Research Statement

Fabian Muehlboeck (fabian.muehlboeck@ist.ac.at)

Programming is the expression of intent through code. A programmer's confidence in their code is a measure of their expertise, the quality of their tests, and the inherent guarantees provided by the programming language and other tools they might use. My research aims to increase both the quality and the accessibility of such tools, from static type-checking and the integrated development environments (IDEs) that are built around it to ways of checking a program's behavior at run-time, and in fact combinations of the two. I believe that what sets my research program apart is the wide range of design considerations that flows into each aspect of my work as I consider how it fits into these combinations of static and dynamic approaches.

Towards better tools. To enable a new generation of computer languages, one must understand the flaws and limitations of current industrial languages and discover ways to transcend these flaws. My work has thus one foot in technologies such as Java, C#, Ceylon, and Kotlin—general-purpose, massively adopted, object-oriented languages—and one foot in future languages such as Nom and MonNom, the languages I implemented for my thesis. I envision my work as an enabler for new language designs to break down the wall that separates loosely structured languages such as Python or JavaScript from highly structured languages like Java and the others mentioned above. My design philosophy is to focus on light-weight techniques that do not foster undue complexity on the users and, wherever extra user effort is required, allow users to apply it in a pay-as-you-go manner. My approach to language design is to consider design holistically as the composition of linguistic features in service of real-world needs. This approach often requires co-designing these features to work well with each other. Furthermore, I advocate the co-design of languages with their development environment and associated tool chain. To achieve these goals, I use a wide range of techniques and tools, from machine-verified proofs and SMT solvers to low-level implementation tricks and kernel hacks.

My work so far. My training has given me the tools to lead this journey. During my PhD at Cornell, I worked on two connected main projects: improving the state of the art of object-oriented type systems in general, with a special eye towards making it compatible with gradual typing, and actually implementing gradually-typed object-oriented languages that both run efficiently and satisfy important theoretical properties. My first paper suggested simple restrictions to Java's type system that would make its subtyping decidable; these restrictions are easy to implement and compatible with existing code [1]. Later, I proposed a general framework to introduce union and intersection types while automatically preserving decidability results like the one above, which was formalized in Coq and whose techniques were adopted by the Ceylon language team [2].

For gradual typing, I first implemented Nom [3] as a gradually-typed version of pre-generics Java, reducing overheads of well-behaved gradual typing found in earlier work from 10,000% [4] to 10%. Nom's successor MonNom recognized that code in Python/JavaScript/etc. differs from code in Java/C#/etc. not just in whether programmers write type annotations, but also in how they structure their code [5]. This work thus contributed to the metatheory of gradual typing by showing how to account for such differences, and besides invariant generics, MonNom's design also included appropriate less-structured features, in particular (untyped) lambdas and dictionary-based records. Despite the added functionality, measured overheads stayed below 25%.

In 2019, I joined IST Austria, where I have been working on run-time monitoring and a theory of language design for IDEs, the latter of which I will describe further below. For the former, I proposed a merge of run-time monitoring with differential testing. The basic idea is that in some cases it can be simpler to write a program a second time to the existing informal specification, rather than generating the full formal specification necessary to monitor the program with traditional tools. Observing these programs as they run side-by-side in production and seeing them produce the same outputs on the same inputs can greatly increase our confidence in them. I established the basic plausibility of this approach [6], showing that a modified Linux kernel that monitors file operations of various parings of the same program written in Java, C, and Python mostly incurs only low-percentage overheads. I also collaborated with a mathematician on formalizing several elegant proofs of Fermat's theorem on the sums of two squares in Coq, writing tactics for greater proof automation [7].

In the following, I describe how I plan to build on this work to fulfill my vision.

Better Type Systems

My previous work in this space focused on decidability and simple algorithms, the lack of which makes compilers, IDEs, and gradually-typed programs crash with little information to diagnose the underlying problem. Another problem for both IDEs and gradual typing is generic type-argument inference, a key feature that makes generics in languages like Java reasonably usable. Generics, in turn, give powerful types to large parts of modern standard libraries. The algorithms for generic type-argument inference are typically ad-hoc—slight changes in their input can have significant effects on a program, and even slight "improvements" to the algorithm can break backwards-compatibility. I am currently working on a more principled approach to generic type-argument inference, ensuring principal types (that is, every expression, has a most-precise type) and semantic coherence, that is, more precise type information does not cause code to fail, which is important for gradual typing). In order to guarantee these properties, I classify type parameters that simply cannot be automatically inferred and exploit my earlier work on unions and intersections to be able to compute joins and meets in the inference process. The design also considers common special cases like the interaction between generic type-argument inference and type inference for lambdas (say as direct arguments to calls to generic map methods), and I propose standard library designs that can better cope with the necessary restrictions to inference [8].

There are always more features that a language could try to add, but oftentimes, composing type system features is hard. There is an unending supply of problems in this space, and I would particularly like to look at addressing issues that come up in integrating type systems for authorization, effects, ownership, and typestate with full-fledged standard type systems, also with an eye towards gradual typing.

Co-Designing IDEs

While a programmer is writing code, that code rarely actually represents a program as defined by the formal language definition, and as such has no formally defined meaning. IDEs use heuristics to assign partial semantics to those partial programs, from which they generate the information the display to the programmer. This means they effectively guess at what the complete program will look like once the programmer is finished editing and try to display information consistent with that. Writing these heuristics is hard, and even the most-supported major industrial IDEs like Visual Studio and Eclipse fail to provide correct information sometimes, in which case they might suddenly underline large parts of the code in red, suggest or even automatically insert wrong completions, or simply provide no information at all.

I am currently working on what would be the first principled way of reasoning about IDE behavior and language design [9]. The key idea is to formalize a core part of IDE design: expectations on how programmers edit their code. For example, typically, IDEs are most effective when we write code line by line, left to right, and not very effective when we just insert (even correct) characters in random locations. Using such a model of how code is expected to evolve gives us an implicit definition of the meaning of partial programs by simply looking at which complete programs are reachable under certain edits. This not only allows us to judge the quality of an IDE's feedback by checking whether it provides information consistent with such a definition, but it also allows us to reason about the quality of information any IDE could reasonably be expected to generate at any point in the editing process, which has implications for syntax design. For example, assuming a programmer locally writes code from left to right means that an expression of the form `let x : int = 5 in [e]` lets the IDE provide them with information about `x`'s type while writing `[e]`, while in an alternative, though semantically equivalent form, `[e] where x : int = 5`, the IDE is simply unable to be similarly helpful, and my framework is able to formally express this difference.

This work has the potential to open up a whole new area of exciting work, from improving the theoretical framework, to user studies on actual fine-grained editing behavior, to designing meta-type-systems for the language definitions used in language workbenches, which generate large parts of compiler and IDE code from such definitions. Eventually, we may have powerful standard IDE heuristics and associated rules for how to design one's syntax and static semantics to be able to simply plug their language into any standard IDE and instantly gain a much larger level of support than is offered by standard editors today.

Run-Time Monitoring

Run-time monitoring can in many cases be easier to use than static approaches, simply because reasoning about an individual run of a program is easier than reasoning about all possible runs. Of course, this still comes with costs: performing checks while a program runs causes overheads, and programmers still have to specify what needs to be checked in the first place, which can be a lot of work for high-level correctness properties of complicated programs. Differential monitoring [6] in principle addresses both problems, restricting the bulk of the specification problem to obtaining a second version of the program, and reducing the monitoring overhead to the necessary comparisons between the two programs' behaviors, as their own code producing these behaviors can be run in parallel. The key challenge in making this idea practical is dealing with "acceptable" non-determinism—such as the order of operations and outputs of different threads being affected by the system scheduler, or simply the precise values of timestamps—and the fact that the monitored programs run on the same system, but to both themselves and the outside should appear as a single program. Both challenges mean the monitor needs to actively help keep the two programs roughly in sync. The benchmarks in [6] make assumptions about synchronization that are compatible with the behavior of deterministic web servers. In future work, I will explore supporting other scenarios with a mix of simple specifications and API re-designs.

Gradual Typing

In order to fulfill my vision of enabling new language designs based on gradual typing, I need to show that there are acceptable designs that achieve a rough feature parity with modern languages. I started with Nom as a rather restrictive language, designed from scratch, and have been extending the language design from there. This is a major difference to almost all existing work in gradual typing, which typically seeks to add it to an existing language and is thus severely restricted in its design decisions, always having to sacrifice either performance or important behavioral guarantees. Nonetheless, replicating the features of modern languages remains a central goal in my work, even though they may have to be slightly modified.

The next key step in the line of Nom-based languages is to improve the state of generics. MonNom's generics are invariant and lack type-argument inference, two key roadblocks on the way to parity with existing industry languages. The work on generic type-argument inference I described above will be a key building block in this endeavor. Another, medium-term issue with generics is enabling generic type-arguments to be discovered and refined dynamically: currently, if a value is ever going to be treated as an array of integers, it needs to be created explicitly as such. It would be desirable to be able to just create a dynamic array object that can be passed into typed code expecting an array of integers if at that point it only contains such values. This is largely an issue of efficiency, and there are various possible designs that would need to be tested.

On the other side, MonNom's key theoretical contribution was to recognize that dynamically-typed code uses different idioms than statically-typed code. Accommodating more such differences will further ease the transition to such a gradually-typed language. Existing attempts at gradual typing, such as TypeScript, are quite informative here because they were designed to capture common idioms. One of these is essentially a dynamic version of type state, which would thus be an interesting target to integrate with the rest of my type system.

In the long-term, I would like to investigate more advanced type-system features like authorization, effects, and ownership. Their adoption into mainstream languages has so far been hampered by the extra effort they would introduce into all parts of a program, not only those where they really matter, and gradual typing might be a suitable approach to addressing this problem in a pay-as-you-go manner.

Closing

As research faculty, I will provide a new generation of tools for programmers to gain confidence in their code: sound gradually typed languages will enable them to use the benefits of type systems wherever they outweigh their costs, and particularly the run-time costs of sound gradual typing will be minimal; the type systems themselves will have fewer confusing corner cases, and the languages and IDEs will be optimized for each other. Wherever type systems are not enough, efficient run-time monitoring can add additional confidence in one's code.

References

[1] B. Greenman, F. Muehlboeck and R. Tate, "Getting F-Bounded Polymorphism into Shape," in *PLDI*, 2014.

[2] F. Muehlboeck and R. Tate, "Empowering Union and Intersection Types with Integrated Subtyping," *PACMPL,* vol. 2, no. OOPSLA, 2018.

[3] F. Muehlboeck and R. Tate, "Sound Gradual Typing is Nominally Alive and Well," *PACMPL,* vol. 1, no. OOPSLA, 2017.

[4] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek and M. Felleisen, "Is sound gradual typing dead?," in *POPL*, 2016.

[5] F. Muehlboeck and R. Tate, "Transitioning from Structual to Nominal Code with Efficient Gradual Typing," *PACMPL,* vol. 5, no. OOPSLA, 2021.

[6] F. Muehlboeck and T. A. Henzinger, "Differential Monitoring," in *Runtime Verification*, 2021.

[7] G. Dubach and F. Muehlboeck, "Formal Verification of Zagier's One-Sentence Proof," *CoRR,* vol. abs/2103.11389, 2021.

[8] F. Muehlboeck and R. Tate, "Towards Practical Inferable and Gradualizable Generics," in *preparation*, 2021.

[9] F. Muehlboeck, C. Schilling and T. A. Henzinger, "Partial Semantics for Partial Programs via Edit Models," in *preparation*, 2021.