

Your password has been changed and you are now signed in to the conference site.

 [Main](#) [Edit](#)

## #194 Ace: Active Typechecking and Translation Inside a Python

### ☒ EMAIL NOTIFICATION

Select to receive email on updates to reviews and comments.

### PC CONFLICTS

Alex Potanin  
Thomas Gross

**Submitted**

320kB

26 Mar 2014 8:00:23am EDT |

968223d53adf1291be1966060d6144b8c904cde7

You are an **author** of this paper.

### ▼ ABSTRACT

Programmers are justifiably reluctant to adopt new language dialects to access stronger type systems. This suggests a need for a language that is *compatible* with existing libraries, tools and infrastructure and that has an *internally extensible type system*, so that adopting and combining type systems requires only importing libraries in the usual way, without the possibility of link-time ambiguities or safety issues.



We introduce Ace, an extensible statically typed language embedded within and compatible with Python, a widely-adopted dynamically typed language. Python serves as Ace's type-level language. Rather than building in a particular set of type constructors, Ace introduces a novel extension mechanism, *active typechecking and translation*, organized around a bidirectional type system that inverts control over typechecking and translation to user-defined type constructors according to a protocol that cannot cause ambiguities and type safety issues at link-time. In addition to describing a full-scale language design, we give a simplified calculus that describes the foundations of this mechanism and show that, as implemented in Ace, it is flexible enough to

### ► AUTHORS


C. Omar, J. Aldrich [\[details\]](#)

### ► TOPICS AND OPTIONS

admit practical library-based implementations of types drawn from functional, object-oriented, parallel and domain-specific languages.

	OveMer	RevExp
<a href="#">Review #194A</a>	1	2
<a href="#">Review #194B</a>	4	3
<a href="#">Review #194C</a>	3	2
 <a href="#">Edit paper</a>	 <a href="#">Add response</a>	

 [Reviews in plain text](#)

**Review #194A** Modified 27 Apr 2014 11:31:56pm  [Plain text](#)  
EDT

OVERALL MERIT (?)

**1.** Reject

REVIEWER EXPERTISE (?)

**2.** Some familiarity

#### PAPER SUMMARY

This paper argues that a custom language dialect is more likely to be adopted if built as a library within the confines of an (extensible) host language, rather than as a new language with separate syntax. To support this approach, the paper formulates a technique called active typechecking and translation.

The first part of the paper shows (a) how to implement a Python library that supports a level indirection to allow analysis writers to insert custom functionality and (b) how to implement a simple type system for Python using this library. The second part of the paper formalizes the active translation approach in a calculus of type-level computation, bidirectional checking, and typed compilation.

#### EVALUATION AND COMMENTS FOR AUTHORS

I find the premise of this paper to be compelling: if language designers are able to implement custom functionality and analysis within the confines of an expressive host language, then the resulting extensions are more likely to be adopted in practice. This is an important reminder (if not a lesson) in contemporary times, when language dialects sprout

throughout industry and research studies. So either (a) a robust tool for active translation for a popular language like Python or (b) a formal foundation for such "pluggable" systems would be a welcome contribution. Unfortunately, I find the current paper to be rather incomplete on both fronts.

Besides the one example of a simple type system for Python, there are no proposals for other analyses that can easily fit into this architecture.

Although the way that syntax is overloaded to support type annotations is cute, this example alone does not demonstrate the flexibility of the active translation framework. Neither is this example analysis fully evaluated by way of a large case study to type check Python programs.

In terms of the formal presentation of active translation, the paper does not make clear, or claim, what are the challenges in this formulation.

Furthermore, as the authors admit, the metatheory for the proposal (which would guarantee that the "hooks" exposed by the system do not compromise basic safety guarantees) have not yet been worked out.

As a result, I do not think this paper is ready for publication.

### \*\*\* Additional Questions and Comments

p.2: Give a citation for the expression problem and say a few words about the inherent tension.

p.2: It's not yet clear what type-level computation will enable. Perhaps emphasize that it would be particularly well suited to, for example, dynamically computed keys, which are common in dynamic languages.

p.2: Related work of interest: Politz et al. (FOOL 2012) proposed a record type system that uses regular expressions (or, more generally, patterns) to describe sets of fields.

p.3: To make the syntax look a bit more like normal Python,

could the type annotations easily be hidden inside comments that are then parsed by the appropriate type checker Ace library?

p.3: How easily can other popular languages support the "hijacking" of syntactic forms like function application, etc? For example, what could be done in Java, which is the language used in the motivation?

p.5: Again, examples of other kinds of custom semantics, besides type annotations, would help give more clarity to what this approach enables.

Section 3: I must admit that I did not read this section very carefully, because it seems to be a very detailed description of how a standard bidirectional type checking algorithm works on a particular example. Is this there something unique to the current formulation, or the implementation in Python, that requires such a great amount of detail?

p.12: Another effort that may be worth discussing is TeJaS (Lerner et al, DLS 2013) which aims to factor JavaScript type checking (using OCaml modules) in such a way that facilitates easy customization.

p.13: That it might be difficult to implement more advanced type systems that use different kinds of contexts leaves room for doubt about how many varied kinds of systems can be implemented in this style. As mentioned earlier, additional translations, beyond the type checking example, would help address this concern.

### \*\*\* Typos and Minor Comments

p.2: "sidestep" ==> "sidesteps"

p.3: "we do lack space" ==> "we lack space"

Listing 3: "def print\_transfer" ==> "def log\_transfer"

p.5: "lines 8.6-8.14" ==> "lines 8.6-8.12"  
"lines 8.19-8.23" ==> "lines 8.17-8.21"

p.8: "It's interface" ==> "Its interface"

p.10: "We functional" is missing a verb

p.13: "(e.g. )" is missing the exempli

## **Review #194B**

Modified 4 May 2014 11:05:15am

 [Plain text](#)

EDT

**OVERALL MERIT (?)**

**4.** Accept

**REVIEWER EXPERTISE (?)**

**3.** Knowledgeable

### **PAPER SUMMARY**

This paper presents a Python library for static type-checking and staging computations. A program consists of type-level computation, introducing types and specifying type-checking and data representation, and run-time computation. Running the type-level program type-checks and generates a program for performing the run-time computation. Python semantics can be extended, but not its syntax. This avoids some of the extensibility issues with the expression problem, but introduces the problem that syntax can have different meanings than expected in vanilla Python. The semantics are formalized in a small calculus.

### **EVALUATION AND COMMENTS FOR AUTHORS**

This is a nice paper, a good idea, well executed. The first part of the paper is the strongest part. I would have liked more examples, particularly of the `trans_X` methods.

I found the formal semantics suprisingly complicated. There should be a broad overview of the formal system syntax before diving into details. It's rather difficult to follow on first reading. Figure 1 introduces a lot of syntax, but it's not all explained. I don't know what an  $\$I\$$  is, or an `intro`, or an `elim` until a few pages into Section 4. Please put details of kind-checking and quoted/unquoted forms in the main body of the paper.

Rules are not described in the order in which they appear in figures, making it more difficult to understand.

Bidirectional type-checking complicates the design of the formal system, obscuring the contribution. I wonder if a simpler formal system would be possible, then have the bidirectional type-checking added as an extension.

Please compare to partial evaluation.

Minor:

p 3, "If we wanted a matrix of dyns ..." You mean floats?


p 4, Listing 3: print\_transfer should be log\_transfer

p 6, "see on line 8.14 that a binding for x ..." I think you mean line 8.12?

p 10, "We functional data structures"

p 10, "It is instructive to rewrite lines 27--28 of Figure 2 using these desugarings." Can you show the result?

p 11, Att-Lam: where does \sigma' in the conclusion come from? Should be \sigma?

**Review #194C** Modified 5 May 2014 11:53:29am  [Plain text](#)  
EDT

OVERALL MERIT (?)

**3.** Weak accept

REVIEWER EXPERTISE (?)

**2.** Some familiarity

#### PAPER SUMMARY

The paper addresses the issue of how to add an extensible type system to an existing language without changing the existing language or requiring alternative language implementations. The authors thus use Python as both the underlying language, and the language in which their type system is expressed. Using only original Python syntax, the authors show how a type system can be added to Python in a relatively natural style. The type semantics itself is expressed in Python in a modular way as a library, w/o the need to modify the Python parser or runtime system. The approach ends up being a staged computation, where the type system executes in Python at compile time and translates typed functions to untyped Python. The second stage is then running the final type checked Python program.

After giving details of how this is implemented in Python, the authors present a core calculus formalizing the ideas of this staged approach.

#### EVALUATION AND COMMENTS FOR AUTHORS

I like the approach of embedding an extensible type system into Python and the results obtained by the author are quite elegant and seem practically useful. The paper is accessible and well written. The modular extension via user defined type constructors seems to be a very clean approach to the problem, guaranteeing the non-interference of different libraries extending the type system. The paper is clear about the limitations of the approach in that the type extensions cannot really be changing the shape of the type contexts or how type contexts are updated. So systems that use linear

types would seem inexpressible in this setting.

On the negative side, I found the formal calculus a bit removed from the Python setting. E.g., the type checking of the target internal language in Python is non-existent (everything seems to be "dyn"). So what's the point of having that be a rich type system in the calculus?

Occasionally, the paper reads as a very low level implementation description.

One aspect that I don't see addressed or don't understand is how typed code interacts with untyped code and what kind of guarantees are attributed to the typed code. In Racket, there's a slew of papers about this issue, and who checks have to be wrapped and delayed in order to execute them in some untyped context where data of a type structure may be updated. It would be good to get a description of such issues. I suppose my question has to do with runtime checks and whether any runtime checks are still present and where in the final code.

Detailed comments

-----

page 2:

"it sidestep[+s] the"

page 3:

"type signature (line [19/17]).

"treat it instead [+as] an ascribed"

"a matrix of [f32/dyns]"

page 7:

"cause the literal to [+be] analyzed"

page 10:

"We [+lift?] functional data structures to the type level:"

"forms take a single [-a] type-level value"

page 11:

Rule ATT-LAM: the final  $\sigma$  should probably be just  $\sigma$ .

"evaluation semantics remove[+s] quotations"

"ITm" spacing  $\mathsf{ITm}$  ?

page 13:

What about Coq in Figure 6? Isn't Coq the ultimate extensible language and type system?

**Response**

Submitting an author response is optional. The author response is an opportunity for you to clarify technical misunderstandings in the reviews and to answer reviewers' questions. Please keep your response to roughly 500 words.



Submit

Save as draft

500 words left