Operational Semantics of Concurrent Memory Management

Abstract

Automatic memory management (garbage collection) is one of the most well-studied subjects in computer science. Most papers on garbage collection either describe the algorithms informally at a high level or they specify and verify the algorithms precisely at a very low level of abstraction, rendering the semantics and proofs too specialized to be usable even under slight modifications to the algorithms.

In this paper, we propose techniques for defining and proving correct sophisticated collectors at the level of the programming language. The goal is to find a middle ground that allows formally specifying the key ideas of interesting concurrent garbage collection algorithms at a high level. To this end, we build on the well-known idea of separating the mutator from the garbage collector and develop a framework for specifying the operational semantics of a garbage-collected system such that the mutator can be paired up with essentially any suitably defined collector. Taking this further, we show that garbage collector can be proved correct in a similarly modular fashion by independently establishing the safety of the mutator and garbage collection algorithms.

We apply these techniques to several collectors including stop-the-world, two concurrent collectors (Baker's and Nettles-O'Toole's), and two multithreaded collectors (Doligez-Leroy-Gonthier and Manticore). The techniques make it possible to specify different collectors using a relatively small number of operational judgments and to prove them correct in a similarly compact fashion. We have mechanized in Coq the proposed specification and proof techniques and have established the memory safety of the aforementioned algorithms.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

Keywords Languages, Memory Management

1. Introduction

Garbage collection (GC) has been studied since John McCarthy introduced it in Lisp over 50 years ago [13], and since then there have been over a thousand publications on the topic [**?**]. Just considering sequential garbage collectors, many approaches have been suggested, including ideas such as mark-sweep, copying, generational, incremental, reference-counting, real-time, tagless, region-based, and replicating. When parallelism and concurrency are considered, the range of possible collectors is much larger. Indeed, only a small sample of the space has surely been studied to date.

Traditionally, these GC algorithms were described informally in English or pseudocode. More recent efforts have formally specified and mechanically verified the correctness of certain GC implementations [3, 7–11, 14, 17] (see Section 5 for more details). The proofs of correctness of these collectors are typically described at the level of machine instructions and can require thousands of lines of proof code [10]. This leaves a gap between high-level ideas in garbage collection algorithms, and proofs of correctness of specific implementations.

In this paper, we develop techniques for specifying realistic, concurrent garbage collectors at the language level by providing a precise and succinct operational account of their key ideas. The purpose is to find a middle ground between very low-level machine-specific specifications and the informal algorithmic specifications found in the literature. To reduce the work required to describe and verify individual collectors, we develop a framework that allows different collectors to be paired with different languages. We define a formal interface between the two that allows proving each correct in isolation, allowing some proof effort to be shared when verifying several collectors for a given language.

We augment a small-step operational semantics for the core of an ML-like functional language with explicit (abstract) heaps. Each step in the semantics represents an atomic action such as a beta-reduction for the mutator, or copying a single location for the collector. We model concurrency and parallelism with arbitrary non-deterministic interleavings of these steps. Correctness is then proved with respect to any interleaving of the steps.

We use the approach to model and prove safety and correctness theorems for several collectors. For sequential programs with assignment, we model two concurrent semispace copying collectors: Baker's collector [2] and Nettles and O'Toole's collector [16]. Our semantics captures the difference between the to-space invariant in Baker's collector (Baker) and the from-space invariant in the Nettles-O'Toole collector (Nettles). In the attached supplementary material, we consider a simple stop-and-copy collector for this language as well.

For multithreaded computations, we extend the language with fork and join operations, and model the Doligez-Leroy-Gonthier collector (DLG) [5, 6]. The semantics models the standard features of DLG including (1) maintaining a local heap for each thread and a global heap across all threads, (2) allocating mutable objects in the global heap, and (3) promoting the transitive closure of pointers to the global heap when they are written into the global heap or when placed in the closure of a new thread. The local heaps are collected using stop-and-copy, and the global heap is collected using concurrent mark-sweep. We also model the extension of DLG used in the Manticore collector [1].

We have fully formalized in Coq the framework for the sequential language of Section 3 (consisting of the semantics and progress and preservation proofs for the language, as well as the definitions and proofs of Section 3.3). Our Coq development also includes the definitions of the multithreaded language of Section 4 and all of the collectors for both languages, as well as sketches of the major proofs (albeit with many intermediate results assumed).

Since our collector does not go down to the level of machine instructions, there are certainly aspects of GC implementations we are not capturing. Therefore the approaches in this paper are not meant to replace more detailed verification of specific implementations, nor are they meant to replace high-level, informal descriptions of algorithms. However, we believe that viewing GC algorithms at the level of abstraction of this paper is valuable for a number of reasons:

- There may be many possible implementations of a given algorithm, and it is therefore beneficial to establish the correctness of an algorithm separately from its implementations. For example, our models of garbage collectors make clear what atomicity assumptions we make (since steps in the operational semantics are assumed atomic). It is important to recognize these assumptions in an *algorithm* before verifying that an *implementation* provides the necessary atomicity guarantees. Implementations can be verified by refining the model and proof of the algorithm, possibly in stages.
- We feel that this level of abstraction captures the essence of a particular collection strategy. For example, while the Baker and Nettles algorithms will look quite different at the implementation level, it may be said that the essential difference between them is that Baker uses a to-space invariant and Nettles a from-space invariant, and indeed our models of the two collectors are quite similar except for this essential difference.
- We can easily specify and evaluate small variations on collection algorithms. For example, our initial presentation of Baker is a concurrent algorithm while Baker's original algorithm was incremental. We discuss how small changes to our model of Baker can capture the incremental version.
- The proof techniques make it relatively simple to explore and evaluate new collection algorithms before investing many programmer-hours in an implementation. Indeed, our initial motivation in this work was to verify a new collection algorithm we are developing.
- We believe that the modularity of our approach will help achieve code reuse in mechanized proofs, and demonstrate this in our Coq development, in which large pieces of code, such as the semantics and type safety proofs for the languages, are shared between collector definitions.

The idea of using an operational semantics for memory management was introduced by Morrisett et al. [15], and some of our motivations are the same. However, that work was limited to a sequential and purely functional language, and only non-concurrent GC. Also, their semantics was more abstract; its treatment of copying, for example, reuses location names in the from-space and to-space. As such, the semantics cannot be used to distinguish some interesting features and differences among garbage collectors.

In summary, the contributions of this paper are:

- We develop techniques for specifying realistic, concurrent garbage collectors at the language level by providing a precise and succinct operational account of their key ideas.
- We develop proof techniques for establishing the memory safety and the correctness of garbage collectors.
- We demonstrate the effectiveness of the methodology by considering a number of garbage collectors both for sequential and multithreaded programming languages.
- We have mechanized the proposed techniques and the definitions of individual languages and collectors.

2. Overview

We present a high-level overview of the techniques developed in the paper and define terms and notations that will be used throughout. The remainder of the paper and our Coq formalization make the techniques precise and applies them to various garbage collection algorithms.

The structure of the operational semantics. We propose a language-based framework for specifying garbage collectors and establishing their memory safety and correctness. One central facet of our approach is to separately specify the different pieces of a garbage collected programming system, which we divide into three parts: the *mutator* (user program), the *garbage collector* or *memory manager*, and the *top-level*, which facilitates interaction between the two. Figure 1 illustrates the different parts in the semantics, the relevant evaluation relations for each and the theorems proven for each. In this paper, we consider both sequential and multithreaded mutators. For each mutator, we define a small-step evaluation relation, usually written by a single right arrow. This judgment also involves abstract heaps, notated H, whose structure is known to the memory manager. We use the notation e for expressions of the mutator, v for *large values*, those which can be stored in the heap (integers, heap locations, functions, pairs and references) and ℓ for heap locations. At an operational level, the interface between the mutator and the memory manager consists of three operations, defined as separate auxiliary judgments:

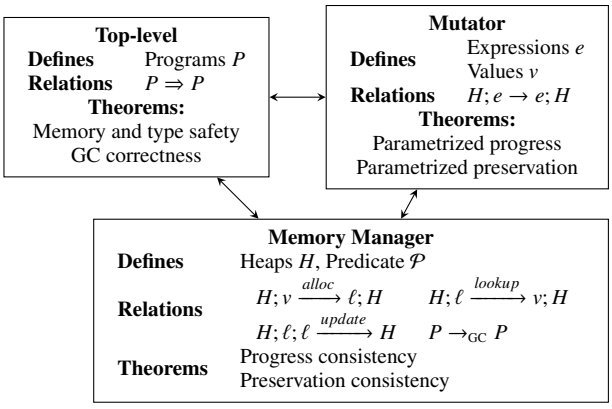

Figure 1. An illustration of the structure of the semantics and the main theorems.

- The allocation judgment $H; v \xrightarrow{alloc} \ell; H'$ indicates that H' is the result of allocating space for value v in heap H, and ℓ is the new location allocated.
- The lookup judgment $H; \ell \xrightarrow{lookup} v; H'$ indicates that location ℓ maps to v in heap H and that H' is the new heap. We return a new heap on a lookup since some collectors need to modify heap metadata after performing a lookup.
- The update judgment $H; \ell; \ell' \xrightarrow{update} H'$ indicates that H' is H updated so that the reference stored at ℓ points to the location ℓ'.

The lookup and update judgments do not fire if ℓ is not in H.

In addition to the structure of the heap and the rules for the judgments above, the memory manager defines an evaluation relation $\rightarrow_{GC}$ for the steps of the GC. The mutator can thus be "linked" with any memory manager that provides the needed definitions. The *top-level* specifies the interaction between the mutator and the memory manager. For example, in a concurrent collector, the top-level may non-deterministically choose whether to step the mutator or the collector. In a stop-the-world collector, the top-level will instead choose to evaluate the collector until a collection is complete before returning to the mutator.

As we illustrate in the rest of this paper, organizing the operational semantics in this separated fashion allows defining garbage collected systems modularly by specifying the desired instance for each of the different pieces. For example, with a sequential mutator, we can use any of many different collection algorithms.

The structure of the correctness proof. We leverage the modularity of the presentation, as described above, to modularize the proofs as well, so that proof effort can be reused and also so that one component (e.g. the mutator) can be modified slightly without

needing to redo all of the proofs for the other components. The high-level idea is to establish the conventional progress and preservation theorems for the mutator by parametrizing over the specific memory manager used. Since both the mutator and the memory manager will have invariants which the other must not violate, to do this in general requires establishing a contract between the mutator and the memory manager to which both adhere. We specify one direction of this contract, the mutator's expectations of the heaps provided by the memory manager, using two *consistency properties*: progress consistency, and preservation consistency. A *progress consistency* property captures the fact that the memory manager's definitions of *alloc*, *lookup* and *update* must be able to make progress (in well-typed states) in order to prove the mutator's parametrized progress theorem. The *preservation consistency* property captures the notion that, for types to be preserved in the mutator, *alloc*, *lookup* and *write* must result in well-typed expressions.

The memory manager's expectations of the mutator are less straightforward. Each memory manager will have different heap invariants which it will require to be preserved by mutator steps. Proving mutator progress and preservation theorems separately for each contract that the memory manager might want would defeat the purpose of separating these components! Instead, we parametrize the progress and preservation theorems over the memory manager's *contract* in addition to its definitions. In our framework, the contract is specified in the form of a predicate over heaps and expressions, which we denote $\mathcal{P}(H, e)$. If the predicate for a given collector definition holds on a heap and a mutator expression, it means that the program state represented by the heap and the expression satisfies the assumptions of the collector. Whenever the state changes (because of a mutator or collector step), we must ensure that the predicate is preserved. We are able to prove parametrized progress and preservation for a very general class of predicates, specifically those which meet the following definition of *compatibility*:

Definition 1 (Compatibility)**.** We say a predicate $\mathcal{P}$ is *compatible* if and only if

- For all e, if e' is a subexpression of e', then

$$\mathcal{P}(H, e) \Rightarrow \mathcal{P}(H, e')$$

- For all e with one or more subexpressions $e_1, \ldots, e_k$,

$$\mathcal{P}(H, e_1) \wedge \cdots \wedge \mathcal{P}(H, e_k) \Rightarrow \mathcal{P}(H, e)$$

For example, if $\mathcal{P}(H, e_1)$ and $\mathcal{P}(H, e_2)$, then $\mathcal{P}(H, e_1\ e_2)$ and vice versa. The compatibility restriction allows preservation proofs to induct on subexpressions (actually on subderivations of the step relation, which step subexpressions). All predicates presented in this paper will be compatible.

Having proved parametrized progress and parametrized preservation for our mutator, we can use the mutator on any heap that meets the consistency properties and any memory manager whose invariants can be expressed by means of a compatible predicate. The overall safety proof for the system consists of establishing type safety for top-level transition judgments, which are an interleaving of those for the mutator and the garbage collector. The type safety (progress and preservation) theorems ensure that well-typed programs will not "get stuck." Since accessing a deallocated memory location is a stuck state, type safety ensures memory safety—that only allocated locations will be accessed, and specifically that garbage collection will never free reachable locations.

At the top-level, the progress theorem consists of establishing that either the mutator can take a step (invoking the mutator-level progress theorem, which applies to the system as long as the heap meets the consistency requirements) or the garbage collector can take a step, or both.

To establish the preservation theorem for the top-level, each transition rule needs to be considered. Much of the proof involves considering mutator dynamic semantics rules, for which we invoke the mutator's (parametrized) preservation theorem. Since the memory manager specifies a small number of additional transition rules for performing garbage collection, we must also show that these steps preserve typing and maintain the predicate. This, together with showing that the predicate is satisfied by a suitable initial state, establishes that the predicate and well-typedness are preserved throughout evaluation.

Proving memory safety by establishing progress and preservation goes a long way but does not guarantee that garbage collection is correct, i.e., it does not preclude errors that do not alter types. Such errors are especially likely in concurrent and semi-space collectors that rely on sophisticated and subtle mechanisms for copying live data even as the mutator continues evaluating. In these cases, we separately prove a garbage collection correctness theorem that establishes that evaluations of the program with and without garbage collection produce the same result.

3. Sequential Computation

The first mutator language we consider is an ML-like sequential (non-parallel) language with mutable references. We first present the syntax, and the static and dynamic semantics of a language for writing mutators that operate on an abstract heap via a set of heap operations. We then define the top-level of *programs* which consist of a mutator expression and a heap, whose definition remains abstract. Leaving the heap abstract makes the semantics modular, allowing us to plug in a variety of memory managers. Next, we show that two concurrent garbage collectors can be specified and proved correct by reusing the infrastructure and theorems established for the mutator and top-level. The specification of a simple, stop-the-world collector for this language can be found in the attached supplementary material.

3.1 The Sequential Mutator Language

We consider an ML-like sequential language. Figure 2 shows the abstract syntax of the language. The types consist of natural numbers, pairs, functions and references. To faithfully model allocation, we consider the only irreducible values to be heap locations, which we notate ℓ. The syntactic class of *large values*, which consist of natural numbers, pairs of locations, functions and references to locations, represents the values which may be stored in the heap. Expressions in the language consist of (free) variables, locations, large values, projection, pair creation (e_1, e_2), application, reference creation $\texttt{ref}(e)$, dereference $!\,e$ and reference update $e_1 := e_2$.

The static semantics of the language, shown in the middle of Figure 2, are standard. The typing judgment $\Gamma \vdash_\Sigma e : \tau$ states that e is a well-formed expression of type τ, relative to two contexts: Γ, which is a finite mapping from variables to types; and Σ, which is a finite mapping from locations to types.

One static judgment, $H : \Sigma$, is omitted. This judgment indicates that the heap H may be typed with the signature Σ, i.e. $\Sigma(\ell) = \tau$ if and only if ℓ is a location in H which stores values of type τ. Since heaps are abstract, this typing judgment will be defined separately for each collector.

The dynamic semantics, defined by the transition judgment $H; e \rightarrow e'; H'$ are as in Figure 2. This judgment indicates that, under heap H, expression e steps to e' and produces new heap H'. The dynamic semantics is largely standard for call-by-value, left-to-right evaluation except for two points. First, the dynamic semantics allocates all large values in the heap (Rule D-Alloc), which realistically models the allocation behavior of the language. Second, the dynamic semantics never manipulates the heap directly,

Syntax:

$$\begin{array}{lrcl} \textit{Types} & \tau & ::= & \texttt{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \tau\ \texttt{ref} \\ \textit{Large Values} & v & ::= & \texttt{n} \mid (\ell, \ell) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \mid \texttt{ref}(\ell) \\ \textit{Expressions} & e & ::= & x \mid \ell \mid v \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid (e, e) \mid \\ & & & e\ e \mid \texttt{ref}(e) \mid\ !\, e \mid e := e \end{array}$$

Statics:

$$\frac{}{\Gamma, x : \tau \vdash_\Sigma x : \tau}\ \textbf{S-Var} \qquad \frac{}{\Gamma \vdash_{\Sigma, \ell:\tau} \ell : \tau}\ \textbf{S-Loc} \qquad \frac{}{\Gamma \vdash_\Sigma \texttt{n} : \texttt{nat}}\ \textbf{S-Nat}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \qquad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma (e_1, e_2) : \tau_1 \times \tau_2}\ \textbf{S-}\times$$

$$\frac{\Gamma \vdash_\Sigma e : \tau_1 \times \tau_2}{\Gamma \vdash_\Sigma \texttt{fst}(e) : \tau_1}\ \textbf{S-Fst} \qquad \frac{\Gamma \vdash_\Sigma e : \tau_1 \times \tau_2}{\Gamma \vdash_\Sigma \texttt{snd}(e) : \tau_2}\ \textbf{S-Snd}$$

$$\frac{\Gamma, x : \tau, f : \tau \to \tau' \vdash_\Sigma e : \tau'}{\Gamma \vdash_\Sigma \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} : \tau \to \tau'}\ \textbf{S-}\to$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau \to \tau' \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma e_1\ e_2 : \tau'}\ \textbf{S-App}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{ref}(e) : \tau\ \texttt{ref}}\ \textbf{S-Ref} \qquad \frac{\Gamma \vdash_\Sigma e : \tau\ \texttt{ref}}{\Gamma \vdash_\Sigma\ !\, e : \tau}\ \textbf{S-Deref}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau\ \texttt{ref} \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma e_1 := e_2 : \tau}\ \textbf{S-Write}$$

Dynamics:

$$\frac{H; v \xrightarrow{alloc} \ell; H'}{H; v \to \ell; H'}\ \textbf{D-Alloc}$$

$$\frac{H; e \to e'; H'}{H; \texttt{fst}(e) \to \texttt{fst}(e'); H'}\ \textbf{D-FstS} \qquad \frac{H; \ell \xrightarrow{lookup} (v_1, v_2); H'}{H; \texttt{fst}(\ell) \to v_1; H'}\ \textbf{D-FstE}$$

$$\frac{H; e \to e'; H'}{H; \texttt{snd}(e) \to \texttt{snd}(e'); H'}\ \textbf{D-SndS} \qquad \frac{H; \ell \xrightarrow{lookup} (v_1, v_2); H'}{H; \texttt{snd}(\ell) \to v_2; H'}\ \textbf{D-SndE}$$

$$\frac{H; e_1 \to e_1'; H'}{H; (e_1, e_2) \to (e_1', e_2); H'}\ \textbf{D-PairS1} \qquad \frac{H; e_2 \to e_2'; H'}{H; (\ell_1, e_2) \to (\ell_1, e_2'); H'}\ \textbf{D-PairS2}$$

$$\frac{H; e_1 \to e_1'; H'}{H; e_1\ e_2 \to e_1'\ e_2; H'}\ \textbf{D-AppS1} \qquad \frac{H; e_2 \to e_2'; H'}{H; \ell_1\ e_2 \to \ell_1\ e_2'; H'}\ \textbf{D-AppS2}$$

$$\frac{H; \ell_1 \xrightarrow{lookup} \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}; H'}{H; \ell_1\ \ell_2 \to [\ell_1, \ell_2/f, x]e; H'}\ \textbf{D-AppE}$$

$$\frac{H; e \to e'; H'}{H; \texttt{ref}(e) \to \texttt{ref}(e'); H'}\ \textbf{D-RefS} \qquad \frac{H; e \to e'; H'}{H;\ !\, e \to\ !\, e'; H'}\ \textbf{D-DerefS}$$

$$\frac{H; \ell \xrightarrow{lookup} \texttt{ref}(\ell'); H'}{H;\ !\, \ell \to \ell'; H'}\ \textbf{D-DerefE} \qquad \frac{H; e_1 \to e_1'; H'}{H; e_1 := e_2 \to e_1' := e_2; H'}\ \textbf{D-WriteS1}$$

$$\frac{H; e_2 \to e_2'; H'}{H; \ell_1 := e_2 \to \ell_1 := e_2'; H'}\ \textbf{D-WriteS2} \qquad \frac{H; \ell; \ell' \xrightarrow{update} H'}{H; \ell := \ell' \to \ell'; H'}\ \textbf{D-WriteE}$$

Figure 2. Sequential Mutator Language: Syntax, Static Semantics, and Dynamic Semantics.

but only via the lookup (e.g., Rule D-DerefE), allocation (Rule D-Alloc), and update (Rule D-WriteE) judgments.

As described in Section 2, we leave the semantics of the mutator parametrized over the definitions contained in the memory manager, and prove a parametrized version of type safety, which assumes only that heaps of the memory manager satisfy progress and preservation consistency, which we will now define.

Definition 2 (Progress consistency for heaps)**.** We say that a heap $H : \Sigma$ of a memory manager is progress consistent with an expression e, if

- for $\ell \in \mathsf{Loc}(e)$, there exist v and H' such that $H; \ell \xrightarrow{lookup} v; H'$.
- for $\ell, \ell' \in \Sigma$, there exists H' such that $H; \ell; \ell' \xrightarrow{update} H'$.
- for all v, there exist ℓ and H' such that $H; v \xrightarrow{alloc} \ell; H'$.

$\mathsf{Loc}(e)$ denotes the set of locations that appear in e.

The preservation consistency properties ensure that heap operations will produce well-typed heaps and expressions.

Definition 3 (Preservation consistency for heaps)**.** Let $\mathcal{P}$ denote the compatible predicate on heaps and expressions. Let H be a heap and suppose $H : \Sigma$. We say that H is preservation consistent under predicate $\mathcal{P}$ if the following conditions are satisfied.

- If $\Sigma(\ell) = \tau$ and $\mathcal{P}(H, \ell)$ and $H; \ell \xrightarrow{lookup} v; H'$, then $\cdot \vdash_\Sigma v : \tau$ and $\mathcal{P}(H', v)$.
- If $\Sigma(\ell) = \tau\ \texttt{ref}$ and $\Sigma(\ell') = \tau$ and $H; \ell; \ell' \xrightarrow{update} H'$ and $\mathcal{P}(H, \ell := \ell')$, then $\mathcal{P}(H', \ell')$.
- If $\cdot \vdash_\Sigma v : \tau$ and $H; v \xrightarrow{alloc} \ell; H'$ and $\mathcal{P}(H, v)$, then $\mathcal{P}(H', \ell)$.

Furthermore, in each case above, we require that $H' : \Sigma'$, where Σ' is an extension of Σ, and that for all e, if $\mathcal{P}(H, e)$ then $\mathcal{P}(H', e)$.

We are now ready to show parametrized progress and parametrized preservation for the mutator. In the below, we assume a memory manager, together with its compatible predicate $\mathcal{P}$, is specified.

Theorem 1 (Parameterized Progress for Sequential Mutator)**.** *Let e be an expression of the mutator and a H be a heap such that $\cdot \vdash_\Sigma e : \tau$ and $H : \Sigma$. If H is progress-consistent with e, then e is a location or there exist e' and H' such that $H; e \to e'; H'$.*

Proof. By induction on the derivation of $\cdot \vdash_\Sigma e : \tau$. □

Theorem 2 (Parameterized Preservation for Sequential Mutator)**.** *Let H be a preservation-consistent heap such that $\mathcal{P}(H, e)$. If $\Gamma \vdash_\Sigma e : \tau$ and $H : \Sigma$ and $H; e \to e'; H'$, then $H' : \Sigma'$ where Σ' is an extension of Σ and $\Gamma \vdash_{\Sigma'} e' : \tau$ and $\mathcal{P}(H', e')$ and for all e_0, if $\mathcal{P}(H, e_0)$ then $\mathcal{P}(H', e_0)$.*

Proof. By induction on the derivation of $H; e \to e'; H'$. □

We end this section by stating the crucial garbage collector property of the expression language: if an expression e is well-typed under a given heap signature Σ, it is well-typed under any subset Σ which contains bindings for all locations that appear in e. This ensures that discarding garbage will preserve the type of the program and will be used in the preservation proof for each collector.

Lemma 1 (Collection Preserves Typing)**.** *If $\Gamma \vdash_\Sigma e : \tau$ and $\Sigma'(\ell) = \Sigma(\ell)$ for all $\ell \in \mathsf{Loc}(e)$, then $\Gamma \vdash_{\Sigma'} e : \tau$.*

Proof. By induction on the derivation of $\Gamma \vdash_\Sigma e : \tau$. In the case of S-Loc, $\Sigma(\ell) = \tau$, so $\Sigma'(\ell) = \tau$. All other cases follow by induction. □

3.2 Top-Level for Sequential Mutators

Figure 3 defines a top-level syntactic class of programs. Top-level programs consist of a sequential mutator expression, defined in the previous subsection, and a heap, whose definition is left abstract. The static semantics of top-level programs, given by the judgment $H \cdot e : \Sigma$, is also defined in terms of those of expressions and heaps: a top-level program $H \cdot e$ is well-typed with respect to a heap signature Σ if the heap is well typed, i.e. $H : \Sigma$ (the rules for this typing

Syntax: $P ::= H \cdot e$

Statics:

$$\frac{H : \Sigma \qquad \cdot \vdash_\Sigma e : \tau}{H \cdot e : \tau}\ \textsc{S-Pgm}$$

Dynamics:

$$\frac{H; e \to e'; H'}{H \cdot e \Rightarrow H' \cdot e'}\ \textsc{D-PStep} \qquad \frac{H \cdot e \to_{\mathrm{GC}} H' \cdot e'}{H \cdot e \Rightarrow H' \cdot e'}\ \textsc{D-GCStep}$$

Figure 3. Sequential Top-Level: Syntax, Static Semantics, and Dynamic Semantics.

judgment are given as part of a memory manager) and the mutator expression is well typed with respect to the signature, i.e. $\cdot \vdash_\Sigma e : \tau$.

We define the dynamic semantics of top-level programs simply as a non-deterministic interleaving of the transitions for the mutator expressions (defined above) and the garbage collector, which remains abstract. This non-determinism models the fact that the garbage collector may be invoked at any time. Top-level programs are thus compatible with stop-the-world garbage collectors (if a single collector transition performs a full collection) and concurrent garbage collectors (if a collector transition performs only a partial collection).

3.3 Concurrent Garbage Collection

We consider two concurrent copying collectors: Baker's (Baker) [2] and Nettles and O'Toole's (Nettles) [16] collectors. These algorithms allow the mutator to run safely while a collection takes place concurrently. The challenge in concurrent collection is to keep the memory in a consistent state so that the mutator can operate as usual while the garbage collector is running and possibly relocating live locations. Both Baker and Nettles are semispace garbage collection algorithms that divide the heap into a from-space and a to-space and collect by copying live objects, relying on periodic synchronization between the mutator and collector (a "read barrier" or "write barrier") to allow the mutator to operate on the heap concurrently. They differ in that, in Baker, the mutator operates on the to-space, requiring a read barrier, whereas in Nettles, the mutator operates on the from-space, requiring a write barrier. We proceed with a few basic definitions shared by both algorithms.

Spaces. We define a space as a mapping from locations to values:

$$\textit{Spaces} \quad s \quad ::= \quad \emptyset \mid s[\ell \mapsto v].$$

A space thus plays the conventional role of a heap, but we use the term *heap* to refer instead to all of the memory allocated to the program, including the from-space, to-space and some metadata. Note that we place no size bound on the domain of a space. This may seem surprising for a model of garbage collection. Indeed, none of our models will *require* a collection to take place. However, all of our models will allow collections to take place at any time, and so our theorems will still demonstrate that collection is safe. Modeling when or how often collection should take place is outside the scope of this paper.

The space typing judgment $s : \Sigma$, defined below, indicates that s is typed under signature Σ. It requires that, for every binding $\ell \mapsto v$ in s, the value v is well-typed (under all of Σ, allowing cycles) and be assigned the correct type in Σ. It also requires that every location in Σ correspond to a binding in s. In particular, weakening does not apply to space signatures: it is *not* the case that if $s : \Sigma$, then $s : \Sigma'$ for some $\Sigma' \supseteq \Sigma$. The definition below relies on an auxiliary judgment $s : \Sigma; \Sigma'$, which is defined inductively on the first signature, but threads through the full signature to use for the purpose of typing expressions.

$$\frac{}{\emptyset : \cdot; \Sigma} \qquad \frac{\cdot \vdash_{\Sigma'} e : \tau \qquad s : \Sigma; \Sigma'}{s[\ell \mapsto e] : \Sigma, \ell : \tau; \Sigma'} \qquad \frac{s : \Sigma; \Sigma}{s : \Sigma}$$

Forwarding maps. Copying collectors typically use forwarding pointers to record the location in the to-space to which from-space objects have been copied. We model such pointers with a *forwarding map* F, which is a finite map from locations to locations. We write $\ell \mapsto \ell'$ to mean that the object at location ℓ in the from-space has been copied to location ℓ' in the to-space.

For mathematical convenience, we implicitly extend the domain of forwarding maps by mapping locations that are not in their domain to themselves:

$$F(\ell) = \begin{cases} F(\ell) & \ell \in \mathrm{dom}(F) \\ \ell & \ell \notin \mathrm{dom}(F). \end{cases}$$

We will also use the notation $F(e)$ to denote an expression in which every location ℓ has been replaced with $F(\ell)$. We extend F to operate on spaces in the following way:

$$F(s) := \{F(\ell) \mapsto F(v) \mid \ell \mapsto v \in s\}$$

and on signatures in the following way:

$$F(\Sigma) := \{F(\ell) \mapsto F(v) \mid \ell \mapsto v \in \Sigma\}$$

A particularly common usage of the forwarding map is the notation $[\ell \mapsto \ell'](e)$ (and similar for spaces and signatures), in which e is forwarded with the singleton forwarding map forwarding ℓ to ℓ'. In addition, we use $F[\ell \mapsto \ell']$ to indicate the extension of the map F with a mapping from ℓ to ℓ'.

Forwarded expressions are well-typed under forwarded contexts.

Lemma 2. *If* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma \vdash_{F(\Sigma)} F(e) : \tau$.

Proof. By induction on the derivation of $\Gamma \vdash_\Sigma e : \tau$. □

Baker's Algorithm

Figure 4 formalizes a memory manager that uses the Baker algorithm for garbage collection. We note that the formalism is fully concurrent, allowing arbitrary interleaving of mutator and collector steps, while Baker's original algorithm [2] was incremental, restricting the interleavings. This is discussed at the end of this section.

Heaps. Heaps and heap operations are given in the upper half of Figure 4. We define heaps as a quadruple consisting of a from-space, a scan set, a to-space, and a forwarding map. The type signature for a heap is the disjoint union of the type signatures for the forwarded from-space and the to-space. Forwarding is necessary since the from-space may refer to locations that have already been copied to the to-space.

In Baker, the mutator operates on the to-space. The specifications of the heap operations follow from this fact. Heap lookup simply finds the binding in the to-space; allocation creates a fresh binding in the to-space; an update of ℓ to ℓ' updates the binding of ℓ in the to-space to contain $\texttt{ref}(\ell')$, i.e. a pointer to ℓ'. The careful reader might notice that the update rule can fire even if the value at ℓ is not already a reference (i.e., not of the form $\texttt{ref}(\ell_0)$). However, this is ruled out by the static semantics. Since Baker is a concurrent algorithm, the mutator may access a memory location before the location is copied to the to-space. In such a case, the *lookup* judgment does not fire; *lookup* will succeed after the location is copied to the to-space. This corresponds to a read barrier in a practical implementation. The read barrier would check on a lookup whether the pointer is an unforwarded pointer in from-space and force a copy if it is.

Heaps: $H ::= \langle s; S; s; F\rangle$

Heap typing:

$$\frac{F(s_f) \uplus s_t : \Sigma}{\langle s_f; S; s_t; F\rangle : \Sigma}\ \textsc{Baker-Heap}$$

Heap operations:

$$\frac{}{\langle s_f; S; s_t[\ell \mapsto v]; F\rangle; \ell \xrightarrow{lookup} v; \langle s_f; S; s_t[\ell \mapsto v]; F\rangle}\ \textsc{Baker-Lookup}$$

$$\frac{\ell \text{ fresh}}{\langle s_f; S; s_t; F\rangle; v \xrightarrow{alloc} \ell; \langle s_f; S; s_t[\ell \mapsto v]; F\rangle}\ \textsc{Baker-Alloc}$$

$$\frac{s_t' = s_t[\ell \mapsto \texttt{ref}(\ell')]}{\langle s_f; S; s_t[\ell \mapsto v]; F\rangle; \ell; \ell' \xrightarrow{update} \langle s_f; S; s_t'; F\rangle}\ \textsc{Baker-Update}$$

Initial heaps:

$$\frac{}{\langle \emptyset; \emptyset; s; \emptyset\rangle \cdot e \text{ initial}}\ \textsc{Baker-Initial}$$

GC transitions:

$$\frac{}{\langle \emptyset; S; s; \emptyset\rangle \cdot e \to_{\text{GC}} \langle s; \mathsf{Loc}(e); \emptyset; \emptyset\rangle \cdot e}\ \textsc{Baker-StartGC}$$

$$\frac{\begin{array}{c}\ell' \text{ fresh} \qquad e' = [\ell \mapsto \ell'](e) \\ F' = F[\ell \mapsto \ell'] \qquad s_t' = ([\ell \mapsto \ell'](s_t))[\ell' \mapsto F'(v)] \\ S' = S \cup (\mathsf{Loc}(F'(v)) \setminus (\mathrm{dom}(s_t) \uplus \{\ell'\}))\end{array}}{\langle s_f[\ell \mapsto v]; S \uplus \{\ell\}; s_t; F\rangle \cdot e \to_{\text{GC}} \langle s_f; S'; s_t'; F'\rangle \cdot e'}\ \textsc{Baker-Copy}$$

$$\frac{\mathrm{dom}(s_f) \cap S = \emptyset}{\langle s_f; S; s_t; F\rangle \cdot e \to_{\text{GC}} \langle \emptyset; S; s_t; \emptyset\rangle \cdot e}\ \textsc{Baker-EndGC}$$

Figure 4. Baker**:** Heap Specification and GC Dynamic Semantics.

GC Dynamic Semantics. The GC dynamic semantics for Baker are shown at the bottom of Figure 4. Rule Baker-StartGC starts garbage collection by flipping the to-space to the from-space position, setting the scan set to the locations of the expression, and resetting the to-space to empty. Rule Baker-Copy copies an uncopied location ℓ contained in the scan set. It first allocates a fresh location ℓ' and extends the forwarding function with the mapping from ℓ to ℓ', updating the mutator expression by forwarding all occurrences of ℓ to ℓ'. It then extends the to-space to map ℓ' to $F'(v)$, the value of ℓ after forwarding, and updates the scan set to contain all the uncopied locations in $F'(v)$. A garbage collection ends (Rule Baker-EndGC) when all the reachable locations in the from-space are copied; that is, the intersection of the scan set and the from-space is empty. When garbage collection ends, the remains of the from-space is thrown out.

Type and memory safety. We establish the progress and preservation theorems for Baker and separately prove its correctness. To this end, we consider the sequential top-level programs in the context of the memory manager based on Baker (Figure 4) and establish type safety and correctness. The proof follows the general outline described in Section 2.

We define a compatible predicate $\mathcal{P}$ on heaps and mutator expressions.

$$\begin{array}{lcll}\mathcal{P}(\langle s_f; S; s_t; F\rangle, e) & \Leftrightarrow & & S \supset \mathsf{FL}(s_t \cdot e) \\ & & \wedge & \mathrm{dom}(F) \cap \mathrm{dom}(s_f) = \emptyset \\ & & \wedge & \mathrm{dom}(s_t) \supset \mathrm{cod}(F).\end{array}$$

The notation $\mathsf{FL}(s_t \cdot e)$ refers to the set of locations that appear in e or in any value in the codomain of s_t, but have not yet been copied to the to-space. Formally,

$$\mathsf{FL}(s_t \cdot e) \triangleq \left(\mathsf{Loc}(e) \cup \bigcup_{\ell \in \mathrm{dom}(s_t)} \mathsf{Loc}(s_t(\ell))\right) \setminus \mathrm{dom}(s_t)$$

The predicate asserts a few basic invariants: (1) the scan set contains all the locations in e and the to-space that are not yet copied to the to-space, (2) that the forwarding map contains no locations from from-space in its domain, and (3) that the codomain of the forwarding-map is within the to-space.

In proving the progress theorem, we consider two cases: if the lookup judgment succeeds on all locations in e, then the mutator-level parametrized progress theorem applies and the mutator takes a step. Otherwise, garbage collection is active and there is an uncopied location that can be copied, i.e., garbage collection takes a step.

Theorem 3 (Progress for Baker)**.** *If $H \cdot e : \tau$, then e is a location or there exist H' and e' such that $H \cdot e \Rightarrow H' \cdot e'$.*

Proof. Suppose $H = \langle s_f; S; s_t; F\rangle$. It is easy to verify that H and e satisfy the progress consistency properties for *update* and *alloc*, since these judgments are always defined. We now consider two cases. If *lookup* in H is defined for all $\ell \in \mathsf{Loc}(e)$, then consistency for *lookup* is satisfied and Theorem 1 applies (so e is a location or the program can step with D-PStep). Otherwise, there is some $\ell \in \mathsf{Loc}(e)$ such that $\ell \notin \mathrm{dom}(s_t)$ (and therefore $\ell \in \mathsf{FL}(s_t \cdot e) \subset S$). By typing inversion, we must have $\ell \in \mathrm{dom}(\Sigma)$, so $\ell \in F(s_f)$. Since $\ell \notin \mathrm{dom}(F)$, this gives $\ell \in s_f \cap S$, so Baker-Copy applies. □

Lemma 3 shows the preservation consistency property for *lookup*. Those for *alloc* and *update* are easily verified.

Lemma 3. *If $H : \Sigma, \ell : \tau$ and $H; \ell \xrightarrow{lookup} v; H'$ and $\mathcal{P}(H, \ell)$, then $\cdot \vdash_\Sigma v : \tau$ and $\mathcal{P}(H', v)$.*

Proof. We have $F(s_f) \uplus s_t : \Sigma$, so since $s_t(\ell) = v$, the definition of space typing gives the typing result. Note that $H = H'$, so we just need to show $\mathcal{P}(H, v)$. For all $\ell' \in \mathsf{FL}(s_t \cdot v)$, since v is in the range of s_t, we have $\ell' \in \mathsf{FL}(s_t \cdot \ell)$, which is a subset of S since $\mathcal{P}(H, \ell)$. □

We then prove the top-level preservation theorem.

Theorem 4 (Preservation for Baker)**.** *If $H \cdot e : \tau$ and $\mathcal{P}(H, e)$ and $H \cdot e \Rightarrow H' \cdot e'$, then $H' \cdot e' : \tau$ and $\mathcal{P}(H', e')$.*

Proof. By cases on the transition rule applied. We show a representative set of the non-trivial cases:

- Baker-Copy. By inversion, $\mathsf{FL}(s_t \cdot e) \subset S \cup \{\ell\}$. We have $F'(s_f) \uplus ([\ell \mapsto \ell'](s_t))[\ell' \mapsto F'(v)] = F'(s_f[\ell \mapsto v]) \uplus [\ell \mapsto \ell'](s_t) = [\ell \mapsto \ell']((F(s_f[\ell \mapsto v]) \uplus s_t))$, so $F'(s_f) \uplus ([\ell \mapsto \ell'](s_t))[\ell' \mapsto F'(v)] : [\ell \mapsto \ell'](\Sigma)$ and Lemma 2 gives $[\ell \mapsto \ell'](\Sigma) \vdash [\ell \mapsto \ell'](e) : \tau$. It is straightforward to show $S' \supset \mathsf{FL}(([\ell \mapsto \ell'](s_t))[\ell' \mapsto F'(v)] \cdot [\ell \mapsto \ell'](e))$ and the other conditions of $\mathcal{P}$.
- Baker-EndGC. By typing inversion, $F(s_f) \uplus s_t : \Sigma$ and $\cdot \vdash_\Sigma e : \tau$. It can be shown in a straightforward way that s_t is well-typed, so suppose $s_t : \Sigma_t$. By the definition of $\mathcal{P}$, we have $S \supset \mathsf{FL}(s_t \cdot e)$, so $\mathsf{FL}(s_t \cdot e) \cap \mathrm{dom}(s_f) = \emptyset$. If $\ell \in \mathsf{Loc}(e)$, then $\ell \in \mathrm{dom}(s_t)$ or $\ell \in \mathrm{dom}(F(s_f)) = \mathrm{dom}(s_f)$. If $\ell \in \mathrm{dom}(s_f)$, then $\ell \in \mathsf{FL}(s_t \cdot e)$, a contradiction, so $\Sigma_t(\ell) = \Sigma(\ell)$. By Lemma 1, $\cdot \vdash_{\Sigma_t} e : \tau_1$. Apply S-Pgm. □

Progress and preservation, as established, come very close to type safety, except for the condition on the predicate. It is, however, straightforward to prove that the predicate is true in an initial state of evaluation (defined in Figure 4 using the judgment initial) because initially $\mathsf{Loc}(s_t)e$ is empty (since all locations would be in s_t), and the

forwarding map would also be empty. Since preservation maintains the validity of the predicate, a standard type safety result (a well-typed expression will proceed to evaluate until it reaches a value) holds true if we begin in an initial state, without any assumptions on the predicate.

We have now shown that sequential top-level programs run with Baker are type and memory safe: they will never become stuck. However, because of the combination of mutation and interleaved copying, it is no longer clear that updates to the heap will always be preserved by copying. We thus prove an additional theorem. Theorem 5 compares two runs of a mutator expression e starting with an initial heap. We run the program with the normal program transition rules, including GC, to get the program $H' \cdot e'$. We also run it without GC using just the expression-level transition rules to get the program $\hat{H}' \cdot \hat{e}'$. Intuitively, we wish to show that these two programs correspond up to the names of locations, which may have been forwarded during one or more rounds of GC. This is stated formally by positing the existence of two forwarding maps, F and F_0. The map F_0 represents the cumulative forwarding effects of all rounds of GC before the current one (if a GC is active), and F includes the current round. If no GC is active, $F = F_0$. We require that e' and $\hat{e}'$ be related up to mappings in F, that bindings in the to-space also be related up to mappings in F, and that (for a strengthened induction hypothesis) any uncopied bindings in the from-space be related up to mappings in F_0.

Below, we use the notations $H.s_f$, $H.S$, $H.s_t$ and $H.F$ to refer to the components of a heap H and $F_1 \circ F_2$ to represent composition: $(F_1 \circ F_2)(\ell) = F_1(F_2(\ell))$.

Theorem 5 (Correctness for Baker)**.** *If $H_0 \cdot e$ initial and $H_0 \cdot e : \tau$ and $H_0 \cdot e \Rightarrow^* H' \cdot e'$, then there exist F and F_0 such that $(H'.F) \circ F_0 = F$ and $H_0; e \rightarrow^* \hat{H}'; \hat{e}'$ and $e' = F(\hat{e}')$ and for all ℓ,*

1. *If $H'; F(\ell) \xrightarrow{lookup} v; H'$ then $\hat{H}'; \ell \xrightarrow{lookup} v'; \hat{H}'$ and $v = F(v')$.*
2. *If $(H'.s_f)(F(\ell)) = v$, then $\hat{H}'; \ell \xrightarrow{lookup} v'; \hat{H}'$ and $v = F_0(v')$.*

Proof. By induction on the derivation of $H_0 \cdot e \Rightarrow^* H' \cdot e'$. The base case is trivial. Suppose that $H_0 \cdot e \Rightarrow^* H'' \cdot e''$ and $H'' \cdot e'' \Rightarrow H' \cdot e'$. By induction, $H_0; e \rightarrow^* \hat{H}''; \hat{e}''$ and $e'' = F(\hat{e''})$ and conditions 1 and 2 hold on the heaps.

Continue by cases on the rule invoked by $H'' \cdot e'' \Rightarrow H' \cdot e'$. We show a representative set of the non-trivial cases. Both deal with GC, so the "non-GC" side of the simulation will not step and we have $\hat{H}'' = \hat{H}'$ and $\hat{e}'' = \hat{e}'$.

- Baker-StartGC. Then $e'' = e'$. Condition 1 on the heap is vacuously satisfied, since lookup will never fire on H'. If $(H'.s_f)(F(\ell)) = v$, then $H''; F(\ell) \xrightarrow{lookup} v; H''$ and so $\hat{H}'; \ell \xrightarrow{lookup} v'; \hat{H}'$ since condition 1 was satisfied before the step.
- Baker-Copy. Then $e' = [\ell \mapsto \ell'](e'')$, so $e' = [\ell \mapsto \ell'](F(\hat{e''}))$. Suppose $H'; F'(\ell_0) \xrightarrow{lookup} v; H'$. If $\ell_0 = \ell$, then $(H''.s_f)(\ell) = v'$, where $v = (H'.F)(v')$ and so, by condition 2, $\hat{H}''; \ell \xrightarrow{lookup} v''; \hat{H}''$ and $v' = F_0(v'')$, so $v = F'(v'')$ and the condition holds. Both conditions continue to hold on other locations.

□

Nettles and O'Toole's Algorithm

Next, we consider the Nettles-O'Toole algorithm (Nettles) [16]. As with the Baker algorithm, Nettles is a semispace collector that proceeds by copying objects from the from-space to the to-space. In Nettles, however, the mutator operates on the from-space. This subtle but important change leads to properties that differ from

Heaps: $H ::= \langle s; S; s; F \rangle$

Heap typing:

$$\frac{s_f : \Sigma \qquad F(s_f \setminus \mathrm{dom}(F)) \uplus s_t : F(\Sigma)}{\langle s_f; S; s_t; F \rangle : \Sigma} \textsc{ Nettles-Heap}$$

Heap operations:

$$\frac{H = \langle s_f; S; s_t; F \rangle \qquad H' = \langle s_f; S \cup \mathsf{Loc}(s_f(\ell)); s_t; F \rangle}{H; \ell \xrightarrow{lookup} s_f(\ell); H'} \textsc{ Nettles-Lookup}$$

$$\frac{\ell \text{ fresh}}{\langle s_f; S; s_t; F \rangle; v \xrightarrow{alloc} \ell; \langle s_f[\ell \mapsto v]; S \cup \{\ell\}; s_t; F \rangle} \textsc{ Nettles-Alloc}$$

$$\frac{s'_f = s_f[\ell \mapsto \mathtt{ref}(\ell')]}{\langle s_f[\ell \mapsto v]; S; s_t; F \rangle; \ell; \ell' \xrightarrow{update} \langle s'_f; S \cup \{\ell\}; s_t; F \rangle} \textsc{ Nettles-Update}$$

Initial heaps:

$$\frac{}{\langle s; \mathsf{Loc}(e); \emptyset; \emptyset \rangle \cdot e \text{ initial}} \textsc{ Nettles-Initial}$$

GC transitions:

$$\frac{\begin{array}{c} s_f = s'_f[\ell \mapsto v] \qquad \ell' \text{ fresh} \\ F' = F[\ell \mapsto \ell'] \qquad s'_t = ([\ell \mapsto \ell'](s_t))[\ell' \mapsto F'(v)] \\ S' = S \cup (\mathsf{Loc}(F'(v)) \setminus (\mathrm{dom}(s_t) \uplus \{\ell'\})) \end{array}}{\langle s_f; S \uplus \{\ell\}; s_t; F \rangle \cdot e \rightarrow_{\mathrm{GC}} \langle s_f; S'; s'_t; F' \rangle \cdot e} \textsc{ Nettles-Copy}$$

$$\frac{\begin{array}{c} s_f = s'_f[\ell \mapsto v] \\ F(\ell) = \ell' \qquad s'_t = s_t[\ell' \mapsto F(v)] \\ S' = S \cup (\mathsf{Loc}(F(v)) \setminus (\mathrm{dom}(s_t) \uplus \{\ell'\})) \end{array}}{\langle s_f; S \uplus \{\ell\}; s_t; F \rangle \cdot e \rightarrow_{\mathrm{GC}} \langle s_f; S'; s'_t; F \rangle \cdot e} \textsc{ Nettles-Recopy}$$

$$\frac{\mathrm{dom}(s_f) \cap S = \emptyset}{\langle s_f; S; s_t; F \rangle \cdot e \rightarrow_{\mathrm{GC}} \langle s_t; \mathsf{Loc}(F(e)); \emptyset; \emptyset \rangle \cdot F(e)} \textsc{ Nettles-Flip}$$

Figure 5. Nettles**:** Heap Specification.

Baker in crucial details. Figure 5 shows our formalization of a memory manager based on the Nettles algorithm.

Heaps. As shown in Figure 5, we define a heap as a quadruple consisting of a from-space, a scan set, a to-space, and a forwarding map; this definition is identical to that of Baker. Since the mutator works on the from-space, the garbage collector does not eliminate copied locations from the from-space, leading to overlap between the from- and to-space. The typing judgment for heaps reflects this fact: s_f always maintains a complete copy of the heap and thus fits the signature Σ, and the disjoint union of the uncopied locations in s_f with s_t are well-typed under the signature $F(\Sigma)$.

The *lookup* operation accesses the contents of the location in the from-space. When returning the heap, it adds to the scan set all locations of the returned value, since these locations are now roots of the expression. The other two operations are straightforward from the fact that the mutator operates on the from-space. The *alloc* operation allocates a fresh location ℓ and extends the from-space with it, extending the scan set S to ensure that this location will be copied. The *update* operation updates the desired location ℓ in the from-space and extends the scan set to include it.

GC dynamic semantics. The GC dynamic semantic rules for Nettles are shown at the bottom of Figure 5. Rule Nettles-Copy copies the location ℓ in the scan set to the to-space by allocating a fresh location ℓ' in the to-space and copying the forwarded contents of ℓ into it. The rule then updates the forwarding map, and extends

the scan set to include the unforwarded locations of the copied value. Rule Nettles-Copy does not remove the copied object from the from-space, nor does it forward any locations in e, since e must still be able to use the from-space. Since a copied location is left for use in the from-space, it may be updated even after it is copied. A second copying transition, Nettles-Recopy, therefore recopies a mutated location which has already been copied (forwarded). In this rule, $s_t[\ell' \mapsto F(v)]$ indicates that the binding of ℓ', which already exists in s_t, should be overwritten with $F(v)$. Since such updated locations are tracked by the scan set, they can be found by inspecting the scan set. Collection terminates when all locations in the scan set are copied to the to-space; in this case, Rule Nettles-Flip makes the to-space the new from-space and atomically forwards all locations in e.

Unlike Baker, this algorithm does not stipulate any conditions on the pointer for lookup, and hence an implementation does not require a read barrier to check which heap the pointer is to. However, it requires a write barrier, which is indicated in rule Nettles-Update by the need to add ℓ_1 to the scan set.

Type and memory safety. We establish the progress and preservation theorems for Nettles and separately prove its correctness. We start by defining the predicate.

$$\begin{array}{rcll} \mathcal{P}(\langle s_f; S; s_t; F\rangle, e) & \Leftrightarrow & & S \supset \mathsf{FL}(s_t \cdot F(e)) \\ & & \wedge & \mathrm{cod}(F) \cap \mathrm{dom}(s_f) = \emptyset \\ & & \wedge & S \supset \mathit{diff}(\langle s_f; S; s_t; F\rangle) \\ & & \wedge & \mathrm{dom}(s_t) \supset \mathrm{cod}(F). \end{array}$$

Numbering the conjuncts of the predicate in order, two of the requirements (1 and 4) are the same as for Baker but the others differ: (2) the codomain of the forwarding map and the domain of the from-space do not overlap, (3) the scan set contains all the mutated locations. The latter condition is formalized using the set $\mathit{diff}(H)$, which collects the set of locations whose values in the from-space and to-space differ by more than a renaming of locations.

$$\begin{array}{l} \mathit{diff}(\langle s_f; S; s_t; F\rangle) = \{\ell \mid \exists v, H'.\langle s_f; S; s_t; F\rangle; \ell \xrightarrow{\mathit{lookup}} v; H' \\ \qquad \wedge F(\ell) \in \mathrm{dom}(s_t) \wedge s_t(F(\ell)) \neq F(v)\} \end{array}$$

We next state and prove a lemma that shows the consistency property for *lookup*, which is always true, unlike in Baker.

Lemma 4. *If $H : \Sigma, \ell : \tau$ and $\mathcal{P}(H, \ell)$, then $H; \ell \xrightarrow{\mathit{lookup}} v; H'$ and $\cdot \vdash_{\Sigma,\ell:\tau} v : \tau$ and $\mathcal{P}(H', v)$.*

Proof. We have $s_f(\ell) = v$ and $s_f : \Sigma$. The typing result follows from the definition of space typing. For all $\ell' \in \mathsf{FL}(s_t \cdot v)$, either $\ell' \in \mathsf{FL}(s_t \cdot \ell)$ (i.e. it was contained in a binding of s_t), or $\ell' \in \mathsf{Loc}(v)$. In the first case, $\ell' \in S$ since $\mathcal{P}(H, \ell)$. In the second, it is clear that $\ell' \in S \cup \mathsf{Loc}(s_f(\ell)) = S \cup \mathsf{Loc}(v)$. □

For Nettles, the progress theorem is relatively easy to establish.

Theorem 6 (Progress for Nettles)**.** *If $H \cdot e : \tau$ and $\mathcal{P}(H, e)$, then e is a location or there exist H' and e' such that $H \cdot e \Rightarrow H' \cdot e'$.*

Proof. With Nettles, a heap is progress consistent with any expression, because *lookup* (Lemma 4), *update*, and *alloc* can always fire. Thus Theorem 1 applies and $H \cdot e$ is a location or can step with D-PStep. □

We can now establish preservation. As in Baker, a significant portion of the proof can be off-loaded to Theorem 2, which established parametrized preservation for the sequential mutator, by proving relevant heap properties as needed.

Theorem 7 (Preservation for Nettles)**.** *If $H \cdot e : \tau$ and $\mathcal{P}(H, e)$ and $H \cdot e \Rightarrow H' \cdot e'$, then $H' \cdot e' : \tau$ and $\mathcal{P}(H', e')$.*

Proof. By induction on the derivation of the program step rules. The proof is similar to the one for Baker. □

Having established preservation in the context of the predicate, we now prove that the predicate always holds by proving that the predicate is true for an initial program (defined in Figure 5). To this end, we set the scan set of the initial heap $S = \mathsf{FL}(s_t \cdot e) = \mathsf{Loc}(e)$; in other words, the scan set contains all the roots of the expression. The other three parts of the predicate follow immediately by the fact that, in the initial heap, the to-space and forwarding map are empty.

In addition to memory safety, we also establish that an evaluation that performs garbage collection using Nettles is equivalent to one that does not perform garbage collection.

Theorem 8 (Correctness for Nettles)**.** *If $H_0 \cdot e$ initial and $H_0 \cdot e \Rightarrow^* H' \cdot e'$, then there exist F and F_0 such that $H_0; e \rightarrow^* \hat{H}'; \hat{e}'$ and $e' = F_0(\hat{e}')$ and for all ℓ and ℓ', if $\ell = F_0(\ell')$ and $H'; \ell \xrightarrow{\mathit{lookup}} v; H'_S$, then there exist v', $\hat{H}'_S$ such that $\hat{H}'; \ell' \xrightarrow{\mathit{lookup}} v'; \hat{H}'_S$ and $v = F_0(v')$.*

Proof. By induction on the derivation of $H_0 \cdot e \Rightarrow^* H' \cdot e'$ in much the same way as the proof of Theorem 5. □

Incremental Garbage Collection

Incremental garbage collectors, such as Baker's original algorithm [2], are a class of concurrent garbage collectors that, instead of allowing arbitrary interleavings of the mutator and collector steps, only make garbage collection steps during specific mutator steps—typically during the memory management steps, lookup, alloc and/or update. Since they limit the possible interleavings between the mutator and the garbage collector, incremental garbage collection algorithms can be viewed as a subset of concurrent algorithms. Their memory safety and correctness therefore follow from those for the corresponding concurrent algorithm. Limiting collector steps to certain points can be specified by changing the semantics of top-level programs in Figure 3.

For example, Baker's original algorithm [2] operates only when the mutator performs an allocation or a lookup—each allocation invokes a constant number of collector steps, and a lookup on a location that hasn't yet been copied takes a step to copy this location. The top-level program could also interleave the steps in a more refined way, such as enforcing two collection steps for every allocation, although this would require adding a counter to the heap.

4. Multithreaded Computation

4.1 The Multithreaded Mutator Language

To investigate collectors for multithreaded languages, we consider the ML-like sequential mutator language described before, extended with UNIX-style fork and join primitives. The syntax and selected semantics for the language are shown in Figure 6. For completeness, the full set of rules is available in the attached supplementary material. Most of the syntax is the same as for the sequential language. The additional constructs are underlined in Figure 6: the `tid` type for thread IDs (TIDs), the `t` TID large value, and the `fork`(e) and `join`(e) expressions. Locations store natural numbers, functions, pairs, and references as before, along with TIDs, which arise from the `fork`(e) expression.

The typing judgment for the language is of the form $\Gamma \vdash_{\Sigma;\Omega} e : \tau$. This statement declares e to be a well-formed expression of type τ under the variable context Γ, the location context Σ, and the TID set Ω. The variable context Γ is a finite mapping of variables to their types. The location context Σ is a finite mapping of locations to the types of values stored at the locations. The TID set Ω is a set of TIDs that have arisen from previous evaluations of `fork`(e). Most of the

Syntax:

Types	τ	::=	$\texttt{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \tau\ \texttt{ref} \mid \underline{\texttt{tid}}$
Large Values	v	::=	$\texttt{n} \mid (\ell, \ell) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \mid \texttt{ref}(\ell) \mid \underline{\texttt{t}}$
Expressions	e	::=	$x \mid \ell \mid v \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid (e, e) \mid e\ e \mid$ $\texttt{ref}(e) \mid\ !e \mid e := e \mid \underline{\texttt{fork}(e)} \mid \underline{\texttt{join}(e)}$

Statics:

$$\frac{}{\Gamma \vdash_{\Sigma;\Omega,\texttt{t}} \texttt{t} : \texttt{tid}}\ \textbf{S-Tid}$$

$$\frac{\Gamma \vdash_{\Sigma;\Omega} e : \tau}{\Gamma \vdash_{\Sigma;\Omega} \texttt{fork}(e) : \texttt{tid}}\ \textbf{S-Fork} \qquad \frac{\Gamma \vdash_{\Sigma;\Omega} e : \texttt{tid}}{\Gamma \vdash_{\Sigma;\Omega} \texttt{join}(e) : \texttt{tid}}\ \textbf{S-Join}$$

Dynamics:

$$\frac{t \text{ fresh}}{H; \mu; \texttt{fork}(e) \to t; \mu[t \mapsto e]; H}\ \textbf{D-ForkE}$$

$$\frac{H; \mu; e \to e'; \mu'; H'}{H; \mu; \texttt{join}(e) \to \texttt{join}(e'); \mu'; H'}\ \textbf{D-JoinS}$$

$$\frac{\mu(t) \neq \ell}{H; \mu; \texttt{join}(t) \to \texttt{join}(t); \mu; H}\ \textbf{D-JoinBlock}$$

$$\frac{\mu(t) = \ell}{H; \mu; \texttt{join}(t) \to t; \mu; H}\ \textbf{D-JoinE}$$

Figure 6. Multithreaded Mutator Language: Syntax, Static Semantics, and Dynamic Semantics.

static semantics are the same as in Figure 2 extended with the TID set Ω. New typing rules are defined for `t`, `fork`(e), and `join`(e).

The transition judgment is of the form $H; \mu; e \to e'; \mu'; H'$. This judgment states that, under heap H and thread map μ, the expression e steps to e' and produces the new thread map μ' and the new heap H'. The thread map μ is a finite mapping of TIDs to expressions. Most of the dynamic semantics are the same as for the sequential mutator language extended with the thread map μ. New transition rules are defined for `fork`(e) and `join`(e). Note that the thread map is directly manipulated in the dynamic semantics. Rule D-ForkE introduces a new thread into the thread map. The two substantive join rules look up the target thread in μ; if it has completed execution, Rule D-JoinE completes the join. If it has not, Rule D-JoinBlock steps to the same join expression, to model blocking until the target thread completes.

To state the parametrized progress and parametrized preservation theorems, we must also be able to type both the heap and the thread map. The heap typing judgment, $H : \Sigma; \Omega$ is defined by the collector. The thread map typing judgment, $\mu : \Omega$ is defined for all collectors:

$$\frac{\text{dom}(\mu) = \Omega}{\mu : \Omega}$$

As with the sequential language, we define progress and preservation consistency.

Definition 4 (Progress Consistency for Heaps)**.** A heap $H : \Sigma; \Omega$ of a memory manager is progress-consistent with an expression e if

- For all $\ell \in \mathsf{Loc}(e)$, there exists v such that $H; \ell \xrightarrow{lookup} v; H'$.
- For all $\ell, \ell' \in \Sigma$, there exists H' such that $H; \ell; \ell' \xrightarrow{update} H'$.
- For all v there exist ℓ and H' such that $H; v \xrightarrow{alloc} \ell; H'$.

Definition 5 (Preservation Consistency for Heaps)**.** Let $\mathcal{P}$ be a compatible predicate on heaps. Let H be a heap and suppose that $H : \Sigma; \Omega$. Then, H is preservation-consistent under predicate $\mathcal{P}$ if the following conditions are satisfied:

- If $\Sigma(\ell) = \tau$ and $\mathcal{P}(H, \ell)$ and $H; \ell \xrightarrow{lookup} v; H'$, then $\cdot \vdash_{\Sigma;\Omega} v : \tau$ and $\mathcal{P}(H', v)$.
- If $\Sigma(\ell) = \tau\ \texttt{ref}$ and $\Sigma(\ell') = \tau$ and $\mathcal{P}(H, \ell := \ell')$ and $\mathcal{P}(H, v)$ and $H; \ell; \ell' \xrightarrow{update} H'$, then $\mathcal{P}(H', \ell')$.
- If $\cdot \vdash_{\Sigma;\Omega} v : \tau$ and $\mathcal{P}(H, v)$ and $H; v \xrightarrow{alloc} \ell; H'$, then $\mathcal{P}(H', \ell)$.
- If $\Gamma \vdash_{\Sigma;\Omega,t} t : \texttt{tid}$, then $\mathcal{P}(H, t)$.

In addition, in the first three cases, we require that $H' : \Sigma'; \Omega$, where Σ' is an extension of Σ and that for all e, if $\mathcal{P}(H, e)$ then $\mathcal{P}(H', e)$.

We are now ready to show parametrized progress and parametrized preservation for the multithreaded mutator.

Theorem 9 (Parameterized Progress for Multithreaded Mutator)**.** *Suppose that H is a heap of a memory manager such that $H : \Sigma; \Omega$ and $\cdot \vdash_{\Sigma;\Omega} e : \tau$. If H is progress-consistent with e, then e is a value or there exist e', H' and μ' such that $H; \mu; e \to e'; \mu'; H'$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\Sigma;\Omega} e : \tau$. □

Theorem 10 states the parametrized preservation theorem. Due to the addition of the thread map and TIDs to the language, an additional conclusion ensuring that the thread map is well-formed after a step is required in order for the theorem to hold.

Theorem 10 (Parameterized Preservation for Multithreaded Mutator)**.** *Let $\mathcal{P}$ be a compatible predicate and H be a preservation-consistent heap such that $\mathcal{P}(H, e)$. If $H : \Sigma; \Omega$ and $\cdot \vdash_{\Sigma;\Omega} e : \tau$ and $\mu : \Omega$ and $H; \mu; e \to e'; \mu'; H'$, then there exist Σ' and Ω', extensions of Σ and Ω respectively, such that*

1. *$H' : \Sigma'; \Omega'$ and $\Gamma \vdash_{\Sigma';\Omega'} e' : \tau$ and $\mathcal{P}(H', e')$ and*
2. *$\mu' : \Omega'$ and either $\mu' = \mu$ or $\mu' = \mu[t \mapsto e_t]$ for some t and e_t, and*
3. *for all e_0, if $\mathcal{P}(H, e_0)$ then $\mathcal{P}(H', e_0)$.*

Proof. By induction on the derivation of $H; \mu; e \to e'; \mu'; H'$. □

In addition to mutator-level parametrized progress and parametrized preservation theorems, we must prove that discarding unreachable locations from the heap will preserve typing.

Lemma 5 (Collection Preserves Typing)**.** *If $\Gamma \vdash_{\Sigma;\Omega} e : \tau$ and $\forall \ell \in \mathsf{Loc}(e). \Sigma'(\ell) = \Sigma(\ell)$, then $\Gamma \vdash_{\Sigma';\Omega} e : \tau$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\Sigma;\Omega} e : \tau$. □

4.2 Top-Level for Multithreaded Mutators

Popular collectors for multithreaded programs use two-level collectors with a shared global heap and private thread-local heaps. For collectors like this, the syntax, static semantics, and dynamic semantics for the top-level syntactic class of programs is shown in Figure 7.

A top-level program consists of the global heap (G), a thread-local heap map (Δ), and the thread map (μ). A top-level program is considered well-typed if for each thread t, the thread heap $(G, \Delta(t))$ is well-typed with respect to a heap signature Σ and TID set Ω and the mutator expression $\mu(t)$ is well-typed with respect to the heap signature and TID set.

The dynamic semantics of top-level programs is defined with three rules. Rule D-PStepNormal and rule D-PStepFork start off by non-deterministically choosing a TID t. Both rules step the thread's expression using a pair of the global heap and its thread-local heap. Because the mutator dynamics treat heaps as abstract, the representation of this thread heap requires no changes to the mutator. Rule D-PStepNormal is invoked when the mutator takes a step that does not invoke rule D-ForkE. In this case, μ is not extended and

Programs: $P ::= G; \Delta; \mu$

Statics:

$$\frac{P = G; \Delta; \mu \qquad \mu : \Omega \qquad \forall t \in \mathrm{dom}(\mu).\, (G, \Delta(t)) : \Sigma; \Omega \wedge \cdot \vdash_{\Sigma;\Omega} \mu(t) : \tau}{P\ \mathsf{ok}}\ \textbf{S-PGM}$$

Dynamics:

$$\frac{\mu(t) = e \qquad \Delta(t) = L \qquad (G, L); \mu; e \to e'; \mu; (G', L')}{G; \Delta; \mu \Rightarrow G'; \Delta[t \mapsto L']; \mu[t \mapsto e']}\ \textbf{D-PStepNormal}$$

$$\frac{\begin{array}{c}\mu(t) = e \qquad \Delta(t) = L \\ (G, L); \mu; e \to e'; \mu[t' \mapsto e_{t'}]; (G', L') \\ (G', L'); e_{t'}; \emptyset \xrightarrow{promote} M; e'_{t'}; (G'', L') \\ \Delta' = \Delta[t \mapsto L'][t' \mapsto \emptyset] \qquad \mu'' = \mu[t \mapsto e'][t' \mapsto e'_{t'}]\end{array}}{G; \Delta; \mu \Rightarrow G''; \Delta'; \mu''}\ \textbf{D-PStepFork}$$

$$\frac{G; \Delta; \mu \Rightarrow_{\mathrm{GC}} G'; \Delta'; \mu'}{G; \Delta; \mu \Rightarrow G'; \Delta'; \mu'}\ \textbf{D-PStepGC}$$

Figure 7. Multithreaded Top-Level: Syntax, Static Semantics, and Dynamic Semantics.

the top-level program step simply updates the local heap and local thread expression for t according to the step.

Rule D-PStepFork is invoked when the mutator invokes the rule D-ForkE, creating a new thread under the TID t'. Before completing the transition, the top-level program first creates the promoted mutator expression $e'_{t'}$ which does not contain any references to locations in L. This is to maintain the invariant that no expression holds references to locations in other local heaps. The *promote* operation is defined by the collector to take as arguments the heap, an expression, and a promotion map, which keeps track of already-promoted locations. Formally, the promotion map is a finite mapping of locations to their promoted locations. The operation recursively promotes the expression and returns the updated promotion map, a new expression using the promoted locations, and the modified heap. Rule D-PStepFork completes by updating the local heap and thread map for the TID t as in rule D-PStepNormal. For the new TID t', a new empty local heap is added to Δ and the promoted expression $e'_{t'}$ replaces $e_{t'}$ in μ. Rule D-PStepGC allows the memory manager to perform one step of collection.

4.3 Doligez-Leroy-Gonthier Algorithm

The Doligez-Leroy-Gonthier, or DLG, collector [5, 6] is a fundamental two-level collector for multithreaded programs. Indeed, many modern collectors for multithreaded programming languages, such as the Manticore [1] and Haskell [12] collectors, trace their roots to the DLG collector. Figure 8 shows a formalism for a memory manager that uses the DLG algorithm for garbage collection.

Heaps. The global heap is defined as a quadruple consisting of a space, a scan set, another space which will serve as the set of marked locations during a global mark-sweep collection, and a predicate indicating whether or not garbage collection is in progress. The local heap is defined as a single space. Heap typing makes use of space typing, but unlike in the collectors for the sequential language, space typing for multithreaded mutators only requires that reachable locations be well-typed. As such, the space typing judgment, defined below, is stated as $s_r \uplus s : \Sigma; \Omega$, declaring that the all locations in the reachable space s_r are well-typed with respect to the heap signature Σ and TID set Ω. The judgment also requires that every location in Σ correspond to a binding in s_r.

$$\frac{}{\emptyset \uplus s : \cdot; \Sigma; \Omega} \qquad \frac{\cdot \vdash_{\Sigma';\Omega} e : \tau \qquad s_r \uplus s : \Sigma; \Sigma'; \Omega}{s_r[\ell \mapsto e] \uplus s : \Sigma, \ell : \tau; \Sigma'; \Omega} \qquad \frac{s_r \uplus s : \Sigma; \Sigma; \Omega}{s_r \uplus s : \Sigma; \Omega}$$

Global Heap: $G ::= \langle s; S; s; p \rangle$

Local Heap: $L ::= \langle s \rangle$

Thread Heap: $H ::= (G, L)$

Heap Typing:

$$\frac{G = \langle s_G; S_G; s_{Gm}; p \rangle \qquad L = \langle s_L \rangle \qquad s_G \uplus s_L : \Sigma; \Omega}{(G, L) : \Sigma; \Omega}\ \textbf{S-Heap}$$

Initial State:

$$\frac{}{\langle \emptyset; \emptyset; \emptyset; \mathtt{f} \rangle; [t_0 \mapsto \langle \emptyset \rangle]; [t_0 \mapsto e]\ \mathsf{initial}}\ \textbf{DLG-Initial}$$

Heap Operations:

$$\frac{G = \langle s_G; S_G; s_{Gm}; p \rangle \qquad \ell \notin \mathrm{dom}(s_G)}{(G, \langle s_L[\ell \mapsto v] \rangle); \ell \xrightarrow{lookup} v; (G, \langle s_L[\ell \mapsto v] \rangle)}\ \textbf{DLG-LookupL}$$

$$\frac{G = \langle s_G[\ell \mapsto v]; S_G; s_{Gm}; p \rangle \qquad \ell \notin \mathrm{dom}(s_L)}{(G, \langle s_L \rangle); \ell \xrightarrow{lookup} v; (G, \langle s_L \rangle)}\ \textbf{DLG-LookupG}$$

$$\frac{v \neq \mathtt{ref}(\ell') \qquad \ell\ \mathsf{fresh}}{(G, \langle s_L \rangle); v \xrightarrow{alloc} \ell; (G, \langle s_L[\ell \mapsto v] \rangle)}\ \textbf{DLG-AllocL}$$

$$\frac{\begin{array}{c}G = \langle s_G; S_G; s_{Gm}; p \rangle \\ (G, L); \ell'; \emptyset \xrightarrow{promote} M; \ell''; \big(\langle s'_G; S'_G; s_{Gm}; p \rangle, L\big) \\ \ell\ \mathsf{fresh} \qquad G'' = \langle s'_G[\ell \mapsto \mathtt{ref}(\ell'')]; S'_G \cup \{\ell\}; s_{Gm}; p \rangle\end{array}}{(G, L); \mathtt{ref}(\ell') \xrightarrow{alloc} \ell; (G'', L)}\ \textbf{DLG-AllocG}$$

$$\frac{\begin{array}{c}G = \langle s_G[\ell_1 \mapsto \mathtt{ref}(\ell)]; S_G; s_{Gm}; p \rangle \\ (G, L); \ell_2; \emptyset \xrightarrow{promote} M; \ell'_2; \big(\langle s'_G; S'_G; s_{Gm}; p \rangle, L\big) \\ G'' = \langle s'_G[\ell_1 \mapsto \mathtt{ref}(\ell'_2)]; S'_G \cup \{\ell_1, \ell\}; s_{Gm}; p \rangle\end{array}}{(G, L); \ell_1; \ell_2 \xrightarrow{update} (G'', L)}\ \textbf{DLG-Update}$$

Figure 8. DLG**:** Heap Specification.

$$\frac{\begin{array}{c}\mu(t) = e \\ \langle s_L; \mathsf{Loc}(e) \cap \mathrm{dom}(s_L); \emptyset \rangle \to^*_{CGC} \langle s''_L; S_L; s'_L \rangle \\ \mathrm{dom}(s''_L) \cap S_L = \emptyset\end{array}}{G; \Delta[t \mapsto \langle s_L \rangle]; \mu \Rightarrow_{\mathrm{GC}} G; \Delta[t \mapsto \langle s'_L \rangle]; \mu}\ \textbf{D-LocalGC}$$

$$\frac{\begin{array}{c}G = \langle s_G; S_G; s_{Gm}; \mathtt{f} \rangle \\ I = \cup_{t \in \mathrm{dom}(\mu)} \mathsf{Loc}(\mu(t)) \cap \mathrm{dom}(s_G)\end{array}}{G; \Delta; \mu \Rightarrow_{\mathrm{GC}} \langle s_G; I; \emptyset; \mathtt{t} \rangle; \Delta; \mu}\ \textbf{D-StartGlobalGC}$$

$$\frac{\begin{array}{c}G = \langle s_G; S_G \uplus \{\ell\}; s_{Gm}; \mathtt{t} \rangle \qquad s'_{Gm} = s_{Gm}[\ell \mapsto s_G(\ell)] \\ S'_G = S_G \uplus (\mathsf{Loc}(s_G(\ell)) \setminus \mathrm{dom}(s'_{Gm}))\end{array}}{G; \Delta; \mu \Rightarrow_{\mathrm{GC}} \langle s_G; S'_G; s'_{Gm}; \mathtt{t} \rangle; \Delta; \mu}\ \textbf{D-GlobalGC}$$

$$\frac{}{\langle s_G; \emptyset; s_{Gm}; \mathtt{t} \rangle; \Delta; \mu \Rightarrow_{\mathrm{GC}} \langle s_{Gm}; \emptyset; \emptyset; \mathtt{f} \rangle; \Delta; \mu}\ \textbf{D-EndGlobalGC}$$

$$\frac{S' = S \cup (\mathsf{Loc}(v) \setminus (\mathrm{dom}(s_t) \uplus \{\ell\}))}{\langle s_f[\ell \mapsto v]; S \uplus \{\ell\}; s_t \rangle \to_{CGC} \langle s_f; S'; s_t[\ell \mapsto v] \rangle}\ \textbf{D-CopyOne}$$

Figure 9. DLG**:** GC Dynamic Semantics.

$$\frac{(G,L)\,;e;M \xrightarrow{promote} M';e';(G',L)}{(G,L)\,;\texttt{fork}(e);M \xrightarrow{promote} M';\texttt{fork}(e');(G',L)}$$

$$\frac{(G,L)\,;e;M \xrightarrow{promote} M';e';(G',L)}{(G,L)\,;\texttt{join}(e);M \xrightarrow{promote} M';\texttt{join}(e');(G',L)}$$

$$\frac{\ell \in \mathrm{dom}(M)}{(\langle s_G;S_G;s_{Gm};p\rangle,L)\,;\ell;M \xrightarrow{promote} M;\ell;(\langle s_G;S_G;s_{Gm};p\rangle,L)}$$

$$\frac{\ell \notin \mathrm{dom}(M) \qquad \ell \in \mathrm{dom}(s_{Gf})}{(\langle s_G;S_G;s_{Gm};p\rangle,L)\,;\ell;M \xrightarrow{promote} M;\ell;(\langle s_G;S_G;s_{Gm};p\rangle,L)}$$

$$\frac{\begin{array}{c}\ell \notin \mathrm{dom}(M) \qquad \ell \notin \mathrm{dom}(s_{Gf}) \qquad s_L(\ell) = e \\ (\langle s_G;S_G;s_{Gm};p\rangle,\langle s_L\rangle)\,;e;M \xrightarrow{promote} M';e';\left(\langle s'_G;S'_G;s_{Gm};p\rangle,\langle s_L\rangle\right) \\ \ell \in \mathrm{dom}(M')\end{array}}{(\langle s_G;S_G;s_{Gm};p\rangle,\langle s_L\rangle)\,;\ell;M \xrightarrow{promote} M';M'(\ell);\left(\langle s'_G;S'_G;s_{Gm};p\rangle,\langle s_L\rangle\right)}$$

$$\frac{\begin{array}{c}\ell \notin \mathrm{dom}(M) \qquad \ell \notin \mathrm{dom}(s_{Gf}) \qquad s_L(\ell) = e \\ (\langle s_G;S_G;s_{Gm};p\rangle,\langle s_L\rangle)\,;e;M \xrightarrow{promote} M';e';\left(\langle s'_G;S'_G;s_{Gm};p\rangle,\langle s_L\rangle\right) \\ \ell \notin \mathrm{dom}(M') \qquad \ell' \text{ fresh}\end{array}}{\begin{array}{c}(\langle s_G;S_G;s_{Gm};p\rangle,\langle s_L\rangle)\,;\ell;M \xrightarrow{promote} \\ M'[\ell \mapsto \ell'];\ell';\left(\langle s'_G[\ell' \mapsto e'];S'_G \cup \{\ell'\};s_{Gm};p\rangle,\langle s_L\rangle\right)\end{array}}$$

Figure 10. DLG**:** Dynamic Semantics of promotion.

A thread heap is well-typed if the space made up of the disjoint union of the global heap's from-space and local heap's space is well-typed with respect to a heap signature Σ and TID set Ω.

Some heap operations make use of the *promote* operation defined by the DLG collector. A selection of the rules governing the *promote* operation are shown in Figure 10. The full set of rules is available in the attached supplementary material. Promotion terminates since every promotion of a location adds the new location to the promotion map, which can only be as large as the (finite) local heap.

Heap lookup locates the binding in the local or global heap. Allocation allocates values except references in the local heap. Since references are impure and cannot be copied, they are instead allocated in the global heap to allow for sharing with other threads. This operation includes promoting the value stored at the reference in order to ensure that no pointers point from the global heap into any local heap. Update also promotes the new value into the global heap before storing it at the reference.

GC Dynamic Semantics. The GC dynamic semantics are split between local heap collection and global heap collection. Each local heap uses a simple atomic copy collector. Rule D-LocalGC initializes the from-space to be the existing space and the scan set to be the locations (the *roots*) of e, and then copies locations one by one with rule D-CopyOne until the termination condition $\mathrm{dom}(s''_L) \cap S = \emptyset$ is reached, at which point the from-space is discarded and the to-space becomes the new local heap.

The global heap uses the Dijkstra concurrent mark-sweep collection technique. In this algorithm, each location is annotated with one of four colors: blue locations are unallocated, white locations are allocated, gray locations are those that have been marked but not yet traversed, and black locations are those that have been marked and fully traversed. In our specification of the global heap, the locations in s_G are white and a fresh location is blue. During a global GC, S is the set of all gray locations and s_{Gm} contains the tree of black locations and their values. In rule D-StartGlobalGC, all of the locations in the global heap reachable from each mutator expression $(\cup_{t\in\mathrm{dom}(\mu)}\mathsf{Loc}(\mu(t)) \cap \mathrm{dom}(s_G))$ are put into S_G, coloring them gray. Each step of the collector, in rule D-GlobalGC, takes one location from S_G and stores it with its value in s_{Gm}, coloring it black. Note that this step can be interleaved with mutator evaluations, which may reset black locations to gray by placing them back in S_G. In rule D-EndGlobalGC, all once-gray locations are colored black, so collection is complete and the white locations are discarded.

It is worth mentioning that, at this level of abstraction, the mark-sweep collection of DLG looks quite similar to the copying collection of Baker and Nettles, with the exception that mark-sweep doesn't include forwarding. In both models, we trace reachable locations and add them to a second space. In the copying collectors, this modeled actually copying objects to the to-space. In the mark-sweep collector, it models tracking (marking) a subset of the locations.

Type and Memory Safety. The predicate $\mathcal{P}$ defines the three invariants of the heaps for the DLG collector: (1) any locations in e not in the global heap must be found in the local heap, (2) any locations in e not in the local heap must be found in the global heap, and (3) the global heap cannot have any references into any local heap.

$$\begin{array}{ll}\mathcal{P}\left((\langle s_G;S_G;s_{Gm}\rangle,\langle s_L\rangle),e\right) \Leftrightarrow & \\ \quad \forall \ell \in \mathsf{FL}(s_G \cdot e).\ell \in \mathrm{dom}(s_L) & \\ \wedge \quad \forall \ell \in \mathsf{FL}(s_L \cdot e).\ell \in \mathrm{dom}(s_G) & \\ \wedge \quad \forall \ell \in \mathrm{dom}(s_G).\mathsf{Loc}(s_G(\ell)) \subseteq \mathrm{dom}(s_G) &\end{array}$$

Type and memory safety are proven for DLG by proving progress and preservation theorems. The proofs proceed in a similar fashion to those for Baker and Nettles.

Theorem 11 (Progress for DLG)**.** *If $P = G;\Delta;\mu$ and P ok, then for all $t \in dom(\mu)$, $\mu(t)$ is a location or there exists P' such that $P \Rightarrow P'$.*

Proof. The lookup, update, and alloc judgments are always defined, so Theorem 9 applies and therefore P can step with D-PStepNormal or D-PStepFork. □

In order to prove preservation, we first prove a small lemma that shows that doing a global GC preserves heap typing.

Lemma 6. *At* D-EndGlobalGC*, for all $t \in dom(\mu)$, we have $(\langle s_{Gm};\emptyset;\emptyset;p\rangle,\Delta(t)) : \Sigma;\Omega$.*

Proof. Assume that the lemma does not hold. This means that $\exists \ell.s_{Gm}(\ell) = e$ where e is not well-typed under Σ and Ω. This is true if some location reachable from e is not in s_{Gm}. However, by D-GlobalGC, any location inserted in s_{Gm} from S has its locations inserted in S. As such, all locations reachable from e must have been inserted into S t some point and then inserted into s_{Gm}. This is a contradiction. □

Theorem 12 (Preservation for DLG)**.** *If $P = G;\Delta;\mu$ and P ok and for all $t \in dom(\mu)$, we have $\mathcal{P}((G,\Delta(t)),\mu(t))$, and $P \Rightarrow P'$, then $P' = G';\Delta';\mu'$ and P' ok, and for all $t \in dom(\mu')$, we have $\mathcal{P}((G',\Delta'(t)),\mu'(t))$.*

Proof. By induction on the derivation of $P \Rightarrow P'$. □

We have now proven type and memory safety of DLG by proving progress and preservation. It remains to show that collection preserves correctness by comparing the result of evaluating the program without collection with the result of evaluating the program with collection. Unlike in Baker and Nettles, because we do not have forwarding, the equivalence of the expressions holds "on the nose" and not just up to forwarding, and the collected heaps are simply subsets of the uncollected heaps.

$$\frac{\begin{array}{c}\mu(t) = e \\ \langle s_L; \mathsf{Loc}(e) \cap \mathrm{dom}(s_L); \emptyset\rangle \rightarrow^*_{CGC} \langle s''_L; S_L; s'_L\rangle \\ \mathrm{dom}(s''_L) \cap S_L = \emptyset\end{array}}{G; \Delta[t \mapsto \langle s_L\rangle]; \mu \Rightarrow_{\mathrm{GC}} G; \Delta[t \mapsto \langle s'_L\rangle]; \mu}\ \textsc{D-LocalGC1}$$

$$\frac{\begin{array}{c}\mu(t) = e \\ \langle s_L; \mathsf{Loc}(e) \cap \mathrm{dom}(s_L); \emptyset\rangle \rightarrow^*_{CGC} \langle s''_L; S_L; s'_L\rangle \\ \mathrm{dom}(s''_L) \cap S_L = \emptyset \\ (G, \langle s'_L\rangle); e; \emptyset \xrightarrow{promote} M; e'; (G', \langle s'_L\rangle)\end{array}}{G; \Delta[t \mapsto \langle s_L\rangle]; \mu \Rightarrow_{\mathrm{GC}} G; \Delta[t \mapsto \emptyset]; \mu[t \mapsto e']}\ \textsc{D-LocalGC2}$$

$$\frac{\begin{array}{c}I = \cup_{t \in \mathrm{dom}(\mu)} \mathsf{greachable}\,(\mu(t)) \\ \langle s_G; I; \emptyset; \mathtt{t}\rangle; \mu \rightarrow^*_{GCGC} \langle s_G; \emptyset; s_{Gm}; \mathtt{t}\rangle; \mu\end{array}}{\langle s_G; S_G; s_{Gm}; \mathtt{f}\rangle; \Delta; \mu \Rightarrow_{\mathrm{GC}} \langle s_{Gm}; \emptyset; \emptyset; \mathtt{f}\rangle; \Delta; \mu}\ \textsc{D-GlobalGC}$$

$$\frac{\begin{array}{c}t \in \mathrm{dom}(\mu) \qquad s'_{Gm} = s_{Gm}[\ell \mapsto s_G(\ell)] \\ S'_G = S_G \uplus (\mathsf{Loc}(s_G(\ell)) \setminus \mathrm{dom}(s'_{Gm}))\end{array}}{\langle s_G; S_G \uplus \{\ell\}; s_{Gm}; \mathtt{t}\rangle; \mu \rightarrow_{GCGC} \langle s_G; S'_G; s'_{Gm}; \mathtt{t}\rangle; \mu}\ \textsc{D-GlobalCopyOne}$$

$$\frac{S' = S \cup (\mathsf{Loc}(e) \setminus (\mathrm{dom}(s_t) \uplus \{\ell\}))}{\langle s_f \uplus \{\ell \mapsto e\}; S \uplus \{\ell\}; s_t\rangle \rightarrow_{CGC} \langle s_f; S'; s_t \uplus \{\ell \mapsto e\}\rangle}\ \textsc{D-CopyOne}$$

Figure 11. Manticore: GC Dynamic Semantics.

Theorem 13 (Correctness for DLG)**.** *Let* $G_0 = \langle\emptyset;\emptyset;\emptyset;\emptyset\rangle$ *and* $\Delta_0 = [t_0 \mapsto \emptyset]$ *and* $\mu_0 = [t_0 \mapsto e]$ *where* t_0 *is the TID of the initial thread. If* $G_0; \Delta_0; \mu_0$ ok *and* $\mathcal{P}((G_0, \Delta_0(t_0)), \mu_0(t_0))$ *and* $G_0; \Delta_0; \mu_0 \Rightarrow^* G'; \Delta'; \mu'$*, then there exist* $\hat{G}'$ *and* $\hat{\Delta}'$ *such that* $G_0; \Delta_0; \mu_0 \Rightarrow^* \hat{G}'; \hat{\Delta}'; \mu'$ *without invoking rule* D-PStepGC*, and* $G' \subset \hat{G}'$ *and for all* $t \in dom(\mu')$*, we have* $\Delta'(t) \subset \hat{\Delta}'(t)$*.*

Without forwarding, the proof is a straightforward induction.

4.4 Manticore Collector

Manticore is a language that allows implicitly and explicitly parallel programs to be concisely expressed and efficiently run. The Manticore collector (Manticore) [1] uses the core design of DLG with a few significant changes: Manticore (1) relaxes the requirement that all mutable data be stored in the global heap through lazy promotion, object proxies, and other techniques, (2) uses an Appel semi-generational collector (instead of a copying collector) for the local heaps, and promotes old generations to the global heap once they reach a certain size (keeping the local heap size constant), and (3) performs parallel stop-the-world collection on the global heap instead of the sequential concurrent collection that DLG uses through the Dijkstra collector.

Our framework allows us to define and prove correct Manticore with a small set of changes to the original DLG specification. We define Manticore to include the Appel semi-generational local heaps and parallel global heap collection features This allows us to keep the same heap and predicate definitions as in DLG. This formulation of Manticore encompasses the second and third changes to DLG described before that comprise the Manticore collector.

GC Dynamic Semantics. The GC dynamic semantics are defined in Figure 11. There are very few changes needed to express Manticore. The local heaps are modeled by having two nondeterministically-chosen local GC steps. Rule D-LocalGC1 is a typical copy collection. Rule D-LocalGC2 promotes the final thread expression into the global heap, expressing the case where the local heap has exceeded some space threshold and the old generation is promoted to the global heap. The global collection is represented by the single rule D-GlobalGC, which atomically performs copy collection on the global heap. The parallelism of each operation is made clear by "choosing" a thread on each GCGC step.

Type and Memory Safety and Correctness The type and memory safety proofs for Manticore use the same progress and preservation theorems as defined for DLG and the correctness theorem is also unchanged. The new and modified GC dynamic rules pose little issue in the proofs.

5. Related Work

As mentioned, our work is motivated by the work of Morrisett et al. (MFH) [15]. We extend that work in several ways, including modeling the interleaving of GC and mutator steps, modeling the heaps at a level in which different kinds of copying as well as mark-sweep collectors can be distinguished (not the case in MFH), and adding parallelism. As such, we are able to model a much broader set of collectors. Ungureanu and Goldberg [18] developed a formal model for memory management in a distributed system which also follows the style of MFH. The model, however, is tightly tied to the particular language they use, which is based on communicating sequential processes as opposed to parallel threads with shared memory. Also, as with MFH, it is not detailed enough to capture the difference between copying and non-copying collectors.

There has been significant work on verifying or certifying different kinds of garbage collectors using mechanized proofs. Much of this research relies on mathematical abstractions or models of the machine and notably the heap. As such, they are more detailed than what is described here, but only capture a specific collector on a particular platform. The GC algorithms are expressed in some kind of language or pseudo-code and the desired invariants on the mathematical abstractions are proven. Initial papers focus on mark-sweep collectors [7–9, 11, 17], while later papers also considered copying collectors. McCreight et al. [14] consider both mark-sweep and copying collectors written in a RISC-like language. This required thousands of lines of Coq proofs for each collector, even under many restrictions. Hawblitzel and Petrank [10] also verified two algorithms: a Cheney copying collector with bump allocation and a mark-sweep collector. Using the Boogie verification-condition generator [3] and the Z3 theorem prover [4], they were able to automate a significant portion of the proof. Both considered only the simplest non-concurrent, non-parallel collectors. Recently, Gammie et al. [**?**] verified an on-the-fly garbage collector for the x86-TSO memory model. While their proof shares many of the same properties as ours (such as an operational semantic model of a concurrent collector and a modular separation with contracts between collector and mutator), their goal is the verification of a particular collector at a very low level of abstraction. Their work is thus complementary to our work in verifying a variety of collectors at a high level which can be subsequently refined.

6. Conclusion

In this paper, we have described a framework for specifying at a high level the key aspects of garbage collectors and their interaction with the language in which they are embedded. The goal has been to allow one to easily specify garbage collectors and their variants while capturing key aspects of the collectors and proving correctness of those aspects. The framework allows us to model concurrency and parallelism using interleavings of a small-step semantics for the mutator and collector. Within the model, we are able to distinguish important aspects of several collectors. For example, we distinguish the to-space invariant of Baker from the from-space invariant of Nettles. We are also able to describe the DLG collector at a sufficient level of detail to capture the key ideas, while avoiding the significant complications that would be needed to, for example, handle issues of weak memory models or other architectural details. Although we are not capturing all the aspects of a real-world collector, by any means, we believe the approach has an important role in developing and comparing collectors.

References

[1] S. Auhagen, L. Bergstrom, M. Fluet, and J. H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57, 2011.

[2] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. URL http://home.pipeline.com/~hbaker1/RealTimeGC.ps.Z. Also AI Laboratory Working Paper 139, 1977.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, 2006.

[4] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[5] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, Jan. 1994. URL ftp://ftp.inria.fr/INRIA/Projects/para/doligez/DoligezGonthier94.ps.gz.

[6] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, Jan. 1993. URL file://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/publications/concurrent-gc.ps.gz.

[7] H. Goguen, R. Brooksby, and R. M. Burstall. Memory management: An abstract formulation of incremental tracing. In *Types for Proofs and Programs, International Workshop TYPES'99*, pages 148–161. Springer, 2000.

[8] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In R. Alur and T. Henzinger, editors, *Computer Aided Verification CAV'96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[9] K. Havelund and N. Shankar. A mechanized refinement proof for a garbage collector. Technical report, Aalborg University, 1997. URL http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund/Publications/gc-refine-report.ps. Unpublished manuscript.

[10] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. *Logical Methods in Computer Science*, 6(3), 2010.

[11] P. B. Jackson. Verifying a garbage collection algorithm. In *Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, Sept. 1998.

[12] S. Marlow and S. L. P. Jones. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management (ISMM)*, pages 21–32, 2011. URL http://doi.acm.org/10.1145/1993478.1993482.

[13] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[14] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proceedings of SIGPLAN 2007 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices. ACM Press, June 2007.

[15] J. G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Record of the 1995 Conference on Functional Programming and Computer Architecture*, June 1995. URL http://www.cs.cornell.edu/Info/People/jgm/papers/fpca_gc.ps.

[16] S. M. Nettles and J. W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*. ACM Press, June 1993. URL http://www.psrg.lcs.mit.edu/ftpdir/james/papers/pldi93.ps.

[17] D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[18] C. Ungureanu and B. Goldberg. Formal models of distributed memory management. In *Proceedings of Second International Conference on Functional Programming*, pages 280–291. ACM Press, June 1997. URL http://www.acm.org/pubs/articles/proceedings/fp/258948/p280-ungureanu/p280-ungureanu.pdf.