Language-Level Support for Co-Creative Programming

Chris Martens
Computer Science Department, North Carolina State University
martens@csc.ncsu.edu

Programming languages research to date has largely been focused on *programs* as artifacts, answering questions about what constitutes meaningful programs, what can we deduce about their execution paths, how can we compile them to programs in other languages. In this paper we argue for a research program more concerned with the *process of constructing programs* as a human activity, taking a broad view of the purposes this activity can serve and the ways that programming language research can and should investigate support for it. However, programming as an activity can serve a wide variety of important purposes, including:

- To create a program artifact that fulfills a set of design requirements;
- To learn how to program, perhaps for the first time or perhaps in a new language;
- As a tool for thought: to define the scope of a problem formally, explore alternative definitions and representations, or simulate hypothetical situations;
- To engage with a creative process by relinquishing aesthetic control to an algorithm;
- As a form of play, indulging one's curiosity or interest in feedback mechanisms.

How can programming languages best support these diverse activities? In this talk, we will examine this issue through the lens of the interactive digital games and arts community. In particular, we focus on this community's use of *generative methods*—techniques for automating and augmenting certain aspects of the creative process. This field includes *generative art*, which uses directed randomness, search, and production grammars, and other techniques to generate drawings, music, 3D models, stories, and other artifacts. It also includes *procedural content generation* (PCG), a term more often used in video game development to refer to generating levels, graphical and animation effects, virtual 3D environments, and progression structures within a game world.

Generative methods are often used *as part of a creative process*: leaving certain design decisions up to an algorithm, rule, or random process gives artists a way to expand their creative horizons and break free of patterns unknowingly developed by their human-controlled creative instincts. This desire for feedback loops between human and digital sources of creativity leads to an interest in *co-creative workflows* in which a designer and a digital tool both propose and judge edits to a work-in-progress, such as a possibility space of generated content (or the program that generates it) (see Figure 1).

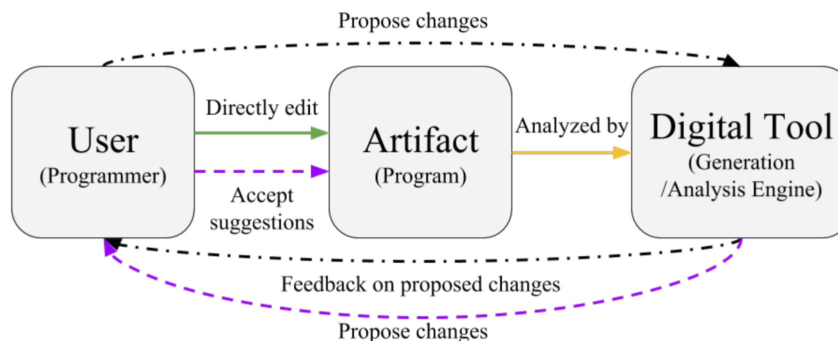
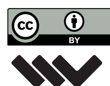


Figure 1 Architecture for a co-creative system.


© Author: Please provide a copyright holder;
licensed under Creative Commons License CC-BY
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

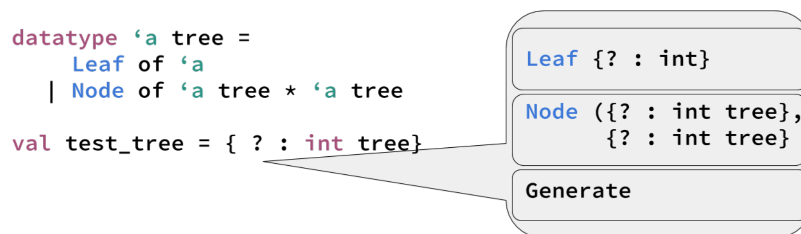


Figure 2 A mock-up interface for a type-driven program construction aid.

A number of ideas in generative methods connect well with active research agendas in programming languages, such as program synthesis, unifying logic programming and functional programming idioms, and dependent type systems. For instance, program synthesis, when thought of as a tool for generating programs that fit a given specification (e.g., for a function, a collection of input-output examples or logical formula relating inputs to outputs), is a very general generative process whose integration into human workflows must contend with all the same questions as PCG: how can we reason about the relationship between specifications and synthesized solutions? What is the possibility space of solutions for a given specification, and if there are many solutions, how is the space navigated, presented to the user, or refined? Likewise, programming language research has identified a need for reasoning about partial or incomplete programs, as, for example, the vision statement for the *Hazel* project [3] points out. The construct of *typed holes* allows the programming language definition (and type system) to support the program editing process: an editor can guide a programmer through the possibility space of expressions that match the type of the hole (see Figure 2). With richer type systems, more constraints can be placed on the expression possibility space and more guidance offered in the language editing environment, as seen in Agda [1]'s typed holes that generate case-branches-with-holes from pattern-matched data. Tools like Haskell QuickCheck [2], which generates random test cases for Haskell functions, demonstrate the need for generativity for practical purposes.

In this talk, we will bring together insights in programming languages and generative methods to propose a future generation of tools that support the diverse purposes of programming. We outline a research program for generative programming, grounded in a foundation of *deductive grammars* that exploits the proofs-as-programs correspondence to combines execution-as-reduction (functional programming) with execution-as-deduction (logic programming). By uniting ideas in these disparate fields, we hope to have the long-term impact of empowering future generations of thinkers and creators with computational thinking, in service of creativity, education, play, and personal growth.

References

1 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

2 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

3 Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A Hammer. Toward semantic foundations for program editors. *arXiv preprint arXiv:1703.08694*, 2017.