FUNCTIONAL PROGRAMMING FOR DUMMIES: THE DATA FLOW PERSPECTIVE.

B. Grieger, Aurora Technology B.V. for the European Space Agency (ESA), European Space Astronomy Centre (ESAC), Camino Bajo del Castillo s/n, 28692 Villanueva de la Cañada, Madrid, Spain (bgrieger@sciops.esa.int).

Introduction: Until recently, functional programming was not widely known and mostly only used in academics (from where it originated). But now, also major companies have started to pick it up and the phrase is heard more often. Many people might in fact have applied functional programming long before they first heard it.

The name may not be quite elucidating to many, as all programming languages know *functions*. The definition of *pure* functions and the lambda calculus used to describe functional programming are very abstract.

The principal property of functional programming is that it establishes a data flow. A data flow *can* be defined in terms of functions, but it does not *have to*. Describing the data flow directly makes functional programming much more intuitive.

Spreadheets: The commonly used spreadsheet programs are in fact manifestations of functional programming. A spreadsheet cell can contain a function of values from other cells (which may also contain functions). Such a function establishes a data flow between cells. These function are *pure* in the functional programming sense in that they only provide their value when called and have no side effects.

Note that the user does not write a sequence of commands (imperative programming), but rather defines a data flow (functional programming). The spreadsheet program decides what to execute, and when.

OpenDX: OpenDX provides a right away view on the data flow. OpenDX is a powerful 3D visualization system introduced by IBM in 1991 as Visualization Data Explorer. It was envisaged to superseed IDL in the world of scientific visualization, but never really succeded. In 2000, it was handed over to the open source community as OpenDX. Further development idled out about 2007.

Open DX is a true data-flow implementation [1], where all modules are pure functions (i. e., their outputs are fully defined by their inputs). While Open DX uses (pure) functions under the hood, the visual programming interface visualizes directly the data flow, see Fig. 1.

The visual program can be executed on demand or automatically if something changes. In the latter case, only modules downstream of the change are re-executed.

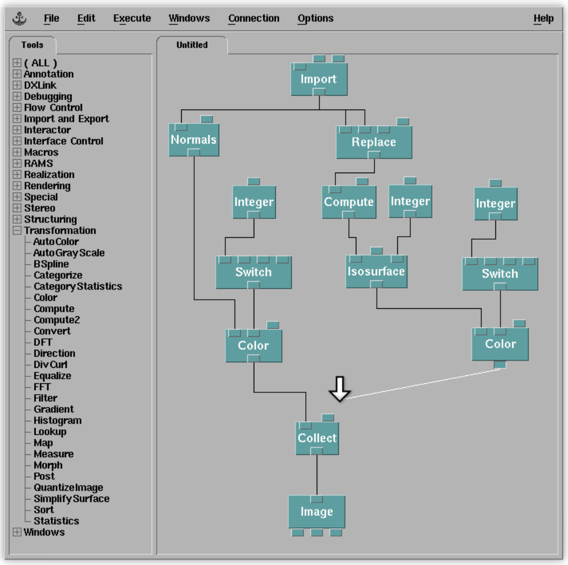

Figure 1: An OpenDX program is composed visually by placing modules from the tool bar on the canvas and connecting them by click and drag.

The ∗nix make utility: The make utility [2] available on all ∗nix systems is mostly used to build executables from source code. The user provides a so-called Makefile which contains rules that describe the dependencies between files (i. e., the object file for a subroutine depends on the file which contains the source code) and commands to create or update files (compile or link commands). The make utility reads the Makefile, checks the modification times of all files, and executes (only) the commands needed to update everything.

The rules in the Makefile represent a data flow description. The sequence of rules is arbitrary (though the placement of other structures, e. g., macros, may matter). The make utility decides which comands to execute, and when. The make language is true funtional programming without functions.

The make utility cannot only be used to build executables but basically for all kinds of computations. A peculiarity is that make connects several main programs, not subroutines, that exchange data via the hard disk. So all intermediate results are preserved between make runs.

The arcs wrapper language for make: Assume one wants to connect two programs with make, e. g., a program that applies a dark correction to an image (or many) with a program that afterwards applies a flat field. The benefit of this is that when the dark correction is updated, everything has to be recomputed, but when the flat fielding is updated, only that has to be rerun. The user does not need to keep track of any changes and take care of appropriate recomputation — the make utility will do that.

Establishing such a connection requires inserting code at three places: code to write the dark corrected image to file in the dark correction program, code to read the dark corrected image from file in the flat fielding program, and the rule with the file dependencies and the program execution commands in the Makefile. This is not only tedious, but also error prone. All three code snippets have to be absolutely consistent which each other.

This made the author creating a pure data flow description language called arcs. The arcs compiler reads a file which contains "arcs" describing connections between "modules". The compiler inserts the code to read and write files as appropriate into the source code of the modules and writes a Makefile with the respective rules. Each "arc" is maintained at a single point in the arcs file.

One application of the arcs language is the tool Envisionary for early science operations studies for the EnVision mission to Venus. The data flow of the computation of ground station events is illustrated in Fig. 2.

The spk arc (top right) is the SPICE spacecraft position kernel, which contains the spacecraft trajectory. The two compute_∗ modules right below are quite time consuming, but they have only to be re-executed when the spacecraft trajectory changes, which does not happen often. Everything else runs pretty fast, so for any changes downstream of the compute_∗ modules, re-running Envisionary takes very little time.

Envisionary can perform various other tasks, e. g., computation of Region Of Interest coverage. The data flow approach has proven to be very efficient in terms of rapid development and computational performance.

The dataflow C++ template library: Functional programming languages like Haskell or functional programming extensions for other languages all use functions to establish a data flow. In order to be able to directly describe the data flow, the author has created a C++ template library called dataflow.

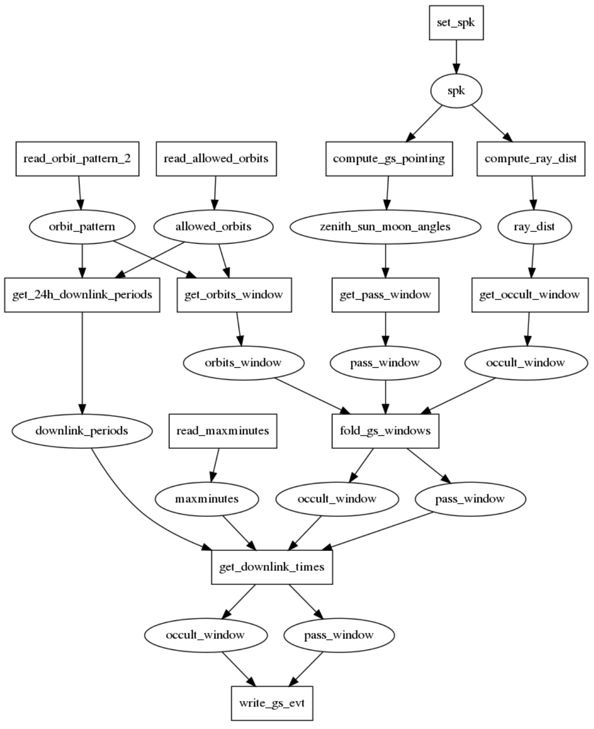

Figure 2: The data flow of the computation of gorund station events with Envisionary. Boxes are modules, ovals are "arcs" (data structures passed between modules). Only a subset of modules (13 out off ≈ 220) is displayed. Such diagrams are automatically created by the arcs compiler from source code.

The dataflow library is implemented object oriented. Modules are objects which provide methods to access their output. The output of a module can be connected to the input of another module by a simple command. If an output is requested, only upstream modules directly or indirectly connected for which any input has changed are re-executed.

Conclusions: Functional programming is a powerful programming paradigm and the specification of a data flow to solve a problem is in fact intuitive and straight forward. However, the use of functions to describe the data flow obscures the inherent simplicity and makes writing and understanding functional programs more difficult. We have discussed some tools that allow the user to directly describe the data flow and facilitate "functional programming without functions".

References: [1] IBM Visualization Data Explorer User's Guide, `http://www.opendx.org/support.html`. [2] GNU Make Manual, `https://www.gnu.org/software/make/manual`.