Science of Computer Programming 93 (2014) 19–38

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

The syntax and semantics of a domain-specific language for flow-network design☆

Assaf Kfoury*

Computer Science Department, Boston University, 111 Cummington Street, Boston, MA 02215, USA

ARTICLE INFO

Article history:
Received 16 February 2012
Received in revised form 17 December 2012
Accepted 19 December 2012
Available online 9 January 2013

Keywords:
Network specification
Flow conservation
Capacity constraint
Typing
Vector space

ABSTRACT

Flow networks are inductively defined, assembled from small components to produce arbitrarily large ones, with interchangeable functionally-equivalent parts. We carry out this induction formally using a *domain-specific language* (DSL). Associated with our DSL are a *semantics* and a *typing theory*. The latter gives rise to a system of formal annotations that enforce desirable properties of flow networks as *invariants* across their interfaces. A prerequisite for a typing theory is a *formal semantics*, i.e., a rigorous characterization of flows that are *safe* for the network (limited to the notion of *feasible* flows in this paper, unfeasible flows being considered unsafe). We give a detailed presentation of a *denotational semantics* only, but also point out the elements that an equivalent *operational semantics* must include.

1. Introduction and motivation

The background leading to the research reported herein is a little unusual. The motivation comes from the modeling and analysis of software systems that are assembled in an incremental and modular way. We devote some space in this introduction to explain this background.

Flow networks. Many large-scale, safety-critical systems can be viewed as interconnections of subsystems, or modules, each of which is a producer, consumer, or regulator of *flows*. These flows are characterized by a set of variables and a set of constraints thereof, reflecting *inherent* or *assumed* properties or rules governing how the modules operate and what constitutes safe operation. Our notion of flow encompasses streams of physical entities (*e.g.*, vehicles on a road, fluid in a pipe), data objects (*e.g.*, sensor network packets, video frames), or consumable resources (*e.g.*, electric energy, compute cycles).

Traditionally, the design and implementation of such *flow networks* follows a bottom-up approach, enabling system designers to certify desirable safety invariants of the system as a whole: Properties of the full system depend on a complete determination of the underlying properties of all subsystems. For example, the development of real-time applications necessitates the use of real-time kernels so that timing properties at the application layer (top) can be established through knowledge and/or tweaking of much lower-level system details (bottom), such as worst-case execution or context-switching times [1–3], specific scheduling and power parameters [4–7], among many others.

While justifiable in some instances, this vertical approach does not lend itself well to emerging practices in the assembly of complex large-scale systems – namely, the integration of various subsystems into a whole by "system integrators" who

☆ The research reported herein is partially supported by NSF awards CNS-0952145, CCF-0820138, CSR-0720604, CNS-1012798, and EFRI-0735974.
* Tel.: +1 617 3538911.
E-mail addresses: kfoury@bu.edu, a.kfoury@gmail.com.
URL: http://www.cs.bu.edu/~kfoury.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.12.009

may not possess the requisite expertise or knowledge of the internals of these subsystems [8]. This latter alternative can be viewed as a *horizontal* and *incremental* approach to system design and implementation, which has significant merits with respect to scalability and modularity. However, it also poses a major and largely unmet challenge with respect to verifiable trustworthiness – namely, how to formally certify that the system as a whole will satisfy specific safety invariants and to determine formal conditions under which it will remain so, as it is augmented, modified, or subjected to local component failures.

Incremental and modular design. Several approaches to system design, modeling and analysis have been proposed in recent years, overlapping with our notion of flow networks. Apart from the differences in the technical details – at the level of formalisms and mathematics that are brought to bear – our approach distinguishes itself from the others by incorporating from its inception three inter-related features/goals: (A) the ability to pursue system design and analysis without having to wait for missing (or broken) components to be inserted (or replaced), (B) the ability to abstract away details through the retention of only the salient variables and constraints at network interfaces as we transition from smaller to larger networks, and (C) the ability to leverage diverse, unrelated theories to derive properties of components and small networks, as long as such networks share a common language at their interfaces – a strongly-typed *domain-specific language* (DSL) that enables assembly and analysis that is agnostic to components' internal details and to theories used to derive properties at their interfaces.

Our DSL provides two primitive constructors, one is of the form $\left(\mathcal{M}_1 \parallel \mathcal{M}_2\right)$ and the other of the form $\textbf{bind}\,(\mathcal{N}, \langle a, b\rangle)$. The first juxtaposes two networks $\mathcal{M}_1$ and $\mathcal{M}_2$ in parallel, and the second binds an output port a to an input port b in a network $\mathcal{N}$. With these two primitive constructors, we define others as *derived* and according to need. A distinctive feature of our DSL is the presence of *holes* in network specifications, together with constructs of the form: $(\textbf{let}\, X = \mathcal{M}\, \textbf{in}\, \mathcal{N})$, which says "network $\mathcal{M}$ may be safely placed in the occurrences of hole X in network $\mathcal{N}$". What "safely" means, depends on the invariant properties that typings are formulated to enforce. There are other useful constructs involving holes which we discuss later in the paper.[1]

Types and formal semantics. Associated with our DSL is a *type theory*, a system of formal annotations to express desirable properties of flow networks together with rules that enforce them as *invariants* across their interfaces, *i.e.*, the rules guarantee the properties are preserved as we build larger networks from smaller ones.

A prerequisite for a type theory is a *formal semantics*, *i.e.*, a rigorous definition of the entities that qualify as safe flows through the networks. There are standard approaches which can be adapted to our DSL, one producing a *denotational* semantics and another an *operational* semantics. In the first approach, a safe flow through the network is denoted by a function, and the semantics of the network is the set of all such functions. In the second approach, the network is uniquely rewritten to another network in *normal form* (appropriately defined), and the semantics of the network is its normal form or directly extracted from it. We give a detailed presentation of the denotational approach only, but also point out the elements that an equivalent operational approach must include, so that an equivalence can be established between the two.

We prefer the denotational approach for several reasons, one of which being to avoid an exponential growth in the size of network specifications when rewritten to normal form in the operational approach. We thus prove the soundness of the typing system ("a type-safe network construction guarantees that flows through the network satisfy the invariants properties enforced by types") without having to explicitly carry out exponential-growth rewriting.

Paper organization and scope. Section 2 is devoted to preliminary definitions. Section 3 introduces the syntax of our DSL and lays out several conditions for the well-formedness of network specifications written in it. Section 4 defines the formal semantics of flow networks. Sections 5–7 present a type theory based on the syntax and semantics of the preceding sections.

For illustrative purposes, we consider only one *safety* property – namely, to be *safe*, a flow must satisfy (1) linear constraints of *flow conservation* at nodes/hubs and (2) linear *capacity constraints* that restrict the range of permissible values along links/connections between nodes/hubs. Types and typings are then formulated precisely to enforce this kind of safety across interfaces.

This paper presents the bare bones of a relatively small DSL for the purpose at hand. The concluding section, Section 8, discusses various extensions of the syntax, the semantics, and the invariant properties that a type theory may enforce.

The technical background presumed by the paper is familiarity with standard formalisms to define the syntax and semantics of programming languages (*e.g.*, the textbooks [13–15] among others), familiarity with conventions and notions of type systems for programming languages (again the textbooks [13–15]), and some knowledge of vector spaces up to and including optimization of linear functions, using any of the standard algorithms for linear programming (*e.g.*, the textbooks [16–18]).

No implementation issues of any of the algorithms, whether directly formulated or invoked, are taken up in this paper. In particular, we leave an analysis of time and space requirements to a subsequent report.

[1] Holes as placeholders have been used in other formal environments for design and analysis, such as in *Susan* (a text templating language tied to the object-oriented modeling languages Modelica and MetaModelica [9–12]). However, these other uses of holes are different from ours in several respects. In particular, they do not involve types and typings that set conditions at hole interfaces/boundaries that must be satisfied for safe placement in the holes.