Imperative Functional Programming

Uday S. Reddy
Department of Computer Science
The University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
⟨*reddy@cs.uiuc.edu*⟩

Our intuitive idea of a function is that it describes the *dependence* of one quantity upon another. There is no condition on what kind of quantities are involved. They could be integers, Turing machines, binary search trees, window hierarchies, digital circuits or whatever. The possibilities are endless.

The idea of functional programming is that functions are described *applicatively*, by plugging together other functions. This recursive dependence of functions on other functions bottoms out with the primitive operators. If the primitive operators are "effective," i.e., they can be used to calculate, compute, construct or build one quantity from another, then all functions expressed applicatively are similarly effective.

Imperative functional programming is the application of these ideas to imperative computations. Quantities here are imperative computations and functions denote dependences between imperative computations. Effectiveness of a function means that a computation produced from it can be effectively carried out whenever the input computations can be effectively carried out.

It is often felt that imperative computations and functional programming are in conflict. Adding imperative computations to a functional programming language is found to destroy its "referential transparency." See, for example, [Abelson *et al.*, 1985, Sec. 3.1] for a discussion of the issue. However, in languages where such issues arise, imperative computations are thrown together by extending or modifying functional computation. The idea that functions denote *dependences* is lost in the process. In contrast, imperative functional programming does not involve altering functional computation in any way. It merely applies it to a new context, that of imperative compu-

$(a+b)+c$	$=$	$a+(b+c)$		$(a;b);c$	$=$	$a;(b;c)$
$a+0$	$=$	a		$a;\mathbf{skip}$	$=$	a
$0+a$	$=$	a		$\mathbf{skip};a$	$=$	a
$a+b$	$=$	$b+a$				

Table 1: Equational properties of the primitive operators

tations.

The imperative base

The basic imperative computations are familiar from vanilla imperative programming. They are

- *expressions*, which read the state and compute a value ("state readers"), and
- *commands*, which modify the state ("state transformers").

As an example of a primitive operator on expressions, consider

$$+ : \mathbf{exp} \times \mathbf{exp} \to \mathbf{exp}$$

For any two state readers a and b, $(a + b)$ is the state reader that uses a and b to read two numbers from the state and calculates their sum. As an example of a primitive operator on commands, consider

$$; : \mathbf{comm} \times \mathbf{comm} \to \mathbf{comm}$$

For any two state transformers a and b, $(a; b)$ is the state transformer that does the state transformation of a followed by the state transformation of b.

The key point is that "+" and ";" are *functions*. They satisfy the standard mathematical properties we expect of functions. One can write terms over them and assert certain terms to be equal. Such equations are either true or false *independent* of any context in which the terms occur. See Table 1 for a sample of such equations.

Computations of higher types

Imperative functional programming is not restricted to basic computations along. The standard mechanisms of functional programming (and type theory) give us higher-type computations via product types, sum types, function types and recursive types. Thus, we envisage a type system with types of the form:

$$t \quad ::= \quad a \mid \mathbf{exp} \mid \mathbf{comm} \mid t_1 \times t_2 \mid \mathbf{unit} \mid t_1 + t_2 \mid t_1 \to t_2 \mid \mu a.\, t'$$

where a stands for type variables and $\mu a.\, t'$ denotes recursive type definition.

Product types give rise to a notion of *objects*. For example, the type $\mathbf{exp} \times \mathbf{comm}$ denotes pairs that have an expression component and a command component. But, both the components can potentially act on the same shared state. Therefore, we think of such values as "objects" with an internal state and operations that act on the state. For instance, the function

$$\begin{aligned} &\text{mkcounter} : \mathbf{var} \to \mathbf{exp} \times \mathbf{comm} \\ &\text{mkcounter}\ v = (\text{get}(v),\ v := \text{get}(v) + 1) \end{aligned}$$

produces a counter object with two methods: a state reader method that reads the value of the counter and a state transformer method that increments the counter. Both the methods act on the same shared state (represented by the variable v).

Values of function types $t_1 \to t_2$ denote *object transformers* that transform objects of type t_1 to objects of type t_2. In general, such functions can have their own internal state that they use to carry out the transformation. The result of a function will be an object that acts on both the state of the function as well as the state of the argument. Function application is then a form of *concurrent composition*. The result object is obtained by composing the function object and the argument object and letting them *interact*. An example is the function $\mathit{add\_scrollbar} : \mathit{window} \to \mathit{window}$ that adds a scrollbar to a window. The result object (window with the scrollbar) incorporates the state of the original window as well as that of the scrollbar.

Beta-equivalence and other equivalences that one expects in a functional language hold here. In fact, it has been proved that all the observational equivalences that hold in the sublanguage without **comm** (a traditionally functional language) continue to hold in the full language [O'Hearn,]. So, we have here a language that is both functional and imperative at the same time. There is no conflict between the two.

The language considered above is a fragment of the Idealized Algol of Reynolds [1981], discussed in detail by Tennent [1988, 1991]. More sophisticated type systems and languages, based on similar principles, may be found in [Reynolds, 1988, Swarup *et al.*, 1991, Peyton Jones and Launchbury, 1995]. Also closely related are the Moggi's [1991] monad-based computational metalanguage and its adaptation by Wadler [1992] for structuring functional programs. The fact that higher-type computations denote objects with hidden states is modelled semantically in [O'Hearn and Tennent, 1995] and [Reddy, 1996] (more explicitly in the latter).

The case for imperative functional programming

Imperative and object-oriented concepts are ubiquitous in computing. The principles of functional programming can help us get a handle on them. By building functional languages on the base of imperative features, we combine the expressiveness and usefulness of imperative programming with the elegant reasoning principles of functional programming. The terms of the resulting languages are free of side effects, carry clear mathematical meaning and satisfy mathematical laws. Such laws can be used for program transformations and proving program equivalences. The mathematical meaning associated with programs helps us write equational (algebraic) specifications for objects. The laws allow us to derive consequences of such specifications. All the mathematical apparatus that one takes for granted in functional programming becomes available.

The reasoning principles of functional programming are too useful to be relegated to functional programming. Imperative programming is too important to be ignored. A combination of the two along the lines outlined here enriches both the paradigms and forms a fruitful direction of research in programming languages.

References

[Abelson *et al.*, 1985] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Moggi, 1991] E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.

[O'Hearn,] P. W. O'Hearn. Note on Algol and conservatively extending functional programming. *J. Functional Program.* (to appear).

[O'Hearn and Tennent, 1995] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3), 1995.

[Peyton Jones and Launchbury, 1995] S. L. Peyton Jones and J. Launchbury. State in Haskell. *J. Lisp and Symbolic Comput.*, 8(4):293–341, 1995.

[Reddy, 1996] U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation*, 9:7–76, 1996.

[Reynolds, 1981] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Reynolds, 1988] J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie-Mellon University, June 1988. URL — ftp://e.ergo.cs.cmu.edu/forsytheintro.

[Swarup *et al.*, 1991] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, volume 523 of *LNCS*, pages 192–214. Springer-Verlag, 1991.

[Tennent, 1988] R. D. Tennent. Denotational semantics of Algol-like languages. Internal Tech. Report 88-IR-02, Queen's University, September 1988. (To appear in Abramsky, S., Gabbay, D. M. and Maibaum, T. S. E. (eds) *Handbook of Logic in Computer Science*, Vol II, Oxford University Press).

[Tennent, 1991] R. D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, London, 1991.

[Wadler, 1992] P. Wadler. The essence of functional programming. In *ACM Symp. on Princ. of Program. Lang.*, 1992.