Actively-Typed Programming Systems

Thesis Proposal

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

August 26, 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 practical manner.

1 Introduction

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. No small, fixed collection of primitive abstractions and tools has been found to fully satisfy these criteria in all situations. Rather, researchers and domain experts continue to design and implement novel abstractions, provide alternative implementations of existing abstractions, and supplement languages with new tools in order to better satisfy these criteria within the constraints of different problem domains.

To realize a new abstraction or system behavior, such experts can choose either a *language-internal approach*, where they work within an existing language and distribute their solutions as libraries, or a *language-external approach*, where they either derive a new programming system or extend an existing system by some mechanism that is not part of the language itself, such as an extension mechanism supported by a particular compiler, editor or other tool.

When possible, taking the *language-internal approach* is significantly simpler and more practical. If an abstraction or system behavior can be realized by creatively repurposing existing language constructs, then providers and clients face fewer barriers to adoption. Unfortunately, this is very often *not* possible today because from the perspective of library code, the programming system is a largely inflexible, *monolithic* entity. That is, the language's syntax and its static and dynamic semantics are fixed in advance, the compiler is a "black box" implementation of these fixed semantics, and other tools operate according to domain-agnostic protocols that use only the basic structure of the program as input. To supplement any of these components of the programming system, experts must step out of the language, and often out of the programming system entirely.

In these situations, experts must take a *language-external approach*, often by developing and distributing new constructs together with a new language and developing an array of necessary tools. This increases the development burden on these experts, although tools

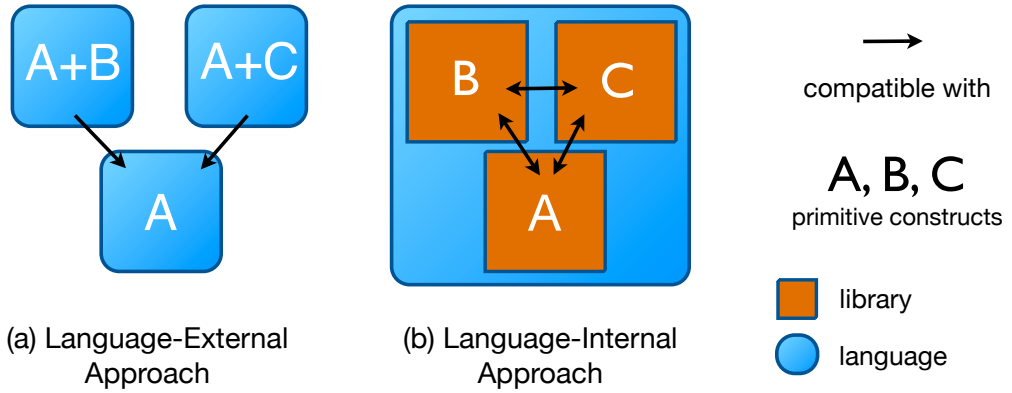

Figure 1: (a) With the language-external approach, novel constructs are packaged into separate languages. Users can only safely and naturally call into languages if the interface uses common constructs and an interoperability layer has been developed (the *interoperability problem*). (b) With the language-internal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

like compiler generators and language workbenches have started to simplify this process, and also requires experts to couple their innovations with a collection of other unrelated design choices, making adoption more difficult for clients.

A more fundamental limitation of the language-external approach arises at the interface between languages. The specialized constructs particular to one language cannot always be safely and naturally expressed in another, so building a program out of components written in different languages, when such constructs are used at the interface boundaries, can be difficult. We refer to this as the *interoperability problem*. One increasingly common strategy taken by language designers to partially address the interoperability problem is to target an established language and runtime, such as the Java Virtual Machine (JVM), and support a superset of its features. Scala [**?**] and F# [**?**] are examples of languages that have taken this approach. This addresses interoperability in only one direction (Figure 1a). While calling into the common language becomes straightforward, calls in the other direction, or between languages sharing the common target, are still restricted by the constructs available in the common language. If interoperability in both directions is desired, the new languages must only include constructs that can be expressed safely and naturally in the target language. More novel innovations, however, are often difficult to define in terms of familiar forms in ways that guarantee all necessary invariants are maintained and that are reasonably natural for users of other languages. In F#, for example, the type system guarantees that null values cannot occur, but this invariant is not maintained when calling from other languages. The F# type system also includes support for units of measure [**?**], but such domain-specific invariants cannot be guaranteed when calling into F# from other CLI languages. Similarly, interfaces built using Scala traits can be difficult to implement from Java or other JVM languages. In some cases, features are omitted entirely due to this requirement. For example, the module system in F# is significantly limited relative to the module system in its predecessor, OCaml, due to the need for bidirectional interoperability between other languages on the common language infrastructure (CLI).

For these reasons, we argue that language-external approaches should be considered harmful. 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* mechanisms that give developers control, from within the language itself, over edit-time and compile-time behaviors that are currently centrally controlled by language and tool designers and design committees[1]. In particular, we will describe how parsing, typechecking, translation and code completion can all be influenced by user-defined constructs in ways that preserve the safety of the language as a whole and admit strong composition principles .

The extension mechanism must be expressive enough to allow users to associate rich run-time, compile-time and edit-time behaviors with user constructs directly, while being sufficiently restrictive to maintain the global safety properties of the language and system as a whole, and to ensure that constructs cannot interfere with one another. This explicit consideration of the full programming system, and the tension between expressiveness and safety, are key aspects of our approach.

describe what this looks like - libraries + reference Fig 1b.

elaborate on safety requirements, merge with next paragraph

2 Approach

The mechanism we propose builds upon and refines the notion of active libraries described above by focusing on user-defined types as the primary constructs around which new system behaviors are defined. More specifically, users specify first-class behavior by equipping a type declaration with executable specifications, in the form of compile-time functions, of new behaviors. These are selectively invoked by tools, including the type checker, compiler and components of the code editor, to provide information as they work with expressions of that type (but not necessarily to handle the expression directly). We refer to such types as *active types*, and to a system designed around this extensibility mechanism as an *actively-typed programming system*, due to this connection with active libraries.

describe active libraries

This mechanism expresses a fundamental *inversion of control* relative to monolithic programming systems. Rather than centralizing behaviors within the tools themselves, the tools instead selectively delegate control to the relevant user-provided function according to some well-defined protocol. These functions are responsible for implementing the domain-specific aspects of the behavior. This leads to a system design where the distinction between first-class constructs and user-defined constructs is far less distinct.

2.1 Example: First-Class Natural Numbers

To make our approach more concrete and emphasize its foundational character, let us begin with a very simple example, first-class natural numbers. If a programming system included first-class support for natural numbers, it might support behaviors like the following:

1. type checking rules that constrain uses of the three operators associated with the **nat** type, as defined in Gödel's System **T** [6]: **z**, **s**(e) and **natrec**(e_1, e_2, e_3)
2. type checker error messages that provide domain-specific assistance, like detecting when the trailing **z** in a constant has been omitted:

```
s(s(s))
    ^ (Line 1, Character 5)
Error: Operator s requires 1 argument of type nat. Provided 0 arguments.
Did you mean s(z)?
```

3. translation into an efficient integer representation at run-time
4. syntax support

phrasing of his

phrasing / merge this into 1?

elaborate on syntax support for nat literals

[1] To be a bit piquant, one might compare today's monolithic programming languages to isolationist centrally-planned economies, whereas extensible languages more closely resemble modern market economies. We leave fleshing out such analogies to the reader.

5. editor support, such as the ability to instantiate nat constants more easily

elaborate on this / reference Graphite section

In a conventional monolithic programming system, each of these behaviors would need to be specified beforehand by the system designer. In the absence of such support, a library-based implementation of natural numbers would need to use a more general language mechanism (such as algebraic datatypes or an object system) to simulate the desired behavior. In this case, the type checking behavior of natural numbers would likely be supported, but the remaining behaviors would be more difficult to simulate.

Each of these behaviors relate specifically to expressions of type **nat**. This suggests that a natural place where these behaviors can be defined is with the declaration of the type itself, rather than centrally in the core of the language and tool implementations. An *actively-typed* declaration of **nat** would thus be equipped with functions, written using an appropriately constrained type-level language, that described how the type checker (items 1 and 2 above), compiler (item 3) and editor (item 4) should operate when encountering expressions of type **nat**. We can abstractly denote this declaration, as it would exist within a user-defined library, as follows:

$$\text{type } \mathbf{nat}[f_{\text{editor}}]\{\mathbf{z}[f_{\text{resolve-z}}, f_{\text{compile-z}}], \mathbf{s}[f_{\text{resolve-s}}, f_{\text{compile-s}}], \mathbf{natrec}[f_{\text{resolve-rec}}, f_{\text{compile-rec}}]\}$$

When type checking an expression like $\mathbf{s}(\mathbf{s}(\mathbf{z}))$, the type checker delegates to the user-provided type-level function $f_{\text{resolve-s}}$. This function would be tasked with assigning a type to expression as a whole, given information including the *types* (but not necessarily the full syntax trees) of all its subexpressions, or if a type cannot be assigned, producing a specific error message. Similarly, the compiler calls the $f_{\text{compile-s}}$ function to determine a representation in the target language for the expression, checking to ensure that it is well-formed and type-correct with respect to the target type system. Finally, elements of the editor may call into the f_{editor} function (or one of several such functions, more generally) to control behaviors like code completion and code prediction when an expression of type **nat** is being entered.

Note that these functions are *not* to be conflated with methods or run-time functions – they are functions written in a type-level language that are called at compile-time and edit-time to define the basic behaviors associated with the type that they are associated with.

2.2 Characteristics of an Actively-Typed Programming System

An actively-typed programming system can be characterized by its choice of type-level language, source grammar, target language and dispatch protocols.

Type-Level Language The type-level language is the language within which the type definitions and the functions that define their behaviors are defined. This language must be constrained so that different definitions do not interfere with one another and so that desirable safety properties for the system as a whole are maintained, as we discuss below.

Source Language The source language is the language with which run-time behavior is defined. In our example above, terms like $\mathbf{s}(\mathbf{s}(\mathbf{z}))$ are part of the source language. In the purest case, the source language is simply a grammar; its semantics are given entirely by active type specifications.

Dispatch Protocol For each syntactic form in the source language, there is a dispatch protocol that determines which type is delegated responsibility over it, and which specific function(s) are called for each behavior the system supports. This fixed protocol makes it possible for users to predict the meaning of a construct using information local to the term, a key differentiator of this approach compared to term-rewriting systems where there can be action at a distance.

Target Language The target language is the language that the front-end compilation phase of the system targets. The limitations and constraints imposed by the target language are final, because all constructs ultimately translate into terms in the target language. In other words, active type specifications can only add additional invariants to the language; they cannot violate invariants imposed by the target language.

2.3 Research Challenges

The example of natural numbers given above is relatively simple, and the solution we outline remains abstract. A key challenge is then to demonstrate that this approach is able to express the behaviors of more sophisticated language constructs that span diverse problem domains, and be implemented in the context of a realistic collection of tools. The resulting system should be usable by developers who lack the expertise needed to define new language constructs themselves.

Simultaneously, we must also demonstrate that this model is well-motivated theoretically, place it within the broader context of the theory of typed programming languages, and demonstrate that it is possible for desirable system safety properties to be maintained. In particular, we are interested in properties like:

- Correctness of active type specifications, so that users of a library need not be forced to debug errors arising within the specifications themselves.
- Correctness of translations, so that the results of translation are guaranteed to be well-typed and consistent with respect to the target language.
- Termination of active type specifications, so that evaluation of the type-level functions cannot cause the compiler or editor to hang.
- Composability of active type specifications, so that the behaviors defined by one type cannot interfere with those defined by another, no matter the order in which they are imported. This property is essential if we wish to place these specifications within normal libraries.

3 Background

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

3.1 Language-Oriented Approaches

3.1.1 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.

3.1.2 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.

3.2 Library-Oriented Approaches

3.2.1 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.

3.2.2 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.

3.2.3 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

Listing 1 The generic `map` function compiled to map the `add5` function over two types of input.

```
1  from ace.OpenCL import OpenCL, get_global_id,
2    global_ptr, double, int
3
4  @OpenCL.fn
5  def map(input, output, f):
6      gid = get_global_id(0)
7      output[gid] = f(input[gid])
8
9  @OpenCL.fn
10 def add5(x):
11     return x + 5
12
13 A = GlobalPtrType(double); B = GlobalPtrType(int)
14 map_add5_dbl = map.compile(A, A, add5.ace_type)
15 map_add5_int = map.compile(B, B, add5.ace_type)
```

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.

4 Preliminary Work

We have started implementing a form of active typechecking and translation, as outlined above, by developing an actively-typed programming language embedded within Python called Ace. In addition, we have demonstrated and empirically evaluated active code completion support for Java, implemented using class annotations and a plugin for the Eclipse editor called Graphite.

4.1 Ace: Active Typechecking and Translation

Ace is an actively-typed programming language with a fixed syntax, shared with the Python language, but user-extensible semantics specified using an implementation of the active typing mechanism described above. The type-level language of Ace is Python, and types are type-level instances of Python classes implementing a provided interface, `ace.Type`. The compiler selectively invokes the methods of these instances when type checking and translating expressions, a mechanism we call *active type-checking and translation (AT&T)*. We have demonstrated the basic practicality of this mechanism by implementing primitives from several widely-known low-level languages, including much of C99 and the entirety of OpenCL, as active libraries.

An example showing usage of Ace with the OpenCL active library is shown in Listing 1. Here, the top-level of the file is type-level code. On line 13, two instances of an active type, corresponding to the OpenCL types `__global double*` and `__global int*`, are created (to emphasize, at compile-time). The *generic functions* defined on lines 4-11 are then programmatically specialized with these types assigned to the input arguments on lines 12-13, triggering active typechecking and translation mechanism when operations on these arguments are encountered (e.g. line 7).

Listing 2 shows the portion of the implementation of the `GlobalPtrType` type family (which `A` and `B` in Listing 1 are members of) that is responsible for the subscripting operation on line 7. The `resolve_Subscript` method is responsible for checking that the subscript is an integer type, returning the appropriate type for the expression as a whole – the target type of the pointer – if so or raising an appropriate error if not. The

Listing 2 A portion of the implementation of OpenCL pointer types implementing subscripting logic using the Ace extension mechanism.

```
1  import ace, ace.astx as astx
2
3  class GlobalPtrType(ace.Type):
4    def __init__(self, target_type):
5      self.target_type = target_type
6
7    def resolve_Subscript(self, context, node):
8      slice_type = context.resolve(node.slice)
9      if isinstance(slice_type, IntegerType):
10       return self.target_type
11     else:
12       raise TypeError('<error message>', node)
13
14   def translate_Subscript(self, context, node):
15     value = context.translate(node.value)
16     slice = context.translate(node.slice)
17     return astx.copy_node(node,
18       value=value, slice=slice,
19       code=value.code + '[' + slice.code + ']')
20
21   # ...
```

`translate_Subscript` function is then responsible for translating the subscript expression into the target language. Here, our target language is OpenCL (that is, we have simply lifted a target language construct into Ace directly), so the translation is straightforward. In general, however, this method may contain sophisticated code generation logic. These two methods correspond to the $f_{\text{resolve-X}}$ and $f_{\text{compile-X}}$ functions described above.

In addition to AT&T, Ace supports more general forms of metaprogramming, and functions can be launched directly from Python with standard numeric data structures as arguments. Using Ace, we designed a scientific simulation framework that allows users to modularly specify, compile and orchestrate the execution of parameterized families of scientific simulations on clusters of GPUs. This framework has been used to successfully conduct large-scale, high-performance neuroscience simulations, providing initial evidence that Ace is useful not only as a foundational tool for researchers, but also for the practice of high-performance computing today.

4.2 Graphite: Active Code Completion

Graphite demonstrates an edit-time aspect of active types, corresponding to the f_{editor} function above, in the context of an existing language, Java, and an existing editor, Eclipse. A paper describing and evaluating this work which was recently published at ICSE 2012 [**?**]. In that paper, we introduced *active code completion*, a mechanism that allows library developers to associate interactive and highly-specialized code generation interfaces, called *palettes*, directly with types using Java's annotation system and HTML5 for implementation. A simple example of such a palette and its operation is shown in Figure 4.2.

Using several empirical methods, we examined the contexts in which such a system could be useful, described the design constraints governing the system architecture as well as particular code completion interfaces, and detailed the design of our implementation, Graphite. Using Graphite, we implemented a more sophisticated palette for writing regular expressions and conducted a small pilot study that showed that this kind of domain-specific interface is useful for professional developers working in that domain.

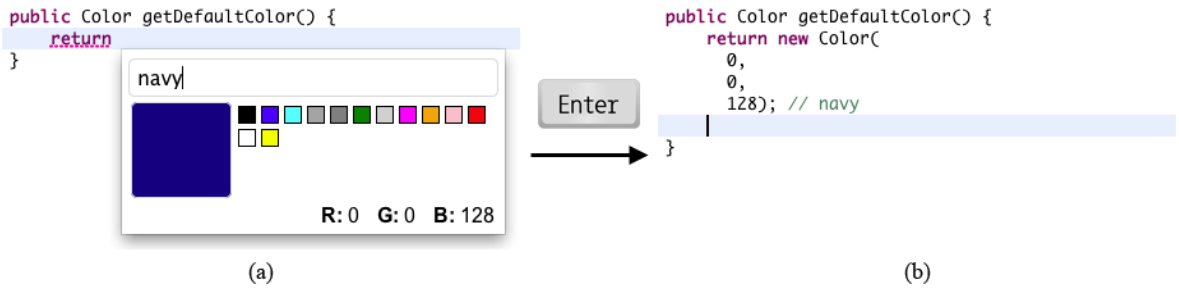

Figure 2: (a) An example code completion palette associated with the `Color` class. (b) The source code generated by this palette.

5 Work To Be Done

5.1 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.

5.2 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.

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.2.4 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.

5.3 Active Code Prediction

5.3.1 Structured, Statistical Code Prediction

Programming languages are formal systems with rich syntactic and semantic structure. They are also human systems, in that they are used extensively by people in patterned ways to express their intent. Many tools are designed to help people write code more efficiently by predicting the source code that a developer intends. For example, code completion systems for editors like Eclipse for Java display pop-up menus containing the members of the class of the variable being manipulated by the developer.

Typical code completion systems use only this sort of semantic information about the language and the libraries being used, and do not incorporate data about how developers have written programs in the past. Recent work by Hindle et al. [3] demonstrated, however, that source code could be successfully predicted statistically using a simple n-gram model that used a tokenized, rather than structural, representation of source code, trained on existing codebases.

Our first aim is to unite the structured and statistical approaches to source code prediction. That is, rather than using a tokenized representation of source code, we would like to do statistical prediction on a more natural representation of source code: the typed syntax tree. We can then condition our predictions using semantic information, specifically:

- the *type*, denoted τ, of the expression being predicted (e.g. `int` or `Color`)
- the *syntactic context*, denoted σ, in which the expression occurs (e.g. whether the expression is an argument of a function call, the guard of an `if` statement, etc.)
- the *program context*, denoted Γ, in which the expression occurs (i.e. the set of variables paired with their types that are in scope at the location that the expression occurs.)

For example, if a user has entered the code `Planet destination = `, where `Planet` is an enumeration containing `Mercury`, `Venus`, `Earth`, etc. (but not `Pluto`, of course), then we have that the expected type at the cursor is `Planet`, the syntactic context is *assignment*, and given a program context, our prediction space should only assign non-zero probabilities to:

- literal members of the `Planet` enumeration (e.g. `Planet.EARTH`)
- variables and fields of type `Planet` available from the program context
- calls to methods available from the program context that have return type `Planet`[2]

To assign a particular probability to an expression, denoted e, we first determine how likely it is that the expression is of each of these three *syntactic forms*, where syntactic forms are denoted ϕ. For each form, we can then assign probabilities to particular expressions of that form according to some form-specific conditional distribution. This model can be understood as a graphical model, as shown in Figure 3, where the syntactic form is a latent variable. The conditional distributions for both the syntactic form and expression are learned using data gathered from analyses of prior code corpuses (smoothed using a suitable method in cases where enough information is not available).

[2]We can consider operators like $+$ and $[]$ as methods of the built-in types in Java.

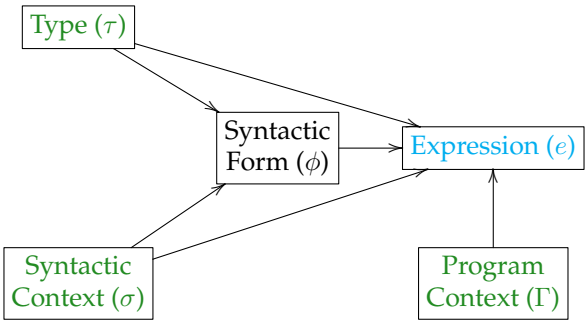

Figure 3: A graphical model representing our approach. Green variables are always observed (we do not assign marginal distributions to them.) The syntactic form, ϕ, is a latent variable, and the expression, e, is unknown. Note that the syntactic form is a function of the expression.

5.3.2 Active Code Prediction

Given a system for predicting expressions in the context of a type syntax tree, we can then naturally ask: what about specialized domain knowledge that cannot be readily extracted from prior code analysis? For example, consider a type representing strings in a particular language, such as English. Constructors of the `EnglishString` type should be biased to accept literals that have high likelihoods according to some natural language model, but we do not wish to encode all possible natural language models in the core of our prediction system. We can instead take an active types approach by allowing users to equip a type, such as `EnglishString`, with a function, f_{prob}, that takes on the job of predicting the probabilities of only expressions of that type. That is, given a reified representation of σ, Γ and ϕ, it should be able to produce $P(e|\tau, \phi, \sigma, \Gamma)$. We can then use this prediction directly as if it had been produced by a method described above.

5.3.3 Foundations, Implementation and Analysis

We propose developing the theoretical foundations of this idea, in terms of probability theory and type theory, equipping it with a notion of active code prediction as described above, and implementing this atop a real language, likely Java. We have sketched and prototyped a small version of this as an Eclipse extension in the context of a prior course project. We will then evaluate the accuracy of these models, develop use cases as in the related work on active code completion described in Section 4.2, and analyze the usability and usefulness of this prediction system in applied contexts. In particular, we are interested in uses within code editors to predict expressions and portions of expressions, as well as uses within programming systems for disabled and paralyzed individuals, who need predictive text entry to be able to communicate at a reasonable pace. The information theoretic foundations for this latter application, in the context of brain-computer interfaces, have been developed as prior work by one of the investigators on this proposal [**?**].

6 Research Plan

6.1 Spring 2013

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

6.2 Summer 2013

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

6.3 Fall 2013

I will finalize all work and write the final dissertation.

7 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.