University of São Paulo
Institute of Mathematics and Statistics (IME-USP)

Department of Computer Science

Bachelor Thesis Proposal

Dependently-typed data-centric programming

Candidate:

Eduardo Sandalo Porto

Supervisor:

Dr. Ana Cristina Vieira de Melo

São Paulo, SP - Brazil
2024-04-22

1 Introduction

Research within the field of Programming Languages (PL) is generally divided into three main categories: theory, design, and implementation. As described by Wegner [14], the development of the field as a branch of research started from the production of languages empirically in the 1950s, evolving into mathematical formalization and theory during the next few decades.

According to the author, the **theory** of programming languages emerged from results in automata theory and formal languages, and their applications to parsing and compilation. Later texts associate a wider range of topics with the theoretical sub-field, such as formal verification, type theory, syntax and semantics of programs, among others [4, 12, 6].

Programming language **design** is a more practical sub-field, involving the constructions used and provided by languages, programming paradigms, human-computer interaction through linguistic means, and others [13].

The domain of programming language **implementation** focuses on topics such as compilation and interpretation, which are approaches to program execution Aho, Lam, Sethi, and Ullman [1].

The three aforementioned sub-fields present significant challenges in programming language development and warrant thorough consideration within their own contexts. This project aims to research and apply elements of each of them to the field of data-centric programming through the development of a **domain specific language** with **dependent types**. Thus, the primary focus of the project lies in the comprehensive study of programming languages, with the secondary objective being their practical application in data-centric programming.

Next, we will examine in detail the principles and sources of inspiration that will guide the development of the domain-specific language (DSL) to be produced, linking them to the subfields of research in PL.

1.1 Dependent types

Dependent type theories are formal systems of intuitionistic mathematics [7] that have found great use within the field of PL. As discussed in Norell [11], the use of dependent type theories is already established in some contexts such as theorem proving, and it can also be highly beneficial in PL for scenarios apart from formal mathematics. Some languages, like the theorem prover Coq, envisioned the use of dependent types from the start; in other cases, they received extensions adding support for versions of these theories, as seen in Haskell [5]. Further examples of prominent languages with dependent types include Idris 2 [3] and Agda [2]. This project aims to be implemented in Lean 4 [10], which is both a theorem prover and an efficient functional programming language.

Briefly discussed, dependent type theories break the division between the type level and the value level, allowing types to depend on values and vice versa. They are widely used in the context of theorem proving because they enable type-level propositions to reference values; for example, assertions like the one below can be made via the Curry-Howard correspondence, as described by Pierce [12].

$$\forall n \in \mathbb{N}, n + 0 = n$$

In Lean 4, this is can be encoded and proved as such:

```
variable (n : Nat)
theorem n_plus_zero : n + 0 = n := Eq.refl _
```

However, this is not the only valid application for such theories. This project aims to use dependent types to encode information about data frames at the type level. Initially, we will consider a few potential applications of dependent types in data-centric programming, but first, we need to consider what this style of programming actually is.

1.2 Data-centric programming

Languages and systems in this domain primarily manipulate and manage data. Perhaps the most well known language in this department is SQL, initially developed in the 1980s and has since become the standard in relational databases. More closely related to the scope of this project is the Python library *pandas*, a data analysis and manipulation tool. The main modelling unit of this library is the *data frame*, a 2-dimensional data structure similar to a spreadsheet. In the scope of Python and pandas, data frames are untyped structures that offer little semantic insight of the data they contain to the programmer or data scientist working on them by default. This is an important short-coming that can be dealt with using dependent types.

More specifically, a few ways dependent types tackle this issue and that will be worked on in the project are the following:

1. **Data typing**

   The columns of a data frame can be assigned types through lists of types, ensuring both the compiler and the programmer know what formats of data to expect and are expected during execution.

2. **Dimensional and type safety**

   Using structures like size-indexed vector and matrices, we can ensure that operations between data frames can only be performed if they happen between compatible data frames, both regarding their dimensions and their column types. These operations include addition, multiplication, joins, etc.

3. **Typed constraints**

   Transformations can change the shape of data, and it can be useful to assert certain constraints or properties still hold or are given by different transformations. Examples include sorting and filtering, which may be constraints of other transformations.

4. **Data integrity**

   Again, properties and constraints given by the type system can prevent the creation of invalid and inconsistent data frames.

5. **Schema evolution**

   Dependent types can facilitate schema evolution by ensuring that transformations and operations preserve the integrity of the data frame's schema.

6. **Custom validation rules**

   They also enable the creation of custom validation rules specific to the domain or application, ensuring that only valid data manipulations are performed. This differs from normal typing constraints because they allow custom behaviour to the specific data set, enforcing rules such as uniqueness of an ID, length constraints, or specific formatting requirements.

Additional features and research directions will be explored during the course of the project.

1.3 Domain specific languages

DSLs are narrowly scoped languages developed for specific application domains, aiming to achieve gains in expressiveness and ease of use compared to general-purpose languages [9]. They do not necessarily have to be programming languages; for example, HTML is a domain-specific markup language specific to the web domain,

There are both advantages and disadvanteges to developing DSLs. On one hand, they are easier to implement than general-purpose languages; on the other hand, they require developers to possess not only a deep understanding of programming languages but also of the application domain,

Overall, the use of DSLs is considered advantageuous, as indicated by both qualitative and quantitative studies cited by Mernik, Heering, and Sloane [9]. This is reflected in the world of technology: numerous DSLs have achieved commercial and/or product success, such as Excel, SQL, VHDL, LaTeX, among others.

1.4 Why dependent types?

As discussed by McKinna [8], dependent types express types in terms of data, making them more flexible and allowing for automatic program verification to the extent desired by the programmer. A well-typed dependent program does more than handle errors: it prevents them in the compilation process itself, before the program executes for the first time. With this theoretical concept, we can represent invariants of practical programs. Dependent types assist programs in achieving the property of *soundness*, defining the impossibility of programs reaching invalid states. They also provide information about the programmer's application domain to the type checker, making the compiler an ally in programming. Finally, we know that there is a high time cost associated with formal verification, and dependent types allow the programmer to perform this verification to the extent they believe will bring benefits.

Furthermore, research on dependent types is still fairly recent. Languages that utilize this concept began to become viable around the 2000s, and to this day, the main languages involved in the subject are still under major development. Idris 2 started its development in 2019, Lean 4 had its initial release in 2021, the first public version of Agda was launched in 2009, etc. There is still much

room for research in the field, and a significant portion of its applications is still unknown. Involving dependent types is the most innovative part of this work; thus, in addition to the discussed benefits inherent to dependent types, working on this subject also involves exploring new frontiers in which they can be used.

In summary, we want to use dependent types to:

1. Provide domain information to the type checker.
2. Explicitly represent the requirements and invariants of program components.
3. Prevent errors statically, before executing the program in question.
4. Explore the use of the concept in a novel field.

2 Goals

Summarizing the discussions made in previous sections, the goal of this project is to develop a domain-specific language for data-centric programming with a focus on dependent types; furthermore, this is a work that has its primary area of research as programming languages, and as a secondary area, data-centric programming.

The language to be developed should be declarative and epxressive, capable of describing and executing algorithms without imposing significant friction on the programmer with specific environments and hardware. Up to this point, we have not discussed performance, and there is a reason for this: this project does not aim to develop a language with exceptional computational performance. Our DSL should provide the programmer with an exploratory environment, better suited for prototyping, but the resulting program may not necessarily be the fastest. There is a wide range of projects that focus on this aspect, which may be compared with our DSL further in its development.

Below, we define a list whose order establishes the priority of each focus item within the project's objectives, dividing them between study and practical components.

1. **Application of dependent types**
2. **Theoretical study on dependent types**
3. **Application and implementation of a domain-specific language**
4. **Application of algorithms in data-centric programming**
5. **Study on domain-specific languages**
6. **Theoretical study on data-centric programming**

In summary, the main objective of the project is to study dependent types and apply them in the discussed domain. Based on these focus items, we will conduct a study that involves the three subareas of PL discussed at the beginning of the text.

3 Methodology

3.1 Activities to be developed

Based on the 6 objetives descrives in section 2, we define the following activities to be developed throughout the project:

1. **Study of dependent types**, when we will take a theoretical approach on the subject, studying the literature on type theory available in books, articles, and others, mostly in the collection of libraries affilited with the University of São Paulo (USP).

2. **Selection and study of algorithms and models in data-centric programming**, when we define which constructs used in data-centric programming will be implemented in this project, using as our main reference the pandas library in Python.

3. **Implementation of algorithms**, to be done in Lean 4.

4. **Definition and implementation of the DSL**, in which we define the syntax and semantics of our language, having already known the necessary requirments from previous steps.

5. **Evaluation of results**, when we will conduct a comparative study between the developed solution and other existing works.

6. **Documentation and writing of the final thesis**.

3.2 Schedule

Based on the tasks enumerated earlier, we have the following schedule:

Table 1: Activity schedule.

Step	**Month of year**									
	3	4	5	6	7	8	9	10	11	12
1. Dependent types study	x	x	x							
2. Algorithm selection		x	x							
3. Algorithm implementation			x	x	x	x				
4. DSL implementation					x	x	x			
5. Result evaluation						x	x	x	x	
6. Documentation								x	x	x

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN: 0321486811.

[2] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda - a functional language with dependent types. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 73–78, Munich, Germany. Springer-Verlag, 2009. ISBN: 9783642033582. DOI: 10.1007/978-3-642-03359-9_6.

[3] E. Brady. Idris 2: quantitative type theory in practice, 2021. arXiv: 2104.00480 [cs.PL].

[4] G. Dowek and J.-J. Lévy. *Introduction to the Theory of Programming Languages*. Springer London, 2011. DOI: 10.1007/978-0-85729-076-2.

[5] R. A. Eisenberg. Dependent types in haskell: theory and practice, 2017. arXiv: 1610.07978 [cs.PL].

[6] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2012. ISBN: 1107029570.

[7] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium '73*. Volume 80, Studies in Logic and the Foundations of Mathematics, pages 73–118. Elsevier, 1975. DOI: 10.1016/S0049-237X(08)71945-1. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08719451.

[8] J. McKinna. Why dependent types matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 1, Charleston, South Carolina, USA. Association for Computing Machinery, 2006. ISBN: 1595930272. DOI: 10.1145/1111037.1111038.

[9] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892.

[10] L. d. Moura and S. Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, Lecture notes in computer science, pages 625–635. Springer International Publishing, Cham, 2021.

[11] U. Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[12] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN: 0262162091.

[13] D. Watt and W. Findlay. *Programming Language Design Concepts*. John Wiley & Sons, 2004. ISBN: 9788126505272.

[14] P. Wegner. Programming languages—the first 25 years. *IEEE Transactions on Computers*, C-25(12):1207–1225, 1976. DOI: 10.1109/TC.1976.1674589.