Actively-Typed Programming Systems

PhD Thesis Proposal

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

September 16, 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. Due to the increasing diversity and complexity of modern problem domains and hardware platforms, it has become clear that no small collection of general-purpose primitives backed by a simple toolchain will be able to fully satisfy these criteria in all cases. Instead, researchers will continue to provide specialized abstractions, notations, type systems, implementation strategies, optimizations, run-time systems and tool support (collectively, specialized *features*) in order to meet these growing challenges.

Ideally, providers would develop and distribute new features as libraries, so that developers could granularly choose those that best satisfy the constraints of their problem domains. Unfortunately, this is often infeasible, because libraries are nearly exclusively vehicles for specifying the run-time behaviors of a program, but providing these kinds of features requires or benefits from some measure of control over the compile-time or edit-time semantics of the programming system. That is, from the perspective of a library, the language's semantics are fixed in advance, the compiler and run-time system are "black box" implementations of these fixed semantics, and the other tools, like code editors and debuggers, operate according to domain-agnostic protocols also based exclusively on these fixed semantics. As a result, providers of new system features must take *language-external approaches*. We will argue that such approaches have several fundamental problems, and that taking them has led to an unnecessary gap between research and practice. In place of these approaches, we will advocate for *language-internal extensibility mechanisms*, and show how, by organizing new features around types and constraining them appropriately, such mechanisms can be made both safe and highly expressive.

1.1 Motivating Example: Regular Expressions

To make the issue concrete, let us begin with a simple example. *Regular expressions* are a widely-used mechanism for finding patterns in unstructured and semi-structured strings (when formal grammars are not appropriate) [**?**]. If a programming system included full compile-time and edit-time support for regular expressions, it might provide features like:

1. **built-in syntax for pattern literals** so that malformed patterns result in intelligible *compile-time* parsing errors, motivated by the frequency of run-time errors relating to pattern syntax in Java [**?**] and other languages that do not have this feature.
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 again 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.
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.

No system today has built-in support for all of the features enumerated above. Instead, libraries generally provide support for regular expressions by leveraging general-purpose constructs. Unfortunately, it is impossible to fully encode the syntax and the specialized static and dynamic semantics described above in terms of standard general-purpose notations and abstractions. Library providers have thus needed to compromise, typically by representing regular expressions using strings and deferring parsing, typechecking and translation to run-time, which 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 features, as described above, are rarely integrated into editors, and never in a way that facilitates their discovery and use directly when the developer is manipulating regular expressions. In most cases, tools must be discovered independently and accessed externally (for example, via a browser), making both their development and use less common and more awkward than necessary [**?**] (see Section 5).

1.2 Language-External Approaches

When the compile-time or edit-time semantics of a system must be modified to fully realize a new feature, as in the example above, providers often 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[1] or other tool.

Unfortunately, when providers of new features take language-external approaches, it causes problems for clients. Features cannot be adopted individually, but instead are

[1]Note that compilers that modify or allow modification of the semantics of their base language, rather than simply implementing meaning-preserving optimizations, should be considered as pernicious means for creating new languages. Many compilers, including `gcc`, `GHC` and `SML/NJ`, are guilty of this sin, meaning that some programs that seem to be written in C, Haskell or Standard ML are actually written in tool-specific derivatives of those languages. Language-internal mechanisms do not lead to such fragmentation.

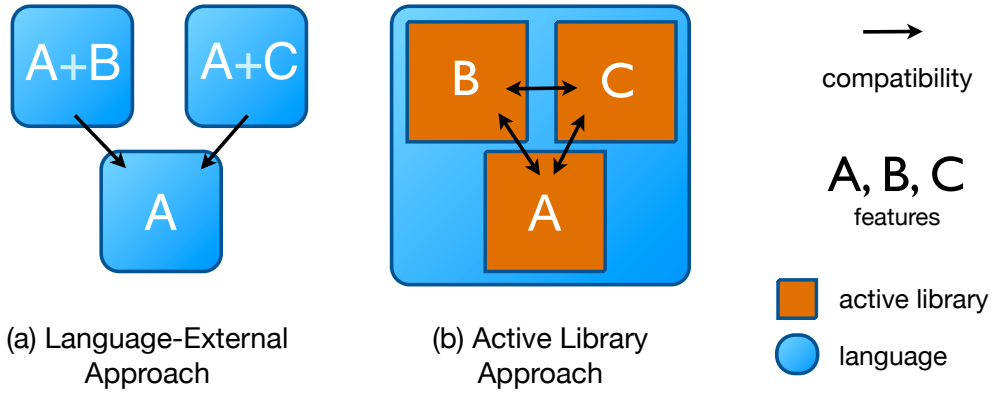

Figure 1: (a) With the language-external approach, novel constructs are packaged into separate languages. Users can only safely and naturally call into languages if the interface uses common constructs and an interoperability layer has been developed (the *interoperability problem*). (b) With the active library 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.

only available coupled to a collection of other often-unrelated features (the incidental features included in a newly-designed language or tool, or the existing features of the particular tool that was externally extended). This makes adoption more costly when these other features are either not appropriate or insufficiently developed, or when the features bundled with a different language or tool are simultaneously desirable. For example, although evidence suggests that developers prefer language-integrated parallel programming abstractions to library-based implementations if all else is equal [**?**], library-based implementations are more widely adopted because parallel programming languages privilege only a few abstractions, even though parallel programming is rarely the only concern relevant to a component or application. Regular expression support, for example, may be simultaneously desirable when processing large amounts of textual data in parallel.

Even when various concerns can be separated into different components, each written in a suitable language (often called the *language-oriented approach* to software development [**?**]), the interfaces between these components remains an issue. The specialized primitives particular to one language cannot always be safely and naturally expressed in terms of those available in another, so building a program out of components written in a variety of different languages is difficult or impossible whenever these primitives are exposed at interface boundaries. We refer to this fundamental issue as the *interoperability problem*.

One strategy often taken by programming language designers to partially address the interoperability problem is to target an established language or bytecode, such as the Java Virtual Machine (JVM) bytecode or LLVM IR, and support a superset of its constructs. Scala [**?**] and F# [**?**] are two prominent examples of general-purpose languages that have taken this approach, and many domain-specific language frameworks also rely on this strategy (e.g. Delite [**?**]). This only enables 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 semantics of the common language. If new languages only include constructs that can already be expressed safely and reasonably naturally in the common language, then this approach can work well. But many of the most innovative constructs found in modern languages (often, the features that justify their creation) are difficult to define in terms of existing

constructs in ways that guarantee all necessary invariants are statically maintained and that do not require large amounts boilerplate code and run-time overhead. For example, the type system of F# guarantees that null values cannot occur within F# data structures, but maintaining this important invariant still requires run-time checks because the typing rules of F# do not apply when F# code is called from other languages on the Common Language Infrastructure (CLI) like C#. The F# type system also includes support for checking that units of measure are used correctly [**?**], but this specialized 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.

1.3 Active Libraries

For these reasons, we argue that taking a language-external approach to realizing a new feature, by packaging it into a new or derivative language or tool, should be considered harmful and avoided whenever possible. The goal of the research being 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 defined externally and thus, in the case of popular languages, become available only subject to the approval of slow-moving governing committees[2]. 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 by the system during development or compilation to provide new domain-specific features [**?**]. Features implemented within active libraries are imported as needed by the 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 developers, the system may become unreliable. Type safety, for example, may not hold if the static and dynamic semantics of the language can be modified or extended arbitrarily from within libraries. Furthermore, even if extensions can be shown not to cause such problems in isolation, there may still be conflicts between extensions that could weaken their semantics and 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.

1.3.1 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 by being able to influence the programming system at compile-time or edit-time, including high-level program optimization, checking programs for correctness against

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

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, partly for this reason).

The first concrete realizations of active libraries, prompting the introduction of the term, were libraries that performed domain-specific program optimization at compile-time by exploiting language mechanisms that allow for limited compile-time computation. A prominent example in the literature is Blitz++, a library that uses C++ template metaprogramming to optimize compound operations on vectors and matrices by eliminating intermediate allocations [**?**]. Although this and several other interesting optimizations have been shown possible by this technique, its expressiveness is fundamentally limited because template expansion allow 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.

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. If SugarJ was used by two different regular expression engines to provide literal syntax for regular expression patterns, there could easily be conflicts at link-time because both will introduce many of the same standard notations but back them with differing implementations. And again, it can be difficult for users to predict what type of data an unfamiliar piece of syntax desugars into, leading to difficulties reading and reasoning about code.

2 Active Types

The 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, but without allowing the kinds of safety violations possible when using previous mechanisms. To motivate the approach we will take to achieve this, let us return to the example of regular expressions. Observe that each of the features described in Section 1.1 relates specifically to how terms representing regular expression patterns, which will always have a consistent user-defined type, let's call it `Pattern`[3], should be processed by various tools. It is only when creating, editing or compiling expressions of type `Pattern` that the logic enumerated in Section 1.1 would ever need to be invoked.

better transition

Indeed, this is not a coincidence of our example, but a commonly-seen pattern. Types are already known to be a natural organizational unit around which the semantics of programming languages and logics can be defined. For example, in both TAPL [6] and PFPL [**?**], most chapters simply describe the semantics and metatheory of a few new types without reference to other types. In these descriptions, the composition of these types and associated operators into languages is a metatheoretic (that is, language-external) notion. For example, in PFPL, the notation L($\to$ `nat dyn`) represents a language composed of the arrow ($\to$), `nat` and `dyn` types and their associated operators, each group of which is defined in separate chapters. This organization suggests a principled language-internal alternative to the mechanisms described in Section 1.3.1 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 operations on expressions in that type or type family. We call types with such logic associated with them *active types* and systems that support them *actively-typed programming systems*. By constraining the extension logic itself by various means, including with its own type system and by applying techniques from the compiler correctness literature, we can ensure that the system as a whole maintains important safety properties.

use Bob's fancy L

2.1 Proposed Contributions

In Section 3, 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 this work is on high-performance computing 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. Many kinds of syntax extensions involve developing special literal forms for some type or family of types, including the examples of regular expression patterns, HTML and XML mentioned above. In Section 4, we will show how such domain-specific syntax can 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 programming language. Our novel contributions include the design of a second parsing phase for processing "TSL literals"

clean up Wyvern contributions

[3]More generally, it may be 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. We should also note that to properly prevent conflicts, naming conflicts must also be avoided by the use of an appropriate qualified name for this type. Suitable namespacing mechanisms are already well-developed and will be assumed in this thesis.

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, a recursive use of the active parsing technique to introduce a declarative syntax for defining actively-typed grammars, and an elegant underlying general-purpose mechanism for associating compile-time data and functionality with structural types. We will describe each of these contributions as implemented in Wyvern, develop minimal formal systems showing how the syntax and semantics of Wyvern supports actively-typed parsing, and examine the expressiveness of this technique for introducing novel literal forms, as well as for use cases beyond literal forms that can also be expressed by creative application of this technique.

Finally, in Section 5, 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 mechanism, and develop one such system for Java[4], based on these constraints, called Graphite. Using Graphite, we implement a palette for working with regular expressions in order to conduct a pilot study that provides evidence for the usefulness of this approach, and of contextually-invoked editor-integrated tools generally.

Taken together, these mechanisms demonstrate that actively-typed mechanisms can be introduced throughout a programming system to allow users to extend both its compile-time and edit-time semantics from within libraries, without weakening the metatheoretic guarantees that the system provides. They also further demonstrate that types are a natural organizational unit for defining programming system semantics, because a great variety of features can be expressed in an actively-typed manner, and doing so guarantees that the features will be safely composable in any combination. In this thesis, each mechanism will be implemented within a different programming system, showing that actively-typed mechanisms are relevant across traditional paradigms, including functional languages (@λ), class-based OO languages (Graphite), structurally-typed languages (Wyvern) and scripting languages (Ace). In the future, we anticipate that mechanisms based on those in this thesis will be brought together into a single highly-extensible system with a minimal, well-specified core, where nearly every feature is distributed within a library.

3 Active Typechecking and Translation

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

fix umlaut on Godel

figure with statics and dynamics of Godel's T + sums + products?

The only recourse researchers have in such situations is to attempt to encode new constructs in terms of existing constructs. Such encodings, collections of which are often called *embedded domain-specific languages (DSLs)* [2], must creatively combine the constructs available in the host "general-purpose" language. Unfortunately, such encodings are at times impossible, and even if possible, often impractical and unintuitive. For our example

[4]In Graphite, palettes are associated with Java classes, which serve many of the same purposes as types do in other languages. Active code completion could be implemented just as well in non-object-oriented systems.

of adding products or sums to Godel's T, a full encoding is impossible. Weak Church encodings are possible [?], but they require a reasonable level of creativity[5]. Moreover, because sums and products are represented directly as lambda terms, they cannot be distinguished from other lambda terms and so they will not offer the same static safety guarantees as a primitive encoding, nor will they obey the same universal laws. They are also likely to incur performance overhead by their use of closures rather than a more direct and better optimized internal representation.

mention the stuff Bob mentioned

This is not only a problem for simple languages like Godel's T. Several embedded DSLs for Haskell have also needed to make significant compromises at times. For example...

find a good example.

An internally-extensible programming language could address these problems by providing a language mechanism for extending, directly, its static and dynamic semantics, so as to support domain-specific type systems and implementation strategies in libraries. But a significant challenge must be addressed before introducing such a mechanism into a language: balancing expressiveness with concerns about maintaining safety properties in the presence of arbitrary combinations of user-defined extensions. Correctness properties of an extension itself should be modularly verifiable, so that its users can rely on it for verifying and compiling their own code. The mechanism must also ensure that desirable metatheoretic properties and global safety guarantees of the language cannot be weakened by extensions. And with multiple independently-developed extensions used at once, the mechanism must further guarantee that they cannot interfere with one another.

Scala/Delite examples?

better transition here

3.1 Theory

In this section, we will describe a minimal calculus supporting a language-internal extensibility mechanism called *active typechecking and translation (AT&T)* that allows developers to specify new primitive type families, associate operators with them, and implement their static semantics as well as their dynamic semantics by translation to a typed internal language. The AT&T mechanism utilizes type-level computation of higher kind and integrates typed compilation techniques into the language to provide strong safety guarantees, while remaining straightforward and expressive.

3.1.1 From Externally-Extensible Implementations to Internally-Extensible Languages

To understand the genesis of our internal extensibility mechanism, it is helpful to begin by considering why most implementations of programming languages are not even *externally extensible*. Let us consider again Godel's T. In a functional implementation, its types and operators will invariably be represented using closed datatypes. For example, a simple implementation in Standard ML may be based around these datatypes:

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

The logic governing the typechecking phase as well as the initial compilation phase (we call this phase *translation* to distinguish it from subsequent optimization phases) will proceed by exhaustive case analysis. In an object-oriented implementation of Godel's T, we could instead represent types and operators as instances of subclasses of abstract classes `Type` and `Exp`. If typechecking and translation then proceed by the ubiquitous *visitor pattern* [?], by dispatching against a fixed collection of known subclasses of `Exp`, the same basic issue is encountered: there is no modular way to add new primitive types or operators

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

to the implementation. Indeed, this basic issue is the canonical example of the widely-discussed *expression problem* [**?**].

A number of language mechanisms have been proposed that allow new cases to be added to datatypes and functions in a modular way. In functional languages, these are known as *open datatypes* [**?**]. The language might allow you to add types and operators for products to an open variant of the above definitions like this:

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

The corresponding logic for functionality like typechecking and translation can then be specified for only the new cases, e.g.:

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

If we allow users to define new modules containing definitions like these and link them into our compiler, we have succeeded in creating an externally-extensible language implementation, albeit one where safety is not guaranteed (we will return to this point shortly). We have not, however, created an extensible programming language, because other implementations of the language will not necessarily support the same mechanism. If our newly-introduced constructs are used at library interface boundaries, we introduce the same compatibility problems that developing a new standalone language can create. That is, **extending a language by extending a single implementation of it is equivalent to creating a new language**. Several prominent language ecosystems today are in a state where a prominent compiler has introduced extensions that many libraries have come to rely on, including the Glasgow Haskell Compiler and the GNU compilers for C and C++. We argue that this practice should be considered harmful.

A more appropriate and useful place for extensions like this is directly within libraries. To enable such use cases, the language must provide a mechanism that allows expert users to declare new type families, like `Prod`, their associated operators, like `Pair`, `PrL` and `PrR`, and their associated typechecking and translation logic. When a compiler encounters these declarations, it adds them to its internal representation of types and operators, as if they had been primitives of the language, so that when user-defined operators are used in expressions, the compiler can temporarily hand control over the typechecking and translation phases to them. Because the mechanism is language-internal, all compilers must support it to satisfy the language specification. Thus, library developers can use new primitive constructs at external interfaces more freely.

Statically-typed languages typically make a distinction between the expression language, where run-time logic is expressed, and the type-level language where, for example, datatypes and type aliases are statically declared. The description above suggests we may now need another layer in our language, an extension language, where users provide extension specifications. In fact, we will show that **the natural place for type system extensions is within the type-level language**. The intuition is that extensions will need to introduce and statically manipulate types and type families. Many languages already support notions of *type-level computation* where types are manipulated as values at compile-time (see Sec. **??** for examples of such languages). These are precisely the properties that such an extension language would have, so we unify the two.

3.2 Ace

4 Actively-Typed Parsing

Wyvern...

5 Active Code Completion

6 Timeline

6.1 Fall 2013

I will complete the work on Ace (ESOP - Oct.) and active type theory (PLDI - Nov.) and Wyvern (ECOOP - Dec.) and submit each for publication.

Ace: - Finish implementation - Finish OpenCL implementation - Finish C implementation - Implement global arrays, functional data parallelism, objects

Theory: - Prove metatheory (mechanized?) - Figure out composition theorem - Figure out how it would look in Coq

Wyvern: - Figure out grammar and type system formalism - See implementation through - Declarative stuff - Write it up

Graphite: - Figure out safety stuff

6.2 Spring 2014

I will finalize all work and write the final dissertation.

7 Conclusion

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.