

Functional Hypersheets (Extended Abstract)

Tony Davie and Kevin Hammond

Division of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife, Scotland.
Email: {ad,kh}@dcs.st-and.ac.uk

Abstract

This paper introduces *hypersheets*, an extension of conventional programming language modules. Whereas modules are static, and usually textually based, hypersheets constitute a local or distributed network of *hyperlinks*::dynamically evolving links to either values or definitions. Hypersheets integrate notions of persistence, hyperprogramming and user interface design.

In the purely functional context we consider here, hypersheets provide a software development environment for functional programming that supports version control, sophisticated scoping, evolution of values, and distributed program development.

This paper reports on very early work. We are therefore unable at present to report on either implementation results or user experience.

A full version of this paper has been submitted to the *1996 International Workshop on Implementing Functional Languages, Bonn, September 1996*.

1 Introduction

1.1. Hypersheets

Programmers are apt to dismiss spreadsheets as mere tools for accountants or other business users. Under the surface, however, the basic model of a spreadsheet reveals some interesting properties that can also be exploited for programming environments: a spreadsheet frequently displays the characteristics of *persistence* — its state needs to be preserved — and spreadsheets may contain *hyperlinks* to other sheets. For example, a company's monthly accounts might well be organised as a chain of spreadsheets, one for each month and each referring via a hyperlink to the previous month.

The restricted topology of the spreadsheet can be generalised to that of a *hypersheet* inhabited by (named) values. Some of these values may indeed be rectangular arrays (or lists) as with a spreadsheet, but in general they may be objects of any type (including functions). A hypersheet therefore constitutes a *model* of a module, in that it comprises a set of name-value bindings. We will use the terms *module* and *hypersheet* interchangeably throughout the remainder of this paper. Figure 1 shows the hypersheet that might correspond to the monthly account spreadsheet described above.

Spreadsheets often distinguish between cells which contain values and cells which contain formulae. From a functional perspective, this distinction is entirely artificial — values are simply special cases of formulae. Similarly, rather than treating formulae specially, perhaps as macros, they should just be ordinary functions. In our model, all hypersheet cells contain functional formulae and the connection between a cell's name and its contents constitutes a binding. Obviously, individual functions may also contain names which constitute uses of such bindings. These names are implicit uses of hyperlinks and compilation/linking will make them explicit (see Section 2.1). There is no reason at all however why a user should not make explicit hyperlinks to other local or remote cells. These hyperlinks may be represented either textually or graphically.

From a programmer's perspective, hypersheets can easily be exploited for program development. Normal ideas of module scoping can be modelled by including one hypersheet or module within another. Ultimately all such hypersheets are included within a “universal” hypersheet, the persistent store (see Section 1.2), but a given programmer will only require a view of those modules which are relevant to the task at hand. Obviously, different programmers may have different views of the same modules.

Hypersheet

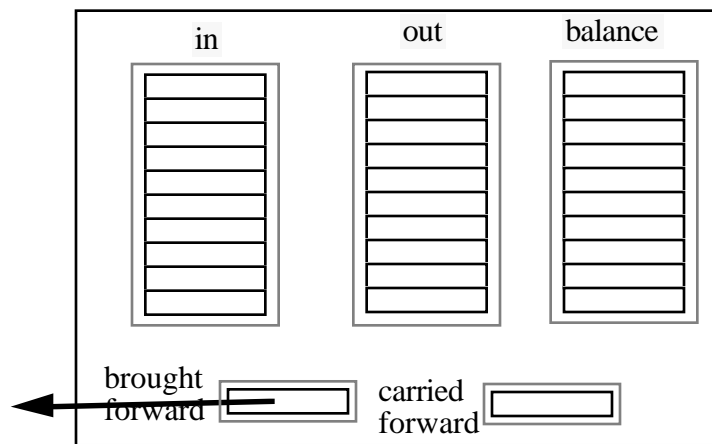


Figure 1: A Simple Hypersheet Corresponding Directly to a Spreadsheet

Figure 2 shows an example of a module under development with hyperlinks to the standard prelude hypersheet, and to another hypersheet which has already been developed. In this figure, all hyperlinks are shown explicitly. Since they clutter the user interface, most of these hyperlinks would normally be elided in a production hypersheet system.

1.2. Lazy Functional Programming and Persistence

The central theme of lazy functional programming is that no object needs to be evaluated more than once (if at all) [5]. The reason for this is that no object ever changes its value in any purely functional environment. As a consequence, any value that is needed to produce the result of the program need only be evaluated once.

Laziness ensures that only the needed parts of a data structure are actually evaluated. If such structures are used in one programming session, it would entirely consistent with the goals of laziness if the work done in that session could be exploited by a later session [10]. This can be achieved using persistence [1]. Where functions or closures must be stored, the stronger notion of *orthogonal persistence* is required: objects of *any* type may need to be saved in the persistent store.

Lazy evaluation is especially important during program development. For example, lazy evaluation can avoid creating parts of large data structures until they are needed, and if the links to other hypersheets cross network boundaries, it is clearly advantageous to evaluate remote objects only when and if they are needed.

1.3. Hyperprogramming

Hyperprogramming [7,8] is a technique for composing programs using links similar to those found in hypertext systems. In hypertext, the links between documents connect one piece of text to another. In hyperprogramming, links connect names in one hypersheet to values in another. Definitions (<name> = <text> in a conventional program development setting — but the placing of a formula in a named box in a hypersheet) may be seen as particular cases of local hyperlinks providing defining instances of bindings. A sophisticated hyperprogramming system allows users to compose programs not only by conventional text editing but also by setting up hyperlinks between program text and objects existing in other more or less remote hyperprograms.

2. Program Construction using Hyperprogramming Techniques

An important difference between the imperative and functional worlds is that in the latter case objects have a single value in any given context. Of course, the implementation may alter the state of evaluation of any object but this is normally hidden from the user. In a hyperprogramming system the distinction between an object and its value is even more blurred, as far as the user is concerned; and the text of a definition can even be taken to be the initial state of (un-) evaluation of some object. This suggests that lazy behind-the-scenes compilation might well be a good way of organising a persistent hyperprogramming system.

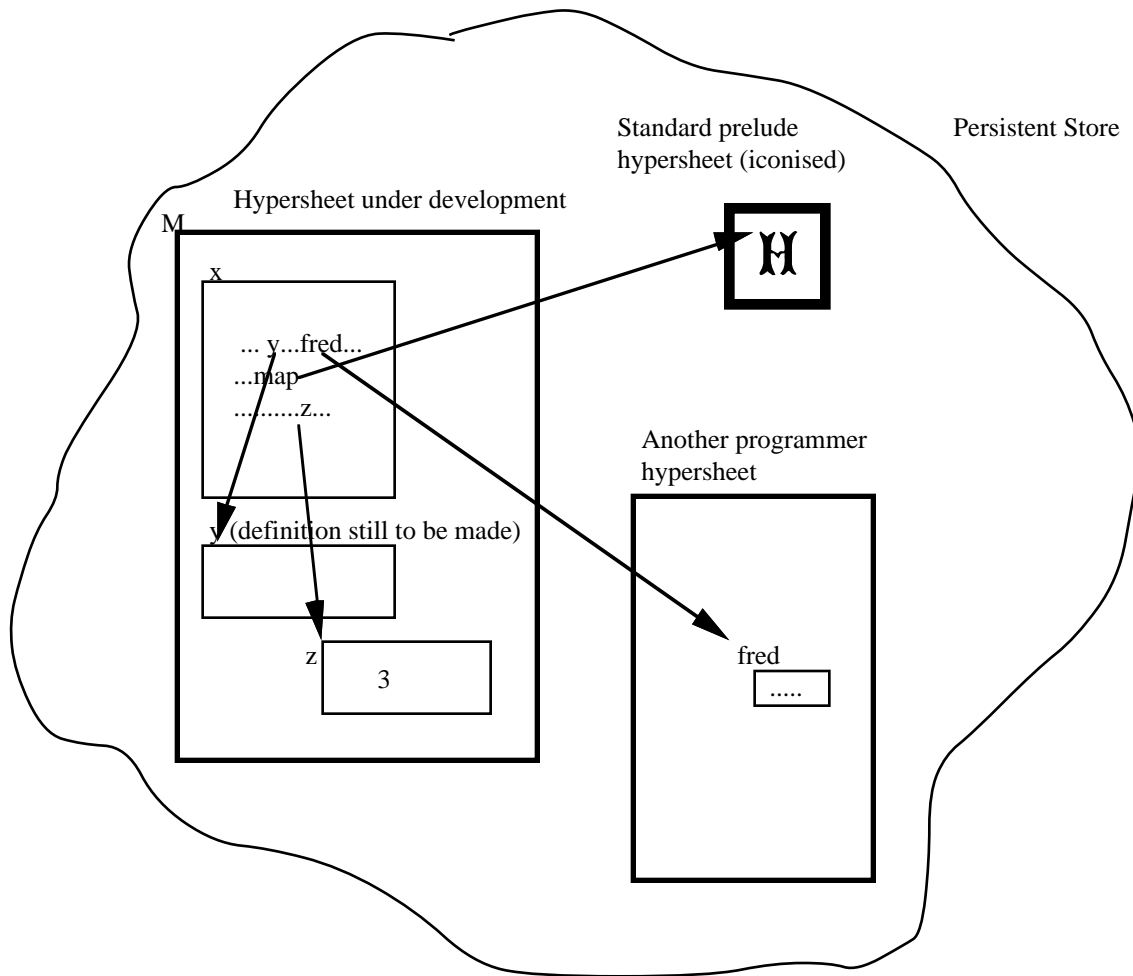


Figure 2: The Persistent Store Showing a Hypersheet Under Development.

Naturally, a user does have one way of altering the value of an object, by text-editing its definition. This can be seen, however, as altering the context, or environment, in which evaluation occurs. All that is really happening is that a new version of a module is being constructed. It is important to note that the alteration of one binding in this way implies the provision of new versions of other bindings which occur in a recursive dependency loop. It does not, however, imply the provision of new versions of *all* the definitions in the module being developed. This problem of dependency analysis is well understood and is described, for instance, by Peyton Jones [11].

One advantage of a hypersheet system is that partially correct or partially constructed programs/modules can be in existence. Objects only need to be checked for correctness or presence if they are encountered (needed) during compilation or execution. Errors would otherwise only be notified to users on a specific request for a fuller check.

2.1. Evolution of Hyperlinks

Users need to be able to reuse software and thus need to be able to refer to objects already stored in the persistent store. We imagine such objects to be located in modules consisting of name-value bindings (though as we shall see below, other information about the names is also held). A hyperlink is a connection between an identifier (which is normally a name, but could be anonymous) in one module and a value in another module.

A hyperlink can be in one of several states of definition. A hyperlink starts out completely undefined, in the sense that an identifier has been used but the system does not know what it is bound to. When a definition is made, it could be local with a conventional scope; it could be in another module; or it could be defined in several places. Links can progress during their lifetime from being completely undefined through stages where several definitions could be used (overdefinition) to being firmly defined.

3. Related Work

The work by de Hoon, Rutten and van Eekelen [3] on functional spreadsheets has clear connections with that described here. De Hoon et al. describe the implementation of a classical spreadsheet with functional features using functional implementation techniques. However they do not cover the hyperprogramming or persistence aspects which are central to our approach. The independently-designed CLOVER implementation from UCL also provides a spreadsheet-based interface to an object-oriented functional language [2], which might be developed to a hypersheet interface.

The ideas of lazy compilation and loading are present in the *Glide* system of Runciman, Toyn & Firth [12] but their unit of persistence is a *definition* rather than an object. Each such definition is mapped onto a text file which contains its source. In our system, however the units of persistence will, consistently with orthogonal persistence, be objects of any type — the values denotable by source expressions. This will ensure that work done in evaluating objects or parts of objects will persist, and not just any alterations made to the source of those objects.

The work we describe here is clearly related to research in Computer-Supported Cooperative Work (CSCW) in that it describes a distributed, cooperative program development environment. In general CSCW research encompasses a broader scope than that covered here, however, in that it considers general support for distributed workgroups of whatever kind (such as distributed email). We are interested only in the particular issue of program development. The relationship does need to be further explored. For example, Wray et al. describe an active module system for Medusa, a system that is intended to support distributed multimedia applications [13]. Hypersheets are clearly active modules of a kind, though our notion differs from that of Wray et al. in being much more implicit—hypersheets do not require any communication to be programmed and are automatically persistent, for instance.

There is an intriguing relationship between our work and the various schemes for parameterised and higher-order modules which, like hypersheets, also manipulate definitions and modules, but in an essentially static rather than dynamic fashion. Recently, for example, Jones [6] has described a higher order module system based on higher order polymorphism, support for structures with polymorphic components and a clear separation between static and dynamic semantics; and there have been several interesting module systems based on the SML model of environment structures [9]. On balance, the two notions seem symbiotic: higher-order modules are a structuring device that can be incorporated into a hypersheet-based system, and even perhaps used to explain the operations that the hypersheet performs when manipulating definitions.

As the hypersheet model is very general, it could, in an obvious way, be used to model the traditional styles of databases: flat, network or relational. The latter two are clearly more suited to the use of hyperlinks, especially network databases but the flat nature of much database work and the imperative style used for programming them (except for pure queries) do not at first glance match functional ideas very well. It might be possible to use hyperlinks to capture database relationships in a more direct way, in the same way that graph links can be exploited in a conventional functional language [4].

Finally, McNally [10] has already integrated persistent and lazy functional programming in the STAPLE system produced at St Andrews University. Our work draws on this experience, but extends that of McNally in two major ways:

- 1) it also integrates the more recent concept of hyperprogramming;
- and 2) it provides a new model of conventional module structures.

4. Summary

This paper has proposed a hypersheet interface to a lazy functional hyperprogramming system. The system is functional at two levels: firstly in the language it supports, and secondly in the operations which are used to construct programs.

Hyperlinks are a natural extension of the links used in conventional graph-reduction. External hyperlinks allow persistent objects to be shared, allowing dynamic graph-reduction. We have discussed how values and hyperlinks evolve during the development of a program. At the user level, hyperlinks start by being undefined, may become overdefined and eventually become firm. At the system level, values reduce lazily to their (unique) normal form. Changes made at both these levels will persist as long as they are accessible by some user.

These ideas have not yet been implemented, though a prototype implementation of a hypersheet interface is currently being undertaken at St Andrews.

References

1. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, P.W., & Morrison R., "An Approach to Persistent Programming", *Computer J.* **26**(4), pp. 360-365, 1983.
2. Clack, C, & Braine, L., "Introducing CLOVER: an Object-Oriented Functional Language", *Proc. 8th. International Workshop on Implementation of Functional Languages*, Bonn, pp. 21–38 (Draft Proceedings), September 16–18th, 1996.
3. de Hoon, W.C.A.J., Rutten, L.M.W.J. & van Eekelen, M.C.J.D. "Implementing a Functional Spreadsheet in Clean", *J. Functional Programming* **5** (3), pp383-413, 1995
4. Hammond, K., & Trinder, P.W., "A Functional Perspective of Bulk Data Types", *Proc. 1995 Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1995, Springer-Verlag WICS, 1995.
5. Hughes, R.J.M., "Why Functional Programming Matters", *Computer J.* **32**(2), pp. 98-107, Apr. 1989.
6. Jones, M.P., "Using Parameterized Signatures to Express Modular Structure", *POPL'96 — Proc. 23rd ACM Symposium on Principles of Programming Languages*, St.Petersburg Beach, Florida, 1996
7. Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R., "Persistent Hyperprograms", *Proc. 5th International Workshop on Persistent Object Systems*, San Miniato, Italy, pp. 73-95, 1992.
8. Kirby, G.N.C., *Reflection and HyperProgramming in Persistent Programming Systems*, Ph.D. Thesis, Department of Mathematical and Computational Sciences, St. Andrews University, 1993.
9. MacQueen, D.B., "Modules for Standard ML", 1984 *ACM Symposium on LISP and Functional Programming*, pp 198-207, Austin, Texas, 1984
10. McNally, D.J., *Models for Persistence in Lazy Functional Programming Systems*, Ph.D. Thesis, Department of Mathematical and Computational Sciences, St. Andrews University, 1993.
11. Peyton Jones, S.L. *The Implementation of Functional Programming Languages*, Prentice Hall International, ISBN 0-13-453325-9, 1987
12. Runciman, C., Toyn, I. & Firth, M. "An Incremental, Exploratory and Transformational Environment for Lazy Functional Programming", *J. Functional Programming* **3**(1), pp. 93-115, Jan. 1993.
13. Wray, S., Glauert, T. & Hopper, A. "The Medusa Applications Environment", *IEEE Multimedia*, **1**(4), pp. 54–63, 1994.