other
      else: return self.sig == other.sig

  def trans_type(self, target):
    return target.dyn
```

We can now pop back up to line 9.11, which can now delegate synthesis of a type for `a.amount` to the type `A` via the `syn_Attribute` method. Recall that `A` was constructed using the `record` constructor in Listing 1. The definition of this type constructor is shown in Listing 10. In Ace, type constructors are classes inheriting from `ace.Type` and types are instances of such classes. Constructor indices are given through the class constructor, called `__init__` in Python. Here, `record` requires a signature, which is a mapping of field names to types (an instance of `examples.fp.Sig`, which performs well-formedness checking, not shown). The class decorator `slices_to_sig` provides the convenient notation shown used in Listing 1 (by adding a metaclass to override the class object's subscript operator, not shown). The `syn_Attribute` method is shown on lines 10.30-10.34. It simply looks in the signature for the provided field name, so `decimal[2]` is synthesized. We can now go back up to the assignment statement that triggered this chain of calls and see on line 8.14 that a binding for `x` is added to the locals dictionary and checking of this statement succeeds.

The next statement also assigns to `x`, but this time, the base asks to analyze the value against `A` using the `ana` method of `Context`, shown on lines 9.28-9.37. Analysis differs from synthesis only for unascribed literal forms. For any other form, the context simply asks for an equal type to be synthesized (we will discuss in Sec. **??** future work on integrating subtyping, where this would be relaxed). For concision, we leave the details of the delegation protocol for binary operators to the appendix, which also shows the definition of the `decimal` type constructor. The addition of two terms of type `decimal[2]` has type `decimal[2]`, so checking the second assignment statement also succeeds.

The final statement in `example` is a return statement. Because it is the first return statement encountered, it too requires that the returned value synthesize a type. Here it is an ascribed literal. We can see on lines 9.17-9.22 how this is handled. Because the ascription is a type constructor, `record`, rather than a type, the literal synthesizes a type by calling a *class method*, `record.syn_Dict`, shown on lines 10.6-10.10. This method constructs a record signature by synthesizing a type for each value (using comprehension syntax for concision) then returns a new instance of the class. It is marked as anonymous. Anonymous records differ only in how equality is decided, shown on lines 10.44-10.47. We want anonymous records to be equal structurally, while records declared like `A` are by default distinct even if they have the same signature (following the convention of most functional languages with record declarations).

Synthesizing a type for the value of the field `y`, also an ascribed literal, takes a different path because the ascription is a type, `A`, not a type constructor. Type ascriptions, as we have seen in the previous examples, cause the literal to analyzed. This proceeds through the `ana_Dict` method on lines 10.12-10.19, which analyzes the value of each field against the corresponding type in the signature, raising a type error if there are missing or extra fields.

Synthesizing a type for the value of the field `remark`, an unascribed literal form, follows a third path, seen on lines 9.13-9.15. Unascribed literals are treated as if they had been given the default ascription specified by the base. Our base specifies the default ascription as `dyn` on line 8.52, which accepts our literal, so synthesis succeeds. The return type of `example` is thus equal to:

```
record(Sig(('y', A), ('remark', dyn)), anon=True)
```

6.11 Active Translation

Once typechecking is complete, the compiler enters the translation phase. The base creates an instance of a class inheriting from `ace.Target` for use during this phase via the `init_target` method (line 8.54). This object provides methods for code generation and supports features like fresh variable generation, adding imports and so on. It's interface is not constrained by Ace (we will see what Ace requires of a target below) and the mechanics of code generation are orthogonal to the focus of this paper, so we will discuss it relatively abstractly. The simplest API would be string-based code generation. For the Python target

we use here, we generate ASTs. The API is based directly on the `ast` library, with a few additional conveniences just mentioned.

This phase follows the same delegation protocol as the typechecking phase. Each `check_/syn_/ana_`X method has a corresponding `trans_`X method. The typechecking phase saved the entity that was delegated control, along with the type assignment, as attributes of each node, `delegate` and `ty` respectively, so that these need not be determined again. This protocol can be seen on lines 9.39-9.47. Translation methods have access to the context and node, as during typechecking, as well as the target.

Because we are targeting Python directly, most of our `trans` methods are direct translations (factored out into a helper function). The main methods of interest that are not entirely trivial are `trans_FunctionDef_outer` on lines 8.33-8-37 and the two translation methods in Listing 10, which implement the logic described in Sec. 6.5.

Each type constructor must also specify a `trans_type` method. This is important when targeting a typed language (so that the translations of type annotations can be generated, for example). As we will see in the next two sections, this is also critical to ensuring type safety. The language *checks* not only that translations having a given type are well-typed, but that they have the type specified by this method (requiring that the target provide a `is_of_type` method). Here, because we are simply targeting Python directly, the `trans_type` method on line 10.49-10.50 simply generates `target.dyn`.

When this phase is complete, each node processed by the context will have a translation, available via the `trans` attribute. In particular, each typed function has a translation. Note that some nodes are never processed by the context because they were reinterpreted by the delegate (e.g. the field names in a record literal), so they do not have translations (as expected, given our discussion above).

To support external compilation, the target must have an `emit` method that takes the compilation script's file name and a reference to a string generator (an instance of `ace.util.CG`) and emits source code. The string generator we provide can track indentation levels (to support Python code generation and make generation for other languages more readable, for the purposes of debugging). It allows non-local string generation via the concept of user-defined *locations*. Each file that needs to be generated is a location and there can also be locations within a file (e.g. the imports vs. the top-level code), specified by a target the first time it finds that a necessary location is not defined. A generated entity (e.g. an import, class definition or function definition) can only be added once at a location. We saw this in Listing 3 for the decimal-to-string conversion. The API will be discussed further later.

6.12 Remaining Tasks and Timeline

We have submitted this work to OOPSLA 2014. The following tasks remain:

1. We have implemented a unidirectional version of Ace but need to update it with the bidirectional mechanism described above, and implement the examples above.

2. We need to write down the full delegation protocol, not just the subset used above.

3. In the unidirectional version, we implemented the entirety of the OpenCL type system as an example. We plan to port this work to the new bidirectional version.

4. We need to show how values from the type-level language can be lifted safely into the expression language to enable interoperability. This is a novel bidirectional protocol as well. I have sketched this out and have implemented a unidirectional variant of it using our OpenCL type system. We plan to implement a bidirectional variant of it and submit this to either in SLE 2014 or GPCE 2014 in late May

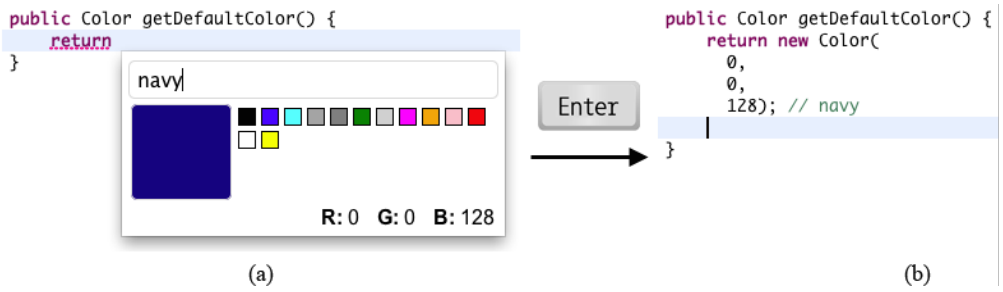

Figure 16: (a) An example code completion palette associated with the `Color` class. (b) The elaboration generated by this palette.

7 Active Code Completion

An abstraction might benefit from support from a variety of tools beyond the compiler. For example, software developers today make heavy use of the code completion support found in modern source code editors [43]. Editors for object-oriented languages provide code completion in the form of a floating menu containing contextually-relevant variables, assignables, fields, methods, types and other code snippets. By navigating and selecting from this menu, developers are able to avoid many common spelling and logic errors, eliminate unnecessary keystrokes and explore unfamiliar APIs without incurring the mental overhead associated with switching to an external documentation tool.

In all such systems, the code completion interface has remained primarily menu-based. When an item in the menu is selected, code is inserted immediately, without further input from the developer. These systems are difficult to extend: a fixed strategy determines the completions that are available, so library providers cannot directly specify new domain-specific or contextually-relevant logic or provide assistance beyond that which a menu can provide. In this paper we propose a technique called *active code completion* that eliminates these restrictions using active types. This makes developing and integrating a broad array of highly-specialized code generation tools directly into the editor, via the familiar code completion command, significantly simpler.

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

For example, consider the following Java code fragment:

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

If the developer invokes the code completion command at the indicated cursor position (_), the editor looks for a *palette definition* associated with the *type* that the expression being entered is being analyzed against, which in this case is `Color` (due to the return type annotation on the method). If an associated palette is found, a menu item briefly describing this palette is added to the standard code completion menu. When selected, the corresponding palette is shown, replacing the standard code completion menu. Figure 16(a) gives an example of a simple palette that may be associated with the `Color` class[9].

The developer can interact with such palettes to provide parameters and other information related to their intent, and receive immediate feedback about the effect these choices will have on the behavior of the object being constructed. When this interaction is complete, the palette produces an elaboration for insertion at the cursor, of the type (here, class) that the palette was associated with (Figure 16(b)).

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

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

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

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

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

7.1 Design Process

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

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

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

The primary concerns relevant to this thesis are:

- The palette mechanism should not be tied to a specific editor implementation. We achieve this by using a URL-based scheme for referring to palettes, which are implemented as webpages, which can be embedded into any editor using standard techniques for embedding browsers into GUIs.
- The palette mechanism should not be able to arbitrary access the surrounding source code (for privacy reasons, as identified by survey participants). By using a browser, and only allowing access to highlighted strings, we avoid this problem.

- We provide a mechanism by which users can associate palettes with types externally. This could cause conflicts with palettes that the type defines itself. To resolve this, we give users a choice whenever such situations occur by inserting all relevant palettes into the code completion menu.

7.2 Remaining Tasks and Timeline

This work has been published at ICSE 2012 [50]. The main remaining tasks have to do with specifying it in terms of the bidirectional typing mechanisms in Sec. 3 more directly, rather than informally as in our paper. We plan to do this when writing the dissertation. There are also some pieces of related work that have been published since this paper was accepted that we need to review.

8 Conclusion

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

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

But we take issue with this unbounded *principle of tolerance*, which he explained more directly as follows [16]:

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

Carnap formulated his principle of tolerance because he wished to freely explore the consequences of reasoning with different connectives and rules and emphasize that conducting philosophical reasoning by methodical logical analysis did not require one to accept any particular axioms. But he was also a prominent member of the Vienna circle, which sought a “unified science” the purpose of which was "to link and harmonise the achievements of individual investigators in their various fields of science" by the construction of a "constitutive system" [29]. These two programs were in conflict: were the Vienna circle’s “constitutive system” to become naïvely tolerant of the inference rules (what Carnap called L-rules) developed by individual investigators, it would find that this would then deny them autonomy of reasoning. According to social scientist Raz, the primary justification of tolerance is to maximize individual autonomy in a multi-party context [55]. Yossie Nehushtan argues that a liberal society “should not tolerate anything that denies the justifications of tolerance” [46]. Thus, we propose a more cooperative *principle of reciprocal tolerance*:

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

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

9 Timeline

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

- The work on type-specific languages has been accepted to ECOOP 2014. The deadline for final revisions is May 12, so we will complete the remaining work, including writing a technical report, by then. This should be sufficient for the chapter on Wyvern in the dissertation.
- The work on @λ and active embeddings is conceptually complete. We plan to complete the technical details and exposition and submit it to POPL in July. This should be sufficient for the chapter on @λ in the dissertation.
- The work described the core of Ace was submitted to OOPSLA 2014 on Mar. 25. Notification is May 26. A paper with Nathan Fulton, a former undergraduate student who I co-supervised, on regular expression types in Ace is in preparation for submission to PLAS 2014 on April 20. We plan to submit a paper to either SLE 2014 or GPCE 2014 on FFIs using Ace (in particular, a FFI for OpenCL, which was already developed); both conferences have due dates at the end of May. Together, these will form the chapters on Ace in the dissertation.
- The work on active code completion has been completed and published at ICSE 2012 [50]. The remaining work is expository and will be completed in the course of writing the Graphite chapter of the dissertation.

References

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

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

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

[4] PEP 3107 – Function Annotations. http://legacy.python.org/dev/peps/pep-3107/, 2010.

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

[6] The python language reference. http://docs.python.org/2.7/reference/introduction.html, 2013.

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

[8] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *Software, IEEE*, 25(4):29–36, 2008.

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

[10] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-Parallel Language. In *PPoPP '93*, pages 102–112. ACM Press, New York, NY, 1993.

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

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

[13] 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.

[14] 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.

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

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

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

[18] V. Cavé, Z. Budimlić, and V. Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, page 9. ACM, 2010.

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

[20] S. Erdweg, T. Rendel, C. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*, pages 391–406, Portland, Oregon, USA, Oct. ACM Press.

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

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

[23] 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.

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

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

[26] J.-Y. Girard. Une extension de l'interpretation de gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. *Studies in Logic and the Foundations of Mathematics*, 63:63–92, 1971.

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

[28] D. Grossman, J. G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst*, 22(6):1037–1080, 2000.

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

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

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

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

[33] 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.

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

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

[36] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *POPL '14*, pages 179–192. ACM, 2014.

[37] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[38] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. Meyrowitz, editor, *OOPSLA*, volume 21:11 of *ACM Sigplan Notices*, pages 214–223, Oct. 1986.

[39] A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 133–144. ACM, 2006.

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

[41] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 82–93. ACM, 2005.

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

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

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

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

[46] Y. Nehushtan. The limits of tolerance: A substantive-liberal perspective. *Ratio Juris*, 20(2):230–257, 2007.

[47] 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.

[48] 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.

[49] A. Ohori and K. Ueno. Making Standard ML a Practical Database Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 307–319, New York, NY, USA, 2011. ACM.

[50] 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.

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

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

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

[54] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.

[55] J. Raz. Autonomy, toleration, and the harm principle. *Justifying Toleration: Conceptual and Historical Perspectives*, pages 155–175, 1988.

[56] J. H. Reppy. Concurrent ml: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

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

[58] 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.

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

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

[61] 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.

[62] 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.

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

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

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

[66] 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.

[67] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 111–121. ACM, 2010.

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

[69] 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.

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

[71] 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.

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

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

[74] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. Specifying properties of concurrent computations in CLF. *Electronic Notes in Theoretical Computer Science*, 199:67–87, 2008.

[75] S. Weirich, D. Vytiniotis, S. L. P. Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In T. Ball and M. Sagiv, editors, *POPL '11*, pages 227–240. ACM, 2011.