Towards validity for a formative assessment for language-specific program tracing skills

Greg L. Nelson
University of Washington
Paul G. Allen School of Computer Science & Engineering
Seattle, Washington
glnelson@uw.edu

Andrew Hu, Benjamin Xie, Amy J. Ko
University of Washington
The Information School, DUB Group
Seattle, Washington
andrewhu@uw.edu,bxie@uw.edu,ajko@uw.edu

ABSTRACT

Formative assessments can have positive effects on learning, but few exist for computing, even for basic skills such as program tracing. Instead, teachers often rely on overly broad test questions that lack the diagnostic granularity needed to measure early learning. We followed Kane's framework for assessment validity to design a formative assessment of JavaScript program tracing, developing "an argument for effectiveness for a specific use." This included: 1) a fine-grained scoring model to guide practice, 2) item design to test parts of our fine-grained model with low confound-caused variance, 3) a covering test design that samples from a space of items and covers the scoring model, and 4) a feasibility argument for effectiveness for formative use (can target and improve learning). We contribute a distillation of Kane's framework situated for computing education, and a novel application of Kane's framework to formative assessment of program tracing, focusing on scoring, generalization, and use. Our application also contributes a novel way of modeling possible conceptions of a programming language's semantics by modeling prevalent compositions of control flow and data flow graphs and the paths through them, a process for generating test items, and principles for minimizing item confounds.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

Assessment, program tracing, validation, validity, item generation

1 INTRODUCTION

Formative assessments can help improve learning when they provide actionable information for better targeted instruction, practice, and feedback [10, 23]. Decades of evidence show formative feedback has one of the best effect sizes for learning interventions, according to a review of 500 meta-analyses [16] and another similar review [17]. Within computing education, validated formative assessments may help in learning foundational skills, such as program tracing, program writing, problem-solving, and other computing skills [55]. Students in computing also prefer formative assessment, looking for "assessment as guidance and opportunity to learn" [43].

Computing education research (CER) has strong initial work creating assessments with validity evidence for measurement of a construct, such as self-efficacy [6] or CS1 knowledge [51], but the opportunity to design or evaluate assessments for validity specifically for formative use remains unexplored. Decker and McGill catalogued instruments used or published between 2012-2016 by searching online and reviewing the table of contents for ICER and the CSE and TOCE journals. They found 13 instruments for knowledge of computing or computer science, 3 for CS1, 1 for CS2, 6 for computational thinking, and 3 for advanced skills; and 31 non-knowledge instruments that "...measured constructs such as self-efficacy, anxiety, confidence, enjoyment, sense of belonging, intent to persist, and perceptions" [8]. Margulieux et al. reviewed measurement practices in CER papers reporting qualitative and quantitative human-subjects studies from 2013-2017 and found 16 standardized computing-specific measurements used. None of these present arguments for validity for formative use of these measures.

While summative and formative assessments can have similar items, formative assessments must be much more carefully designed to diagnose what learners do and do not know. For example, a formative assessment that asked learners to write programs might fail to precisely identify what learners do and do not know about prerequisite skills, such as a programming language's syntax and semantics [55]. And in fact, prior work shows many course exams and even validated assessments of programming knowledge have items that require 10-20 concepts to answer correctly, even for basic skills like program tracing [27, 28, 38, 51]. This lack of granularity can result in learners successfully learning several aspects of a programming language's semantics, but then getting a zero on an exam because they do not understand a single operator (as occurred with modulus operators in [22]). Moreover, if such assessments produce low scores that do not reflect learning progress, a learner might feel confused, demotivated, and even come to believe they lack sufficient ability in programming to continue studying it.

To better design and evaluate formative assessments, we need a validity framework that accounts for specific uses of a test's score. Widely accepted approaches to validity, however, tend to overlook how a test's score will be used, instead focusing more narrowly on statistical properties of a test such as accuracy and reliability [19, 32]. For example, in making and doing validity studies for the FCS1 test, Tew et al. describe how they created items to cover CS1 concepts, and did item-response theory analysis and think-aloud studies to identify items with variance caused by confounds (such

Koli Calling '19, November 21–24, 2019, Koli, Finland

ACM ISBN 978-1-4503-7715-7/19/11.
https://doi.org/10.1145/3364510.3364525

as poor quality distractors or difficult to understand questions) [12]. While this traditional approach to validity helps verify that a test measures what it intends to measure, it does not necessarily verify that the measure itself is meaningful or helpful in the world - for example, how using the measure affects a variety of learning outcomes over time.

To address this problem in this paper, we will use Kane's framework for validity to design a formative assessment for program tracing skills. The educational measurements research community increasingly uses Kane's validity framework, which focuses on evaluating a *validity argument* for the effectiveness of an assessment for a specific use (such as formative assessment) [20]. As a measure of adoption, Kane's work has over 900 citations in just 5 years. Kane's framework suggests formative assessments also need a *fine-grained scoring model* useful for guiding remedial instruction and practice; *granular items* that test small parts of the scoring model; and an *argument for why a test is valid for formative use*, describing how the assessment can target weak skills to improve learning, ideally without indirectly harming other outcomes like self-efficacy.

While prior work on CS assessments achieve some of these goals, most lack key elements from Kane's framework. For example, existing assessments [2, 35, 48, 49, 52, 58] lack a *fine-grained scoring model*, because their single numeric score output is difficult or impossible to map to specific feedback (e.g., when a learner gets 7 of 27 questions correct on the SCS1 [35], it is unclear what they need to learn to improve). Concept inventories try to identify known misconceptions [1, 41, 50] and benchmark exams have been used to evaluate granular educational outcomes [26, 29, 46, 47], but none of these present arguments or evidence for using them for diagnostic feedback intended for learners. Some prior work has proposed item designs [9, 24, 25] or identified issues with existing item designs [28, 54], but without evaluating these items for formative use. Several have studied item difficulty and item measurement characteristics for program writing [27, 46, 47, 57], but few have qualitatively studied the response process to see how learners answer the items and what reasoning they actually use (e.g. [12, 22]). Some assessments offer more granular scoring models [14, 15, 44, 57], but only occasionally offer small evaluations of item characteristics [11, 27]. Some work lists example questions to assess the neo-Piagetian developmental stage of a person's tracing skills, without a validity argument for formative use of those questions [25]. Finally, while some adaptive learning [13] and tutoring systems [4] give feedback based on performance and may improve learning, none have tried to validate how well the models of knowledge internal to these systems measure knowledge or skills for computing.

To address these gaps, this paper contributes two things: 1) a distillation of Kane's framework, situated in the domain of computing, and 2) a novel application of Kane's framework to the design of formative assessment of program tracing knowledge. As we demonstrate how to apply Kane to a formative tracing assessment, we describe a model of program tracing skills (Section 3.1), then contribute a fine-grained scoring model for part of it (Section 3.2), the design of granular items to precisely observe tracing skills (Section 3.3), a test design that samples from a space of possible items (Section 3.4), and a feasibility argument for formative use of our test (Section 3.5). These contributions provide an exemplar for our community's continued and critical work on CS assessments, building upon advances in educational measurement.

2 KANE'S FRAMEWORK FOR ASSESSMENT VALIDITY

To help us design an assessment for formative use, in this section we describe Kane's framework for validity. Within our description, we will contribute a demonstration of how the framework can help us better understand the strengths and weaknesses of computing assessments, with respect to some particular use.

Prior validity studies for assessments in CER have partly but not fully embraced the depth of validity frameworks from educational measurement. In practice, studies have defined validity as a form of statistical checklist: if one performs a set of experiments and observes particular statistical properties, then the test is "valid." These conceptions of validity are not necessarily wrong, they are just narrow, overlooking factors that improve our ability to evaluate and design assessments. The educational assessment and measurement community have recently embraced validity frameworks that account for broader factors such as how a test will be used and what consequences it might have to learners [19, 32], such as Messick [31] and Kane [20], each with over 900 citations.

Kane, which we will distill and situate in computing education in this section, defines validity as *the quality of a four-part argument* about the assessment and the assessment's effectiveness for a specific use. An argument is a network of claims, evidence, and assumptions. To illustrate argument quality, suppose a test used in college admissions is biased against particular groups, and using the test changes the distribution of admitted students in an unfair/unequitable way, which is ethically undesirable. Thus, even if that test might have desirable psychometric properties in the aggregate, the test may still have a low quality argument–and thus low validity–for use in admissions. Now, if someone added to the argument, noting that, after evaluating ten different assessments for admissions, this test had the *lowest* bias, and the test is better than alternatives (even not using any assessment), then we might say it has a *stronger* argument–and thus stronger validity–for admissions.

In the rest of this section, we detail the four parts of a validity argument in Kane's framework, providing examples of how they apply in the domain of computing. We then interpret the implication of Kane's framework for the specific use of formative assessment.

2.1 Validity arguments address scoring, generalization, extrapolation, and use

Kane's validity framework breaks down a validity argument into four parts, starting from 1) *scoring*: how items are scored, to 2) *generalizing* from the scoring model to expected performance on test tasks, to 3) *extrapolating* scores from test tasks to other tasks in other contexts (including construct claims, e.g. measuring CS1 knowledge), to finally 4) *use*: using the scoring model for some decision and evaluating the broader effects of this use. The main benefit of this breakdown is avoiding applying evidence for one part (e.g. extrapolation) as evidence for another (e.g. use), as each requires different evidence. Kane also recognizes validity arguments take *many* published studies to develop incrementally (as we will note in the evidence for the tracing assessment we present later).

To explain these four parts, imagine a CS1 midterm we have designed for formative assessment, offering learners granular, diagnostic feedback about what skills they have and have not yet mastered. For practical purposes we can't include every possible item, so we take some sample of them we hope is representative. We make the test, administer it to students, have TAs score it, give solutions back with grades and solutions to the problems, noting how well they have mastered each concept. We hope this improves learning in the class, and might use the assessment iteratively.

2.1.1 Scoring. **The *scoring* argument has claims about the procedure for translating observed performance on the test to the scoring model for a student.** This argument should include a description of how test performance is observed (such as learners' handwritten markings on an exam printout) and then translated into a scoring model (such as points for each question, sub-scores for tracing and writing parts, and an overall score). For the CS1 midterm example, the scoring argument might include claims like "The TAs use our grading rubrics and don't see the test-taker's name, so their scoring procedure lacks bias" and "The tracing questions have objective, unambiguous answers so we can score them without bias."

To provide evidence for scoring claims, Kane describes several sources of evidence, from analytical arguments to empirical studies. Arguments may justify how the structure of items matches the scoring model; for example in our CS1 midterm, the tracing score is the total of the score of each tracing item. To further strengthen the argument, a panel of experts might review scoring procedures for each item and check implementation correctness (e.g. the correct answers are the actual correct answers for each item). For human-rated questions, empirical studies might assess inter-rater reliability, including analysis of sources of bias.

By considering an assessment's scoring argument, we can better understand an assessment's strengths and weaknesses, and perhaps improve them. Decker's CS1 assessment includes program writing items graded by humans, so Decker makes claims for inter-rater reliability by sharing the explicit objective and subjective scoring rubrics for individual writing questions, using triage theory to justify partial credit rubrics [7]. According to Kane, to further strengthen this claim, one might study multiple graders grading the same person's assessment for a sample of TAs with varying experience. This might lead to improvements in the rubrics.

2.1.2 Generalization. Even with an unbiased scoring process, an assessment may have other flaws, such as poor test-retest reliability. Test-retest reliability refers to the variation in scores when we give the same person our CS1 midterm, then have them do the same or similar test again (e.g., last year's midterm). Some variation seems inevitable, but too much variation might signal flaws in the generalizability of the test. If we knew where this variation comes from, we might improve the test or use it more appropriately.

Kane's framework breaks down test-retest reliability into several different sources of variation. The test's items are just one sample from a space of possible test items and item designs; for example, our CS1 midterm might have hard writing questions that do not represent our CS1 learning objectives well. The testing context includes how the test is administered; for example, time limits, room size and noise, giving the test in a class for credit vs. extra-credit or other compensation. A testing occasion is a particular instance of giving the test, and includes other factors that vary with time; for example, a test-taker's knowledge, mood, tiredness, etc.

According to Kane, a *generalization* argument should include claims about these sources of variation, and broadly includes claims about how the scoring model relates to expected performance on possible test tasks (not real-world tasks). This argument should define the items on the test and the test design, and define the space of test tasks; for example, for our CS1 midterm we might include the types of tracing questions, what constructs they use, and the types of writing questions. This argument may also describe or quantify the sources of variation in scores; for example, sampling variation in scores that comes from aspects of items on the test, including item design, generation and selection processes. This argument may also qualify the testing context, such as testing conditions like time-limits, time of administration, or other factors the assessment designers consider important for lowering score variation.

To provide evidence for generalization claims, Kane describes several sources, from analytical arguments about the process for choosing test items, to stronger empirical studies of test-retest reliability. The main source of empirical evidence is measuring test-retest reliability (ideally including a parallel test form with a different sample of items). Additional studies should target potentially large sources of variation in depth; for example, varying the amount of time learners have between test and re-test, or varying testing conditions. Theoretical and analytical arguments should focus on the item selection process, such as having experts review the space of items, item designs, test design, and chosen items. Arguments should also focus on how the test and test conditions mitigate the strongest potential sources of variation. For example, for large (and hard to define) domains like CS1 test questions, the relatively small sample of items on the test may lead to large variation.

Prior work on CS assessments have contributed some evidence of generalization, but have many opportunities for improvement. For example, in the FCS1 [12] and replicated SCS1 [34], potentially high sources of variation are the use of pseudocode and the selection of items from the very broad space of possible CS1 tasks. The FCS1 empirical test-retest reliability evaluation focused on using pseudocode vs. familiar language, finding a .572 correlation between the two. Kane's framework suggests this could be strengthened by generating a parallel form with different items and measuring test-retest reliability; this was partly done in a validity study not specifically designed to estimate the sources of variation. Parker et al. created the SCS1 with many new items then measured test-retest reliability with the FCS1 [34]. They found a correlation of .566, about the same as the FCS1 testing and retesting with the pseudocode and language-specific FCS1 versions with the same questions. This means sampling variation from item selection may be similar to language transfer variation in score. Kane's framework also suggests broad domains like CS1 tasks may be hard to sample from (as the test's sample of items is small compared to the broad space); thus, additional more specific tests may also be useful.

2.1.3 Extrapolation. Even if a test has strong arguments for its scoring model and generalization, the scores themselves may not

predict real-world task performance, i.e. scores may not *extrapolate* to other tasks in non-test contexts. Real-world contexts differ in many ways, including motivation, incentives, tools, task duration, and access to resources like the internet and other people. Beyond context, the nature of test tasks may lead to a different problem-solving *process* than real tasks; for example, the process of answering multiple choice questions about program writing differs from writing software in a team setting.

According to Kane, to make an *extrapolation* argument, one must 1) describe which real-world tasks and contexts and 2) show how real-world task performance is similar to test-task performance. Task descriptions range from broad (e.g "computing", "writing code") to more specific (e.g. "writing sorting algorithms", "creating database schemas"). The most specific descriptions describe the *process* of doing the task; for example, the steps and conditions needed to complete each step, with examples. To describe task contexts, one describes conditions on the environment, such as places (in an organization, in a class), social situations (alone, with others, with access to StackOverflow), and available resources (having test suites, having an IDE). One must also describe the limits of extrapolation; for example, a software engineering test might extrapolate to front-end software development but not embedded systems development.

To show how real-world task performance is similar to test-task performance, Kane talks about analytical and empirical evidence. Analytical arguments describe key factors shared between real-world and test task performance; they may also describe similarities between the processes of real and test tasks. Empirical evidence of processes comes from observation and think-aloud studies of real and test tasks. Empirical studies may directly associate scores and a comparable metric on real tasks; this may involve comparing the test with a more expensive measure of real-world performance (such as a panel of expert software developers reviewing a more real-world writing task).

To put this in practice in computing, we should think about strengthening extrapolation claims for writing assessment tasks to more real-world writing tasks. The FCS1 used multiple choice questions (MCQs) to measure writing skills, and also made a validity argument analytically by mapping FCS1 tasks to CS1 content areas, with experts reviewing their mapping for representativeness [12]. Using MCQs to measure writing likely limits extrapolation due to process differences, compared to writing in a real-world IDE with internet access. Instead of MCQs, Decker's CS1 assessment had learners write code on paper, using a rubric for grading [7]; that better supports an extrapolation argument because of similarity between tasks. Kane suggests directly comparing test scores and a comparable metric on more real tasks. While some studies compare incidentally when using multiple instruments, purposefully designed studies will have higher quality; for example, a study correlated the FCS1 and a single 90-110 minute writing task with an IDE [53]. Beyond this, Kane's framework suggests we could further strengthen writing assessments by comparing them with real-world writing in software development in different contexts.

2.1.4 Use. Kane argues that even when a test predicts real-world or test task performance, this does not imply we can use the test for any arbitrary purpose effectively. For example, if we wanted to evaluate two designs that teach a sub-part of CS1 knowledge, using a broad CS1 assessment may not be sensitive enough to detect differences. Similarly, while our hypothetical CS1 formative midterm might be great at identifying learners' brittle knowledge about programming language, it might be quite poor at measuring the programming knowledge of TAs you might hire for a class, as it may not discriminate well between learners who have all excelled at a course's learning objectives.

For Kane, the argument for a specific *use* includes 1) how the scoring model is used to support decisions, and 2) intended and unintended effects of the decision and giving the test. First, the argument should describe the decision procedure for how the score is used; for example, a learner may take the next class in a sequence if their final exam score > 70%. The argument should include the inputs and outputs and people involved. For example, an assessment may be used by researchers as input to calculate, report, and interpret some statistic based on it, such as an effect size; the decision procedure includes the particular statistical estimation procedures, and documentation written by assessment designers to guide appropriate statistical use and interpretation. The argument should also include the context of the decision; for example, using the assessment in K-8 classrooms.

Second, the argument should consider the intended and unintended effects of the decision. The argument should evaluate intended effects; for example, for formative use, the intended effects are improving learning for particular skills. The argument should also evaluate unintended effects of giving the test and using it; for example, if our CS1 midterm hurts some learners self-efficacy so they drop the class or give less effort, it may not be helpful overall for improving learning. The argument should include costs, unintended effects, and how negative unintended effects are mitigated; for example, mitigating harm to self-efficacy. The argument should include systemic effects; for example, a standardized computing test may lead to teaching to the test instead of teaching for real-world tasks. Beyond evaluating intended, unintended, and systemic effects in isolation, the argument should compare alternatives; for example, in our earlier argument for using a biased test for college admissions, that test had the lowest bias of the options and was better than using no test at all.

To provide evidence for use claims, Kane describes several sources, from analytical arguments about expected effects to direct empirical evaluation of effects and a cost-benefit analysis. Analytical arguments should estimate effects using properties of the test design, scores, and the decision procedure; these properties come from earlier scoring, generalization, and extrapolation claims. For example, one might argue, "The software design test scores poorly predicts success but somewhat predicts failure in real-world software design tasks. Using the test to recommend training to new hires with low scores has small downsides and larger upsides: if they don't need training, they can opt-out of it, and if they opt-in and need it they may get large benefits." The most direct evidence for use is to use the assessment in practice and empirically assess the effects as part of a cost-benefit analysis. For example, to claim a formative assessment helps in a class, it should be comparatively evaluated in that context, as other alternatives like going to office hours or seeking human help may already address that material sufficiently for the control group. Pragmatically, smaller initial studies in lab

settings may be required to overcome cost, logistical, and political barriers to larger studies.

By considering an argument for a specific use of an assessment, we can better understand an assessment's strengths and weaknesses, and perhaps improve them. For example, the SCS1 has been used as a pre and post test to compare learning interventions [33], yet Kane points out that validity argument for the score's interpretation as indicating CS1 knowledge "... does not, in itself, validate [any] score use." While the SCS1 has excellent validity studies for our field, such as internal reliability measures, it lacks SCS1 test-retest reliability studies to strengthen an empirical argument specifically for pre-post use. This could be strengthened by doing an instructional sensitivity study that attempts to carefully examine the test's ability to detect changes in knowledge caused by direct instruction [40].

2.1.5 Summary. Obviously, a summary of Kane's extensive validity framework cannot fully capture the nuances of Kane's positions on validity and prior conceptions of validity. For more detail, see [3] for an applied summary, [20, 32] for depth, and [21] for discussion.

3 A PROGRAM TRACING ASSESSMENT FOR FORMATIVE USE

In the previous section we showed how Kane's framework for a validity argument can help researchers better design assessments by considering their strengths and weaknesses for a specific use. In this section we will use Kane's framework as we carefully design our assessment and provide theory-based validity arguments as design rationale. We'll discuss five aspects of our assessment design: our model of how tracing is performed in non-test contexts, how we model test performance with a *fine-grained scoring* model, how we design *granular* items suitable for scoring, how we compose those items into a test, and how we score the test and use scores for formative use.

3.1 Tracing Performance Model

Within Kane's framework [20], a key requirement for developing a validity argument for a formative assessment is a "model of the target domain." For program tracing, that means having a theory of what *causes* both novice and skilled tracing performance in contexts outside test tasks. Such a model helps us design an assessment, but also understand limitations of our design and how to evaluate it.

Our model of the target domain of program tracing builds upon prior work [33], which proposes that program tracing performance by any learner is ultimately caused by some set of beliefs about how specific programming language (PL) constructs execute (in PL terms, beliefs about their semantics). For example, learners skilled in program tracing likely have beliefs that closely mirror a PL's *actual* semantics; for simple `if` statements without an `else` branch, this would mean having the ability to precisely follow such a conditional's two execution paths (the true and false branches through and around the *then* block). Less skilled behavior may have different beliefs about semantics, which may lead learners to execute a conditional differently (and incorrectly), such as *always* executing the *then* block, independent of the condition.

Beyond beliefs about a PL, we also argue that tracing performance is mediated by other skills, using prior work. For example, *perception* matters because if a learner misperceives an operator (e.g., > instead of <), they may trace incorrectly, even if they know the correct semantics. Similarly, *strategic* skills matter: many studies document how learners sketch and mark paper questions to keep track of values during tracing [5]; teaching strategies for distributing cognition in this way helps people avoid errors [56]. As Schulte's Block Model describes, some learners may use higher level program comprehension strategies, like detecting idioms or inferring semantics from variable names and context, to avoid tracing code [45]. Finally, learners may simply guess in a test setting.

Of course, the brief summary above does not fully describe the rich complexity of program tracing skills. It does, however, illustrate the challenge of accurately measuring tracing ability at a level of granularity sufficient for guiding learning. It also illustrates the kind of domain nuances necessary for strong validity argument.

3.2 Scoring Model

As Section 2.1.1 and 2.1.4 discussed, Kane's framework suggests we need a *fine-grained* scoring model for formative use. Achieving this requires requires some simplification of the complexities in the domain of tracing. Here we describe the simplifications we made to model tracing skills while retaining granularity needed for targeting weak skills with remedial instruction and practice. This constitutes the foundation of our *scoring* argument.

First, accounting for all factors in Section 3.1 would require a very sophisticated and complex assessment that covers both programming language (PL) semantics knowledge, strategic knowledge, and many other factors. In this exploratory work, we will simplify this by focusing on PL semantics knowledge, by not explicitly modeling tracing strategy, code perception skills, meta-cognition, or other response processes (like guessing). We will partly address these simplifications in our item and test design in later sections.

Given our decision to only model PL semantics knowledge, we next need to decide how to model this knowledge; our choices here require a more extended justification and discussion. Novice understanding of a PL's semantics may not align completely with a PL's actual semantics. Novices can have brittle models of a particular language construct's execution, but they can also believe that language constructs have dependencies that the PL actually does not have (e.g., assuming assignments to local variables change aliased global variables). If our scoring model cannot represent these varying conceptions of semantics, we will not be able to guide the specific feedback needed to refine these conceptions.

To model varying conceptions of semantics, we begin with the observation that if a learner was entirely free of misconceptions, their PL semantics knowledge *could* be precisely modeled as what semantics they can and cannot accurately mentally simulate. This might be true for people who have mastered a programming language, but is certainly not true for everyone else: prior work has identified misconceptions including *imagined* interactions between the semantics of different constructs [36, 37]. For example, a novice may understand paths involved in a function that calls another function, but trace incorrectly when a function calls itself, even though the PL's semantics makes no distinction of such cases. Thus, we cannot model that conception if our scoring model only tries to model knowledge of individual language construct semantics.

A much more fine-grained scoring model for tracing skills is to model a learner's ability to accurately trace *every possible* concrete

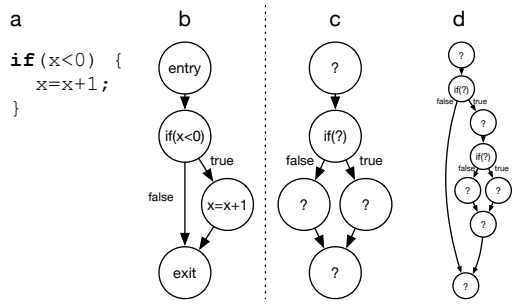

Figure 1: At label (a) a concrete program, then (b) its control flow graph (CFG), with specific conditions and assignments. At (c) a *semantic* CFG, (d) a *composite* semantic CFG, with ? placeholders.

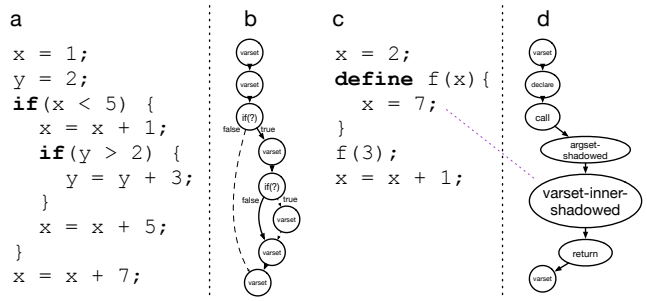

Figure 2: (a) An example item (item 28) and (b) the SemCFG path it attempts to assess. (c) A code example with (d) SemCFG with data flow varset-inner-shadowed.

program that a PL could express. Obviously, this highly granular approach is infeasible, as the space of possible programs is infinite.

Instead, we propose a middle ground, which attempts to achieve *sufficient* granularity by modeling prevalent *compositions* of individual PL constructs, and modeling learners' ability to correctly trace all possible executions of those compositions. For example, rather than just modeling knowledge of conditionals in isolation, we will model knowledge of conditionals nested inside of conditionals (and all their possible execution paths), or conditionals inside of loops (and all their possible executions paths). For example, `if(...){ if(...){...} }` has three possible execution paths: the first `if` executes but not the second `if`, both execute, or neither executes.

To explain this more precisely, we will use two representations of program execution from program analysis. We will represent PL construct semantics as *control flow graphs* (CFGs) to capture the possible executions of an individual construct, and *data flow graphs* (DFGs) to capture dependencies between uses and definitions of variables. A CFG represents potential executions through AST; for example, the program in Figure 1a has the CFG shown in Figure 1b. However, rather than just using CFGs to represent concrete programs, we use them here to represent the abstract semantics of a PL construct. For example, Figure 1c shows the *semantic* CFGs (SemCFG) for any *arbitrary* conditional, one of which is the concrete CFG in Figure 1b, with placeholders (represented by ?) for whatever code, if any, might execute for the then and else branches. All of the (two) paths through such SemCFGs represent all of the valid semantics a learner could know about conditionals.

With these concepts alone, all we can represent are individual PL constructs. Returning to our idea of representing compositions of pairs of semantics, we can create permutations of SemCFGs to represent prevalent compositions that learners might misunderstand. These permutations fill one or more of the the unspecified ? subgraphs with other SemCFGs. For example, Figure 1d shows a *composite* SemCFG that represents a nested conditional with no outer else branch. All of the (three) paths through this composite SemCFG represent all of the valid semantics a learner could know about nested conditionals.

By defining a collection of composite SemCFGs to detect many possible misconceptions of a PL's semantics , we can build a scoring model that covers all possible paths through all of the SemCFGs we choose to include in our test, giving a 1 or 0 for each path that a learner can or cannot mentally simulate correctly. We will define a notation for these SemCFG paths as nested lists of SemCFG paths, since interpreter execution paths are just node-edge combinations in the graph. For example, for the nested SemCFGs in Figure 1d, the three paths in our notation would be: 1) *if-true{if-false}*, 2) *if-true{if-true}*, and 3) *if-false*. Our proposed scoring model has a 0 or a 1 for each of those paths. We define correctness as executing a complete valid path for each node in the SemCFG.

To account for data flow semantics, we further extend this model by adding a limited set of *data flow path elements*, which are data flow patterns representing variable assignments and references. These can be included in the unspecified subgraphs of SemCFGs (the ?'s in Figure 1c & 1d) to help model interactions between control flow and program state. To express data flow, we will also include in our SemCFG path notation the *scope* that was accessed; for example, *varset-local* represents setting a value in local scope, *varset-global* a value in global scope, and likewise for *varget-local* and *varget-global*. To express more complicated data flow, we encode which scopes were available for a variable resolution, for example *varset-local-shadowed* for when a local variable was set but it also shadowed a global variable (this represents the nodes available for data flow at that point) (see Figure 2c for code whose SemCFG has a varset-local-shadowed and later Figure 3). While these do not represent *all* data flow scenarios in a PL, we focus on important cases for common PL semantics where novice learners might benefit from formative feedback if they got it wrong (as that is our intended use, per Kane's framework).

Combining all of these concepts, our theoretical scoring model represents whether a learner can accurately trace each path through a set of SemCFGs that we believe can detect many possible misconceptions of PL semantics, at a fine-grained level of detail. This granularity is critical to identifying specific misconceptions of both individual PL constructs and compositions of PL constructions.

3.3 Item Design

The scoring model in the previous section represents what skills we wish to test. To actually observe these skills and score them, we need items. Kane's framework says we should argue how our item structure matches our scoring model. Therefore, here we extend our scoring argument by discussing how to design items compatible with our scoring model that minimize confounding factors not

accounted for in our scoring model. Figure 3 shows an example item at right.

For the program part of our item design, most of our items followed the program structure in Figure 2a. Figure 2a shows an example of code that assesses the true-then-false path (*if-true{if-false}*) of a nested conditional (represented by the SemCFG in Figure 2b). The program's x and y variables serve three roles. First, their values ensure a particular control flow path is followed; for example in Figure 2a, `x=1` ensures the first `if(x<5)` is true, and the y value of 2 ensures the `if(y>2)` is not true, which makes the code follow *if-true{if-false}* instead of *if-true{if-true}*. Second, the variables are the basis of the two prompts for each item; for example, "*x is ...*" and "*y is ...*" in Figure 3. Third, by modifying one or both variables before and after each branch in the program, the item ensures that every unique control flow path the learner *might* follow while mentally simulating the program (correct or incorrect) would result in different values for x and y, enabling unambiguous interpretations of the item response. This item design helps ensure that if the learner knows the values in the variables at the end of execution, we can infer they know all the semantics required to compute those values.

To generate most items for the types of *SemCFG* paths we defined in our scoring model, we followed a program template that supported at most two PL constructs (e.g., a single conditional, a conditional inside of a loop, a function call inside of a loop, etc.): `varset path { varset path { varset } varset } varset` where each `varset` was the assignment of x and/or y and each `path` represented the branch of a control structure (e.g. *if-true*, *while-false*). The brackets `{}`'s represent potential nesting. For example, in the nested conditional example in Figure 2a, we instantiated the template as `update-x update-y if-true { update-x if-false { update-y } update-x } update-x`. We then translated these instantiated templates into the program like the one above, using inequality expressions to ensure the desired branch, and assignments to ensure non-overlapping values of x and y for all possible incorrect executions of the program (for example, see Figure 2a).

The template above did not work for all of the semantics we wanted to assess (e.g., local and global variables, shadowed variables, function calls and returns), but we generated similar templates for such programs that similarly mitigated confounds. For example, our function call and return items used global variables to isolate knowledge of scoping, function arguments, and return values, such as the item in Figure 3.

While the item design above avoids some confounds, it also introduces some. In particular, it introduces concrete program elements such as literal values, operators, and expressions that may result in *slips* and *guesses*, which confound our interpretation of item responses (these terms come from models used in educational assessment). *Slips* occur when some aspect of an item causes a learner's answer to be incorrect, even when the learner can trace the item (for example, misreading an operator, or overlooking a return statement). *Guesses* occur when we infer incorrectly that the learner knows when they do not (for example, making an arithmetic error that inadvertently led to a correct answer).

To mitigate the influence of confounds that cause slips and guesses, we developed the following principles for what program literals and operators to choose to fill in our item templates:

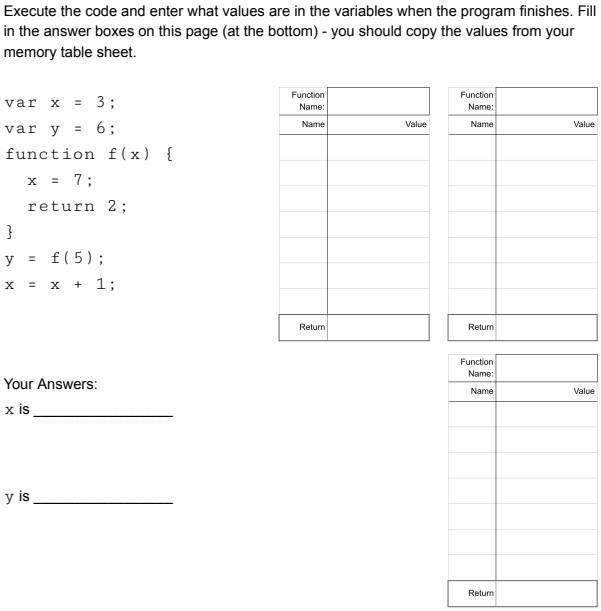

Figure 3: Item 21 on our tracing assessment, with tracing scaffolding tables on right and answer spaces on bottom.

3.3.1 Arithmetic. For any arithmetic, we preferred small numbers and simple operations to maximize the likelihood people know the operation and minimize working memory errors. While arithmetic errors *might* indicate some misunderstanding when assessing the + operator, arithmetic errors confound measuring more advanced constructs. We also avoided negative numbers, as they likely lead to more arithmetic mistakes.

3.3.2 Perception. Tracing requires accurately perceiving syntax (e.g., not mistaking x for y). To address this, we ensured that small errors in variable look up still led to correct performance for conditionals. For example, in the program `x=4, y=3, if(x<5){ }`, the control flow will always do the inner body of the `if`, even if the learner accidentally looks up y.

3.3.3 Idioms. Some patterns in programs are so common, they allow learners to "short circuit" program tracing. To avoid this, we avoided idioms when possible.

3.3.4 Metacognition. Learners can guess or slip based on cues in an item. For example, variable names like *counter* can give clues to behavior (or be distractors that may cause a slip); or a learner may doubt their answer if no variables change. To avoid this, we used single-letter variable names, and for larger nested items we ensured one or more variables modified from their initial value.

3.3.5 Guessing. To motivate honest responses, we explained in the test that guessing makes it harder to infer what learners know.

3.4 Test Design

Given our scoring model and item design, we now discuss our proposed test design for a practical subset of JavaScript semantics and a practical set of compositions of JavaScript language constructs.

As discussed in Section 2.1.2, Kane's framework says we should describe our sample of items, and describe ways to lower score variation to improve *generalization* (and potentially mitigate unintended effects like harming self-efficacy). The argument below therefore constitutes our *generalization* argument. We will not make an extrapolation argument, although we describe steps for one in our discussion section.

Table 1 lists all of the items in our test, corresponding to a set of SemCFGs. The test covers variable assignments, arithmetic operators, inequalities, conditionals, loops, and functions, and prevalent compositions of these constructs. We generated all items using the procedures and guidelines described in Section 3.3.

In addition to these items, our test attempts to remove additional confounds by teaching a tracing strategy and scaffolding tracing on each item. We teach the tracing strategy described in [56], using a written script for proctoring the test and not giving personalized feedback on mistakes learners were making. The script instructs the proctor to show an example of the strategy by writing it in pen, then having the learner try the same example, for three examples.

To support a tracing strategy during the test, the items include memory tables [56] on each page to scaffold notional machine state recall and representation (see right of Figure 3, previous page). We put them next to each problem to make them at-hand, to reduce visual working memory effort in going back and forth between where they were in the code and making changes to the memory table. We also provide extra table sheets.

To mitigate confounds of the test format:

- The test has no time limit, reducing time-management confounds; this may also improve fairness for people with cognitive differences and perceptual differences like low-vision.
- The test is typeset using a larger mono-spaced font, balancing the trade-off of making character recognition easier and distance needed to move perception to keep track of location and moving between the code and the memory tables.
- To mitigate confounding effects of self-efficacy, we order items from least to most difficult, to avoid initial shock and dismay; the test cover page also states "Whatever you know right now about programming, you can learn more with more practice. Your performance on these questions has NO RELATIONSHIP with your ability to learn programming." and "The purpose of this assessment is to help you focus your learning on parts you are less strong with at the moment. Please give your best effort, work carefully, and use the explicit tracing strategy we will show you on the next page."

Given our item design for resisting guesses and slips and our mitigation strategies, we believe it analytically plausible that our scores will generalize within the scope of items on the test, but future work needs to empirically evaluate this. We do not present an argument our test scores extrapolate, but will describe steps for one in our discussion section.

#	Item template and *added SemCFG information*	
1	x=c1>c2	*booleanExpressionFalse*
2	x=c1>c2	*booleanExpressionTrue*
3	x=c1; x=c2	
4	x=c1; y=x-c2	
5	x=c1; y=x+c2	
6	x=c1; y=c2; y=x-y	
7	x=c1; y=x; y=c2	
8	x=c1; y=c2; if(x>c3) { y = y -c4}	*if-false-no-else*
9	x=c1; y=c2; if(x<c3) { y = y +c4}	*if-true-no-else*
10	x=c1; y=c2; if(x>c3) { y = y +c4} else { x=y+c5}	*if-true-has-else*
11	x=c1; y=c2; if(x>c3) { y = y +c4} else { x=y+c5}	*if-false-has-else*
12	x=c1; y=c2; if(x>c3) { y = y -c4} x = x +c4	*if-false*
13	x=c1; y=c2; if(x<c3) { y = y +c4} x = x +c5	*if-true*
14	x=c1; y=c2; while(x>c3) {x=x-c4}	*while-true while-true while-false*
15	x=c1; y=c2; while(x>c3) {x=x-c4}	*while-true while-false*
16	x=c1; y=c2; while(x<c3) {x=x-c4}	*while-false*
17	x=c1; y=c2; while(x>c3) {x=x-c4} y = y+c4	*while-false*
18	x=c1; y=c2; while(x>c3) {x=x+c4} y = y+c4	*while-true while-true while-false*
19	x=c1; y=c2; f(z){return **z**+c3} y=f(c3)	
20	x=c1; y=c2; f() {x=c3} y=x+c4; f();	
21	x=c1; y=c2; f(x){**x**=c3; return c4}; y=f(c5); x=x+c6	
22	x=c1; y=c2; f(x){**x**=y}; y=y+c3; f(c4); y=y+c5	
23	x=c1; y=c2; f(z){y=**z**} f(c3); x=x+c4	
24	x=c1; y=c2; f(x){**x**=**x**+c3; return g()}; g(){return x}; y=f(c4); x=x+c5	
25	x=c1; y=c2; f(z){y=g(**z**); return **z**+c3} g(k){ return **k**+c4}; x=f(c5}	
26	x=c1; y=c2; if(x<c3){y=y+c4; if(x<c5){y=y+c4} x=x+c5;} x=x+c6	*if-true { if-true }*
27	x=c1; y=c2; if(x<c3){y=y+c3; if(x>c4){y=y+c5} x=y+c4;} x=x+c4	*if-false*
28	x=c1; y=c2; if(x<c3){x=x+c4; if(x<c2){y=y+c3} x=x+c5;} x=x+c1	*if-true { if-false }*
29	x=c1; y=c2; f(z){ if (**z** < c3){ return **z**;} return f(**z**-c3);} x=f(c4);	*recursion with 2 recursive calls*
30	x=c1; y=c2; f(z){ if (**z** > c3){ f(**z**-c3);} return **z**-c3);} x=f(c4);	*non-idiomatic recursion with 2 recursive calls*

Table 1: Test items, including the semantic paths they covered. The SemCFG for item 21 is *varset-global varset-global declare varset-global(call{argset-shadow varset-inner-shadow return{const}) varset-global.* For readability we show them as item templates with color for data flow paths local and local-shadowed. We also list parts of the SemCFG in italics when ambiguous, e.g. for item 15 showing the loop executed once with *while-true while-false.* $c1, c2, c3, \ldots$ are integers chosen to fit our item design principles (Section 3.3).

3.5 Scoring and formative use of our assessment

As Section 2.1.4 discussed, Kane's framework says we also need an *argument for formative use*, describing how the assessment can target weak skills to improve learning, ideally without indirectly harming other outcomes like self-efficacy. Given our assessment designed based on a fine-grained scoring model, in this section we will describe 1) how to score our assessment for a learner, and 2) how to use the score to target weak skills with remedial instruction and practice. (This constitutes our *use* argument).

After the learner takes the assessment, we can use each item's overall correctness to assign a *knows* or *does not know* to the SemCFG that item was constructed to assess. If the learner correctly gives the values of variables asked for by the item, then we assign *knows* for the item's SemCFG in the scoring model. If the learner gives any incorrect value, we assign *does not know* for that item's SemCFG in the scoring model.

To use the score to target weak skills with remedial instruction and practice, we can give learners an instructional design specific to the SemCFG, with instruction, a worked example of how to trace code that conforms to the SemCFG, and some practice. For example, for item 15 in Table 1, we might give a worked example for tracing a while loop that only executes once and updates a variable in the body of the loop, give conceptual instruction for while loops such as "while loops can execute zero, one, or many times, depending on the condition", and give additional practice problems.

Our test might be used in many ways and contexts. Instructors might use our assessment or subsets of it in classroom contexts or in lab, to decide what remedial instruction to include in the next class session. Our items might also be useful on exams, as they have more granularity than typical exam questions [27]. Within a specific classroom context, we also envision instructors giving subsets of our assessment throughout a course, corresponding to material covered by that time, then use scores to recommend remedial instruction and practice for those items only (such as a lookup table for each test item with what practice to do if incorrect). Instructors might also give parts or the entire assessment as part of reviewing for midterm or final exams. Lastly, learners might use the test themselves to diagnose their own knowledge, then lookup relevant practice.

4 DISCUSSION

Our contributions in this paper include 1) a distillation of Kane's validity framework, and 2) an application of Kane's framework to design a novel formative assessment of language-specific program tracing. Our particular approach to assessing tracing knowledge has valuable properties for CS learning, including its granular measurement and its potential to support and guide learning. While our core assessment validity arguments for formative use are analytical in nature, rather than empirical, as Kane notes, evidence for the validity of an assessment is accrued over time across many studies.

We plan to prioritize improving our generalization and extrapolation argument by studying response process via think-alouds for confounds on performance. After changing items or item designs to address any issues there, we should strengthen our use argument directly by giving the assessment then using it to target instruction and practice, to evaluate if that improves learning. After that, we should evaluate extrapolation and use claims such as to what degree formative use for test-like tasks supports learners in mental tracing during debugging, broader program comprehension tasks, and program writing. While we hypothesize using our assessment formatively may support that later learning of those more complex tasks, we should study deploying the assessment and gather direct evidence on changes in later learning outcomes.

While there are significant opportunities for improving the assessment and gathering evidence for our validity argument further, Kane's conception of validity [20] as the quality of an *argument for effectiveness for some use* leads us to consider more than just accuracy and reliability. As a first example, if our goal is to use an assessment to formatively support learning, even if our assessment had perfect accuracy but negatively impacted self-efficacy or mindset in the process, it might have a net negative effect on long-term learning, and therefore might not be "valid." As a second example, even if including strategy instruction and scaffolding produced some more slips initially on the test due to unfamiliarity, which is undesirable for traditional notions of validity as accuracy, prior work shows that teaching tracing strategies can help learners [56], so including it may still make for a more "valid" assessment for the goal of improving learning. The question is therefore what accuracy is good enough for particular uses of the assessment, and what are the trade-offs and opportunity costs? As a third example, using Kane's framework led us to expand our considerations beyond just accuracy and reliability, to also include self-efficacy, mindset, and other learning outcomes, which have not typically been measured in knowledge assessment studies in CER (e.g., [34, 51]).

Kane's framework emphasizes considering extrapolation carefully; our field should further investigate assessments that extrapolate well to real-world contexts. For example, writing code is actually done with IDEs that can run code; why then assess writing and tracing code without IDE features [39]? Some past work has tried more authentic IDE-based assessments in labs [18, 42]; to study extrapolation, work should ultimately compare test performance with performance in job or other real-world settings. As another example, our work in this paper is guilty of assessing tracing as an abstract skill, rather than situating it in more real-world contexts of program comprehension, testing, debugging, and other authentic activities that occur in programming and software engineering contexts. Our only defense of this choice is that the skills required for our items seem so fundamental that people need to learn them robustly to learn later concepts (e.g. [25]); this may not be true, nor is it necessarily true that all in computing can be abstractly learned then practically applied. We should seek assessments that extrapolate to real-world contexts, lest we teach only easy-to-assess tasks and skills.

Beyond improving the formative assessment of tracing, our work may have implications for other forms of assessment of computing knowledge. For example, a key idea in our work was building cognitive models of tracing ability out of the formal semantics of programming language constructs. Future work could explore similar approaches for program *writing*, building upon theories that decompose writing into other skills [28, 30, 55].

As we all develop and improve both formative and summative assessments of computing knowledge, it is important to think broadly about their use. We should not only consider factors like accuracy and reliability, but also a much broader diversity of learning outcomes and how assessments shape learning over time. Our contributions are a small step in that direction, including introducing Kane's framework and giving an example of applying it; we hope our community will explore many more.

5 ACKNOWLEDGEMENTS

This material is based upon work supported by Microsoft, Google, Adobe, and the National Science Foundation (Grant No. 1836813, 1703304, 1735123, and 1539179). We have seen a little further by standing on the shoulders of those in CER and other communities.

REFERENCES

[1] Caceffo, R., Gama, G., Benatti, R., Aparecida, T., Caldas, T., and Azevedo, R. A concept inventory for cs1 introductory programming courses in c. Tech. rep., University of Campinas, SP, Brasil, 2018.

[2] Caceffo, R., Wolfman, S., Booth, K. S., and Azevedo, R. Developing a computer science concept inventory for introductory programming. SIGCSE '16, ACM, pp. 364–369.

[3] Cook, D. A., Brydges, R., Ginsburg, S., and Hatala, R. A contemporary approach to validity arguments: A practical guide to Kane's framework. *Medical Education 49*, 6 (2015), 560–575.

[4] Crow, T., Luxton-Reilly, A., and Wuensche, B. Intelligent tutoring systems for programming education. *Proceedings of the 20th Australasian Computing Education Conference on - ACE '18* (2018), 53–62.

[5] Cunningham, K., Blanchard, S., Ericson, B., and Guzdial, M. Using Tracing and Sketching to Solve Programming Problems. In *ICER '17* (2017), ACM Press, pp. 164–172.

[6] Danielsiek, H., Toma, L., and Vahrenhold, J. An Instrument to Assess Self-Efficacy in Introductory Algorithms Courses. In *ICER* (New York, New York, USA, 2017), ACM Press, pp. 217–225.

[7] Decker, A. *HOW STUDENTS MEASURE UP: AN ASSESSMENT INSTRUMENT FOR INTRODUCTORY COMPUTER SCIENCE*. PhD thesis, 2007.

[8] Decker, A., and McGill, M. M. A Topical Review of Evaluation Instruments for Computing Education. In *SIGCSE* (2019), ACM Press, pp. 558–564.

[9] Deimel, L. E. J., and Makoid, L. Developing program reading comprehension tests for the Computer Science classroom. *Proc. IFIP TC 34th World Conf. on Computers in Education WCEE 85* (1985), 535–540.

[10] Dunn, K. E., and Mulvenon, S. W. A Critical Review of Research on Formative Assessment: The Limited Scientific Evidence of the Impact of Formative Assessment in Education. *Practical Assessment, Research & Evaluation 14*, 7 (2009).

[11] Duran, R., Rybicki, J.-M., Sorva, J., and Hellas, A. Exploring the Value of Student Self-Evaluation in Introductory Programming. In *ICER '19* (2019), ACM Press, pp. 121–130.

[12] Elliott Tew, A. *Assessing fundamental introductory computing concept knowledge in a language independent manner*. PhD thesis, 2010.

[13] Ericson, B. J., Foley, J. D., and Rick, J. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *ICER '18* (2018), ACM Press, pp. 60–68.

[14] Giordano, D., Maiorana, F., Csizmadia, A. P., Marsden, S., Riedesel, C., Mishra, S., and VinikienÄŮ, L. New Horizons in the Assessment of Computer Science at School and Beyond. In *Proceedings of the 2015 ITiCSE Working Group Reports* (2015), ACM Press, pp. 117–147.

[15] Gluga, R., Kay, J., Lister, R., Kleitman, S., and Lever, T. Coming to terms with Bloom : an online tutorial for teachers of programming fundamentals. *14th Australasian Computing Education Conference* (2012), 147–156.

[16] Hattie, J. Influences on student Learning. Tech. rep., 1999.

[17] Hattie, J., and Timperley, H. The Power of Feedback. *Review of Educational Research 77*, 1 (March 2007), 81–112.

[18] Jacobson, N. Using on-computer exams to ensure beginning students' programming competency. *ACM SIGCSE Bulletin 32*, 4 (2000), 53–56.

[19] Kane, M. J. Current Concerns in Validity Theory. *Language Learning 38*, 4 (2016), 319–342.

[20] Kane, M. T. Validating the Interpretations and Uses of Test Scores. *Journal of Educational Measurement 50*, 1 (mar 2013), 1–73.

[21] Kane, M. T. Validation as a Pragmatic, Scientific Activity. *Journal of Educational Measurement 50*, 1 (2013), 115–122.

[22] Kennedy, C., and Kraemer, E. T. What are they thinking?: Eliciting student reasoning about troublesome concepts in introductory computer science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (2018), Koli Calling '18, ACM, pp. 7:1–7:10.

[23] Kingston, N., and Nash, B. Formative assessment: A meta-analysis and a call for research. *Educational Measurement: Issues and Practice 30*, 4 (2011), 28–37.

[24] Lemos, R. S. Measuring programming language proficiency. *AEDS Journal 13*, 4 (1980), 261–273.

[25] Lister, R. Toward a Developmental Epistemology of Computer Programming. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education - WiPSCE '16* (2016), 5–16.

[26] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., and et al. *A Multi-national Study of Reading and Tracing Skills in Novice Programmers*. ITiCSE-WGR '04. ACM, 2004, pp. 119–150.

[27] Luxton-Reilly, A., and Petersen, A. The Compound Nature of Novice Programming Assessments. In *Proceedings of the Nineteenth Australasian Computing Education Conference on - ACE '17* (2017), ACM Press, pp. 26–35.

[28] Luxton-Reilly, A., Whalley, J., Becker, B. A., Cao, Y., McDermott, R., Mirolo, C., Mühling, A., Petersen, A., Sanders, K., and Simon. Developing Assessments to Determine Mastery of Programming Fundamentals. In *ITiCSE-WGR '17* (2017), ACM Press, pp. 47–69.

[29] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bull. 33*, 4 (Dec 2001), 125–180.

[30] Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., and Thomas, L. A cognitive approach to identifying measurable milestones for programming skill acquisition. *ITiCSE-WGR '06*, December 2006 (2006), 182.

[31] Messick, S. Validity of psychological assessment: Validation of inferences from persons' responses and performances as scientific inquiry into score meaning. *American Psychologist 50*, 9 (1995), 741–749.

[32] Moss, P. A., Girard, B. J., and Haniford, L. C. Validity in Educational Assessment. *Review of Research in Education 30* (2006), 109–162.

[33] Nelson, G. L., Xie, B., and Ko, A. J. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. *ICER* (2017).

[34] Parker, M. C., and Guzdial, M. Replication, validation, and use of a language independent CS1 knowledge assessment. *ICER* (2016), 93–101.

[35] Parker, M. C., Guzdial, M., and Engleman, S. *Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment*. ICER '16. ACM, 2016, pp. 93–101.

[36] Pea, R. D. Language-Independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research 2*, 1 (Feb. 1986), 25–36.

[37] Perkins, D., and Martin, F. Fragile Knowledge and Neglected Strategies in Novice Programmers. IR85-22. Tech. rep., 1985.

[38] Petersen, A., Craig, M., and Zingaro, D. Reviewing CS1 exam question content. In *SIGCSE '11* (2011), ACM Press, p. 631.

[39] Piech, C., and Gregg, C. BlueBook: A Computerized Replacement for Paper Tests in Computer Science. In *SIGCSE '18* (2018), ACM Press, pp. 562–567.

[40] Polikoff, M. S. Instructional sensitivity as a psychometric property of assessments. *Educational Measurement: Issues and Practice 29*, 4 (2010), 3–14.

[41] Porter, L., Zingaro, D., Liao, S. N., Taylor, C., Webb, K. C., Lee, C., and Clancy, M. BDSI: A Validated Concept Inventory for Basic Data Structures Leo. In *ICER '19* (2019), ACM Press, pp. 111–119.

[42] Prior, J. C. Online Assessment of SQL Query Formulation Skills. *Ace '03 20* (2003), 247–256.

[43] Riese, E. Students' Experience and Use of Assessment in an Online Introductory Programming Course. In *2017 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)* (apr 2017), IEEE, pp. 30–34.

[44] Sanders, K., Spacco, J., Ahmadzadeh, M., Clear, T., Edwards, S. H., Goldweber, M., Johnson, C., Lister, R., McCartney, R., and Patitsas, E. The Canterbury QuestionBank. In *ITiCSE -WGR '13* (2013), ACM Press, pp. 33–52.

[45] Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., and Paterson, J. H. An introduction to program comprehension for computer science educators. *ITiCSE-WGR '10* (2010), 65.

[46] Sheard, J., Souza, D. D., Klemperer, P., Porter, L., and Zingaro, D. Benchmarking Introductory Programming Exams: Some Preliminary Results. *ICER '16* (2016), 103–111.

[47] Simon, Sheard, J., D'Souza, D., Klemperer, P., Porter, L., Sorva, J., Stegeman, M., and Zingaro, D. Benchmarking Introductory Programming Exams. In *ITiCSE '16* (2016), vol. 159, ACM Press, pp. 154–159.

[48] Snow, E., Rutstein, D., Bienkowski, M., and Xu, Y. Principled Assessment of Student Learning in High School Computer Science. In *ICER '17* (2017), ACM Press, pp. 209–216.

[49] Syang, A., and Dale, N. B. Computerized adaptive testing in computer science: Assessing student programming abilities. In *Proceedings of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* (1993), SIGCSE '93, ACM, pp. 53–56.

[50] Taylor, C., Zingaro, D., Porter, L., Webb, K., Lee, C., and Clancy, M. Computer science concept inventories: past and future. *Computer Science Education 24*, 4 (2014), 253–276.

[51] Tew, A. E., and Guzdial, M. *Developing a Validated Assessment of Fundamental CS1 Concepts*. SIGCSE '10. ACM, 2010, pp. 97–101.

[52] Tew, A. E., and Guzdial, M. The fcs1: a language independent assessment of cs1 knowledge. In *SIGCSE 2011* (2011), ACM, pp. 111–116.

[53] Utting, I., Sorva, J., Wilusz, T., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersen, M., and Kolikant, Y. B.-D. A fresh look at novice programmers' performance and their teachers' expectations. *ITiCSE-WGR '13* (2013), 15–32.

[54] Xie, B., Davidson, M. J., Li, M., and Ko, A. J. An item response theory evaluation of a Language-Independent CS1 knowledge assessment. In *SIGCSE '19* (2019), ACM.

[55] Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., and Ko, A. J. A theory of instruction for introductory programming skills. *Computer Science Education* (2019), 1–49.

[56] Xie, B., Nelson, G. L., and Ko, A. J. An Explicit Strategy to Scaffold Novice Program Tracing. In *SIGCSE '18* (2018), ACM Press, pp. 344–349.

[57] Zingaro, D., and Petersen, A. Stepping Up to Integrative Questions on CS1 Exams. 253–258.

[58] Zur-Bargury, I., Pârv, B., and Lanzberg, D. A nationwide exam as a tool for improving a new curriculum. In *ITiCSE '18* (2013), ACM, pp. 267–272.