Type-Oriented Foundations for Extensible Programming Systems

PhD Thesis Proposal

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

January 10, 2014

Abstract

We propose a thesis defending the following statement:

Active types allow providers to extend a programming system with new compile-time and edit-time features from within libraries in a safe and expressive manner.

1 Motivation

Specifying and implementing a programming language together with its supporting tools (collectively, a *programming system*) that has sound theoretical foundations, helps users identify and fix errors as early as possible, supports a natural programming style, and performs well across diverse problem domains and hardware platforms remains a grand challenge in computing. Due to the increasing diversity and complexity of modern problem domains and hardware platforms, it has become clear that no small collection of general-purpose constructs backed by a simple, all-purpose implementation can fully satisfy these criteria in all cases. Instead, researchers and domain experts (collectively, *providers*) continue to develop specialized notations, static semantics, dynamic semantics, implementation strategies, optimizations, run-time systems and tools (collectively, *features*) designed to address these modern challenges in different circumstances.

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

1.1 Motivating Example: Regular Expressions

To make the issue concrete, let us begin with a simple example. *Regular expressions* are a widely-used mechanism for finding patterns in semi-structured strings (e.g. DNA sequences) [21]. A programming system that included full compile-time and edit-time support for regular expressions might simultaneously provide features like these:

1. **Built-in syntax for pattern literals** (e.g. [2]) so that malformed patterns result in intelligible *compile-time* parsing errors. In languages lacking such literals, run-time errors relating to pattern syntax can occur, even in well-tested, deployed code [18].

2. **Typechecking logic** that ensures that key constraints related to regular expressions are not violated at compile-time:

   (a) only appropriate values are spliced into regular expressions, to avoid splicing errors and injection attacks [3]

   (b) out-of-bounds backreferences are not used [18]

   (c) strings known to be in the language of a regular expression are given a type that tracks this information and ensures that string manipulation operations do not inadvertently lead to a string that is not in the assumed language [9].

   When a type error is found, an intelligible error message is provided.

3. **Elaboration logic** that partially or fully compiles known regular expressions into the efficient internal representation that will be used by the regular expression matching engine (e.g., a finite automata [21]) ahead of time. In most languages, this compilation step occurs at run-time, even if the pattern is fully known at compile-time, thereby introducing performance overhead into programs. If the developer is not careful to cache compiled representations, regular expressions used repeatedly in a program might be needlessly re-compiled on each use.

4. **Editor services** for quickly testing regular expression patterns against test strings, referring to documentation or searching databases of common patterns (e.g. [1]).

No system today has built-in support for all of the features enumerated above. Instead, libraries generally provide support for regular expressions by leveraging general-purpose constructs. Unfortunately, it is impossible to fully encode the syntax and the specialized static and dynamic semantics described above in terms of general-purpose notations and abstractions like objects or inductive datatypes. Library providers have thus needed to compromise, typically by asking clients to provide regular expressions as strings, thereby deferring parsing, typechecking and elaboration to run-time, which introduces performance overhead and can lead to unanticipated run-time errors (as shown in [18]) and security vulnerabilities (due to injection attacks when user inputs are spliced into patterns, for example). Similarly, useful tools for working with regular expressions are rarely integrated into editors, and even more rarely in a way that facilitates their discovery and use directly when the developer is manipulating regular expressions. Tools that must be discovered independently and accessed externally are used less frequently and can be more awkward than necessary [5, 14], leading to lower productivity [14].

1.2 Language-External Approaches

When the semantics or implementation of a system must be extended to fully realize a new feature, as in the example above, providers typically take a *language-external approach*, either by developing a new or derivative programming system (supported by *language workbenches* [7], *DSL frameworks* [8] or *compiler generators*), or by extending an existing system by some mechanism that is not part of the language itself, such as an extension

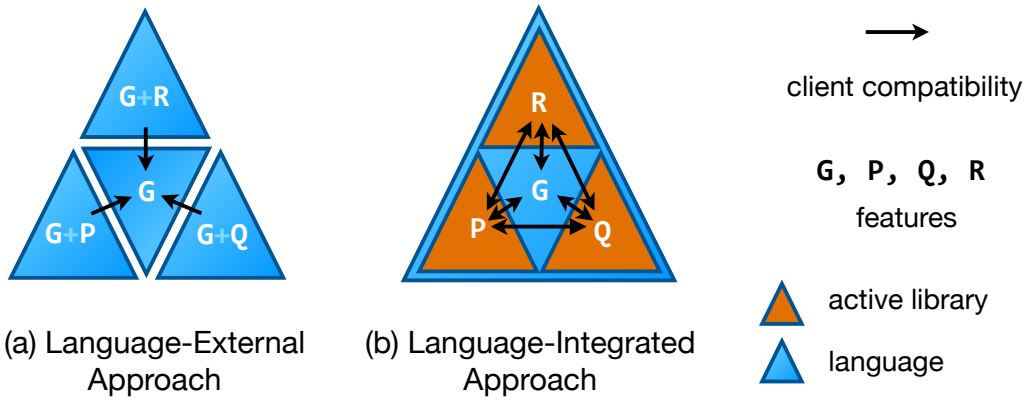

Figure 1: (a) When taking a language-external approach, new features are packaged together into separate languages and tools, causing problems with compatibility (described in the text). (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time logic governing new constructs is expressed within "active" libraries. It is critical that the extension mechanism guarantees that extensions cannot interfere with one another, so that the compatibility problem is avoided.

mechanism for a particular compiler[1] or other tool. For example, a researcher interested in providing regexp-related features (let us refer to these collectively as `R`) might design a new language with built-in support for regular expressions, perhaps basing it on an existing language (containing some general-purpose constructs, `G`). A different researcher developing a new language-based parallel programming abstraction or implementation strategy (collectively, `P`) might take the same strategy. A third researcher, developing an alternative parallel programming abstraction (collectively, `Q`) might again do the same. This results in a collection of distinct systems as diagrammed in Figure 1a.

Unfortunately, when providers of new features take language-external approaches like this, it causes problems for clients. Features cannot be adopted individually, but instead are only available coupled to a collection of other unrelated features (e.g. the incidental features of the newly-designed language and its associated tools). This makes adoption more costly when these incidental features are not desirable or insufficiently developed, or when the features bundled with a different language or tool are simultaneously desirable. Evidence supports the prominence of this problem: developers prefer language-integrated parallel programming abstractions to library-based implementations if all else is equal [6], but library-based implementations are more widely adopted because "parallel programming languages" privilege only a few chosen abstractions and implementation strategies. This is problematic because different parallel abstractions or implementation strategies are more appropriate in different situations [20] and moreover, parallel programming support is rarely the only concern relevant to client developers outside of a classroom setting. Regular expression support, for example, may be simultaneously desirable for processing large amounts of textual data in parallel, but using these features together in the same compilation unit when they have been implemented by language-external means is difficult or impossible. One must either use the system containing features `G+R` or `G+P`. There is no system containing both.

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

Different application concerns can sometimes be separated into different components, each written using a suitable language and tooling. This has been called the *language-oriented approach* to software development [22]. Unfortunately, this is insufficient: the interface between components written in different languages remains an issue. The specialized constructs particular to one language (e.g. futures in a parallel programming language) cannot always be safely and naturally expressed in terms of those available in another (e.g. a general-purpose language), so building a program out of components written in a variety of different languages is difficult whenever these are exposed at interface boundaries. Tool support is also lost when calling into a different language. We call this fundamental issue the *client compatibility problem*: clients of a certain collection of features cannot always interface with clients of a different collection of features in a safe, performant and natural manner.

One strategy often taken by proponents of the language-oriented approach to partially address the compatibility problem is to target an established intermediate language, such as the Java Virtual Machine (JVM) bytecode, and support a superset of its constructs. Scala [13] and F# [15] are examples of prominent general-purpose languages that have taken this approach, and most DSL frameworks also rely on this strategy. As indicated in Figure 1a, this only enables true client compatibility in one direction. While calling into the common language becomes straightforward, calls in the other direction, or between the languages sharing the common target, are still restricted by the semantics of the common language.

If new languages only include constructs that can already be expressed safely and reasonably naturally in the common language, then this approach can work well. But many of the most innovative constructs found in modern languages (often, the features that justify their creation) are difficult to define in terms of existing constructs in ways that guarantee all necessary invariants are statically maintained and that do not require large amounts boilerplate code and run-time overhead. For example, the type system of F# guarantees that null values cannot occur within F# data structures, but maintaining this important invariant still requires run-time checks because the typing rules of F# do not apply when F# code is called from other languages on the Common Language Infrastructure (CLI) like C#. The F# type system also includes support for checking that units of measure are used correctly [19, 11], but this specialized invariant is left completely unchecked at language boundaries. In some cases, desirable features must be omitted entirely due to concerns about interoperability. F#, for example, is based on Ocaml, but due to the need for bidirectional interoperability with CLI languages, it cannot support advanced features like polymorphic variants, modules or functors [12].

1.3 Language-Integrated Approaches

We argue that, due to these compatibility problems, taking a language-external approach to realizing a new feature should be considered harmful and avoided whenever possible. The goal of the research being proposed here is to design *language-integrated extension mechanisms* that give developers control over edit-time and compile-time behaviors that have previously been defined centrally[2] so that such language-external approaches are less frequently necessary. More specifically, we will show how control over **parsing**, **typechecking**, **elaboration** 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 clients of features already available in the system, they contain logic invoked by the system during development or compilation to provide new features [**?**]. Features implemented within active libraries are imported as needed by the clients of libraries that rely on them, unlike features implemented by language-external means, avoiding the compatibility problem.

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

We must proceed with caution, however: critical issues having to do with safety must be overcome before 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 to design language-integrated extensibility mechanisms.

1.3.1 Background: Active Libraries

The term *active libraries* was first introduced 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 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 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 [17].

More powerful compile-time *term rewriting mechanisms* integrated into some languages can also be used for optimization, as well as for introducing specialized error checking and reporting logic and extending the language with new abstractions. These mechanisms are highly expressive because they allow users to programmatically manipulate syntax trees directly, but they suffer from problems of composability and safety. Compile-time macros, such as those in MetaML [**?**], Template Haskell [**?**] and Scala [4], can take full control over all the code that they enclose, so outer macros can neglect to invoke or manipulate the output of inner macros, sometimes inadvertantly. Once a value escapes a macro's scope, there is no way to rely on the guarantees and features that were available within its scope, because the output of a macro is simply a value in the underlying language (a problem fundamentally related to the compatibility problem described in Section 1). Thus, macros can be used to automate code generation, but not to truly extend the semantics of a language. It can also be difficult to reason about the semantics of code when any number of enclosing macros may be manipulating it, and to build tools that can handle them.

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 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 activated within, so these mechanisms are highly expressive and avoid some of the difficulties of block-scoped macros. But libraries containing such global rewriting logic cannot be safely composed because two different libraries may attempt to rewrite the same piece of code differently. It is also difficult to guarantee that such logic is correct and difficult 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 [**?**] and other languages in the Sugar* family [**?**], like SugarHaskell. These languages permit libraries to extend the base syntax of the core language in a nearly-arbitrary manner, and these extensions are imported transitively throughout a program. These extensions are also not safely composable. For example, a library that defines a literal syntax for HTML would conflict with another that defines a literal syntax for XML because they define differing semantics for some of the same syntactic forms. If SugarJ was used by two different regular expression engines to provide literal syntax for regular expression patterns, there could easily be conflicts at link-time because both will introduce many of the same standard notations but back them with differing implementations. And again, it can be difficult for users to predict what type of data an unfamiliar piece of syntax desugars into, leading to difficulties reading and reasoning about code.

2 Active Types

The mechanisms that we will introduce in this thesis are designed to be highly expressive, permitting library-based implementations of features comparable to built-in features found in modern programming systems, but without allowing the kinds of safety violations possible by previous mechanisms, such as those described above.

To motivate the basic approach we will take in designing each mechanism to achieve these goals, let us return to the example of regular expressions. Observe that each feature described in Section 1.1 relates specifically to how terms representing regular expression patterns – that is, terms classified by a single user-defined type[3], `Pattern`[4] – should be processed by various tools. It is exclusively when editing or compiling expressions of type `Pattern` that the logic enumerated in Section 1.1 would need to be invoked.

Indeed, this is not a property unique to our chosen example, but a commonly-seen pattern. Types are already known to be a natural concept around which the semantics of programming languages and logics can be organized. For example, in both TAPL [16] and PFPL [10], most chapters simply describe the semantics and metatheory of a few new types without reference to other types. The composition of these types and associated operators into languages is a metatheoretic (in other words, language-external) notion. For example, in PFPL, the notation $\mathcal{L}\{\rightarrow \texttt{nat}\ \texttt{dyn}\}$ represents a language composed of the arrow ($\rightarrow$), `nat` and `dyn` types and their associated operators. These are defined in separate chapters, and it is left unstated that the semantics and metatheory developed separately will compose without trouble (and indeed, almost any combination of types defined separately in PFPL can be combined in a straightforward manner).

This type-centered 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*

[3]More generally, it may be several different types within an indexed type family. For example, `Pattern`(n) may be the type of 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.

[4]We should note at the outset that to fully prevent conflicts between libraries, naming conflicts must also be avoided. Suitable namespacing mechanisms (e.g. [**?**]) are already well-developed and will be assumed.

systems. By constraining the extension logic itself by various means we can ensure that the system as a whole maintains important safety properties.

2.1 Proposed Contributions

In Section 3, we will describe how to extend the concrete 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 along with a formally definition of its compiler, proving several key safety theorems, and examining the connections between active types and several well-defined prior notions, including type-level computation, type abstraction and typed compilation. We will then go on to demonstrate the expressiveness of this mechanism by designing and implementing a full-scale language, Ace, showing how it relates to the core calculus, and implementing a number of interesting semantic constructs 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 demonstrate how a broad variety of other primitive abstractions can coexist within Ace.

In both @λ and Ace, user-defined semantic extensions operate over a fixed syntax. Specialized syntax that makes code more concise or readable can have a significant impact on productivity, however [**?**]. Many kinds of specialized 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 type-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. We will describe several unique mechanisms, implemented in Wyvern, that ensure that syntax extensions are maximally flexible while remaining safely composable, 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 that do not fit the conventional mold of a "literal" but that can nevertheless be expressed by creative application of this technique.

Finally, in Section 5, we will show how editor-integrated domain-specific tooling for instantiating 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[5], 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

[5]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.

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

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

Figure 2: Static semantics of Gödel's T

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

Figure 3: Static semantics of products

(@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, Gödel's T [10], a typed lambda calculus with recursion on primitive natural numbers (Figure 2). Although a researcher may casually speak of "extending Gödel's T with primitive product types" (Figure 3), modularly adding this new primitive type family and its corresponding operators to this language from within is impossible. That is, Gödel's T is not *internally extensible*.

The only recourse researchers have in such situations is to attempt to define new constructs in terms of existing constructs. Such encodings, collections of which are often called *embedded domain-specific languages (DSLs)* [8], must creatively combine the constructs available in the host "general-purpose" language. Unfortunately, such encodings can be difficult to conceive of and impractical or impossible in some cases. For our example of adding products to Gödel's T, a full encoding is impossible. Limited forms of Church encodings are possible, where products are represented by lambda terms, but they require a reasonable level of creativity[6] and in doing so, the type system cannot enforce a distinction between product types and the function types they are encoded using. It is also more difficult to reason about products represented in this way (for example, they will not have unique canonical forms in the same situations). This strategy will also likely incur a performance penalty because it uses closures rather than a more direct and better optimized internal representation.

cite the stuff Bob mentioned on universality

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

Scala/Delite examples? Haskell GPU thing?

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

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

3.1 Theory

In this section, we will describe a minimal calculus that captures our language-internal extensibility mechanism, called *active typechecking and translation (AT&T)*. AT&T allows developers to declare new primitive type families, associate operators with them, implement their static semantics in a functional style, and realize their dynamic semantics by simultaneously implementing a translation into a typed internal language. Note that this latter mechanism is closely related to how Standard ML was (re-)specified, but that we are fundamentally interested in extending language *implementations*, not their declarative specifications; proving the adequacy of such implementations against mechanized specifications, or extracting them directly from such specifications, will be investigated in future work.

cite/read a bit about this/ask Bob/flesh this out

The AT&T mechanism utilizes type-level computation of higher kind and integrates typed compilation techniques into the language to allow us to give strong metatheoretic guarantees, and uses a mechanism notionally related to abstract types (such as those found in the ML module system) to guarantee that extensions cannot interfere with one another, while remaining straightforward and expressive. In this section, we will develop a core calculus, called @λ, which uses a minimal, uniform grammar for primitive operators. Then in Section 3.2, we will show how to realize this minimal mechanism within a widely-used language with a more expressive grammar.

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 Gödel'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:

```
datatype Type = Nat | Arrow of Type * Type
datatype Exp = Var of var
             | Lam of var * Type * Exp
             | Ap of Exp * Exp
             | Z | S of Exp
             | 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 Gödel'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 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:

```
newcase Prod of Type * Type extends Type
newcase Pair of Exp * Exp extends Exp
newcase PrL of Exp extends Exp
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.:

```
typeof PrL(e) =
  case typeof e of
```

```
    Prod(t1, _) => t1
  | _ => 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...

4.1 TODO

- Figure out separate compilation (e.g. of CSS)

5 Active Code Completion

Murphy-Hill and Murphy, CSCW paper on peer interaction to enable discovery of tools

Improving program navigation with an active help system. CASCON 2010.

Improving software engineer's fluency by recommending dev environment commands. FSE 2012.

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

[2] perlre - Perl regular expressions. `http://perldoc.perl.org/perlre.html`.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

[12] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2013.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

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