The Lang-n-$\mathcal{X}$ Series of Systems[1]

Matteo Cimini

website: https://www.cs.uml.edu/~mcimini/

As the theory and practice of programming languages advance, we have seen the creation of a plethora of new programming languages in the past decades: TypeScript (Microsoft), Hack (Facebook), Go (Google), Dart (Google), Rust (Mozilla), Julia, Elixir, and many others. Similarly, many domain-specific languages have been created: Halide for GPU computing (by MIT with help from Stanford, Google, Adobe), Turi for ML (formerly GraphLab, by CMU/UoW, acquired by Apple), GraphIt for graph analytics, as well as many others. This trend is not confined to big companies and teams but has been embraced at all levels including at the level of individuals.

Language designers need methods and tools for quickly deploying and validating their new programming language designs. Safer programming languages mean more secure and reliable software systems and, in turn, a safer world. We all benefit when complex software is written in programming languages that provide strong safety guarantees.

To help language designers, I have created the Lang-n-$\mathcal{X}$ series of systems during the past several years since my joining UMass Lowell. This is a multi-year project that comprises a series of systems that are all connected and part of the same story. Each system addresses a different aspect within the same overall objective. The series contains the following systems.

- Lang-n-Play: Languages as First-Class Citizens [1,3,4]
- Lang-n-Send: Processes That Send Languages [6,7]
- Lang-n-Change: A DSL for Language Transformations [9,14,16–18]
- Lang-n-Check: Type Systems for Programming Language Definitions [2,13]
- Lang-n-Prove: A Proof Language for Language Proofs [5,11,15]
- Lang-SQL (*a.k.a.* Lang-n-Query): A Query Language for Querying Languages [8,10]

The series is divided into two parts. The first three systems help with the **flexible creation and use of programming languages**: Lang-n-Play, Lang-n-Send, and Lang-n-Change. The last three systems help with the **verification of programming languages**: Lang-n-Check, Lang-n-Prove, and Lang-SQL. The following sections describe each system.

1 Lang-n-Play: Languages as First-Class Citizens [1,3,4]

I have designed and implemented Lang-n-Play, a programming language with *languages as first-class citizens* [1,4]. Like in other systems, Lang-n-Play programmers can define their own programming languages, these are *programmer-defined languages*, and then use them to execute programs in multi-language programming style. Lang-n-Play also handles programming language definitions with the same status as any other expression, and elevates them to be *runtime values* that can be bound to variables, passed to functions, returned by functions, and inserted into lists, among other operations. This approach offers flexible features in the runtime creation and modification of languages, and promotes new idioms in multi-language programming.

Milner's type safety is an essential theorem in programming languages theory and says that "*well-typed programs cannot go wrong*". I have set forth the observation that Milner's type safety is too strong a property in the context of multi-language programming [4]. Indeed, language designers may intentionally execute programs with unsafe programming languages that they have created. I have therefore proposed a refined safety theorem for multi-language programs:

Well-typed multi-language programs can go wrong on programmer-defined languages only.

[1] *Disclaimer:* This is an excerpt from a job related document. As such, it is tuned on an "I have done" writing style and on discussing main features and results briefly rather than providing a careful description of the systems.

I have proved this theorem for a core formalization of Lang-n-Play (a λ-calculus with languages as first-class citizens) [4]. This theorem can be a guiding principle for multi-language programming systems: It captures the idea that the surrounding management of programmer-defined languages must be correct while there is nothing we can do when programmers use ill-defined languages.

I have implemented Lang-n-Play with higher-order logic programming and I have made the point that the features of higher-order logic programming are an exceptional fit in implementing multi-language programming languages [3]. The core interpreter is only 73 lines of code and yet it enables sophisticated features such as passing languages to functions and changing evaluation strategies (call-by-value *vs* call-by-name) at runtime.

2 Lang-n-Send: Processes That Send Languages [6, 7]

Preamble After I have explored languages as first-class citizens in the *sequential* programming language Lang-n-Play, I have investigated the same for *concurrent programming languages*.

Milner's π-calculus is a formal concurrent language that has a similar role for modern concurrent languages based on message-passing communication (as opposed to shared-memory) that the λ-calculus has for functional programming languages. I have designed Lang-n-Send, a formal concurrent language that extends the π-calculus in order to handle programmer-defined languages as data that can be communicated [7]. Processes can define programmer-defined languages and use them to execute programs. Furthermore, processes can send and receive pieces of programmer-defined languages through channels. Next, I have augmented Lang-n-Send with monitoring capabilities for analyzing the execution of programs, and with the ability to send pieces of regular expressions to verifiers for them to include into larger properties to be verified [6]. I have provided a formal semantics of Lang-n-Send and several examples that demonstrate that it promotes new idioms and new client-server scenarios for the runtime creation and modification of languages.

3 Lang-n-Change: A DSL for Language Transformations [9, 14, 16–18]

Preamble Lang-n-Play and Lang-n-Send have the ability to take several pieces of programming language definitions from different sources and combine them to form a finished language. Literature also offers more sophisticated operations for composing/modifying languages. However, when we write functions for *automatically adding features to programming languages* we need to manipulate programming language definitions with more fine-grained precision.

A prominent example is that of adding subtyping. (To recall, it is by subtyping that we can pass an integer to a function that requires a floating point even though integers and floating points are different types.) Subtyping cannot be achieved by simply combining pieces of languages together but it requires modifying the type checker in non-trivial ways. My question then is: "*Can we develop a domain-specific language to express fine-grained language transformations?*" Together with my former student Benjamin Mourad, I have developed such a DSL, Lang-n-Change [16, 18], and have used it to automatically add subtyping to languages, as well as to add imperative programming, object-oriented programming, pattern-matching, and gradual typing, among other features [14, 17, 18], and to automatically generate abstract machines [14].

Lang-n-Change contains striking features for retrieving information from programming language definitions. I have leveraged these features to write a concise validator that guarantees that bisimilarity is a congruence for process algebras given as input [9].

Education research: Courses in programming languages typically cover several programming language features. I have made the point that Lang-n-Change language transformations are akin to pseudocode that highlights the insights behind programming languages features. I have set forth the thesis that employing Lang-n-Change language transformations in teaching programming languages features can help students deepen their understanding of the feature at hand [14], and I have conducted a small study on this thesis in the context of one of my courses [14].

4 LANG-N-CHECK: Type Systems for Language Definitions [2, 13]

Preamble The systems that we have seen so far address language designers' flexibility in creating programming languages. However, once we have a programming language it is important to verify that it satisfies the properties that we have intended.

Type systems play a major role in *program analysis*. My question then is: *Can they play a similar role in programming language analysis?* (That is, the analysis of the meta-theoretic properties of programming languages such as type safety.) Together with my colleagues, I have created a type system for programming language definitions that guarantees that the programming language in input is type safe [13]. I have proved a theorem that guarantees that result and that says "*Well-typed programming language definitions are safe*". (Intrinsic typing proposes a similar theorem but I have made the point that such approach does not use types in the way they are typically used in programming languages [13].) I have developed the LANG-N-CHECK tool that type checks programming language definitions given in input as type safe. Our results are able to guarantee type safety for a class of functional languages that includes several common types and operators [13]. Further studies are needed to scale the approach to more sophisticated languages.

Education research: Thanks to a type-based approach, LANG-N-CHECK has a high level view of the programming language being analyzed and can report error messages using the same jargon of language designers. I have then set forth the thesis that a language type checker such as LANG-N-CHECK can be an effective tool for 1) teaching functional language design and for 2) collecting statistics on the design mistakes that students make during their modeling of programming languages. (Such statistics may inform our teaching style.) I have conducted a small study on that thesis in one of my courses [2].

5 LANG-N-PROVE: A Proof Language for Language Proofs [5, 11, 15]

Preamble LANG-N-CHECK handles a class of languages that exhibits a common organization. I observed that their type safety proofs also have a common structure. Then, I have used this observation to automatically generate *mechanized proofs* of the type safety of those languages.

More generally, proofs of programming language properties often follow a schema that can be applied not to just one language but, rather, to a class of languages. My question then is: *Can we develop a domain-specific language to express proofs in a way that applies to classes of languages?* I have designed and implemented LANG-N-PROVE, a DSL for expressing such proofs [5]. LANG-N-PROVE contains linguistics that are specific to interrogate programming languages. LANG-N-PROVE takes two elements as input, a programming language definition and a LANG-N-PROVE proof, and generates the mechanized proofs that are specific to the language given as input. I have developed the LANG-N-PROVE proofs that describe the type safety proofs of the class of functional languages mentioned in the previous section [5], and which is the target of the LANG-N-CHECK type checker. Furthermore, my former student Seth Galasso has extended this work to capture the type safety proofs for that same class of languages when subtyping is added to them [15]. These works have demonstrated that LANG-N-PROVE generates the valid mechanized proofs (with some limitations, please refer to the papers [5, 15]) of a great number of programming languages.

LANG-N-PROVE proofs are akin to the pseudocode of *programming language proof methods* when applied to classes of languages. My next question then is: *What is the complexity of carrying out a programming language proof method?* specifically as related to language design quantities such as the number of operators, the number of rules for the evaluator, for the type checker, and so on. I have investigated that question for the standard type safety proof method (called "*syntactic approach to type soundness*" in the literature) when applied to the class of functional languages that we have targeted with LANG-N-PROVE [11]. This represents the first example of this type of analysis of the complexity of carrying out programming language proof methods.

6 Lang-SQL: A Query Language for Querying Languages [8, 10]

Preamble Lang-n-Check and Lang-n-Prove, as well as related tools in the literature, focus on a "big" property such as type safety. Given a programming language, however, it is interesting to also validate "smaller" questions: What are the binders of the language? What are the contravariant types? What type of data is mutable? And others. Analyzing these smaller aspects goes a long way to help debug a programming language. Then, I have built methods for analyzing these smaller aspects.

I have made the point that *interrogating programming language definitions should be akin to interrogating databases* [8]. In that field, SQL is a standard query language and SQL queries are concise, declarative, and mostly readable. I have proposed the *languages-as-databases* approach where language definitions are stored as database tables [8]. I also have designed and implemented Lang-SQL [8], a SQL-inspired query language that can express queries over programming language definitions. Lang-SQL contains linguistic features that are specific to query language design aspects. I have demonstrated Lang-SQL with several queries that interrogate programming languages over various and diverse aspects of language design [8]. I have also used Lang-SQL queries to rewrite the majority of Lang-n-Check [8]. This latter implementation is very concise. I have used Lang-SQL queries to implement a concise validator that guarantees that bisimilarity is a congruence for process algebras given as input [10]. (This is another take at the implementation of the validator previously mentioned, here with Lang-SQL queries rather than Lang-n-Change.)

How This Story Continues (Only the Very Next Step Is Discussed) Most of the techniques in language-parametric verification that establish results at the level of formal proofs target *big* properties (such as type safety). Lang-SQL can target *smaller* aspects of programming languages with checks in the form of queries, but these queries do not establish results at the level of formal proofs. My archived paper [12] proposes *language logics*, a counterpart of program logics for programming language analysis rather than program analysis. One of the benefits of language logics is that they can analyze small questions and can do so at the level of formal proofs, therefore filling a gap. I have implemented a prototype prover for a language logic. This is another tool within the Lang-n-$\mathcal{X}$ series, which is called Lang-n-Assert. However, much work will be needed to elevate language logics to a mature formal method that enables full programming language analysis.

I will also work on enlarging the scope of the systems that I have presented. Indeed, some of the techniques that I have developed apply to a limited class of functional languages. **It took several years to take these new techniques off the ground in their basic form. It is time to scale all these new techniques so that they apply to sophisticated programming languages**.

Education Research: Lang-n-Prove proofs are akin to the pseudocode of programming language proof methods. My claim is that teaching these proofs can help students see the proof method at hand with some generality and deepen their understanding of why some programming languages satisfy a property while others do not. In my future research, I will conduct studies to investigate the merits of this thesis.

References

[1] Matteo Cimini. Languages as first-class citizens (vision paper). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, page 65–69, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Matteo Cimini. Early experience in teaching the basics of functional language design with a language type checker. In William J. Bowman and Ronald Garcia, editors, *Trends in Functional Programming - 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12–14, 2019, Revised Selected Papers*, pages 21–37, Cham, 2020. Springer International Publishing.

[3] Matteo Cimini. On the effectiveness of higher-order logic programming in language-oriented programming. In Keisuke Nakano and Konstantinos Sagonas, editors, *Functional and Logic*

Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14–16, 2020, Proceedings, pages 106–123, Cham, 2020. Springer International Publishing.

[4] Matteo Cimini. A calculus for multi-language operational semantics. In *Software Verification: 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18–19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18–19, 2021, Revised Selected Papers*, page 25–42, Berlin, Heidelberg, 2021. Springer-Verlag.

[5] Matteo Cimini. Lang-n-prove: A dsl for language proofs. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, pages 16–29, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Matteo Cimini. Lang-n-send extended: Sending regular expressions to monitors. In Clément Aubert, Cinzia Di Giusto, Larisa Safina, and Alceste Scalas, editors, *Proceedings 15th Interaction and Concurrency Experience, ICE 2022, Lucca, Italy, 17th June 2022*, volume 365 of *EPTCS*, pages 69–84, 2022.

[7] Matteo Cimini. Lang-n-send: Processes that send languages. In *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES 2022)*, volume 356, pages 46–56. Open Publishing Association, mar 2022.

[8] Matteo Cimini. A query language for language analysis. In Bernd–Holger Schlingloff and Ming Chai, editors, *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, volume 13550 of *Lecture Notes in Computer Science*, pages 57–73, Cham, Switzerland, 2022. Springer.

[9] Matteo Cimini. A declarative validator for gsos languages. In *Proceedings of the 14th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES 2023)*, volume 378, pages 14–25. Open Publishing Association, 2023.

[10] Matteo Cimini. Testing languages with a languages-as-databases approach. In *Tests and Proofs: 17th International Conference, TAP 2023, Leicester, UK, July 18–19, 2023, Proceedings*, page 108–126, Berlin, Heidelberg, 2023. Springer-Verlag.

[11] Matteo Cimini. Towards the complexity analysis of programming language proof methods. In Erika Ábrahám, Clemens Dubslaff, and Silvia Lizeth Tapia Tarifa, editors, *Theoretical Aspects of Computing – ICTAC 2023*, pages 100–118, Cham, 2023. Springer Nature Switzerland.

[12] Matteo Cimini. From program logics to language logics. https://arxiv.org/abs/2408.01515, 2024.

[13] Matteo Cimini, Dale Miller, and Jeremy G. Siek. Extrinsically typed operational semantics for functional languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 108–125, 2020.

[14] Matteo Cimini and Benjamin Mourad. Language transformations in the classroom. In Ornela Dardha and Valentina Castiglioni, editors, *Proceedings Combined 28th International Workshop on Expressiveness in Concurrency and 18th Workshop on Structural Operational Semantics, EXPRESS/SOS 2021, and 18th Workshop on Structural Operational SemanticsParis, France (online event), 23rd August 2021*, volume 339 of *EPTCS*, pages 43–58, 2021.

[15] Seth Galasso and **Matteo Cimini**. Language-parameterized proofs for functional languages with subtyping. In Jeremy Gibbons and Dale Miller, editors, *Functional and Logic Programming - 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15–17, 2024, Proceedings*, pages 291–310, Singapore, 2024. Springer Nature Singapore.

[16] Benjamin Mourad and **Matteo Cimini**. A calculus for language transformations. In *46th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2020)*, pages 547–555. Springer, 2020.

[17] Benjamin Mourad and **Matteo Cimini**. A declarative gradualizer with language transformations. In Olaf Chitil, editor, *IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020*, pages 44–54. ACM, 2020.

[18] Benjamin Mourad and **Matteo Cimini**. System description: Lang-n-change - A tool for transforming languages. In Keisuke Nakano and Konstantinos Sagonas, editors, *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*, volume 12073 of *Lecture Notes in Computer Science*, pages 198–214. Springer, 2020.