XMonad

A Tiling Window Manager

Don Stewart
Computer Science and Engineering
University of New South Wales
dons@cse.unsw.edu.au

Spencer Janssen
Computer Science and Engineering
University of Nebraska-Lincoln
sjanssen@cse.unl.edu

Abstract

`xmonad` is a tiling window manager for the X Window system, implemented, configured and dynamically extensible in Haskell. This demonstration presents the case that software dominated by side effects can be developed with the precision and efficiency we expect from Haskell by utilising purely functional data structures, an expressive type system, extended static checking and property-based testing. In addition, we describe the use of Haskell as an application configuration and extension language.

Categories and Subject Descriptors D.4.9 [*Systems Programs and Utilities*]: Window managers

General Terms Design, Languages, Reliability

Keywords Haskell, functional programming

1. Overview

System software is all about effects: manipulation of the file system, the network or graphics devices. Such tasks are often complex, error-prone and ill-specified, yet the software needs to be robust to ensure system stability. Incongruously, the system software domain is still dominated by programs written in legacy unsafe languages, despite the difficulty developing correct software with these tools. In this demonstration we will present the `xmonad` window manager, and describe approaches to producing robust system software in Haskell, using purely functional data structures, type system safety, extended static checking and property-based testing. We argue that Haskell is ideally suited for the production of concise, efficient and safe system software, using tools available *now*!

`xmonad` is a window manager for the X Window System. It acts as the brain of the X server, receiving events and using them to instruct the server to arrange windows on the desktop. Unlike most window managers, which require manual organisation of the desktop by the user (typically by dragging windows around), `xmonad` tiles windows *automatically*, ensuring maximum use of screen real estate. Tiling algorithms may be specified by the user, in Haskell, in their configuration files. Clearly, receiving external events and driving an X server requires a lot of IO – so how do we ensure we build a well-specified, checked and tested window manager, without giving up on purely functional programming?

The key decision to tackle the "IO problem" is to break the application into a model-view-controller design, where a complete window manager model is encoded in a purely functional data structure: a `Stack` type modeling a multiple screen, multiple workspace set of windows. Window manager state, including window and screen focus, geometry and layout is encoded directly in this data structure, which can then be rigorously specified and tested with QuickCheck and Catch. The event handler becomes a controller which traverses and manipulates the model structure. Actually interacting with the X server (the view) is simply a thin IO skin which renders the `Stack` structure with a sequence of calls to the server.

A benefit of this approach has been the use of semi-formal specification to guide the development of a clean window manager API. We have found that window manager operations that are hard to specify in QuickCheck tend to be hard for users to understand too – QuickCheck has a fine nose for "code smell". By using feedback from property specification to guide design the result is a simpler, more robust system with cleaner semantics than would otherwise have been developed. In effect, QuickCheck and Catch can be used to provide mechanical support for developing a clean, orthogonal API for a complex system.

In addition, wherever feasible we encode system invariants in the type system (for example, functions that require a non-empty workspace will reflect that in their type). The use of the type system in this way again limits the fallout from refactoring and extension of the system, by disallowing entire classes of bugs.

The other critical design taken is to allow extension of `xmonad` directly in Haskell. No ad-hoc extension language is developed, instead the `xmonad` configuration file is just a Haskell module. This has enabled a diverse range of extensions to be built up rapidly by users who take advantage of an extension language with a wide array of libraries and development tools. We argue that when the core application is written in a high level language, the system itself should be extended in that language.

The use of a high level language like Haskell, for implementation and extension, that takes safety and correctness seriously, along with static analysis and strong testing, makes it possible to produce robust system software that is efficient, correct and a joy to develop.

Haskell'07, September 30, 2007, Freiburg, Germany.
ACM 978-1-59593-674-5/07/0009.