Cooperating Solvers vs. Cooperating Languages

Alan Borning

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2012-005

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Cooperating Solvers vs. Cooperating Languages

Alan Borning

31 August 2012

1 Introduction

We initially started designing an architecture for cooperating constraint solvers, but quickly moved to designing an architecture for cooperating languages. It seems worth revisiting this issue and more carefully differentiating the approaches. So in this informal note I first sketch the two approaches (in as different forms as possible), then discuss how things are actually less distinct, and finally describe some additional issues around cooperating languages.

2 Cooperating Constraint Solvers

The goals of this version are:

- to design a framework to accomodate an extensible collection of constraint solvers that can interoperate
- to populate that framework with some useful solvers
- to design an API for the framework that lets it be used from your favorite language (or even multiple APIs for different languages)
- to use the framework to solve some useful problems

Designing the framework includes defining how the problem is represented and decomposed into different pieces, and how the solvers communicate. It also includes defining an "executive" that can decide which solvers to use and when, perhaps querying the solvers themselves regarding their capabilities. Assuming we are at least for now still writing in Squeak, the solvers might be in Squeak (e.g., Cassowary, Ted's exhaustive search solver, or DeltaBlue), or external (e.g., Z3, Kodkod).

However, the cooperating solvers don't attempt to be interpreters for full programming languages. Instead, they are used from some language (or from multiple languages). The Cassowary API is an example of an API for a solver framework: there is a class for constraints, for constrainable variables, and for instances of the Cassowary solver.

In the simplest form of this project, constrained variables aren't the same as ordinary variables in the host language, and the host language doesn't have any built-in knowledge of them or of the solvers – you have to explicitly invoke it. This is simple and flexible, and also means that the programmer can easily just bypass the constraint solver. Again, the Cassowary API is an example.

The "Architectures for Cooperating Solvers" memo of 18 March 2012 describes a multi-level architecture for cooperating solvers. The constraint graph is partitioned into regions joined by variables that are read-only for all but one of the solvers. The overall graph is acyclic. This gives a simple way for the top-level solvers to cooperate. Within a region, there can also be subregions with again cooperating solvers that permit richer kinds of interaction. The memo also references related work in the area (there is a reasonable amount of work on cooperating solvers already, for example SMT solvers), and outlines some additional goals, such as supporting soft constraints as well as required ones, read-only annotations on variables, incremental updating of solutions, and perhaps compilation.

3 Cooperating Languages

In this version of the project, we do have cooperating languages rather than just cooperating solvers: at least some of them should be Turing-complete. The multi-level architecture for cooperating solvers actually still seems like a good one for this version as well: at the top level, programs still communicate by shared variables, which are read-only for all but one of the region. There can also be subregions with cooperating languages that interact in richer ways. (It's also reasonable to start with just the simple version, with no subregions.)

There are still some unresolved issues I think:

- How is time handled in the different regions? Presumably each region (or perhaps even each subregion) should have its own clock and notion of time — a single global clock doesn't seem right.
- How does this interact with KScript and streams? Do the shared variables sometimes hold streams of values?
- Does there need to be a host language in which the DSLs are embedded? Or are they standalone, with just a "wiring language" for expressing the interconnections?

Note that this design uses values communicated via the shared variables. Another very standard way for processes to communicate is via message streams. (These are probably equivalent at some level — you can simulate message streams with a stream of values where the type of the value is "message"; and you can simulate a stream of values with messages that just hold a "here's a new value" message. I think.)

4 Blurring the Projects

The two versions aren't mutually exclusive: we could do them both, and have cooperating languages, with some of the languages calling solver libraries.

Further, for the different cooperating solvers we can define languages in which to write the constraints. The Things language is an example, and also illustrates how we may still want some external language to manipulate the objects. (There could be an interactive editor that lets the user construct electrical circuits or bridges or whatever; this could also be done programmatically, using iteration, conditionals, and recursion in the commands to build some structure.)

Not all of the DSLs need to be Turing complete, but certainly at least one should be.

How far can we go with DSLs? I think we still need a general purpose language, if nothing else for expressing computations where we don't (yet) have a clean, very compact description in a DSL. Even if the goal is to express all aspects of a personal computer environment using concise, specialized DSLs, a reasonable development strategy would be to evolve toward that, with parts written initially in some general language in a less elegant way. (Is KScript intended to be that language?)

Finally, are the programs written in the DSLs essentially standalone, with just a "wiring language" for expressing the interconnections? Or are the programs in a DSL embedded in a general-purpose language, in the same way that currently KScript programs can be written as strings in Squeak and called from Squeak?