Errata and Addenda for
“On Abstraction and Informatics” [17]

Tom Verhoeff

October 2011, August 2013, October 2014, April 2016, June 2019

p.7, below (5) The definition of morphism composition $\circ$ should read

$$(f \circ g)(x) \quad = \quad f(g(x))$$

Note that the definition as was given (i.e., $(g \circ f)(x) = f(g(x))$), is indeed used by some mathematicians. Also see [6]. Traditional notation for functions is inherently confusing, because the implicit “directions” in the signature descriptor $f : A \to B$ (left to right) and function application $f(x)$ (argument comes from the right) are in conflict.

For signature composition, it makes sense to write the composition of $f : A \to B$ with $g : B \to C$ as $f \circ g : A \to C$, preserving the order ($A \xrightarrow{f} B \xrightarrow{g} C$; and taking for granted that then $(f \circ g)(x) = g(f(x))$, where the order switches).

For function application, it makes sense to write the composition of these same f and g as $(g \circ f)(x) = g(f(x))$, maintaining the order of the functions on the left and right, but their signatures do “match” in the opposite order ($A \xrightarrow{f} B \xrightarrow{g} C$).

All this can easily be resolved by writing function application in the reverse order: $x.f = f(x)$. In that case, we could define $x.(f \circ g) = (x.f).g$, and everything would be “in order”. It seems less “natural” to reverse the order in the signature, and write $f : B \leftarrow A$. However, in some programming languages (C, C++, Java), that order is used in definitions of functions/methods: `double sqrt(double x)`.

p.10, line 3 below §6 Change ‘Others textbooks’ into ‘Other textbooks’.

p.6, recursive definitions It is worth-while to mention that a computational formalism can be universal even if it lacks both iteration and recursion. Also see [18]. Examples are Untyped Lambda Calculus and Combinatory Logic, but also in JavaScript, iteration and recursion are not needed. These formalisms offer mechanisms to *abstract from functions* and do *self application*, to obtain the same looping effect as iteration or recursion. For instance, the recursive definition of the factorial function

$$\mathit{fac}(n) \quad := \quad 1 \textbf{ if } n = 0 \textbf{ else } n \times \mathit{fac}(n)$$

can be de-recursified by abstracting from the function *fac* in the function body, replacing it by a parameter g, which itself is a function:

$$\mathit{FAC}(g)(n) \quad := \quad 1 \textbf{ if } n = 0 \textbf{ else } n \times g(n)$$

Note that FAC is neither recursive, nor iterative, and that *fac* is a *fixed point* of FAC:

$$\mathit{FAC}(\mathit{fac}) \quad = \quad \mathit{fac}$$

However, to calculate $\mathit{fac}(n)$, the g in $(\mathit{FAC}(g))(n)$ need not be (a completely defined) *fac*, but could be an 'approximation' of *fac* that only works correctly for arguments less than n. In particular, for $n = 0$, we could take any g, because g is then basically ignored.

Generalizing parameter g further to g', so that g' takes an additional parameter of the same type as g', we can define

$$\mathit{FAC}'(g')(n) \quad := \quad 1 \textbf{ if } n = 0 \textbf{ else } n \times g'(g')(n)$$

Note the self application of g' to g'. We now have (by induction on n)

$$\mathit{fac}(n) \quad = \quad \mathit{FAC}'(\mathit{FAC}')(n)$$

or more concisely

$$\mathit{fac} \quad = \quad \mathit{FAC}'(\mathit{FAC}')$$

This de-recursification can even be mechanized through a fixed-point combinator Y with the property

$$Y(F) \quad = \quad F(Y(F))$$

i.e., $Y(F)$ is a fixed point of F. We then have (by induction on n)

$$Y(FAC)(n) \quad = \quad fac(n)$$

Thus, we can define

$$fac \quad = \quad Y(FAC)$$

All we now need is a non-recursive, non-iterative definition of Y. This exists for the formalisms mentioned above (see [14, 19]). The key is self-application, as in the (non-recursive, non-iterative) definition $\omega(f) := f(f)$. Note that $\omega(\omega)$ does not terminate, in spite of the absence of recursion and iteration.

p.5, "The biggest danger of axiomatic definitions is *inconsistency*" Inconsistency can be the result of *over-specification*, by imposing too many constraints, which turn out to be contradictory. Another danger is *under-specification*, by imposing too few constraints, thereby allowing unintended 'solutions' to the axioms.

Quotes From [5, p.4]:

> The result of being more abstract is not being more vague, on the contrary: the purpose of abstraction is the creation of a new semantic level at which one can again be absolutely precise, but with less commitment. The virtue of the new theory is that one can work in it, unburdened by the irrelevant details of the model that inspired it. Experience has shown that people's first confrontation with mathematical abstraction is often emotionally disturbing; the rest of the educational process hardly teaches the potential intellectual advantages of ignoring available knowledge and the manifest freedom of creating one's own universe of discourse could very well be frightening.

References Inexcusably missed references: [4, 10, 11, 13, 15]. Possibly excusably missed references: [1, 2, 3, 7, 9], [8, Ch.2]; newer material: [12] (especially Ch.1, "The Many Faces of Complexity in Software Design" by José Luiz Fiadeiro), [16].

References

[1] Abbott, Russ. "The Reductionist Blind Spot", *Complexity*, **14**(5):10-22, May 2009.

[2] A.V. Aho, J.D. Ullman. Chapter 1, "Computer Science: The Mechanization of Abstraction", *Foundations of Computer Science: C Edition*. W.H. Freeman, 1992. i.stanford.edu/~ullman/focs.html (accessed 05-Sep-2012).

[3] Grady Booch. *Object-Oriented Design with Applications*. Benjamin-Cummings, Redwood City, 1990.

[4] T. Colburn, G. Shute. "Abstraction in Computer Science", *Minds and Machines*, **17**(2):169–184 (2007). DOI: 10.1007/s11023-007-9061-7

[5] E. W. Dijkstra. *The unification of three calculi*. EWD1123, 10 June 1992.

[6] Functional Composition. *Wikipedia*. en.wikipedia.org/wiki/Function_composition (accessed 03-Oct-2011).

[7] P. Guo. *What is Computer Science? Efficiently Implementing Automated Abstractions*. Feb. 2010. www.pgbovine.net/what-is-computer-science.htm (accessed 05-Sep-2012).

[8] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004. softwarefactories.com (accessed 27-Apr-2016).

[9] M. Hailperin, B. Kaiser, K. Knight. *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. Brooks/Cole, 1999. https://gustavus.edu/+max/concrete-abstractions.html (accessed 05-Sep-2012).

[10] O. Hazzan. "Abstraction in Computer Science & Software Engineering: A Pedagogical Perspective", *System Design Frontier* (Dec. 2006) edu.technion.ac.il/Faculty/OritH/HomePage/FrontierColumns/OritHazzan_SystemDesigFrontier_Column5.pdf (accessed 05-Sep-2012).

[11] O. Hazzan, J. Kramer (Eds.). *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering*, Leipzig, Germany, ACM, 2008.

[12] Mike Hinchey, Lorcan Coyle (Eds.). *Conquering Complexity.* Springer, 2012.

[13] B. Liskov. *The Power of Abstraction.* ACM A.M. Turing Award 2008 Lecture. Nov. 2009.

[14] Matt Might. *Fixed-point combinators in JavaScript: Memoizing recursive functions.* `http://matt.might.net/articles/implementation-of-recursive-fixed-point-y-combinator-in-javascript-for-memoization/` (accessed 23-Aug-2013).

[15] Giuseppe Primiero. "Proceeding in Abstraction: From Concepts to Types and the Recent Perspective on Information". *History and Philosophy of Logic*, **30**(3):257–282 (2009). DOI: `https://doi.org/10.1080/01445340902872630`

[16] Alexander A. Stepanov, Daniel E. Rose. *From Mathematics to Generic Programming.* Addison-Wesly, 2015. Blurb: "In this substantive yet accessible book, pioneering software designer Alexander Stepanov and his colleague Daniel Rose illuminate the principles of generic programming and the mathematical concept of abstraction on which it is based, helping you write code that is both simpler and more powerful."

[17] T. Verhoeff. "On Abstraction and Informatics", Proceedings of ISSEP 2011, Bratislava, Slovakia.

[18] Tom Verhoeff. "A Master Class on Recursion". in *Adventures Between Lower Bounds and Higher Altitudes.* Lecture Notes in Computer Science Vol.11011, pp.610–633, Springer, 2018. DOI: `https://doi.org/10.1007/978-3-319-98355-4_35`

[19] Wikipedia. *Fixed-point Combinator.* `http://en.wikipedia.org/wiki/Fixed-point_combinator` (accessed 23-Aug-2013).