Concretising Computational Abstractions: What works, what doesn't, and what is lost

Ken Kahn

Abstract. In order to create programs in a programming language, one needs to understand the abstractions of the computation model underlying the language. How can one program without understanding the underlying conceptual building blocks such as variables, data structures, procedures, recursion, and the like? Since 1992, I have been developing a programming language whose primitives are not abstract, but instead are familiar objects that exist in an animated game-like virtual world. The primitives of this language, ToonTalk (www.toontalk.com), are robots, boxes, birds, nests, number and text pads, magic wands, trucks, and bombs rather than the corresponding abstractions of methods, arrays, send and receive capabilities, numbers, strings, and ways of expressing copying, process spawing and termination. ToonTalk is a general-purpose concurrent programming language that even very young children are able to program. They do so because they understand the behaviour of the basic elements, e.g. that if you or a robot you've trained gives a bird something, she'll take it to her nest.

Children, in some cases on their own, have learned to build ToonTalk programs because each of the primitives can be understood metaphorically and concretely. Furthermore, there is a sense that there is no syntax and hence the mechanics of programming is trivial to master. But there is more to programming than mastering the primitives and the mechanics: planning and design remain. These are the remaining hurdles to making programming "child's play". ToonTalk program fragments cannot be inspected; they can only be observed as they execute. You can't see at a glance what a program does; you can only see a program unfold over time. Iterative design can be difficult as it often entails rebuilding many program fragments.

Despite ten years of usage by children, many questions remain. Even if the concrete analogs preserve all the expressive power of the computational abstractions they have replaced, has something been lost? Do abstractions enable a kind of thinking that the corresponding concretisations don't?