PLDI 2014 - Author Response form for Papers

PLDI 2014

Title	Ace: Growing a Statically-Typed Language Inside a Python
authors	Cyrus Omar, Carnegie Mellon University, comar@cs.cmu.edu Jonathan Aldrich, Carnegie Mellon University, aldrich@cs.cmu.edu

Meaning of **Classification**:
A: I will champion this paper at the PC meeting (Advocate/Accept).
B: I can accept this paper, but I will not champion it (accept, but could reject).
C: This paper should be rejected, though I will not fight strongly against it (reject, but could accept).
D: Serious problems. I will argue to reject this paper (Detractor).

First reviewer's review

```
          >>> Classification <<<

C: This paper should be rejected, though I will not fight strongly against it (reject, but could accept).

          >>> Summary of the submission <<<

The paper describes a Python library called 'Ace'. Ace is designed to be used
to develop Python-embedded domain-specific languages. The basic idea is for the
user to define the translator or compiler for the embedded language (let's call
it 'L') by implementing appropriate Ace framework classes. The user then writes
L code itself in Python syntax (but not necessarily with Python semantics),
using Python's annotation facility to separate L code from actual Python code.
The Ace annotations cause the L code to be delivered to the compiler in the
form of a Python syntax tree. The syntax tree is then translated to a suitable
target language (e.g., C), or can be executed directly in Python. The authors
highlight the fact that their framework allows embedding a statically-typed
language in Python, using Python-encoded type annotations. The primary running
example in the paper is a small parallel language targeting OpenCl.



          >>> Strengths <<<

* Reasonably clean library design

* Clever repurposing of Python syntax

* Good running example



          >>> Weaknesses <<<

* Functionality provided by Ace is fairly routine and unsurprising

* Claim of "compositionality" not well justified



          >>> Evaluation <<<

Given the authors' goals and the constraint of embedding Ace in Python, I
they've done a good job engineering a library that achieves those goals. With
the exception of some minor typos and puzzling statements here and there, the
paper does an excellent job explaining Ace's basic functionality. The examples
are quite good. However, I don't see a significant enough research contribution
here to warrant acceptance. The Ace library basically provides facilities for
syntax-directed type analysis and syntax-directed translation of Python ASTs,
both of which are routine. The embedding into Python is about as clean as I can
imagine such an embedding could be, but still suffers the usual awkwardness
that arises when two distinct languages with the same syntax are intermixed.
Your claim that Ace supports "compositional reasoning" seems like a nontrivial
contribution, but it's not fleshed out in much detail in 3.4. You assert that
competitive approaches to embedded language don't have this property, but I
can't verify this claim given the minimal information provided. I would very
much like to have seen an example where failure of the consistency check
identifies, say, subtle errors. Perhaps there are more details in the ESOP
paper?

Detailed Comments
-----------------
```

	---------- - p. 1, "Evidence suggests": I suspect that you're right, but what is the evidence you're referring to? - p. 2, Listing 2: Can I reference "scale" by that name inside another clx function? As far as I can see, all your examples compose functions/methods at the Python level, but surely there will be cases where this is insufficient. - p. 3, Listing 4: "does not support []": You mention the importance of good error messages (and I certainly agree), but this message is rather cryptic. In particular, I don't understand what "[]" means in this context. - p. 4, Listing 6: "d_in", "d_out": I think you mean "d_input" and "d_output". - p. 6, "Ace's special constructor form: [clx.Cplx](3,4)": This isn't valid Python syntax, is it? How would it be used in context? - p. 6, "These methods have access to the context and node": What about access to type information? It appears that this is necessary to do correct translation, e.g., of the clx examples. Is the type information made available via the context? - p. 6, "hypothetical terms, where the overhead...might be better avoided": I don't understand what you mean by a hypothetical term, or why it lead to excessive overhead. - p. 7, Listing 8, "@ace.fun(py.Base...)[[DT]]": I was slightly surprised that this works, i.e., that a Python decorator can be "subscripted" in this manner...a few words about how/why this works would be helpful.

Second reviewer's review	>>> Classification <<<
	
	 D: Serious problems. I will argue to reject this paper (Detractor).
	
	 >>> Summary of the submission <<<
	
	 This paper discusses Ace, an extensible language built on top of Python. The authors sketch some of the features of Ace, focusing on building a new language for OpenCL programming---using Ace, a developer can write high-level code that is then compiled into OpenCL Python code. The authors argue that Ace is extensible and expressive, and then end with a comparison to related work and discussion.
	
	 >>> Strengths <<<
	
	 + Extensible languages is an important and often neglected research topic.
	 + The developed system seems to make OpenCL programming much easier.
	
	 >>> Weaknesses <<<
	
	 - Evaluation is weak.
	 - No definitive result in paper.
	 - Mismatch between high-level aims of paper and what is actually accomplished.
	
	 >>> Evaluation <<<
	
	 I applaud the authors for working on the problem of extensible language design---this is an important, interesting problem that is often neglected in the community. I think that targeting a domain like OpenCL is very attractive, as well, because their current methods of programming could stand significant improvement.
	
	 However, I think this paper fails to deliver what it promises, and this version of the paper is not publishable. At a high level, the introduction and early text of the paper talks about how many problems demand better programming languages (vs just new libraries), and that

Ace helps in building these languages. But the issues that Ace is solving are, in the end, murky and not well evaluated. After reading the paper, I only have a fairly vague idea what problem Ace is trying to solve, and I have very little evidence that it has solved it. Critically, almost all of the paper revolves around OpenCL, and the application of Ace to other domains is really sketchy.

My feeling is that it may simply be too soon to write this particular paper. That is, the authors most likely need to gather a lot more experience with Ace before it's time to write an overall design and lessons learned paper. As a way forward, I'd suggest focusing on narrower, more technically crisp problems that Ace is solving; the authors mention an ESOP submission, which might be just such a paper. Those results will then set the stage for this kind of paper.

Minor comments:

* Don't add hyphens after -ly words, e.g., "statically-typed" should be "statically typed."

* On p7, the authors write: "Ace was first conceived to answer the question: can we build a statically-typed language with the semantics of a C but the syntax and ease-of-use of a Python?" When I read this sentence, I thought, aha! this is what this paper is actually about.

* Listing 2, Listing 3. The code here struck me as inelegant, and it did not help to convince me that Ace lets developers write better code. I guess this is a side effect of needing to use Python syntax, and I'm not sure there's anything to do about it.

* 3.1.1, I was hoping to see more details about the dispatch protocol. This struck me as a very interesting design choice, but I wasn't convinced by what's in the paper that this is the right way of adding extensibility.

Third reviewer's review

>>> Classification <<<

C: This paper should be rejected, though I will not fight strongly against it (reject, but could accept).

>>> Summary of the submission <<<

This paper presents a set of libraries for defining, via translation and type-checking, new languages that use Python's concrete syntax. The system, Ace, is built entirely within Python and Ace languages are defined via implementations of Python-defined interfaces for type-checking and translation to a target language. The interface for Ace languages works on the level of a (syntactic) Python function, and defines methods for manipulating each piece of Python syntax within the function.

>>> Strengths <<<

+ nice (if not novel) idea
+ useful implementation

>>> Weaknesses <<<

- no foundations
- missing important details

>>> Evaluation <<<

I think there is a strong system, design, and set of ideas behind this paper, but it lacks a number of details that are necessary for me to adequately judge several aspects of the work. I would want to see answers to a lot of the questions below, *especially* about contexts, show up in the final paper. This would be a good use of the response.

One goal of Ace is to more easily mingle languages that require different underlying targets (perhaps targeting a device-specific language), from within a larger language---Python---with a well-tested and broad set of libraries for dealing with e.g. the filesystem, data sets, and other scripting tasks. To my knowledge, there aren't similar projects for Python specifically, though the goals are similar to (or perhaps a subset of) those of e.g. #lang in PLT Racket.

Theoretical foundations of this work are absent, as they are covered in a separate paper that is in concurrent submission. Thus, the motivation for this work is primarily engineering- and expressiveness-based, without focusing on, for example, a modular soundness theorem or a statement of compositionality between multiple languages.

To me, the key innovation in the design of Ace languages' type-checking and transformation appears is the type-based dispatch that determines how the analysis and compilation process proceed. This seems like the key decision that should be evaluated for its usability and applicability, which this paper demonstrates to some extent, but also leaves open a number of questions about it that I would like to understand more fully.

This work hits a nice point in the design space of DSLs, the creation of which can be quite daunting for the uninitiated. Two main features of Ace make it a promising choice for embedded DSL development:

- Sticking to Pythonic syntax. Much HCI work remains to be done to actually determine optimal syntaxes, and Python seems to be one of the better choices around to use as a starting point. I say this based on the knowledge that it is widely known and, with the programmers and researchers I talk to, considered relatively "friendly", "normal", and "expressive". In addition, Python's syntax has plenty of existing tooling and DSL authors don't really need to be in the business of doing syntax design to get off the ground. It also ensures that two languages interacting with one another won't look bizarrely different.

- The idea of organizing a DSL's type-checking and transformation control flow with type-based dispatch is an interesting one. It strongly suggests certain algorithms, especially bottom-up algorithms, for type-checking. I complain about this in the cons below, but there is a corresponding strong positive point to be made: It strikes me as the kind of API that could serve as a guide to implementors. They simply have to follow the pattern of dispatch that Ace provides, so Ace is providing a workflow for how to design "Ace-friendly" types. A nice side effect of this is some kind of compositionality, but without knowing more about the underlying model in the theoretical work I find it hard to understand exactly what is orthogonal and compositional and what might interact. However, the restriction, though it has downsides, is a sensible, interesting design decision to explore.

Some questions and criticisms:

1. Non-interaction of languages

In practice, it will be hard to define and enforce the set of "well-behaved" ACE extensions that don't interfere with one another, simply through using reflection in Python's runtime, or shared state in the data structures that are passed around in type checking. It's not clear if this could happen accidentally or not (see my questions about the "context" data structure below), but there certainly needs to at least be a definition of what it means to be an Ace language definition and "play nice" with the rest of the system. Do the authors have such a definition in mind, or a sense of what a language is and isn't allowed to observe and modify about Python's runtime or the Ace datastructures? Perhaps the model in the concurrently-submitted work addresses this, but it is an engineering problem as well to ensure that what the theoretical model guarantees is upheld within the running system, and I didn't see any mention of this in the paper.

Note that in related work, the authors call out PLT Racket, SugarJ, and Xroma for allowing too much global extension, so there's a strong claim being made distinguishing Ace from these systems. Given an implementation of an Ace language interface, what guarantees that the language definition won't, e.g. change the typing context that other

language definition won't, e.g. change the typing context that other languages see?

2. How do numpy and OpenCL interact in the paper's example?

Section 2.6 has a description of how Ace uses the array information from numpy to instantiate generic functions. Does this rely on numpy providing reflective operations on its values to get the type at runtime, or is there an Ace typechecker for numpy that's implemented to pass this information around, or is this type assigned by some ActiveBase or other rule of the OpenCL typechecker? I assume it must

be the latter, otherwise the dynamic and static semantics would be mingled, but this section doesn't make that point clear to me, and it seems like an important point about the interaction of Ace languages.

If numpy does provide this information, do the OpenCL active types need to be aware of the numpy active types, or is there some more generic interface they both understand? It's not unreasonable to require that different extensions know about one another, but the description here doesn't help me understand if that's the case, and if it is, what that interaction between Ace extensions looks like.

3. The typing context, variables and identifiers

What is the data structure of the typing context in Ace? This would help me answer several questions that I have that I don't feel were adequately addressed. Especially with respect to variables, the only mention I can find is "[The ActiveBase] also initializes the context and governs the semantics of variables." How does/should it govern them? I have a number of questions about this:

- Can functions be nested inside one another in Ace? If so, can functions of different active bases be nested?

- If not, what impact does that restriction have on the programs that can be written in Ace?

- If so, are the variables in nested Ace functions managed by the outer function's active base, the inner function's active base, or some combination? If an outer function defines a variable that should be mutable, but the inner one uses an active base that disallows mutation, which context wins? What about vice versa?

- How do multiple different extensions participate in extending and reading the context?

- Does the context just contain mappings of nodes to instances of ace.ActiveType as in Listing 7? Was the existence of that mapping defined by Ace or by the ActiveBase (not shown) for the OpenCL language? Can a language definition create its own representation of a context?

- How are typed identifier bindings represented?

Please tell us a lot more about contexts, how much Ace does for the programmer with respect to them, and how much the Ace language author does explicit management of the context.

Minor questions:

What "it" is being referred to in this sentence (s 3.3)

"An active type or active base can support multiple targets if desired by simply examining it during translation"

What is the key for the symbols in Figure 1? I have no idea how to interpret hollow vs. filled circles, or their relationship to "some". I could guess that since Ace has solid circles for most things that were discussed in this paper, solid means "is present most strongly". The "some" is really weird, though, isn't that what hollow circle means?

The dispatch protocol uses the word "type" in two ways, as far as I can tell, and it was a challenge to tease them apart. On the one hand, there's the AST node type, which determines to which method on the active type to dispatch. On the other hand, there's the

Ace-defined active type that is assigned to the expression by type-checking, and contains the called methods. I would think hard about the ways the paper uses the word "type" in the description of

about the ways the paper uses the word "type" in the description of the protocol, because it was overloaded.

Just for reference, here's what I'm pretty sure happens:

- First, Ace type-checks a sub-expression according to the bulleted list in 3.1.1 (and more not shown), eventually deferring to ActiveBases at the leaves of the AST.

- Then, from the leaves up, Ace uses the language-assigned type of that sub-expression (an instance of ace.ActiveType), to call the appropriate method on that type based on the whole expression being type-checked (e.g. Binop or Attribute).

Fourth reviewer's review

>>> Classification <<<

B: I can accept this paper, but I will not champion it (accept, but could reject).

>>> Summary of the submission <<<

This paper introduces a programming language Ace, implemented as a library within Python, meaning that it has access to Python tools and libraries itself.
It is statically typed, with an extensible type system, and compiles to a variety of target languages. It outlines several examples to illustrate Ace's expressiveness.

>>> Strengths <<<

The benefits of implementing a type system as a library are well motivated, and
it seems a good idea to extend Python in this way. The examples show that the language is suitably expressive (being able to capture algebraic data types, records, and higher order functions for example).

>>> Weaknesses <<<

The examples don't give me much of a sense of what the type system is, its soundness, or even how the most basic parts of the system are implemented. Further evaluation of some claims in the discussion (e.g. increasing performance of the run-time system) would be good.

>>> Evaluation <<<

This seems a promising idea, and the system is expressive enough to implement several desirable type systems. By embedding in Python, it is possible to interoperate with a large body of existing libraries, tools and compiler infrastructure. The paper is nicely structured and gives a good overview of how
the system works in practice.

The examples presented show the flexibility of the system, though I wonder if it would be helpful to show some implementation details for an even simpler abstraction. Listing 7 shows the implementation of a complex number type, but I
would like to see something even simpler (e.g. arithmetic) with reference to some simple typing rules that it is implementing. This would allow me to relate
how an active type is implemented to the theory.

One concern I have is about the soundness of the system as a whole, given that it is extensible in various ways. Perhaps this is tackled in the related paper in submission to ESOP, but it would be good to discuss. How do different extensions interact, should this be avoided, are there enough benefits already for this not to be considered too serious at the moment, etc?

Other questions, comments:

The theoretical foundations lie in type level computation - how does this relate to dependently typed languages such as Agda, Coq, Idris, etc? Each of these support language extensions in various ways (although clearly not widely used like Python).

Could similar techniques be used in other languages? How much of what you have done relies on special features of Python?

What about using the system to give more precise types to existing Python libraries? One use of static typing could be to ensure that libraries are used and composed consistently, for example.

p3 I would find it really helpful at this point to see how a simple type system
(even something as simple as a language for well-typed arithmetic) would be implemented using the ActiveType interface. At the moment, it's a bit mysterious.

p7 Listings 8/9 present a nice example, familiar to programmers in statically typed functional languages. But what do the errors look like?

p9 Statically typed languages don't need to be verbose, though - e.g. with type
inference. This is more a comparison to existing mainstream languages than the state of the art in research, surely...

p9 I'd be really interested to see this future work on capturing linear or dependent types in particular - I wonder if such a system could be used to verify resource usage or safety properties of concurrent programs.

Fifth reviewer's review

>>> Classification <<<

D: Serious problems. I will argue to reject this paper (Detractor).

>>> Summary of the submission <<<

This paper presents Ace, a system for heterogenous metaprogramming in Python. It works by combining Python's decorators with the ability to reflectively access the source code of Python functions. Ace allows programmers to specify both typechecking and translation rules for the object language as methods on types, which are then applied to functions via decorator annotations. A fixed set of rules determines how types are propagated in Python ASTs.

Ace also allows so-called "generic functions", which are effectively Python ASTs which have not been annotated with types. The programmer can then use the generic function with multiple types to generate multiple functions in the target language, allowing a form of polymorphism similar to C++ templates.

The authors present one major application of Ace, a compiler targeting the GPGPU language OpenCL. They also briefly discuss a target for C99, and for plain Python.

>>> Strengths <<<

See below.

>>> Weaknesses <<<

See below.

>>> Evaluation <<<

This paper presents an interesting system, but the paper itself is extremely confused and should not be accepted in its current form.

The central problem with the paper is that it presents Ace in a different and inaccurate manner than the summary above. In particular, it presents Ace as an "extensible language" with "statically-typed

semantics" in the style of Scala (with macros), Typed Racket, or Qi/Shen. However, Ace is not extensible in the way these other systems are, but in a different (and interesting!) way. It is also not statically-typed in the way these systems are -- instead, it uses a translation process driven by "active types", which are in fact Python objects that implement appropriate methods, and not types in a conventional sense at all.

The system most similar to Ace is the Lightweight Modular Staging system in Scala by Rompf, Amin, Odersky and others. That system uses the Scala type system to drive AST creation, and uses operator overloading instead of the methods of active types used in Ace, but is otherwise very similar, used to generate languages ranging from JavaScript to OpenCL (via Delite). Unfortunately, LMS is not discussed in the present paper. Other type-directed heterogenous metaprogramming systems include Accelerate and Flask for Haskell, also not mentioned.

Another confusing aspect of the presentation is the treatment of "compositionality". In particular, the paper takes compositionality to mean that a single program can call one library written using Ace in one place, and another in another part of the program, and the existence of these two uses in the same program does not interfere with each other. This is not, however, what is usually meant by compositionality of language extension. For example, Lisp-style macros are compositional because the `and` and `or` macros can be used together in the same expression without interference. In the sense intended here, all of the related work covered in Figure 1 is compositional.

Finally, the paper does not address at all one of the central issues in meta-programming: that of scope. Since the metaprogramming pipeline in Ace runs fundamentally through strings of source code, it is not at all clear how scoping information is maintained, how the Ace dispatch system determines which variables are referenced and what their types are, or other important questions. As an example,

"compile-time expressions" are mentioned in section 3.1.1, but never explained. What bindings are available when is a central concern in any metaprogramming system, so this is not just a minor detail.

Other issues

* The question posed in section 4 "can we build a statically-typed language with the semantics of a C but with the syntax and ease-of-use of a Python" makes no sense. Even granting the implied claim about Python's ease of use (for which the authors provide no evidence), surely the semantics of Python are a key aspect of its claimed ease of use! In fact, leaving aside its syntax as the authors do, the semantics are all that is left to provide the ease of use.

* It is unclear if the systems mentioned in 4.5 and 4.5.1 (which should be 4.6) are actually implemented or not from the description.

* The paper is confused about type abstraction. At the top of page 4, the definition of Cplx is claimed to "extend the type system", rather than being a "type alias". These are not the only options -- in fact, Cplx is an abstract type, as is possible with, for example, ML modules. Cplx does not extend the type system in this sense.

* Listing 7 is the definition of Cplx, not of Ptr.

* Haskell does not treat non-exhaustive pattern matches as type errors.

Response (Optional)

Please note the following:

- **Author identities have now been disclosed to reviewers, and any supplemental materials that were included with your submissions have been made available to them.**
- **You are encouraged to use the response to point out factual errors in the reviews, to clarify issues that are unclear to reviewers, and/or answer questions raised by reviewers.**
- **Author responses are optional. If you don't have anything to say, there is no need**

to file a response.

- **There no word limit, but please keep it to about 500 words. The reviewers have been informed that they have no obligation to read excessively long rebuttals.**

Please note that we may solicit additional reviews of your paper after the author response.

Count Words

Submit Author Response

In case of problems, please contact Richard van de Stadt.