# Mini Projects MP2
# Garbage Collection

15-814: Types and Programming Languages
Frank Pfenning

Due Friday, December 11, 2020
250 points

Please pick one among the following two mini-projects and hand in your PDF writeup code in Canvas. As for the first set of mini-projects, you may do this project by yourself or team up with a partner.

As with homework, you are allowed to use online resources such as papers and technical reports, but you must give credit to the sources you consult.

These mini-projects explore generalizations and extensions of the statics and dynamics of computation with an explicit store and shared memory concurrency, as presented in Lectures 19–23.

In general, we see the dynamics using cell and proc semantic objects as a description of the *runtime system* that supports execution of programs while remaining at a high level of abstraction. As such, it is a suitable framework for describing, for example, garbage collection or call-by-need. For an actual implementation further refinements would of course be needed.

As a starting reference semantics for these mini-projects, please see the Lecture 20 Rule Set for ordinary processes, slightly streamlined from the rules in the actual Lecture 20, and the Lecture 23 Rule Set for linearly typed processes. We have purposely omitted recursion to keep matters simple, but see Exercises L21.1 and L21.2 for two ways to integrate them.

Most languages nowadays (and particularly those that are type-safe, that is, satisfy preservation and progress) require some form of garbage collection to reclaim space for cells no longer relevant to the outcome of a computation. In these mini-projects we explore reference-counting and mark-and-sweep garbage collection for our concurrent language.

## 1 Reference-Counting Garbage Collection

*Reference-counting garbage collection* is a technique where we maintain a counter with every relevant semantic object that tracks the number of references to that object. This technique is related to *linearity* which ensures uniqueness of reference (see Lecture 23). In a linear system, the reference count for any cell would be 1. Here we want to extend the underlying idea to arbitrary well-typed processes. We suggest you carefully read all parts before attempting to solve any of them because the goals expressed in later parts may affect your earlier design.

For parts 1–9, restrict yourself to pairs ($\tau_1 \times \tau_2$), unit (1), and functions ($\tau_1 \to \tau_2$).

1. Rewrite the *typing rules* for processes $P$ in a form suitable for reference counting. You may modify or extend the language of processes as you see fit. Let's call this (possibly slightly extended) language and type system *quasi-linear*. It should be possible for every ordinary process to have at least one quasi-linear translation, but you do not need to prove this.

   [Hint: We suggest each variable is created with a unique reference, coupled with language constructs that effectively increment or decrement reference counts.]

2. Write quasi-linear processes with the following types and destination $d$.

   (a) $(\alpha \times \beta) \to (\beta \times \alpha)$

   (b) $\alpha \to (\alpha \times \alpha)$

   (c) $(\alpha \times \alpha) \to \alpha$

   (d) $((\alpha \to \beta) \times (\alpha \to \gamma)) \to (\alpha \to (\beta \times \gamma))$

   (e) $(\alpha \to (\beta \times \gamma)) \to ((\alpha \to \beta) \times (\alpha \to \gamma))$

3. In preparation for the dynamics, define the semantic objects that form quasi-linear configurations. You may modify the cell and proc objects and/or define new objects as you see fit. State the informal interpretation of your objects.

4. Provide the dynamics for quasi-linear configurations using multiset rewriting.

5. Design typing rules for quasi-linear configurations that contain sufficient information to guarantee the correctness of reference counting.

6. State the properties of *preservation* and *progress*. You do not need to prove them, but you need to take care that (a) preservation contains enough information to maintain quasi-linear typing, and (b) progress takes advantage of quasi-linear typing.

7. Show a counterexample to *progress* for a configuration that appears to be well-typed with ordinary types, but whose reference counting information is incorrect, or conjecture no such counterexample exists.

8. Show a counterexample to *preservation* as in the previous question or conjecture no such counterexample exists.

9. Assume the initial configuration contains a single process $\cdot \vdash P :: (d_0 : \tau)$ and explain in what sense the preservation and progress properties guarantee garbage collection. If you can, capture this property technically in the form of a theorem or conjecture.

10. Add rules for sums ($\sum_{i \in I}(i : \tau_i)$), lazy records ($\&_{i \in I}(i : \tau_i)$), and recursive types ($\rho\alpha.\,\tau$) to the statics and dynamics. All properties, including preservation and progress, should continue to hold.

## 2 Mark-and-Sweep Garbage Collection

Reference counting, as explored in the first mini-project is naturally concurrent but it does not work well when there are circular references (such as could arise, for example, in the modeling of recursion as in Exercise L21.1). Another approach is a so-called mark-and-sweep collector.

In order to describe a mark-and-sweep collector we need to refine the memory model so that addresses are natural numbers. Cells are allocated from a *free list* which is modeled as a linked list of cells. When the allocator runs out of memory (that is, the free list is empty), we scan the configuration and *mark* all cells and processes that are reachable from an initial *root* $d_0$. We then *sweep* memory and collect all unmarked cells into the new free list and resume.

The key to this project is to redefine the dynamics in the context of the multiset rewriting style we have adopted so that (a) it behaves just as before when no garbage collection is needed, and (b) it implements the mark-and-sweep garbage collection behavior described above when it runs out memory. Regarding part (a): the objects may look slightly different, but the correspondence should be clear. Regarding part (b): the actions of the garbage collector should be described by multiset rewriting rules, likely using additional semantic objects to capture the state of the collector. It is important that the rules are not "clairvoyant" and do not have complex non-local side conditions. For example, marking a cell under the "side condition" that it "*is reachable from the root*" would beg the computational questions and is not an acceptable part of a rule.

You may restrict yourself to pairs ($\tau_1 \times \tau_2$), unit (1), and functions ($\tau_1 \to \tau_2$).

1. As a preliminary step, add semantic objects of the form $\mathsf{cell}\ c\ \mathsf{alloced}$ that expresses that a cell has been *allocated but not yet written to*. Additionally, we model the free list with objects $\mathsf{cell}\ c\ (\mathsf{next}\ c')$ and $\mathsf{cell}\ c\ \mathsf{null}$. Finally, we have an object $\mathsf{free}\ c$ where $c$ is the beginning of the free list. Reformulate the dynamics (from page 3 of the Lecture 20 Rule Set) to account for these new semantics objects. For the moment, just identify the situation when allocation fails—you will add an explicit rule for this situation later in part 7.

2. Previously, cell addresses were abstract, and typing just worked with an *invariant* (for judgments also called a *presupposition*) that all addresses in a configuration were distinct. In order to describe a mark-and-sweep collector the addresses are now natural numbers $0, 1, \ldots, \mathit{size}-1$ where *size* is a fixed constant.

   Informally state the invariants of the configuration with the new kinds of semantic objects you need to maintain for computation to be meaningful (including typing).

3. Express each of your invariants mathematically, using inference rules where appropriate and general mathematical assertions otherwise. For example, one of your invariants might be "*For all cells* $\mathsf{cell}\ c\ \_ \in \mathcal{C}$ *it is the case that* $0 \leq i < \mathit{size}$". You do not need to rewrite the typing rules for processes—just describe if and how they would be updated.

4. Pick *one* of your invariants and prove a *preservation property*: if the configuration matching the left-hand side of each rule satisfies it, then configuration resulting from the step will also satisfy it.

5. Carefully state the analogue of the *progress theorem* for your rules so far. You do not need to prove it.

6. The initial configuration should contain proc $0\ P$, cell $0$ alloced for some $\cdot \vdash P :: (0 : \tau)$ with all other cells in the free list. Formally define an initial configuration and verify that it satisfies your invariants from part 3.

7. Now specify the behavior of the *mark* phase of the algorithm via multiset rewriting rules. You initiate this phase when allocation fails (see part 1) by adding to the configuration a new form of object that represents the actions of the garbage collector. You may also add new kinds of objects or modify the form of objects introduced earlier. In the latter case, you need to restate the rules of the process dynamics. Explicitly state when the *mark* phase terminates.

8. State the properties that are guaranteed for the configuration when the *mark* phase terminates, the more rigorous the better.

9. Specify the behavior of the *sweep* phase of the algorithm using multiset rewriting rules. Explicitly state when this phase terminates. Assuming the properties guaranteed by the *mark* phase when it terminates, the *sweep* phase should have restored the invariants from part 3.

10. Tie together computation of user processes, mark, and sweep. State the end-to-end progress property that the combination of all rules enjoys.

11. Informally explain the end-to-end correctness property relating the garbage-collected dynamics to the reference dynamics.

12. What (if any) changes would be required to support sums, lazy records, and recursive types?

13. Does your algorithm garbage-collect processes? If not, describe how it might be modified to do so. Discuss the consequences of such a modification for end-to-end correctness (part 11).

14. Is it necessary to stop all processes during the mark and sweep phases, or is it possible to conduct garbage collection *concurrently* with the execution of the program? Discuss to what extent the garbage collector is concurrent, or could be modified to operate concurrently.