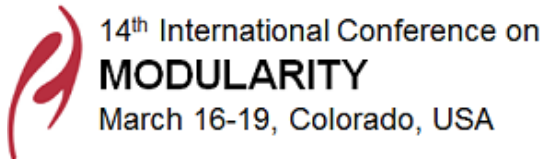


MODULARITY 2015 - Author Response form for Papers



Note: Responses consisting of more than 500 words will NOT be stored!

| | |
|--------------------------------|--|
| Title | Extensible Gradual Typing Inside a Python |
| authors | Cyrus Omar, Carnegie Mellon University, comar@cs.cmu.edu Jonathan Aldrich, Carnegie Mellon University, aldrich@cs.cmu.edu |
| First reviewer's review | <p>>>> Summary of the submission <<<</p> <p>This paper describes how type system extensibility can be achieved in a library rather than by language extension. It is achieved in the context of Python, and used to create the language @lang and core calculus @\lambda, and applied to several example type system extensions, including function programming abstractions, strings, and foreign functions. The core idea behind it all is the notion of an 'active type constructor', which uses the type of the 'target' of an expression to select the rule that determines its dynamic semantics.</p> <p>>>> Evaluation <<<</p> <p>This paper presents a language design approach where static typing is relied upon very heavily when translating programs, which enables the fixed syntax of @lang to achieve a very flexible and powerful range of dynamic semantics.</p> <p>In some ways, the approach resembles a compiler architecture more than a language design, because the translation process makes use of user-defined code, almost like a compilation process where the compiler is equipped with a large number of plugins. This may make the resulting language hard to use, because those 'plugins' must be written based on an understanding of the context in which it will run, and that seems to include a lot of elements that resemble internal compiler implementation issues.</p> <p>It would have been nice if the paper had discussed the basic design choice of letting types be so crucial for the choice of runtime behavior for a given syntactic construct. In fact, it is the opposite end of the spectrum relative to a well-established position: "Types have no effect on the dynamic semantics", which may be exploited to compile types away entirely (whether in strictly typed functional languages where types need not be represented at runtime because the discipline enforced by typing has already been ensured when type checking succeeds, or in a language like Dart where type annotations can be ignored in production mode because the semantics of ill-typed programs is well-defined). In either case, it is often considered as a "clean" position to insist that types make no difference at runtime, and it would be interesting to have some thoughts about the authors' take on this issue.</p> <p>The use of bidirectional type analysis (working with 'analytic' as well as 'synthetic' positions, i.e., top-down vs. bottom-up type processing) is interesting.</p> <p>The core calculus seems to be consistent, but some guesses must be made with respect to the meaning of some parts of the notation.</p> <p>The case study with @lang involving the OpenCL type system is not described in much detail, it is just used to set the scene for the FFI, but that's an understandable prioritization of the space.</p> <p>All in all, this paper is thought-provoking, its ideas are promising, the technical elements are convincing (but not very easy to disassemble). Nice work.</p> <p>Detailed comments:</p> |

- page 1, column 2: Keeping the syntax fixed is indeed an interesting and liberating idea!

- p1c1, 'defining only new type constructors': Given that a type constructor would often be understood as a function from types to types (e.g., 'list' is a type constructor when it can be used to create types like 'list of int'). If your concept of type constructor is significantly different then it would be useful to explain it already here.

- p2c1, 'attr["r"](e)': You describe this as a function attr["r"] applied to an argument e. Of course, the function could look up another function f in the value of e and then return the result of applying f to that value, but this basically means that "r" would be ignored (e.x would do the same thing for all x), so maybe this description is a bit unnatural in the OO case?

- p11c2, 'naming conflicts have been resolved by some extrinsic mechanism': How could you expect to resolve naming conflicts without reference to your typing analysis (which would generally need to include scope rules, though it is not obvious how to characterize that aspect of your design).

Second reviewer's review

>>> Summary of the submission <<<

This paper describes a technique to extend static type systems of an existing language, so that it becomes easier to develop reuse libraries written in one language in programs written in another language. The approach is mainly discussed on Python, which has a very simple static type system (with only a single type), and the technique discussed in this paper basically develops a static type system on top of Python, with a compiler that can get rid of all the static type information, provided type checking succeeds.

The approach is illustrated on some examples, and moreover a simple core-calculus is also proposed, to give a formal account of the technique.

>>> Evaluation <<<

General comments

I find the paper to be very complete, and am convinced about the value of the work. At some points, the presentation was a bit dense. There are many details discussed, in particular of the type checking, and it is easy to get lost in them. What I would like is to have a concrete example of usage already at the beginning of Section 2 (instead of only at the end). This will make it easier to digest all the details.

Two other points that I would have liked to see discussed are:

- performance (how long does the compilation process take, and how efficient is the resulting code)
- how easy or difficult would it be to do a similar exercise for other languages. Could @ lang be a front end for other dynamically typed languages? And what happens if you go from one statically typed language to another statically typed language.

However, overall I would recommend to accept this paper.

Detailed comments

- p. 3: I could not parse: ... tracks the regular language an immutable string is ...
- p. 5: item 5 of the translation: what is the 'fore'?

Third reviewer's review

>>> Summary of the submission <<<

The paper presents a system for extending the type system of Python

using libraries. No syntactic changes are required in Python. Rather, Python syntax is repurposed for encoding type annotations. Code is annotated with types and executed at compile time. Users define analysis and synthesis functions for type-checking and translation functions. Running the program produces a statically type-checked Python program.

>>> Evaluation <<<

Points in favor:

- The paper presents a very nice idea, well done. I'd be happy to see the paper appear. I'm really interested in experimenting with the library, perhaps to use in teaching.
- Library works with existing Python tools.
- Several interesting case studies.

Points against:

- The more interesting part of the paper for me was the beginning. The formal semantics take a lot of effort to understand. I think they're necessary for explaining how type-checking works.
- The presentation can be improved. Section 2 is very implementation heavy. I'd rather have the basic idea discussed first. There's a lot of mixing of syntactic issues and semantic issues.
- The abuse of Python syntax is rather unpleasant. It's too bad Python doesn't have extensible syntax. I fear a proliferation of incompatible dialects of Python. The modularity of the design should be discussed more. How do different type systems interact. It is claimed that it works, but not demonstrated with a case study.
- The bidirectional type checking needs to be motivated. It seems to complicate the formal system considerably. I wonder if the formal system should be split out into a different paper. As is, it's very compressed.

Comments:

Figure 2 (page 9) occurs before Figure 1 (page 10)

"e.g." should be followed by a ","

Several references have words in incorrect case (e.g., "gpu" in [11])

p. 9: mentions rule "ATT-ASC". Should be "ATT-SYN-ASC"?

Fourth reviewer's review

>>> Summary of the submission <<<

Presents an active type system approach in which types are defined programmatically (in this case in Python) and checked by static functions. Also presents a calculus for supporting bidirectional type elaboration.

>>> Evaluation <<<

For:

A lot of technical content in this paper, an appealing vision for extensible typing that is difficult to argue against, and the paper itself is well written.

Against:

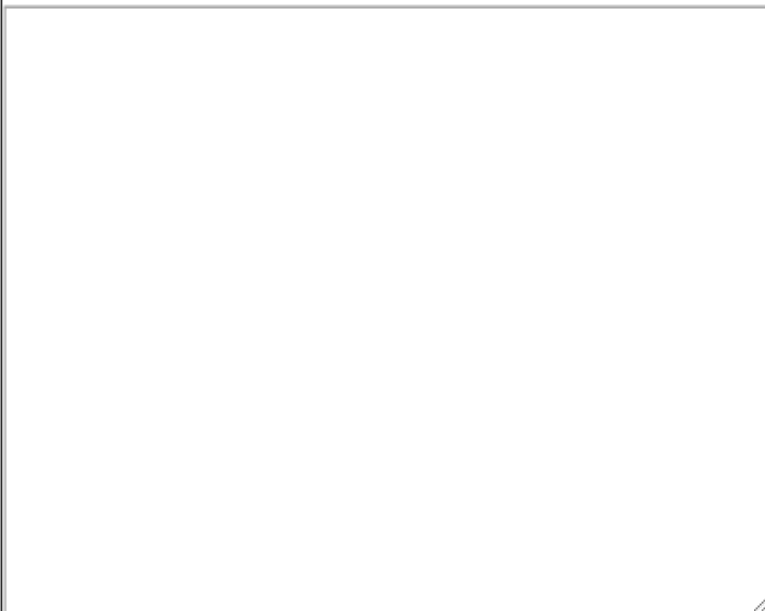
I am somewhat skeptical of the overall acceptance that this vision will find. Aesthetically the approach seems somewhat clunky to me in comparison with a type system integrated into the language design (but I do acknowledge that it provides benefits not available with such approaches).

My overall evaluation is that I do not find the basic approach and techniques in the paper that appealing to me personally.

However, I can see the technical merit in the research and time will tell whether the approach gets any traction in the broader community.

**Response
(Optional)**

Note: This (optional) response form is to be used **only to make corrections to **factual** errors in reviews, if any, or to answer specific reviewer questions. It is not to be used to clarify or elaborate on the submission, report on more recent work, or debate subjective evaluation points with reviewers. The limit is 500 words! For better readability for the reviewers, please use newlines near the right side of the text box, instead of using your browser's automatic linefeed.**



Responses consisting of more than 500 words will not be stored!

In case of problems, please contact [Richard van de Stadt](#).