Readability of Domain-Specific Languages: A Controlled Experiment Comparing (Declarative) Inference Rules with (Imperative) Java Source Code in Programming Language Design

Kai Klanten[1], Stefan Hanenberg[1][a], Stefan Gries[2] and Volker Gruhn[1][b]

[1]*University of Duisburg–Essen, Essen, Germany*

[2]*codecentric AG, 42697 Solingen, Germany*

Keywords: Domain-Specific Languages, Empirical Study, User Study.

Abstract: Domain-specific languages (DSLs) play a large role in computer science: languages from formal grammars up to SQL are integral part of education as well as industrial applications. However, to what extent such languages have a positive impact on the resulting code, respectively on the activity of writing and reading code, mostly remains unclear. The focus of the present work is on the notation of inference rules as they are applied in programming language education and research. A controlled experiment is introduced where given type rules are either defined using a corresponding DSL or the general-purpose language Java. Thereto, a repeated N-of-1 experiment was executed on 12 undergraduate students in computer science, where the participants had to select for a randomly generated typing rule and a randomly generated term from a list of possible types the correct one. Although the formal notation of inference rules is typically considered as non-trivial (in comparison to code in general–purpose languages), it turned out that the students were able to detect the type of a given expression significantly faster than using Java ($p < .001$, $\eta_p^2 = .439$): on average, the response times using Java were almost twice as much as the response times using inference rules ($\frac{M_{Java}}{M_{inference}} = 1.914$). Furthermore, the participants did less errors using inference rules ($p = .023$). We conclude from that the use of inference rules in programming language design also improves the readability of such rules.

1 INTRODUCTION

It is quite widespread to use different notations for different programming related tasks. Such notations are commonly called domain-specific languages (DSL, cf. (Mernik et al., 2005)). Examples for DSLs that are often taught and applied are markup languages (such as XML, HTML, Latex, etc.), query languages (such as SQL) or grammar languages (such as BNF). However, among the list of typical domain-specific languages, there is one kind of language that is often not mentioned despite its central role in education and research: the notation used for defining programming language semantics and types.

In programming language design, it is common to define the semantics of a language (and its type system) using inference rules. Such rules are based on a notation that has its foundation in natural deduction that originates from the work by Gentzen from 1935 (Gentzen, 1935). Gentzen defined natural deduction in a way where premise and conclusion were graphically separated by a horizontal line. In programming language design, such notation was adapted to define the semantics and the type system of programming languages in a similar way. The resulting notation by Wright and Felleisen was not only used to describe the semantics and type system, but also to perform type system proofs on them (Wright and Felleisen, 1994). In the following, we will call this notation (declarative) inference rules.

Actually, despite the fact that inference rules are standard in teaching and research, it is also standard to translate these rules into source code. For example, Pierce's book "*Types and Programming Languages*" (Pierce, 2002), that is widespread in teaching, uses inference rules to describe programming languages and then describes how these rules can be

[a] https://orcid.org/0000-0001-5936-2143
[b] https://orcid.org/0000-0003-3841-2548

Klanten, K., Hanenberg, S., Gries, S. and Gruhn, V.
Readability of Domain-Specific Languages: A Controlled Experiment Comparing (Declarative) Inference Rules with (Imperative) Java Source Code in Programming Language Design.
DOI: 10.5220/0012857800003753
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 19th International Conference on Software Technologies (ICSOFT 2024)*, pages 492-503
ISBN: 978-989-758-706-1; ISSN: 2184-2833
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

implemented using a given programming language. The reason for this translation is quite plausible: in the end a programming language is designed and one wants to experience how this language works by executing programs.[1] However, it is also noteworthy that the inference rules are the core of Pierce's book: it is not the case that the implementations of these rules play the same role as the formal descriptions.

From a teaching perspective one needs to ask whether it is necessary to introduce two different notations, both describing the same phenomenon, because it is questionable whether learners get an additional benefit from one or the other notation (respectively from both in combination). The implementation language – i.e., the language into which the inference rules are translated – does not seem to be a candidate to be removed in teaching, because this language permits in the end to execute the final product. However, whether declarative inference rules should play such a central role in is unclear. Asked in a more provocative manner: "*Wouldn't it be more helpful to describe a language only using an imperative general-purpose language?*"

Answering this question has a practical implication for the authors of the present paper. We teach programming language design at our institute in the traditional way using (declarative) inference rules. At the same time, our students implement the defined language in Java (in an imperative style). And we often get feedback from our students that the inference rules are unnecessarily complex and rather decrease their progress in learning (instead of promoting it).

We think our question could be even generalized: Under the assumption that the often-heard objection is correct, one should ask, whether the massive use of inference rules at programming language conferences (such as PLDI, ICFP, POPL, etc.) helps researchers and reader of papers or whether the use of such notations is counterproductive.

Considering that inference rules in programming language design are not new, one would expect that the effect of that notation on the readability is well-known and corresponding empirical studies can be easily found. However, this is not the case. Instead, it is a well-documented phenomenon that in programming language research empirical studies play hardly any role. According to the study by Kaijanaho, the number of human-centered studies using randomized controlled trials (RCTs) in the field of programming language design up to 2012 was just 22 (Kaijanaho, 2015, p. 133). This low number was confirmed by Buse et al. (Buse et al., 2011), and even a broader view on the field of software construction in general does not lead to a substantial higher number of studies (see for example (Ko et al., 2015; Vegas et al., 2016)). Hence, taking into account that empirical studies play only a subordinate role in programming language research, it is quite understandable that not much empirical evidence exists on the effect of using the notation of declarative typing rules.

One could consider our question from a slightly different angle: instead of concretely referring to the notation of (declarative) inference rules, one could ask what evidence exists on the use of imperative instead of declarative languages. Although one can find some smaller studies on that topic, the results are far from being conclusive.

To answer the question whether the use of declarative typing rules have a measurable benefit in comparison to imperative typing rules, the present work introduces a controlled experiment executed on 12 students. In the experiment, 64 tasks were given, and participants had to determine the type of a given expression. As a measurement we used response times and correctness. It turned out that the students required for the Java-based notation almost twice the time compared to the inference rules ($p < .001$, $\eta_p^2 = .443$, $\frac{M_{Java}}{M_{inference}} = 1.922$). With respect to correctness, the Java source code increased the number of errors.

2 DECLARATIVE VERSUS IMPERATIVE LANGUAGE DEFINITIONS

In order to explain the motivation for the present work, we explain two ways to define the semantics and the type system of a programming language: the declarative way and the imperative way. The underlying programming language in this section is the lambda calculus which is a basic programming language that is typically taught in courses such as "foundations of programming languages". To describe such languages, it is necessary to describe its syntax and its semantics. While for syntax definitions languages such as regular expressions and grammars are used (see for example (Aho et al., 1986, part I and II)), the semantics of such languages is typically described using declarative inference rules.

[1] Actually, there are other reasons for such transforming: one wants to proof language characteristics with the aid of proof-assistant systems (see for example (Dubois, 2000; Grewe et al., 2015)).

2.1 Declarative Language Definition

In order to define the semantics of a programming languages and (in case it is statically typed) its type system, it is widespread to use inference rules. Figure 1 contains the application such inference rules. The rules are taken from the teaching book by Pierce (Pierce, 2002, p. 103) but the same rules – especially those ones for type checking – can be found elsewhere as well (see, e.g., (Bruce, 2002, p. 126)).

$$\text{(E-App1)} \qquad \frac{t_1 \rightarrow t_1'}{t_1\, t_2 \rightarrow t_1'\, t_2}$$

$$\ldots$$

$$\text{(T-App)} \qquad \frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}}$$

$$\ldots$$

$$\text{(T-if)} \qquad \frac{\Gamma \vdash c : \mathit{Bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathit{if}(c)\ \mathit{then}\ t_1\ \mathit{else}\ t_2 : T}$$

Figure 1: Example evaluation and typing rules (taken from (Pierce, 2002, p. 103) for the statically typed lambda calculus. The evaluation and typing rules are described in a declarative style based on natural deduction.

The first rule E-App1 describes parts of lambda's semantics. It is part of the semantics of the so-called *application*, where a term t_2 is applied to a term t_1 (which corresponds to a function call). It expresses that in case t_1 can be reduced to a term t_1' (because t_1 is possibly a function call itself), the term $t_1\, t_2$ is reduced to a term $t_1'\, t_2$ (i.e., the left-hand side is reduced).[2]

The second rule T-App describes the typing rule for an application. It says that (for a given environment Γ) the application $t_1\, t_2$ has the term T_{12} if the term t_1 has the function type $T_{11} \rightarrow T_{12}$ (i.e., a type T_{11} can be applied to the function and the result type of that function is T_{12}) and if the term t_2 has the required type T_{11}.

In order to exemplify how a typing rule for a new language construct is described, Figure 1 contains the typing rule for the language construct `if` where the term in the condition is required to be of type Boolean and the type of the `then` and `else` branch need to be the same (which finally is the type of the `if` expression).

2.2 Imperative Language Definition

Although the rules from Figure 1 are rather trivial for people trained in programming language design, their meaning is not obvious for people who are not trained in such notations. However, it is not hard to translate the rules into imperative languages.

```java
class Application implements LambdaTerm {
  LambdaTerm left, right;
  ...
  LambdaTerm reduce() {
    if(left.isReducible()) {
      return new Application(
                     left.reduce(),
                     right.clone()
                 );
    }
    ...
  }

  Type typeOf(Environment e) {
    FunctionType f_type
      = (FunctionType) left.typeOf(e);
    Type right_type = right.typeOf(e);
    if(f_type.left().equals(right_type)) {
      return f_type.right();
    } else {
      throw ...;
    }
  }
}
```

Figure 2: Possible translation of the declarative evaluation and typing rule using the programming language Java. Method reduce is the translation of the reduction rule, typeOf is the translation of the typing rule.

Figure 2 contains one possible way to describe the rules for applications from Figure 1 in an imperative style in Java. The language construct *application* is defined by a class and the methods `reduce()` , respectively `typeOf(Environment)` represent the imperative translation of the declarative rules. In order to execute the code, it is necessary to provide a corresponding API (such as an interface `LambdaExpression` with declared methods `reduce()`, `isReducible()`, `typeOf(Environment)`, `clone()` as well as an interface `Type` with a sub-interface `FunctionType` with the methods `left()` and `right()`).

There are obviously multiple ways to implement reduction and typing rules using languages such as Java. The here proposed approach is based on an object-oriented design (each language construct has its own class) with overridden methods (such as `reduce()`, `isReducible()`, `typeOf(Environment)`). However, the code in the methods correspond to the typical imperative style. For example, method `typeOf(Environment)` stores the intermediate results in local variables where a follow-up condition operates on such variables.

The imperative code contains fragments that are not directly contained in the declarative notation. For example, `typeOf(Environment)` in Figure 2

[2]The whole semantics for applications require some more rules, but the goal here is only to introduce into the use of inference rules.

contains a type cast to type `FunctionType`. In case this cast fails, the term has no valid type.[3] In the declarative description, this is expressed by the absence of a typing rule where the precondition is fulfilled.

2.3 Pros and Cons of Both Notations from the Readability Perspective

Again, we should emphasize that the notation for declarative inference rules originate from a different purpose, namely the execution of type proofs on the language. However, our motivation comes from a different angle, namely the readability of the rules.

Obviously, the inference rules are much shorter: just comparing for example the rule `E-App1` from Figure 1 and the code snippet in method `reduce()` makes this obvious – and it should be emphasized that not even the other methods (such as `is_reducible()`) are contained. Hence, one could simply state that "being shorter" is the reason why inference rules should outperform the source code.

However, it is possibly harder to read the inference rules, because such rules possibly require from the reader to handle a permanent change in the reading direction. In the imperative style, it is clear that the code can be read top to bottom[4]. For example, when someone reads the typing rule (method `type_of(Environment)`), one sees that first the type of field `left` is determined (which has to be of type `FunctionType`, see line 16), then the type of field `right` left is determined. In case the input type of the function type matches the type of the latter expression (line 18), the function type's output type is returned (line 19). For the declarative notation (typing rule `T-App` in Figure 1), one probably starts with the conclusion and sees that a type T_{12} is returned. Possibly, one starts searching for this type in the condition (and sees that it is the right-hand side of a function type). Possibly, one then looks at the term t_1 that has this function type and one possibly looks again in the conclusion where this term appears. Then, one possibly reads the second condition and sees that t_2 needs to have a certain type T_{11} Then, possibly one checks where in the other condition T_{11} appears.

[3]In a practical implementation, one would catch an exception to make explicit that there is a type error in the term with some understandable error messages.

[4]We are aware that in general source code is not read from top to bottom as shown by Busjahn et al. (Busjahn et al., 2015). However, we still think that the here used imperative code is read top to bottom, because just one single method needs to be understood.

Hence, we think it is possible that readers are confronted with the problem that it is not clear where to look first and where to look next. And finally, we think that there is some cognitive effort to match the multiple occurrences of the different identifiers (such as the type T_{11}). Altogether, we do think that it is not obvious whether the imperative or the declarative notation is more readable.

3 RELATED WORK

As argued in the introduction, the present work can be considered from two different perspectives. The first one is on the effect of a specific domain-specific language on the readability of code, the second perspective is on the possible difference between declarative and imperative languages with respect to usability or readability. Hence, we consider in the following works as related, where empirical evidence is given either on domain-specific languages or on the comparison of declarative and imperative languages.[5]

3.1 Controlled Experiments on Domain-Specific Languages

Kosar et al studied in 2010 the possible effect of using the DSL XAML in comparison to C# Forms (Kosar et al., 2010). 35 programmers answered 11 questions on a given code fragment (defined either via XAML or C# Forms) that created a GUI element. It turned out that on average all questions (except one) had a higher success rate when code was represented via XAML and the overall mean success rate using XAML was 37.62% higher than for C# Forms.

In 2012, the same approach as the previous study was followed but now applied to two more domain-specific languages (i.e., altogether applied to three different domains, each with its own domain-specific language) (Kosar et al., 2012): features diagrams (FDL as DSL versus a FD library in Java as GPL) and graphical descriptions (DOT as DSL versus a GD library in C as GPL). Again, it turned out that for

[5]One could argue that the here studied inference notation is also a graphical notation (because of the graphical separation of condition and conclusion) and in that context, additional works could be mentioned that study the effect of graphical notations (see for example (Shneiderman et al., 1977; Hollmann and Hanenberg, 2017) just to mention a few). However, we do not think that the inference notation should be considered as a graphical one, because in the end the horizontal line is the only graphical element in the notation.

all DSLs the average success rate of participants was higher than for the GPL counterparts.

In 2017, Johanson and Hasselbring studied the effect of the Sprat Ecosystem DSL, a DSL in the domain of marine ecosystems, in comparison to C++ code (Johanson and Hasselbring, 2017). 40 participants participated in an experiment where (among others) the correctness in comprehension tasks and the time spent on comprehension tasks was measured. The DSL lead to higher correctness (on average an increase by 61%) and required less time (31%) in comparison to the application of the GPL.

In 2022, Hoffman et al. studied the effect of the DSL Athos – a DSL from the domain of traffic and transportation simulation and optimization – in comparison to the use of the Java-based library JSpirit (Hoffmann et al., 2022). The study, which was designed as a crossover-trial, was conducted among two groups consisting of altogether 159 participants. The general outcome of the study was that the DSL led to a higher efficiency.[6] Additionally, the participants showed a higher user satisfaction using the DSL.

3.2 Declarative versus Imperative Language Constructs

As soon we speak about possible differences between declarative and imperative constructs, it turns out that such studies are mainly in the domain of programming, with a special focus on lambda-expressions – a traditional declarative language construct that was integrated in imperative languages over the last decades.

Uesbeck et al. (Uesbeck et al., 2016) compared, whether lambda expressions in C++ (in comparison to loops) had an effect on 58 participants: it was the participants' task to create source code that iterated over a given data structure. The main result of the study was, that lambda expressions required significantly more time. Additionally, the study analyzed error fixing times. Again, the study revealed a positive effect of traditional loops.

A similar study, but applied to a very specific Java API (Java Stream API) was performed by Mehlhorn et al. (Mehlhorn and Hanenberg, 2022) where the Stream API (that uses lambda expressions) was compared to traditional loops. The experiment measured the time required by 20 participants to detect the result of collection-processing operators on collections. The result was in contrast to the result of the previous study: it required participants less time to identify the result if the (declarative) Stream API was used.

Another study that also had lambda expressions in its focus was performed by Lucas et al. (Lucas et al., 2019) where code that was changed from imperative constructs to lambda expressions was evaluated by developers. It turned out that about half of the developers considered the introduction of lambda expressions as an improvement of the code. Most interestingly – when compared to the two previously mentioned studies – developers perceived some code migrations towards lambda expressions negatively when for-loops were replaced.

A study that is not too closely related to programming is the ones by Pichler et al. (Pichler et al., 2011). The authors studied the difference between declarative and imperative business model languages. Students were given two types of tasks and four questions per task[7] on models that were formulated either using a declarative or an imperative language. The dependent variables response time and correctness were influenced by the choice of the language.

Davulcu et al. studied the effect of introducing an imperative language feature into the declarative database query language SQL (Davulcu et al., 2023): a sequence operator was added to SQL that permitted to define SQL statements in a stepwise manner. A study on 24 participants revealed that the introduction of the imperative feature reduced the time participants required to define an SQL statement for a given task: participants required only 52% of the time required for the same task using regular SQL.

4 EXPERIMENT DESCRIPTION

4.1 Initial Considerations and Design Decisions

The goal is to measure the readability of rules either defined as declarative inference rules or using an imperative general–purpose language. Hence, we need to define the GPL to be used, the rules be used in an experiment, what task should be given to participants, how participants can respond to these tasks, and what should be measured in the experiment.

[6]The study disinguishes between what language has been used by the participants first – and it turned out that there was a larger carry-over effect between the groups starting with the DSL or the GPL.

[7]From the paper, it cannot be derived what questions were asked or to what extent the questions were related to the declarative or imperative nature of the given model.

The first step was to define the GPL. We chose Java for practical reasons: Java is taught at our university and hence, we assume that the participants – we assumed that these participants would be students from our university – are familiar with the language.

Next, we decided to use typing rules in the experiment. The reason for this is rather trivial: if some simple typing rules are given, it is easy to define a new typing rule. For example, if it is known that `true` is of type `Bool`, and `1` is of type `Num`, one can easily determine that a term `construct true 1` (in conjunction with a typing rule $\frac{\Gamma \vdash t_1{:}Bool \quad \Gamma \vdash t_2{:}T}{\Gamma \vdash construct\ t_1\ t_2{:}\ T \rightarrow T}$) has the type `Num`→`Num`. Furthermore, it is easy to determine that a term `construct 1 1` has no valid type, because `1` has not the type `Bool`). However, in order to have no trial typing rules, we still see the need to have typing rules for some other language constructs. We decided to generate typing rules for given language constructs in the following way: we selected arbitrary identifiers from a dictionary and generated a typing rule for them consisting of function types `Bool` and `Num` (i.e., the identifiers become literals in the language). Figure 3 illustrates three of such typing rules for the words `Couch`, `Book`, and `Number`. In order to not confuse participants, we used the same set of generated typing rules throughout the experiment.

Next, we generated typing rules (as tasks) consisting of four or five terms. In order to not confuse participants with additional identifiers, we decided to define such a rule without any additional identifiers. Instead, a term consisting of (for example) four literals is given and the rule simply consists of these four elements. For example, the term in Figure 3 consists of four literals `Couch Book 18 Couch` and the typing rule consists of four terms as well (t_1 to t_4).

In order to give participants the ability to give a quick feedback, we decided to generate possible answers. We generated six answers and gave additionally the ability to express "Error" or "None".

We need to take into account that in case a term has no valid type, that the effort for determining such error might be less than determining a valid type of an expression. Because of that, we determine for each type rule (and type expression) how many terms need to be read. For example, if the term `true Book 18 Couch` would have been given in Figure 3, the reader just needs to read the first precondition (which requires that the first literal needs to have a function type) in order to determine that the term has no valid type.

In order to take this difference in effort into account, we see the need to consider the terms that need to be read in the experiment definition as a separate variable.

Next, we also need to decide how the Java source code should be represented in the experiment. We decided to use the same representation as shown in Figure 2: a method is shown where the types of each field are assigned to local variables. Then, an if-statement checks the preconditions, and a return statement determines the resulting type.

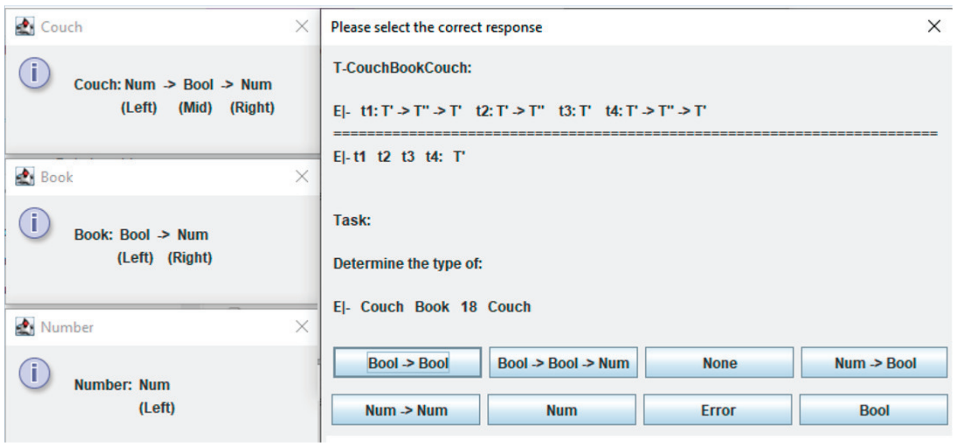
With respect to the measurements, we decided to measure the response times, i.e., the time it took for a participant to answer, and the correctness of a response. The time measurement starts when a task is shown to the participant. It stops when a response button was pressed.

A general design decision for the experiment is, to run the experiment as a repeated N-of-1 experiment (see (Hanenberg and Mehlhorn, 2021)), where a participant receives all (randomly ordered) treatment combinations (with multiple repetitions). The benefit of this experiment design is that relatively few participants are required – and that a statistical analysis of the results is even possible on one single participant. In order to not confuse participants with frequently changing notations, we decided to define block of tasks, each with the same used notation.

4.2 Experiment Layout

The experiment was designed as a repeated N-of-1 experiment where all subjects received 64 typing rules. The experiment layout is as follows.

- **Dependent Variables:**
  - **Response Time:** The time until an answer was given. The measurement starts, when a rule is shown to the participant. The moment an answer was given, the time is stopped.
  - **Correctness:** Whether or not the given answer was correct (yes / no),
- **Independent Variables:**
  - **Notation:** With the treatments inference rules or Java code.
  - **Terms to Read:** The number of terms that need to be read (2–5).
  - **Valid Term:** Whether or not the term has a valid type (yes/no).
- **Fixed Variable:**
  - **Repetitions:** Altogether, there were four repetitions per treatment combination. I.e., 64 (=2x4x2x4) typing rules (and terms) were shown to the participants.



```
Couch: Num -> Bool -> Num
       (Left)  (Mid)  (Right)

Book: Bool -> Num
      (Left)  (Right)

Number: Num
        (Left)
```

```
T-CouchBookCouch:

E|-  t1: T' -> T'' -> T'   t2: T' -> T''   t3: T'   t4: T' -> T'' -> T'
==========================================================================
E|- t1  t2  t3  t4:  T'


Task:

Determine the type of:

E|-  Couch  Book  18  Couch
```

Figure 3: Screenshot from the experiment environment (for the inference rules). For illustration purposes, we put four response boxes in one line. In the experiment, all response boxes appeared next to each other. The illustrated term has no valid type and two terms need to be read in order to determine the valid answer "Error".

  - **Ordering:** All participants received a block of 8 tasks per notation (starting with Java code), then, the notation was switched. I.e., the participants received the following order of tasks: (8 Java, 8 inference, 8 Java, 8 inference, 8 Java, 8 inference, 8 Java, 8 inference).
  - **Randomization:** In all blocks, 4 terms had valid types and 4 terms had invalid types. The terms to read were randomly assigned to the blocks. This randomization was initially done (before the experiment execution) and all participants received the same tasks in the same order.
- **Task:** *"Determine the type for the given expression."*

4.3 Experiment Environment

We developed an application for the experiment that permitted on the one hand participants to practice the experiment before running it. Additionally, the same application was used for the experiment itself.

Figure 3 illustrates a screenshot of the environment given to the participants. The window on the right-hand side shows the typing rule (top) and the task (bottom), the windows on the left hand side describe additional typing rules required for the task. The term, whose type should be determined, is `Couch Book 18 Couch`.

In order to solve the task, one has to find out that the first parameter `Couch` has the type `Num→Book→Num`. The second parameter `Book` has the type `Bool→Num`. The typing rule expresses, that the second parameter's input type must match the first parameter's input type. Since this is not the case, the participant does not need to read any further, because this contradiction implies that the resulting term has no valid type. I.e., in order to answer the task correctly, the participant has to click "Error".

The experiment environment also does the measurements. A timer is started once a typing rule is shown and stopped when the participants click on an answer button. After the experiment is finished, a csv file is generated that contains for each task the measured time (and whether the participant gave the correct answer).

4.4 Experiment Execution

For the experiment 12 undergraduate students (6th semester of below) were recruited. Each participant received a training video of 10 minutes that explained the type rules and the experiment in addition to the experiment environment. Additionally, the participants were given a handout of typing rules for terms that appear in the experiment, but which were not defined by the given typing rule.

The experiment was executed on the participants' personal machines. Participants were told upfront that they require a display with a resolution of at least 2560x1600.

The participants were asked to train themselves. Thereto, participants could start the experiment environment with different seed values (where this value is responsible for generating tasks and task ordering in a quasi-random way). In order to start the experiment, the participants did not have to enter a seed (in that case, the experiment was started for all participants with the same seed).

After the csv file was generated, the participants were asked to submit this file to the experimenter.

5 ANALYSIS AND RESULTS

All statistical tests were executed using the statistics package Jamovi 2.3.19. Since two different dependent variables were measured, we report the analysis of the effects of the independent variables on the dependent variables in separate sections, starting with the correctness.

5.1 Analysis of Correctness

In order to analyze the correctness of responses we performed a χ^2-Test on the (independent) variable notation and (the dependent variable) correctness.

It turned out that altogether relative few errors were done: among the 768 responses, only 20 were incorrect (error rate = 2.6%). However, it turned out that the errors were not equally distributed among the notations: 15 errors were done when Java code was shown (error rate = 3.9%) while only 5 errors were done when inference rules were shown (error rate = 1.3%). This difference was statistically significant (p = .023, see Table 1).

A closer look into each participant's results revealed that only one participant did no error, while the maximum number of errors per participant was 4. Running the χ^2-Test on each participant did not lead to any significant difference. Hence, a difference in the number of errors could be detected on all participants (where the Java code leads to more errors), while such difference was not observable on individual participants.

Table 1: χ^2-Test on the independent variable correctness (and the dependent variable notation).

Variable	df	χ^2	p	N_{total}	Kind	Answer	N
Notation	1	5.13	.023	384	Java Code	Correct	369
						Errors	15
				384	Inference	Correct	379
						Errors	5

5.2 Analysis of Reponse Times

We analyzed the time measurements using an ANOVA on the dependent variables notation, participant, terms to read, and valid term. It turned out that all main variables were significant with p < .001 (see Table 2).

The main variable notation has a strong and large effect ($p < .001$, $\eta^2_p = .439$) and on average, it required participants 92% more time to answer if the typing rule was defined using Java instead of a declarative inference rule ($\frac{M_{Java}}{M_{inference}} = \frac{67.0}{35} = 1.914$). However, it also turned out that this mean ratio was influence by other factors as well.

First, notation interacts with the variable terms to read (N * T, $p < .001$, $\eta^2_p = .382$) and the ratio $\frac{M_{Java}}{M_{inference}}$ gets larger the more terms are required to determine the type for the given term ($\frac{M_{Java_2}}{M_{inference_2}} = \frac{29.9}{22.0} = 1.360$, $\frac{M_{Java_3}}{M_{inference_3}} = \frac{44.2}{28.3} = 1.562$, $\frac{M_{Java_4}}{M_{inference_4}} = \frac{65.3}{36.7} = 1.780$, $\frac{M_{Java_5}}{M_{inference_5}} = \frac{129}{52.9} = 2.439$)[8]. And it is worth emphasizing how large the ratio is for 5 terms to read, where Java requires 143.9% more time to read. Additionally, there is a strong (but small) interaction effect between notation and valid term ($p < .001$, $\eta^2_p = .030$). Hence, the effect of the notation should not be interpreted alone, but it should be taken into account how many terms need to be read in order to determine the type of an expression.

The effect that has an even stronger effect on the ratio of means between Java and inference is the participant. The variable participant has a strong and large effect in the experiment ($p < .001$, $\eta^2_p = .506$) – which simply states that participants do largely vary in their response times.[9] However, it is more interesting how it interacts with the variable notation: the interaction between participant and notation is strong and large ($p < .001$, $\eta^2_p = .160$), i.e., the notation effect works different on different participants. While the variable participant interacts with the other variables as well, we think that the interaction N * P is worth to be studied in more detail (taking into account that the main goal of the present study is to study the effect of the notation).

Table 3 describes for each participants whether the participant was reactive on the variable notation (p-value), how large the effect was (in terms of the effect size η^2_p) and how large the ratio $\frac{M_{Java}}{M_{inference}}$ was for each participant. It turned out that two participants (row 11 and 12) were not reactive to the model (at least, the number of data points per subject was not sufficient to detect a significant difference between the notations), but the ratio $\frac{M_{Java}}{M_{inference}}$ still suggests that Java code required more time to read - but in those cases the ratios were not impressive (1.369, respectively 1.283, i.e. Java required 36.9% and 28.3% more time to read). At the same time, $\frac{7}{12}$ participants showed a ratio $\frac{M_{Java}}{M_{inference}} > 2$: in those cases, Java took twice as

[8]The numerical index in the ratios describes the terms to read. For example, M_{Java_3} describes the mean value for Java with three terms to read.

[9]A large variation of participant times is something that can be already found in the literature under the term 10x problem (see for example (McConnell, 2011)), although it should be mentioned that there is not much evidence for concrete factor (see (Bossavit, 2015, p. 36)).

Table 2: Experiment Results for the measured times – Confidence intervals (CI) and means (M) are given in rounded seconds. N describes the number of datasets per treatment(-combination).

	df	F	p	η_p^2	**Treatment**	**N**	**CI$_{95\%}$**	**M**
Notation (N)	1	451.407	<.001	.439	Java Code	384	61.1; 72.9	67.0
					Inference	384	32.8; 37.1	35.0
Participant (P)	11	53.61	<.001	.506	*see Table 3*			
Terms to Read (T)	3	365.27	<.001	.655	2	192	23.7; 28.2	25.9
					3	192	33.3; 39.2	36.3
					4	192	45.9; 56.1	51.0
					5	192	81.1; 100.0	90.7
Valid Term (V)	1	56.70	<.001	.090	Valid (V)	384	48.1; 59.1	53.5
					Invalid (I)	384	44.6; 52.1	48.4
N * T	3	118.779	<.001	.382	Inference / 2	96	19.8; 24.1	22.0
					Java / 2	96	26.2; 33.7	29.9
					Inference / 3	96	25.4; 31.2	28.3
					Java / 3	96	39.5; 48.9	44.2
					Inference / 4	96	32.4; 41.1	36.7
					Java / 4	96	57.0; 73.5	65.3
					Inference / 5	96	48.2; 57.6	52.9
					Java / 5	96	113; 144	129
N * V	1	17.551	<.001	.030	Inference / I	192	29.8; 35.2	32.5
					Java / I	192	57.9; 70.6	64.2
					Inference / V	192	34.1; 40.8	37.4
					Java / V	192	59.7; 79.8	69.8
T * V	3	48.306	<.001	.201	*ommited due to large number of rows*			
P * N	11	9.965	<.001	.160	*ommited due to large number of rows*			
P * T	33	5.146	<.001	.228	*ommited due to large number of rows*			
P * V	11	4.367	<.001	.077	*ommited due to large number of rows*			
N * P * T	33	2.921	<.001	.143	*ommited due to large number of rows*			
N * P * V	33	1.865	.041	.034	*ommited due to large number of rows*			
N * T * C	3	34.014	<.001	.150	*ommited due to large number of rows*			
N * P * T * C	33	.787	.798	.043	*ommited due to insignificance*			

Table 3: Experiment Results for each participant (only p-values, η_p^2, and ratios for the variable notation), participants ordered by ratio $\frac{M_{Java}}{M_{inference}}$.

Rank	p	η_p^2	$\frac{M_{Java}}{M_{inference}}$
1	.002	.151	$\frac{38.8}{18.2} = 2.132$
2	<.001	.214	$\frac{55.7}{26.4} = 2.110$
3	<.001	.190	$\frac{95.1}{46.1} = 2.063$
4	<.001	.196	$\frac{93}{45.5} = 2.044$
5	<.001	.181	$\frac{106}{51.9} = 2.042$
6	<.001	.185	$\frac{74.9}{36.7} = 2.041$
7	<.001	.190	$\frac{101}{49.6} = 2.036$
8	<.001	.183	$\frac{83.3}{42.3} = 1.970$
9	.011	.099	$\frac{42.7}{23.3} = 1.833$
10	.004	.125	$\frac{37.8}{21.9} = 1.726$
11	.106	.042	$\frac{32.3}{23.6} = 1.369$
12	.228	.023	$\frac{44}{34.3} = 1.283$

long as the inference rules.

Despite the large differences between the participants, it is worth pointing out that it is not the point that the fastest participants got most from the inference rules, nor can we state the opposite. Although there is some tendency that those participants who read the Java code fast in the experiment do not get as much from the inference rules as those participants who rather slowly read the Java code, this effect was not significant in the experiment.

Hence, we can conclude that the effect of notation is strong and large in general, but this effect is much influenced by other factors (namely the terms to read and the validity of the term): the larger the number of terms to read, the larger the positive effect of the declarative inference notation on typing rules. However, a strong influence on the difference between inference notation and Java code is the participant and while there were in the experiment participants with a ratio $\frac{M_{Java}}{M_{inference}} > 2$ (with a significant effect of the variable notation), there were two participants who did not reveal a significant effect of the variable notation.

6 THREATS TO VALIDITY

Participants: The present study used students as participants and possibly this influenced the results. It is an often-articulated claim that experiments in software engineering should be mainly executed on professionals. However, we need to emphasize that the distinction between students and professionals is probably too coarse grained as a distinction criterion.

In fact, we have to accept that there is not much evidence on selection criterion of participants and their possible performance in experiments (see for example (Siegmund et al., 2014)). Having said this, we are aware that probably the choice of participants has an effect on the experiment's results.

Terms to Read: The study shows a large effect of the variable terms to read on the time measurements and an interaction effect on the difference between Java code and inference rules. The experiment used between two terms to read up to five terms to read – and the overall effect of notation is determined by this choice. It is unclear to us, what a realistic number for the terms to read is, because it depends on two different characteristics. The first one is the rule itself: if there are five terms to read, it means that a rule's precondition consists of five terms. Taking a look into text books such as the one by Pierce (Pierce, 2002) gives the impression that two to three preconditions are a more realistic scenario for the there mentioned language constructs – but it remains unclear what the average number of preconditions for a typing rule is. Furthermore, the effect depends on the term to be checked: even if there is a larger number of preconditions, it is possible that an incorrect type can be already determined by reading the first precondition. Hence, the general question is, how often one is confronted with incorrect types – i.e., how many terms need to be read in general. Such a question cannot be answered by looking into textbooks or code repositories.

Reading Direction: Our notion of terms to read is much influence by our perspective on how terms are read – and we assume that even the inference rules are read from left to right, from top to bottom. Actually, there is evidence in the literature that reading directions are possibly different. Not only in general, but by certain participants (see for example (Busjahn et al., 2015)). One could argue that a more fined grained measurement technique such as the use of eye trackers could solve this problem. However, one should keep in mind that in case it turns out that participants have different reading strategies and different reading directions, it becomes even harder to define a controlled experiment: in such a case, it is probably necessary to determine upfront from a participant what the reading strategy and direction is.

Measurement Technique and Choice of Answers: Obviously, the experiment is influenced by the choice of answers given to participants. If, for example, the possible answers are "absurd enough" so that is becomes clear for a participant that none of them (except "Error") can match, the choice of the answers would become a major factor for the experiment. In the present experiment, the provided answers were from our perspective plausible, but until now we are not aware how evidence for such a plausibility can be given.

Choice of Typing Rules: The terms were chosen in a way that a typing rule already expresses that in case a term has a function type `A→B`, that this is already explicitly defined in the typing rule. As an alternative, one could have chosen typing rules in a way that for example one type `T` appears, where a latter rule refines this (for example, by a function type `A→B`). We are aware that this possibly has an influence on the absolute measurements in the experiment (because in such a case one needs to keep in mind all constraints of a type), but we are not aware whether this has an effect on the difference between inference rules and Java source code.

Reading Versus Writing: The here introduced experiment focusses on reading time (and errors). Hence, we cannot state whether the notations have an effect on how the rules are written.

Source Code Representation: We are aware that the way how typing rules are expressed in Java has (probably) an effect on the readability of the code. However, we are not yet aware of readability metrics for source code that express (with strong evidence) whether a code presentation is better than the other.[10]

7 SUMMARY, DISCUSSION AND CONCLUSION

The present paper studies the effect of one specific domain-specific language (declarative inference rules) in comparison to a general-purpose language (imperative Java source code) on the readability of typing rules in programming language design.

The work was motivated by the fact that in programming language design two different languages are typically applied: one DSL for giving the formal definition of a language (inference rules) and one GPL for implementing the formal definition. From our personal, subjective background we are often confronted with the statement (typically by students) that inference rules are unnecessarily complex (and could be or even should be completely replaced by code written in a general-purpose language).

[10]We are aware of the existence of code metrics such as McCabe's cyclomatic complexity (see (McCabe, 1976)), but we also aware that not much evidence exists that such metrics correlate with the readability of source code (see for example (Jay et al., 2009) among many others).

12 students participated in an experiment where a term was given in addition to a typing rule (described either by an inference rule or by Java source code). The participants' task was to choose among a list of eight possible answers the correct one. It turned out that on average the response times for Java source code took 91% more time ($\frac{M_{Java}}{M_{inference}} = 1.914$): the experiment result was even the opposite of what we initially expected. However, it also turned out that this mean ratio is influenced by several additional parameters and the most noteworthy ones are the terms to be read and the participant. Concerning the terms to be read, the experiment gave strong evidence that the more terms are required to read, the larger is the positive effect of the inference notation: with 5 terms to read, the Java notation already required 143.9% more time than the use of inference rules. Concerning the participant, it turned out that while there are participants who require (significantly) 113% more time to read for the Java code, there are others who (non-significantly) require only 28% more to read (for the Java code). Additionally, it turned out that the use of inference rules reduced the number of errors in responses.

In addition to the concrete DSL (and the concrete GPL), the present work also contributes to the question, to what extent declarative or imperative languages are more readable – with the result that the declarative language was more readable for the participants.

Based on the present study, we feel confident that inference rules should be used in teaching and research, but we are aware that the present study can only be considered as a starting point of such studies and should not be considered as a definitive answer. And we are aware that typing rules can be more complex than those ones used in the present experiment.

One characteristics of the present study is, that it is designed as a repeated N-of-1 experiment where each participant can be analyzed in separation: without such design it would be impossible to determine the effect of individuals on the dependent variables. Actually, it turned out that only two of the twelve participants were not reactive to the experiment, i.e., for two participants no significant effect of the notation was detected. We think that such experimental design could help in general to give single participants the chance to rerun a study (which is also a very cheap way to replicate a study). Due to the generative nature of the experiment, it is even possible to select more data on one single participants without the need to rediscuss possible new tasks.

In general, we think that computer science too quickly applies new languages – which is not only a question of the DSLs, but a question of languages in general. The here introduced experiment could help other researchers to do such studies in order to find out whether a given language (that might or might not be a DSL) should replace a different language. And we think that the approach of generating tasks (and answers) is not only helpful for researchers to find answers to their research questions, but also for people who doubt in the results of a given study and who want to rerun a study on their own (with themselves possibly as the only participant).

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., USA.

Bossavit, L. (2015). *The Leprechauns of Software Engineering*. Laurent Bossavit.

Bruce, K. B. (2002). *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA.

Buse, R. P., Sadowski, C., and Weimer, W. (2011). Benefits and barriers of user evaluation in software engineering research. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 643–656, New York, NY, USA. Association for Computing Machinery.

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., Sharif, B., and Tamm, S. (2015). Eye movements in code reading: relaxing the linear order. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 255–265. IEEE Press.

Davulcu, S., Hanenberg, S., Werger, O., and Gruhn, V. (2023). An empirical study on the possible positive effect of imperative constructs in declarative languages: The case with SQL. In Fill, H., Mayo, F. J. D., van Sinderen, M., and Maciaszek, L. A., editors, *Proceedings of the 18th International Conference on Software Technologies, ICSOFT 2023, Rome, Italy, July 10-12, 2023*, pages 428–437. SCITEPRESS.

Dubois, C. (2000). Proving ml type soundness within coq. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '00, pages 126–144, Berlin, Heidelberg. Springer-Verlag.

Gentzen, G. (1935). Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210.

Grewe, S., Erdweg, S., Wittmann, P., and Mezini, M. (2015). Type systems for the masses: deriving soundness proofs and efficient checkers. In *2015 ACM International Symposium on New Ideas, New*

Paradigms, and Reflections on Programming and Software (Onward!), Onward! 2015, pages 137–150, New York, NY, USA. Association for Computing Machinery.

Hanenberg, S. and Mehlhorn, N. (2021). Two n-of-1 self-trials on readability differences between anonymous inner classes (aics) and lambda expressions (les) on java code snippets. *Empirical Software Engineering*, 27(2):33.

Hoffmann, B., Urquhart, N., Chalmers, K., and Guckert, M. (2022). An empirical evaluation of a novel domain-specific language – modelling vehicle routing problems with athos. *Empirical Softw. Engg.*, 27(7).

Hollmann, N. and Hanenberg, S. (2017). An empirical study on the readability of regular expressions: Textual versus graphical. In *IEEE Working Conference on Software Visualization, VISSOFT 2017, Shanghai, China, September 18-19, 2017*, pages 74–84. IEEE.

Jay, G., Hale, J. E., Smith, R. K., Hale, D. P., Kraft, N. A., and Ward, C. (2009). Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2(3):137–143.

Johanson, A. N. and Hasselbring, W. (2017). Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Softw. Engg.*, 22(4):2206–2236.

Kaijanaho, A.-J. (2015). *Evidence-based programming language design: a philosophical and methodological exploration*. University of Jyväskylä, Finnland.

Ko, A. J., Latoza, T. D., and Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Softw. Engg.*, 20(1):110–141.

Kosar, T., Mernik, M., and Carver, J. C. (2012). Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Softw. Engg.*, 17(3):276–304.

Kosar, T., Oliveira, N., Mernik, M., JoÃ£o, M., Pereira, M., RepinÅ¡ek, M., Cruz, D., and Rangel Henriques, P. (2010). Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 438.

Lucas, W., Bonifácio, R., Canedo, E. D., Marcílio, D., and Lima, F. (2019). Does the introduction of lambda expressions improve the comprehension of java programs? In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, pages 187–196, New York, NY, USA. Association for Computing Machinery.

McCabe, T. J. (1976). A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 407, Washington, DC, USA. IEEE Computer Society Press.

McConnell, S. (2011). What does 10x mean? measuring variations in programmer productivity. In Oram, A. and Wilson, G., editors, *Making Software - What Really Works, and Why We Believe It*, Theory in practice, pages 567–574. O'Reilly.

Mehlhorn, N. and Hanenberg, S. (2022). Imperative versus declarative collection processing: an rct on the understandability of traditional loops versus the stream api in java. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1157–1168, New York, NY, USA. Association for Computing Machinery.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., and Reijers, H. A. (2011). Imperative versus declarative process modeling languages: An empirical investigation. In Daniel, F., Barkaoui, K., and Dustdar, S., editors, *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I*, volume 99 of *Lecture Notes in Business Information Processing*, pages 383–394. Springer.

Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press, 1st edition.

Shneiderman, B., Mayer, R., McKay, D., and Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Commun. ACM*, 20(6):373–381.

Siegmund, J., Kästner, C., Liebig, J., Apel, S., and Hanenberg, S. (2014). Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334.

Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. (2016). An empirical study on the impact of c++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 760–771.

Vegas, S., Apa, C., and Juristo, N. (2016). Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135.

Wright, A. and Felleisen, M. (1994). A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94.