Journal of Logical and Algebraic Methods in Programming 84 (2015) 708–729

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Lightening global types [☆]

Tzu-Chun Chen

Università di Torino, Dipartimento di Informatica, Corso Svizzera 185, 10149 Torino, Italy

ARTICLE INFO

Article history:
Received 2 September 2014
Received in revised form 17 June 2015
Accepted 23 June 2015
Available online 29 June 2015

Keywords:
Communication centred programming
Session types
Global types

ABSTRACT

Global session types prevent participants from waiting for never coming messages. Some interactions take place just for the purpose of informing receivers that some message will never arrive or the session is terminated. By decomposing a big global type into several light global types, one can avoid such kind of redundant interactions. This work proposes a framework which allows to easily decompose global types into light global types, preserving the interaction sequences of the original ones but for redundant interactions.

1. Introduction

Since cooperating tasks and sharing resources through communications under network infrastructures (e.g. clouds, large-scale distributed systems, etc.) have become the norm and the services for communications are growing with increasing users, there is a need to give programmers an easy and powerful programming language for developing interaction-based software applications. For this aim, Scribble [2], a communication-based programming language built upon the theory of global types [3,4], is introduced. A developer can use Scribble as a tool to code a global protocol, which stipulates any local endpoints (i.e. local applications) participating in it. The merits of global types, which describe global protocols, are (1) giving all local participants a clear map of what events they are involved in and what are the behaviours for those events, and (2) efficiently exchanging, sharing, and maintaining communication plans across platforms.

1.1. Motivation

However, global types do not ensure an efficient communication programming. The scenario of a global communication can be very complex, so it becomes a burden for programmers to code interactions to satisfy protocols. At runtime, the cost of keeping all resources ready for a long communication and for maintaining the safety of the overall system can increase a lot.

Consider the global scenario, shown in Fig. 1, which describes how a gift requester can get a *key* (with her identity) to fetch a wanted gift. Assume identity, guide, key, and gift are abstract type names, which can be types of string (denoted by Str), integer (denoted by Int), or bool (denoted by Bool). In order to get the key, she needs to get a *guide* from `map` for finding the key (communication labelled *reqGuide*). If the requester successfully gets the *guide*, she then proceeds to ask `issuer` for the *key* (communication labelled *reqKey*), and she can ask `store` for the gift with the given key (communication labelled *reqGift*); otherwise, she does not have the right to ask for it. First, it is not efficient nor economic to involve `issuer` and

---

[☆] This work is a revised and extended version of [1], prepresented in the Proceedings of the 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES). This work has been sponsored by Torino University/Compagnia San Paolo Project SALT.

E-mail address: gina.tcchen@gmail.com.

http://dx.doi.org/10.1016/j.jlamp.2015.06.003

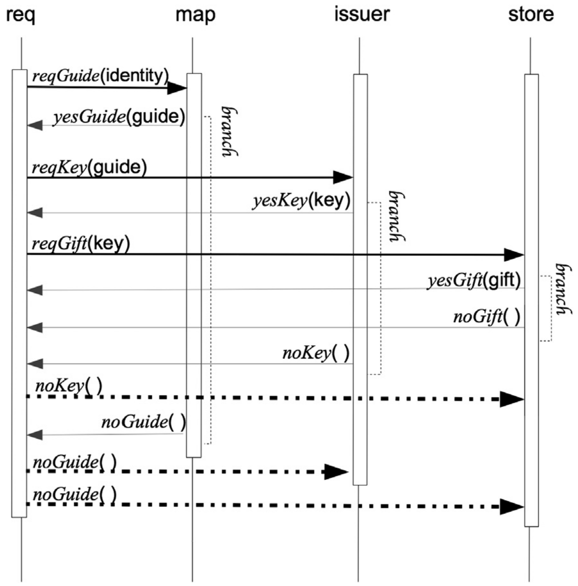

Fig. 1. Nested communications among `req` (i.e. the requester), `map`, `issuer`, and `store`.

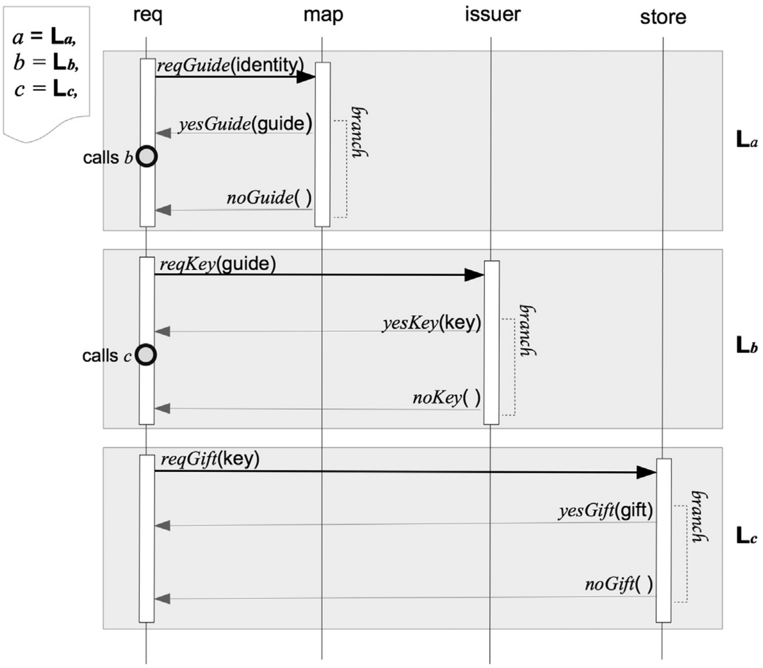

Fig. 2. Separated communications for `req`, `map`, `issuer`, and `store`.

`store` at the stage while the requester negotiates with `map` (i.e. the guide provider) because `issuer` and `store` have nothing to do at this phase. Secondly, it will be no need to invoke `issuer` and `store` if the requester fails to get the *guide* (i.e. `map` replies `req` with *noGuide*()).

The dashed arrows are needed only because both `issuer` and `store` are invoked when the communication starts. They are used to inform `issuer` and `store` to terminate. If we do not invoke `issuer` and `store` at the beginning, these interactions represented by dashed lines are *redundant*: they become unnecessary. If `issuer` is only invoked as the requester gets a *guide* (of type guide) from `map`, and `store` is only invoked as the requester gets the *key* (of type key) from `issuer`, as shown in Fig. 2, the interactions become simpler and more readable.