Resource Limits for Haskell

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

Abstract

We describe the first iteration of a resource limits system for Haskell, taking advantage of the key observation that resource limits share semantics and implementation strategy with profiling. We pay special attention to the problem of limiting resident memory usage: we describe a simple implementation technique for carrying out incremental heap censuses and describe a novel information-flow control solution for handling forcible resource reclamation. This system is implemented as a set of patches to GHC.

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

General Terms Languages, Reliability, Security

Keywords Resource Limits, Profiling, Fault Tolerance, Haskell

1. Introduction

One of the great benefits of functional languages is how they encourage modular reasoning. Pure functions without side-effects can be analyzed and tested in isolation, without regard to the larger context of the whole program. Several systems take such reasoning a step further, leveraging purity and type safety to reason about the behavior of potentially malicious code [7].

Unfortunately, even in a "pure" functional language such as the recently introduced Safe Haskell [21], functions still have side effects, namely they consume memory and CPU time in a globally visible way. In many systems, this resource consumption can impact correctness by inducing timeouts or even crashing a program by exhausting the stack or heap. Indeed, a look through the Haskell base library source reveals that much thought has gone into optimizations that have no bearing on the mathematical semantics of the functions, and for good reason: the perceived difficulty of controlling time and space consumption in extremely high-level languages is a perennial source of skepticism towards functional languages by imperative programmers.

On the other hand, Haskell comes with an excellent time and space profiler, which has long been used by developers to gain insight into the execution of a program. Conventional wisdom states that the profiler will only be used during the development of a program; when the program is actually deployed, it should be deployed in un-profiled form.

In this paper, we propose that resource limits and profiling parallel each other, both conceptually and in terms of implementation. To profile, one must define a *cost semantics* for the language, which specifies what execution costs get attributed to what source locations, as well as implementing a mechanism for measuring these costs and reporting them to the developer. To enforce resource limits, one must also define an assignment of costs to users of the system and also implement a mechanism for measuring these costs. But rather than report these costs to the developer, the mechanism instead takes action at runtime when some limits are exceeded. If your language has a profiler, you're already halfway down the road to resource limits.

What does the remainder of this road look like? First, you must deal with the fact that a profiler does not normally need to deal with adversarial code which may be actively attempting to hide its memory footprint or indirectly provoke high resource usage in a system: are your cost semantics truly adequate against a determined attacker? Next, you must figure out what to do after your limits have been broken and the culprit identified: you might decide to kill the thread in question, but what if references to the memory it allocated linger in other threads? Killing only that thread is no good. Finally, you have to make it run fast enough that developers are willing to use profiled code both for development and production.

This paper describes answers to all of these questions, in the form of a resource limits system for Haskell. Specifically, we make the following contributions:

- We extend Haskell's cost semantics to accommodate *dynamic cost centers*, dynamically allocated abstract entities to which costs are charged, allowing us to defer choice of who to attribute costs to until runtime. (Section 4) The setting of resource limits on untrusted, "black-box" code leads to the observation that is desirable to support both dynamic scoping (**scc**) and lexical scoping (**withcc**) as operators.
- We describe and implement an API for monitoring cost centers and invoking callbacks when these cost centers exceed predefined resource limits. (Section 3) This API is implemented as a set of patches on the Glasgow Haskell Compiler (GHC). One crucial implementation technique is *incremental heap census*, which amortizes the cost of tallying resource consumption against garbage collection. (Section 5.2)
- We propose a novel information-flow control solution for the "retainer" problem, which occurs when a heap value allocated by one cost center is retained by another cost center. (Section 3.3) Our solution balances the competing concerns of allowing a cost center to reclaim its memory and giving the guarantee that the memory pointed to by a reference they hold will always stay good. We do this by introducing *weak labeled references* to LIO [20], an existing library for performing IFC in Haskell.
- We articulate design principles for building systems which employ resource limits (Section 2.3) and give an evaluation of the

performance of GHC's existing profiling mechanisms and the accuracy of our system. (Section 6)

2. Background

To understand our implementation of resource limits, we must first give some background about Haskell's existing profiling support. We first give an informal explanation of Haskell's *cost semantics*, which specify how costs are attributed during profiling. Next, we briefly describe *Safe Haskell*, a language extension which guarantees type safety and referential transparency of compiled code. These two features give us the tools to implement resource limits. In the final section, we articulate what the goals of our resource limits system are.

2.1 Profiling in Haskell

Profiling in Haskell is performed by charging the costs of computation to the "current cost center." A *cost center* is an abstract, programmer-specified entity to which costs can be charged; only one is active per thread at any given time, and the *cost semantics* determines how the current cost center changes as the program executes. For example, the `scc cc e` expression (set-cost-center) modifies the current cost center during evaluation of `e` to be `cc`. Cost centers are defined statically at compile time.

A cost semantics for Haskell was defined by Sansom et al. [17] Previously, there had not been a formal account for how to attribute costs in the presence of lazy evaluation and higher-order functions; this paper resolved these questions. The two insights of their paper were the following: first, they articulated that cost attribution should be independent of evaluation order. For the sake of understandability, whether a thunk is evaluated immediately or later should not affect who is charged for it. Secondly, they observed that there are two ways of attributing costs for functions, in direct parallel to the difference between lexical scoping and dynamic scoping.

The principle of order-independent cost-attribution can be seen by this program:

```
f x = scc "f" (Just (x * x))
g x = let Just y = f x in scc "g" y
```

When `g 4` is invoked, who is charged the cost of evaluating `x * x`? With strict evaluation, it is easy to see that `f` should be charged, since `x * x` is evaluated immediately inside the `scc` expression. Order-independence dictates, then, that even if the execution of `x * x` is deferred to the inside of `scc "g" y`, the cost should *still* be attributed to `f`. In general, `scc "f" x` on a variable `x` is a no-op. In order to implement such a scheme, the current cost-center at the time of the allocation of the thunk must be recorded with the thunk and restored when the thunk is forced.

The difference between lexical scoping and dynamic scoping for function cost attribution can be seen in this example:

```
f = scc "f" (\x -> x * x)
g = \x -> scc "g" (x * x)
```

What is the difference between these two functions? We are in a situation analogous to the choice for thunks: should the current cost-center be saved along with the closure, and restored upon invocation of the function? If the answer is yes, we are using lexical scoping and the functions are equivalent; if the answer is no, we are using dynamic scoping and the `scc` in `f` is a no-op. The choice GHC has currently adopted for `scc` is dynamic scoping.[1]

For more detail, please refer to a reproduction of the original cost semantics in Appendix A.

2.2 Safe Haskell

Safe Haskell [21] is a language extension to Haskell which eliminates loopholes by which programs can bypass type safety and module encapsulation. Code compiled with Safe Haskell is guaranteed to be referentially transparent and encapsulation-preserving. The intent is that if you would like to run some untrusted code, you compile it with Safe Haskell, and if the compilation succeeds, the code is safe to run. Pure computation will be pure, and a *restricted IO* monad can be defined to only allow authorized IO actions.

The checks that Safe Haskell can perform are limited to the expressivity of GHC's type system. Because Haskell has no type system support for dealing with non-termination or allocation, Safe Haskell compiled code can still loop infinitely or allocate arbitrarily large amounts of memory. Furthermore, Safe Haskell cannot even prevent a user from running any function with type `a -> b`; it will only prevent the user from realizing the side effects of the function, in the case that `b` is `IO c`. Nevertheless, Safe Haskell type-system will still play a crucial role in both securing untrusted code, as well as securing the APIs by which we implement resource limits.

2.3 The Goal

Informally, a resource limits system should put a bound on the resource consumption of some fragment of code. Haskell's cost semantics offer one way of accounting for resource consumption: an example of a simple limit would be imposing an upper bound on the number of function applications a cost center can be charged, as specified by the cost semantics. However, this model is a simplification: for instance, it doesn't capture the fact that code in one cost center can induce resource consumption in another. A good resource limits system should provide protection against such cases.

Instead, we define our goal in terms of an *attack model*. Assume an attacker who has the ability to run a Safe Haskell compiled program with the current cost-center set to a cost-center allocated just for them. They have access to some number of APIs, including constants, pure functions and possibly a restricted IO monad. We now ask: can this program induce unbounded resource usage which is not attributed to itself?

Reviewing Haskell's cost semantics (Appendix A), we see that the only times when the current cost-center changes are when a thunk is evaluated and when another set-cost-center is encountered. (With evaluation scoping, function applications are always attributed to the attacker's cost center.) If the attacker is forbidden from using `scc` expressions, and no thunks in scope represents an infinite data-structure,[2] then an adversarial program can only induce a bounded amount of computation not attributed to themselves. This is done by evaluating these thunks: however, thunks can only be forced once, this resource consumption is bounded.

Beyond identifying when an attacker is consuming too many resources, we must also ensure that we can forcibly reclaim these resources. If a thread is allowed to pass references to other threads, one could end up in a situation where the originally allocating thread has been killed, but the resources remain unreclaimable.

In light of all of these issues, we will require the following:

1. It must not be possible for a program to induce unbounded resource usage that is not attributed to itself.

[1] The original paper advocated a hybrid of the two styles; the move to dynamic scoping is a recent change [15]. These changes were motivated by the fact that lexical scoping has some undesirable properties; it imposes restrictions on applicable optimizations and has bad interactions with constant-applicative forms (see Section 5.3)—as it turns out, dynamic scoping is also highly desirable for resource limits.

[2] Unfortunately, this requirement cannot be easily enforced by Safe Haskell. Our belief, however, is such a situation is relatively unusual.

2. It must be possible to induce all of the memory owned by a cost center to be freed. This operation may be invoked by both the trusted computing base and an untrusted user who is interested in reclaiming resources they paid for, but perhaps lost track of due to a space leak or errant reference.
3. It must not be possible to interfere with another thread without its consent. This requirement is in direct tension with the second requirement, as any thread holding a reference to data not owned by itself is liable to be killed in a reclamation event. What we'd like is a way to "opt-in" to the possibility of being killed.

3. Design

In this section, we describe an API which allows us to achieve the criteria we put forth in the previous section. We divide the API of our resource limits interface into three parts: *dynamic cost centers*, which allows us to dynamically allocate cost centers for instrumentation, *notification*, which is concerned with running callbacks when a limit on a cost center is exceeded, and *reclamation*, which is concerned with ensuring all memory used by a cost-center is eventually freed.

3.1 Dynamic cost centers

In the original cost semantics, cost-centers are always literals. In order to allow *dynamic* instrumentation of code (in the case of loading an untrusted module), we need the ability to allocate cost-centers and set the current cost-center with a non-literal cost-center. The core of our API is a mere two extra functions:

```
data CostCenter

newCC :: IO CostCenter
withCC :: CostCenter -> IO a -> IO a
```

`newCC` allocates a fresh cost-center while `withCC` takes a cost-center and runs an IO action, attributing its resource usage to this cost-center.[3] `withCC` is in the IO monad to enforce ordering.

Here is an example usage of this API:

```
main = do
    cc <- newCC
    withCC cc $ do
        x <- readInput
        print (x * x)
    main
```

In this example, the cost of evaluating `x * x` is attributed to the cost-center in `cc`, which is freshly allocated for each input: this would not be implementable using static `sccs`!

A keen reader may notice that this API is not symmetric with the original `scc` interface; a more detailed discussion of this can be found in Section 4.

3.2 Notification

Listeners are callbacks that trigger when resource limits are hit.

```
data Listener
data ProfType = Resident Int | Allocated Int | ...

listenCC :: CostCenter
         -> ProfType   {- performance metric -}
         -> IO ()      {- callback -}
         -> IO Listener
unlistenCC :: Listener -> IO ()
```

`listenCC` takes a cost-center to monitor, a metric to measure (e.g. resident memory or allocations) and a callback to run in a new thread if that metric is exceeded. A callback is only ever run once; otherwise, the system could be flooded with new threads repeatedly triggering from a limit violation. When the user-computation concludes and no limits have been hit, the listener should be unregistered with `unlistenCC` (an idempotent action). A simple example of combining listeners with dynamic cost centers is this bracketing function:

```
runWithLimit :: ProfType -> IO () -> IO ()
runWithLimit p m =
    cc <- newCC
    forkIO $ do
        tid <- myThreadId
        l <- listenCC cc p (killThread tid)
        m `finally` unlistenCC l
```

We expect there to be a bit of flexibility of both design and implementation of these functions (e.g. the precise details of `ProfType`), as these functions deal with implementation-specific details; as such, we elide any formal specification of their behavior.

3.3 Reclamation

The reclamation API is concerned with ensuring that after a computation has finished (or failed due to excessive resource consumption), all of the memory it allocated is freed.

Pure computation When the computation in question is pure, i.e. without side-effects, eliminating references to data owned by the cost-center is straightforward, since the pure computation cannot stash pointers to objects it allocates in references. The only way references can be leaked is through the result value, leading to two requirements:

1. The result of the computation must not contain any unevaluated thunks, as thunks can contain arbitrary references;
2. The result itself must be finite in size and attributed to a cost-center that is not the one that was allocated for the pure computation (e.g. the result is copied out).

A simple way to ensure both of these requirements are fulfilled is to require users to provide a *truncation function*—a function guaranteed to produce a finite amount of data—and eagerly copy the result, charging the allocations to the external cost-center:

```
newtype Trunc a = Trunc (a -> a)
trPair :: Trunc a -> Trunc b -> Trunc (a, b)
trList :: Int -> Trunc a -> Trunc [a]
-- ...

-- Deep copy on data types
class NFData a => Copyable a where
  copy :: a -> a

sandboxThunk :: Copyable a => Trunc a -> a -> IO a
sandboxThunk (Trunc k) x = evaluate (copy (k x))

sandboxFun :: Copyable b
           => CostCenter
           -> Trunc b
           -> (a -> b)  {- sandboxed function -}
           -> a         {- function argument -}
           -> IO b
sandboxFun ccs (Trunc k) f x =
```

[3] In modern GHC, resources are actually charged to cost-center stacks. For clarity, we elide this from this API.

```
  withCC ccs x (evaluate (f x)) >>= sandboxThunk k
-- sandboxFun generalizes in the usual way
```

Since users may want to set size limits dynamically in different contexts, truncation functions are built from a library of combinators, rather than inferred via type classes. Truncation functions force a user to specify exactly how much data they are willing to retain long-term: for *pure* short-term use of the data, the fold can be composed with the sandbox function and run *inside* the sandbox. (We address the impure case in the next section.)

There are a few subtleties in this API:

- `sandboxThunk` does not take a cost-center to charge its computation to: the cost-center charged for the thunk is already determined. However, in the case of `sandboxFun`, a new thunk is guaranteed to be allocated—thus, it must be assigned to an appropriate cost-center.
- The function argument of `sandboxFun` does *not* need to be copied: it will only be retained as long as the computation proceeds.
- In the case of performance-critical code, a carefully coded truncation function could perform all of the allocation and forcing necessary. The current API is intended to make it difficult for users to accidentally introduce unintended sharing.
- Exceptions can also hold onto references; exception handlers must either ignore the actual exception value, or truncate and copy it out in a similar fashion.
- `Copyable` is type-class which defines the ability to perform deep copies of data types; it could be implemented using `ghc-dup` [3] or entirely from userland as long as it is possible to prevent an optimizer from replacing the duplication with a no-op.

Impure computation While pure computation is inherently isolated, computation in the IO monad may leak references through mutable references or other channels of communication. Reclamation is now considerably more difficult, as we must be able to identify all possible retainers of an object.

We propose a novel mechanism based on information flow control (IFC). Traditionally, IFC systems track and restrict the flow of information. To this end, they associate *labels*, which encode security policies, with objects and ensure that labels propagate alongside the data they protect. In turn, the system can enforces individual micro-policies when objects are read or written. In our mechanism, a label on an object (e.g., a reference) encodes the cost-centers of all the objects that it may have a reference to.

Our system builds atop the Haskell LIO IFC system [20], which already associates a label with every thread, reference, mutable variable, etc. The thread label serves the role of protecting every "unlabeled" term in scope, as opposed to other language-based systems (e.g., Jif [12]) which are more coarse-grained and do not associate an explicit label with every term. The label of a thread is raised, i.e., the thread gets "tainted," when sensitive data is read.

Accordingly, the label of a thread reflects the call centers of all the references it has dereferenced. For example, if a thread C, which already dereferenced a reference to an object owned by cost center cc_a, subsequently dereferences a reference to an object owned by cost center cc_b, it's label will reflect this by adding cc_b to its label:

```
myThread r = do  -- Initial label: { cc_a }
  x <- readRef r -- New label:     { cc_a , cc_b }
  printVal x     -- Thread label:  { cc_a , cc_b }
```

This "taint tracking" feature of LIO is an obvious mechanism for determining who has references to what. However, since LIO is an IFC system, it *also* restricts writes according to labels (e.g., a secret thread cannot write to a public channel): this allows us to ensure that references are not accidentally leaked to threads who *didn't* want them.

Since LIO does not directly handle reclamation, we extend the library with weak labeled references and the ability to kill threads. The weak reference API is given below:

1. **Allocate:** `newRef :: a -> LIO (WeakRefHandle a, WeakRef a)` creates a new weak reference and returns a handle (used for deallocation) and the reference itself. Internally the reference is labeled according to the thread label, which reflects the call centers of all the objects that the object passed to `newRef` may point to. Importantly, the deallocation action is also added to the underlying monad state, which keeps a map between labels and deallocation actions.
2. **Read:** `readRef :: WeakRef a -> LIO a` returns the object that the reference points to, raising the label of the thread to reflect the dereference. If the weak reference has been deallocated, the function throws an exception.
3. **Read+Copy:** `readRefTrunc :: Copyable a => Trunc a -> WeakRef a -> LIO a` returns a copy of the object that the reference points to. Importantly, this does not raise the label of the thread.
4. **Write:** `writeRef :: WeakRef a -> a -> LIO ()` modifies the value that the reference points to, throwing an exception if the reference has been deallocated.
5. **Deallocate:** `freeRef :: WeakRefHandle a -> LIO ()` deallocates the supplied reference. In addition, any reference to an object that may depend on this reference is deallocated. This is implemented by looking up from the underlying map all the references whose labels "contain" the label of the supplied reference and executing the corresponding deallocation functions. Threads that may be actively using the reference are also killed. To this end, we use feature unique to LIO, called *clearance*. Clearance bounds the label of a thread, so that the execution context cannot access data above a certain label. Hence, when forking a thread, we require the programmer to specify the references they intend to use and set the clearance of the new thread to the upper bound on all the reference labels. When deallocating a reference, we inspect the clearance of all the threads and kill them if their clearance contains the label of the reference we are deallocating.

This system has the property that only threads that are tainted by the cost-center of a deallocated reference are terminated. Thus, if you want to inspect an untrustworthy reference, you fork a new handler thread and copy out the result only when you truly need it in the original context. This gives us the opt-in mechanism, which accomodates shared references *and* forced resource reclamation.

4. Semantics

In this section, we present modified cost semantics for a generalized dynamic cost center API we presented in Section 3.1. This is not formalism for formalism's sake: these cost rules impose some important constraints on how `withCC` and `scc` can be implemented. We elide semantics for Section 3.2 and Section 3.3; the former because its semantics are heavily implementation dependent, the latter as out of scope for the paper.

Our setting is a fragment of the lambda calculus in Figure 1. We omit all discussion of actual cost attributions and many conventional terms in the lambda calculus to focus on how the current cost-center evolves; all of the other details can be added in a straightforward way. The primary differences of our syntax are as

$$
\begin{array}{lcl}
e & ::= & \\
& | & \lambda x.e \\
& | & x \\
& | & cc \\
& | & \textbf{scc}\ e\ e' \\
& | & \textbf{newcc} \\
& | & \textbf{withcc}\ e\ e'
\end{array}
$$

Figure 1. Simplified language syntax

$$\frac{}{cc,\Gamma : cc_0 \Downarrow \Gamma : cc_0, cc}\ \text{CC}$$

$$\frac{cc,\Gamma : e \Downarrow \Delta : cc_0, cc' \qquad cc_0,\Delta : e' \Downarrow \Theta : z, cc''}{cc,\Gamma : \textbf{scc}\ e\ e' \Downarrow \Theta : z, cc''}\ \text{SCC}$$

$$\frac{cc_0\ \textbf{fresh}}{cc,\Gamma : \textbf{newcc} \Downarrow \Gamma : cc_0, cc}\ \text{NEWCC}$$

$$\frac{cc,\Gamma : e \Downarrow \Delta : cc_0, cc' \qquad cc,\Delta : e' \Downarrow \Theta : \lambda x.e'', cc''}{cc,\Gamma : \textbf{withcc}\ e\ e' \Downarrow \Theta : \lambda x.\textbf{scc}\ cc_0\ e'', cc''}\ \text{WITHCC}$$

Figure 2. New cost semantics rules

follows: cost-centers *cc* are now first-class values, **scc** now takes an expression for its first argument, and two new expressions **newcc** and **withcc** have been added.

The new cost semantics rules for these new constructs are described in Figure 2. The rule for **withcc** is worth closer consideration. Recall that we are using dynamic scoping for **scc**: in this light, **withcc** can be thought of as a way of using the lexical scoping rules for cost-centers: it pushes the **scc** inside of the lambda. What is the correspondence of `withCC` to **withcc**? Well, the `IO a` monad is really just `RealWorld -> (a, RealWorld)`; that is to say, a function!

For pragmatic reasons, we favor **withcc** over **scc** for our dynamic API:

- **withcc** is less likely to be misused, as there is less temptation to use it on a pre-existing thunk (which is a no-op).
- **scc** can be syntactically translated into **withcc** with only constant overhead: simply translate **scc** $e\ e'$ into **withcc** $e\ (\lambda x.e')\ ()$. The overhead is a single extra function application. We don't know of a way to the opposite transformation.
- **scc** is a new language construct and cannot be implemented as a simple function (in GHC, `sccs` are spelled using the `{-# SCC #-}` pragma, rather than a keyword which "looks" like a function). Absent optimization, **withcc** follows the evaluation rules of ordinary functions (the evaluation of e' uses cc rather than cc_0).

With optimizations, however, things are a bit more troublesome. In particular, GHC implements let-floating, where:

```
main = do
    cc <- newCC
    n <- getArg
    withCC cc (f n)
```

transforms into

$$\frac{ccs,\Gamma : e \Downarrow \Delta : \lambda y.e', ccs_1 \qquad ccs \star ccs_1, \Delta : e'[x/y] \Downarrow \Theta : z, ccs_2}{ccs,\Gamma : e\,x \Downarrow \Theta : z, ccs_2}$$

Figure 3. Modified application rule for cost-center stacks

```
main = do
    cc <- newCC
    n <- getArg
    let fn = f n
    withCC cc (f n)
```

The let has been floated out of the `withCC`: this optimization does not preserve the cost semantics of the program! To make matters worse, converting `withCC` into a built-in still leaves us with a composability problem: the definition

```
myWithCC cc m = withCC cc m
```

is not equivalent to the original.

To implementors, we only have two pieces of advice. The first is that with inlining, definitions like `myWithCC` can be made to work the majority of cases. The combinator is first inlined into the program, and the optimizer is able to see the call to `withCC` and avoid floating lets beyond it. The second is that there is a simple trick for preventing floating, where one defines a variant `withCC1`:

```
withCC1 :: CostCenter -> a -> (a -> IO b) -> IO b
```

with the usual extensions for arbitrary arity. Free variables of the monadic action are now converted into function arguments and, as `withCC1` is opaque to the optimizer, will not be floated outside.

5. Implementation

We have implemented our design in GHC. While much of the implementation relies on already available infrastructure, there are a number of nontrivial interactions with other features of the GHC which deserve some discussion.

5.1 Cost-center stacks

Our previous discussion has considered a simplified model involving only cost-centers. Modern GHC does not attribute costs to cost-centers; rather, costs are attributed to a generalized notion of *cost-center stacks*, which allow profiling information to more fine-grained information about what the call stack looked like before entering an `scc`.

Central to the generalization is a *cost-center stack combining function* $\star$, which takes two stacks upon function entry (the current stack of the dynamic scope, and the lexical stack saved with the function) and produces a new stack. If $ccs \star ccs' = ccs$, then dynamic scoping is being applied; if $ccs \star ccs' = ccs'$, then lexical scoping is being applied. A hybrid scheme appends the stacks together in an interesting way, to capture both lexical and dynamic information. The modified rule for function application can be found in Figure 3.

With cost-center stacks under the hood, one may wonder why such an interface is not exposed to the user in our implementation. In fact, they are (the true interfaces refer to cost-center stacks and not cost-centers). However, we found that for the purposes of enforcing resource limits, the extra expressivity afforded by a sophisticated stack combining function was both unnecessary and detrimental to an efficient implementation of resource checking. Costs are attributed to cost-center stacks, and in order to recover the aggregate costs for a cost-center, all of the stacks which may contribute to the cost must be traversed! Instead, we just use a

simple stack combining function, which discards its first argument and returns the second argument; in such a regime, cost-center stacks correspond directly to cost-centers with dynamic scoping.

5.2 Incremental heap census

GHC calculates resident memory usage by performing a *heap census*; that is, it walks the entirety of the heap and tallies up the memory usage of each cost-center. The heap census step is separate from garbage collection, allowing it to take advantage of superior data locality, avoid the expense of tracing and eliminate overhead when censuses are not enabled. In order to do this, however, the current heap census algorithm requires that objects on the heap be packed together, with no wasted space between objects (i.e. *slop*). Prior to this work, GHC enforced this invariant by inducing a major garbage collection before taking a census.

GHC uses a generational garbage collector, which means that under normal circumstances we would like to avoid performing major GCs as much as possible. Without heap censuses, however, we have no way of telling if a program is consuming too much memory. We are in a situation where performance tells us to run the census as infrequently as possible, while security tells us to run it as frequently as possible.

The solution to this problem is to carry out an *incremental* heap census. The idea is simple: convert the census algorithm into a generational one which caches census information from old generations and only traverse newly GC'd portions of the heap. The trickies part of this extension is handling objects which are promoted to old generations: they must be added to the cached census. While this information is trivial to reconstruct if the census is being taken *in* the garbage-collector, it is less easy to reconstruct after the fact. It can be done, however, by maintaining "fingers" that demarcate freshly garbage-collected portions of the generations from old sections. A census algorithm only needs to scan the parts of the generation which are new.

GHC, like many other garbage collectors, partitions the memory of its generations into blocks. Thus, the finger actually divides the blocks into old and new ones. The insides of blocks may also have old and new heap objects, so each block also must maintain a finger where the last garbage-collection ended. The heap census traverses the block chain using the outer finger, and each of the blocks inside the chain with the inner finger.

5.3 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. However, CAFs do not necessarily have to be defined in the top-level of a program—it just needs to be possible to lift them to the top-level.

For example, in the following program:

```
f x = let xs = [1..x] in product xs + sum xs
main = print (f (100000 :: Integer))
```

The expression `f 100000` is a CAF, despite its occurrence in `main`, seemingly to be computed at runtime. Floating the thunk to the top-level may seem to be of little use, since it has only one use-site and might be expected to only be used once. However, if `main` is invoked multiple times, then there are substantial savings to be had from memoizing its result. According to our cost semantics, CAFs accrue costs separately from the rest of the program, since in principle they only ever are evaluated once, and it shouldn't matter *who* ends up evaluating them.

There are a few important implications of CAFs. First, a Safe Haskell trusted module must not export a CAF representing an infinite data structure; these are essentially global thunks and can be used to induce non-attributable infinite allocation. Second, a system must be willing to pay for the space consumed by all evaluated CAFs accessible to untrusted code; in the absence of infinite data structures, this value is bounded by a constant. Finally, when dynamically loading code, resource limits must also be placed on the cost-center representing the CAFs of the dynamically loaded module. By default, GHC assigns the costs of computing all CAFs to a single cost-center per module, so this is simple in practice; nevertheless, it is a detail that must be attended to.

Program	Census	Valgrind
retainer	2,097,424	+4,227,072
opl [10]	2,160,568	+3,178,496
suml[4]	2,223,480	+6,324,224
tree [11]	2,317,360	+3,178,496

Figure 4. Self-reported and actual memory usage in bytes at time of callback, with a 512 KB allocation area size and 2 MB memory limit. The baseline memory usage reported by Valgrind when no computation is performed was 36,679,680 bytes. 1 KB = 1024 bytes.

Limit	Census	Valgrind
64 KB	65,880	0
128 KB	131,112	0
256 KB	262,312	0
512 KB	525,240	+32,768
1 MB	1,048,952	+1,081,344
2 MB	2,097,424	+4,227,072
4 MB	4,194,808	+9,469,952
8 MB	8,388,648	+19,955,712

Figure 5. Self-reported and actual memory usage for various resource limits on "retainer"; allocation area size was set to be a fourth of the limit. The shaded rows in this figure and Figure 4 correspond to identical runs of "retainer."

6. Evaluation

Does it work? To show that resource limits are effective at capping resource consumption by code, we ran a variety of allocating programs and measured how long it took for a listener to trigger and kill the process. With incremental heap censuses, we expect the margin of error to be at most the size of the allocation area, as specified by the runtime flag `-A` (when this area overflows, a garbage collection and heap census is triggered); when there are other threads performing allocation, this should be even smaller.

To keep us honest, we also measured resident memory usage from the operating system side using Valgrind Massif, to complement the self-reported amounts from the heap census (we subtract out the baseline memory usage). Our results are recorded in Figure 4.

As GHC uses a copying collector, the actual memory usage of these programs is approximately twice the amount recorded in the heap census; this space was used to perform a major GC some point in the past of the computation. Furthermore, GHC will not immediately release free memory back to the operating system: thus `suml` which might expect to only have 4.2 MB of resident memory (twice its census), actually holds onto 452 free blocks at the end of execution, resulting in Valgrind overstating GHC's actual usage by 1.8 MB.

In Figure 5, we show how resource limits scale up for larger amounts of memory on one particular test-case, "retainer":

Program	Allocs	Runtime	Elapsed	TotalMem
atom	+68.6%	+97.6%	+97.6%	+0.0%
awards	+65.1%	0.00	0.00	+0.0%
banner	+58.0%	0.00	0.00	+0.0%
bernouilli	+29.2%	+22.4%	+22.4%	+0.0%
binary-trees	+50.2%	+64.1%	+64.2%	+52.0%
cacheprof	+58.9%	+113.2%	+113.2%	+72.0%
fannkuch-redux	+66.7%	+15.4%	+15.4%	+0.0%
fasta	+40.4%	+21.5%	+21.4%	+0.0%
pidigits	+0.6%	+2.8%	+3.2%	+0.0%
power	+62.4%	+36.5%	+36.5%	+50.0%
spectral-norm	+40.5%	-0.2%	-0.3%	+0.0%

Figure 6. Changes in nofib results when turning on profiling

```
f n = let xs = [1..n::Integer]
      in sum xs * product xs
```

For extremely low limits, the memory usage is not perceptible due to the granularity of GHC's OS-level memory management; otherwise, things scale up nicely.

Performance The cost of turning on resource limits is dominated by the cost of running profiled versions of GHC-compiled code. To quantify these costs, we ran a series of measurements in computation-intensive benchmarks as well as IO-heavy benchmarks. Our experiments were conducted using a patched version of GHC 7.6.2, on a machine with two dual-core Intel Xeon E5620 (2.4Ghz) processors and 48GB of RAM.

Our first set of tests was the nofib benchmark suite [14] with and without profiling. A selection of results can be found in Figure 6.[5] Different programs respond quite differently to profiling, but there is one characteristic that is universal: profiled code results in *a lot* more allocation. This is due to the fact that every object on the heap must now store the cost-center that allocated it. In fact, GHC requires an extra two words to be added to every heap object, a hefty price to pay when the payload of many objects is only one word.

nofib benchmarks are artificially computation heavy, and one might wonder if things are any better on IO-bound workloads. For a more "real-world" comparison, we ran the Warp Cross-Language [19] benchmarks, which is a suite that measures the requests per second achieved by several popular Haskell web servers on a simple "pong" application. In addition to running profiled versions of web servers, we also measured the overhead of incremental heap census running on top of the profiled program. Alas, as can be seen in Figure 7, we still must pay the piper.

While a factor-of-two slowdown from profiling is a hefty one, it is not a deal breaker: compiled Haskell code continues to outperform interpreted languages, e.g. Goliath, a web server written in Ruby, as seen in Figure 8. Furthermore, improvements to the speed of resource limits will also give benefits to users of GHC's profiler, and vice versa; while the authors of GHC have attempted to preserve as many optimizations as possible when profiling is turned on, this coverage is by no means complete, and there is likely low hanging fruit that will dramatically speed up profiled code.

7. Related work

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

[5] Binary size change is omitted from the figure, but there is consistently a ~60% increase.

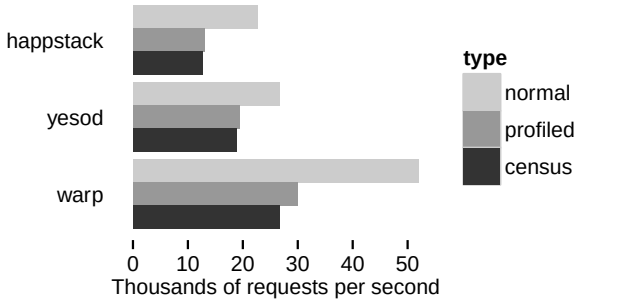

Figure 7. The effect of profiling on the performance of various Haskell web servers. Profiling induces a factor of two slow down; however, the additional overhead of heap censuses is negligible. We tested against Yesod 1.1.9.2, Happstack 7.0.1 and Warp 1.3.7.5.

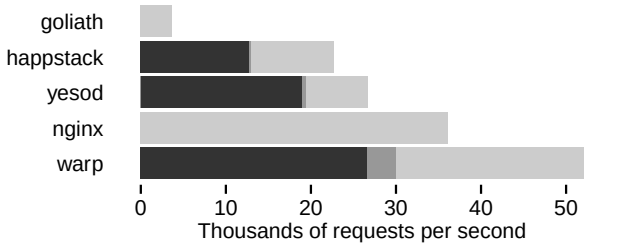

Figure 8. Comparison of Haskell web servers with servers written in other languages. The black bars reiterate the profiled versions of the servers, and are superimposed for easy visual comparison with other servers. We tested against Goliath 1.0.1 and nginx 1.2.7.

7.1 Operating systems

Starting with mechanisms as simple as the `limit` command, resource limits have been supported by POSIX-style operating systems for a very long time. 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 [23] organized space usage into containers with quotas. Threads were charged for quota usage by summing the quotas of all of the containers they have links to; if multiple threads reference a container, both are charged for it. These systems are *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. In HiStar, the quota usage of any object hard-linked into multiple containers must be *fixed*; otherwise, process A can induce arbitrary quota usage in process B by increasing the size of an object which is shared in multiple containers.

In contrast, EROS [18] 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 a forced reclamation must be handled at the language level.

7.2 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 [9] 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 is 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 [5]. Proof-based approaches include work by Gaboardi and Péchoux [6], which develop proof 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 [13]. 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 been towards resource limits for Java, since Java is perhaps the most widely used programming language that also has some ability to run untrusted code. These systems include JRes [4], Luna [8] and KaffeOS [1]. These systems utilize a variety of implementation mechanisms, including dynamic code translation in order to record allocation events and weak references to detect when memory is deallocated.

The systems most similar to ours for handling memory usage are those proposed by Wick et al. [22] and Price et al. [16]. Both utilize modifications to the garbage collector of a language, by fully tracing the set of objects retained by a thread in order to determine its resource consumption, making them consumer-based systems.

The switch from consumer-based and producer-based systems has many consequences, not all for better. As we have to record the individual who produced the data on all heap objects, we pay a much more substantial relative cost for enabling profiling. On the flip side, our accounting algorithm requires no modifications to the garbage collector, is completely deterministic (both systems suffer from non-determinism when two threads share the same data), and has a much clearer semantic interpretation in the presence of laziness and higher-order functions. Additionally, neither of these papers draws the parallel between resource limits and profiling: we believe this will be an important factor for language implementors who may not want to make accommodations for resource limits, but recognize the importance of profiling support in a language.

8. Future work

Our system has not yet landed in GHC proper; however, the patches are quite minimal and we hope to integrate them into the mainline in the near future and get some on the ground feedback. Additionally, we have in mind some performance improvements for profiling which help reduce the overhead of our scheme (starting with reducing the profiling overhead from two words per object to one). On more theoretical grounds, the problem of optimizations changing the asymptotic resource usage of a program is a well recognized one: the *full laziness* transformation, for example, is known to cause data structures to be retained for too long. A principled way of selectively disabling these optimizations which encapsulates our let-floating technique and other problems would be quite interesting.

$$
\begin{array}{lll}
e & ::= & \\
 & | & \lambda x.e \\
 & | & e\ x \\
 & | & x \\
 & | & \textbf{let}\ x_1 = e_1\ ..\ x_n = e_n\ \textbf{in}\ e \\
 & | & C\ x_1\ ..\ x_n \\
 & | & \textbf{case}\ e\ \textbf{of}\ \{\ \overline{C_j\ v_j \to e_j}\ \} \\
 & | & \textbf{scc}\ cc\ e
\end{array}
$$

Figure 9. Language syntax

9. Conclusion

Even purely functional code has side effects in the form of memory and CPU consumption. To control the impact of these side-effects requires the ability to reason about and enforce resource limits. Indeed, bounding resource consumption is crucial to any fault tolerant system or any system that hopes to confine and execute untrusted code. Ensuring resource bounds is non-trivial in the presence of lazy evaluation and data sharing. Fortunately, there is a practical solution, which this paper presents: *when a language implementation provides a well-designed profiling system, the profiling facility can be re-purposed to realize resource limits.* Using this technique, we present and evaluate an implementation of resource limits for Haskell in the GHC compiler. Though still a bit tricky to use and somewhat expensive, our evaluation shows the approach is squarely practical and ripe for optimization. With further improvement, better control over resource consumption can address a major concern of imperative programmers considering the switch to functional languages.

A. Appendix: Original cost semantics

Reproduced for convenience from [17].

Figure 9 gives the syntax of a lambda calculus with mutual recursion and constructors. The primary twist is that function application can only be performed on variables—thus allowing us to easily model heap allocation as let-binding. Values z in our language are λ-abstractions and (saturated) constructor applications.

Values on the heap are mappings $x \overset{cc}{\mapsto} e$ from variable names x to expressions e (possibly unevaluated), with an associated cost-centre cc. Cost-attributions θ are mappings from cost centres to costs c; costs are modeled abstractly as units of different types: application (A), case expression (C), thunk evaluation (V), thunk update (U) and heap allocation (H).

The big-step cost semantics in Figure 10 map a current cost centre cc, a heap Γ and an expression e to a new heap Δ, a value z and a new current cost centre cc', recording the costs of the execution as θ.

SUB is used to accomodate top-level functions, and is defined as:

$$
\begin{aligned}
\mathrm{SUB}(\text{“SUB”}, cc) &= cc \\
\mathrm{SUB}(cc_z, cc) &= cc_z
\end{aligned}
$$

$$\frac{}{cc, \Gamma : \lambda y.e \ \Downarrow_{\{\}} \ \Gamma : \lambda y.e, cc} \quad \textsc{Lambda}$$

$$\frac{}{cc, \Gamma : C\ \overline{x_i} \ \Downarrow_{\{\}} \ \Gamma : C\ \overline{x_i}, cc} \quad \textsc{Constructor}$$

$$\frac{cc, \Gamma : e \ \Downarrow_{\theta_1} \ \Delta : \lambda y.e', cc_1 \qquad cc, \Delta : e'[x/y] \ \Downarrow_{\theta_2} \ \Theta : z, cc_2}{cc, \Gamma : e\ x \ \Downarrow_{\{cc \mapsto \mathrm{A}\} \cup \theta_1 \cup \theta_2} \ \Theta : z, cc_2} \quad \textsc{Application}$$

$$\frac{x \overset{cc_z}{\mapsto} z \ \mathbf{in} \ \Gamma}{cc, \Gamma : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\}} \ \Gamma : z, \mathrm{SUB}(cc_z, cc)} \quad \textsc{Var(whnf)}$$

$$\frac{cc_e, \Gamma : e \ \Downarrow_{\theta} \ \Delta : z, cc_z}{cc, \Gamma[x \overset{cc_e}{\mapsto} e] : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\} \cup \theta \cup \{cc_e \mapsto \mathrm{U}\}} \ \Delta[x \overset{cc_z}{\mapsto} z] : z, cc_z} \quad \textsc{Var(thunk)}$$

$$\frac{cc, \Gamma[\ \overline{y_i \overset{cc}{\mapsto} e_i[y_i/x_i]}\ ] : e[\ \overline{y_i/x_i}\ ] \ \Downarrow_{\theta} \ \Delta : z, cc_z}{cc, \Gamma : \mathbf{let}\ \overline{x_i = e_i}\ \mathbf{in}\ e \ \Downarrow_{\{cc \mapsto \mathrm{H}\} \cup \theta} \ \Delta : z, cc_z} \quad \textsc{Let}$$

$$\frac{cc, \Gamma : e \ \Downarrow_{\theta_1} \ \Delta : C_k\ x_1\ ..\ x_{a_k}, cc_C \qquad cc, \Delta : e_k[x_i/y_{k\,i}] \ \Downarrow_{\theta_2} \ \Theta : z, cc_z}{cc, \Gamma : \mathbf{case}\ e\ \mathbf{of}\ \{\ \overline{C_j\ y_{j_1}\ ..\ y_{j_{a_j}} \to e_j}\ \} \ \Downarrow_{\{cc \mapsto \mathrm{C}\} \cup \theta_1 \cup \theta_2} \ \Theta : z, cc_z} \quad \textsc{Case}$$

$$\frac{cc_{scc}, \Gamma : e \ \Downarrow_{\theta} \ \Delta : z, cc_z}{cc, \Gamma : \mathbf{scc}\ cc_{scc}\ e \ \Downarrow_{\theta} \ \Delta : z, cc_z} \quad \textsc{SCC}$$

Figure 10. Original cost semantics

Acknowledgments

We thank Deian Stefan for many useful comments and assistance in preparing the paper. 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.

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. doi: 10.1145/1075382.1075383. 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] J. Breitner. dup – explicit un-sharing in haskell. *Haskell Implementors Workshop*, 2012.

[4] 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. doi: 10.1145/286936.286944. URL `http://doi.acm.org/10.1145/286936.286944`.

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

[6] 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.

[7] 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.

[8] 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. doi: 10.1145/844128.844164. URL `http://doi.acm.org/10.1145/844128.844164`.

[9] 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.

[10] Ingdas. Space leak in list program, 2010. URL `http://stackoverflow.com/questions/3190098/space-leak-in-list-program`.

[11] P. Jankuliak. How to reason about space complexity in Haskell, 2011. URL `http://stackoverflow.com/questions/5552433/how-to-reason-about-space-complexity-in-haskell`.

[12] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

[13] 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.

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

[15] S. L. Peyton Jones and S. Marlow. GHC status update, 2011. URL `http://www.youtube.com/watch?v=QBFtnkb2Erg`.

[16] 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, pages 263–, Washington, DC, USA, 2003. IEEE Computer Society.

[17] 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.

[18] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. pages 170–185.

[19] M. Snoyman. Preliminary Warp cross-language benchmarks, March. URL `http://www.yesodweb.com/blog/2011/03/preliminary-warp-cross-language-benchmarks`.

[20] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

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

[22] 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.

[23] 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.