Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 322 (2016) 87–102

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

An Imperative Pure Calculus[1]

Andrea Capriccioli

DIBRIS
Università di Genova
Italy

Marco Servetto[2]

Victoria University of Wellington
New Zealand

Elena Zucca[3]

DIBRIS
Università di Genova
Italy

Abstract

We present a simple calculus where imperative features are modeled by just rewriting source code terms, rather than by modifying an auxiliary structure which mimics physical memory. Formally, this is achieved by the block construct, introducing local variable declarations, which also plays the role of store when such declarations have been evaluated. In this way, we obtain a language semantics which is more abstract, and directly represents at the syntactic level constraints on aliasing, allowing simpler reasoning about related properties. We illustrate this possibility by a simple extension of the standard type system which assigns a `capsule` tag to expressions that will reduce to (values representing) isolated portions of store.

Keywords: operational semantics, imperative calculus, aliasing

1 Introduction

Traditional execution models for imperative languages use an auxiliary structure, called *memory* or *store*, which is a mathematical abstraction of the physical memory, and is typically a map from *locations* (modeling memory addresses) into storable values. Locations are a runtime notion, and their names are globally available, that

[1] This work has been partially supported by MIUR CINA - Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

[2] Email: marco.servetto@ecs.vuw.ac.nz

[3] Email: elena.zucca@unige.it

http://dx.doi.org/10.1016/j.entcs.2016.03.007

is, memory is flat, whereas *variables* are a language notion, and their names obey scoping rules (shadowing) and α-conversion.

In this paper, we propose an alternative, more abstract, execution model which is a *pure calculus*. That is, execution is modeled by just rewriting source code terms, in the same way lambda calculus models functional languages.

The following is an example of reduction sequence in the calculus, where we emphasize at each step the redex which is reduced.

```
D z=new D(0)   C x=new C(z,z)   [C y=x]   D w=new D(y.f1.f+1) x.f2=w  x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D( [x.f1] .f+1) x.f2=w x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D( [z.f] +1) x.f2=w  x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D( [0+1] ) x.f2=w  x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D(1)    [x.f2=w]  x ⟶
D z=new D(0)   C x=new C(z,w)   D w=new D(1)    x
```

The main idea is to use local variable declarations, as in the `let` construct, to directly represent memory. That is, a declared variable is not replaced by its value, as in standard `let`, but the association is kept and used when necessary. [4]

The calculus is designed with an object-oriented flavour [5], inspired to Featherweight Java [14]. That is, assuming a program (class table) where class `C` has two fields `f1` and `f2` of type `D`, and class `D` has an integer field `f`, in the initial term the first two declarations can be seen as a store which associates to `z` an object of class `D` whose field contains `0`, and to `x` an object of class `C` whose two fields contains (a reference to) the previous object. The first reduction step eliminates an alias, by replacing occurrences of `y` by `x`. The next three reduction steps compute `x.f1.f+1`, by performing two field accesses and one sum. The last step performs a field assignment. The final result of the evaluation is an object of class `C` whose fields contain two objects of class `D`, whose field contains `0` and `1` field, respectively. As usual, references in the store can be mutually recursive, as in the following example, where we assume a class `B` with a field of type `B`.

```
B x= new B(y)   B y= new B(x)   y
```

In the examples until now, memory is flat, as it usually happens in models of imperative languages. However, in our calculus, we are also able to represent a hierarchical memory, as shown in the example below, where we assume a class `A` with two fields of type `B` and `D`, respectively.

```
D z= new D(0)
A w= ( B x= new B(y)   B y= new B(x)   A u= new A(x,z)   u)
w
```

Here, the store associates to `w` a block introducing local declarations, that is, in turn

[4] As it happens, with different aims and technical problems, in cyclic lambda calculi [4,3], see the Conclusion for more comments.

[5] This is only a presentation choice: all the ideas and results of the paper could be easily rephrased, e.g., in a ML-like syntax with data type constructors and reference types.