Dynamic Space Limits for Haskell

Edward Z. Yang David Mazières
Stanford University
{ezyang,⊥}@cs.stanford.edu

Abstract

We describe the semantics and implementation of a space limits system for Haskell, which allows programmers to create resource containers that enforce bounded resident memory usage at runtime. Our system is distinguished by a clear allocator-pays semantics drawn from previous experience with profiling in Haskell and an implementation strategy which uses a block-structured heap to organize containers, allowing us to enforce limits with high accuracy. To deal with the problem of deallocating data in a garbage collected heap, we propose a novel taint-based mechanism that unifies the existing practices of revocable pointers and killing threads in order to reclaim memory. Our system is implemented in GHC, a production-strength compiler for Haskell.

Categories and Subject Descriptors D.3.4 [*Run-time environments*]

General Terms Languages, Reliability, Security

Keywords Resource Limits, Profiling, Fault Tolerance, Haskell

1. Introduction

High-level languages encourage programmers to think about problems in abstract mathematical terms, which is often conducive to concise, algorithmically correct solutions. Unfortunately, high-level languages can also obscure the time and space requirements of programs, making seemingly correct code unsuitable for the real world. As an example, in e-commerce, an incorrect answer may be preferable to a slow one [4]. As another example, many protocols, including HTTP, specify no limits on the size of message fields. Yet a web server that rejects RFC2616-compliant HTTP headers larger than some arbitrary limit (e.g., 8 KB) is clearly preferable to one that "correctly" parses multi-gigabyte headers after inducing virtual memory paging. These examples demonstrate the importance of a mechanism for bounding resource consumption.

Operating systems like Linux already provide resource limits at the process-level. However, to take advantage of this support, programs must be divided into separate processes—a heavy architectural penalty when an application does not normally use multiple processes. Furthermore, these processes are far more heavyweight than operating system threads or user-level threads: Google Chrome, which uses separate processes per tab, requires 30 MB per tab, while Mozilla Firefox, which runs in a single process, only requires 2 MB per tab [25].

The alternative is to employ sub-process *resource containers* inside of the programming language. Past work has employed a variety of mechanisms to implement resource containers in this setting, from bytecode rewriting [3], to revocable pointers [10] to completely separate heaps [1], to using the garbage collector to directly trace the retainers of objects [17, 26]. The semantics of these systems vary on a few key design points, including whether or not the allocator or the retainer pays for objects and how one can free a resource container that is no longer needed. One difficulty that arises in garbage collected languages is the fact that a cross-container reference can keep a container alive: the container cannot be freed without violating memory safety.

This paper introduces a new point in the design space. Our system utilizes a single garbage-collected heap and allows cross-container references to be treated as normal references. The key insight is that we do not need to modify the garbage collector to support forcibly reclaiming objects. Instead, we conservatively track which containers a thread may have access to via a mandatory *tainting* mechanism. To reclaim a resource container, simply kill all threads tainted with access to that container. To make our approach practical, our API helps threads avoid excessive taint and takes advantage of Haskell's purity. In particular, a thread may spawn "disposable" subcomputations to compute over data in other resource containers, the results of which can be read back without acquiring taint by copying the object out of the container.

We have implemented our approach in GHC, the most popular Haskell compiler. Haskell is a good example of a powerful, high-level language that complicates reasoning about memory allocation. Our choice was motivated by several factors. First, the language is ideal for formally specifying the semantics of resource containers. To our knowledge, this has not been done before. These semantics come by way of a previous cost semantics formalized for GHC's profiler [19], which tells us how resource containers should work for lazy evaluation and higher-order functions. Second, GHC uses a block-structured heap, which allows for an easy implementation of resource containers, optimizing for the rapid allocation and deallocation of containers. Finally, GHC has hardened Haskell's type system against malicious code with a feature called Safe Haskell [24]. Resource containers provide immediate value to Safe Haskell, which is already in production use to confine potentially malicious third-party code [8] but until now could only recover from heap overflow at the process granularity.

Our contributions are as follows. First, we introduce a new approach to resource containers. We describe an implementation for Haskell and provide a precise semantics that accounts for lazy evaluation, higher-order functions, and exceptions. We provide a library to facilitate resource container revocation, built on mandatory tracking and control of reference propagation. We evaluate the memory utilization and performance of our approach. Finally, we comment on the applicability of our technique to other languages.

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2784-8/14/06. . . $15.00.
http://dx.doi.org/10.1145/2594291.2594341

2. Resource limits

The goal of our resource limits system is to limit the memory usage of untrusted or buggy pieces of code. Thus, a user should be able to enforce resource limits on a piece of code in this manner:

```
rc <- newRC 400 {- 4KB pages of memory -}
withRC rc (... untrusted code ...)
```

Here, the user allocates a fresh resource container (**newRC**) with a fixed limit of 400 pages of memory[1] and runs a fragment of code, attributing its memory usage to this container (**withRC**). If the code exceeds the limit, a heap overflow exception is triggered.

As usual, the devil is in the details. It may be easy to say what **withRC** should do with straight-line code, but what about code that returns a function or a thunk that is evaluated outside the **withRC** block? Furthermore, what does it mean to trigger a heap overflow exception when a limit is exceeded? We start by discussing these two aspects of our system; next, as one of our key contributions is a semantics for resource containers, we describe a formal semantics for our system.

Functions and thunks An allocator-pays model of resource usage usually involves a *current resource container*, to which costs are charged. This current container changes when a user invokes **withRC**. An important design choice is whether or not function calls change the current resource container. In the case of a lazy language, one must also decide if thunk evaluation should change the current resource container.

We think the choice for thunks is clear: a thunk should run in the same container it was originally allocated in. The reason for this is predictability: a thunk may be simultaneously forced by several threads, only one of which actually performs the evaluation. If we did not revert the current container upon evaluation, the cost of evaluating the thunk would be charged to the current container of whichever thread won the race, resulting in a non-deterministic cost attribution.

For functions, there are two choices: either functions change the current resource container (lexical scoping), or they do not (dynamic scoping). In the first case, it is unsafe to pass a function to another container, as the user may repeatedly reinvoke your function (which is allocating on your behalf) and induce a large amount of memory usage. In the second case, it is unsafe to call an untrusted function, as it may blow up and induce a large amount of memory usage. The first choice offers users no recourse: functions simply cannot be shared. However, in the second case, a container can allocate a temporary subcontainer (paid out of its own budget) to run the computation. Thus, our system adopts dynamic scoping.

It is no accident that our choices here closely match the cost semantics utilized by GHC's profiler [19]. Profiling and resource limits both seek to answer the same question: "What is the resource consumption of a cost center/resource container?" By adhering to the existing cost semantics, developers can reason about resource containers the same way they reason about code when they are using a heap profiler.[2]

What happens when a resource limit is hit? Assuming that the limit of a resource container has been reached, what should be done? Experience with operating-system based resource limits suggests that some sort of signal should be raised, which in garbage-collected languages has translated into killing threads. Killing threads in a language like Java, however, is a very dangerous operation that can leave a system in a deadlocked state, as it does not give a thread the opportunity to cleanup after itself. Haskell, on the other hand, specifically has support for asynchronous exceptions [13], which are an excellent mechanism for delivering stack overflow and heap overflow exceptions that respect the exception handling stack. When a thread triggers a heap overflow, its exception handlers are invoked, giving it a chance to cleanup or recover (the handler may even operate in a different resource container, guaranteeing that it will run). This approach easily accommodates multiple threads running under the same resource container. Furthermore, existing support for masking asynchronous exceptions enables a thread to temporarily ignore the fact that a resource limit has been hit. Trusted code may decide to allocate beyond its resource limit to ensure a critical region completes.

$$
\begin{array}{lcll}
e & ::= & & \\
 & | & \mathsf{lit} & \text{Literal} \\
 & | & f\ \overline{a} & \text{Application} \\
 & | & x & \text{Thunk} \\
 & | & K\ \overline{a} & \text{Constructor} \\
 & | & \mathsf{op}\ \overline{a} & \text{Primitive} \\
 & | & \textbf{case}\ e\ \textbf{of}\ \overline{K_j\ \overline{x_j}\ \ \to e_j}^{\,j} & \text{Pattern match} \\
 & | & \textbf{let}\ x = \mathit{rhs}\ \textbf{in}\ e & \text{Let binding}
\end{array}
$$

$$
\begin{array}{lcll}
\mathit{rhs} & ::= & & \\
 & | & \lambda \overline{x}\ .e & \text{Function} \\
 & | & \ulcorner e \urcorner & \text{Thunk} \\
 & | & K\ \overline{a} & \text{Constructor}
\end{array}
$$

Figure 1. Syntax for simplified STG

2.1 Big-step cost semantics

We now give a formal cost semantics for resource containers. Rather than give semantics for Haskell, we give semantics for an intermediate language used by GHC called STG [12], which user code is compiled into after optimization. Figure 1 describes the syntax of STG. STG is a simple *untyped* lambda calculus, containing only function applications, constructors, pattern-matching over constructors, let-bindings and thunks. It also includes domains of literals (lit) and primitive operations (op); **withRC** and **newRC** are considered primitive operations, and resource containers (rc) are considered literals. For simplicity, we omit the limit argument from **newRC**, as our semantics do not directly model resource consumption. Functions, constructors and operators can have $\overline{\text{arbitrary}}$ arity, so we simply notate a vector of arguments using an $\overline{\text{overline}}$. We use the identifiers f, g, h, r, x, y and z to represent variables, with the convention that f, g and h are functions and r evaluates to an rc. K ranges over data constructor names.

STG has a few restrictions that make it amenable for compilation to machine code: lambdas occur only as the *rhs* of let-bindings, constructor and primitive applications must be saturated (fully applied), and function arguments must be either literals or variables (represented using a). We will say an *rhs* is a value v, if it is either a constructor application or a lambda. Previous literature [12] describes how to transform programs to obey these restrictions. Thunks are written inside top corner brackets ($\ulcorner$ and $\urcorner$).

Our big-step cost semantics is stated in Figure 2 and is a modernized version of previous cost semantics by Sansom et al. [19] extended for resource containers. This semantics only models the evolution of the current resource container and the attribution of costs; the small-step semantics in the next section handles excep-

[1] It would be relatively easy to build an abstraction layer which permits relative resource limits (e.g., give one third of my limit to the created containers). However, the raw unit of measure is implementation dependent: the choice of 4KB pages in particular is motivated by our use of a block structured heap, described in Section 4.3.

[2] In fact, our very first implementation of this system directly reused GHC's profiling support.

$$\frac{}{\Gamma : \mathsf{lit} \Downarrow_{\mathsf{rc}} \Gamma : \mathsf{lit}}\ \textsc{Lit} \qquad \frac{x \overset{\mathsf{rc}'}{\mapsto} v \ \mathbf{in}\ \Gamma}{\Gamma : x \Downarrow_{\mathsf{rc}} \Gamma : x}\ \textsc{Whnf}$$

$$\frac{\Gamma : e \Downarrow_{\mathsf{rc}'} \Delta : z}{\Gamma[x \overset{\mathsf{rc}'}{\mapsto} \ulcorner e \urcorner] : x \Downarrow_{\mathsf{rc}} \Delta[x \overset{\mathsf{rc}'}{\mapsto} z] : z}\ \textsc{Thunk} \qquad \frac{\Gamma : e\ \overline{[a_i/x_i]}^{\,i} \Downarrow_{\mathsf{rc}} \Delta : z}{\Gamma[f \overset{\mathsf{rc}'}{\mapsto} \lambda \overline{x_i}^{\,i}.e] : f\ \overline{a_i}^{\,i} \Downarrow_{\mathsf{rc}} \Delta : z}\ \textsc{App}$$

$$\frac{\Gamma : e \Downarrow_{\mathsf{rc}} \Delta[y \overset{\mathsf{rc}'}{\mapsto} K_k\ \overline{a_{k,i}}^{\,i}] : y \qquad \Delta[y \overset{\mathsf{rc}'}{\mapsto} K_k\ \overline{a_{k,i}}^{\,i}] : e'_k\ \overline{[a_{k,i}/x_{k,i}]}^{\,i} \Downarrow_{\mathsf{rc}} \Theta : z}{\Gamma : \mathbf{case}\ e\ \mathbf{of}\ \overline{K_j\ \overline{x_{j,i}}^{\,i} \to e'_j}^{\,j} \Downarrow_{\mathsf{rc}} \Theta : z}\ \textsc{Case}$$

$$\frac{z\ \mathbf{fresh}}{\Gamma : K\ \overline{a_i}^{\,i} \Downarrow_{\mathsf{rc}} \Gamma[z \overset{\mathsf{rc}}{\mapsto} K\ \overline{a_i}^{\,i}] : z}\ \textsc{ConApp} \qquad \frac{y\ \mathbf{fresh} \qquad \Gamma[y \overset{\mathsf{rc}}{\mapsto} rhs] : e\ [x/y] \Downarrow_{\mathsf{rc}} \Delta : z}{\Gamma : \mathbf{let}\ x = rhs\ \mathbf{in}\ e \Downarrow_{\mathsf{rc}} \Delta : z}\ \textsc{Let}$$

$$\frac{\mathsf{rc}'\ \mathbf{fresh}}{\Gamma : \mathbf{newRC} \Downarrow_{\mathsf{rc}} \Gamma : \mathsf{rc}'}\ \textsc{NewRC} \qquad \frac{\Gamma : r \Downarrow_{\mathsf{rc}} \Delta : \mathsf{rc}' \qquad \Delta : f\ a \Downarrow_{\mathsf{rc}'} \Theta : z}{\Gamma : \mathbf{withRC}\ r\ f\ a \Downarrow_{\mathsf{rc}} \Theta : z}\ \textsc{WithRC}$$

Figure 2. Big-step cost semantics

tions and heap overflow. The basic transition is $\Gamma : e \Downarrow_{\mathsf{rc}} \Delta : a$, which states that expression e with heap Γ transitions to a value or literal a with new heap Δ, where the current resource container is rc. Bindings on the heap $x \overset{\mathsf{rc}}{\mapsto} hval$ are associated with a resource container rc, stating that this container is being charged for the binding; alternatively, one can consider the address x to reside in container rc. We conflate heap locations with variables; heap values may be any valid *rhs* or an indirection to another heap value (a nod to how thunk update is actually implemented).

The most important thing these semantics model is how the current resource container changes (otherwise, they are standard); allocating operations (such as a let-binding) simply charge their allocations to the current resource container. As an example, consider the THUNK rule. It states that a variable/heap pointer x which points to a thunk e can be evaluated to a new heap pointer z (and the binding x updated), if e evaluates to z with the current resource container rc' (the container the thunk was charged to on the heap.) In comparison, the APP rule does not change the current resource container.

It is worth making a brief remark about the WITHRC rule, which states **withRC** takes a resource container r along with arguments f and a, rather than a single expression. There is a good reason for this: **withRC** $r\ (f\ x)$ is illegal in STG: and one would need to write **let** $z = \ulcorner f\ x \urcorner$ **in withRC** $r\ z$ with a single-argument **withRC**. However, this would not achieve the intended effect, as $f\ x$ is a thunk associated with the parent resource container, and would revert the resource container immediately on entry.

We can give the following guarantee: modulo **newRC** and **withRC**, code running in one resource container cannot induce unbounded resource usage in another container, *as long as* there are no infinite thunks reachable from other containers. In Section 4.5, we describe a way to deal with infinite *global* thunks.

2.2 Small-step operational semantics

While the big-step semantics provide a clear picture of how the current resource container changes, we also need to reason about how resource containers interact with exceptions. To do this, we first recast the previous cost-semantics as a small-step operational semantics in Figure 3, with an explicit stack.

The form of transitions is $\Gamma, s, C \longrightarrow \Gamma', s', C'$, where s points to the stack in the heap (stacks constitute resource usage!), and C is the program code. A program code is either $\mathtt{return}\ a$, which states that the program is returning the value or literal a to the top continuation on the stack, or $\mathtt{eval}\ e$, which means that the program is evaluating some expression. While we don't model multiple threads explicitly, a thread is merely a stack pointer and a program code. The **allocated** condition indicates that a variable is fresh.

A stack $\mathbb{S}$ is a sequence of stack frames $f_1 \triangleright f_2 \ldots \triangleright f_n$ which grows to the right. Stack growth constitutes allocation. A stack frame can be an update frame **upd** x (given a return value z, update x to point to z, and return z), a continuation **case of** $\overline{alt}$ (given a return value, case-match on it and continue evaluation in the appropriate branch), an application frame **ap** $\overline{a_i}^{\,i}$ (given a return value f, apply it to the arguments $\overline{a_i}^{\,i}$), or a stack link **link** s, which points to a linked stack chunk with which to continue execution when the current chunk underflows. Technically, the arguments of functions should also be stored on the stack, but for simplicity we continue to denote binding using substitution and do not explicitly model the argument stack.

There are two things to note about the new semantics. First, the current resource container is now implicitly represented as the resource container of the *current* stack chunk; thus, **withRC** allocates a new stack chunk in the desired resource container and links it to the previous chunk in the old container (see Section 4.4 for an optimization). When a stack chunk underflows, the current resource container resets to the current container of the previous stack chunk. Second, when a thunk is entered, the thunk is replaced with a placeholder $\bullet$ known as a *black hole* [18]. As there is no rule for evaluating $\bullet$, a black hole blocks any other thread that attempts to force it until the thunk is updated.

In Figure 4, we introduce transition rules for synchronous and asynchronous exceptions by adding a corresponding pair of program codes, $\mathtt{raise}_w\ w_0$, which represents a synchronous exception w being processed up the stack, and $\mathtt{suspend}_w\ e$, which represents an asynchronous exception being processed up the stack. We use the convention that w indicates the heap location of an exception value and w_0 indicates the heap location of an exception-raising thunk. There is also a new stack frame, **catch** h (catch an exception and run handler h). Synchronous exceptions are thrown, while asynchronous ones are non-deterministically induced by external events (which we indicate by placing a w over the transition arrow).

Synchronous and asynchronous exceptions are handled nearly identically, except in how they handle update frames. A normal exception overwrites the thunk with a closure w_0 which always throws an exception: this works because we know that any future attempt to evaluate this thunk will always cause an exception. However, in the case of an asynchronous exception, a second attempt to evaluate the thunk may succeed; thus, we should record any partial work we may have achieved in evaluating the thunk for next time [13].

$$
\begin{array}{rcll}
\Gamma, s, \mathsf{eval}\ x & \longrightarrow & \Gamma, s, \mathsf{return}\ x & (x \overset{\mathsf{rc}'}{\mapsto} v\ \mathbf{in}\ \Gamma) \\
\Gamma, s, \mathsf{eval\ lit} & \longrightarrow & \Gamma, s, \mathsf{return\ lit} & \\
\Gamma[x \overset{\mathsf{rc}'}{\mapsto} \ulcorner e \urcorner], s, \mathsf{eval}\ x & \longrightarrow & \Gamma[x \overset{\mathsf{rc}'}{\mapsto} \bullet,\ s' \overset{\mathsf{rc}'}{\mapsto} \mathbf{link}\ s \triangleright \mathbf{upd}\ x], s', \mathsf{eval}\ e & \\
\Gamma[x \overset{\mathsf{rc}}{\mapsto} \bullet,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{upd}\ x], s, \mathsf{return}\ z & \longrightarrow & \Gamma[x \overset{\mathsf{rc}}{\mapsto} z,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{return}\ z & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ K\ \overline{a_i}^{\,i} & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S},\ z \overset{\mathsf{rc}}{\mapsto} K\ \overline{a_i}^{\,i}], s, \mathsf{return}\ z & (z\ \mathbf{allocated}) \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\, f\ \overline{a_i}^{\,i} & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{ap}\ \overline{a_i}^{\,i}], s, \mathsf{eval}\, f & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{ap}\ \overline{a_i}^{\,i}], s, \mathsf{return}\, f & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ e\ \overline{[a_i/x_i]}^{\,i} & (f \overset{\mathsf{rc}'}{\mapsto} \lambda \overline{x_i}^{\,i}.e\ \mathbf{in}\ \Gamma) \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ \mathbf{let}\ x = rhs\ \mathbf{in}\ e & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S},\ z \overset{\mathsf{rc}}{\mapsto} rhs], s, \mathsf{eval}\ e\ [z/x] & (z\ \mathbf{allocated}) \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ \mathbf{case}\ e\ \mathbf{of}\ \overline{alt_j}^{\,j} & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{case\ of}\ \overline{alt_j}^{\,j}], s, \mathsf{eval}\ e & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{case\ of}\ \overline{K_j\ \overline{x_{j,i}}^{\,i} \ \to e'_j}^{\,j}], s, \mathsf{return}\ z & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ e'_k\ \overline{[a_{k,i}/x_{k,i}]}^{\,i} & (z \overset{\mathsf{rc}'}{\mapsto} K_k\ \overline{a_{k,i}}^{\,i}\ \mathbf{in}\ \Gamma) \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S},\ s' \overset{\mathsf{rc}'}{\mapsto} \mathbf{link}\ s], s', \mathsf{return}\ z & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{return}\ z & \\
\Gamma, s, \mathsf{eval}\ \mathbf{newRC} & \longrightarrow & \Gamma, s, \mathsf{return}\ \mathsf{rc}' & (\mathsf{rc}'\ \mathbf{allocated}) \\
\Gamma, s, \mathsf{eval}\ \mathbf{withRC}\ \ \mathsf{rc}'\, f\ a & \longrightarrow & \Gamma[s' \overset{\mathsf{rc}'}{\mapsto} \mathbf{link}\ s], s', \mathsf{eval}\, f\ a &
\end{array}
$$

Figure 3. Small-step operational semantics

$$
\begin{array}{rcll}
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ \mathbf{catch}\ \ f\ x\ h & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{catch}\ h], s, \mathsf{eval}\, f\ x & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{catch}\ h], s, \mathsf{return}\ z & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{return}\ z & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ \mathbf{throw}\ \ w & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S},\ w_0 \overset{\mathsf{rc}}{\mapsto} \ulcorner \mathbf{throw}\ \ w \urcorner], s, \mathsf{raise}_w\ w_0 & (w_0\ \mathbf{allocated}) \\
\Gamma[x \overset{\mathsf{rc}}{\mapsto} \bullet,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{upd}\ x], s, \mathsf{raise}_w\ w_0 & \longrightarrow & \Gamma[x \overset{\mathsf{rc}}{\mapsto} w_0,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{raise}_w\ w_0 & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{case\ of}\ \overline{alt_j}^{\,j}], s, \mathsf{raise}_w\ w_0 & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{raise}_w\ w_0 & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S},\ s' \overset{\mathsf{rc}'}{\mapsto} \mathbf{link}\ s], s', \mathsf{raise}_w\ w_0 & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{raise}_w\ w_0 & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{catch}\ h], s, \mathsf{raise}_w\ w_0 & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ h\ w & \\
\Gamma, s, \mathsf{eval}\ e & \xrightarrow{w} & \Gamma, s, \mathsf{suspend}_w\ e & \\
\Gamma, s, \mathsf{return}\ z & \xrightarrow{w} & \Gamma, s, \mathsf{suspend}_w\ z & \\
\Gamma[x \overset{\mathsf{rc}}{\mapsto} \bullet,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{upd}\ x], s, \mathsf{suspend}_w\ e & \longrightarrow & \Gamma[x \overset{\mathsf{rc}}{\mapsto} z,\ z \overset{\mathsf{rc}}{\mapsto} \ulcorner e \urcorner,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{suspend}_w\ z & (z\ \mathbf{allocated}) \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{case\ of}\ \overline{alt_j}^{\,j}], s, \mathsf{suspend}_w\ e & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{suspend}_w\ \mathbf{case}\ e\ \mathbf{of}\ \overline{alt_j}^{\,j} & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S},\ s' \overset{\mathsf{rc}'}{\mapsto} \mathbf{link}\ s], s', \mathsf{suspend}_w\ e & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{suspend}_w\ e & \\
\Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{catch}\ h], s, \mathsf{suspend}_w\ e & \longrightarrow & \Gamma[s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{eval}\ h\ w &
\end{array}
$$

$$
\begin{array}{rcll}
\Gamma[x \overset{\mathsf{rc}}{\mapsto} \bullet,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S} \triangleright \mathbf{upd}\ x], s, \mathsf{suspend}_{w'}\ e & \xrightarrow{w} & \Gamma[x \overset{\mathsf{rc}}{\mapsto} w_0,\ s \overset{\mathsf{rc}}{\mapsto} \mathbb{S}], s, \mathsf{raise}_w\ w_0 & \left(\substack{w_0 \mapsto \ulcorner \mathbf{throw}\ \ w \urcorner \\ w \mapsto \mathbf{heapOverflow}}\ \mathbf{in}\ \Gamma\right)
\end{array}
$$

Figure 4. Small-step rules for synchronous and asynchronous exceptions

The rules for asynchronous exceptions automatically give us a rule for heap overflow, which is an asynchronous exception. Furthermore, we can nearly always throw an asynchronous exception rather than take a transition that requires allocation. The only case where this is not true is when processing update frames for asynchronous exceptions, which allocate in order to move the suspended computation from the stack to the heap. Thus, we introduce one more rule, below the dashed line: if we cannot allocate space for this suspended computation, we instead update the thunk to point to a global exception thunk for heap overflows. This behavior can be justified by observing that any other thread which might force the thunk must allocate into the (overflowed) heap to store the result, inducing a heap overflow exception. We can now show, for these semantics:

Theorem 2.1. *For any small-step transition that may induce heap allocation (indicated by **allocated** or by a new stack frame), it is always a valid step to instead induce a **heapOverflow** asynchronous exception.*

Proof. By case-analysis over program codes: eval and return can directly transition to suspend, no transition rules from raise allocate, and suspend is handled by the rule below the dashed line. □

Theorem 2.2. Progress. *When a thread heap overflows, it is guaranteed to progress to the top-most catch frame.*[3]

Proof. Induction on the size of the stack: all transitions after an asynchronous exception decrease their stack. □

These guarantees do not hold in the presence of exception masking. Exception masking can be introduced into these semantics by adding one additional piece of thread state. When exceptions are masked, the *w* rules are not applicable. Thus, masking offers a safety escape from resource limits: when a thread is masked, we always fulfill its allocation requests, even when the current resource container has exceeded its limit.

3. Reclamation

We have described how to bound memory allocation by a container, but we have not said how to reclaim the memory consumed by a container. In a garbage collected language, one cannot simply free a container, since doing so would result in dangling pointers. However, it would be a severe bug if a user could slowly leak memory by retaining pointers to data that should be garbage collected—especially if this is data that other users are paying for!

There are generally two approaches to deallocating objects in a memory-safe, garbage-collected language: one can use special *revocable pointers* that raise an exception on any attempt to access a deallocated object, or one can simply eliminate all reachable pointers to the object, which requires killing threads holding such pointers. Each approach has been individually proposed in the literature and offers different trade-offs.

The first approach requires that all cross-container references be revocable pointers. To avoid leaking plain references, any access to a revocable pointer must be mediated by functions that copy data and/or return more revocable pointers. Revoking pointers is a natural way to think about reclamation from an operating systems perspective, where data is explicitly transferred across protection boundaries. However, this style can be awkward in a programming language. Worse, a functional language such as Haskell encourages programming with immutable data; transforming a previously examined immutable value into an exception would be quite disconcerting for functional programmers.

The second approach requires identifying and killing all threads that might have access to the data one wants to deallocate. Here, it is members of the root set that are explicitly revoked, as opposed to direct pointers to deallocated objects. Thus, no special support is necessary for cross-container references. However, under this scheme, a thread working with cross-container data has to be very careful, as at any moment, it may be killed due to a resource-container deallocation.

Given that these two approaches have different benefits, a good solution should support both! Hence, we conservatively track which resource containers a thread may have access to. However, we also introduce special revocable resource-container references, called `RCRefs`, which mediate inter-thread communication. A thread can dereference an `RCRef` in one of two ways: it may follow the pointer normally, thereby "tainting" itself as a potential retainer of the object's resource containers; alternatively, it may opt to copy the value into its own resource container.

To understand our API, a small amount of background in Haskell is necessary. The next subsection discusses some basic aspects of Haskell's design. We then present the details of our reclamation mechanism. Finally, we discuss how our solution dovetails nicely with information flow control, a technique frequently used to confine untrusted Haskell code.

3.1 Haskell background

Haskell is a pure functional language, meaning variables are immutable and functions are like mathematical functions, deterministic and without side-effects. Haskell does allow IO, but the fact that a computation performs IO must be reflected in its type. For example, a value of type `Char` represents a particular, immutable Unicode code point. By contrast, a value of type `IO Char` represents some computation that, when executed, may produce side effects and will return a `Char`. An example of such a computation in the system library is `getChar :: IO Char` (the keyword `::` specifies the type of a symbol), which reads a character from standard input and returns it. Syntactic sugar facilitates hooking together IO computations, e.g.:

```
copyChar :: IO () -- type () is unit (like void)
copyChar = do c <- getChar
              putChar c
```

An important point is that IO computations can invoke pure functions, while pure functions cannot cause IO to happen; this is enforced statically by the type checker. Haskell does have mutable values, but these can only be manipulated from within IO computations. For example, the system library provides the following three functions to allocate, read, and write mutable values:

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Finally, type `IO` is an instance of a typeclass called `Monad`. The syntactic sugar for hooking together IO computations actually applies to any instance of `Monad`. Hence, it is common for programmers to implement new `IO`-like types and make them monads. Introducing monads is further encouraged by the fact that many library functions are polymorphic over all monads.

A common technique for constructing monads is to define a new type that internally contains `IO` computations, effectively "wrapping" `IO` in another level of type constructor. Through appropriate use of modularity, such wrapped versions of `IO` can expose only a subset of available `IO` operations to code. Such restricted monads are typically how people safely confine untrusted third-party Haskell code [23, 24].

3.2 Revocation API

The minimal set of threads to kill to reclaim a container is precisely the set of threads from which the container is reachable. One might imagine calculating this set during garbage collection; indeed Section 6.2 discusses previous systems which do this. However, this approach can lead to a subtle bug: suppose that a thread temporarily references another container, intending to drop the reference when it is done processing the data. If the thread fails to drop a *single* reference, it will be summarily killed when the data is reclaimed; worse, the program may give the impression of working correctly, if the thread drops all of the references most of the time.

Instead we opt for a conservative scheme which "fails fast". Every thread is associated with a set of containers to which it may have references, called the *current label*. A thread inherits the label of the thread that spawned it. When a thread changes container using **withRC**, the new container is *permanently* added to the thread's label. Similarly, when a thread reads an `RCRef`, the `RCRef`'s container is added to the current label. Critically, the label of an `RCRef` is immutable, ensuring that a developer can always tell what containers its data may depend on. Moreover, using an appropriately defined *container monad*, called `CM`, the label can be computed with no special assistance from the language runtime.

Figure 5 shows a subset of the API available in the `CM` Monad. A computation in the container monad can be run using `startCM`, and

[3] However, the catch handler may fail to terminate: thus, exception handlers which can catch heap overflows should be considered trusted code.

```
startCM  :: CM () -> IO ()
withRC   :: RC -> CM a -> CM a   -- lifted from IO
newRC    :: Int -> CM RC         -- lifted from IO
killRC   :: RC -> CM ()

forkRC        :: RCSet -> CM a -> CM (RCResult a)
readRCResult  :: RCResult a -> CM a
copyRCResult  :: RCResult a -> Transfer a -> CM a

newRCIORef    :: a -> RCSet -> CM (RCIORef a)
readRCIORef   :: RCIORef a -> CM a
copyRCIORef   :: RCIORef a -> Transfer a -> CM a
writeRCIORef  :: RCIORef a -> a -> CM ()
```

Figure 5. Subset of revocation API.

we provide variants of newRC and withRC specialized for the CM monad. In particular, withRC rc adds rc to the current label of a thread. We also provide a new function, killRC, which deallocates a resource container by revoking any RCRefs to it and killing all threads that could retain references to it. Here is a simple example:

```
startCM (do  rc <- newRC 400
             withRC rc (... expr ...)
             killRC rc
             ...)
```

This code seems to do some computation and then free the container rc. Surprisingly, as withRC permanently added rc to the current label of the main thread, killRC will kill itself, and the rest of the code is never run. To avoid this situation, we provide the function forkRC, which spawns a new thread—with its own current label—to run the computation.[4] forkRC takes an argument rcset, which states what containers the result of the thread may have a dependence on and returns an RCResult, which is simply an RCRef that also performs thread synchronization (e.g., an IVar). Now, we could simply just read the result with readRCResult, tainting the main thread with the containers in rcset—however, this would defeat the point of forking in the first place. Alternatively, we could *copy* the result to our original resource container and avoid tainting the container completely. This is achieved using copyRCResult with a *transfer function* which describes how to copy the argument. A trusted library provides combinators for copying primitive data (e.g., trPrim for boxed integers) as well as more complex data (e.g., trList n t for lists, where n is the maximum number of elements to copy and t is the transfer function to apply to items).

Thus, the correct version of our example looks like this:

```
r <- forkRC rcset (withRC rc (... expr ...))
x <- copyRCResult r trPrim
killRC rc
```

Generally, one wants to take cross-container data (possibly very large) and convert it into a small result, which is then copied back to the original thread. As an example, suppose we wanted to record the average size of HTTP requests, where each request lives in its own resource container. The length calculation requires all of the HTTP request, but only a single integer needs to be copied back to the main thread.

RCResult is not the only mechanism by which interthread communication can occur: any existing communication primitive can be augmented with RCRef to be incorporated into the CM monad. As a simple example, we consider IORef, which is replaced with RCIORef. Like forkRC, creating an RCIORef with newRCIORef requires the set of resource containers the RCIORef is allowed to retain. readRCIORef adds this set to the label of any thread reading the reference; copyRCIORef copies the value into the current resource container. However, the most interesting operation on RCIORef is writeRCIORef, which performs an additional check to ensure the current thread's label is a subset of the RCIORef's. Otherwise, the written value might depend on a container the reference is not permitted to retain, and the write must be rejected. In our library, we also provide RCMVar, which is the container-aware version of MVars, a simple abstraction for interthread communication and synchronization.

3.3 Information flow control

We have presented this reclamation API in isolation for the sake of understandability; however, this interface has a close relationship to *information flow control* (IFC), a technique used for managing the propagation of sensitive information. In an IFC system, security policies are captured by *labels*, which dictate how sensitive (secrecy) or trusted (integrity) a piece of data is. To give a concrete example of an integrity policy, suppose we have a database which should only allow writes from Alice or Bob: its label is the set of *principals* {Alice, Bob}. Alice can give a piece of data the {Alice} label by *endorsing* it. We now say that a label l may flow to l' if $l \subseteq l'$: thus data Alice endorses can flow to the database, but not data Carl endorses.[5]

Our resource reclamation scheme is, in fact, an information flow control scheme which enforces an integrity policy: principals correspond to resource containers, labels correspond to sets of reachable containers, and endorsement corresponds to a copy operation. In particular, the CM Monad is merely a type-specialized version of an existing, publicly-available information flow control monad called LIO [23]. Taking advantage of this correspondence, our implementation of the CM monad is closely modeled after LIO, with some simplifications to make it easier to use. An advantage of building on LIO is that it is one of the more widely-used Haskell libraries for confining untrusted code, allowing us to draw on prior experience designing the API and semantics. Additionally, adopting these semantics gives us a proof of *noninterference*, which translates into an assurance that our thread tracking is accurate.

4. Implementation

Our implementation utilizes GHC's block-structured heap, so we first give a brief description of it, and then discuss our implementation in more detail. We also discuss three incidental details related to implementation.

4.1 Block-structured heap

The conventional design for a garbage collector is to allocate a few large, contiguous blocks of memory to serve as the heap. GHC's block-structured heap [5, 14, 21] is an alternative to this scheme, which overcomes some of the inflexibilities of a large chunks of memory. The idea is to divide memory into fixed-size B-byte blocks (in our case, B is 4 KB). These blocks are linked together in order to provide memory for the heap. Since most objects are much smaller than the size of a block, these linked blocks do not have to be contiguous. When a heap runs out of space, more blocks can be easily chained onto it. For example, the nursery, in which new objects are allocated, is simply a chain of blocks.

Blocks are associated with a *block descriptor*, which contains information about the block such as what generation it belongs to, how full it is, etc. Block descriptors are placed in an easy-to-

[4] It's not strictly necessary to create a new thread; however, heap overflow exceptions should not cross over the forkRC boundary.

[5] These labels can be generalized to be arbitrary propositional formulas [22], with our sets forming disjunctions of principals.

calculate location: any pointer into a block can be converted into a pointer for the block descriptor with a few instructions. We can maintain contiguous blocks by collecting block descriptors together in a block descriptor table. Block descriptor tables and blocks are allocated together in a unit called a *megablock* (1 MB in our case); if an object exceeds the size of a megablock, it can spill into the next megablock, although the remaining space is unusable (as the block descriptor table has been overwritten).

GHC currently uses the block-structured heap to good effect for a parallel generational-copying garbage collector [14]. This garbage collector utilizes blocks as the unit of work for garbage collection; as the copying collector operates, it copies objects into “todo blocks”, which may then get passed to other GC threads for further scavenging, to look for more live objects.

4.2 Resource containers are chains of blocks

The flexibility of the block-structured heap allows for a direct implementation resource containers: a block of memory is marked as belonging to a container in its block descriptor.[6] We maintain the invariant that the owner of a nursery is the current resource container. If the current resource container changes, we swap the nursery blocks with a set of nursery blocks stored in the new resource container.[7] There is one extra detail with respect to lazy evaluation: when evaluating a thunk, we need to determine which container owns this thunk. This can be done by calculating the block descriptor of the thunk, which contains a reference to the resource container. This costs only one extra memory dereference, and experimentally (Section 5.2), we’ve found paying this dereference every thunk evaluation to be relatively cheap.

In fact, the primary complication is adjusting the garbage collector to preserve the containers of objects. GHC utilizes a copying generational garbage collector, which operates by repeatedly scavenging the “todo blocks” associated with each generation, looking for more live objects to copy into the to-generation. With containers, every resource container has a “todo block” (multiplied by the number of generations) which live objects are copied into. Partially filled blocks are tracked using a scan stack, [9] and once these blocks fill up, they can be added to the pool of work available for the parallel work-stealing collector.

4.3 Resource limits are enforced during block allocation

Rather than check if a resource limit has been exceeded at every allocation, we instead perform resource limit checks when blocks are allocated. Is this a good metric? It will certainly over-estimate the space used, compared to the actual space occupied by live objects. On the other hand, in a garbage collected language, the live object residency is only known after a garbage collection; in the case of a generational collector, it is only known after a major garbage collection. Using blocks as our metric also has the singular advantage of accounting for space wasted due to heap fragmentation (some relevant measurements can be found in Section 5.) And, of course, interposing at the block allocation layer is a lot simpler than interposing at the general allocation layer.

Running out of blocks for a container is very similar to an out-of-memory event. However, we have considerably more flexibility, as we are not *actually* out-of-memory and can comfortably maneuver ourselves to a desired state. From our experience implementing these checks for GHC, we can classify these heap overflows into a few cases:

- The block was explicitly requested by user code, by way of an allocation of an object. These cases can be handled simply: reject the request and raise a heap-overflow exception.
- The block was requested, but it may not be appropriate to immediately raise an exception. When code is in a critical region, exceptions may be *masked*, deferring any asynchronous exceptions. In this case, we fulfill the request, and raise a heap overflow exception when the mask is lifted.
- The block was requested during the course of garbage collection to serve as the to-space for an object of a container. In this case, there is no thread that was directly responsible for triggering the limit. Instead, we mark the resource container as killed and empty the nursery. The next time any thread in the container allocates, it discovers that there is no memory left in the nursery. Instead of requesting a GC, however, it will simply trigger an asynchronous exception.

4.4 Optimizing stacks

Stack chunks are usually preallocated contiguous blocks of memory which have extra space for new stack frames. In our formal model, we suggested that a new stack chunk be allocated whenever we enter a thunk or invoke **withRC**. This can be quite expensive, since thunk entry is quite common in lazy programs. A simple optimization is to not allocate a new stack chunk when the resource container changes (recording the change separately). When the thread ends up needing to allocate a new stack chunk, the new stack chunk will be properly attributed; similarly, if a stack is reified due to an asynchronous exception, the new thunk will be charged to the appropriate container.

Some costs will get misattributed: the resource container which originally allocated the stack chunk may pay for other container’s stack frames. The inaccuracy is small: usually, the space used by stacks is temporary and quite small (by default, GHC 7.6 enforces a maximum stack size of 8M). Furthermore, an attacker seeking to inflate the memory usage of another container would need to convince the victim to *repeatedly* enter into the adversarial code from their own stack: any single thunk can waste the remainder of the stack chunk, but it cannot cause the caller to allocate more memory.

4.5 Constant-applicative forms

A *constant-applicative form* (CAF) is frequently described as a top-level value defined in a program, which is allocated statically in the program text, rather than at runtime during program execution. For example, the expression `someGlobal = 25` would be considered a CAF. Who is responsible for having allocated a CAF? We place CAFs in a static resource container, separate from the rest of the program. After all, they are only ever evaluated once, and it shouldn’t matter *who* ends up evaluating them.

In some circumstances, however, CAFs can use up quite a lot of resources. A common pattern in Haskell is to use a lazy infinite data-structure to represent data which is conceptually infinite (e.g., a table of prime numbers). Unfortunately, these infinite data structures can induce infinite allocation when they are fully forced. To combat this situation, we developed a tool which looks through all of the CAFs exported by a program and speculatively evaluates them to detect infinite or very large CAFs. When the time it takes to fully evaluate a CAF is longer than some threshold, we replace it with a *non-updatable* thunk (a zero-arity function); untrusted users of the CAF now pay for the execution of that code and no sharing occurs. Another possibility might be to offer more control over

[6] Our system does not track memory that does not live on the heap (e.g., malloced memory from an FFI binding)—functions which provide access to these resources are expected to do manual accounting. This does not mean that all FFI code is untracked: for example, arbitrary precision integers in Haskell are stored on the heap and are accounted properly.

[7] In a multithreaded setting, each container maintains a set of nursery blocks per thread, so independent threads can switch into the same container without synchronizing.

what resource container a CAF is placed in; this works well when code is being dynamically loaded.

4.6 Interaction with the optimizer

One challenge with working with a highly optimizing compiler in a non-strict language is that the optimizer may cause costs to be attribute to containers differently from what you might expect. While attribution is clear in post-optimization STG (the intermediate representation), users of Haskell do not generally write STG directly. To see what may go wrong, consider a simple program:

```
rc <- newRC 100
x <- readInput
withRC rc $ do
    print (x * x)
```

The intent of the program is to attribute the cost of `x * x` to the resource container `rc`. However, the container associated with `x * x` is not actually well defined. An aggressive compiler will notice that `x * x` is a pure computation and lift it as far up lexically as possible (in hopes of exposing some other optimization opportunities), resulting in this code:

```
rc <- newRC 100
x <- readInput
let r = x * x -- ***
withRC rc $ do
    print r
```

When this program is run, `r` will not be charged to `rc`! Indeed, out of the box, `withRC` only guarantees correct attribution with respect to monadic actions, which enforce ordering.

An obvious fix is to convert `withRC` into a special form, so that it is treated specially by the optimizer, e.g., as is employed for profiling with annotations. There were two reasons why we did not use this design. First, this approach gives up composability: it is no longer possible to create larger combinators using `withRC`. While lack of composability is not a big deal for profiling annotations, it is important that users can build their own libraries around our API. Second, GHC implements semantics-preserving optimizations by duplicating these annotations as necessary; it is much harder to safely duplicate proper function calls with arguments, and the most plausible implementation would probably turn off optimizations around `withRC`.

Thus, we instead require programmers to rewrite their code so that the *free variables* of a computation are threaded through `withRC1`, which is a built-in function that is opaque to the optimizer:

```
withRC1 :: RC -> a -> (a -> CM b) -> CM b

rc <- newRC 100
x <- readInput
withRC1 rc x $ \x' -> do
    print (x' * x')
```

The optimizer cannot "see" that `x'` is the same as `x`: all it sees is a function which (lazily) takes a variable as an argument, and produces another another object.

5. Evaluation

5.1 Correctness

In order to show that resource limits were effective at bounding memory usage, we ran a variety of allocating programs with various resource limits and measured the memory usage of the programs. By "memory usage", we mean two things:

1. The self-reported heap *residency* estimate, i.e., the space productively taken up by live objects.[8]
2. The true memory usage, as seen by the *operating system*. This quantity will generally be higher than heap residency, as it accounts for heap fragmentation, temporary blocks of memory allocated to perform garbage collection and other miscellaneous allocation. We collected this information using Valgrind Massif (an external heap profiler) and GHC's internally reported number of allocated blocks (we found these two metrics to be nearly equal in all of our experiments).

In Figure 6, we report how our resource limits system scaled up with successively larger resource limits, on a combination of different programs. opl and tree were programs we found on Stack Overflow, where the authors had needed help debugging a space leak in their programs.[9] suml is a program that sums a list of integers using a linear amount of heap-allocated stack. block and megablock were synthetic programs constructed to repeatedly allocate data greater than the block size (4k) and greater than the megablock size (1M). Finally, ghc tests resource limits on a proper, real-world system. Our program compiled the test case for bug #7428, which induces an exponential space blow-up in the optimizer.[10]

The graphs can be interpreted as follows: the x-axis indicates the resource limit we set, whereas the y-axis indicates the actual memory usage. Each vertical line represents a data point, where the top of the line indicates the true memory usage, and the bottom of the line indicates the self-reported heap residency upper bound. We normalized both these numbers against a baseline process which did not do anything, to discard fixed overhead of the GHC runtime. The graphs include two slope lines: the light gray line plots actual memory usage to the resource limit one-to-one, while the dark line plots them two-to-one. These graphs demonstrate some interesting behavior about our resource limits system:

Garbage collection is privileged As we can see, sometimes the true memory usage exceeds the resource limit, although it never exceeds twice the resource limit. This is because our implementation allows containers to exceed their specified resource limit during garbage collection. As a copying garbage collector may require up to twice as much memory as the size of its heap to do collection, we see some programs (e.g. suml) which can exceed their limit by twice as much. Fortunately, this is only temporary, as after garbage collection the usage returns to previous levels (or better), and use of a different type of garbage collector (for instance, a compacting collector) would eliminate this entirely. If a major garbage collection uses a lot of space, it can result in the quantized behavior you can see in opl, tree and ghc, where programs allocated a lot of memory during a major GC, exceeding the resource limit by a large amount.

Heap fragmentation is properly attributed In megablock, we see the true utilization of the heap is far worse than the resource limit. As we mentioned previously, block structured heaps cope poorly with allocations greater than a megablock, because there is no way to use the extra space at the tail end of a megablock group (fragmentation). However, because we count this wasted space towards the container, a program cannot skirt its resource limit by inducing bad heap fragmentation.

[8] Why an estimate? Residency can only be calculated accurately after a major garbage collection that traverses the entire heap. However, we can provide an upper bound after minor collections by assuming all data that was live in an old generation continues to be live.

[9] `http://stackoverflow.com/questions/3190098/` and `http://stackoverflow.com/questions/5552433/`

[10] `http://ghc.haskell.org/trac/ghc/ticket/7428`

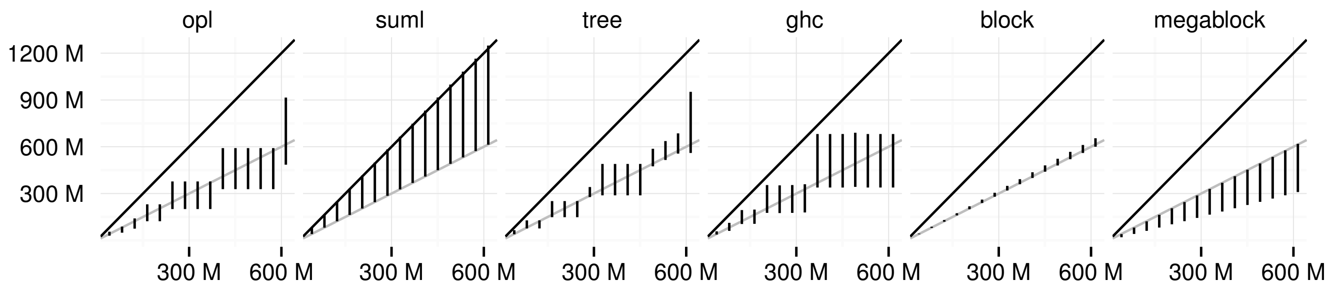

Figure 6. Resource limit accuracy, where x-axis records the limit set and y-axis records the heap residency and the true memory used.

Program	Allocs	Time	Elapsed	TotalMem
circsim	+0.0%	+3.2%	+3.1%	-5.0%
constraints	+0.0%	+2.8%	+2.9%	+0.0%
fibheaps	+0.2%	+2.9%	+2.9%	-0.6%
fulsom	+0.0%	+2.1%	+2.1%	-5.5%
gc_bench	+0.0%	+0.9%	+0.9%	+0.0%
happy	+0.9%	+5.4%	+5.5%	+0.5%
hash	+0.0%	+6.4%	+6.3%	+0.0%
lcss	+11.2%	+5.0%	+4.9%	+1.9%
mutstore1	+0.0%	+1.3%	+1.3%	+3.4%
mutstore2	+0.0%	-0.2%	-0.3%	-0.6%
power	+0.0%	+3.1%	+2.9%	+2.1%
spellcheck	+0.0%	+3.3%	+4.0%	+0.0%

Table 1. Garbage collector overhead by nofib

5.2 Overhead

Enforcing Resource limits requires modifications to code generation as well as the garbage collector, which incurs overhead even when resource limits are not being used. To quantify this overhead, we compared our resource limits to a mostly vanilla[11] checkout of GHC HEAD using the nofib benchmark suite [16]. Specifically, we tested against the garbage collector tests, which are designed to stress-test the storage manager. Our experiments were conducted on a machine with two dual-core Intel Xeon E5620 (2.4Ghz) processors and 48GB of RAM.

Our experiments are shown in Table 1. Binary size change is omitted from the figure, but we consistently pay on the order of a 14–15%; the primary expense here is the extra code we need to check if a resource container has changed upon thunk entry. However, there is only a modest performance impact of 3–5%, as compared to mainline GHC. This is not *quite* good enough to turn on by default, but it is certainly close.

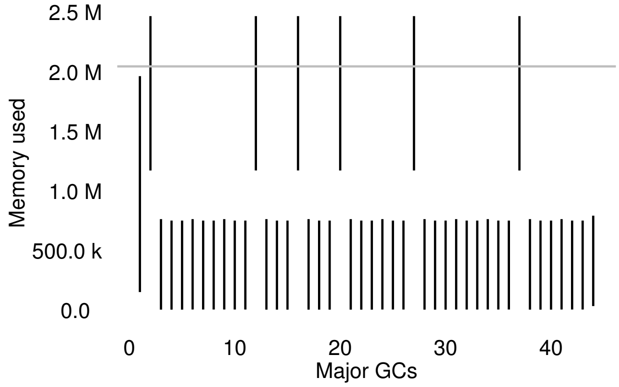

Figure 7. Memory usage in a single run of the prisoner's dilemma server, sampled over major GCs.

Conns	RC	RC disabled	Vanilla
10	2,511.7	2,515.2	2,514.5
50	12,271.3	12,311.2	12,351.3
100	19,891.2	20,756.2	20,885.6
1000	18,484.0	22,434.5	23,104.8

Table 2. Happstack measurements (requests per second)

5.3 Applications

Finally, we ran experiments on a few nontrivial programs. Our first program was a game server for the iterated prisoners dilemma, where agents were implemented as threads limited to 2M of memory which could transmit over a channel a boolean indicating cooperate/defect, and then read out the choice of their opponent. We implemented a few strategies, as well as a buggy strategy that leaked memory. Every major GC, we then sampled the heap residency and the actual memory usage (GHC-reported); a representative run can be seen in Figure 7, using the same conventions as the graphs in Figure 6, except that the x-axis varies over time. There are two things to note about the graph: first, the spikes correspond to the six times the buggy strategy was run (and killed for exceeding its resource limit); second, there was no memory leakage, as we were able to completely reclaim the resources allocated from each run.

Our second program was Happstack 7.3.1, an existing open-source web server for Haskell, which had a bug in its request parser, wherein it would happily accept infinite HTTP headers in its default configuration. Fixing this bug was a simple three-line modification to the source code to apply a resource container per connection, which caused Happstack to correctly terminate the bad connection. We then tested the overhead of resource containers with a simple benchmark and `httperf --rate=1000 --burst-length=10`, varying the number of simultaneous connections we attempted, carried out on the same machine as the overhead experiments. We only used a single core for the webserver. Table 2 shows results for Happstack with resource containers and without resource containers. We also included a baseline "vanilla" measurement taken with stock GHC 7.8RC1. This figure shows that there is some slowdown when there are a large number of resource containers. This is due to an initialization phase at the beginning of every minor garbage collection which takes time linear in the number of live containers. We believe it should be possible to further reduce this cost.

[11] In the process of developing our system, we also implemented a generic optimization for the collector which improved GC performance by about 5%. Since this optimization is not specific to our system, we included it in the baseline comparison.

6. Related work

Resource limits for untrusted code have been well-studied in a variety of settings, although with much less coverage for functional programming languages.

6.1 Operating systems

Starting with mechanisms as simple as the `setrlimit` system call, limits have long been supported by POSIX-style operating systems. While these systems usually operate on a coarse-grained level (managing pages of memory rather than individual heap objects), they still elucidate many of the high level issues that come with enforcing resource limits.

For example, the need for an abstract entity to charge costs to is well recognized; many systems define some equivalent of a cost-center rather than tie resource consumption to processes. Resource containers [2], for example, were a hierarchical mechanism for enforcing limits on resources, especially the CPU.

HiStar [27] organizes space usage into a hierarchy of containers with quotas. Any object not reachable from the root container is garbage collected. Containers are charged for the sum of the quotas of all objects they contain. When multiple containers have hard links to the same object, each is separately charged for the full cost of the object. Any object so linked to multiple containers must have a fixed quota; otherwise, one process can too easily induce arbitrary quota usage in a container belonging to another process to whom it has granted the object.

The above systems are *retainer-* or *consumer-based accounting* systems: they do not care who created the data, just who is holding on to it. In contrast, our system is a *producer-based accounting* system: the individual who produced the data is held accountable for the data. By contrast, EROS [20] checks resource usage at allocation time, when a page is requested from a space bank—since the page will always be attributed to the space bank it was allocated from, this is a producer-based accounting system. A space bank's limit can easily be increased; however, destroying a space bank can cause resources in use by a subsystem to unceremoniously disappear. In our system, use of garbage-collection means such forced reclamation must be handled at the language level.

6.2 Programming Languages

A number of programming languages have support for resource limits. These systems divide into those which *statically* ensure that resource limits are respected, and those that perform these checks *dynamically*.

Static resource limits PLAN [11] is an early example of a programming language with extra restrictions in order to ensure bounded resource usage. PLAN takes the time-honored technique of removing *general recursion* in order to ensure the termination of all programs. Unfortunately, such a restriction would be a bitter pill to swallow for a general purpose programming language like Haskell, and even so PLAN cannot prevent programs from taking large but bounded amounts of resources. Another restriction that can be imposed is eliminating the garbage collector and utilizing some other form of memory management such as monadic regions [6]. Proof-based approaches include work by Gaboardi and Péchoux [7], which develop techniques for proving resource properties on programs which compute over infinite data. These proofs can be combined with code in a proof-carrying code scheme [15]. We think these are all promising lines of work and nicely complement dynamic resource limits.

Dynamic resource limits A number of programming languages have support for dynamic resource limits.

A lot of work has gone towards resource limits for Java, since Java is perhaps the most widely used programming language that also has some ability to run untrusted code. JRes [3] is one of the original systems for Java, and, like us, took the approach of having a single heap. Resource usage was tracked by dynamically rewriting Java bytecode to increment usage counters and track deallocation via weak references. They suggested that resources could be reclaimed by killing Java threads.

Luna [10] observed that killing Java threads was a very unsafe operation. While our system addresses this problem by utilizing Haskell's support for asynchronous exceptions, Luna attempted to build a system which could manage revocation without killing any threads. They achieved this by introducing remote pointers, which look like normal pointers but are revocable. An important restriction demanded by this design is that ordinary pointers cannot be accessed through remote pointers. As our system shows, you can safely support both operations.

KaffeOS [1] provides each resource container its own separate garbage collected heap. While each heap can be garbage-collected separately, KaffeOS must treat inter-container references specially, using a write barrier to detect these references and replace them with entry and exit items. Like our system, KaffeOS has the desirable property that resource limits account for true resource usage as seen by the operating system.

The line of work proposed by Wick et al. [26] and Price et al. [17] takes a different approach than these Java-based systems. These systems cleverly utilize the garbage collector to trace the set of objects retained by a thread in order to determine its resource consumption. This gives them a retainer-based cost model. The authors of these papers argue that consumer-based is more appropriate for a majority of applications. However, we think that retainer-based accounting conflates resource accounting and resource reclamation. It is counter-intuitive to charge a thread for accepting an object before the thread has even had a chance to examine the object, and consumer-based systems must introduce weak/unaccountable references to accommodate this fact. Similarly, it's useful to know the retainers of an object, even when the actual memory usage of a thread is below quota. Furthermore, both of these systems have difficulty dealing with multiple retainers, having to charge the cost of an object to an arbitrary container. Finally, tracing-based accounting charges only for the size of objects, and does not consider other incidental but important factors such as memory fragmentation.

Resource containers have also shown up in the architecture of Mozilla Firefox [25], under the name of *compartments*. While they do not address the issue of resource limits directly, their architecture closely mirrors ours: compartments are composed of arenas (blocks which only hold a single type of object). However, to support per-compartment GC, cross-compartment references must be mediated by wrappers, which serve as a remembered set and enforce cross-compartment security policies. We think it would be very easy to apply the ideas in our system to Firefox, by generalizing compartments beyond their close association to a single website.

7. Conclusion

In this paper, we have described how to implement dynamic space limits system for Haskell. Our system has not yet landed in GHC proper; the primary blockers are eliminating the overhead when resource limits are not being used and fixing scalability issues when there are a lot of resource containers live at the same time. We hope to integrate the patchset into the mainline in the not-so-distant future, if only as a flavor of GHC for resource limit minded users.

What is the applicability of this system beyond Haskell? Our system relies on three key features of Haskell and GHC: the ability to create multiple regions in the heap cheaply (a block-structured heap [14]), the ability to safely terminate threads (asynchronous exceptions [13]), and the ability to statically isolate code (restricted monads/Safe Haskell [24]). A language which has all three of these features would be able to adopt our system easily.[12]

While a block-structured heap is fairly general and could be implemented by most storage systems (making this paper an advertisement for the block-structured design), asynchronous exceptions and restricted monads are quite distinctive to Haskell. Purity in Haskell, for example, makes it more likely that arbitrary code can recover from being killed by an asynchronous exception. In languages like Java, killing a thread in a critical region can leave the system in an inconsistent state, and systems like Luna [10] take great care to avoid needing to kill threads. The situation for monads is similar: while in principle our monadic container API could be implemented in any language,[13] the inability to enforce its usage in the type system could lead to difficult to diagnose leaks of references. Still, we think that our system occupies a very interesting point in the design space, advocating for languages which provide these features.

Acknowledgments

We thank Deian Stefan and Stefan Heule for many useful comments and assistance in preparing the paper, as well as several anonymous referees. This work was funded by the DARPA Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH) program, BAA-10-70, under contract #N66001-10-2-4088, and the Department of Defense through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

References

[1] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, July 2005. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/1075382.1075383`.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[3] G. Czajkowski and T. von Eicken. JRes: a resource accounting interface for Java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, 2007.

[5] R. K. Dybvig, D. Eby, and C. Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical report, 1994.

[6] M. Fluet and G. Morrisett. Monadic regions. 16(4-5):485–545, July 2006. ISSN 0956-7968.

[7] M. Gaboardi and R. Péchoux. Upper bounds on stream I/O using semantic interpretations. In *Proceedings of the 23rd CSL international conference and 18th EACSL Annual conference on Computer science logic*, CSL'09/EACSL'09, pages 271–286, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.

[9] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*, 2002.

[10] C. Hawblitzel and T. von Eicken. Luna: a flexible Java protection system. *SIGOPS Oper. Syst. Rev.*, 36(SI):391–403, Dec. 2002. ISSN 0163-5980.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: a packet language for active networks. *SIGPLAN Not.*, 34(1):86–93, Sept. 1998.

[12] S. L. P. Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, 1992.

[13] S. Marlow, S. P. Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2006.

[14] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7.

[15] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[16] W. Partain. The nofib benchmark suite of Haskell programs. *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, 1992.

[17] D. W. Price, A. Rudys, and D. S. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, Washington, DC, USA, 2003. IEEE Computer Society.

[18] A. Reid. Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable black-holes. In *In Proc. IFL'98 (selected papers), volume 1595 of LNCS*, pages 186–199. Springer-Verlag, 1998.

[19] P. M. Sansom and S. L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 355–366, New York, NY, USA, 1995. ACM.

[20] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *In Symposium on Operating Systems Principles*, pages 170–185, 1999.

[21] G. Steele. Data representations in PDP-10 MACLISP. Technical report, MIT, 1977.

[22] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *16th Nordic Conference on Security IT Systems, NordSec*, volume 7161 of *LNCS*, pages 223–239. Springer, October 2011.

[23] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proc. of the 4th Symposium on Haskell*, pages 95–106, September 2011.

[24] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. Safe Haskell. *SIGPLAN Not.*, 47(12):137–148, Sept. 2012.

[25] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 119–128, New York, NY, USA, 2011. ACM.

[26] A. Wick and M. Flatt. Memory accounting without partitions. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 120–130, New York, NY, USA, 2004. ACM.

[27] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[12] In particular, our system handles lazy evaluation, but doesn't *require* it.

[13] If you aren't interested in running untrusted code, you don't even need Safe Haskell.