Readability of Domain-Specific Languages in Language Design: A Controlled Experiment on Declarative Inference Rules and Imperative Java Source Code

Stefan Hanenberg[1], Kai Klanten[2],
Stefan Gries[3], Volker Gruhn[1]

University of Duisburg-Essen, 45127 Essen, Germany
[2] Independent, Germany
[3] codecentric AG, 42697 Solingen, Germany

Abstract. Domain-specific languages (DSLs) play a large role in computer science. Languages from formal grammars up to SQL are essential part of education as well as industrial applications. In programming language design, inference rules that describe the semantics or the type system of programming languages are DSLs that play a larger role in education and academia as well. However, to what extent such languages have a positive impact on developers in comparison to general-purpose languages (GPLs) mostly remains unclear. A previous study executed on twelve participants detected a strong and large positive effect of such inference rules on the readability of typing rules in comparison to Java source code (p $< .001$, $\eta_p^2 = .439$, $\frac{M_{Java}}{M_{inference}} = 1.914$) in terms of response times for given typing rules, a given term, and a number of response alternatives. The present work introduces a follow-up experiment that compares inference rules with Java source code using a slightly different design: instead of varying the number terms in the inference rules, the present study keeps that constant (three terms) and varies the number of terms that need to be read in order to detect the type of a given expression. The experiment, which was (again) designed as a repeated N-of-1 trial and executed on five participants, revealed (again) a strong positive effect of inference rules on the response times (p $< .001$), still with a large effect but the effect was much smaller than in the original study ($\eta_p^2 = .140$, $\frac{M_{Java}}{M_{inference}} = 1.36$). However, the effect was only detected on three of the five participants. Our interpretation is, that (although again a positive effect of inference rules was shown) the effect might be smaller than originally expected. Hence, we conclude that we found (again) evidence for the positive effect of the inference notation, but we are aware that this effect was measured on only three of five participants.

Keywords: Domain-specific Languages · Readability · Programming Language Design · Type Systems

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. [21]). 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). The goal of such languages is clear: their intention is to make it easier for developers to define computations in a given domain. I.e., the common assumption is, that DSLs are easier to use than general-purpose languages (GPLs), i.e., traditional (turing complete) programming languages.

In programming language design, it is quite common to use inference rules to define the (dynamic) language semantics as well as the type system. This notation has its foundation in the work by Gentzen from 1935 (see [6]). Gentzen defined the so-called natural deduction in a way where premise and conclusion were graphically separated by a horizontal line. This notation was slightly adapted for the use of programming language design. The resulting notation by Wright and Felleisen was not only used to describe the semantics and type system, but also to perform type proofs [29].

However, 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*" [25], that is widespread in teaching, uses inference rules to describe programming languages and then describes how these rules can be 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 computation rules, 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 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?*" We think that this question plays not only a role for teaching. One could ask whether the massive use of inference rules at programming language conferences (such as PLDI, ICFP, POPL, etc.) helps researchers and readers of papers or whether the use of that notation is counterproductive.

[1] There are other reasons for such transformation: one wants to proof language characteristics with the aid of proof-assistant systems (see for example [5,7]), i.e., in such situations the rather graphical inference notation is transformed into a different language. Such language is then typically the language of a theorem prover.

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 [14, p. 133]. This low number was confirmed by Buse et al. [2, p. 649], 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, [16,28]). 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 inference notation.

A first experiment towards the possible effect of inference rules was introduced by Klanten et al. (see [15]). In that experiment, a definition of a typing rule, a term using that rule, and possible types for that term were given. Participants had to chose among the list of types the correct one. The experiment varied the number of terms used in the premise of the typing rule and the typing rule was either described via an inference rule or via Java source code. The experiment, that was executed on twelve participants, had two dependent variables: the response times and the correctness. Each participant had to answer 64 of such questions. It turned out that the inference rules had a strong and large effect on the response times (p $< .001$, $\eta_p^2 = .439$, $\frac{M_{Java}}{M_{inference}} = 1.914$) as well as on the correctness ($\chi^2(1, N = 768) = 5.13$, p $= .023$, 15 errors using Java code, 5 errors using inference rules). Furthermore, the experiment revealed a strong and large interaction effect between the terms to be read and the notation where the more terms appeared in a typing rule's premise, the larger was the difference between Java source code and inference rules (p $< .001$, $\eta_p^2 = .382$, $\frac{M_{Java}^{2\,terms}}{M_{inference}^{2\,terms}} = 1.35$, $\frac{M_{Java}^{5\,terms}}{M_{inference}^{5\,terms}} = 2.44$).

However, despite the clear results, one could argue that the experiment had also some weaknesses. First, although the number of premises in the typing rules varied, it was not controlled that in some situations participants can answer a question although they have not yet read all terms. Second, the difficulty of the types was not controlled. And third–which is from our perspective a larger issue–the replication of the experiment is quite expensive: the average time a single participant required to do all tasks was approximately 54 minutes. Hence, to replicate the study on 10 participants, more than 9 participant hours are required.

The present paper introduces a follow-up study of the one described by Klanten et al. that follows the following goals:

– **a more detailed study on the terms to read (1):** in contrast to the study by Klanten et al. the number of terms in the premises are kept constant and only those terms are varied that are required to answer the question,

- **a more detailed study on the terms to read (2):** the complexity of the typing rules are kept constant (the same typing rules appear in all tasks, just in a different order),
- **a reduction of the effort for participants:** the study by Klanten et al. required on average–excluding the training phase–more than 54 minutes experimentation time and was executed on 12 participants. I.e., altogether more than 10 participants hours were spent on the study. The goal of the present study was to reduce the effort per participant as well as the number of participants in the study.

The present study (that was executed on only 5 participants) required on average only 6 minutes per participant to do all tasks. Again, a strong, positive effect of the inference rules in comparison to Java source code was detected. However, the effect was smaller than in the original study ($\eta_p^2 = .140$, $\frac{M_{Java}}{M_{inference}} = 1.36$). Furthermore, neither an effect on the number of errors, nor an interaction effect between the terms to read and the notation was determined. Hence, we conclude that we were able to replicate the positive effect of the inference notation, but we were not able to replicate all facets of the study.

However, we measured the benefit of the inference notation only on three of five participants. From that we conclude that it is possible to give evidence for the positive effect of the inference notation with much reduced effort. But we also conclude that it was not possible to detect an interaction effect between the notation and another variable (terms to read). Hence, we cannot confirm that the originally measured interaction effect was in fact caused by the terms that need to be read. A possible interpretation is, that simply the length of the inference rules caused that interaction effect in the original study.

2 Background: Notations for Programming Language Semantics

Although the general focus of the present study is on domain-specific languages, the concrete application of the present study is on the notation of programming language semantics (reduction rules and type system). Hence, we focus here only on two notations: inference rules and the application of a given programming language (in our specific case Java).

2.1 Inference Rules in Programming Language Design

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, which go back to the notation of Wright and Felleisen [29].

Figure 1 contains the application such inference rules. The rules are taken from the teaching book by Pierce [25] but the same rules – especially those ones for type checking – can be found elsewhere as well (see, e.g., [1, p. 126]). The rules describe parts of the semantics of the statically typed lambda calculus (which

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

...

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

...

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

Fig. 1. Example evaluation and typing rules (taken from [25, p. 103] for the statically typed lambda calculus. The evaluation and typing rules are described in a declarative style based on natural deduction, i.e., they use inference rules. The figure (with the identical rules) was taken from [15, p. 494].

is a simple programming language that is typically used to teach programming language design).

The first two rules refer to the language construct *application* which means that one term t_2 is applied to a term t_1. The first rule refers to the execution semantics of *application*, the second rule to its role in type checking. With respect to the execution semantics, the first rule (named E-App1) defines 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 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 expressions of the `then` and `else` branch need to have the same type T (which finally is the type of the `if` expression). What makes this rule different from the previous ones is that the type of the condition is a type constant, i.e., a type that is already predefined in the language.

One characteristic of inference rules is, that they combine the syntax of a language with the rules. For example, the type rule T-if does not only state that the first child term c has the type Bool. Instead, one can directly see from the syntax that the variable c represents the condition of the if expression.

Another characteristics of inference rules is the unification of variables with the same names. For example, in the typing rule T-App the type T_{11} appears in the premise for the term t_1 as well as in the premise for the term t_2. I.e., all types that fulfill the one and the other premise are valid types. In the rule T-if the situation is comparable: the type T appears two times in the premise, but this time also in the conclusion. I.e., all types that fulfill both premises are

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

also valid types of a given if-construct. Actually, the application of unification in type checking rules becomes more interesting, as soon as the underlying language supports subtyping (because in that situation more than one type possibly matches a premise or a conclusion).

2.2 Source Code as Alternative Notation

Although the use of inference rules is standard in language design, one often forgets that such rules can be described via any programming language. For example, if one wants to write down the previously described rules in Java, the resulting code could look like shown in Figure 2.

One has to mention that for given typing rules and semantics there are many different ways to implement them. The approach followed in Figure 2 is from our perspective the rather classical object-oriented design where each language construct is described in a single class. The different elements of that language constructs are represented by instance variables (such as three terms in the class representing if expressions–one for the condition, one for the then, and one for the else branch), different methods implement the reduction semantics (`reduce()`), the type checking (`type_of(Environment e)`), respectively are some helper functions (`is_reducible()`). Even in that concrete scenario, there are multiple different implementations possible, such as, whether a reduction should change an object's state, or whether it should return a different object, etc. Furthermore, there are different ways to report type errors. In the given code, runtime exceptions are thrown (either explicitly or implicitly–caused by a failing cast operation).

An essential part of the code is the use of the method `equals(Object o)` in, respectively the return type `Type` of the method `type_of(Environment e)`: as long as the type of an expression is unique, one can simply compare it with a different type.[3] I.e., in contrast to inference rules, there is no feature that implicitly unifies variables. In case such behavior (respectively a comparable behavior such as a simple check for equality) is required, it must be explicitly written down. Additionally, the code in Figure 2 has (in contrast to inference rules) no representation of the language's syntax. I.e., the code is the (abstract) data model in combination with some functionality–but without any hint, how the syntactical representation might look like.

2.3 Research Question

The obvious question is, whether the application of inference rules, i.e., the domain-specific language, has measurable, positive effects in comparison to the application of Java source code. While such rather general question is not unusual in the context of DSLs, it is not obvious what exactly this question means. A notation might:

[3] We already mentioned in Section 2.1 that in presence of subtyping this might change – however, the present study does not consider subtyping in the given language.

```
abstract class LambdaTerm {
  abstract Type type_of(Environment e);
  abstract boolean is_reducible();
  abstract LambdaTerm reduce();
}

class Application extends LambdaTerm {
  LambdaTerm t1, t2;
  ...
  LambdaTerm reduce() {
   // E–App1
    if (t1.is_reducible()) {
      t1 = t1.reduce();
    } else {
      ...
    }
    return this;
  }

  Type type_of(Environment e) {
    // T–App
    Function_Type ft = (Function_Type) t1.type_of(e);
    Type left_type = t2.type_of(e);

    if(ft.left.equals(t2))
      return t2.right;
    else
      throw ...;
  }
}

class If {
  LambdaTerm condition, t1, t2;
  ...
  LambdaTerm reduce() {
    ...
  }

  Type type_of(Environment e) {
    // T–if
    Boolean_Type ft = (Boolean_Type) t1.type_of(e);
    Type t1_type = t1.type_of(e);
    Type t2_type = t1.type_of(e);

    if(t1_type.equals(t2_type))
      return t1_type;
    else
      throw ...;
  }
}
```

Fig. 2. Excerpt from a possible object-oriented implementation of the given inference rules.

– have time effects on the application of a notation (when writing code),
– time effects when reading code,
– effects on the number of errors while writing code,
– effects on the number of errors found while reading incorrect code, or
– psychological effects on developers (motivation, anxiety, etc.)

I.e., notations could possibly have multiple, different effects and it is also possible that a notation has a positive effect (such as a positive effect on readability), and a negative effect (such as a negative effect on the number of errors when using the notation) at the same time. Furthermore, one has to take into account that there are possibly different measurement techniques for the same effect.

In the present study we focus (likewise its predecessor) on the readability of terms written in a notation. I.e., we ask, whether the notation has an effect on the readability of typing rules written either using the inference notation or using an ordinary programming language. As a measurement, we used reaction time, i.e., the time participants required to give an answer (independent of whether this answer is correct). This measurement had been already applied multiple times in the past (see, for example, [15,10,9] among many others).

3 Related Work

The present work can be considered from multiple, different perspectives. The first one is on the effect of a specific domain-specific language on the readability of code – which is the focus of the present work. However, one could also point out one special characterstics of inference rules: such rules are written in a declarative style (in contrast to the rather imperative style of Java source code). Hence, works that compare declarative to imperative code can also be considered as related work. I principle, one could also ask, whether inference rules are a graphical notation and additional works could be mentioned that study the effect of graphical notations (see for example [26,12] just to mention a few). However, we do not focus on that perspective here, because the graphical part of inference rules are just the horizontal line while above and below that line traditional code appears.

As mentioned, the here described experiment is a follow-up study of the one described in [15]. Because of that, we describe the experiment by Klanten et al. in a separate section.

3.1 Controlled Experiments on Domain-Specific Languages

One experiment on domain-specific languages is the one by Kosar et al. who studied in 2010 the possible effect of using the DSL XAML in comparison to C# Forms [18]. 35 programmers answered 11 questions on code fragments that were either defined either via XAML or C# Forms. The code fragments created GUI elements. On average all questions (except one) had a higher success rate when code was represented via XAML: The overall mean success rate using XAML was 37.62% higher than for C# Forms.

The same approach was followed in a study from 2012, but this time applied to two more domain-specific languages (i.e., altogether applied to three different domains, each with its own domain-specific language) [17]: 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 all DSLs the average success rate of participants was higher than for the GPL counterparts.

In 2017, Johanson and Hasselbring studied a DSL in the domain of marine ecosystems–the Sprat Ecosystem DSL–in comparison to C++ code [13]. 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 [11]. 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.[4] 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 which is a traditional declarative language construct that was integrated in imperative languages over the last decades.

Uesbeck et al. [27] 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. [20] 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: participants required less time to identify the result if the (declarative) Stream API was used.

[4] The study distinguishes 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.

In a comparable study, the readability of lambda expressions was studied in comparison to anonymous inner classes [8]. We are aware that it is unclear whether such comparison actually compares a declarative construct (lambda expressions) with a imperative construct (anonymous inner classes), because it is disputable whether an anonymous inner class is actually an imperative language construct. Nevertheless, the study found a (small) positive effect lambdas.

Another study that also had lambda expressions in its focus was performed by Lucas et al. [19] 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. [24]. The authors studied the difference between declarative and imperative business model languages. Students were given two types of tasks and four questions per task[5] 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 [4]: 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.

3.3 Inference experiment by Klanten et al.

The experiment by Klanten et al. already focused on the application of inference rules in comparison to Java source code [15]. In the experiment, 64 typing rules were generated which were either shown as inference rules or as Java code. In order to reduce the possible effect of the concrete syntax in typing rules, the generated rules did not introduce a syntactic element, but generated a rule that only consisted of two to five terms, each associated with a premise. This number of terms was called in the experiment "terms to read". One additional variable in the experiment was the validity of the terms.

In the experiment, a concrete term was given that only consisted of literals. The typing rules for the literals were given using inference rules. The participant's task was to determine the type for the given term. Thereto, six possible answers were generated, each describing a type. One additional answer was "none" in case the participant assumed that none of the given answers described the type. Another additional answer was "error" in case the given type had no

[5] 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.

valid type. The dependent variables were the reaction time and the correctness of the given answer. Each participant had to answer 64 questions.

The experiment was executed on twelve participants. It turned out that the inference notation had a strong and large effect on the response times (p $< .001$, $\eta_p^2 = .439$, $\frac{M_{Java}}{M_{inference}} = 1.914$) as well as on the correctness ($\chi^2(1, N = 768) = 5.13$, p $= .023$, 15 errors using Java code, 5 errors using inference rules). Furthermore, the experiment revealed a strong and large interaction effect between the terms to be read and the notation where the more terms appeared in a typing rule's premise, the larger was the difference between Java source code and inference rules (p $< .001$, $\eta_p^2 = .382$, $\frac{M_{Java}^{2\,terms}}{M_{inference}^{2\,terms}} = 1.35$, $\frac{M_{Java}^{5\,terms}}{M_{inference}^{5\,terms}} = 2.44$).

Hence, following the conclusion of that experiment, it looks like the more terms need to be read, the larger is the difference between the inference notation and Java source code, where there is already a difference in short typing rules (with only 2 terms).

4 Critical Reflections on the Experiment by Klanten et al.

Although we appreciate the original study by Klanten et al., we still think that there is room for improvement. First, we think that the variable "terms to read" in the originally study could be interpreted in a different way. Second, we think that the study did not take into account that typing rules could themselves differ with respect to their difficulty. And finally, the effort for running the experiment was quite high: a single participant required on average approximately 54 minutes to do all tasks. Consequently, running such a study on multiple participants (in order to execute a replication of the study) requires multiple hours of participant time.

The original study by Klanten et al. varied the number of terms in the premises of a typing rule and measured a large effect of that on the reaction time of participants. However, we think that the study did not take into account that the number of terms to read depends on possible typing errors.

Figure 3 illustrates an example taken from the study by Klanten et al.[6] In the figure, a term is given that uses the literals Chair, Cup, and False. If one detects the type of the first literal (Bool $\rightarrow$ Num $\rightarrow$ Bool) and the type of the second literal (Num $\rightarrow$ Bool), and checks the premises of the typing rule, one can already determine that the given expression is not well-typed (because it would require from the second term to have the type Bool $\rightarrow$ Num). Hence, it is not necessary to detect for all literals in the given term their types: the last one can be skipped. One neither needs to find in the typing rules what the type of

[6] We slightly adapted the example to ease the reading of the present paper. For example, the original study by Klanten et al. did not use a concrete syntax in the typing rule, i.e., we added the keyword feature to the typing rule. Additionally, the here presented example is not a screenshot of the original application. Instead, the typing rules and the given term are written to match the style of the present paper.

Chair : Bool $\rightarrow$ Num $\rightarrow$ Bool

Cup : Num $\rightarrow$ Bool

False : Bool

$$\frac{\Gamma \vdash t_1{:}T_1 \rightarrow T_2 \rightarrow T_1 \quad \Gamma \vdash t_2{:}T_1 \rightarrow T_2 \quad \Gamma \vdash t_3{:}T_1}{\Gamma \vdash \mathit{feature}\ t_1\ t_2\ t_3{:}\,T_1 \rightarrow T_2}$$

Given term: feature(Chair Cup False)

Fig. 3. Example task taken from the original experiment by Klanten et al. (with small variations). Participants just need to read the first and the second premise in order to determine that the given term is not well-typed.

the literal False is, nor is it necessary to detect what the premise is for the third parameter of the new language feature. Klanten et al. considered the example as a task where it is necessary to read three premises (i.e., three terms to read). However, if we take into account that the error can be detected after the second term, we think one could take into account that it is only necessary to read two terms.

Chair : Bool $\rightarrow$ Num $\rightarrow$ Bool

Cup : Bool $\rightarrow$ Num

False : Bool

$$\frac{\Gamma \vdash t_1{:}T_1 \rightarrow T_2 \rightarrow T_1 \quad \Gamma \vdash t_2{:}T_1 \rightarrow T_2 \quad \Gamma \vdash t_3{:}T_1}{\Gamma \vdash \mathit{feature}\ t_1\ t_2\ t_3{:}\,T_1 \rightarrow T_2}$$

Given term: feature(Chair Cup False)

Fig. 4. Previous example with a small change: all three premises need to be checked in order to answer the question what the type of the given term is.

Figure 4 is almost identical to the content of Figure 3 with one small difference: this time, the type of the second term Cup matches the typing rule. Now, the reader has to detect the type of the literal False and needs to check, whether it matches the given typing rule (in order to determine that the type of the given term is Bool $\rightarrow$ Num). I.e., we think that a reader has to read one additional term and needs to determine that the resulting term is well-typed.

Hence, we believe that the original experiment had a potential problem in the interpretation of the variable terms to read and did not take into account that not all terms are required to be read in order to answer the question.

Additionally, the experiment by Klanten et al. did not take into account that the typing rules themselves might differ in difficulty. For example, if a literal has a type Bool → Num → Bool, but the typing rule requires that literal to have a type Bool → Num → Num, it is probably a different situation than if a literal has the type Bool and a typing rule requires from that literal to have the type Num. However, such kind of (potential) difficulties are not considered in the experiment by Klanten et al.

Hence, we think that a follow-up study should reduce these problems.

5 Experiment Description

We start with the experiment description by providing some initial considerations. After that, the experiment layout is described, followed by the experiment protocol and execution. Then, we analyse the results.

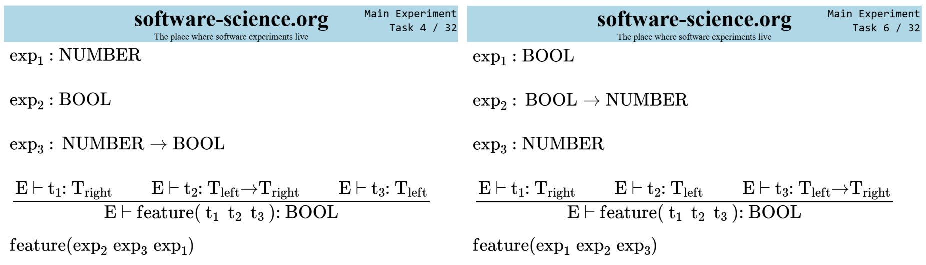
5.1 Initial considerations: Task Design

The experiment was designed with the original study by Klanten et al. in mind. Again, one goal was to reduce the effort for each participant and the effort for running the complete experiment–under the impression that the effect of the inference notation is (probably) large. Another goal was to study the effect of the terms of read in more detail, taking into account that the variable terms to read in the original experiment did not consider those terms that are required to be read in order to determine the type (respectively a potential type error) of a given term.

In order to overcome the potential problem with the variable terms to read (and the potential problem with the difficulty of different types in the typing rule), we decided to use the same patterns for all typing rules: each typing rule consisted of three terms (t_1, t_2, and t_3). One of these terms has the function type $T_{left} \to T_{right}$, the other terms have either the type T_{left} or T_{right}. The given term always has the form feature(exp_1 exp_2 exp_3) (where the order of the literals exp_1, exp_2, and exp_3 was randomly chosen). One literal has the type $NUMBER$, the other one the type $BOOL$. The third literal has either the type $Number \to Bool$ or the type $Bool \to Number$.

Because of this very controlled way of generating types, we assume that the difficulty of the typing rules do not vary any longer. Figure 5 shows a screenshot of the application used for the data collection which gives an impression of the generated typing rules (and terms).

Because of that kind of generated rules, we chose a different question for the task. Participants had to answer the question what position in the given term causes a type error, with the possible answers 1–3 and 0 (in case the term is well-typed). The term on the left hand side in Figure 5 is well-typed, hence, the correct answer to that question is 0. The term on the right hand side is not well-typed: exp_1 has the type $BOOL$ (which is T_{right} in the typing rule), exp_2 has the type $Bool \to Number$ (T_{left} in the typing rule). But exp_3 is expected to have a function type $T_{left} \rightarrow T_{right}$ which is not the case (exp_3 has the type $NUMBER$). Hence, the third expression causes a type error and the correct answer to that task is 3.



exp_1 : NUMBER

exp_2 : BOOL

exp_3 : NUMBER $\rightarrow$ BOOL

$$\frac{E \vdash t_1 : T_{right} \qquad E \vdash t_2 : T_{left} \rightarrow T_{right} \qquad E \vdash t_3 : T_{left}}{E \vdash \text{feature}(\ t_1\ \ t_2\ \ t_3\ ) : \text{BOOL}}$$

feature(exp_2 exp_3 exp_1)

exp_1 : BOOL

exp_2 : BOOL $\rightarrow$ NUMBER

exp_3 : NUMBER

$$\frac{E \vdash t_1 : T_{right} \qquad E \vdash t_2 : T_{left} \qquad E \vdash t_3 : T_{left} \rightarrow T_{right}}{E \vdash \text{feature}(\ t_1\ \ t_2\ \ t_3\ ) : \text{BOOL}}$$

feature(exp_1 exp_2 exp_3)

Fig. 5. Screenshots of the application used for the data collection, one showing the fourth task (valid type, i.e., correct answer is 0), one showing the sixth task (error appears in exp_3, i.e., correct answer is 3). Both tasks use the inference notation.

Hence, if we generated rules and expressions as described, the error position 1–3 describe the terms to be read. In case there is no type error, all three types need to be read (and compared to each other). In our interpretation, the effort for reading this is higher, and we describe no type error as four terms to read.

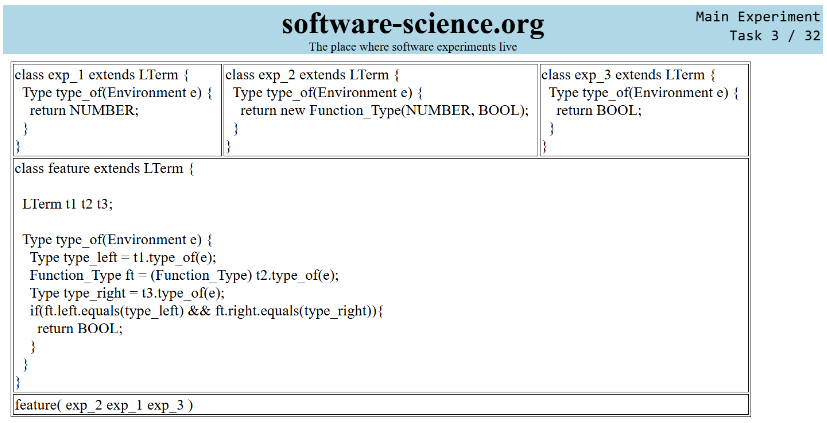
We also generated the Java code is a homogenous way. We used classes that represent each expression (such as a class `exp_1` that represents the literal `exp_1`) and used the identical form for expressing the typing rule for the language feature: the first three lines store the types of all three expressions in local variables with a type cast where the function type is expected. Then, there is an if-statement that checks, whether the left and the right type correspond to the left and the right type of the function type.

We are aware that the code is no valid Java code: the last line (after the if-statement) must throw an exception. We skipped that line in order to ease the readability of the code. Because of existing evidence for the positive effect of indentation (see [22,10,9]), we indented the code. Furthermore, we represented the Java code in a table to reduce the problem that participants need to scroll up and down during the experiment.

5.2 Experiment layout

The experiment consisted of the following variables, respectively the following fixed factors, and used the following post-questionnaire.

- **Dependent variables:**
  - **Reaction time:** The time required by the participant to give an answer (measured in milliseconds).
  - **Correctness:** Whether the given answer was correct (true or false).



```
class exp_1 extends LTerm {
  Type type_of(Environment e) {
    return NUMBER;
  }
}
```

```
class exp_2 extends LTerm {
  Type type_of(Environment e) {
    return new Function_Type(NUMBER, BOOL);
  }
}
```

```
class exp_3 extends LTerm {
  Type type_of(Environment e) {
    return BOOL;
  }
}
```

```
class feature extends LTerm {

  LTerm t1 t2 t3;

  Type type_of(Environment e) {
    Type type_left = t1.type_of(e);
    Function_Type ft = (Function_Type) t2.type_of(e);
    Type type_right = t3.type_of(e);
    if(ft.left.equals(type_left) && ft.right.equals(type_right)){
      return BOOL;
    }
  }
}
```

```
feature( exp_2 exp_1 exp_3 )
```

Fig. 6. Screenshots of the application showing the third task in the experiment (using Java source code). The second expression `exp_1` is required to have a function type but has the type $NUMBER$. Hence, it causes a type error and the correct answer to that question is 2.

- **Independent variables:**
  - **Notation:** The typing rules were either described as inference rules or as Java source code.
  - **Terms to read:** The terms that are required to read in order to determine a typing error (1–4), where 4 is the equivalent to "there as no typing error", i.e., the whole term needed to be read.
- **Fixed factors:**
  - **Number of terms:** A typing rule consisted of three terms in the premise.
  - **Structure of terms in typing rules:** The language construct for which the typing rules were generated had the name *feature* and consisted of three terms. One of the terms had a function type of the form $T_{left} \rightarrow T_{right}$, the other two rules refer to the input or output type of the function type (i.e., they have the type T_{left} or T_{right}). The type of feature was always BOOL.
  - **Number of repetitions:** Each treatment combination was repeated four times, i.e. altogether, a participant had to respond 32 times (2x4x4).
  - **Task order:** The tasks were randomly generated for the experiment. Each participant received the tasks in the same order.
  - **Task:** "*What position in the given term causes a typing error?*" (possible answers were 0–3).
- **Post-Questionnaire:** After the experiment, a questionnaire was given to the participants consisting of the following questions
  - **Age:** Possible responses were "younger than 18", "between 18 and (excluding) 25", ..."between 35 and (excluding) 40" and "older than 40".

- **Status:** Possible responses were “undergraduate student”, “graduate student”, “PhD student”, “professional software developer”, “teacher”, and “other”.
- **Study subject:** Possible responses were “I do not study”, “Computer science”, “Computer science related (such as information systems)”, “something else in natural sciences”, and “something else”.
- **Years in industry:** Possible responses were ”none”, “less or equal to one year”, “more than 1 year, but less than 3 years”, “more than three years, but less than 5 years”, “more than five years”.
- **Impression:** Possible responses were ”I do not think that there was a difference between the notations” (= 0), “The inference notation made it slightly easier for me” (= 1), “Java made it slightly easier for me” (= -1), “The inference notation made it much easier for me” (+ 2), and “Java made it much easier for me”(-2).[7]

5.3 Experiment Protocol and Execution

The experiment was executed via an online application available at github.[8] The application introduced into the experiment by introducing the inference notation as well as comparable Java source code. After that, the experiment starts a training phase consisting of 32 randomly generated tasks. Participants could leave the training phase whenever they like – but they could also restart another training phase after finishing the first one. After each response, the correct answer was shown to the participants. Additionally, participants were invited to make a short break (in case they do not feel concentrated enough). After all tasks were done by a participant, a csv file was generated and the participant was asked to send this file to the experimenter.

Five participants were asked to participate in the experiment.[9] All participants were chosen based on purposive sampling [23], where only participants were invited that were familiar with the inference notation. Four professional developers participated in the experiment and one undergraduate student.

5.4 Results

With respect to the number of errors we executed a χ^2- test on all participants (see Table 1). No difference between both notations was detected (p = .633), although participants had slightly more errors using the inference notation.

[7] The numerical responses were not visible to the participants. Instead, we used that notation in the following to easy the presentation of the responses.

[8] The experiment’s source code is available at `https://github.com/shanenbe/Experiments/tree/main/2024_LanguageTypesDSL_Readability`, it can be executed via the link `https://shanenbe.github.io/Experiments/2024_LanguageTypesDSL_Readability/index.html`

[9] The low number of participant was justified by the very strong and very large effect of the notation in the previous experiment by Klanten et al.

Table 1. χ^2- test on the number of errors. Treatments (TRT) are abbreviated to ease the readability of the table (i = inference notation, c = Java code)

Variable	**df**	χ^2	**p**	N	**TRT**	**Errors**
indentation	1	.229	<.633	90	c	9
				90	i	11

The reaction times were analyzed via an ANOVA using Jamovi v2.3.28.[10] In addition to the previously mentioned independent variables, the independent variable participants was added to the analysis in order to detect possible differences between participants (see Table 2). It turned out that notation was a significant and large factor (p < .001; η_p^2=.014).[11] While terms to read was significant and large as well (p < .001; $\eta_p^2 = .154$), there was no significant interaction between notation and terms to read. The variable participant had a strong and large effect (p < .001; $\eta_p^2 = .623$) and there was a strong and medium interaction effect between notation and participant (p = .006; $\eta_p^2 = .113$).

Due to the strong interaction effect between notation and participant, it is worth to take a closer look at it. Figure 7 illustrates the estimated means for all participants and both notations. It turns out that two participants (no 1 and no 2) have a clear positive effect of the inference notations. However, for participant no 3 and participant no 4 there is only a very small positive effect of the inference notation, while for the last participant the inference notation had again a positive effect. Such differences in the effects per participants could be already determined from Table 2 that also contained the ratios $\frac{M_{Java}}{M_{inference}}$ per participant: such ratios varied from 1.05 (participant 3) up to 1.51 (participant 2). Due to this large variations, it is worth to check individually for each participant the results.

Table 3 shows the responses given by participants with respect to their age, their background, their working experience and their impression of the chosen notations. No participant identified the Java source code as more helpful than the inference notation – but only three of them had a measured positive effect of the inference notation. No single participant revealed an interaction effect between the number of terms and the notation. It is noteworthy that on three participants–for two of them the variable notation did not have an effect on the reaction times–the variable terms to read had an effect on the reaction time (but, again, no interaction effects with that variable were detected).

6 Threats to Validity

We see a number of potential treats to validity of the present experiment.

[10] `https://www.jamovi.org/`

[11] We describe here the effect as large following the convention by Cohen (see [3, p. 287 and p. 283]) where $\eta_p^2 > .01379$ is considered as a large effect.

Table 2. ANOVA results on reaction times. Confidence intervals (CI) and means (M) are given in seconds; Treatments (TRT), respectively treatment combinations, are abbreviated to ease the readability of the table (i = inference rules, c = Java code, p_n = participant number n)

Variable	**df**	**F**	**p**	η_p^2	**TRT**	**CI$_{95\%}$**	**M**	**Ratio**
Notation (n)	1	19.50	< .001	.140	c	10.978; 14.913	12.946	1.36
					i	8.234; 10.845	9.539	
Terms to read (t)	3	7.31	< .001	.154	1	6.596; 11.641	9.119	
					2	8.385; 12.561	10.473	
					3	8.603; 14.036	11.319	
					4	11.807; 16.310	14.059	
Participant (p)	4	46.04	< .001	.623	p_1	10.990;16.156	13.573	
					p_2	17.014; 23.660	20.337	
					p_3	6.125; 7.981	7.053	
					p_4	4.997; 7.063	6.030	
					p_5	8.118; 10.319	9.219	
n * t	3	1.22	.305	.030		*omitted*		
n*p	4	3.84	.006	.113	p_1/c	11.754; 20.763	16.259	1.49
					p_1/i	8.550; 13.226	10.888	
					p_2/c	19.879; 28.996	24.438	1.51
					p_2/i	11.891; 20.582	16.237	
					p_3/c	5.751; 8.694	7.222	1.05
					p_3/i	5.578; 8.191	6.884	
					p_4/c	4.679; 7.793	6.236	1.07
					p_4/i	4.285; 7.362	5.823	
					p_5/c	8.951; 12.195	10.573	1.34
					p_5/i	6.551; 9.178	7.864	
t*p	12	1.25	.254	.112		*omitted*		
t*ps*p	38	.999	.484	.245		*omitted*		

Table 3. Results per participant. BG = Background, Exp = reported number of working experience, Impr = the impression participants had about the usefulness of the different notations. P-values and η_p^2 for the variables Notation (n), Terms to read (t), and the interaction n*t are given in the corresponding columns.

No	**Age**	**BG**	**Exp**	**Impr.**	**Notation (n)**		**Terms to read (t)**		**n*t**		$\frac{M_{Java}}{M_{inference}}$
					p	η_p^2	p	η_p^2	p	η_p^2	
1	35–40	D	1-3	+2 inf	**.041***	.163	.491	.094	.801	.040	1.49
2	30–35	D	> 5	+1 inf	**.006****	.278	.132	.205	.153	.193	1.51
3	25–30	D	3-5	0	.663	.008	**.003****	.438	.847	.032	1.05
4	18-25	U	none	+1 inf	.458	.023	**<.001*****	.765	.370	.120	1.07
5	35–40	D	> 5	+1 inf	**<.001*****	.373	**<.001*****	.552	.477	.097	1.34

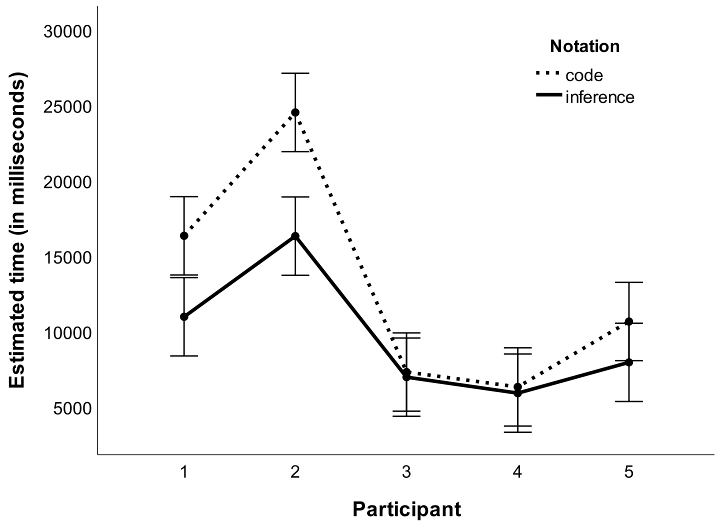


Fig. 7. Estimated means per participant for both notations.

Generated rules/code (1): The rules, respectively the code were generated in a quite homogenous way. The literals were identical (exp1, exp2, and exp3), the type constants were identical (Bool, Num) and only one single function type appeared per task. We think it is quite plausible that participants do not read the complete Java code after training, because they assume that the if-statements are correct (who simple check the left and right type of the function type). In case that multiple terms would had been used in the experiment, this would (probably) imply that participants need to read a complete if-statement. Possibly, differences in the notations (inference notation, respectively Java source code) are much influenced by this if-statement.

Generated rules/code (2): The rules were (from our perspective) quite simple, because either a type constant needed to be read or a function type that consisted of two type constants needed to be read. Possibly, larger differences between notations appear as soon as types are more difficult (such as functions that return functions).

Generated rules/code (3): Another facet with respect to the typing rules is, that the chosen type system is relatively simple. Possibly, the existence of recursive types, generic types, etc. also have an influence on the results where it is unclear to us, whether the statement that inference rules are easier to read also holds for more difficult type systems.

Generated rules/code (4): Finally, the present study hardly makes use of unification in the inference notation. Actually, we believe that unification is a language construct that plays a larger role in the inference notation and probably requires to be studied in more detail. Hence, we are aware that it is unclear whether multiple appearances of unification have an effect on the experiment results. In fact, we are not aware what the effect of unification with respect to readability is.

Training: The training phase was not controlled. We are not aware whether differences in the reaction times between the participants were (potentially) caused by a different intensity of training (which was freely chosen by each participant). Actually, we used such rather free kind of training in previous experiments (see, for example, [9]), but are not aware what the actual impact on experiment results is.

Reaction time as dependent variable: Based on experiences with a previous study we have the feeling that reaction times–and maybe times in general–could be a problematic measure: in a previous experiment (see [10]) we found a larger number of outliers in the data set. Actually, we think that there are multiple different problems in this measurements, but the most important one is, that experimenters assume that measured times were used by participants to solve a task. However, we need to take into account that a very quick reaction time could mean that participants became just bored in an experiment (instead of being able to quickly answer a question). Actually, reaction times (like most measurements known to us) can be easily influenced by participants. While other disciplines have experimental procedures such as blinding or double-blinding as standard procedures since decades, it is more or less unclear, how such approaches could be applied to software experiments.

Sample size: The sample size of the present experiment was relatively small: just 5 people participated in the experiment, each contributing 32 tasks. Hence, we cannot exclude that the non-significance of some variables were caused by this small sample size. However, one still needs to take into account that the interaction effect (notation*terms to read) showed not even a tendency towards significant result (p = .305). Hence, we do not think that a larger sample size would changed that.

Sample: Finally–and we think this is quite common for software experiments–, it is unclear whether the used sample is "valid" or "representative". In the present study we had four professional developers and one undergraduate student. Actually, it turned out that the results of the student were comparable to the results of one developer. However, we are aware that this could also mean that the comparable developer could be considered as an outlier.

7 Summary and Discussion

The present paper introduced a follow-up experiment to the one described by Klanten et al. (see [15]), where a strong and large positive effect of the inference notation in comparison to Java source code was detected (on the dependent variables reaction time and number of errors). The original experiment by Klanten et al. also used a variable terms to read as an independent factor where the number of premises in a type rule were varied. Additionally, the experiment by Klanten et al. had a independent variable that described whether or not the given term is valid. Both of these variables were significant as well and revealed an interaction effect with the variable notation. In the experiment by Klanten et al., each participant required approximately 54 minutes to finish all tasks.

The goal of the present experiment was to study the same main variable (notation with the treatments inference notation and Java source code), but to study the terms to read in a more controlled way (with a constant number of terms in the typing rules). Thereto, the question to participants was adapted (in the present experiment participants were required to determine what term for a given expression causes a type error). Altogether, 32 tasks needed to be done by participants. On average, a participant required 6 minutes for all tasks. Again, the dependent variables were reaction time and number of errors.

Five participants took part in the experiment (four professional developers, one undergraduate student), all familiar with the inference notation. An analysis on all participant showed (again) a positive effect of the inference notation on the reaction time (p $< .001$, $\eta_p^2 = .140$, $\frac{M_{Java}}{M_{inference}} = 1.36$). However, neither an effect of the variable terms to read was detected, nor an interaction effect with notation. A separate analysis on all five participants revealed only for three of them a positive effect of the variable notation (while all participants had a ratio $\frac{M_{Java}}{M_{inference}} > 1$). With respect to the number of errors, no effect was measured.

In summary, the experiment was able to replicate the originally measured positive effect of the inference notation in comparison to Java source code. This could be done with a much reduced effort: the experiment required 5x6 minutes (=30 minutes) participant time for all participants in comparison to the original experiment that required 12x55.4 minutes (approximately 11 hours participant time). However, not all parts of the experiment could be replicated: neither the interaction effect between the variable terms to read, nor the effect on the variable number of errors. We see different possible interpretations of this phenomenon.

First, it is possible that an effect was not shown because of the much reduced number of data points. In the original study 12x64=768 data points were used for the analysis, the results of the present study were derived from 5x32=160 data points. However, taking into account that the effect of the original study were very strong and large (p $< .001$, $\eta_p^2 = .439$, $\frac{M_{Java}}{M_{inference}} = 1.914$), we do not think that this is the (main) reason for this difference.

We think that the different kind of tasks influenced the results. While the original study varied the number of premises of the typing rules, this was not the case for the present study. Instead, the present study tried to focus on those terms that really need to be read in order to answer the given question. However, this had not the expected effect. It is possible that the very homogenous way of generating typing rules (respectively source code) compensated the possible interaction effect. But it is also possible that the explanation is simpler: Possibly an increasing length of a typing rule substantially increases a rule's difficulty. Because maybe the main problem occurs in the readability of the Java code that becomes more complex as soon as multiple terms are required by a typing rule.

I.e., we think it is possible that the main problem occurs in Java's if-statement in the typing rule instead of the other parts of the code.

8 Conclusion

The present paper introduces a follow-up experiment to the one by Klanten et al. In the experiment, the possible effect of the inference notation–a DSL that is typically applied in the domain of programming language design–in comparison to Java source code is determined. Again, a strong and large positive effect of the inference notation was measured (p $< .001$, $\eta_p^2 = .140$, $\frac{M_{Java}}{M_{inference}} = 1.36$). However, the present experiment did not reveal interaction effects that were comparable to the one identified by Klanten et al. But it should be emphasized that the current experiment was able to detect the effect of the inference notation with low costs: the experiment required from a participant only 6 minutes of participant time (and was executed on only 5 participants). Hence, we think that the current experiment can be used as a template for providing experiments whose replications are rather cheap.

We think that the present experiment contributes in general to the domain of domain-specific languages by providing an experimental design where all tasks were randomly generated (in a very controlled way). From our perspective, it would be desirable to see comparable studies for different domain-specific languages, especially those ones that are quite common in the field of computer science.

Conflict of Interest

The authors declare that they have no conflict of interest.

Data Availability Statement

A replication package of the experiment, the experiment's raw data and the analysis files are public accessible under the following link: `https://github.com/shanenbe/Experiments/tree/main/2024_LanguageTypesDSL_Readability`. The experiment can be executed via the public accessible link: `https://shanenbe.github.io/Experiments/2024_LanguageTypesDSL_Readability/index.html`.

Acknowledgement

The authors would like to thank the volunteers who participated in the here presented experiment. Without their contribution, experiments such as the present one could not be executed.

References

1. Kim B. Bruce. *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA, 2002.
2. Raymond P.L. Buse, Caitlin Sadowski, and Westley Weimer. 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, 2011. Association for Computing Machinery.
3. J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Taylor & Francis, 2013.
4. Seyfullah Davulcu, Stefan Hanenberg, Ole Werger, and Volker Gruhn. An empirical study on the possible positive effect of imperative constructs in declarative languages: The case with SQL. In Hans-Georg Fill, Francisco José Domínguez Mayo, Marten van Sinderen, and Leszek A. Maciaszek, editors, *Proceedings of the 18th International Conference on Software Technologies, ICSOFT 2023, Rome, Italy, July 10-12, 2023*, pages 428–437. SCITEPRESS, 2023.
5. Catherine Dubois. 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, 2000. Springer-Verlag.
6. Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210, Dec 1935.
7. Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. 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, 2015. Association for Computing Machinery.
8. Stefan Hanenberg and Nils Mehlhorn. 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, Dec 2021.
9. Stefan Hanenberg, Johannes Morzeck, and Volker Gruhn. Indentation and reading time: a randomized control trial on the differences between generated indented and non-indented if-statements. *Empir. Softw. Eng.*, 29(5):134, 2024.
10. Stefan Hanenberg, Johannes Morzeck, Ole Werger, Stefan Gries, and Volker Gruhn. Indentation and reading time: A controlled experiment on the differences between generated indented and non-indented JSON objects. In Hans-Georg Fill, Francisco José Domínguez Mayo, Marten van Sinderen, and Leszek A. Maciaszek, editors, *Software Technologies - 18th International Conference, ICSOFT 2023, Rome, Italy, July 10-12, 2023, Revised Selected Papers*, volume 2104 of *Communications in*, pages 50–75. Springer, 2023.
11. Benjamin Hoffmann, Neil Urquhart, Kevin Chalmers, and Michael Guckert. An empirical evaluation of a novel domain-specific language – modelling vehicle routing problems with athos. *Empirical Softw. Engg.*, 27(7), dec 2022.
12. Niklas Hollmann and Stefan Hanenberg. 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, 2017.
13. Arne N. Johanson and Wilhelm Hasselbring. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Softw. Engg.*, 22(4):2206–2236, aug 2017.

14. Antti-Juhani Kaijanaho. *Evidence-based programming language design: a philosophical and methodological exploration.* University of Jyväskylä, Finnland, 11 2015.
15. Kai Klanten, Stefan Hanenberg, Stefan Gries, and Volker Gruhn. Readability of domain-specific languages: A controlled experiment comparing (declarative) inference rules with (imperative) java source code in programming language design. In Hans-Georg Fill, Francisco José Domínguez Mayo, Marten van Sinderen, and Leszek A. Maciaszek, editors, *Proceedings of the 19th International Conference on Software Technologies, ICSOFT 2024, Dijon, France, July 8-10, 2024*, pages 492–503. SCITEPRESS, 2024.
16. Andrew J. Ko, Thomas D. Latoza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Softw. Engg.*, 20(1):110–141, February 2015.
17. Tomaz Kosar, Marjan Mernik, and Jeffrey C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Softw. Engg.*, 17(3):276–304, jun 2012.
18. Tomaz Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Crepinsek, Daniela Carneiro da Cruz, and Pedro Rangel Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 438, 05 2010.
19. Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcílio, and Fernanda Lima. 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, 2019. Association for Computing Machinery.
20. Nils Mehlhorn and Stefan Hanenberg. 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, 2022. Association for Computing Machinery.
21. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, dec 2005.
22. Johannes Morzeck, Stefan Hanenberg, Ole Werger, and Volker Gruhn. Indentation in source code: A randomized control trial on the readability of control flows in java code with large effects. In Hans-Georg Fill, Francisco José Domínguez Mayo, Marten van Sinderen, and Leszek A. Maciaszek, editors, *Proceedings of the 18th International Conference on Software Technologies, ICSOFT 2023, Rome, Italy, July 10-12, 2023*, pages 117–128. SCITEPRESS, 2023.
23. M.Q. Patton. *Qualitative Research & Evaluation Methods: Integrating Theory and Practice.* SAGE Publications, 2014.
24. Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A. Reijers. Imperative versus declarative process modeling languages: An empirical investigation. In Florian Daniel, Kamel Barkaoui, and Schahram Dustdar, 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, 2011.
25. Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

26. Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Commun. ACM*, 20(6):373–381, jun 1977.
27. Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 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, 2016.
28. S. Vegas, C. Apa, and N. Juristo. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135, 2016.
29. A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, nov 1994.