Programming as collaborative reference (full presentation)

Oleg Kiselyov and Chung-chieh Shan

We argue that programming-language theory should face the pragmatic fact that humans develop software by interacting with computers in real time. This interaction not only relies on but also bears on the core design and tools of programming languages, so it should not be left to Eclipse plug-ins and StackOverflow. We further illustrate how this interaction could be improved by drawing from existing research on natural-language dialogue, specifically on *collaborative reference*.

The users and implementations of modern programming languages are stymied by a communication bottleneck: Programs are so long that the need to express them in full detail imposes a heavy cognitive and physical burden even if one already understands the exact algorithm intended. Whenever you're programming and struggle to enter an expression even though it would be obvious to another programmer looking over your shoulder what you mean, that's an instance of this bottleneck in the human-to-computer direction.

It would be excusable if the reason the program we want takes n bits to express is that there are $2^n - 1$ other programs that we might want. But the vast majority of well-formed programs in today's programming languages are undesirable, and many desirable programs are equivalent. If only we can just input a desired program by entering the number of lexicographically preceding desired programs! But that's intractable for humans and computers alike. Abstractions and types may reduce this communication burden somewhat, but the boilerplate necessary to access highly parametrized abstractions and the annotations necessary to help type inference along contribute to the problem themselves.

We face a similar problem when talking to fellow human beings. Most of the meanings we intend to convey using natural-language utterances would be excruciating to spell out fully in any language. For example, pronouns (such as "him") and descriptions ("the president") are on one hand much more ambiguous than proper names ("Barack Obama") and explicit variable names ("x"), but on the other hand more concise as well as easier to process for speakers and hearers alike. To take another example, the familiar question "Can everyone hear me?" is typically spoken to ask about everyone in the room rather than everyone in the world [4]. It scantly helps to rephrase the question to "Can everyone in the room hear me?"—Which room? (Do you mean the city of Philadelphia or the movie Philadelphia?)

Despite such odds, humans manage to communicate all the time—whether to name a room, a proof, or a program. Key to this feat are

- our use of context, which narrows down the meanings that make sense to convey, and
- our exchange of feedback, which makes it unnecessary to get an utterance right the first time or to ever give a complete explicit description.

Programming languages today already use some context to lessen the burden of communication; examples include scope, type inference, and overloading resolution. Development environments today also provide feedback in rudimentary forms such as continuous compilation and identifier completion. Comparing these facilities against human communication shows that it is possible for us to program more concisely, especially if the language is designed with context and feedback in mind. And program more concisely we must, as we build larger systems and check stronger properties, which might be expressed today in fancy type systems that require many (often "obvious") annotations.

For example, whenever you want to talk to me about something or somebody, maybe someone you spot across the stadium, we usually manage to agree on the same person pretty quickly, even though we haven't agreed on a numbering of all objects ahead of time. The psychologists Clark and Wilkes-Gibbs [2] point out that this task is very interactive—not at all like taking one shot to write out a clear noun phrase.

A: the guy reading with, holding his book to the left.
B: Okay, kind of standing up?
A: Yeah.
B: Okay.

Clark and Wilkes-Gibbs study this *collaborative reference* task by asking experiment subjects to perform a matching task in pairs. Part of their study classified the moves people tend to make and identified how they signal and anticipate these moves.

Programming is a human-computer collaboration whose goal is also reference. We are not alone to suggest easing this daunting goal by making the object of reference not just the raw code but also what it does and why it works. A flexible interaction can take advantage of how few programs make sense to relieve humans from reading and writing most boilerplate and annotations. Perhaps, what a program does would be expressed roughly by a type, and why it works would be expressed by a type derivation; like Conor McBride, we "think of types as warping our gravity, so that the direction we need to travel becomes 'downhill'."

What's tricky is that the intended program is seldom the exact minimum. For example, given a signature like Haskell's `Show` or `Ord`, sometimes the implementation you want is not the default instance that type classes give you,

but almost! So close, yet so far to have to compose all the functor invocations by hand! So the ideal trip to the desired program is like

- Marble Madness;
- filling in holes in Agda;
- IntelliSense, not only constrained by types, but also liberated by completing complex expressions;
- input method editors; and
- pair programming or code review, which goes faster and more smoothly as participants develop a shared vocabulary like between Clark and Wilkes-Gibbs's director and matcher.

It is crucial to allow decisions to be made (facets to be specified) in many different ways during the collaboration.

That's not all. We wouldn't be here if we just wanted to suggest that HCI researchers and IDE builders take more context into account in the interfaces they are already building. Neither do we want to program computers using an existing natural language [1].

It turns out that implementing collaborative reference, whether in natural languages or in programming languages, requires some tools familiar to you such as semantics and logic:

- Participants in collaborative reference maintain their discourse context by updating a virtual scoreboard with constraints, or logical statements about the intended referent.
- The task often spawns sub/meta-tasks to clear up collateral ambiguity, whether about a constituent subroutine, a constraint predicate, or a discourse move.
- Previously established referents are added to a vocabulary that, unsurprisingly, looks a lot like a type environment.
- The pervasive uncertainty can be expressed using nondeterministic programming and managed using compositional heuristics for search through probability distributions with the programmer's guidance.

DeVault's COREF dialogue agent [3] can play both the director role and the matcher role in a task like Clark and Wilkes-Gibbs's experiment task. To show concretely that the approach is viable, we will play a one-minute video of COREF in action (`http://www.cs.rutgers.edu/~ccshan/devault-contribution/example-coref-matches.mp4`).

An initial attempt at programming by collaborative reference may target the problem of (overlapping) overloading resolution or (undecidable) type inference. The key is to identify a space of possible discourse moves, including the space of programs that make sense and a variety of strategies that both the director and the matcher can use to describe a candidate they have in mind and distinguish it telegraphically from competitors in play. For example, instead of insisting that all overloading be unambiguous, a programming language can resolve overloading interactively if there is enough doubt as to which referent the programmer intends. The referent that results from this collaboration would then become part of the program text (in particular, part of the *indexical* content [5] that any copy-and-paste operation would operate on), so meta-theorems such as type safety would still hold.

---

[1] Allen, James F., Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary D. Swift, and William Taysom. 2007. PLOW: A collaborative task learning agent. In *AAAI-2007: Proceedings of the 22nd national conference on artificial intelligence*, 1514–1519. The American Association for Artificial Intelligence, Menlo Park, CA: AAAI Press.

[2] Clark, Herbert H., and Deanna Wilkes-Gibbs. 1986. Referring as a collaborative process. *Cognition* 22(1):1–39.

[3] DeVault, David. 2008. Contribution tracking: Participating in task-oriented dialogue under uncertainty. Ph.D. thesis, Rutgers University.

[4] von Fintel, Kai. 1994. Restrictions on quantifier domains. Ph.D. thesis, Department of Linguistics, University of Massachusetts.

[5] Kaplan, David. 1989. Demonstratives: An essay on the semantics, logic, metaphysics, and epistemology of demonstratives and other indexicals. In *Themes from Kaplan*, ed. Joseph Almog, John Perry, and Howard Wettstein, chap. 17, 481–563. New York: Oxford University Press.