Actively-Typed Programming Systems

Thesis Proposal

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

September 10, 2013

Abstract

We propose a thesis defending the following statement:

Active types allow developers to extend the compile-time and edit-time semantics of a programming system from within libraries in a safe and expressive manner.

1 Motivation

Designing and implementing a programming language together with its supporting tools (collectively, a *programming system*) that has a sound theoretical foundation, helps users identify and fix errors as early as possible, supports a natural programming style, and that performs well across diverse problem domains and hardware platforms remains a grand challenge in computing. It has become clear that no small, fixed collection of primitive abstractions and tools can fully satisfy these criteria in all situations. Rather, researchers and domain experts must continue, in a principled manner, to design and implement novel abstractions, provide alternative implementations of existing abstractions, and supplement languages with new tools, and tools with new behaviors, to better satisfy these criteria within the constraints of different problem domains.

When possible, taking a *language-internal approach* to implementing a new abstraction or system behavior (collectively, a new *feature*) is simpler and more practical. If a feature can be realized by creatively repurposing existing features and distributed as a library, both providers and clients face fewer barriers to adoption because it is easy to gradually integrate library-based features into existing code and they leverage well-understood and well-developed mechanisms. But taking this approach is often *not* possible today because libraries are generally vehicles for specifying the run-time behavior of a program. The compile-time and edit-time behaviors of a programming system cannot easily be adapted or extended in a specialized way from within libraries in most programming systems. That is, the language's syntax, static semantics and corresponding dynamic semantics are fixed in advance, the compiler is a "black box" implementation of these fixed semantics, and the other tools, like code editors and debuggers, operate according to fixed, domain-agnostic protocols based on these fixed semantics.

If any of these components of the system must be modified to realize a new feature, providers must instead take a *language-external approach*, either by developing a new or derivative programming system (often centered around a so-called *domain-specific language* [**?**]) or by extending an existing system by some mechanism that is not part of the language itself, such as an extension mechanism supported by a particular compiler or other tool.

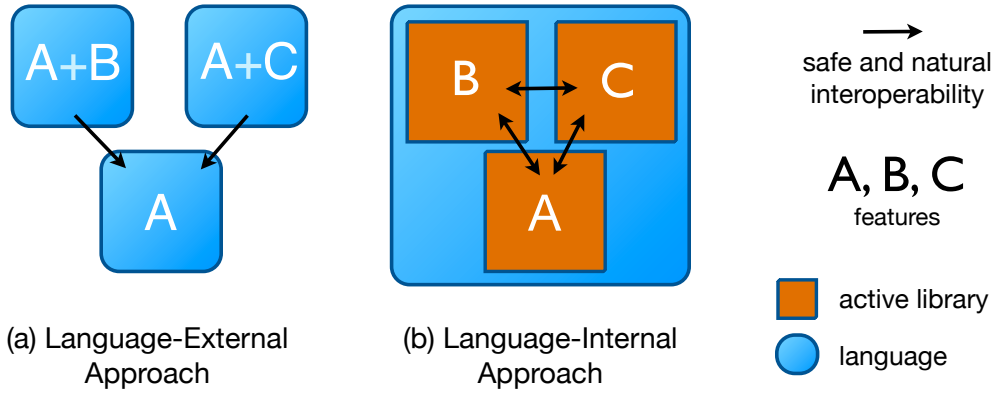

Figure 1: (a) With the language-external approach, novel constructs are packaged into separate languages. Users can only safely and naturally call into languages if the interface uses common constructs and an interoperability layer has been developed (the *interoperability problem*). (b) With the language-internal approach, there is one extensible host language and the compile-time and edit-time logic governing novel constructs is expressed within libraries. If the extension mechanism guarantees the safety of arbitrary compositions of extensions, the necessary primitives can simply be imported by library clients as needed, and so interoperability is not a problem.

Unfortunately, when providers of new features take language-external approaches, it causes problems for clients. Features become coupled to a collection of other unrelated and often underdeveloped features, making adoption more costly when these other choices are not appropriate. For example, although recent evidence suggests that developers prefer language-integrated parallel programming abstractions to library-based implementations if all else is equal [**?**], library-based implementations remain more widely adopted because they do not require porting applications to a language designed around a few privileged parallel programming abstractions. These abstractions may only be useful in performance-critical portions of the application, and then only in cases where no competing abstraction would be more appropriate.

If it were easy to call between languages, then having a variety of specialized languages would be less of a problem. In fact, some have advocated for just this development model, calling it the the *language-oriented approach* to software development [**?**]. A fundamental problem with this approach, however, arises at the interfaces between languages. The specialized abstractions particular to one language cannot be safely and naturally expressed in another, in general, so building a program out of components written in many different languages is difficult or impossible whenever these specialized abstractions are exposed at library interface boundaries. We refer to this as the *interoperability problem*.

One strategy increasingly being taken by programming language designers to partially address the interoperability problem today is to target an established language, such as the Java Virtual Machine (JVM) bytecode, and support a superset of its constructs. Scala [**?**] and F# [**?**] are examples of general-purpose languages that have taken this approach. This only provides full interoperability in one direction (Figure 1a). While calling into the common language becomes straightforward, calls in the other direction, or between the languages sharing the common target, are still restricted by the constructs available in the common language. If full interoperability is desired, new languages must only include constructs that can already be expressed safely and reasonably naturally in the common language. Many innovative features, however, can be difficult to define in terms of existing features in ways that guarantee all necessary invariants are maintained and that do not require large amounts of boilerplate code. For example, the type system of F# guarantees that null

values cannot occur within F# data structures, but maintaining this important invariant still requires run-time checks because the typing rules of F# do not apply when F# code is called from other languages on the Common Language Infrastructure (CLI) like C#. The F# type system also includes support for checking that units of measure are used correctly [**?**], but this domain-specific invariant is left completely unchecked at language boundaries. In Scala, interfaces built around traits that have default method implementations are difficult to implement from Java or other JVM languages and the workaround can break if the trait is modified [**?**]. In some cases, desirable features must be omitted entirely due to concerns about interoperability. The module system in F#, for example, is substantially simpler than that in its predecessor, OCaml, despite F# otherwise aiming to maintain compatibility, due to the need for bidirectional interoperability between F# and other CLI languages.

For these reasons, we argue that the language-oriented approach is fundamentally flawed and so taking a language-external approach to realizing a new feature, where new features are packaged into mutually incompatible languages and associated tools, should be considered harmful and avoided when possible. The goal of the research proposed here is to fundamentally reorganize the core components of the programming system so that such language-external approaches are less frequently necessary, by designing *language-internal extension mechanisms* that give developers control over edit-time and compile-time behaviors that have previously been centrally-controlled by the committees that govern languages or tools[1]. Specifically, we will show how control over **parsing**, **typechecking**, **translation** (the first stage of compilation) and **code completion** can be delegated to user-defined logic distributed in libraries, as illustrated in Figure 1b. Such libraries are called *active libraries* because, rather than being passive consumers of features already available in the system, they contain logic invoked directly by tools to provide new domain-specific functionality during development or compilation [**?**]. Features implemented within active libraries can be imported as needed by clients of libraries that rely on them, unlike features implemented by language-external means.

read Arch D. Robison. Impact of economics on compiler optimization.

Some critical issues having to do with safety must be overcome before library-based extension mechanisms can be introduced into a programming system, because if too much control over such core aspects of the system is given to users, the system may become unreliable. For example, an extension could weaken important metatheoretic guarantees previously provided by the system. Type safety, for instance, may no longer hold if the static and dynamic semantics of the language can be modified arbitrarily 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 lead to subtle problems at link-time. For example, if two active libraries defined differing semantics for the same syntactic form, the issue would only manifest itself when both libraries were imported somewhere within the same program. These kinds of safety issues have plagued previous attempts at language-internal extensibility (discussed further in the next section). To prevent them, the system must strictly constrain extensions such that they are guaranteed safe in isolation and also safely composable in any combination.

The mechanisms described in this thesis are designed to be highly **expressive**, allowing library-based implementations of features comparable to built-in features found in modern programming systems, without compromising on these important notions of **safety**. This is accomplished by organizing extension logic around types, rather than languages. By scoping new compile-time and edit-time logic to expressions of the type with which it is associated, rather than applying it globally or only within contiguous syntactic blocks as in many previously-developed mechanisms, conflicts are avoided. We call types with such logic associated with them *active types* and systems that support them *actively-typed programming systems*. By constraining the extension logic with a type system and using techniques from the compiler correctness literature, we can ensure that the system as a whole maintains important safety properties.

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

2 Background: Active Libraries

The term *active libraries* was first used by Veldhuizen et al. [**?**, **?**] to describe "libraries that take an active role in compilation, rather than being passive collections of subroutines". The authors went on to suggest a number of concrete reasons libraries might benefit from closer interactions with the programming system, including 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. Our definition, given in Section 1, differs from theirs for this reason.)

move into separate tex file

The first concrete realizations of active libraries, prompting the introduction of the term, were scientific libraries that performed domain-specific program optimization at compile-time by exploiting language mechanisms, like C++ templates, that require compile-time computation. A prominent example in the literature is Blitz++, a numerics library that uses C++ template metaprogramming to optimize compound operations on vectors and matrices by eliminating intermediate allocations [**?**]. Although this and a number of other interesting optimizations are possible by this family of techniques, their expressiveness is limited because template expansion supports only substitution of compile-time constants into pre-written code, and templates are notoriously difficult to read, write and debug.

More powerful compile-time *term rewriting mechanisms* integrated into some languages can also be used for optimization, as well as for inserting specialized error checking and reporting logic and extending the language with powerful domain-specific abstractions. These mechanisms are highly expressive because they allow users to programmatically manipulate syntax trees directly, but they suffer from problems of composability and safety. Macros, such as those in MetaML [**?**], Template Haskell [**?**] and others, can take full control over all the code that they enclose, so there is no way to guarantee that nested macros will compose as intended. Outer macros can neglect to invoke or manipulate the output of inner macros in ways that can cause conflicts or weaken important guarantees. Once data escapes a macro's scope, there is no way to rely on the guarantees and features that were available within its scope – the output of a macro is a value in the underlying language, so the underlying language is all that governs it (a problem related to the interoperability problem of Section 1). It can also be difficult to reason about the semantics of code when a number of enclosing macros may be manipulating it.

reference other macro systems

Some rewriting systems go beyond the block scoping of macros. Xroma (pronounced "Chroma"), for example, is designed around active libraries and allows users to insert custom rewriting and error checking passes into the compiler from within libraries [**?**]. Similarly, the derivative of Haskell implemented by the Glasgow Haskell Compiler (GHC) allows libraries to define custom compile-time term rewriting logic if an appropriate flag is passed in [**?**]. In both cases, the user-defined logic can dispatch on arbitrary patterns of code throughout the component or program the extension is active in, so these mechanisms are highly expressive and avoid some of the difficulties of block-scoped macros. But libraries containing such global rewriting logic cannot be safely composed because two different libraries may attempt to rewrite the same piece of code differently. It is also difficult to guarantee that such logic is correct and difficult for users to reason about code when simply importing a library can change the semantics of the program in a non-local manner.

Another example of an active library approach to extensibility is SugarJ [**?**]. SugarJ allows libraries to extend the base syntax of Java in a nearly-arbitrary manner, and these extensions are imported transitively throughout a program. SugarJ libraries are thus also not safely composable. For example, a library that defines a literal syntax for HTML would conflict with another that defines a literal syntax for XML because they define differing semantics for some of the same syntactic forms. And again, it can be difficult for users to predict what an unfamiliar piece of syntax desugars into, leading to readability issues.

3 Active Types

Developers specify type-specific compile-time and edit-time features by equipping type families, when they are defined, with functions written in an appropriate metalanguage. These are selectively invoked by tools, such as the parser, type checker, translator and editor, when they work with expressions of that type, and only in those situations.

3.1 Example: Regular Expressions

To make our approach more concrete, let us begin with a simple example that demonstrates the use cases examined in this thesis. If a programming system included full compile-time and edit-time support for regular expressions, it might support features like the following:

1. **built-in syntax for regex literals** so that malformed regular expression patterns result in intelligible compile-time parsing errors. One study found that malformed regular expressions are a common problem in Java [**?**].

2. **type checking logic** that ensures that key constraints related to regular expressions are not violated, such as that out-of-bounds group indices are not used [**?**] and that only values with correct types are spliced into regular expressions. When a type error is found, the error message should be intelligible.

3. **translation logic** that partially or fully compiles regular expressions into the efficient internal representation that will be used by the regular expression matching engine at run-time. In most languages, this compilation step occurs at run-time, even if the pattern is fully known at compile-time, thereby introducing latency into programs. If the developer is not careful, regular expressions used repeatedly in a program might be needlessly re-compiled on each use. By performing this step ahead-of-time, these dangers can be avoided.

4. **editor-integrated tooling** for interactively testing patterns against example strings, quickly referring to documentation, searching databases of common patterns and other domain-specific edit-time facilities.

In a conventional *monolithic* programming system, support for each of these features would need to be built into the language and tools. No such system today has support for all of the features above. Instead, libraries provide support for regular expressions by leveraging general-purpose abstractions. Unfortunately, it is impossible to fully encode the syntax and the specialized static and dynamic semantics described above in terms of standard general-purpose notations and abstractions. These libraries have thus needed to compromise, typically by representing regular expressions using strings and deferring parsing, typechecking and translation to run-time. This introduces performance overhead and can lead to unanticipated run-time errors (as shown in [**?**]) and security vulnerabilities (due to injection attacks when splicing is used, for example). Similarly, edit-time tools that make working with regular expressions easier, as described above, are rarely integrated into editors, and never in a way that facilitates their discovery and use directly when manipulating regular expressions. In most cases, such tools must discovered independently and accessed externally (for example, via a browser), making their use both less common and more awkward than necessary (we provide evidence for this [**?**], see Section **??**).

Existing active library approaches could be used to implement some of these features, but the composition issues described in Section 2 make such solutions suboptimal. For example, if SugarJ was used by two different regular expression engines to provide literal syntax for regular expression patterns, there could easily be conflicts at link-time because both will introduce many of the same standard notations but back them with differing implementations.

We can observe, however, that each of these features relates specifically to how terms representing regular expression patterns are processed by tools, and that such terms are always classified by a particular user-defined type[2]. We will refer to it by a simple name, `Pattern`, but it would need to have a fully-qualified name to ensure that conflicts due to name clashes cannot occur. It is only when creating, editing or compiling expressions of this type that the logic enumerated above would ever need to be invoked. Indeed, this is a common pattern – types are widely seen as a natural organizational unit around which the semantics of programming languages and logics are defined (e.g. in both TAPL [6] and PFPL [**?**], most chapters simply describe the semantics and metatheory of a few new types without reference to other types). This suggests a principled alternative to the mechanisms described earlier that preserves most of their expressiveness but eliminates the possibility of conflict and makes it easier to reason locally about a piece of code: associating extension logic to a single type or type family as it is defined and scoping it only to expressions, or basic operations on expressions, in that type or type family. This is the common motif behind all of the actively-typed mechanisms proposed in this dissertation.

3.2 Proposed Contributions

In Section **??**, we will describe how to extend the core static and dynamic semantics of a language in an actively-typed and safe manner. We will distill the essence of our approach, which we call **active typechecking and translation (AT&T)**, by specifying an actively-typed lambda calculus called @λ, proving several key safety theorems, and examining the connections between active types and type-level computation, type abstraction and typed compilation techniques. We will then go on to demonstrate the expressiveness of this mechanism by designing and implementing a full-scale language, Ace, and implementing a number of interesting type families from existing full-scale languages as active libraries within Ace. A primary focus of these is on high-performance programming abstractions, but we will also show some uses of Ace for other domains.

In both @λ and Ace, semantic extensions operate over a fixed syntax. In Section **??**, we will show how domain-specific syntax can also be introduced in an actively-typed and safely composable manner. Our technique is called **actively-typed parsing** and it will be implemented within the Wyvern language. Our novel contributions include the design of a second parsing phase that runs in tandem with typechecking, a method for using whitespace to delimit domain-specific syntax, a generalization of standard literal forms so that they can be used for more than one type, an underlying mechanism for associating compile-time data and functionality with structural types, and a recursive use of the active parsing technique to introduce a domain-specific syntax for defining new actively-typed grammars. We will describe each of these contributions as implemented in Wyvern, give minimal formalisms showing how the core syntax and the type system of Wyvern support actively-typed parsing, and show a number of examples expressible by this technique.

Finally, in Section **??**, we will show how editor-integrated domain-specific tooling for working with expressions of a single type can be introduced from within active libraries, by a technique known as **active code completion**. Developers associate domain-specific user interfaces, called *palettes*, with types. Users discover and invoke palettes from the code completion menu at edit-time, populated according to an actively-typed mechanism similar to that of actively-typed parsing. When they are done, the palette generates a term of that type based on the information received from the user. Using several empirical methods, we survey the expressive power of this approach, describe the design and safety constraints governing the system architecture as well as particular palette user interfaces, and develop one such system for the Java language[3], based on these constraints, called

[2]More generally, several different types within an indexed type family. For example, `Pattern`(n) represents a pattern containing n matching groups, so `Pattern` is a type family indexed by a natural number, n [**?**]. We will return to this distinction in Section **??**.

[3]In Graphite, palettes are associated with Java classes, which are technically different from types. However,

Graphite. Using Graphite, we implement a palette for working with regular expressions to conduct a pilot study to provide evidence for the usefulness of this approach, and of contextually-relevant editor-integrated tooling generally.

Together, ...

write something about a unified mechanism that can be used across concerns

4 Active Typechecking and Translation

4.1 Theory

4.2 Ace

5 Active Parsing

Wyvern...

6 Active Code Completion

7 Related Work

We review existing approaches that are available to researchers and domain experts who wish to develop novel constructs below.

7.1 Language-Oriented Approaches

7.1.1 Language Frameworks

A number of tools have been developed to assist with this task of developing new languages and their associated tools, including compiler generators, language workbenches and domain-specific language frameworks (see [2] for a review). In some cases, these tools allow language features to be defined modularly and composed differentially to produce a variety of different languages. To our knowledge, none of these mechanisms guarantee that any combination of features can be safely composed, nor do they guarantee safety properties about the resulting languages. User-defined code is still ultimately compiled against a particular language, and because language composition cannot be automatic or guaranteed safe, interoperability with code written in other languages is thus limited by the difficulties described above despite the modular construction.

7.1.2 Extensible Compilers and Tools

A related methodology is to implement language features as compiler and tool extensions directly. A number of extensible compilers have been developed to support this approach (see [1]). As with language frameworks, this approach can lead to *de facto* implementation of a new language, since user libraries must be compiled against a particular combination of extensions, and these extensions cannot be guaranteed to compose in general. Moreover, a language may have multiple competing compilers and other tools. By relying on implementation-specific features of a single tool to define core semantics and behaviors, the clean conceptual separation between languages and tools is broken, leading to compatibility issues and hard-to-anticipate behaviors for user code. We argue that this approach should be considered harmful.

active code completion could also be implemented just as well in non-object-oriented languages.

7.2 Library-Oriented Approaches

7.2.1 Embedded DSLs

Embedded domain-specific languages are languages that creatively repurpose existing language constructs to create interfaces that resemble those of a distinct language. In languages with rich type systems, such as Haskell, this approach can be quite successful (e.g. [**?**]). Ultimately, however, this approach is limited by the host system, and as discussed in Section 2, thus limits experts who want to express particularly novel constructs, relative to the host language.

7.2.2 Term Rewriting Systems

Many languages and tools allow developers to rewrite expressions according to custom rules. These can broadly be classified as *term rewriting systems*. Macro systems, such as those characteristic of the LISP family of languages [5], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [7]), and external rewrite systems also exist for many languages. These systems are expressive if used correctly, but verifying correctness and non-interference properties is difficult for the same reason. Manipulating source trees directly is a complex task, even in languages with simple grammars like LISP. Finally, term rewriting systems focus on rewriting terms to support alternative language semantics but do not intrinsically support extensions that cover the full programming system.

7.2.3 Active Libraries

Active libraries, as proposed by Czarnecki et al. [8], "are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code". They go on to suggest a number of areas in which the library could interact with the programming system, including optimizing code, checking source code for correctness, reporting domain-specific errors and warnings, and "rendering domain-specific textual and non-textual program representations and for interacting with such representations".

This paper was largely a proposal. The concrete implementations of this concept have largely been term rewriting systems described above. One prominent example within the active libraries literature is Blitz++, which uses the C++ template expansion system as a metalanguage to support optimizations of array operations. An example of tool support comes from the Tau package for tuning and analyzing parallel programs. Tau allows libraries to instrument themselves declaratively to hide internal details and complex internal representations from users. A more extensive system with support for active libraries is Xroma. Xroma allows users to provide annotations that intercept compilation of a component at various stages of compilation to support rewriting, custom error handling and custom error checking. The approach taken by Xroma, by still requiring direct syntax manipulation and allowing arbitrary interception, is flexible but not compositional, and generally more complex than may be necessary. There also does not appear to be a clear, well-founded theoretical foundation for this approach, nor for active libraries in general.

8 Work To Be Done

8.1 Completion and Evaluation of Ace

The work done so far on the Ace language establishes the practicality of active typechecking and translation as a foundational language mechanism and begins to apply it to one realistic problem domain, high-performance scientific computing on GPUs and other coprocessors, by internalizing OpenCL. However, to more fully establish the practicality of

this approach, it must be shown that Ace can support more sophisticated, higher-level language constructs as well, ideally from a variety of application domains. It must also be further shown that users who are not language design experts, and will not be utilizing the extension mechanism directly, can nevertheless use the language in practice.

Thus, a portion of the resources allocated to this project will be dedicated to completing the implementation of Ace, documenting it and releasing it both to the research community and to practitioners in areas where a useful set of constructs have been developed, beginning with the members of the high-performance computing community. This will also support synergy with ongoing collaborations in our group with other problem domains, including front-end and back-end web development, security, functional programming and object-oriented programming.

With broader usage by the research community and in practice, we will be able to continue to conduct case studies and empirical evaluations examining the usability and usefulness of Ace and the active typing mechanisms it introduces. Indeed, because of the flexibility of active typing and our implementation strategy, we plan on instrumenting the Ace compiler to send data and direct feedback from willing users back to us for analysis. This will allow us to analyze questions like:

- How often are different constructs and mechanisms used in practice?
- What kinds of errors appear most frequently in practice, and how long does it take users to resolve these errors?
- How do users feel about the mechanisms, constructs and error messages that they encounter, as determined by direct feedback solicited at the time of occurrence.
- How commonly do users utilize the active typing mechanism directly?

To our knowledge, no such detailed study of usage patterns of a programming language in the wild has been conducted previously, and we anticipate producing insights that are relevant to programming language design broadly.

8.2 Active Type Theory

The work on Ace is aimed at establishing a practical implementation of active typechecking and translation. However, because it uses a dynamically-typed language, Python, for the type-level language, it is not suitable for rigorous theoretical analysis, nor does it achieve safety in the strictest sense, due to high levels of dynamism exploitable in Python's object model. Moreover, it is a full language and so it does not necessarily distill the essential concepts that we wish to introduce to the research community in their simplest possible form or connect them to existing concepts in the theory of typed programming languages. For these reasons, we plan to develop a minimal type theoretic formulation, which we call λ_A.

8.2.1 Type-Level Computation

We have chosen to use the term *type-level language*, rather than *metalanguage* or, as it is called in the original work on active libraries, *metacode* intentionally. This is because the concept of a type-level language is already well-established and includes a key feature necessary for active types – the reification of types as compile-time entities that can be manipulated programmatically. We believe that this connection between type-level computation and active types will provide new insights into each mechanism individually, and provide a clean theoretical basis for active typechecking and translation as an extension to type-level computation.

8.2.2 Active Types and Wadler's Expression Problem

Another key insight is that in monolithic programming languages, the fixed set of base types can be written as an algebraic datatype. Indeed, in functional implementations of typecheckers, this is precisely what is done, for example for Gödel's **T**:

```
datatype Type = Arrow of Type * Type
              | Nat
```

If we wish to allow users to introduce new base types and type families, it can then be seen that we face a variant of the expression problem, so named by Wadler [9]. That is, we wish to add new constructors of kind `Type`, but this conventional formulation of datatypes does not allow us to do so. There are competing approaches that aim to solve this problem. The most common is to use object-oriented inheritance, where subtypes of `Type` represent new type families. This is the approach taken in Ace. The functional programming approach is to instead use *open data types* [4]. We will thus aim to further explore this approach, drawing a clear connection between active types, the expression problem and type-level open data types.

8.2.3 Rigorous Safety Proofs

A benefit of a simple core theory for active typing is that it will allow us to formally state and prove key safety theorems, formalizing the somewhat informal notions described in Section 3.3. Based on early sketches of a theory that we have discussed, we have implemented some run-time checks into Ace that ensure some of these properties for compiled programs based on a notion borrowed from the area of verified compilation called *type-directed compilation* [**?**]. We would like to be able to ensure these properties for all possible programs, and a type theory will allow us to demonstrate the machinery needed to do this in a rigorous manner.

8.2.4 Implementation With a Dependently-Typed Language

Finally, we would like to implement a functional version of active type-checking and translation, both as a counterpoint to Ace and as a useful tool for theoretical researchers. We plan on doing so as an extension to the Coq theorem prover, which contains a dependently-typed functional programming language at its core, and is implemented in Ocaml. This will allow us to understand the challenges and opportunities of using a dependently-typed functional language at the core of an extensible programming system.

9 Research Plan

9.1 Spring 2013

I will complete the work on Ace and active type theory and submit it for publication.

9.2 Summer 2013

I will complete the work on active code prediction and submit it for publication.

9.3 Fall 2013

I will finalize all work and write the final dissertation.

10 Conclusion

Monolithic programming systems enforce a dichotomy between *built-in constructs*, which enjoy support throughout the programming system but must be designed and implemented by the system designer in advance, and *user-defined constructs*, which can be distributed as user libraries, but must creatively combine and repurpose the small set of available built-in constructs to express all desired run-time, compile-time and edit-time behaviors. This dichotomy has forced researchers and domain experts who want to significantly advance the state-of-the-art in programming systems, particularly in compiled, statically-typed systems where this dichotomy is most salient, to design and develop new languages. But this then couples their core innovations with a collection of unrelated design decisions, requires more effort both for the experts developing the innovation and its targeted user community, and leads to intrinsic problems for users developing applications consisting of components written in different languages.

...

Todo list

References

[1] A. Clements. *A comparison of designs for extensible and extension-oriented compilers*. PhD thesis, Citeseer, 2008.

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

[3] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

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

[5] J. McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978.

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

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

[8] 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.

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