The Sesame Specification

Jonathan Busch & Günthner

Summer 25

1 Introduction

This document specifies the Bert language, associated tools and the runtime implementation of it.

2 The Language

The specification of syntax and of semantics go here.

3 The Implementation

3.1 Gabage Collection

On the language level the garbage collection will resemble the popular Mark-and-Sweep schemes used in f.e. Java. However in the implementation and in the code emitted by the compiler there is a lot of optimization possibe.

3.1.1 Avoiding Mark-and-Sweep

The difficulty in garbage collection lies in finding cycles in the reference graph. Now parts of this process can be done at compile time on the type graph. For this, some formalism will prove useful: Denote the reference digraph (w. self-reference) by (O, r) and the type digraph by (T, r'). Now the canonical projection map π will be a graph homomorphism:

$$\pi : O \to T$$

This sends and object of type t to the type t and an arrow (alt. a reference) (a, b) with objects of types t_a, t_b to (t_a, t_b).

Now we will introduce standard graph operations:
For $o \in O$, $P(o)$ $(N(o))$ denotes the previous (next) elements for o. Formally:

$$P(o) \coloneqq \{ x \in O \mid (x, o) \in r \}$$
$$N(o) \coloneqq \{ x \in O \mid (o, x) \in r \}$$

For reachability we must define the root set, the set of all elements which are by definition reachable, for example if they are stored in a global variable (not applicable for Bert) or if they represent a stack context.

Remark 1. *Why do we have objects representing stack contexts? They ease the fomalisation of garbage collection.*

This set we will denote by root. Now define for a Graph (G, e) and a subset of nodes $X \subset G$ the set of nodes reachable starting from X:

$$\begin{aligned} X &\subset \mathrm{reach}(X) \\ N(\mathrm{reach}(X)) &\subset \mathrm{reach}(X) \end{aligned}$$

The problem of garbage collection can now be stated as determining the set $O \setminus \mathrm{reach}(\mathrm{root})$. One classic and performant solution is tracking the indegree of each node $n \in O$:

$$\mathrm{indeg}(n) \coloneqq |\{\, n' : (n', n) \in r \,\}|$$

This method is called reference counting and if $\mathrm{indeg}(o) = 0$ for $o \in O$, then we deallocate o. This of couse has the problem of cycles, is however performant, instant and distributes the load of GC.

This is where the type graph T comes in. We can formulate the following lemma:

Lemma 1. *If $o_1, \cdots, o_n \in O$ is a cycle in O then $\pi(o_1), \cdots, \pi(o_n) \in T$ is a cycle in T.*

Now we can partition the type graph into the set of nodes that are (non-trivially) reachable from themselves (the problematic nodes P) and the rest (the fine nodes). This partition also extends to the objects via taking the type of each object (using π).

We can now define two new types of node-degree: The problem degree $\mathrm{pdeg}(o)$, the indeg counting only from problematic objects and $\mathrm{fdeg}(o) = \mathrm{indeg}(o) - \mathrm{pdeg}(o)$ counting only from fine objects.

Method 1. *At all times and for all objects we keep track of* $\mathrm{fdeg}(o)$ *and* $\mathrm{pdeg}(o)$*. If both are* 0*, the object is no longer referenced and can be deleted. However if only* $\mathrm{fdeg}(o) = 0$ *then we apply another method to detect whether the object is garbage.*

3.1.2 Something something Mark-and-Sweep

This will either specify the implementation of Mark-and-Sweep or an aternative scheme.

4 Tools

Here we specify usage of the `ernie` command and others.