Lecture Notes on Garbage Collection

15-417/817: HOT Compilation
Frank Pfenning

Lecture 19
April 1, 2025

1 Introduction

Already at the inception of linear logic, Girard and Lafont [1987] developed a semantics that explicates garbage collection. The typing rules were intuitionistic and using recursive types rather than the exponential modality that featured in Girard's more definitive, classical account [1987].

We can account for garbage collection (or its lack) for adjoint natural deduction only in a somewhat approximate manner [Jang et al., 2024] because the heap is not explicitly represented. Sax, however, makes it quite explicit in a simple and elegant manner. This was exploited by Gupta [2023] who analyzed garbage collection for a mixed linear/nonlinear language based on Sax, also considering so-called multilinear types. The basic language can be described in the adjoint setting as one with two modes U and L with $\mathsf{U} > \mathsf{L}$ and $\sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$ and $\sigma(\mathsf{L}) = \{\,\}$. An important aspect of the dynamic was *reference counting*, which has gained currency through it usage in several widely used languages including Rust. It should be noted, however, that in the parallel setting there is a price to pay for reference counting, namely that it turns a reader (which normally does not need to worry about race conditions) into a writer when it has to increment the reference count. This increment is potentially subject to race conditions.

In this lecture we cover some basics, first for a copying collector then for reference counting memory management.

2 Revisiting the Dynamics

We begin by revisiting the dynamics for Sax in light of the adjoint refinements. We recall it from Lecture 7 for the linear case, with a small update for the shifts. However, this dynamics is only

correct if all modes are linear; we will update it shortly.

$$
\begin{array}{llll}
\text{Commands} & P, Q & ::= & \mathbf{write}\ c\ S \\
& & \mid & \mathbf{read}\ c\ S \\
& & \mid & \mathbf{cut}\ (x : A)\ P\ Q \\
& & \mid & \mathbf{id}\ a\ b \\
& & \mid & \mathbf{call}\ F\ d\ b_1\ \ldots\ b_n \\
\text{Storables} & S & ::= & v \mid K \\
\text{Small values} & v & ::= & (a, b) \qquad (\otimes, \rightarrow) \\
& & \mid & (\,) \qquad (\mathbf{1}, [\bot]) \\
& & \mid & k(a) \qquad (\oplus, \&) \\
& & \mid & \langle a \rangle \qquad (\downarrow, \uparrow) \\
\text{Continuations} & K & ::= & (x, y) \Rightarrow P \qquad (\otimes, \rightarrow) \\
& & \mid & (\,) \Rightarrow P \qquad (\mathbf{1}, [\bot]) \\
& & \mid & \{\ell(x_\ell) \Rightarrow P_\ell\}_{\ell \in L} \qquad (\oplus, \&) \\
& & \mid & \langle x \rangle \Rightarrow P \qquad (\downarrow, \uparrow)
\end{array}
$$

Passing a small value to a continuation is now a more symmetric relationship, but the essence is the same as before.

$$v \bowtie K = K \bowtie v = v \rhd K$$

where

$$
\begin{array}{lcllll}
(a, b) & \rhd & (x, y) \Rightarrow P(x, y) & = & P(a, b) & \\
(\,) & \rhd & (\,) \Rightarrow P & = & P & \\
k(a) & \rhd & \{\ell(x_\ell) \Rightarrow P_\ell(x_\ell)\}_{\ell \in L} & = & P_k(a) & (k \in L) \\
\langle a \rangle & \rhd & \langle x \rangle P(x) & = & P(a) &
\end{array}
$$

and

$$
\begin{array}{lcll}
\mathsf{proc}\ (\mathbf{write}\ c\ S) & \longrightarrow & \mathsf{cell}\ c\ S & \\
\mathsf{cell}\ c\ S, \mathsf{proc}\ (\mathbf{read}\ c\ S') & \longrightarrow & \mathsf{proc}\ (S \bowtie S') & \\
\mathsf{proc}\ (\mathbf{cut}\ (x : A)\ P(x)\ Q(x)) & \longrightarrow & \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a\ \text{fresh}) \\
\mathsf{cell}\ b\ S, \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}\ a\ S & \\
\mathsf{proc}\ (\mathbf{call}\ F\ c\ \overline{b}) & \longrightarrow & \mathsf{proc}\ P(c, \overline{b}) & \text{where } F(x, \overline{y}) = P(x, \overline{y})
\end{array}
$$

Recall that we have a configuration consisting of executing processes and memory cells.

$$\text{Configurations} \quad \mathcal{C} \quad ::= \quad \mathsf{proc}\ P \mid \mathsf{cell}\ a\ S \mid \cdot \mid \mathcal{C}_1, \mathcal{C}_2$$

Also recall that multiset rewriting rules of the form that specifies the semantics are interpreted by matching the left-hand side against a part of the configuratin and replacing it by the right-hand side. This means that the dynamics shown above only work for the linear or affine cases. If a mode admits contraction, a cell might have multiple readers, so removing it when it is read will lead to a failure of the progress theorem.

In order to account for this, we introduce another aspect of multiset rewriting. Components of a configuration can be *ephemeral* (that is, they are removed during an application of a rewrite rule) and *persistent* (that is, they remain in the configuration). The only semantic objects that can be persistent are cells, and we write $!\mathsf{cell}\ a\ S$ for them. The notation using the exponential from linear logic is not a coincidence (see Cervesato and Scedrov [2009], Watkins et al. [2002], Cervesato et al. [2002] for its origins).

Before rewriting the rules we make one further change: when a fresh cell is allocated we actually create an object $\mathsf{cell}\ a\ \square$ do indicate an unwritten cell. In the case of a parallel implementation, this can be "seen" by a reading thread, which may then have to block until the cell is written.

$$\text{Configurations}\quad \mathcal{C} \quad ::= \quad \mathsf{proc}\ P \mid \mathsf{cell}\ a\ \square \mid \mathsf{!cell}\ a\ S \mid \cdot \mid \mathcal{C}_1, \mathcal{C}_2$$

$$\begin{array}{lcll}
\mathsf{cell}\ c\ \square, \mathsf{proc}\ (\mathbf{write}\ c\ S) & \longrightarrow & \mathsf{!cell}\ c\ S & \\
\mathsf{!cell}\ c\ S, \mathsf{proc}\ (\mathbf{read}\ c\ S') & \longrightarrow & \mathsf{proc}\ (S \bowtie S') & \\
\mathsf{proc}\ (\mathbf{cut}\ (x:A)\ P(x)\ Q(x)) & \longrightarrow & \mathsf{cell}\ a\ \square, \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a\ \text{fresh}) \\
\mathsf{!cell}\ b\ S, \mathsf{cell}\ a\ \square, \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{!cell}\ a\ S & \\
\mathsf{proc}\ (\mathbf{call}\ F\ c\ \overline{b}) & \longrightarrow & \mathsf{proc}\ P(c, \overline{b}) & \text{where } F(x, \overline{y}) = P(x, \overline{y})
\end{array}$$

At this point, unfortunately, we have thrown out the baby with the bath water! In the adjoint setting, modes that do not admit contraction (that is, linear or affine modes) nevertheless will have persistent cells. In order to capture this distinction we annotate addresses with their mode and write

$$!_m\mathsf{cell}\ c_m\ S = \begin{cases} \mathsf{!cell}\ c_m\ S & \text{provided } \mathsf{C} \in \sigma(m) \\ \mathsf{cell}\ c_m\ S & \text{provided } \mathsf{C} \notin \sigma(m) \end{cases}$$

Configurations and transitions then track modes.

$$\text{Configurations}\quad \mathcal{C} \quad ::= \quad \mathsf{proc}\ P \mid \mathsf{cell}\ a_m\ \square \mid !_m\mathsf{cell}\ a_m\ S \mid \cdot \mid \mathcal{C}_1, \mathcal{C}_2$$

$$\begin{array}{lcll}
\mathsf{cell}\ c_m\ \square, \mathsf{proc}\ (\mathbf{write}\ c_m\ S) & \longrightarrow & !_m\mathsf{cell}\ c_m\ S & \\
!_m\mathsf{cell}\ c_m\ S, \mathsf{proc}\ (\mathbf{read}\ c_m\ S') & \longrightarrow & \mathsf{proc}\ (S \bowtie S') & \\
\mathsf{proc}\ (\mathbf{cut}\ (x:A_m)\ P(x)\ Q(x)) & \longrightarrow & \mathsf{cell}\ a_m\ \square, \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a\ \text{fresh}) \\
!_m\mathsf{cell}\ b_m\ S, \mathsf{cell}\ a_m\ \square, \mathsf{proc}\ (\mathbf{id}\ a_m\ b_m) & \longrightarrow & !_m\mathsf{cell}\ a_m\ S & \\
\mathsf{proc}\ (\mathbf{call}\ F\ c\ \overline{b}) & \longrightarrow & \mathsf{proc}\ P(c, \overline{b}) & \text{where } F(x, \overline{y}) = P(x, \overline{y})
\end{array}$$

In this dynamics, cells or linear or affine mode are deallocated as soon as they are read, while others remain. Some of these may no longer be accessible from the initial destination that will hold the result of the overall computation. A natural garbage collector will therefore start at the initial destination and either move the reachable cells to a new heap (a *copying collector*) or mark them as being in use (a *mark-and-sweep collector*). One difficulty is what to do with running processes. In a sequential setting, any heap address directly referenced from the runtime stack must be preserved (either by copying or by marking). In the parallel setting there may be thread-local or task-local stack that track this information. Gupta [2023] maintained an explicit environment η for each running parallel processes. This environment can be used to track the "live" locations. One can also run a garbage collector concurrently with the program to mitigate efficiency issues.

At the level of the semantics above, we need an operation $\mathbf{used}\, P$, and similarly for continuations $\mathbf{used}\, K$ and small values $\mathbf{used}\, v$ that adds the addresses used in P or K to the set of reachable addresses.

One question is what to do about processes whose destination is no longer being read. For example, we might have a process computing P with destination d, but there is no longer a reader for d. With futures, such processes nevertheless need to be computed down to final configurations consisting only of cells. Under a different semantics like that for speculations Harper [2016] it could be possible to garbage-collect and kill such processes.

3 Statics with Reference Counting

An alternative to the traditional kind of garbage collection hinted at in the previous section, we can also consider reference counting. The idea is that each cell maintains a count of how many references there are to it. A cell is deallocated if its reference count becomes zero. This requires the program to take full account of all references to the heap. This can be accomplished using explicit structural rules as mapped out in the last lecture. Another obstacle are circular data structures, but in our pure language they cannot arise. Finally, we need to consider interactions with optimizations such as reuse. These can be managed because they are detected and maintained statically.

In the context of adjoint ND and Sax, a key step is to make structural rules explicit. We do this at the level of Sax, because it is a concern about the heap, which is explicit in Sax but not in ND. The goal is to define a typing for Sax in which structural rules are explicit, and variables (that stand for addresses) are treated entirely linearly everywhere. Our strategy to get there is to exploit the additive formulation of typing. Intuitively, whenever a variable x (legally) occurs on both sides of join operation $\Xi_1 \,;\, \Xi_2$ we should create two aliases for this variable. This is the only place where aliasing will be required. The possibility of weakening is tested in two places: when computing $\Xi \setminus x_m$ where x_m is not in Ξ and for $\Xi_1 \sqcup \Xi_2$ when a variable x_m is used in one context but not the other.

The new constructs in our language are $\mathbf{alias}\ x\ \mathbf{as}\ (y, z)\ ;\ P$ and $\mathbf{drop}\ x\ ;\ P$. We check them with the following rules, which are written assuming precise linear contexts Δ (regardless of the modes in them).

$$\frac{(\mathsf{C} \in \sigma(m)) \quad \Delta, y : A_m, z : A_m \vdash P :: \delta}{\Delta, x : A_m \vdash \mathbf{alias}\ x\ \mathbf{as}\ (y, z)\ ;\ P :: \delta}\ \mathsf{alias}$$

$$\frac{(\mathsf{W} \in \sigma(m)) \quad \Delta \vdash P :: \delta}{\Delta, x : A_m \vdash \mathbf{drop}\ x\ ;\ P :: \delta}\ \mathsf{drop}$$

We call the two systems "*adjoint Sax with implicit structural rules*" (what we have studied so far) and "*adjoint Sax is explicit structural rules*" (what we create for the purpose of reference counting memory management). We do not fully formalize how to translate the implicit to the explicit systems, but proceed by showing prototypical examples and informal explanations. A more rigorous treatment can be found in the work of Gupta [2023] and some further references cited therein.

We start with the identity.

$$\frac{b : A_m \in \Gamma}{\Gamma \vdash \mathbf{id}\ a\ b :: (a : A_m)\ /\ (b : A_m)}\ \mathsf{id}$$

Here, no translation is necessary. The command $\mathbf{id}\ a\ b$ is linear in b independently of the mode m.

$$\frac{a : A_m \in \Gamma \quad b : B_m \in \Gamma}{\Gamma \vdash \mathbf{write}\ c\ (a, b) :: (c : A_m \otimes B_m)\ /\ (a : A_m)\ ;\ (b : B_m)}\ \otimes X$$

Here, we see a use of the join operator. This means, if $a = b$ (which is possible if $\mathsf{C} \in \sigma(m)$), then we need to apply contraction just before. So the tranlsation is just $\mathbf{write}\ c\ (a, b)$ if $a \neq b$ and is $\mathbf{alias}\ a\ (a_1, a_2)\ ;\ \mathbf{write}\ c\ (a_1, a_2)$ if $a = b$.

$$\frac{c : A_m \otimes B_m \in \Gamma \quad (m \geq r) \quad \Gamma, x : A_m, y : B_m \vdash P :: (d : D_r)\ /\ \Xi}{\Gamma \vdash \mathbf{read}\ c\ (x, y) \Rightarrow P :: (d : D_r)\ /\ (\Xi \setminus x_m \setminus y_m)\ ;\ (c : A_m \otimes B_m)}\ \otimes L$$

First, we recall that $\Xi \setminus x_m$ removes x from Ξ if it occurs. If it doesn't occur, then it is an error unless $\mathsf{W} \in \sigma(m)$. This means we have to make a first transformation to

$$\mathbf{read}\ c\ (x, y) \Rightarrow \mathbf{drop}\ x\ ;\ P$$

if x does not occur in Ξ. Similarly, we would drop y if it is not in Ξ, and both if neither occurs in Ξ. We want to drop unused variables as early as possible, so we would do this at the beginning of P rather than the end.

In addition, we also have to consider the possible occurrences of c in Ξ, which means that c was used again in P. In that case, we have to alias c before the read, as in

$$\mathbf{alias}\ c\ (c_1, c_2)\ ;\ \mathbf{read}\ c_1\ (x, y) \Rightarrow [c_2/c]P$$

where $[c_2/c]P$ substitutes the fresh c_2 for c in P. Here, the strategy is to alias as late as possible (just before we need to share), again for simplicity and efficiency.

For pattern matching, we give here the binary version where $A \oplus B \triangleq \oplus\{\pi_1 : A, \pi_2 : B\}$. Then we have two axioms, neither of which has to be concerned with weakening or contraction.

$$\frac{a : A_m \in \Gamma}{\Gamma \vdash \mathbf{write}\ c\ \pi_1(a) :: (c : A_m \oplus B_m)\ /\ (a : A_m)}\ \oplus X_1$$

$$\frac{b : B_m \in \Gamma}{\Gamma \vdash \mathbf{write}\ c\ \pi_2(b) :: (c : A_m \oplus B_m)\ /\ (b : B_m)}\ \oplus X_2$$

The left rule introduces some new considerations.

$$\frac{c : A_m \oplus B_m \in \Gamma \quad m \geq r \quad \Gamma, x : A_m \vdash P_1 :: (d : D_r)\ /\ \Xi_1 \quad \Gamma, y : B_m \vdash P_2 :: (d : D_r)\ /\ \Xi_2}{\Gamma \vdash \mathbf{read}\ c\ (\pi_1(x) \Rightarrow P_1 \mid \pi_2(y) \Rightarrow P_2) :: (d : D_r)\ /\ (\Xi_1 \setminus x_m) \sqcup (\Xi_2 \setminus y_m)}\ \oplus L$$

We see that we may need to explicitly drop x_m from P_1 and y_m from P_2. Then we take the least upper bound of the results. For each variable w_k that occurs in Ξ_1 but not Ξ_2, we have to drop w_k explicitly (ideally at the beginning of P_2). Typechecking fails (or should have failed before) if w_k does not permit weakening. The symmetric remark applies for variables in Ξ_2 but not Ξ_1. Contraction (that is, aliasing) is not involved in this rule.

4 Dynamics with Reference Counting

With the above transformations we have arrived at explicit Sax, where all variables are used linearly, with explicit appeals to aliasing (to model contraction) and dropping (to model weakening).

How do we reflect this change back into the dynamics with explicit reference counting? We annotate each cell in the dynamics with a reference count. These reference counts are not "transitive" but only count direct references. For example, there might be 5 references to the head of a list, but only 1 for all the cells in the tail of the list.

Because typing is now essentially linear, all cells in the dynamics are ephemeral and carry a reference count $n \geq 0$.

$$\text{Configurations}\quad \mathcal{C} \quad ::= \quad \mathsf{proc}\ P \mid \mathsf{cell}_n\ a\ \Box \mid \mathsf{cell}_n\ a\ S \mid \cdot \mid \mathcal{C}_1, \mathcal{C}_2$$

Creating an alias is quite straightforward. It doesn't matter if a cell has been written or not—we can always increase its reference count. We use the notation $\mathsf{cell}_n\ c\ S^\square$ to indicate either $\mathsf{cell}_n\ c\ S$ or $\mathsf{cell}_n\ c\ \square$. We also introduce a new semantic object $\mathsf{inc}\ c$ that increment the reference count of c.

$$\begin{array}{lcl} \mathsf{proc}\ (\mathbf{alias}\ c\ (x, y)\ ;\ P(x, y)) & \longrightarrow & \mathsf{inc}\ c, \mathsf{proc}\ P(c, c) \\ \mathsf{cell}_{n+1}\ c\ S^\square, \mathsf{inc}\ c & \longrightarrow & \mathsf{cell}_{n+2}\ c\ S^\square \end{array}$$

Dropping a reference to a cell is more complex, because the reference count might hit zero. Using another new semantic object $\mathsf{dec}\ c$, we specify

$$\mathsf{proc}\ (\mathbf{drop}\ c\ ;\ P) \quad \longrightarrow \quad \mathsf{dec}\ c, \mathsf{proc}\ P$$

When the reference count hits 0, we need to deallocate the cell, but we also need to decrease the reference count for everything that the storable in the continuation refers to. We write $\mathbf{free}\ S$ for the free addresses in a storable S. For continuations, this should **not** include the address written by S. In a realistic implementation, we'd have to make sure this can be computed efficiently, especially if S is a continuation K. This was partially addressed by Gupta [2023] using explicit environments.

We add a new semantic object $\mathsf{dec}\ c$ with the intent that it should decrease the reference count of c.

$$\begin{array}{lcl} \mathsf{cell}_{n+2}\ c\ S^\square, \mathsf{dec}\ c & \longrightarrow & \mathsf{cell}_{n+1}\ c\ S^\square \\ \mathsf{cell}_1\ c\ S, \mathsf{dec}\ c & \longrightarrow & \{\mathsf{dec}\ a \mid a \in \mathbf{free}\ S\} \end{array}$$

Note that $\mathsf{cell}_1\ c\ \square, \mathsf{dec}\ c$ blocks: we shouldn't deallocate a cell that hasn't been written yet because its writer would no longer have a valid destination. It can, however, be deallocated as soon as it is written.

Continuing, the rule for writing is straightforward.

$$\mathsf{cell}_{n+1}\ c\ \square, \mathsf{proc}\ (\mathbf{write}\ c\ S) \quad \longrightarrow \quad \mathsf{cell}_{n+1}\ c\ S$$

Any references in the storable S are transferred from the process to the cell it is written to. And since the writer's reference does not count, the reference count for the cell c remains unchanged.

For reading, if this is the last reference to a cell, the reference to the free addresses in S is transferred from the cell to the continuation process. Otherwise, we have to increment the reference count of all free addresses in S, because the cell c persists, and the continuation process now also has a reference.

$$\begin{array}{lcl} \mathsf{cell}_1\ c\ S, \mathsf{proc}\ (\mathbf{read}\ c\ S') & \longrightarrow & \mathsf{proc}\ (S \bowtie S') \\ \mathsf{cell}_{n+2}\ c\ S, \mathsf{proc}\ (\mathbf{read}\ c\ S') & \longrightarrow & \mathsf{cell}_{n+1}\ c\ S, \mathsf{proc}\ (S \bowtie S'), \{\mathsf{inc}\ a \mid a \in \mathbf{free}(S)\} \end{array}$$

For cut, we introduce a fresh cell with initial reference count 1.

$$\mathsf{proc}\ (\mathbf{cut}\ x\ P(x)\ Q(x)) \quad \longrightarrow \quad \mathsf{cell}_1\ a\ \square, \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) \quad (a\ \text{fresh})$$

Identity behaves like a read followed by a write.

$$\begin{array}{lcl} \mathsf{cell}_1\ b\ S, \mathsf{cell}_n\ a\ \square, \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}_n\ a\ S \\ \mathsf{cell}_{m+2}\ b\ S, \mathsf{cell}_n\ a\ \square, \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}_{m+1}\ b\ S, \mathsf{cell}_n\ a\ S, \{\mathsf{inc}\ c \mid c \in \mathbf{free}(S)\} \end{array}$$

Finally, a call is neutral regarding the reference count because the references $\overline{b}$ are passed to the body of the invoked procedure.

$$\mathsf{proc}\ (\mathbf{call}\ F\ c\ \overline{b}) \quad\longrightarrow\quad \mathsf{proc}\ P(c, \overline{b}) \qquad \text{where } F(x, \overline{y}) = P(x, \overline{y})$$

In comparison to the dynamics from Section 2 we lose something, in particular in a truly parallel semantics: reading now actually creates some race conditions because the reference count needs to be updated atomically. We delegate this to an increment process, but this is just kicking the can down the road. The rules can be simplified under a sequential semantics, and in particular the race conditions disappear. Another gain we make is that the dynamics no longer cares about modes. Cells are always ephemeral and are deallocated essentially when the are no longer referenced by the running program. However, linear and affine cells have always have reference count 1. The latter may lead to a cascading reference count decrement. The absence of circularity in the memory prevents one of the deeper problems with diverging, or keeping unreferenced memory from being deallocated.

$$\begin{array}{lcl}
\mathsf{proc}\ (\mathbf{alias}\ c\ (x, y)\ ;\ P(x, y)) & \longrightarrow & \mathsf{inc}\ c, \mathsf{proc}\ P(c, c) \\
\mathsf{proc}\ (\mathbf{drop}\ c\ ;\ P) & \longrightarrow & \mathsf{dec}\ c, \mathsf{proc}\ P \\
\mathsf{cell}_{n+1}\ c\ \square, \mathsf{proc}\ (\mathbf{write}\ c\ S) & \longrightarrow & \mathsf{cell}_{n+1}\ c\ S \\
\mathsf{cell}_1\ c\ S, \mathsf{proc}\ (\mathbf{read}\ c\ S') & \longrightarrow & \mathsf{proc}\ (S \bowtie S') \\
\mathsf{cell}_{n+2}\ c\ S, \mathsf{proc}\ (\mathbf{read}\ c\ S') & \longrightarrow & \mathsf{cell}_{n+1}\ c\ S, \mathsf{proc}\ (S \bowtie S'), \{\mathsf{inc}\ a \mid a \in \mathbf{free}(S)\} \\
\mathsf{proc}\ (\mathbf{cut}\ x\ P(x)\ Q(x)) & \longrightarrow & \mathsf{cell}_1\ a\ \square, \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) \qquad (a\ \text{fresh}) \\
\mathsf{cell}_1\ b\ S, \mathsf{cell}_n\ a\ \square, \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}_n\ a\ S \\
\mathsf{cell}_{m+2}\ b\ S, \mathsf{cell}_n\ a\ \square, \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}_{m+1}\ b\ S, \mathsf{cell}_n\ a\ S, \{\mathsf{inc}\ c \mid c \in \mathbf{free}(S)\} \\
\mathsf{proc}\ (\mathbf{call}\ F\ c\ \overline{b}) & \longrightarrow & \mathsf{proc}\ P(c, \overline{b}) \qquad \text{where } F(x, \overline{y}) = P(x, \overline{y})
\end{array}$$

$$\begin{array}{lcl}
\mathsf{cell}_{n+1}\ c\ S^{\square}, \mathsf{inc}\ c & \longrightarrow & \mathsf{cell}_{n+2}\ c\ S^{\square} \\
\mathsf{cell}_{n+2}\ c\ S^{\square}, \mathsf{dec}\ c & \longrightarrow & \mathsf{cell}_{n+1}\ c\ S^{\square} \\
\mathsf{cell}_1\ c\ S, \mathsf{dec}\ c & \longrightarrow & \{\mathsf{dec}\ a \mid a \in \mathbf{free}\ S\}
\end{array}$$

Figure 1: Summary of Reference Counting Dynamics

References

Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical frame-

work II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

Aditi Gupta. Ergometric multilinear futures. Honors thesis, Computer Science Department, Carnegie Mellon University, May 2023. URL http://www.cs.cmu.edu/~fp/courses/15417-s25/misc/Gupta23.pdf.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. Adjoint natural deduction. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*, pages 15:1–15:23, Tallinn, Estonia, July 2024. LIPIcs 299. Extended version available as https://arxiv.org/abs/2402.01428.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.