CS 490 Project Proposal

Benjamin Lerner

October 1, 2003

1 Abstract

Yampa is a functional reactive language developed as a for the specific domain of robotic control. By combining both discrete events and time-varying signals, it incorporates the main types of actions and stimuli a robot likely encounters. To efficiently implement Yampa, some control over the types of signals is required, as otherwise space- and time-leaks develop, as explained in [HCNP03]. To achieve this end, signals are represented in Yampa as arrows, hiding their actual implementation and only exposing an interface by which the programmer can manipulate them. This abstraction evidently cures the leaks from before. However, it now makes the assumption that the signal class it uses is in fact an arrow, as otherwise the implementation would be unsound.

We formally prove that the type SF (the basic signal function class used in Yampa), defined in [HCNP03], satisfies the arrow laws as defined in [Hug00], for the purposes of showing that Yampa, which is built around the properties of this class, in fact is sound.

2 Introduction

The Yampa language has been used to program or model the programming of full-scale robots as mentioned in [HCNP03]. Also described in that paper is a model of a simple mobile robot, and examples of how to program it with Yampa. The paper also introduces the notion of an arrow, and explains some of the simple combinators possible using them.

Initial development on Yampa was built on the functional-reactive systems previously developed – Fran, FAL and FRP (as noted in [HCNP03]) – which used a similar architecture. However, as also mentioned in that paper, and explained in more detail elsewhere, many programs written in Yampa developed space- and time-leaks, degrading the performance of the system to the point where it was no longer a real-time responsive model. The reasons underlying this were nontrivial, but primarily stemmed from the availability of signals as first-class values. To address this issue, the underlying implementation of signals was changed to use an arrow, thus hiding the signals themselves from computation, and only exposing the manipulations permissible on them. This abstraction is valid as long as the signal class in fact is an arrow; that is, that it satisfies the axioms an arrow must uphold.

Arrows were initially defined in [Hug00] as an extension of monads, a standard concept in functional programming. The extension was motivated by noting that certain parsing constructs could not be implemented as monads, which was worrisome to the extent that parsers are rather ubiquitous, and if they could not be implemented as such, perhaps the definition was too restrictive. Other examples were presented later, showing how the arrow interface defined in this paper can be used with many other common applications. The paper also proves that monads can be simply and almost trivially embedded in an arrow construction, showing that all the functionality of any given monad can be expressed by a suitably defined arrow as well. Manipulations of arrows are achieved by various combinators, which express formally what is intuitively a "wiring diagram" for the flow of data through a program. There exist minimal sets of three combinators which together are universal, that is, can define all other possible combinators for arrows. The selection of which three should form such a set is a matter of convenience and taste; most often a larger set of combinators is employed, for ease of use [HCNP03].

Just as monads satisfy three axiomatic "monad laws", arrows must satisfy a set of at least fifteen axioms as well. These laws describe the associativity and extensionality of arrows, as well as more abstract proprties such as requiring that various combinators preserve composition of arrows[Hug00]. Satisfying these laws, together with implementations of a universal set of combinators, defines an arrow.

3 Project Questions and Deliverables

This project will attempt to prove that the class SF, defined for the Yampa language, does in fact satisfy the arrow laws, and will present formal proofs of each. In the event that a law is provably untrue for the current implementation, some attempt will be made to show what portion of the implementation violates the laws, and how to fix the code to then conform to it.

References

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson, *Arrows, robots, and functional reactive programming*, Summer School on Advanced Functional Programming 2002, Oxford University, Lecture Notes in Computer Science, Springer-Verlag, 2003, To Appear.

[Hug00] John Hughes, *Generalising monads to arrows*, Science of Computer Programming **37** (2000), no. 1–3, 67–111.