Assessing User Preferences in Programming Language Design

Roger D. Chamberlain

Roger D. Chamberlain, “Assessing User Preferences in Programming Language Design,” in *Proc. of ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward!), October 2017, pp. 18-29.

DOI: 10.1145/3133850.3133851

Dept. of Computer Science and Engineering
Washington University in St. Louis

Assessing User Preferences in Programming Language Design

Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington University in St. Louis, St. Louis, MO, USA
roger@wustl.edu

Abstract

The design of new programming languages has primarily been guided by the preferences of a few (the authors of the language), rather than systematic study of the various options available. This is in part due to the fact that user studies to effectively test usability or understandability hypotheses are cumbersome and expensive. An interesting question is whether crowdsourcing techniques can be leveraged to improve this situation.

We explore this idea using a specific example. While the streaming data paradigm is a popular one for expressing parallelism within applications, there has been little consensus on the methods used to express streaming topologies. Here, we explore the use of Mechanical Turk to recruit self-described programmers as a community to assess user preferences and code readability for two techniques currently in use for the expression of streaming application topology.

The positive results of this study point to the idea that crowdsourcing techniques can be an effective technique that can inexpensively assist language developers in making good design choices.

CCS Concepts • **General and reference → Empirical studies**; • **Software and its engineering → Parallel programming languages**;

Keywords Crowdsourcing, streaming language design, user studies

ACM Reference Format:
Roger D. Chamberlain. 2017. Assessing User Preferences in Programming Language Design. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3133850.3133851

1 Introduction

Programming language design is challenging. There are many issues to address (e.g., semantics, syntax), and judging whether or not a particular choice is a good one is often quite difficult. While some judgments can be made based on sound, theoretical reasoning (e.g., whether or not a language is LALR(1) parsable [8]), others require empirical evaluation using user studies (e.g., assessing understandability and preference).

It is enabling these latter judgments that is the focus of the present work. Specifically, to what good effect can we put crowdsourcing techniques for the purpose of making sound language design decisions? We explore this question in the context of languages for streaming data computations, seeking to discern knowledge about the understandability and user preferences of two distinct approaches to application topology expression.

With the recent ascendancy of multicore embedded systems, there is an increasing need to design languages that allow the expression of computational parallelism effectively in terms of programmer ease and understandability. Our current general-purpose languages (e.g., Java, C/C++) are not well suited to this task at all. Parallelism is generally expressed at the thread level, with complex thought processes needed to ensure correctness using locks, semaphores, and the like [20]. Alternatively, directives that muddle the line between semantic expression and comments (e.g., OpenMP [7]) are used to bolt concurrency semantics onto the language well after the language was originally designed. In spite of these limitations, these are the languages that are most commonly used for parallel application development. Because of the inherent sequential semantics of the underlying language, there are significant challenges in extracting and exploiting the concurrency that is inherent in the application but hidden by the expression of the application in a poor language choice.

What is needed are new languages that *do* expose the inherent concurrency available in the application; however, new language design is a particularly difficult thing to do well. A language designer must address both the programming paradigm (i.e., what is the mental model of the computation as perceived by the programmer) as well as the syntactic conventions of the language itself (i.e., how, specifically, is the computation expressed).

Currently popular sequential languages have historically been developed by a single individual (e.g., Java by Gosling [12], C by Ritchie [32], C++ by Stroustrup [38]), and the syntactic conventions where primarily guided by the individual opinions of the language developers. Yet, who among us thinks that the precedence rules of operators in C were a wise choice? Wouldn't it have been better for Ritchie to have done a user study on precedence rules before finalizing them?

In the abstract, sure, a user study would have been a good idea. In the real world, the overhead and expense of an effective user study was way beyond the resources available, and it was never considered. Buse et al. [5] report that *recruiting* is the most highly reported barrier to user evaluations, cited by 60% of their survey's respondents. Recently, however, web-based mechanisms that support crowdsourcing have been used for a variety of purposes [4, 18, 23], and the present question is whether these techniques can be leveraged for parallel language design. This paper describes an empirical study that attempts to use crowdsourcing for the assessment of syntactic mechanisms in languages that explicitly support parallelism, specifically the streaming data paradigm.

An example of languages that do a good job of expressing parallelism are those focused on data parallelism. From origins in High Performance Fortran [17] and the like, Parallel Global Address Space (PGAS) languages allow a programmer to state an operation (or set of operations) to take place on an entire data set (typically described in terms of multi-dimensional arrays). This is well suited for applications whose data references are well structured and regular; however, it doesn't work as well for irregular data references or applications that are not readily expressed in array format.

One programming paradigm that holds some promise as a general-purpose, parallel-friendly way to author applications is the streaming data paradigm. Here, applications are expressed as a set of compute kernels that communicate via explicit channels. The streaming paradigm is not new, early languages and systems that support streaming are described by Stephens [37]. More recent streaming systems include the following: Auto-Pipe [9], Brook [3], Cg [27], DirectFlow [25], RaftLib [1], S4 [30], S-Net [13], ScalaPipe [42], Spidle [6], StreamIt [41], Streamware [14], and System S [11]. Examples of application domains that exploit streaming data include media [29], data mining [10], signal processing [33], computational science [26], and others [40].

Figure 1 illustrates an example streaming application topology. Each of the individual kernels

`Input, Distribute, Compute, Merge, Output`

represents a separate computation that receives data from its input(s) and sends data to its output(s). The edges indicate data communication, and each kernel has an isolated memory footprint (i.e., the explicit edges are the only inter-kernel communications allowed, there is no shared memory). The figure shows a common application topology, in which the data are split into two streams for parallel computation in the `Compute` kernels and then recombined in the `Merge` kernel. This example is a straightforward expression of data parallelism.

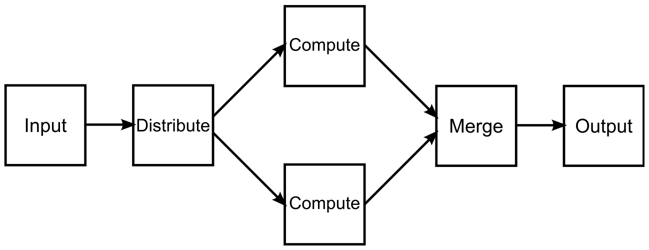


Figure 1. Simple streaming application topology.

One of the attractions of the streaming data paradigm is that it supports multiple forms of parallelism while simplifying the correctness semantics that the programmer must consider. In the above example, there is data parallelism between the two `Compute` kernels. In addition, there is pipelining between upstream kernels and downstream kernels (e.g., while the `Compute` kernels are operating on earlier data elements, the `Distribute` kernel can be operating on later data elements in the stream). With all of the above parallelism explicitly expressed in the application's streaming topology (which makes it straightforward to extract by the compiler), the dominant semantics that the programmer must consider are still sequential. Individual data elements are processed by one kernel at a time, in sequence, diminishing the "parallel thinking" that is the responsibility of the programmer.

While interest in the streaming data paradigm is increasing, there has not been any serious consensus as to the syntactic mechanisms for expressing streaming programs. Each of the example streaming systems listed earlier has its own approach to describing the topological properties illustrated in Figure 1. Here, we explore the use of Amazon's Mechanical Turk as an approach to recruiting study participants for the purpose of judging language features. Specifically, we are interested in assessing programmer preferences across two primary techniques for topological expression, and also investigate the readability of each. Our user community consists of self-described application developers recruited via Mechanical Turk. We successfully recruited two sets of 60 participants (each in under 3 hours), qualified their skills, and showed statistically significant benefit for one approach to topological expression.

2 Background and Related Work

As noted in the introduction, there have been many streaming data systems introduced in the past [1, 3, 6, 9, 11, 13, 14, 25, 41, 42]. The seminal work on streaming semantics is Synchronous Data Flow (SDF) [21], in which kernel executions have known data volume at each input and output port. When kernels have filtering properties, deadlock is possible; however, deadlock avoidance techniques have been described by Li et al. [24]. Hirzel et al. [16] recently cataloged a set of stream processing optimizations that were independently developed across a range of communities, ranging from digital signal processing to databases and operating systems. In addition, object algebras [2] can be used on the stream specification to yield flexible implementation semantics (e.g., fusing of kernels, lazy computation, etc.).

While the theory of streaming computation is quite advanced, the practice is significantly hindered by the lack of commonality in the approaches used to express applications. In some cases, streaming semantics are added as library extensions to existing languages (e.g., RaftLib is a template library for C++, Java 8 has a new Stream abstraction). More commonly, a dedicated language is used, almost always closely resembling a familiar language (e.g., ScalaPipe is based on Scala, StreamIt is very much like Java).

Independent of the above, there remains the question of how is the application topology expressed. That is the question explored in this work. There are at least two schools of thought on this subject:

1. *Functional* specification – here, we borrow the syntactic mechanisms of function calls to express the stream topology. The simple stream of Figure 2 would be expressed as follows:

   ```
   Output(Sum(Square(Input())))
   ```

   where the kernel's inputs are specified in the same way as parameters to a function, and the kernel's outputs are specified in the same way as return values. The notion here is fairly natural, e.g., the data coming out of the `Square` kernel are input to the `Sum` kernel. It is, however, more difficult when describing more complicated topologies than a tandem connection. Another potential confusion is the fact that since the syntactic appearance is the same as that for function calls, there is the potential that the reader of an application doesn't appropriately distinguish between the two. ScalaPipe is an example of a streaming system that uses the functional specification style.
2. *Literal* specification – here, we explicitly state the nodes and edges of the topology graph. In this case, the stream of Figure 2 would be expressed as:

   ```
   Input -> Square -> Sum -> Output
   ```

   where the `->` symbol is an explicit operator literally depicting an edge in the application's topology. A benefit here is the obvious similarity in appearance between the text of the source and the graphical representation in the figure. One disadvantage is that this style is generally not available in library implementations associated with existing languages. It requires customized parsing that is generally not supported for library developers. Auto-Pipe is an example of a streaming system that uses literal specifications.

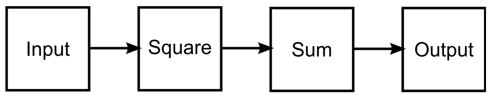


Figure 2. Simple streaming application that computes the sum of squares of an input stream.

The functional and literal styles are by no means the only approaches to specifying application topologies. Lee et al. [22] describe an approach to extracting streaming semantics from applications expressed in a more traditional imperative execution style.

We are interested in the efficacy of Mechanical Turk as a source of participants for user studies in the area of language assessment. Hanenberg [15] argues strongly that software science research should explicitly include human factors, which implies the need to recruit subjects for experimental purposes. Markstrum [28] surveys historical language design papers assessing how often claims are or are not backed by evidence. Stefik et al. [35] followed up with an assessment of the foundation of evidence based on human factors in language design, studying papers published in programming language workshops. Buse et al. [5] performed a comprehensive study of the presence (and absence) of user evaluations in software engineering research, including an assessment of perceived barriers (on the part of the researchers) to performing user evaluations. Recruitment of subjects topped the list of perceived barriers.

One common approach to the recruitment of study participants is the classroom. For example, Stefik and Siebert [36] recruited 288 students from undergraduate courses to evaluate programming language syntax for novices. While this approach works well when the research questions to be addressed are focused on individuals with limited experience, it doesn't translate well to more experienced programmers. Another approach leverages commercial resources. As an illustration, Stylos and Clarke [39] recruited participants for their API study using the Microsoft Usability Research website (http://microsoft.com/usability). They were able to recruit 30 experienced programmers in this way.

Buhermester et al. [4] and Kittur et al. [18] both investigate the use of Mechanical Turk for the purpose of recruiting subjects for user studies. While both conclude that the technique can be successful, there are caveats that must be considered as part of the experimental design. Mechanical Turk has been used to recruit subjects in the context of learning programming by Lee and Ko [23], although we are unaware of its prior use in evaluating programming language designs. Ko et al. [19] list Mechanical Turk as a potential source of participants for empirical user studies, but do not report a study that utilizes it for language design.

3 Specification Styles

In this section, we describe the two specification styles to be investigated. Each is used in both academic streaming systems as well as commercial systems.

We use two application topologies (shown in Figure 3) to illustrate the two specification styles. In both applications, there are 4 stages of computation, labeled 'A', 'B', 'C', and 'D'. Figure 3(a) is the same topology as Figure 2, with the names of the compute kernels altered to conform to the pattern we will follow for the rest of the paper. Figure 3(b) replicates kernel B to illustrate the expression of data parallelism.

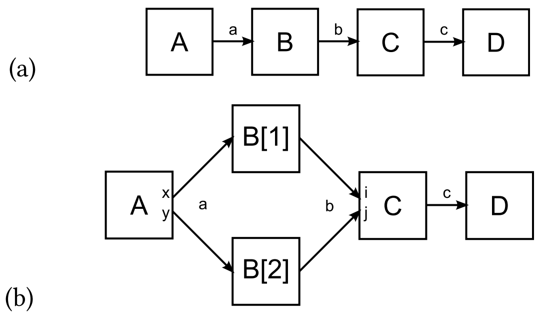


Figure 3. Two application topologies: (a) tandem and (b) data parallel.

3.1 Functional Style

For the tandem topology, the functional style is quite straightforward. The simplest functional description of Figure 3(a) is as follows.

```
D(C(B(A())))
```

Alternatively, one can declare one (or more) of the edges in the graph, and then reference the edge(s) as part of the description. This yields the following description of the same graph.

```
edge b = B(A())
D(C(b))
```

In this case, the edge b is declared as the output of kernel B in the first statement, and then used as the input to kernel C in the second statement. In this paper, we will not concern ourselves with the data types being communicated along the edges. For our purposes, assume the data type of edge b (and all edges) is inferred from the data type of the output port(s) of kernel B.

Changing our focus to the application of Figure 3(b), there is a data-parallel set of kernels B, which can be referenced individually (as B[1] and B[2]) or collectively (as B[*]). Arrays of edges can then be declared and utilized to describe the topology as follows.

```
edge[2] a,b
a = A()
b[1] = B[1](a[1])
b[2] = B[2](a[2])
D(C(b[1],b[2]))
```

There are several things to note about this description: (1) the output of kernel A is an array of edges, named a, of length 2, (2) each element of a is fed into an instance of kernel B, (3) the two inputs to kernel C are explicitly listed (in the syntactic style of parameters); however, that statement could have alternatively been expressed as D(C(b)), referencing the entire array b by its name.

A simpler data parallel topology expression is shown below:

```
D(C(B[*](A())))
```

which exploits the ability to reference the entire collection of kernels B using the notation B[*].

3.2 Literal Style

In the literal style, each edge is explicitly described, so the full description of the tandem topology of Figure 3(a) is

```
A -> B
B -> C
C -> D
```

and the shortened version is as follows.

```
A -> B -> C -> D
```

Considering the data parallel topology of Figure 3(b), the literal style relies less on edge names and instead references the names of kernel input and output ports. For example, kernel A has two output ports, named x and y. They are referenced using the notation A.x and A.y. Similarly, inputs i and j of kernel C are referenced as C.i and C.j.

```
A.x -> B[1] -> C.i
A.y -> B[2] -> C.j
C -> D
```

Utilizing the ability to reference the collection of kernels B, this can be simplified to the following.

```
A -> B[*] -> C -> D
```

3.3 Discussion

Clearly, either specification style is capable of describing application topology. The questions we are interested in are the following: (1) can systems like Mechanical Turk be effectively used to empirically assess the alternatives, (2) is there a preference among developers between the two styles, and (3) is either style more or less prone to understandability errors. We will explore question (1) by devising an experiment that explores questions (2) and (3), attempting to recruit developers on Mechanical Turk as experimental subjects, and assessing all three questions by analyzing the results of the experiment.

4 Experimental Procedure

We are interested in exploring both the understandability of streaming data applications expressed in different specification styles and the preferences of developers for each style. Mechanical Turk is a crowdsourcing system that has been used in the past to recruit participants for user studies [4, 18], and we are interesting in assessing its utility for the purpose of answering questions like these in language design.

Each "job" requires that a participant self-identify whether or not they have any significant programming experience (including amount and language(s)), read a description of both the functional and literal specification styles, correctly answer a set of simple questions about those styles, respond to the questions designed to assess understandability, and finally express their preference between the two styles. Those participants that respond to all the questions (correctly answering the validation question described below) are paid \$5 per job. As a further incentive to expend serious effort, those who answer 80% or more of the understandability questions correctly receive an additional \$5 bonus.[1]

Upon expressing a willingness to serve as a study participant, each individual is first asked some demographic information (age, gender, attained education level) and then asked about their programming experience (how much experience and with what languages).

4.1 Validation Test

It is important in the experimental design to provide incentives for participants to not simply game the system by providing arbitrary answers, but rather to exert effort in an attempt to answer correctly [4, 18]. This can take the form of positive incentives (the bonus described above for a sufficient number of answers being correct) as well as negative incentives. In this experiment, we ask participants to correctly identify both the functional and literal specification of the topology shown in Figure 4, called the *validation test*.

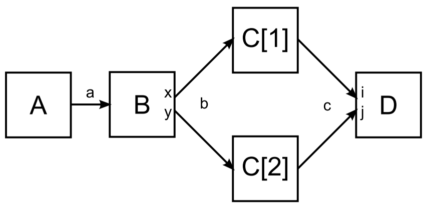


Figure 4. Application topology of validation test.

[1]The study described here was conducted in accordance with Washington University in St. Louis's IRB guidelines.

The four possible responses for the functional specification are:

```
1.    D(C[*](B(A())))
2.    edge[2] c = C[*](B[*](A()))
      D(c)
3.    D(C(B(A())))
4.    D(C[*]())
      edge[2] b = B(A())
```

Option 1 is the correct response, options 2 and 3 describe different topologies, and option 4 is not syntactically correct.

The possible responses for the literal specification are:

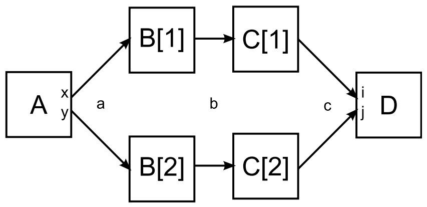
```
1.    A -> B
      B.x -> C[1] -> D.i
      B.y -> C[2] -> D.j
2.    A -> B[*] -> C[*] -> D
3.    A -> B -> C -> D
4.    A -> B.x -> C -> D.i
      A -> B.y -> C -> D.j
```

Again, option 1 is the correct response, options 2 and 3 describe different topologies, and option 4 is not syntactically correct.

The notion of the validation test is that the queries are about a topology that is sufficiently similar to that of Figure 3(b) that anyone with any ability to read and understand code should get it right. If they get it wrong, we are not interested in their responses to the remaining questions.

4.2 Understandability Test

In addition to the validation test, the participants are asked to identify the functional and literal specifications of five more application topologies. Both the topology and the correct specifications for one of questions in this test are shown in Figure 5. The remaining topologies and correct specifications are provided in the Appendix. These five application topologies constitute the *understandability test*. We are trying to assess the degree to which the participant understands the relationship between the actual topology (shown in pictorial form) and the two specifications.



(a) Application topology.

```
D(C[*](B[*](A())))
```

(b) Functional style specification.

```
A -> B[*] -> C[*] -> D
```

(c) Literal style specification.

Figure 5. Understandability test: Topology 1.

For the incorrect answers presented as possibilities to the participant, the pattern used in the validation test is maintained. Two responses describe different (incorrect) topologies, and one response is not well-formed syntactically. The 6 questions that constitute the validation test and the understandability test are asked in random order. In addition, the 4 possible responses are also presented to the participant in random order.

4.3 Preference Query

Once the participant has completed his/her responses to the questions described above, the final question asks about preference. **Please indicate the degree to which you prefer one specification style over the other.**

The possible responses are:

1. **Strongly prefer the functional style**,
2. **Weakly prefer the functional style**,
3. **No preference between the two styles**,
4. **Weakly prefer the literal style**,
5. **Strongly prefer the literal style**.

5 Results

The experiment was conducted in two rounds. We will first discuss results that are exclusively from the first round, followed by a description of why the second round was conducted and results from both rounds.

5.1 First Round Results

The experiment described in Section 4 was conducted on Mechanical Turk, requesting 60 participants, with two deviations from the description above. First, we payed individuals who did not successfully complete the validation test, and second, we did not offer rewards to those who did well. This provides a control to enable us to investigate the impact of incentives in the next round.

We successfully recruited 60 participants (30% female) in less than 3 hours. The mean completion time was 21 min, with a std. dev. of 14.6 min (histogram shown in Appendix). Of the responses, 35 correctly completed the validation test. A histogram of the specification style preferences of these 35 individuals is shown in Figure 6, and there is a very strong preference for the literal style over the functional style.

To assess understandability, we separately tallied the number of correct responses for each specification style. These results are shown in Table 1, which indicates mean and standard deviation for correct responses in the functional style, the literal style, and combined. The results for validated participants is shown in the first data column. The difference between the two styles is statistically significant ($p = 0.01$, two-tailed, paired data), with an improved understandability associated with the literal style.

The importance of the validation test is illustrated by comparing the results of the first data column of Table 1 to those

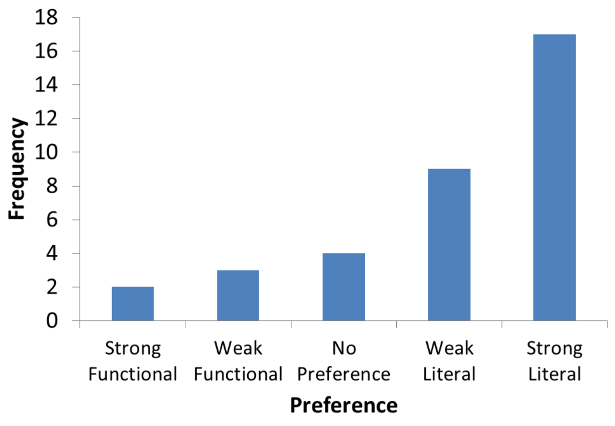


Figure 6. Specification style preferences for first round validated participants.

Table 1. Correct responses (mean ± std. dev.) given by first round participants.

Specification style	Validated participants (% correct)	Non-validated participants (% correct)
Functional	60.6 ± 20.9	42.4 ± 26.7
Literal	71.4 ± 20.7	46.4 ± 22.2
Both	66.0 ± 17.2	44.4 ± 19.4

in the second data column. There, the correct responses are tabulated for those individuals who did not successfully complete the validation test. Not only are the scores lower (as one would expect), but in addition the statistically significant difference between the two specification styles is no longer present ($p = 0.51$, two-tailed, paired data). This is one illustration of the importance of including the validation test, the results depend on it.

Leaving the question of incentives for the second round, it is clear that Mechanical Turk provides a fast, cost-effective mechanism for recruiting test subjects. The total elapsed time for 60 individuals to be recruited and complete the requested tasks was less than 3 hours, and the participant support costs totaled $\$5 \times 60 = \$300$. Both of these are dramatically lower than other forms of recruiting test subjects.

Assuming that individuals took the 30 minutes of estimated/advertised time to complete the task, this translates to an effective wage of \$10/hr, which, while certainly not seriously competitive for an experienced programmer in the developed world, isn't so low as to be insulting, either. In actuality, participants averaged less than this amount of time, resulting in an effectively higher average pay rate.

Turning next to the warnings articulated by both Buhrmester et al. [4] and Kittur et al. [18], there is a tendency for some respondents to game the system by not reading the instructions or the questions at all, but merely clicking quickly through the selections make available. We find this behavior in our respondents as well, with the minimum completion time recorded at 104 seconds (note, this individual did not correctly answer the validation test, so his/her responses were not included in the set of validated results). The suggestion from the literature is to appropriately tailor the incentives for respondents to address this issue, which we investigate in the second round.

5.2 Second Round Results

To help us assess the impact of incentives, a second round experiment was subsequently conducted, again requesting 60 participants. There were two changes to the instructions provided to participants: (1) they were cautioned that they would not be paid if they did not successfully complete the validation test, and (2) they were told they would be provided a bonus payment if they completed with an understandability score of 80% or higher.

Again, we successfully recruited 60 new participants (28% female) in less than 3 hours (there were no individuals who participated in both rounds). The mean completion time was 29 min, with a std. dev. of 15.6 min, statistically significantly longer ($p = 0.003$, two-tailed, non-paired data) than the first round results. We interpret this as evidence that participants worked harder to successfully understand the applications. At the very least, they clearly took more time.

Of the responses, 34 correctly completed the validation test. A histogram of the specification style preferences of these individuals is shown in Figure 7. Both the fraction of participants that correctly completed the validation test and their specification style preferences are qualitatively the same as from the first round (unimpacted by the incentives). Again, there is a very strong preference for the literal specification style over the functional specification style.

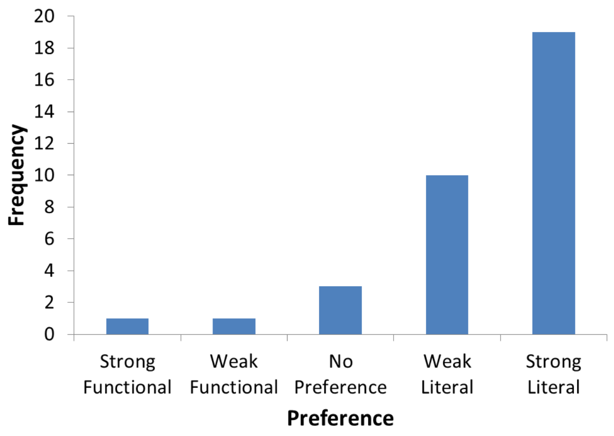


Figure 7. Specification style preferences for validated second round participants.

The understandability results for the second round are shown in Table 2, which indicates mean and standard deviation for correct responses in the functional style, the literal style, and combined. The difference between the two styles for validated participants (first data column) is statistically significant ($p = 0.0001$, two-tailed, paired data), with improved understandability for the literal style.

Table 2. Correct responses (mean ± std. dev.) given by second round participants.

Specification style	Validated participants (% correct)	Non-validated participants (% correct)
Functional	64.7 ± 22.6	49.2 ± 25.4
Literal	81.2 ± 19.7	60.8 ± 23.0
Both	72.9 ± 18.2	55.0 ± 20.8

As in the first round, the understandability scores for those who did not pass the verification test are noticeably lower, with a combined mean score of 55% and std. dev. of 21%. These results are presented in the second data column of Table 2. Interestingly, within this group there is a statistically significant difference between the understandability scores for the literal versus functional style ($p = 0.03$, two-tailed, paired data).

Like the first round, the second round proved to be quite efficient both in terms of recruitment time (60 participants in less than 3 hours), and in the budget that must be allocated (max($10) × 60 = $600). While there were still a few respondents that spent almost no time, the number of such individuals dropped considerably (see completion time histograms in Appendix). For example, the number that spent less than 10 minutes decreased by more than half.

With the incentive pay included, a 30 minute task now pays up to $10, for an effective rate of $20/hr. Again, not truly competitive with first world salaries, but well enough for us to have willing participants.

5.3 Discussion

The primary question we are interested in addressing in this paper is the viability of recruiting suitable participants for assessing language features. In two separate rounds of experimentation, 60 individuals were recruited in less than 3 hours of elapsed time, and over half of those individuals successfully completed a validation test to assess suitability.

It is apparent that the validation test is an important component of the recruitment process, as there are clear differences in responses between the subset of participants who did or did not pass validation. This is consistent with previously reported results [4, 18] in other disciplines. The demographics of the validated participants are reasonable as well: 29% are female, the mean age is 30, and the overall programming experience is shown in the Appendix.

Another consideration, highlighted by Hanenberg [15], is the potential for results that are different for experienced versus non-experienced programmers. To investigate this, we filtered the responses (from both rounds) into two groups: less than 1 year of programming experience, and 1 year or more of programming experience. Table 3 shows the understandability results from these two groups.

Table 3. Correct responses (mean ± std. dev.) given by all validated participants, separated by programming experience.

Specification style	Participants with < 1 year exp. (% correct)	Participants with ≥ 1 year exp. (% correct)
Functional	59.1 ± 19.5	64.0 ± 22.8
Literal	65.2 ± 20.2	82.2 ± 18.7
Both	62.2 ± 17.3	73.1 ± 17.4

For the inexperienced programmers, there does *not* exist a statistically significant difference in understandability scores between the two styles ($p = 0.15$, two-tailed, paired data). For the group with 1 year or more of programming experience, the understandability scores *are* statistically significant ($p = 0.00003$, two-tailed, paired data). This clearly points to the need to differentiate between these two groups.

The languages that the experienced programmers reported familiarity with are similar to the general popularity of languages as reported by Ray et al. [31]. The distribution is shown in the Appendix.

Given a validated set of respondents, the conclusions that can be drawn from both rounds of experimentation are clear. Our participants overwhelmingly prefer the literal style over the functional style. Also, for experienced programmers, the understandability scores for the literal style are statistically significantly greater than for the functional style.

What this doesn't tell us is, actually, quite a number of things. (1) Is the literal style easier or more effective to author than the functional style? It might be quite reasonable to design a crowdsourced study to ask this question. (2) Is it easier to debug either style? This is much harder to assess in a simple study. (3) Does preference really matter? While we asked about preference, and there was a pretty strong result, does this actually impact programmer productivity in any meaningful way? Again, much harder to assess via crowdsourcing methods.

In spite of what we didn't learn, it should be clear that crowdsourcing methods (such as Amazon's Mechanical Turk) do enable user studies at lower cost (both in terms of time and money) than traditional methods. They have a place in programming language design.

6 Conclusions and Future Work

The evaluation of programming languages implies user studies, and a primary challenge associated with any user study is the recruitment of participants. Using Amazon's Mechanical Turk, we were able to recruit 120 study participants in less than 6 hours (60 each in two rounds that were under 3 hours each). Using a validation test that was part of the study itself, we were able to ensure that data were collected from qualified individuals, and over 50% of the recruited individuals successfully completed the validation test.

We explored the use of financial incentives, and those who had increased incentives spent a greater amount of time. While the mean score of those with financial incentives did increase from 66% to 73%, this increase was not statistically significant ($p = 0.11$, two-tailed, non-paired data).

Across the board, for all qualified participants, there was a strong preference for the literal style specification of streaming applications. In addition, for experienced programmers, there was a statistically significant improvement in understandability scores from the functional style specification to the literal style specification.

There are a number of things that need to be considered for future work. First, the study we have completed is far from definitive. Additional rounds with greater numbers of participants are definitely in order. This, at least, is a specific strength of our participant recruitment mechanism, as scaling up study size using Mechanical Turk is straightforward. In addition, we need to include a wider variety of application topologies, especially larger applications that contain greater numbers of kernels.

More generally, Hanenberg makes the argument than experiments of the type described here are difficult to generalize. Quoting from [15] (discussing questions that compare techniques A or B):

> ... it is hard to get any more insights from these studies that could be used in different experimental settings.

The criticism is valid. Ko et al. [19] provide guidance in experimental design with human participants, and the field of Human-Computer-Interaction (HCI) has much to tell us about how to perform empirical studies that effectively test more general theories (see, e.g., [34]). This experiment was but a simple example.

For example, two explicit improvements that make sense for the studies we report here are the following: (1) by having participants be directed to our website rather than using Mechanical Turk's forms interface, we could make more detailed measurements of things like time on task, and (2) while a full, in-person, post-experiment debriefing is difficult in this context, a more comprehensive on-line debriefing might have added to our take-away knowledge. The above two examples are not at all overly onerous.

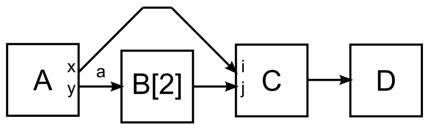
Mechanical Turk potentially allows a more data-driven approach to language design than has been previously affordable to many research groups. Given the ability to recruit and qualify participants with the appropriate background and skill set, we believe that it can be a significant driver in the future for the programming language research community.

A Appendix

This appendix provides information on the understandability test, the programming experience questions, the completion time statistics for participants, and the availability of the raw experimental data.

A.1 Understandability Test

Figures 8 through 11 show the topology and the correct specifications for the four application topologies that (along with Figure 5) constitute the understandability test.



(a) Application topology.

```
edge[2] a = A()
D(C(a[1],B(a[2])))
```

(b) Functional style specification.

```
A.x -> C.i
A.y -> B -> C.j
C -> D
```

(c) Literal style specification.

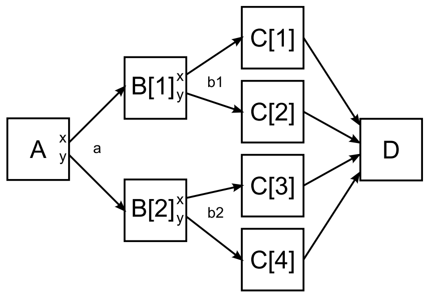
Figure 8. Understandability test: Topology 2.

In all of the topology diagrams, 'A' through 'D' represent compute kernels, the letters 'x' and 'y' represent output ports, 'i' and 'j' represent input ports, and 'a', 'b', and 'c' represent communication edges.

A.2 Programming Experience

The following questions were used to inquire about programming experience on the part of participants.

1. **How much experience to you have programming?** The possible responses are:
   a. **none**
   b. **up to 1 year**
   c. **1 to 5 years**
   d. **more than 5 years**
2. **What language(s) are you reasonably familiar with (i.e, would feel comfortable using if given an application development task)?** The possible responses are:
   a. **C/C++**
   b. **C#**



(a) Application topology.

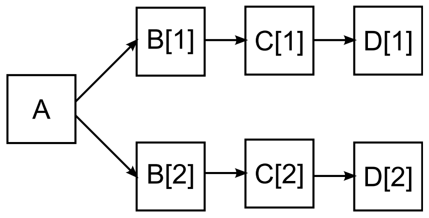
```
edge[2] a, b1, b2
a = A()
b1 = B[1](a[1])
b2 = B[2](a[2])
D(C[*](b1[1],b1[2],b2[1],b2[2]))
```

(b) Functional style specification.

```
A -> B[*]
B[1].x -> C[1]
B[1].y -> C[2]
B[2].x -> C[3]
B[2].y -> C[4]
C[*] -> D
```

(c) Literal style specification.

Figure 9. Understandability test: Topology 3.



(a) Application topology.

```
D[*](C[*](B[*](A())))
```

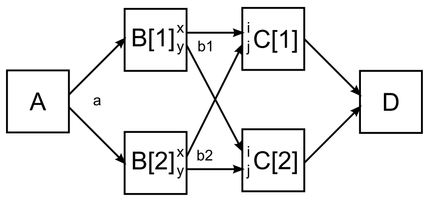
(b) Functional style specification.

```
A -> B[*] -> C[*] -> D[*]
```

(c) Literal style specification.

Figure 10. Understandability test: Topology 4.

   c. **Java**
   d. **JavaScript**
   e. **Objective-C**
   f. **PHP**
   g. **Python**
   h. **Ruby**
   i. **Verilog**
   j. **VHDL**
   k. **Visual Basic**
   l. **Other**



(a) Application topology.

```
edge[2] a = A()
edge[2] b1 = B[1](a[1])
edge[2] b2 = B[2](a[2])
D(C[1](b1[1],b2[1]),C[2](b1[2],b2[2]))
```

(b) Functional style specification.

```
A -> B[*]
B[1].x -> C[1].i
B[1].y -> C[2].i
B[2].x -> C[1].j
B[2].j -> C[2].j
C[*] -> D
```

(c) Literal style specification.

Figure 11. Understandability test: Topology 5.

The programming experience of the validated participants is shown in Figure 12. The languages that the experienced programmers (1 year or more) reported familiarity with are shown, in histogram form, in Figure 13.

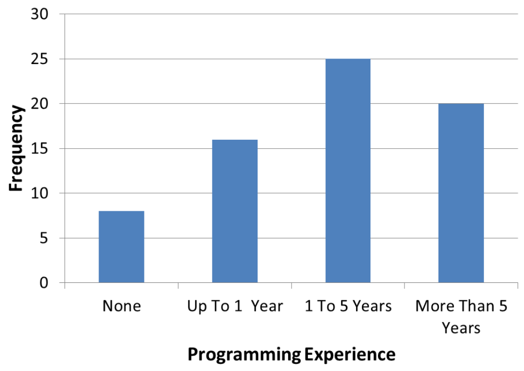


Figure 12. Programming experience of validated participants.

A.3 Completion Times

The histograms of completion times for the two experimental rounds are given in Figures 14 and 15.

The mean completion time for the first round was 21 min (std. dev. 14.6 min). The mean completion time for the second round was 29 min (std. dev. 15.6 min).

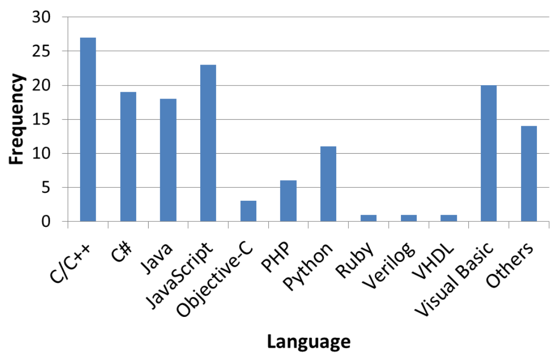


Figure 13. Histogram of languages for experienced programmers.

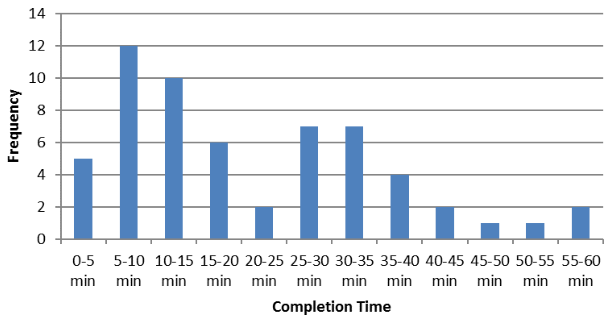


Figure 14. Completion times for all participants in first round.

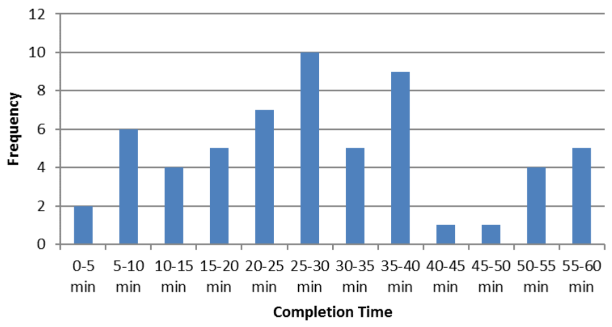


Figure 15. Completion times for all participants in second round.

A.4 Data Availability

Artifacts, including the IRB instructions document, study instructions and questionnaire provided to participants, and raw data, are available from the ACM Digital Library.

Acknowledgments

This work was supported by the National Science Foundation under grant CNS-1527510.

References

[1] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. 2015. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *Proc. of 6th Int'l Workshop on Programming Models and Applications for Multicores and Manycores*. 96–105. https://doi.org/10.1145/2712386.2712400

[2] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. 2015. Streams à la carte: Extensible Pipelines with Object Algebras. In *Proc. of 29th European Conf. on Object-Oriented Programming*. 591–613. https://doi.org/10.4230/LIPIcs.ECOOP.2015.591

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. on Graphics* 23, 3 (Aug. 2004), 777–786. https://doi.org/10.1145/1015706.1015800

[4] Michael Buhrmester, Tracy Kwang, and Samuel D. Gosling. 2011. Amazon's Mechanical Turk: A New Source of Inexpensive, Yet High-Quality Data? *Perspectives on Psychological Science* 6, 1 (Jan. 2011), 3–5. https://doi.org/10.1177/1745691610393980

[5] Raymond P.L. Buse, Caitlin Sadowski, and Westley Weimer. 2011. Benefits and Barriers of User Evaluation in Software Engineering Research. In *Proc. of ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*. 643–656. https://doi.org/10.1145/2048066.2048117

[6] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. 2003. Spidle: A DSL approach to specifying streaming applications. In *Proc. of Int'l Conf. on Generative Programming and Component Engineering*. 1–17. https://doi.org/10.1007/978-3-540-39815-8_1

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[8] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc. 2009. *Crafting A Compiler*. Addison-Wesley, Boston, MA, USA.

[9] Mark A. Franklin, Eric J. Tyson, James Buckley, Patrick Crowley, and John Maschmeyer. 2006. Auto-Pipe and the X Language: A Pipeline Design Tool and Description Language. In *Proc. of Int'l Parallel and Distributed Processing Symp.* 10 pp. https://doi.org/10.1109/IPDPS.2006.1639353

[10] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. 2005. Mining data streams: A review. *SIGMOD Rec.* 34, 2 (2005), 18–26. https://doi.org/10.1145/1083784.1083789

[11] Buğra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. 2008. SPADE: The System S declarative stream processing engine. In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data*. 1123–1134. https://doi.org/10.1145/1376616.1376729

[12] James Gosling. 1997. The feel of Java. *Computer* 30, 6 (June 1997), 53–57. https://doi.org/10.1109/2.587548

[13] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. 2008. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters* 18, 2 (2008), 221–237. https://doi.org/10.1142/S0129626408003351

[14] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. 2008. Streamware: programming general-purpose multicore processors using streams. In *Proc. of 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. 297–307. https://doi.org/10.1145/1346281.1346319

[15] Stefan Hanenberg. 2010. Faith, Hope, and Love: An Essay on Software Science's Neglect of Human Factors. In *Proc. of ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*. 933–946. https://doi.org/10.1145/1869459.1869536

[16] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *Comput. Surveys* 46, 4 (April 2014), 46:1–46:34. https://doi.org/10.1145/2528412

[17] Ken Kennedy, Charles Koelbel, and Hans Zima. 2007. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proc. of 3rd ACM SIGPLAN Conf. on History of Programming Languages*. 7–1–7–22. https://doi.org/10.1145/1238844.1238851

[18] Aniket Kittur, Ed H. Chi, and Bongwon Suh. 2008. Crowdsourcing User Studies with Mechanical Turk. In *Proc. of SIGCHI Conf. on Human Factors in Computing Systems*. 453–456. https://doi.org/10.1145/1357054.1357127

[19] Andrew J Ko, Thomas D Latoza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3

[20] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (May 2006), 33–42. https://doi.org/10.1109/MC.2006.180

[21] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept. 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[22] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. *ACM Trans. on Parallel Computing* 2, 3, Article 17 (Sept. 2015), 42 pages. https://doi.org/10.1145/2809808

[23] Michael J. Lee and Andrew J. Ko. 2012. Investigating the role of purposeful goals on novices' engagement in a programming game. In *Proc. of IEEE Symp. on Visual Languages and Human-Centric Computing*. 163–166. https://doi.org/10.1109/VLHCC.2012.6344507

[24] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. 2010. Deadlock Avoidance for Streaming Computation with Filtering. In *Proc. of 22nd ACM Symp. on Parallelism in Algorithms and Architectures*. 243–252. https://doi.org/10.1145/1810479.1810526

[25] Chuan-Kai Lin and Andrew P. Black. 2007. DirectFlow: A domain-specific language for information-flow systems. In *Proc. of 21st European Conf. on Object-Oriented Programming*. 299–322. https://doi.org/10.1007/978-3-540-73589-2_15

[26] Ying Liu, Nithya Vijayakumar, and Beth Plale. 2006. Stream processing in data-driven computational science. In *Proc. of IEEE/ACM Int'l Conf. on Grid Computing*. 160–167. https://doi.org/10.1109/ICGRID.2006.311011

[27] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. on Graphics* 22, 3 (July 2003), 896–907. https://doi.org/10.1145/882262.882362

[28] Shane Markstrum. 2010. Staking Claims: A History of Programming Language Design Claims and Evidence: A Positional Work in Progress. In *Proc. of Workshop on Evaluation and Usability of Programming Languages and Tools*. 7:1–7:5. https://doi.org/10.1145/1937117.1937124

[29] Peter Mattson, Andrew Chang, Ujval J. Kapasi, Scott Rixner, John D. Owens, Jinyung Namkoong, Brucek Khailany, William J. Dally, and Brian Towles. 2001. Imagine: Media processing with streams. *IEEE Micro* 21, 2 (March/April 2001), 35–46. https://doi.org/10.1109/40.918001

[30] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. 2010. S4: Distributed Stream Computing Platform. In *Proc. of IEEE Int'l Conf. on Data Mining Workshops*. 170–177. https://doi.org/10.1109/ICDMW.2010.172

[31] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proc. of ACM Int'l Symp. on Foundations of Software Engineering*. 155–165. https://doi.org/10.1145/2635868.2635922

[32] Dennis M. Ritchie. 1993. The Development of the C Language. *SIGPLAN Not.* 28, 3 (March 1993), 201–208. https://doi.org/10.1145/155360.155580

[33] John W. Romein, P. Chris Broekema, Ellen van Meijeren, Kjeld van der Schaaf, and Walther H. Zwart. 2006. Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer. In *Proc. of ACM Symp. on Parallelism in Algorithms and Architectures*. 59–66. https://doi.org/10.1145/1148109.1148118

[34] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Niklas Elmqvist, Steven Jacobs, and Nicholas Diakopoulos. 2016. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (6 ed.). Pearson Education, Limited, London, England.

[35] Andreas Stefik, Stefan Hanenberg, Mark McKenney, Anneliese Andrews, Srinivas Kalyan Yellanki, and Susanna Siebert. 2014. What is the Foundation of Evidence of Human Factors Decisions in Language Design? An Empirical Study on Programming Language Workshops. In *Proc. of 22nd Int'l Conf. on Program Comprehension*. 223–231. https://doi.org/10.1145/2597008.2597154

[36] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4 (Nov. 2013), 19:1–19:40. https://doi.org/10.1145/2534973

[37] Robert Stephens. 1997. A Survey of Stream Processing. *Acta Informatica* 34, 7 (1997), 491–541. https://doi.org/10.1007/s002360050095

[38] Bjarne Stroustrup. 1986. *The C++ Programming Language*. Addison-Wesley, Boston, MA, USA.

[39] Jeffrey Stylos and Steven Clarke. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proc. of 29th Int'l Conference on Software Engineering*. 529–539. https://doi.org/10.1109/ICSE.2007.92

[40] William Thies and Saman Amarasinghe. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*. 365–376. https://doi.org/10.1145/1854273.1854319

[41] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proc. of 11th Int'l Conf. on Compiler Construction*. 179–196. https://doi.org/10.1007/3-540-45937-5_14

[42] Joseph G. Wingbermuehle, Roger D. Chamberlain, and Ron K. Cytron. 2012. ScalaPipe: A Streaming Application Generator. In *Proc. of Symp. on Application Accelerators in High-Performance Computing*. 44–53. https://doi.org/10.1109/SAAHPC.2012.14