University of Innsbruck
Faculty of Mathematics, Computer Science and Physics

Department of Computer Science

Seminar Report

The Epigram Programming Language

A Comprehensive Overview

by

Ilyas Satik
Matriculation ID.: 12044902

Submission Date: July 15, 2023
Supervisor: assoz. Prof. Dr. Cezary Kaliszyk

Contents

List of Figures

1 Introduction

Epigram is an intriguing language in the landscape of programming languages. It is a dependently typed functional programming language developed by Conor McBride

and others at the University of Durham and the University of Nottingham. The initial design of Epigram focused on interactive development, aiming to create an environment where code and its specifications evolve concurrently.

2 Features of Epigram

Epigram has several unique and non-standard features that set it apart from other languages. These features largely revolve around the concept of dependent types, interactive programming, and its unique approach to type definition.

2.1 Unique Type Definition

In Epigram, types are defined interactively and spread over two dimensions, with parameters and their types specified on the left, and the body of the definition on the right. This approach has the advantage of providing precise, expressive, and interactive type definitions. However, it also comes with the downside of a steep learning curve, potentially verbose notation, efficiency concerns due to extensive compile-time checks, and a limited ecosystem due to Epigram's niche status. These features, while intriguing, do add to the complexity of programming in Epigram. It requires a different mindset from conventional programming, as well as a strong understanding of type theory and formal logic.

2.1.1 How to read Epigram code?

1. Data Declaration The code begins by defining the concept of natural numbers (denoted as Nat). This is done using the data keyword, which starts the declaration of a new data structure. Here, Nat : * defines the new data structure Nat. The constructors for this data structure are given in the where clause: zero and suc n. These represent the number zero and the successor of the number n, respectively.

2. Let Declaration After defining the Nat data structure, the code creates a function named plus using a let declaration. This function takes two natural

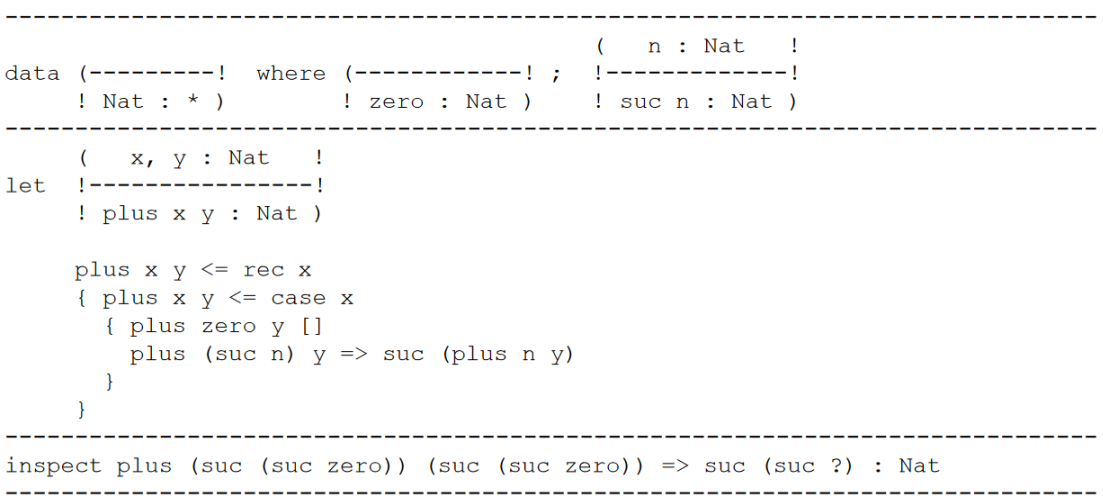

Figure 1: Epigram Code

numbers, x and y, as input and also outputs a natural number.

3. Function Definition Following the function signature, the behaviour of the plus function is defined. This function uses recursion on the first argument, x, and pattern matching on x to define addition. If x is zero, the function does not yet have a defined behaviour (indicated by []). If x is suc n (the successor of some natural number n), the function calls itself with n (the predecessor of x) and y, then takes the successor of the result. This mimics the behaviour of addition: adding n and y, then incrementing the result by one, is equivalent to adding suc n and y.

4. Inspect Declaration The code concludes with an inspect declaration, which evaluates the plus function with two instances of suc (suc zero) (which represents the number 2) as input, expecting a natural number as output. The ? in suc (suc ?) indicates an unknown value that the user expects to be filled in by the function's output. In this case, the expected output would be suc (suc (suc (suc zero))). McBride (2005b)

2.1.2 Advantages of Epigram's Approach to Type Definitions

1. Precision Epigram's two-dimensional notation and dependent types allow for incredibly precise type definitions. Programmers can specify their intentions

clearly and encode sophisticated properties directly into the types, reducing the chance of runtime errors.

2. Program Verification The strict and expressive type system enables advanced forms of program verification. You can ensure that the function behaves as expected for all inputs, which is especially valuable in critical systems where bugs could have severe consequences.

3. Interactive Development The interactive nature of type definition allows for a rich dialogue between the developer and the language environment, helping in understanding the problem better and leading to correct implementations.

2.1.3 Disadvantages of Epigram's Approach to Type Definitions

1. Learning Curve The syntax and concepts of Epigram, such as dependent types and interactive programming, are quite different from those in mainstream languages. It can be challenging to learn, especially for those without a background in type theory or formal logic.

2. Verbose Notation The explicit nature of type definitions can lead to verbose and complex code, especially for larger programs. This can be daunting and may reduce readability.

3. Efficiency Concerns While dependent types provide great safety and correctness guarantees, they may lead to a trade-off with efficiency. The extensive compile-time checks can slow down the development process.

4. Limited Ecosystem Epigram is a niche language, so it lacks the extensive libraries, tools, and community support available for more popular languages. This can make it harder to find resources for learning the language or solving specific problems.

2.1.4 Two-Dimensionality Across Programming Languages

In the field of programming languages, the majority of mainstream, general-purpose languages adopt a one-dimensional syntax. Python, while fundamentally adhering

to this model, gives a semblance of two-dimensionality due to its syntactical conventions, specifically the use of indentation to delineate blocks of code and the application of list comprehensions and nested data structures. Languages like R and MATLAB, primarily used in the scientific and engineering domains, while one-dimensional in syntax, afford strong support for two-dimensional data structures—data frames and matrices, which are instrumental in data analysis, statistics, and numerical computation.

Visual or block-based languages such as Scratch, Blockly, and Max/MSP, on the other hand, offer a more literal interpretation of two-dimensionality. These languages utilize a two-dimensional, graphical interface for coding, either through a block-based or node-based approach, which is visually oriented and typically used in educational or multimedia contexts, respectively.

Esoteric languages provide an additional perspective. APL and its descendant J are one-dimensional but use a dense and expressive syntax that might give an impression of two-dimensionality. Whitespace, an esoteric language, interprets various forms of whitespace as commands, enabling unconventional two-dimensional code structures.

However, Epigram presents a unique approach to two-dimensionality in programming languages, specifically using a two-dimensional syntax for type definitions. This is a part of its broader focus on dependent types and interactive programming—an approach that is not commonly found in mainstream languages and that underscores the innovation and distinctiveness of Epigram.

2.2 Dependent Types and Their Constructors in Epigram

2.2.1 Dependent Types in Epigram

Dependent types, the cornerstone of Epigram's design, allow types to be predicated on values, allowing programmers to specify and verify more properties of their code statically. For example, we can define a type of lists of a given length, or a type of sorted arrays.

Here, the code defines another data structure, Vec, representing a vector of

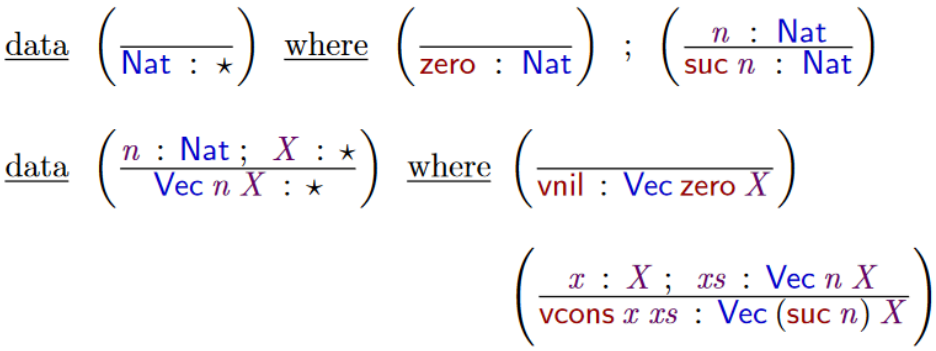

Figure 2: dependent types

elements of type X and length n. This structure is an example of an inductive family, a collection of data types defined mutually and systematically, indexed by other data types (in this case, Nat). The Vec structure has two constructors:

vnil, representing an empty vector, and

vcons, representing the addition of an element x of type X to a vector xs of length n, resulting in a vector of length suc n (the successor of n). These kinds of dependent types make sense in the context of rigorous specification and proof systems. They are used extensively in Epigram and have become more widely accepted and incorporated in languages like Idris and Agda, although they are still considered quite esoteric in mainstream programming.

2.2.2 Structurally Recursive Programming with Dependent Families

One of Epigram's key features is its support for structurally recursive programming with dependent families. These dependent families, or types indexed by values, allow for a high degree of expressiveness in type definitions. In Epigram, we can construct the basic apparatus for structurally recursive programming using standard induction principles and heterogeneous equality.

2.2.3 Heterogeneous Equality in Epigram

Heterogeneous equality is a crucial concept in Epigram. It allows for the comparison of two elements of potentially different types, where those types are themselves propositionally equal. This form of equality is more general than homogeneous

equality, which only allows the comparison of two elements of the same type. In the context of dependent types, heterogeneous equality provides a powerful tool for expressing and proving properties about programs.

2.2.4 Unification Constraints and First-Order Terms

Epigram also represents unification constraints as equational hypotheses, reducing them where possible. Unification, the process of finding a substitution of types that makes two types equal, is a key process in type checking and inference in dependently typed languages. Epigram's approach to unification is complete for all first-order terms composed of constructors and variables, providing a robust mechanism for type inference.

2.2.5 Mutual Inductive Definitions in Epigram

Furthermore, Epigram extends its constructions to mutual inductive definitions, allowing for the definition of multiple interdependent types. The constructors of one type can refer to another type and vice versa, enabling the expression of complex data structures. This feature showcases the power of dependent types in expressing intricate relationships between data. Conor McBride (2004)

Having explored the intricacies of dependent types and constructors in Epigram, we can see how these features push the boundaries of what is expressible in a programming language. As we transition to our next topic, interactive programming in Epigram, it's worth noting that future work in this area will likely explore the adaptation of generic functional programming for programs and proofs in Type Theory. This promises to further enhance the expressiveness and power of dependently typed languages, and potentially open up new possibilities for interactive programming as well.

2.3 Interactive Programming in Epigram

Epigram champions a unique style of interactive programming, where the programmer and the machine work together to construct a program. This interaction often

follows a "dialogue" format, where the programmer provides partial information, the machine responds with a transformed context and goals, and the programmer then continues the dialogue based on this response.

For example, in Epigram, a programmer might start with a function definition with a hole, like this:

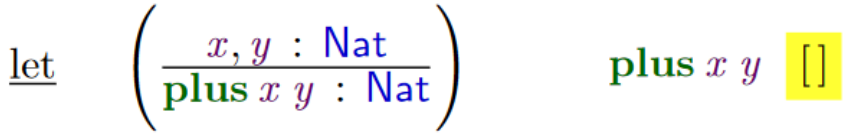

Figure 3: data definition

When implementing the function definition, the programmer's main task is to fill in the "bits in the boxes", which are the core parts of the function. The rest of the code, including the type declarations and the structure of the function, is generated by the machine based on the programmer's input. This is a key feature of Epigram and similar languages: the programmer specifies the overall plan for the function, and the machine helps to implement that plan. This approach can make the programming process more efficient and help to ensure that the function behaves as expected. McBride (2005a)

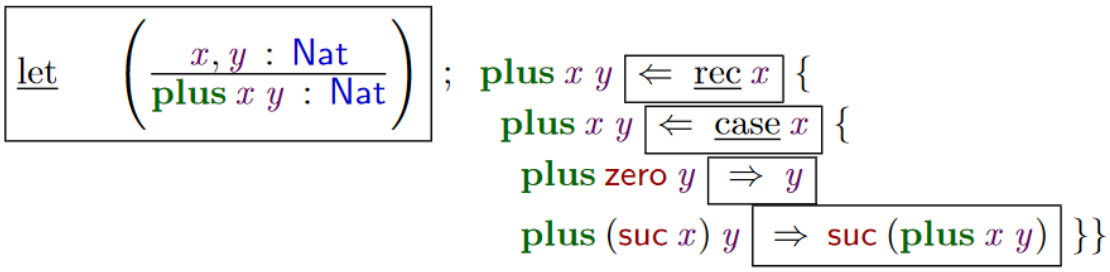

Figure 4: interactive programming

Epigram also supports the development of incomplete programs with unfinished sections, referred to as "sheds". The type checker is forbidden to tread in these sheds, allowing programmers to develop programs interactively. The machine shows the available context and the required type wherever the cursor may be.

The interactive development of a program in Epigram is a kind of dialogue. The system poses the problems—the left-hand sides of programs. Solutions are supplied

by filling in the right-hand sides, either by directly giving the program's output or by invoking a programming pattern which reduces the problem to subproblems which are then posed in turn.

Epigram's approach to meta-variables follows McBride's OLEG system. Meta variables represent not only the missing contents of sheds, but also all the unknowns arising from implicit quantification. The latter is resolved, where possible, by solving the unification constraints which arise during type checking.

These interactive features, while intriguing, do add to the complexity of programming in Epigram. It requires a different mindset from conventional programming and a strong understanding of type theory and formal logic. However, the benefits of this interactive approach include a more direct connection between the programmer's intentions and the resulting program, as well as the ability to develop programs incrementally with constant feedback from the type checker. McKinna (2006)

3 Dedicated Purposes

The Epigram language was born out of a need for a language that can provide rigorous proofs of program correctness, a core tenet of dependent type theory. The creators aimed to leverage the strength of dependent types and blend it with an interactive programming model to create a potent environment for program specification, development, and verification.

If Epigram were to disappear, programmers in need of the guarantees that Epigram provides would likely gravitate towards other dependently-typed languages like Idris, Agda, or Coq. These languages also offer rich type systems that can encode intricate properties of programs, and they share many features and concepts with Epigram, making them suitable alternatives.

The creators of Epigram did achieve their goal in that they successfully designed and implemented a language that uses dependent types and interactive programming to help developers write more correct and reliable code. Epigram has also made a lasting impact on the field of programming languages by demonstrating novel concepts and approaches that have since been picked up by other languages.

4 Conclusion

Epigram, while relatively esoteric and complex to master, stands as a testament to the power of type theory and interactive programming in the pursuit of correct software. Its features, while not standard in the general sense, have nevertheless had a profound impact on a niche area of programming language design, and continue to inspire other languages in the realm of dependent types and formal verification.

References

Conor McBride, Healfdene Goguen, J. M. (2004). *A Few Constructions on Constructors*. Springer, Berlin, Heidelberg.

McBride, C. (2005a). *Epigram: Practical Programming with Dependent Types*. Springer, Berlin, Heidelberg.

McBride, C. (2005b). The epigram prototype: a nod and two winks. Technical report, http://www.e-pig.org/.

McKinna, J. (2006). *Why dependent types matter*. Association for Computing Machinery.