Practical Fully Relocating Garbage Collection in LLVM

Philip Reames, Sanjoy Das

Azul Systems

Oct 28, 2014

This is a talk about how LLVM can better support garbage collection.

It is *not* about how write an LLVM based compiler for a garbage collected language.

About Azul

We have one of the most advanced production grade garbage collectors in the world.

If you're curious:

- The Pauseless GC Algorithm. VEE 2005
- C4: The Continuously Concurrent Compacting Collector. ISMM 2011

This presentation describes advanced development work at Azul Systems and is for informational purposes only. Any information presented here does not represent a commitment by Azul Systems to deliver any such material, code, or functionality in current or future Azul products.

A GC Overview

Late Insertion

Statepoints

Garbage Collection: 101

- Objects considered live if reachable
- Roots include globals, locals, & expression temporaries
- “Some” collectors move objects

Compiler Cooperation Needed!

The challenges:

- Identifying roots for liveness
- Updating heap references for moved objects
- Ensuring application can make timely progress
- Intercepting (some) loads and stores

Parseable thread stacks

- thread stacks are “parseable” when the GC knows where all the references are
- stacks are usually parsed using a *stack map* generated by the compiler

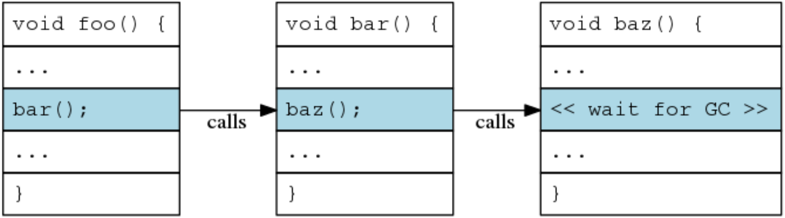
Introducing safepoints

How to give the GC a parseable thread stack?

- keeping stacks parseable at all times is too expensive
- make stacks parseable at points in thread's instruction stream called **safepoints** and ...
- ... make a thread be at a safepoint when needed

Safepoints and parseability

A thread at a safepoint



- the youngest frame is in a parseable state
- older frames, now frozen at a callsite, are parseable

Safepoints and polling

Usually

- GC requests a safepoint
- threads periodically **poll** for a pending request
- and, if needed, come to a safepoint in a "reasonable" amount of time

Where might you poll?

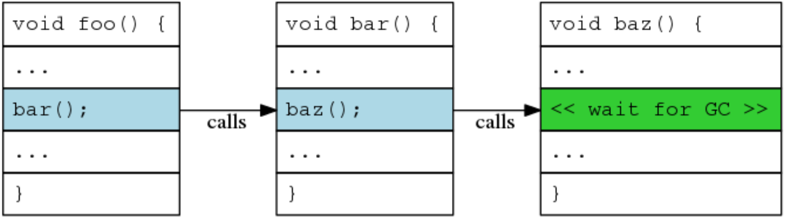
"reasonable" is a policy choice. Some typical places to poll:

- method entries or exits
- loop backedges

Safepoint polls can inhibit optimization

From the compiler's perspective

Two main concepts:



- parseable call sites
- parseable safepoint polls

From the compiler's perspective

Objects relocations become visible when a safepoint is taken. The compiler must assume relocation can happen during any parseable call or safepoint poll.

A GC Overview

Late Insertion

Statepoints

Assume for the moment, we can make all that work.

What effect does this have on the optimizer?

We'll come back to the *how* in a bit..

Example

```
void foo(int* arr, int len) {
  int* p = arr+len;
  while(p != arr) {
    p--;
    *p = 0;
  }
}
```

This loop is vectorizable.

Unfortunately, not after safepoint poll insertion...

Early Safepoint Insertion

```
void foo(int* GCPTR arr, int len) {
  int* GCPTR p = arr+len;
  while(p != arr) {
    p--;
    *p = 0;
    ... safepoint poll site ...
  }
}
```

What does that poll site look like to the optimizer?

Early Safepoint Insertion

```
void foo(int* GCPTR arr, int len) {
  int* GCPTR p = arr+len;
  while(p != arr) {
    p--;
    *p = 0;
    (p, arr) = safepoint(p, arr);
  }
}
```

p and **arr** are unrelated to **p** and **arr**. The loop is no longer vectorizable.

How to resolve this?

- Option 1 - Make the optimizer smarter
  - Adds complexity to the optimizer
  - Long tail of missed optimizations
  - Or, worse, subtle GC related miscompiles

Safepoint polls prevent optimizations *by design*

How to resolve this?

- Option 1 - Make the optimizer smarter
  - Adds complexity to the optimizer
  - Long tail of missed optimizations
  - Or, worse, subtle GC related miscompiles
- Option 2 - Insert poll sites *after* optimization

Safepoint polls prevent optimizations *by design*

Early vs Late Insertion

```
$ # Option 1
$ opt -place-safepoints -O3 foo.ll
```

vs

```
$ # Option 2
$ opt -O3 -place-safepoints foo.ll
```

Late Insertion Overview

Given a set of future poll sites:

1. distinguish references from other pointers
2. identify potential references live at location
3. identify the *object* referenced by each pointer
4. transform the IR

AZUL SYSTEMS®

Distinguishing *references*

The source IR may contain a mix of references, and pointers to non-GC managed memory

- Runtime structures, off-heap memory, etc..

Two important distinctions:

- Pointer vs other types
- gc-reference vs pointer

Distinguishing *references*

Using address spaces gives us this property

- Disallow coercion through inttoptr and addrspacecast or in memory coercion

Distinguishing *references*

In practice, LLVM's passes do not introduce such coercion constructs if they didn't exist in the input.

And there are good reasons for them not to.

Finding references which need relocated

Just a simple static liveness analysis

Aside: When relocation isn't needed

Depending on the collector, not every reference needs to be relocated. For example, relocating null is almost always a noop.

Other examples might be:

- References to pinned objects
- References to newly allocated objects
- Constant offset GEPs of relocated values
- Non-relocating collectors ☺

Note: Liveness tracking still needed.

Terminology: Derived Pointers

```
Foo* p = new Foo();
int* q = &(p->field);
...safepoint...
*q = 5;
```

Terminology: Derived Pointers

Given a pointer in between two objects, how do we know which object that pointer is offset from?


```
int* p = new int[1]{0};
int* q = p + 1;
...safepoint...
int* p1 = q - 1;
*p1 = 5;
```

AZUL SYSTEMS

What about base pointers?

Figuring out the base of an arbitrary pointer at compile time is *hard*..

```
int* p = end+3;
while(p > begin) {
  ...
  if( condition ) {
    p = foo();
  }
}
```

Thankfully, we only need to know the base object at *runtime*. We can rewrite the IR to make sure this is available at runtime, and record where we should look for it.

We'll create something like this:

```
int* p = end+3;
int* base_p = begin;
while(p > begin) {
  ...
  if( condition ) {
    p = foo();
    base_p = p;
  }
}
```

We'll create something like this:

```
int* p = end+3;
int* base_p = begin;
while(p > begin) {
  ...
  if( condition ) {
    p = foo();
    base_p = p;
  }
}
```

But for SSA...

The base of 'p'

Assumptions:

- arguments and return values are base pointers
- global variables are base pointers
- object fields are base pointers

A few simple rules

- baseof(gep(p, offset)) is baseof(p)
- baseof(bitcast(p)) is bitcast(baseof(p))

What about PHIs?

What about PHIs?

Each PHI can have a "base phi" inserted.

```
bb1:
  p1 = ...
  p1_base = ...
  br bb2
bb2:
  p = phi(p1 : bb1, p_next : bb2)
  p_base = phi(p1_base, p_base)
  ...
  p_next = gep p + 1
  br bb2
```

AZUL SYSTEMS

What about PHIs?

```
bb1:
  p1 = ...
  p1_base = ...
  br bb2
bb2:
  p = phi(p1 : bb1, p_next : bb2)
  (p_base == p1_base)
  ...
  p_next = gep p + 1
  br bb2
```

A case of dead PHI removal (but with safepoints)

Safepoint Poll Insertion

We now know:

- The insertion site
- The values to be relocated
- The base pointer of each derived pointer

This is everything we need to insert a safepoint with either gcroot or statepoints.

Safepoint Verification

SSA values can not be used after being potentially relocated. Applications for the verifier:

- frontend authors doing early insertion
- validating the results of the late insertion code
- validating safepoint representations against existing optimization passes

The verifier may report some false positives. e.g.

```
safepoint(p)
icmp ne p, null
```

Restrictions on Source Language

- Conversions between references and non-GC pointers are disallowed
- Derived pointers can't escape
- IR aggregate types (vector, array, struct) with references inside aren't well supported

Back to our example

```
void foo(int* arr, int len) {
  int* p = arr+len;
  while(p != arr) {
    p--;
    *p = 0;
  }
}
```

With no changes to the optimizer and our new safepoint insertion pass, we can run:

```
opt -O3 -place-safepoints example.ll
```

Runtime of our example

```
$ ./example.nosafepoints-O0.out
real 0m10.077s

$ ./example.nosafepoints-O3.out
real 0m2.180s

$ ./example.early-O3.out
real 0m10.702s

$ ./example.late-O3.out
real 0m2.167s
```

A simple observation

While we've described the transformation in terms of safepoint poll sites, the same techniques work for *parseable calls* as well.

This can enable somewhat better optimization around call sites, particularly w.r.t. aliasing.

A GC Overview

Late Insertion

Statepoints

Representing safepoints in LLVM IR

In a way that

- transforms that break safepoint semantics also break llvm IR semantics
- it admits a range of lowering strategies
- it is easy to optimize safepoints post insertion

llvm.gcroot

references are “boxed” around parseable calls and polls

```
%box = alloca i8*
call void @llvm.gcroot(i8** %box, i8*
    null)
...
store %ref, %box
call void @block()
%ref.r = load %box
```

llvm.gcroot

However ...

- keeping references in registers does not follow naturally
- we have to track memory to do safepoint optimizations

gc.statepoint

- one level more abstract than `llvm.gcroot`
- tries to be semantic, not operational
- explicitly encodes base pointers

Our late safepoint insertion and verification passes work on this

gc.statepoint

Our implementation is a set of “GC intrinsics” we add to llvm:

- `gc.statepoint` – clobbers heap, relocates tuple of references
- `gc.relocate` – projection function

gc.statepoint

```
%token = call i32 @gc.statepoint(
   call_target,
   < call args >, < heap refs >)
%ref_i.relocated =
   call i8* @gc.relocate(%token, %ref_i,
                                 %base_of_ref_i)
```

Future Work

- Relocation Optimizations
  - See list from previous slide
- Statepoint Infrastructure
  - Inlining of statepoints
  - References in callee saved registers
- Default Polling Strategy
  - Call in loop, Inner loop chunking
  - Leaf functions

Help wanted! Please review!

Conclusions

- Late insertion of safepoints (and barriers)
- Minimal impact on the compiler
- Doesn't limit any existing IR optimization

github.com/AzulSystems/llvm-late-safepoint-placement

reviews.llvm.org/D5683

Conclusions

- Late insertion of safepoints (and barriers)
- Minimal impact on the compiler
- Doesn't limit any existing IR optimization

Questions?

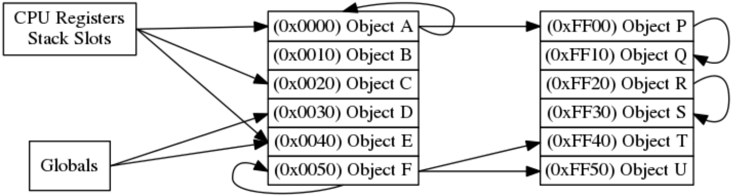
Backup Slides

Warning: These backup slides are mostly things which didn't make into the actual deck. We included them for distribution since they make some interesting points, but they're also decidedly rough. These slides are fairly likely to contain accidental mistatements or bugs.

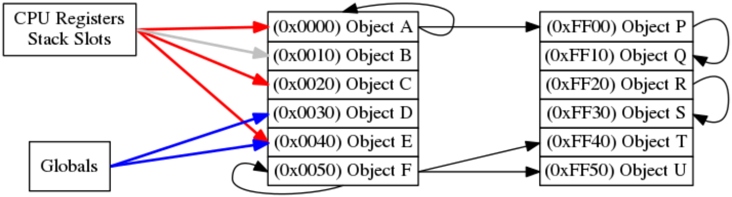
What's a safepoint poll?

```
define void @gc.safepoint_poll() #6 {
entry:
  %safepoint_needed = ...
  br i1 %safepoint_needed, label %
     do_safepoint, label %done

do_safepoint:
  ...
  call void @"YourRuntime::do_safepoint
     "()
  ...
  br label %done

done:
  ret void
}
```

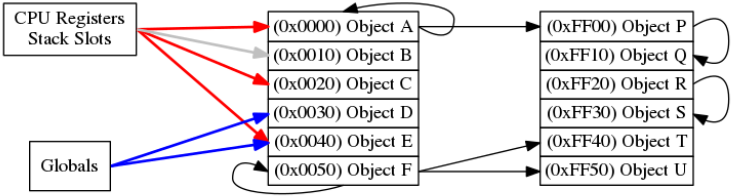
How a GC sees the world



Identifying Roots

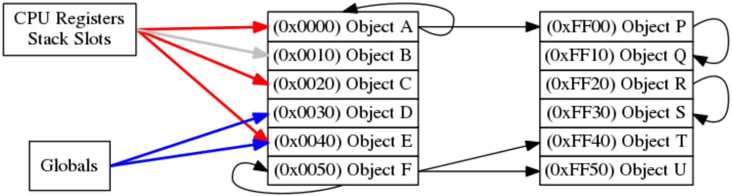


Identifying Roots



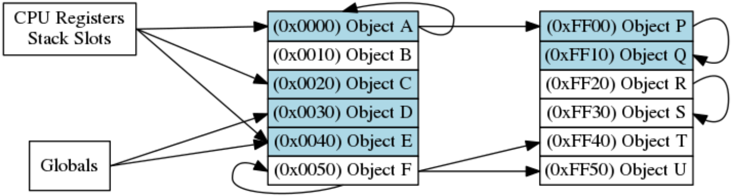
- A conservative GC might falsely identify roots that aren't actually pointers. A precise one will not.

Identifying Roots

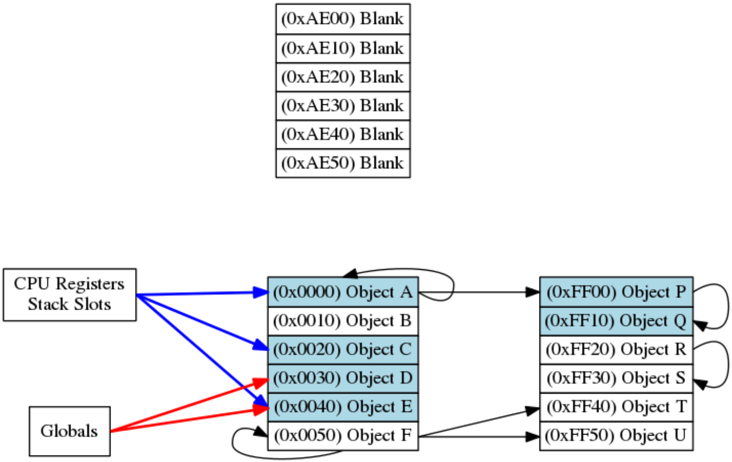


- A conservative GC might falsely identify roots that aren't actually pointers. A precise one will not.
- Root identification is done with the thread stopped at a well defined place. This makes *call sites* interesting.

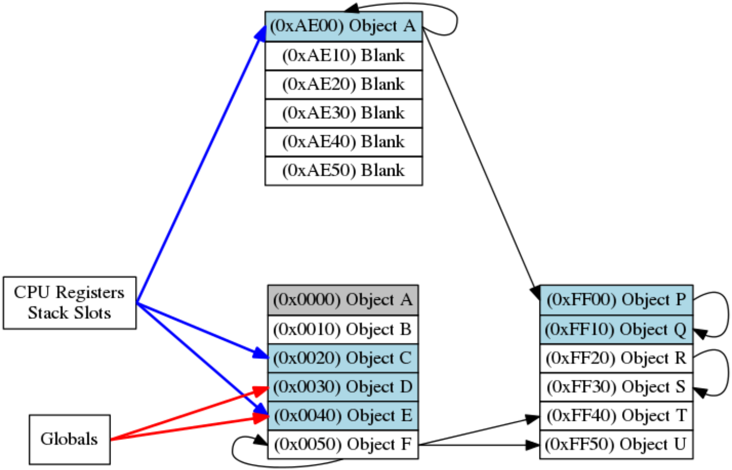
Figuring out what's live



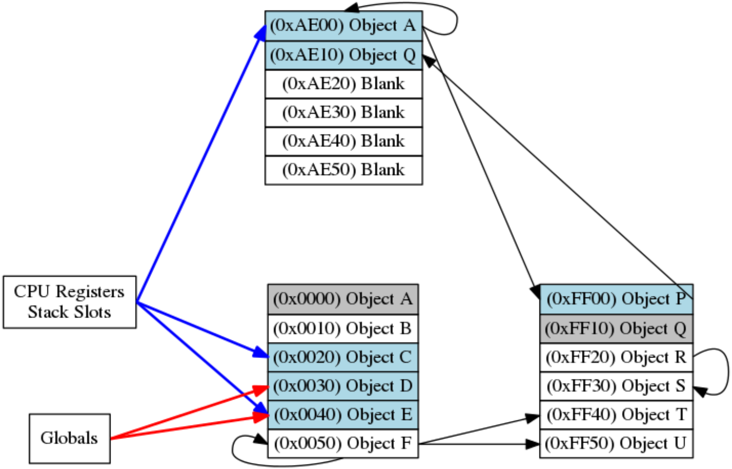
Relocating GC



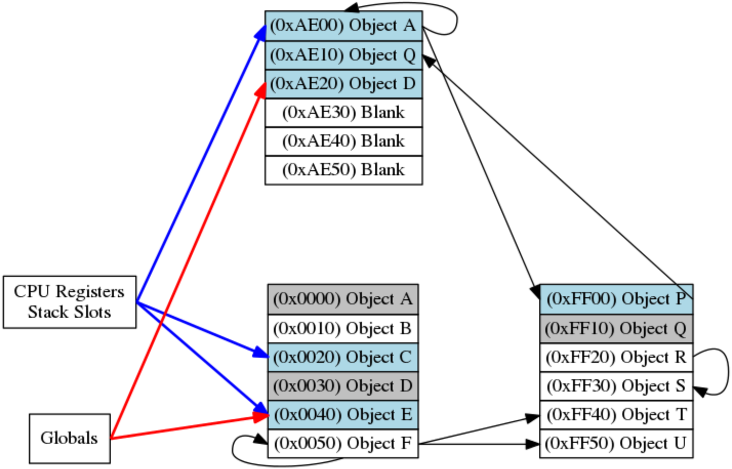
Relocating GC



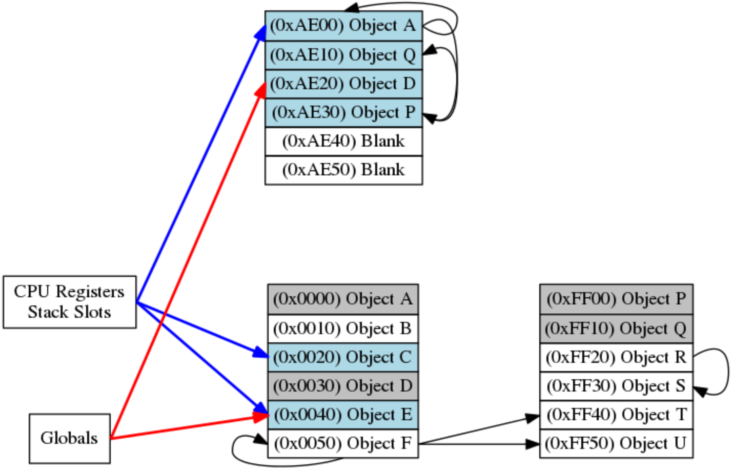
Relocating GC



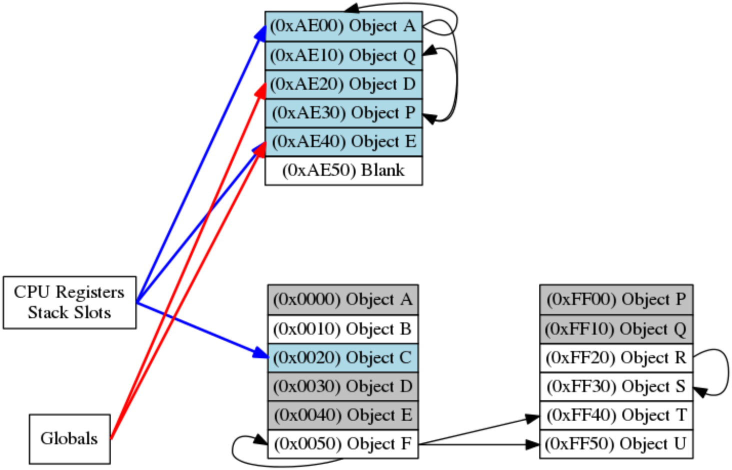
Relocating GC



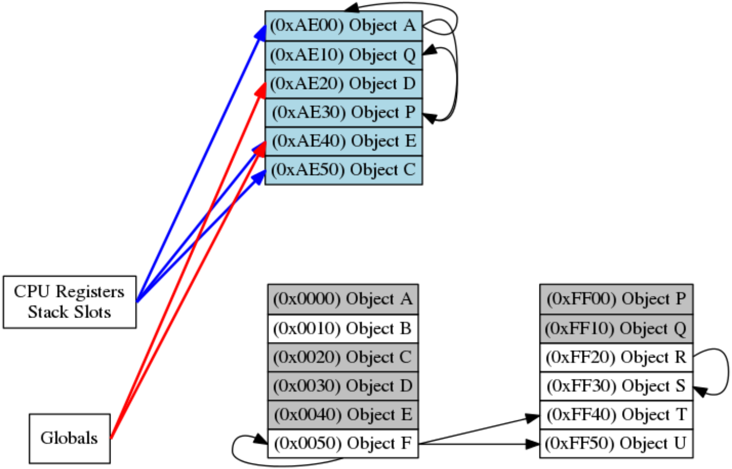
Relocating GC



Relocating GC



Relocating GC



What cannot be

```
void @foo(i32* %arr, i32 %len) {
 ...
b2:
  %p = phi [%p.0, %b],[%p.dec, %b4]
  %c = icmp ne %p, %arr
  br %c, label %b4, label %b6
b4:
  %p.dec = getelementptr %p, -1
  store i32 0, %p.dec
 ... safepoint poll site ...
  br label %b2
 ...
}
```

What cannot be

```
void @foo(i32* %arr, i32 %len) {
 ...
b2:
  %p = phi [%p.0, %b],[%p.dec, %b4]
  %c = icmp ne %p, %arr
  br %c, label %b4, label %b6
b4:
  %p.dec = getelementptr %p, -1
  store i32 0, %p.dec
  call void @parse_point(%p.dec, %arr)
  br label %b2
 ...
}
```

What cannot be

```
void @foo(i32* %arr, i32 %len) {
 %arr.0 = getelementptr %arr, 0
 ...
b2:
  %p = phi [%p.0, %b],[%p.dec, %b4]
  %c = icmp ne %p, %arr.0
  br %c, label %b4, label %b6
b4:
  %p.dec = getelementptr %p, -1
  store i32 0, %p.dec
  call void @parse_point(%p.dec, %arr)
  br label %b2
 ...
}
```

The Statepoint Artifact

- the first half of the problem: adequately representing parse-points in llvm IR
- in way that optimizations don't break parse-point semantics.
- semantics follow from constituent parts, not a new IR instruction with weird semantics, for example.

AZUL SYSTEMS

Statepoints: motivation

- so, um, we just need a way to tell the GC about the heap references in my frame, right?
- how about the most obvious thing – a function call whose sole purpose is to “remember” a set of heap references?

```
%r0 = ...
%r1 = ...
call void @parse_point(i8* %r0, i8* %r1
    )
call void @use(i8* %r0)
```

- ... and some lowering magic to discover what registers or stack slots `%r0` and `%r1` end up in at the call to`@parse_point`.

Statepoints: motivation

- this approach doesn't work for a relocating GC.

Statepoints: motivation

- this approach doesn't work for a relocating GC.
- consider this "meaning preserving" transform:

  From

```
%r0 = ...
%r1 = ...
call void @parse_point(i8* %r0, i8* %r1
    )
call void @use(i8* %r0)
```

  To

```
%r0 = ...
%r1 = ...
%r2 = getelementptr i8* %r0, 0 ;; COPY
call void @parse_point(i8* %r0, i8* %r1
    )
call void @use(i8* %r2)
```

Statepoints: motivation

- this approach doesn't work for a relocating GC.
- consider this "meaning preserving" transform:

From

```
%r0 = ...
%r1 = ...
call void @parse_point(i8* %r0, i8* %r1
    )
call void @use(i8* %r0)
```

To

```
%r0 = ...
%r1 = ...
%r2 = getelementptr i8* %r0, 0 ;; COPY
call void @parse_point(i8* %r0, i8* %r1
    )
call void @use(i8* %r2)
```

Statepoints: motivation

- this approach doesn't work for a relocating GC.
- consider this "meaning preserving" transform:

From

```
%r0 = ...
%r1 = ...
call void @parse_point(i8* %r0, i8* %r1
    )
call void @use(i8* %r0)
```

To

```
%r0 = ...
%r1 = ...
%r2 = getelementptr i8* %r0, 0 ;; COPY
call void @parse_point(i8* %r0, i8* %r1
    )
call void @use(i8* %r2)
```

Statepoints: motivation

We broke SSA! SSA values are forever – they can't be changed or relocated "in place".

Statepoints: motivation

To fix this, we make the relocation explicit. Our original example now looks like

```
%r0 = ...
%r1 = ...
%tuple = call tuple_ty @parse_point(i8* %r0
    , i8* %r1)
%r0.relocated = project %tuple, %r0
call void @use(i8* %r0.relocated)
```

The original problem disappears – we've effectively communicated that `@use` sees a value *different* from `%r0`. This is conservative since it admits semantics other than `%r0` is relocated to `%r0.relocated`.

Statepoints: correctness

Parse-point semantics are *admissible* in the above scheme. Hence, llvm cannot do transforms that invalidate parse-point semantics.

Statepoints: optimizations

We model parse points conservatively, so not may optimizations kick in. However, certain operations are "relocation agnostic", and we can exploit that to optimize IR with statepoints (R is "relocated version of"):

- $t = null \Leftrightarrow R(t) = null$
- $t \neq null \Leftrightarrow R(t) \neq null$
- $t = s \Leftrightarrow R(t) = R(s)$
- $t \neq s \Leftrightarrow R(t) \neq R(s)$
  - Note that $t \neq s \not\Leftrightarrow t \neq R(s)$