# A Blocks-based Language for Program Correctness Proofs

Peter-Michael Osera
*Department of Computer Science*
*Grinnell College*
Grinnell, USA
osera@cs.grinnell.edu

David G. Wonnacott
*Department of Computer Science*
*Haverford College*
Haverford, USA
davew@cs.haverford.edu

***Abstract*—While formal mathematical reasoning is the cornerstone of computer science, undergraduates often fail to appreciate the value of mathematical proof in their studies. To alleviate this problem, we propose a novel pedagogy uniting logical reasoning with proofs of program correctness along with a proof assistant, ORC$^2$A, that helps students author proofs in this domain. One of the defining features of ORC$^2$A is that it has a blocks-based surface language of proof to reduce friction when adopting the tool in the classroom. We report on the current progress on ORC$^2$A, in particular, its blocks-based interface, current design consideration, and our plans for evaluating the system.**

## I. Introduction

Mathematics sits at the foundation of computer science, providing practitioners with techniques to model and reason about programs. In particular, the study of mathematical proof is essential to algorithmic design and reasoning about program correctness in both formal and informal settings. However, computer science undergraduates often fail to appreciate the value of mathematical proof in their studies. In conversations with computer science educators over the years [1, 2], we have found that this problem manifests itself in two forms:

1) Students never truly understand the mechanics of proof and its similarities to the mechanics of programming.
2) Students believe that rigorous, formal reasoning is completely divorced from the more informal sorts of reasoning they carry out day-to-day when programming.

Previously, educators have taken two different approaches to solve these problems. The first approach involves developing pedagogies that approach proof in non-traditional ways, *e.g.*, strategies for developing proof [3] or choosing novel proof domains [4]. The second approach involves developing tools to help students author general mathematical proofs, *e.g.*, proof assistants [5] or specific sorts of proofs, *e.g.*, automata proofs [6]. However, few have tried to *combine* these approaches with the explicit purpose of helping undergraduates understand the purpose of mathematical reasoning in computer science.

In this work, we attempt to help undergraduates understand the value of mathematical proof in computer science by combining novel pedagogy with tool-based support. In particular, we are currently building a tool, ORC$^2$A[1] [7], to support the 3-2-1 pedagogical approach advocated by Dougherty and Wonnacott. 3-2-1 is a approach to introductory CS which introduces three programming paradigms (functional, imperative, and object-oriented) and two conceptual models of program execution in the context of a single programming language. This approach has been demonstrated to increase student understanding of the relevance of logic and proof in computer science at the introductory level by using program correctness as the domain and motivation for learning mathematical reasoning [8, 9].

Like traditional mathematical proof, proofs of program correctness are tedious and error-prone to construct and time consuming to grade. ORC$^2$A is a proof assistant built for undergraduates equipped with a proof language that allows students with little knowledge of formal reasoning to write proofs in this domain. However, initial testing with ORC$^2$A has shown that the textual nature of the proof language creates a steep learning curve to using ORC$^2$A, one that makes it difficult to integrate the tool into the classroom. To this end, we are currently developing a blocks-based surface language for ORC$^2$A that we hypothesize will reduce this barrier to adoption.

## II. A Brief Overview of ORC$^2$A

ORC$^2$A is an undergraduate-focused proof assistant that helps students construct proofs of program correctness. In contrast with industrial-strength proof assistants and theorem provers, ORC$^2$A strips away all unnecessary functionality and hides complexity so that undergraduates with little knowledge of formal logic can write correctness proofs. The tool currently supports pure, functional programs written in an appropriate subset of the Java programming language over integers and booleans. ORC$^2$A supports reasoning about these programs using a mixture

[1]Online Reasoning of Computational Correctness Application

of term-rewriting and Hoare-style logic with pre- and post-conditions. Behind the scenes, ORC$^2$A uses the Z3 theorem prover [10] to aid in reasoning about constraints over primitive values.

```
 1  claim forall n:int. pre: n >= 0 => lemma:
        factorial(n) >= 1 by
 2  factorial(n) --> [[ if (n < 1) { return 1; }
        else { return n*factorial(n-1); } ]] in
 3  case cond: n < 1 of
 4    true: // base case
 5      ... --> [[ return 1; ]] in
 6      ... --> 1 in
 7      => goal
 8  | false:
 9      ... --> [[ return n*factorial(n-1); ]] in
10      ... --> n*factorial(n-1) in
11      assume IH: ((n-1 < n) -> (n-1 >= 0) ->
            (fact(n-1) >= 1) by
12        show n-1 < n by
13          => goal
14        end
15        show n-1 >= 0 by
16          pre, notcond => goal
17        end
18      end in // closes the assume block
19      IH, notcond => goal
20    end
21  end
```

Fig. 1. Example ORC$^2$A proof of the claim $\forall n. n \geq 0 \rightarrow$ `factorial(n)` $\geq 1$.

In ORC$^2$A, the user writes a proof as the composition of a set of *proof strategies* that advances the proof goal forward to a final, proven state. Beginning with the pre-conditions of a given claim, the initial goal of an ORC$^2$A proof is the post-condition itself which is then manipulated via rewrites and conditional reasoning to arrive at a proposition that can be directly solved with Z3. Each proof strategy leaves zero or more or more proof obligations that must be solved using successive proof strategies on the updated goal.

Figure 1 gives an example ORC$^2$A proof of the claim that whenever the argument to the standard recursive implementation of `factorial` is non-negative, then the result is at least one. The paper proof follows by a straightforward induction on the input $n$, appealing to the induction hypothesis in the recursive case and the fact that multiplying two numbers that are at least one produces a number that is also at least one.

The corresponding ORC$^2$A proof requires the user to explicitly describe how `factorial` behaves to arrive at this conclusion. In ORC$^2$A, we first state our claim, realizing the pre- and post-conditions of our function as a single implication (line 1). We proceed by evaluating the call to `factorial(n)` to the body of the function (line 2). To make progress, we then perform conditional reasoning on `n` (line 3) which allows us to simplify the function call further into the conditional's if- and else-branches. In the if-branch (when `n` $< 1$), the function produces the value 1 and we are left with the proof obligation that $1 \geq 1$ which is trivially discharged.

In the else-branch (when `n` $\geq 1$), we must utilize our induction hypothesis (line 11) that states that `factorial(n-1)` $\geq 1$. To use this fact, we must prove two auxiliary conditions:

1) The recursive function call is made on a structurally decreasing argument, *i.e.*, `n-1` $<$ `n` (line 12).
2) The precondition to `factorial` is met for this recursive invocation (line 15). Proving this fact requires the use of two hypotheses: the precondition on the function, `n` $\geq 0$ (`pre`) and the negation of the conditional's guard, `n` $\geq 1$ (`notcond`) which we know holds by virtue of being in the else-branch of the conditional.

This allows us to assume that the inductive hypothesis holds, and we use it (`IH`) coupled with the fact that in the else-branch, we know that the conditional holds (`notcond`) to prove our goal.

The ORC$^2$A system checks that our proof is well-formed and logically consistent. Furthermore, ORC$^2$A also provides basic feedback in the form of counterexamples if we do not supply Z3 with the appropriate set of facts to solve a goal. For example, if on line 16, we did not specify to use the precondition `pre`, then attempting to solve the goal will fail and ORC$^2$A will report that the user needs to consider the case when `n` $= 0$.

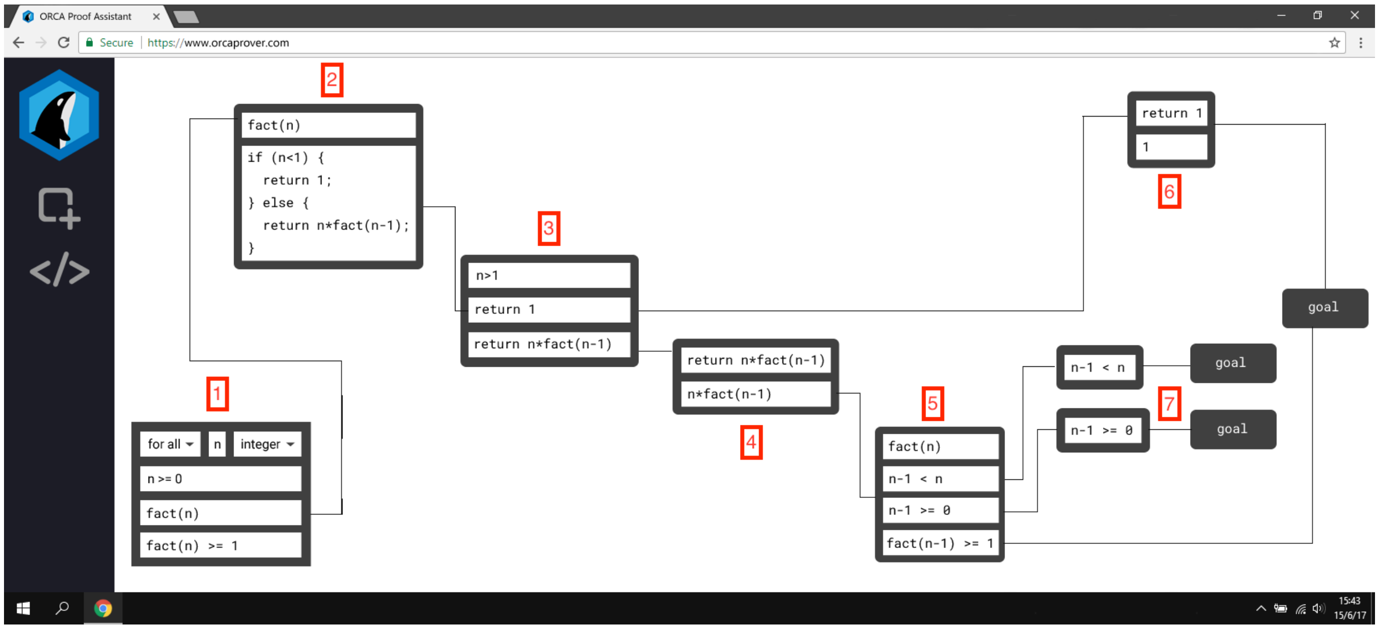
## III. Blocks-based ORC$^2$A

While ORC$^2$A met our initial goals for providing tooling support for proof pedgagogy focuses on program correctness, we immediately identified a pair of problems with our design.

1) While the ORC$^2$A proof language is pared down from a general-purpose proof assistant, it is still extremely verbose, making it unwieldy to use for undergraduates.
2) ORC$^2$A itself does not provide rich support for users to visualize the state of their proof during development.

The first problem is particularly worrisome because we envision using ORC$^2$A in contexts where learning about proof assistants and automated theorem proving is *not* among the primary learning goals. Indeed, in courses like discrete structures, we would like ORC$^2$A to be as simple to use as possible to ease adoption.

To remedy these problems, we are developing a blocks-based language for proof around ORC$^2$A to make it easier to develop formal proofs within the system and adopt the tool in other contexts. Figure 2 showcases a preliminary version of the ORC$^2$A blocks-based language. We map the small number of proof strategies supported by ORC$^2$A directly onto blocks, one block for each strategy. Each block constitutes a partial proof where connected blocks fulfill the proof obligations left by the original block.



```
fact(n)
if (n<1) {
  return 1;
} else {
  return n*fact(n-1);
}
```

```
n>1
return 1
return n*fact(n-1)
```

```
return n*fact(n-1)
n*fact(n-1)
```

```
for all  n  integer
n >= 0
fact(n)
fact(n) >= 1
```

```
fact(n)
n-1 < n
n-1 >= 0
fact(n-1) >= 1
```

```
n-1 < n
```

```
n-1 >= 0
```

```
return 1
1
```

Fig. 2. Mock-up of the factorial proof in ORC$^2$A with the prototype blocks system.

With blocks-based ORC$^2$A, a proof begins with a *claim block* that allows the user to specify what they are proving in terms of pre-conditions and post-conditions (block 1 in the diagram). Claim blocks also allow us to directly prove sub-obligations such as the pre-conditions of the induction hypothesis that we must show before utilizing its conclusion (block 5). *Rewrite blocks* (blocks 2 and 4), like the rewrite proof stategy, allow the user to rewrite portions of the goal through steps of evaluation. *Conditional blocks* (block 3) allow the user to perform conditional reasoning, leading to two proof obligations: one for when the condition is true and another for when the condition is false. And finally, *goal blocks* (blocks 6, and 7) allow the user to solve a proof goal by stating a series of hypotheses from the context that imply the correctness of the goal.

Even with this simple translation scheme, we gain several important benefits in moving to blocks-based ORC$^2$A. We conjecture that the lack of concrete syntax and additional help in creating well-formed proofs in terms of graphical support will reduce the learning barrier to using ORC$^2$A. Furthermore, a nice side-effect of transitioning to blocks is that the tree-like nature of proof is made explicit with this scheme. We hope that easily authoring out a proof in this manner will help students understand the general structure of proofs and logical reasoning.

In conjunction with the blocks language, we are also developing a visualization to allow the user to inspect the proof state at any step of the proof. Figure 3 gives an example of a simple, textual-based version of this visualization. The proof state consists of the proof goal that the user must prove as well as the set of hypotheses available to the user in proving that goal. This visual pane is analogous to Scratch's animation pane which allows user

```
Hypotheses:
  pre: n >= 0
  notcond: !(n < 1)
  IH: fact(n-1) >= 1

Goal:
  n*factorial(n-1) >= 1
```

Fig. 3. A mock-up of a simple ORC$^2$A proof state visualization. This particular proof state occurs after proving the pre-conditions of the recursive call (line 19 of the paper proof, block 5 of the blocks-based proof).

to see their programs execute in real-time alongside their code. We can think of an ORC$^2$A proof as a program that manipulates the state of a proof in a similar way.

## IV. Design Considerations

Through the development of our blocks-based ORC$^2$A implementation, we have encountered a number of design considerations that seem unique to our particular context of applying blocks-based language design towards education-focused proof assistants.

*a) Information Display:* Like a conventional computer program, much of the difficulty of authoring a proof is keeping in mind the state that the proof is manipulating. Proof assistants help us manage this complexity by tracking and reporting the state. However, in an educational context, we may decide that giving the user such information is not conducive to helping them learn about the structure and semantics of proof. For example, consider the induction hypothesis of an inductive proof. A proof assistant like ORC$^2$A can generate induction hypotheses automatically, but it may be an explicit learning goal of the instructor to have students derive these hypotheses on their own. In this case, it would be better for the system to not give this information to the user but instead require that the user provide a potential induction hypothesis and the system check that it is correct before being introduced as a valid hypothesis.

Unlike the programming domain where giving more information to the user is generally better, in the pedagogical proof domain, we must determine for every piece of information whether that information should be given to the user, provided by the user and checked by ORC$^2$A, or not tracked at all.

*b) Automation and Formality:* Building upon the previous point, industrial-strength proof assistants make heavy use of automated theorem proving to make proofs more manageable for the user. However, like overdisplaying information, performing too many steps of proof on behalf of the user may be contrary to the goals of the instructor. For example, many correctness proofs of simple programs follow by straightforward evaluation of the program with a case analysis. But we might want a student to demonstrate that they understand how the steps of simplification proceed or what cases they are analyzing rather than having the system fill in these holes for them. Identifying the balance between automated and manual-but-checked reasoning will be important in building ORC$^2$A to maintain both usability and pedagogical usefulness.

In its current incarnation, ORC$^2$A takes a particular stand on this topic with its use of Z3. ORC$^2$A requires that the user specify how a program evalautes with the rewrite proof strategy/block but leaves equational reasoning to Z3. However, the user must still specify to ORC$^2$A which hypotheses will be given to Z3 to attempt to prove the goal. As development of ORC$^2$A continues, we expect that we may need to refine how much automation and information the system provides as we gain more experience using the tool in the classroom.

## V. Deployment and Evaluation Plans

As of summer 2017, the blocks-based version of ORC$^2$A is in active development. We expect to begin preliminary testing and data collection of the ORC$^2$A system in our own classrooms in the fall of 2017:

1) Introduction to Computer Science (CS 205) at Haverford College, a CS1 course for majors with programming experience.
2) Algorithms and Object-oriented Design (CSC 207) and Discrete Structures (CSC 208) at Grinnell College at Grinnell College, mid-level courses in data structures and discrete mathematics.

At Haverford, ORC$^2$A will be used extensively throughout the semester as program reasoning is one of the core components of the course. In contrast, ORC$^2$A will be used in smaller modules at Grinnell that focus on program correctness and mathematical logic.

From this initial deployment, we wish to see if ORC$^2$A is able to address any of the concerns about proof pedagogy posed in the introduction:

- Does ORC$^2$A significantly change the fraction of students who can produce a correct and complete proof?
- Does ORC$^2$A significantly change the amount of time needed for students to produce a correct proof?
- Does ORC$^2$A significantly change students' attitudes towards the relevance of mathematical proof in computer science?

## References

[1] P.-M. Osera and B. Yorgey, "Making induction meaningful, recursively (abstract only)," in *Symposium on Computer Science Education (SIGCSE)*, 2014.

[2] J. P. Dougherty, J. E. Hollingsworth, J. Krone, and M. Sitaraman, "Mathematical reasoning in computing education: Connecting math we teach with writing correct programs (abstract only)," in *Symposium on Computer Science Education (SIGCSE)*, 2016.

[3] J. L. Cordova, "Mathematical proofs as graph search problems in theory courses," in *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '99. New York, NY, USA: ACM, 1999, pp. 110–113.

[4] C. Eastlund and M. Felleisen, "Automatic verification for interactive graphical programs," in *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, ser. ACL2 '09. New York, NY, USA: ACM, 2009, pp. 33–41.

[5] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey, *Software Foundations*. Electronic textbook, 2015.

[6] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan, "Automated grading of dfa constructions," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, pp. 1976–1982.

[7] J. Chen, M. Gopalaswamy, P. Pradhan, S. Son, and P.-M. Osera, "ORC$^2$A: A proof assistant for undergraduate education," in *Symposium on Computer Science Education (SIGCSE)*, 2017.

[8] J. P. Dougherty and D. G. Wonnacott, "Use and assessment of a rigorous approach to cs1," in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '05. New York, NY, USA: ACM, 2005, pp. 251–255.

[9] D. Wonnacott, *From Vision to Execution: The Creative Process in Computer Science*. Raleigh, North Carolina, USA: lulu.com, 2016, lulu.com ID 1094615.

[10] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.