Object-Flow

Lee Braine* and Chris Clack

Department of Computer Science, University College London,
Gower Street, London, WC1E 6BT, UK

Abstract

The use of visual programming (VP) to assist either functional programming (FP) or object-oriented programming (OOP) has been extensively researched. However, the use of VP to assist the integration of FP and OOP has been largely neglected. This paper presents the key aspects of object-flow, a new visual notation that facilitates visual object-oriented functional programming (VOOFP).

1 Introduction

There have been many attempts to integrate OOP with FP [3], VP with OOP [4], and VP with FP [6]. However, to date we know of no language that integrates OOP and FP with VP, and yet retains the key features of both the functional and object-oriented paradigms. Existing attempts, such as *object-oriented dataflow* [7], typically sacrifice important features from either OOP or FP. In this paper, we present the novel visual aspects of object-flow, a VOOFP notation that is 100% functional and 99% object-oriented. The key contributions of this paper include:

1. an application of VP to the integration of OOP and FP, giving a visual representation for OOFP;
2. a visual representation of type-safe curried higher-order method sending.

2 Overview of CLOVER

We recently introduced CLOVER [3], an object-oriented functional language that is 100% functional and 99% object-oriented. CLOVER provides the FP features of referential transparency, single-assignment attributes, higher-order functions, curried partial applications, lazy evaluation and complete type safety. It also provides the traditional OOP features of a class hierarchy with inheritance and pure encapsulation, subsumption, subtyping, method overloading, method overriding and dynamic method despatch. CLOVER is intended for application development, particularly the creation of *executable specifications* [8].

3 Object-Flow

The object-oriented message send `o.f(x)` can be represented visually using an *object diagram* [2], as in Figure 1(a). The functional definition `a = f(x,y)` can be represented visually using a *dataflow graph* [5], as in Figure 1(b).

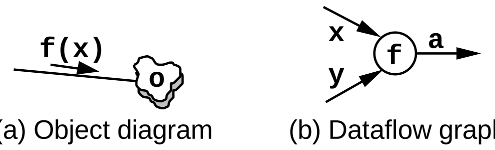

Figure 1. Standard visual representations

Our goal is to provide a visual notation that integrates the semantics of both the object-oriented and functional representations, despite their apparently-conflicting requirements. In particular, we wish to integrate object identity with referential transparency, and support higher-order methods, curried partial applications and lazy evaluation. This section presents *object-flow*, a new visual notation which provides these features.

We follow CLOVER's approach of extending FP towards OOP, rather than the other way around. This requires us to build upon a referentially transparent dataflow base. We first note that standard dataflow semantics do not provide support for key object-oriented notions such as dynamic despatch. We thus provide extra semantics to facilitate dynamic despatch by identifying the final parameter to be applied as the distinguished object.

Our next step is a notation change so that higher-order methods can be handled naturally. In the traditional functional dataflow model, each node contains a function name, and this function is applied to its incoming arguments. In order to permit the function itself to flow into a node, it is necessary to make each node an *application site* (see Figure 2) that receives a method, its arguments and a distinguished object.

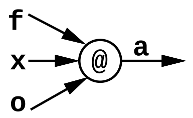

Figure 2. Application site

Object-flow does not permit fan-out; we use aggregate types instead, with explicit selection. This results in an equally expressive and powerful, but less concise, notation. However, a pleasant consequence is that we can eliminate arrows indicating flow direction; we merely work backwards from the result.

*During the course of this work, Lee Braine was supported by an EPSRC research studentship and a CASE award from Andersen Consulting.

The semantics of lazy evaluation are captured and visualised in object-flow by the use of a mechanical *winder* (see Figure 3) that "pulls" wires through application nodes from the left. Each method definition contains one winder – the result that is returned.

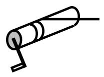

Figure 3. Object-flow winder

This visual metaphor can be extended with a node represented as a stack of tubes, each tube open at the appropriate end depending on whether it produces or consumes. Object-flow places the result at the top, followed by the method, its arguments and finally the distinguished object to give nodes with structures like Figure 4(a). We can also reduce visual clutter due to a plethora of arcs by allowing named values, as in Figure 4(b).

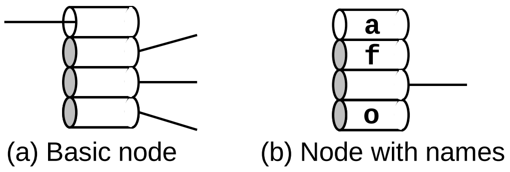

Figure 4. Object-flow nodes

Object-flow is naturally curried – adding another pipe to an application node adds another argument. To represent partial applications, we omit one or more pipes. To aid identification of partial applications, the editor automatically adds an exposed connector to the bottom-most tube, indicating that further pipes are required for full application. For example, we can partially apply `+` to create the local definition `inc` (see Figure 5) which increments a number.

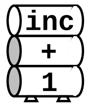

Figure 5. Local definition for `inc`

As a final example, we can represent the function definition `incList self = map (+ 1) self`, which increments every element in a list by employing the higher-order function `map` to apply `(+ 1)` to each element, as the object-flow method in Figure 6.

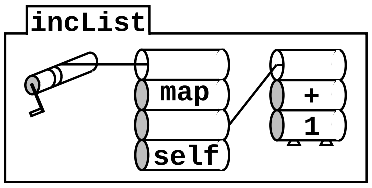

Figure 6. Method definition for `incList`

In object-flow, each arc carries a single atomic object or message, not a stream of objects or messages. This benefits abstraction, formal analyses and provides clean object-oriented semantics. We thus adopt the Actor model [1] view of sequences of behaviours rather than state changes. Also note that recursion is supported by simply naming a method within its object-flow definition (space precludes an example).

4 Development Environment

In our current CLOVER prototype, the development environment consists of a three pane Smalltalk-like browser for class hierarchy, class attributes and method types, and a graphical editor for defining methods using object-flow notation.

5 Further Work

We are currently extending object-flow notation to support file I/O and user interaction. Further work includes extending CLOVER with incremental type checking, algorithm animation and visual profiling.

6 Summary and Conclusion

Object-flow is a new visual notation that facilitates the integration of OOP and FP. In particular, it integrates object identity with referential transparency, and supports higher-order methods, curried partial applications and lazy evaluation.

References

[1] G. Agha, C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming," in B. Shriver, P. Wegner (Eds.), *Research Directions in Object-Oriented Programming,* MIT Press, pp. 47-74, 1987.

[2] G. Booch, *Object-Oriented Analysis and Design with Applications, 2nd Edition,* Benjamin Cummings, 1994.

[3] L. Braine, C. Clack, "Introducing CLOVER: an Object-Oriented Functional Language," *Proc. 8th International Workshop on Implementation of Functional Programming Languages,* pp. 21-38, 1996. Also to appear in Springer-Verlag LNCS.

[4] M. Burnett, A. Goldberg, T. Lewis (Eds.), *Visual Object-Oriented Programming: Concepts and Environments,* Manning Publications, 1995.

[5] A. Davis, R. Keller, "DataFlow Program Graph," *IEEE Computer*, 15(2), pp. 26-41, 1982.

[6] D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages," *Journal of Visual Languages and Computing,* 3(1), pp. 69-101, 1992.

[7] T. Kimura, "Object-Oriented Dataflow," *Proc. 11th IEEE Symposium on Visual Languages (VL'95),* pp. 180-186, 1995.

[8] D. Turner, "Functional Programs as Executable Specifications," in C. Hoare, J. Shepherdson (Eds.), *Mathematical Logic and Programming Languages,* Prentice-Hall, 1987.