Fine-Grained Staeful Computations

We propose a new approach for expressing stateful computations in Haskell. We use Haskell's type system features to construct highly polymorphic descriptions of stateful computations. These descriptions allow for multiple interpretations, including static analysis, simulation and symbolic execution. As an application, we build a proof-of-concept model of a simple hypothetical instruction set architecture.

Background

A computation is called stateful if its result depends on a mutable state. This may be the values of global variables in procedural programming or values of class members in the object-oriented paradigm. Pure functional languages emphasise immutability of values and referential transparency of functions and thus take a different approach.

The State monad is the idiomatic way of expressing stateful computations in Haskell. Hiding the state threading under the monadic hood greatly simplified the code and makes it more composable. However, it has one crucial limitation: all state modifications are monolithic, i.e. there is no way to statically track the reading or altering of a certain component of a composite state, i.e. a record datatype comprising multiple fields.

In some situations, we do want this granularity in order to perform static analysis of stateful computations and reason about their dependencies. We want to have a way to scrutinise a stateful computation without actually running it, and extract its read and write dependencies, e.g. to understand which components of the state are being read or altered by it.

Approach

We propose to mitigate the lack of granularity by forcing ourselves to think of the state as a key/value store. We want to be able to read a value associated with a key, performing the effects caused by reading; and write an effectful value into the store, performing two effects: one associated with the value itself, and the one required for writing. Moreover, we want to be aware of the structure of the effects we perform, i.e. whether that structure is static or dynamic in order to distinguish the computations which can be analysed statically. To achieve this, we parametrise computations with a polymorphic constraint, which could be instantiated with an Applicative for static effects or with a Monad for dynamic effects.

Applications

The primary application of our approach is modelling of instruction set architectures. We describe the semantics of a simple hypothetical instruction set architecture in terms of fine-grained stateful computations. By interpreting this polymorphic semantics in different computations contexts, we build simulation, symbolic execution and dependency analysis backends for the ISA model.