Université de Montréal

Type Checking a Garbage Collector

Stefan Monnier

DIRO - Université de Montréal

Software reliability and security

Avoiding and detecting bugs is done in many ways

- Good design: Analysis and decomposition of the overall problem
- Good organization: Enough time, enough sleep, motivation, pride
- Good code: Make sure the code does what it's intended to do

We need and usually use them all

Good code

Code quality is checked in various ways

- Code review by humans
- Tests: execute the code and check its *dynamic* behavior
- Verification: analyze the code *without* running it

We need them all, and often use them all

Program verification

Here as well, many techniques exist

- Hoare logic: Encode in logic the semantics of the language, the specification of the program, then prove correctness
- Automatic program derivation: Write a clean and naive version of the program, plus automatically-verified transformations
- Model checking: Abstract a model of some aspect of the code; exhaustively and automatically check it against a specification
- Ad-hoc analyses: Look for known error patterns; often imprecise
- Type checking: *modular* verification against a predefined set of rules

Usually can't be all used at the same time

This week's menu

3 parts to try and show what advanced type systems can do

- Type checking a GC: based on a PLDI'03 paper
- Type-preserving compilation: based on an ICFP'08 paper
- Design of Typer: on-going work

First two taken as (somewhat out-dated) experiments

Use GC and compilation as examples, because that's what I know best

Play the role of introduction and motivation for the third

Today's menu: Type checking a GC

Motivation: why bother?

Type-safe memory management

Progressive refinement towards a first GC

Preserving sharing with forwarding pointers

Extending the GC to be generational

Type checking

By far the most popular formal method

Fully integrated in the toolchain: “no extra work”

Good error reporting, directly pointing to offending line

Early detection $\Rightarrow$ cheap fix

Guaranteed 100% pure formal: checks the *actual* code

Example, to eliminate NULL dereferences:

```
data Maybe a = Nothing | Just a
```

- `String`: type of strings; can’t be “NULL”!
- `Maybe String`: this one can be “NULL”!

Guaranteed 100% pure formal?

Is the machine code really faithful to your source code?

What about the runtime functions?

Does the garbage collector (GC) affect the semantics?

The compiler and runtime are complex programs, hard to trust

We would like to take them out of the Trusted Computing Base (TCB)

Type-checking the garbage Collector (GC)

We would like to write the GC as a normal type-checked library

- Not in the TCB any more
- Each applications can choose its favorite
- Type checking will hopefully help us catch bugs

Impossible without adjustments:

- Adjustments on the side of the type system
- Adjustments on the side of the GC

Why only check the types

We could just prove full correctness, with Hoare logic:

- No need to adjust the type system, nor the GC

But:

- Checking types is usually easier
- We do not need full correctness
- Type system adjustment is hopefully generally useful
- Only need *type preservation*

So we want to prove preservation of types via type checking

What kind of GC to use

Stop&copy:

- Allocate a new *To* space
- Starting from the *roots*, copy all reachable objects into *To*
- Free the old *From* space (or swap *From* and *To*)

Mark&sweep:

- Starting from the *roots* mark all reachable objects
- Then sweep the heap and free all unmarked objects

Type safe memory management

Main goal: checking type-safety of *free*

Two free-able areas cannot have the same type!

Free-able *objects* $\Rightarrow$ every reference's type is unique

- E.g. *alias types*, where refs have type $\mathtt{ptr}\,\ell$

Free-able *regions* $\Rightarrow$ reference's type annotated with the region

- Region type systems have types like $\tau_1 \times^{\rho} \tau_2$

Function types can look like $[\Psi_1]\tau_1 \rightarrow [\Psi_2]\tau_2$

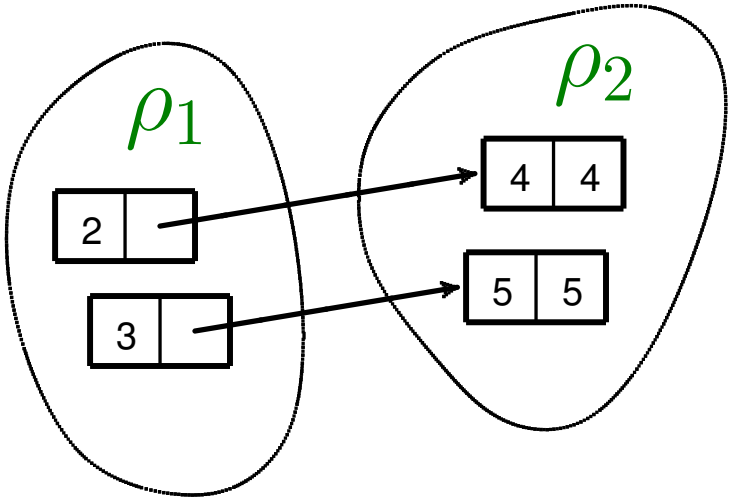
After *free*, ℓ or ρ is removed from Ψ

Regions

Types track regions rather than individual objects

$$\sigma ::= \ldots \mid \sigma_1 \times^{\rho} \sigma_2 \mid \forall \rho.\sigma$$

$$e ::= \ldots \mid (e_1, e_2)^{\rho} \mid \Lambda\rho.e \mid e[\rho] \mid \texttt{let region}\ \rho\ \texttt{in}\ e \mid \texttt{free}\ \rho\ \texttt{in}\ e$$

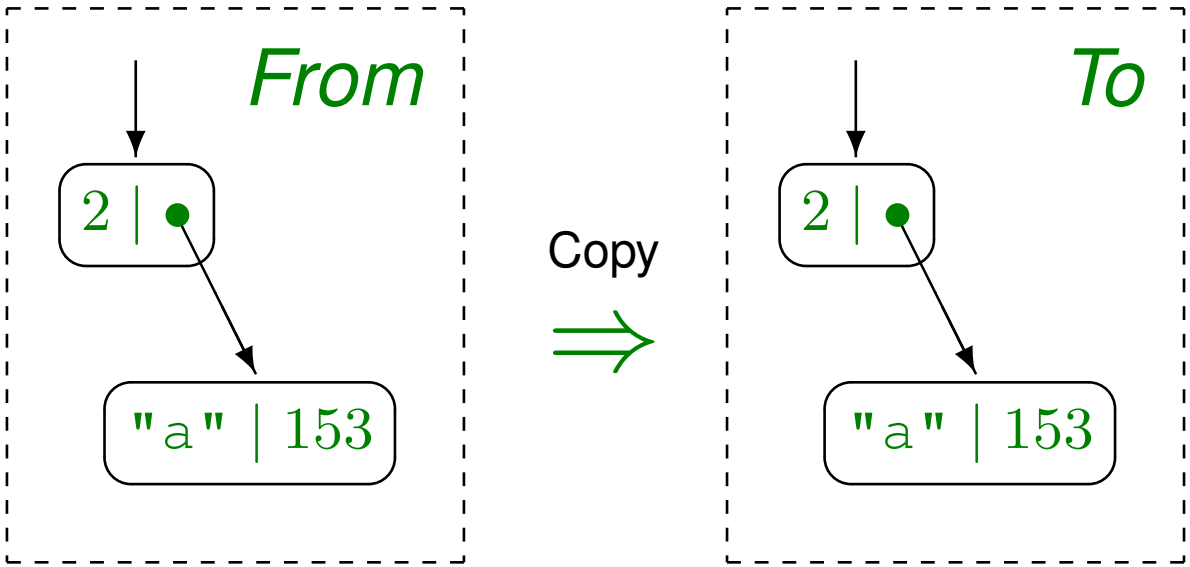


$\texttt{int} \times^{\rho_1} (\texttt{int} \times^{\rho_2} \texttt{int})$

Continuation Passing Style, unlike Tofte's.

Stop-and-Copy

Copy the heap from *From* to *To*

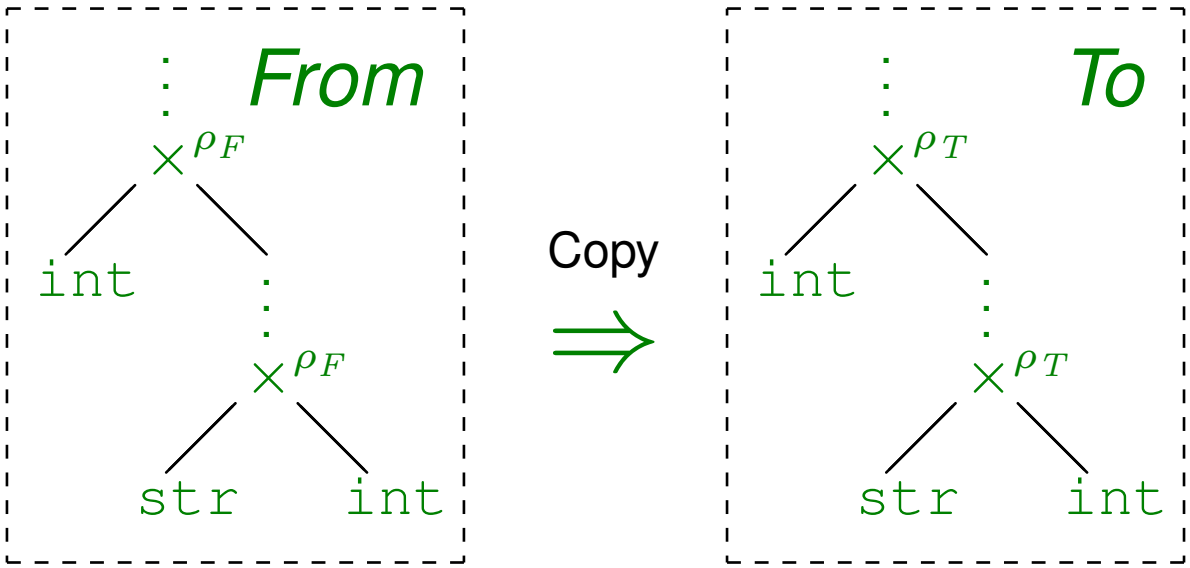


Free *From* and go back to work

No correctness or completeness, just type safety

Type-preserving GC

Use region annotations to free *From* safely



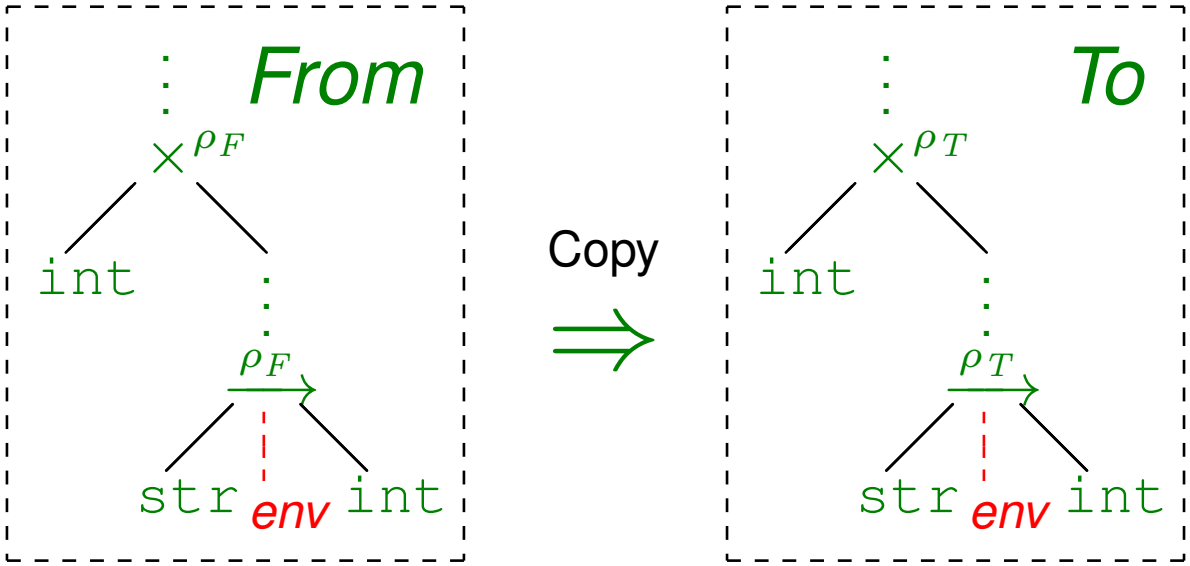
After *copy*, types of *roots* do not refer to ρ_F any more

$\Rightarrow$ *From* can be freed safely

Find the right *copy* function!

Closures

What about hidden free variables ?



Need to look under the hood

Same would apply to other abstract data, e.g. *objects*

A rough draft of copy

$$
\begin{array}{l}
\texttt{fun}\ copy[\rho_F, \rho_T][t](x : t) : t\{\rho_T/\rho_F\} = \\
\quad \texttt{typecase}\ t\ \texttt{of} \\
\quad\quad \texttt{Int} \Rightarrow x \\
\quad\quad t_1 \times t_2 \Rightarrow \texttt{let}\ x1 = copy[\rho_F, \rho_T][t_1]x.1\ \texttt{in} \\
\quad\quad\quad\quad \texttt{let}\ x2 = copy[\rho_F, \rho_T][t_2]x.2\ \texttt{in} \\
\quad\quad\quad\quad (x1, x2)^{\rho_T} \\
\quad\quad t_1 \to t_2 \Rightarrow x \\
\quad\quad \ldots \Rightarrow \ldots
\end{array}
$$

A rough draft of copy

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : t) : t\{\rho_T/\rho_F\} =$$
$$\texttt{typecase}\ t\ \texttt{of}$$
$$\texttt{Int} \Rightarrow x$$
$$t_1 \times t_2 \Rightarrow \texttt{let}\ x1 = copy[\rho_F, \rho_T][t_1]x.1\ \texttt{in}$$
$$\texttt{let}\ x2 = copy[\rho_F, \rho_T][t_2]x.2\ \texttt{in}$$
$$(x1, x2)^{\rho_T}$$
$$t_1 \rightarrow t_2 \Rightarrow x$$
$$\exists t.\tau \Rightarrow \ldots$$

Preserve types

A rough draft of copy

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : t) : t\{\rho_T/\rho_F\} =$$

$$\texttt{typecase}\ t\ \texttt{of}$$

$$\texttt{Int} \Rightarrow x$$

Preserve types

$$t_1 \times t_2 \Rightarrow \texttt{let}\ x1 = copy[\rho_F, \rho_T][t_1]x.1\ \texttt{in}$$

$$\texttt{let}\ x2 = copy[\rho_F, \rho_T][t_2]x.2\ \texttt{in}$$

$$(x1, x2)^{\rho_T}$$

$$t_1 \rightarrow t_2 \Rightarrow x$$ ← Closures are encoded as existentials

$$\exists t.\tau \Rightarrow \ldots$$

A rough draft of copy

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : t) : t\{\rho_T/\rho_F\} =$$

$$\texttt{typecase}\ t\ \texttt{of}$$

$$\texttt{Int} \Rightarrow x$$

$$t_1 \times t_2 \Rightarrow \texttt{let}\ x1 = copy[\rho_F, \rho_T][t_1]x.1\ \texttt{in}$$

$$\texttt{let}\ x2 = copy[\rho_F, \rho_T][t_2]x.2\ \texttt{in}$$

$$(x1, x2)^{\rho_T}$$

$$t_1 \rightarrow t_2 \Rightarrow x$$

$$\exists t.\tau \Rightarrow \ldots$$

Type manipulation operation

Preserve types

← Closures are encoded as existentials

A rough draft of copy

Can we really copy *any* object?

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : t) : t\{\rho_T/\rho_F\} =$$

Type manipulation operation

$$\texttt{typecase}\ t\ \texttt{of}$$

$$\texttt{Int} \Rightarrow x$$

Preserve types

$$t_1 \times t_2 \Rightarrow \texttt{let}\ x1 = copy[\rho_F, \rho_T][t_1]x.1\ \texttt{in}$$
$$\texttt{let}\ x2 = copy[\rho_F, \rho_T][t_2]x.2\ \texttt{in}$$
$$(x1, x2)^{\rho_T}$$

$$t_1 \rightarrow t_2 \Rightarrow x$$ ← Closures are encoded as existentials

$$\exists t.\tau \Rightarrow \ldots$$

Separating types and tags

$$\tau ::= \texttt{Int} \mid \tau_1 \times \tau_2 \mid \tau_1 \to 0 \mid \exists t.\tau \mid t$$

Tags for the oblivious mutator

Separating types and tags

$$\tau ::= \mathtt{Int} \mid \tau_1 \times \tau_2 \mid \tau_1 \to 0 \mid \exists t.\tau \mid t$$

Tags for the oblivious mutator

$$\sigma ::= \mathtt{int} \mid \sigma_1 \times^{\rho} \sigma_2 \mid \sigma_1 \to 0 \mid \exists^{\rho} t.\sigma \mid \forall t.\sigma \mid \forall \rho.\sigma \mid \mathtt{M}_{\rho}(\tau)$$

More detailed underlying types for the GC

Separating types and tags

$$\tau ::= \mathtt{Int} \mid \tau_1 \times \tau_2 \mid \tau_1 \to 0 \mid \exists t.\tau \mid t$$

Tags for the oblivious mutator

$$\sigma ::= \mathtt{int} \mid \sigma_1 \times^{\rho} \sigma_2 \mid \sigma_1 \to 0 \mid \exists^{\rho} t.\sigma \mid \forall t.\sigma \mid \forall \rho.\sigma \mid \mathtt{M}_{\rho}(\tau)$$

More detailed underlying types for the GC

τ tags correspond to the source-level view

σ types express the underlying lower-level reality

The mysterious M

$\mathtt{M}$ is an *iterator* operation on types

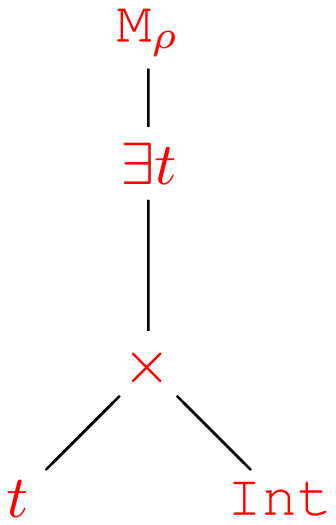
$$
\begin{aligned}
\mathtt{M}_\rho(\mathtt{Int}) &\leadsto \mathtt{int} \\
\mathtt{M}_\rho(\tau_1 \times \tau_2) &\leadsto \mathtt{M}_\rho(\tau_1) \times^\rho \mathtt{M}_\rho(\tau_2) \\
\mathtt{M}_\rho(\exists t.\tau) &\leadsto \exists^\rho t.\mathtt{M}_\rho(\tau) \\
\mathtt{M}_\rho(\tau \to 0) &\leadsto \forall \rho'.\mathtt{M}_{\rho'}(\tau) \to 0
\end{aligned}
$$

The mysterious M

$\mathtt{M}$ is an *iterator* operation on types

$$
\begin{aligned}
\mathtt{M}_\rho(\mathtt{Int}) &\leadsto \mathtt{int} \\
\mathtt{M}_\rho(\tau_1 \times \tau_2) &\leadsto \mathtt{M}_\rho(\tau_1) \times^\rho \mathtt{M}_\rho(\tau_2) \\
\mathtt{M}_\rho(\exists t.\tau) &\leadsto \exists^\rho t.\mathtt{M}_\rho(\tau) \\
\mathtt{M}_\rho(\tau \to 0) &\leadsto \forall \rho'.\mathtt{M}_{\rho'}(\tau) \to 0
\end{aligned}
$$

$\mathtt{M}_\rho(\exists t.t \times \mathtt{Int})$

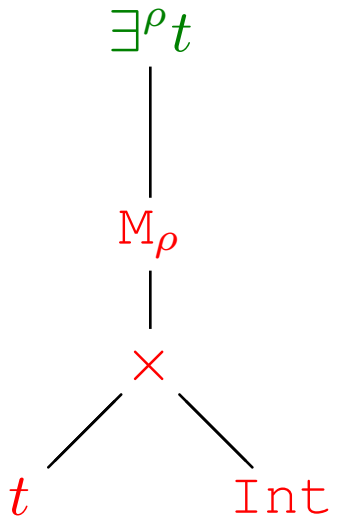


The mysterious M

$\mathtt{M}$ is an *iterator* operation on types

$$
\begin{aligned}
\mathtt{M}_\rho(\mathtt{Int}) &\leadsto \mathtt{int} \\
\mathtt{M}_\rho(\tau_1 \times \tau_2) &\leadsto \mathtt{M}_\rho(\tau_1) \times^\rho \mathtt{M}_\rho(\tau_2) \\
\mathtt{M}_\rho(\exists t.\tau) &\leadsto \exists^\rho t.\mathtt{M}_\rho(\tau) \\
\mathtt{M}_\rho(\tau \to 0) &\leadsto \forall \rho'.\mathtt{M}_{\rho'}(\tau) \to 0
\end{aligned}
$$

$\exists^\rho t.\mathtt{M}_\rho(t \times \mathtt{Int})$

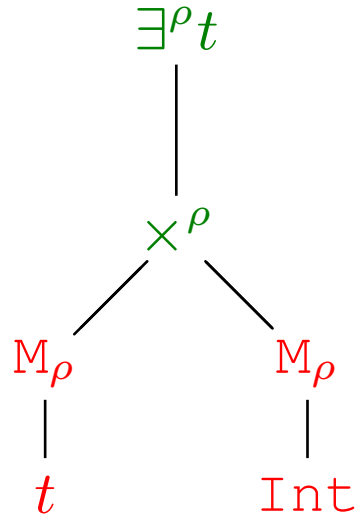


The mysterious M

$\mathtt{M}$ is an *iterator* operation on types

$$
\begin{aligned}
\mathtt{M}_\rho(\mathtt{Int}) &\leadsto \mathtt{int} \\
\mathtt{M}_\rho(\tau_1 \times \tau_2) &\leadsto \mathtt{M}_\rho(\tau_1) \times^\rho \mathtt{M}_\rho(\tau_2) \\
\mathtt{M}_\rho(\exists t.\tau) &\leadsto \exists^\rho t.\mathtt{M}_\rho(\tau) \\
\mathtt{M}_\rho(\tau \to 0) &\leadsto \forall \rho'.\mathtt{M}_{\rho'}(\tau) \to 0
\end{aligned}
$$

$$\exists^\rho t.\mathtt{M}_\rho(t) \times^\rho \mathtt{M}_\rho(\mathtt{Int})$$

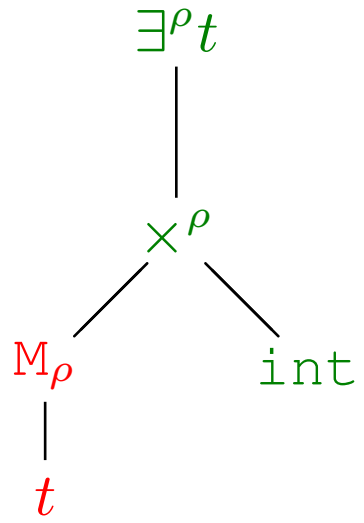


The mysterious M

$\mathtt{M}$ is an *iterator* operation on types

$$
\begin{aligned}
\mathtt{M}_\rho(\mathtt{Int}) &\leadsto \mathtt{int} \\
\mathtt{M}_\rho(\tau_1 \times \tau_2) &\leadsto \mathtt{M}_\rho(\tau_1) \times^\rho \mathtt{M}_\rho(\tau_2) \\
\mathtt{M}_\rho(\exists t.\tau) &\leadsto \exists^\rho t.\mathtt{M}_\rho(\tau) \\
\mathtt{M}_\rho(\tau \to 0) &\leadsto \forall \rho'.\mathtt{M}_{\rho'}(\tau) \to 0
\end{aligned}
$$

$$\exists^\rho t.\mathtt{M}_\rho(t) \times^\rho \mathtt{int}$$



How it works

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : t) : t\{\rho_T/\rho_F\}$$

$$t \quad \xrightarrow{\mathcal{R}} \quad t\{\rho_T/\rho_F\}$$

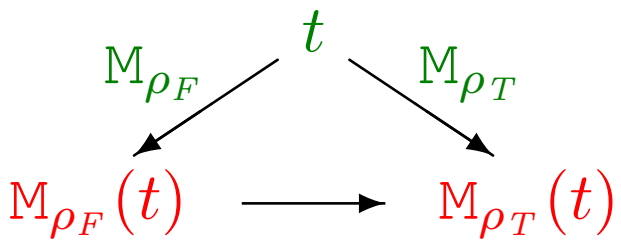
How it works

$$\cancel{\texttt{fun}\ copy[\rho_F, \rho_T][t](x : t) : t\{\rho_T/\rho_F\}}$$

$$\cancel{t \xrightarrow{\mathcal{R}} t\{\rho_T/\rho_F\}}$$

Replace a potentially complex relation $\mathcal{R}$ with a simple function

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : \mathtt{M}_{\rho_F}(t)) : \mathtt{M}_{\rho_T}(t)$$



Now $\mathtt{M}_\rho$ can impose constraints on the shape of the heap

Preserving sharing

Current *copy* gleefully turns a DAG into a tree!

Add a “forwarded” bit: when set the object is replaced by a pointer

After copying an object, overwrite it with a *forwarding pointer* to the copy

$\Rightarrow$ objects need to have types like $\sigma + (\texttt{fwd}\ \sigma)$

A “sum type” means the value is either one of the two alternatives

And it comes with a tag (a single bit) to say which one it is

Forwarding pointers

$$
\begin{aligned}
\sigma ::= \; & \texttt{int} \\
& \mid \sigma_1 \times^{\rho} \sigma_2 \\
& \mid \sigma_1 \to 0 \mid \forall t.\sigma \\
& \mid \exists^{\rho} t.\sigma \mid \forall \rho.\sigma \\
& \mid \mathtt{M}_{\rho}(\tau)
\end{aligned}
$$

Forwarding pointers

$$\begin{aligned}\sigma ::= & \ \mathtt{int} \\ & \mid \sigma_1 \times^{\rho} \sigma_2 \\ & \mid \sigma_1 \to 0 \mid \forall t.\sigma \\ & \mid \exists^{\rho} t.\sigma \mid \forall \rho.\sigma \\ & \mid \mathtt{M}_{\rho}(\tau) \\ & \mid \mathtt{left}\ \sigma_1 + \mathtt{right}\ \sigma_2\end{aligned}$$

Need GC types of the shape: $\sigma + (\mathtt{fwd}\ \sigma)$

Forwarding pointers

$$
\begin{aligned}
\sigma ::= \ & \texttt{int} \\
& \mid \sigma_1 \times^{\rho} \sigma_2 \\
& \mid \sigma_1 \to 0 \mid \forall t.\sigma \\
& \mid \exists^{\rho} t.\sigma \mid \forall \rho.\sigma \\
& \mid \mathtt{M}_{\rho}(\tau) \\
& \mid \texttt{left}\ \sigma_1 + \texttt{right}\ \sigma_2 \\
& \mid \texttt{left}\ \sigma
\end{aligned}
$$

Need GC types of the shape: $\sigma + (\texttt{fwd}\ \sigma)$

Hide the `right` part

Forwarding pointers

$$
\begin{aligned}
\sigma ::= & \ \mathtt{int} \\
& \mid \sigma_1 \times^{\rho} \sigma_2 \\
& \mid \sigma_1 \to 0 \mid \forall t.\sigma \\
& \mid \exists^{\rho} t.\sigma \mid \forall \rho.\sigma \\
& \mid \mathtt{M}_{\rho}(\tau) \\
& \mid \mathtt{left}\ \sigma_1 + \mathtt{right}\ \sigma_2 \\
& \mid \mathtt{left}\ \sigma \\
& \mid \mathtt{C}_{\rho,\rho'}(\tau)
\end{aligned}
$$

Need GC types of the shape: $\sigma + (\mathtt{fwd}\ \sigma)$

Hide the `right` part

$\mathtt{M}_{\rho}(\tau)$ while mutating and $\mathtt{C}_{\rho,\rho'}(\tau)$ while collecting

Forwarding pointers

$$
\begin{aligned}
\sigma ::= \ & \texttt{int} \\
& \mid \sigma_1 \times^{\rho} \sigma_2 \\
& \mid \sigma_1 \to 0 \mid \forall t.\sigma \\
& \mid \exists^{\rho} t.\sigma \mid \forall \rho.\sigma \\
& \mid \texttt{M}_{\rho}(\tau) \\
& \mid \texttt{left}\ \sigma_1 + \texttt{right}\ \sigma_2 \\
& \mid \texttt{left}\ \sigma \\
& \mid \texttt{C}_{\rho,\rho'}(\tau)
\end{aligned}
$$

Need GC types of the shape: $\sigma + (\texttt{fwd}\ \sigma)$

Hide the `right` part

$\texttt{M}_{\rho}(\tau)$ while mutating and $\texttt{C}_{\rho,\rho'}(\tau)$ while collecting

`M` needs to be redefined

Forwarding pointers

$$
\begin{aligned}
\sigma ::= \ & \texttt{int} \\
& \mid \sigma_1 \times^{\rho} \sigma_2 \\
& \mid \sigma_1 \to 0 \mid \forall t.\sigma \\
& \mid \exists^{\rho} t.\sigma \mid \forall \rho.\sigma \\
& \mid \mathtt{M}_{\rho}(\tau) \\
& \mid \texttt{left}\ \sigma_1 + \texttt{right}\ \sigma_2 \\
& \mid \texttt{left}\ \sigma \\
& \mid \mathtt{C}_{\rho,\rho'}(\tau)
\end{aligned}
$$

Need GC types of the shape: $\sigma + (\texttt{fwd}\ \sigma)$

Hide the `right` part

$\mathtt{M}_{\rho}(\tau)$ while mutating and $\mathtt{C}_{\rho,\rho'}(\tau)$ while collecting

`M` needs to be redefined

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : \mathtt{C}_{\rho_F,\rho_T}(t)) :$$

Forwarding pointers

$$
\begin{aligned}
\sigma ::= \ & \texttt{int} \\
& \mid \sigma_1 \times^{\rho} \sigma_2 \\
& \mid \sigma_1 \to 0 \mid \forall t.\sigma \\
& \mid \exists^{\rho} t.\sigma \mid \forall \rho.\sigma \\
& \mid \texttt{M}_{\rho}(\tau) \\
& \mid \texttt{left}\ \sigma_1 + \texttt{right}\ \sigma_2 \\
& \mid \texttt{left}\ \sigma \\
& \mid \texttt{C}_{\rho,\rho'}(\tau)
\end{aligned}
$$

Need GC types of the shape: $\sigma + (\texttt{fwd}\ \sigma)$

Hide the `right` part

$\texttt{M}_{\rho}(\tau)$ while mutating and $\texttt{C}_{\rho,\rho'}(\tau)$ while collecting

`M` needs to be redefined

$$\texttt{fun}\ copy[\rho_F, \rho_T][t](x : \texttt{C}_{\rho_F,\rho_T}(t)) : \texttt{M}_{\rho_T}(t)$$

Updated types for forwarding

$$\mathtt{M}_\rho(\mathtt{Int}) \leadsto \mathtt{int}$$

$$\mathtt{M}_\rho(\tau_1 \times \tau_2) \leadsto \mathtt{left}\,(\mathtt{M}_\rho(\tau_1) \times^\rho \mathtt{M}_\rho(\tau_2))$$

$$\mathtt{M}_\rho(\exists t.\tau) \leadsto \mathtt{left}\,(\exists^\rho t.\mathtt{M}_\rho(\tau))$$

$$\mathtt{M}_\rho(\tau \to 0) \leadsto \forall \rho'.\mathtt{M}_{\rho'}(\tau) \to 0$$

$$\mathtt{C}_{\rho,\rho'}(\mathtt{Int}) \leadsto \mathtt{M}_\rho(\mathtt{Int})$$

$$\mathtt{C}_{\rho,\rho'}(\tau_1 \times \tau_2) \leadsto \mathtt{left}\,(\mathtt{C}_{\rho,\rho'}(\tau_1) \times^\rho \mathtt{C}_{\rho,\rho'}(\tau_2)) + \mathtt{right}\,(\mathtt{M}_{\rho'}(\tau_1 \times \tau_2))$$

$$\mathtt{C}_{\rho,\rho'}(\exists t.\tau) \leadsto \mathtt{left}\,(\exists^\rho t.\mathtt{C}_{\rho,\rho'}(\tau)) + \mathtt{right}\,(\mathtt{M}_{\rho'}(\exists t.\tau))$$

$$\mathtt{C}_{\rho,\rho'}(\tau \to 0) \leadsto \mathtt{M}_\rho(\tau \to 0)$$

Changing the view of the heap

Mutator calls GC with a heap of type $\mathtt{M}_{\rho}(\tau)$

But $copy$ needs a heap of type $\mathtt{C}_{\rho,\rho'}(\tau)$

$\Rightarrow$ need to cast between the two views:

Changing the view of the heap

Mutator calls GC with a heap of type $\mathtt{M}_{\rho}(\tau)$

But $copy$ needs a heap of type $\mathtt{C}_{\rho,\rho'}(\tau)$

$\Rightarrow$ need to cast between the two views:

$$\begin{aligned}
&\mathtt{fun}\ GC[\rho_F][t](x{:}\mathtt{M}_{\rho_F}(t), \ldots) = \\
&\quad \mathtt{let}\ \rho_T = \mathtt{newregion}\ \mathtt{in} \\
&\quad \mathtt{let}\ w : \mathtt{C}_{\rho_F,\rho_T}(t) = \mathtt{widen}\ x\ \mathtt{in} \\
&\quad \mathtt{let}\ y = copy[\rho_F, \rho_T][t](w)\ \mathtt{in}\ \ldots
\end{aligned}$$

The widen *operation*

Widen takes a value x and *casts* it to a new type

Safe because it only turns $\texttt{left}\ \sigma_1$ into $\texttt{left}\ \sigma_1 + \texttt{right}\ \sigma_2$

To be safe, has to apply to the whole heap

- Otherwise, risk 2 references to the same object with different types!

Very ugly, very ad-hoc (and with a bug in the proof)

Found better since

Generational GC

Split heap into 2 zones: *nursery* and *old space*

Allocate new objects in the nursery

Restrict pointers from *old space* to the *nursery*

Can GC the *nursery* without consulting the *old space*

- Start from the *roots*
- For references to the *nursery*: copy to the *old space*
- For reference to the *old space*: leave them unchanged
- Free/empty the *nursery*

Very effective at collecting garbage in functional languages

Generations

$$
\begin{aligned}
\sigma ::= & \ \mathtt{int} \\
& \mid \sigma_1 \times^{\rho} \sigma_2 \\
& \mid \sigma_1 \to 0 \mid \exists^{\rho} t.\sigma \\
& \mid \forall t.\sigma \mid \forall \rho.\sigma \\
& \mid \mathtt{M}_{\rho}(\tau)
\end{aligned}
$$

Generations

$$\begin{aligned}\sigma ::= & \ \mathtt{int} \\ & \mid \sigma_1 \times^{\rho} \sigma_2 \\ & \mid \sigma_1 \to 0 \mid \exists^{\rho} t.\sigma \\ & \mid \forall t.\sigma \mid \forall \rho.\sigma \\ & \mid \mathtt{M}_{\rho_Y,\rho_O}(\tau)\end{aligned}$$

$\mathtt{M}_{\rho_Y,\rho_O}(\tau)$ confines objects

inside ρ_O or ρ_Y

Generations

$$\begin{aligned}\sigma ::= & \ \mathtt{int} \\ & \mid \sigma_1 \times^{\rho} \sigma_2 \\ & \mid \sigma_1 \to 0 \mid \exists^{\rho} t.\sigma \\ & \mid \forall t.\sigma \mid \forall \rho.\sigma \\ & \mid \mathtt{M}_{\rho_Y,\rho_O}(\tau) \\ & \mid \exists \rho \in \Delta.\sigma\end{aligned}$$

$\mathtt{M}_{\rho_Y,\rho_O}(\tau)$ confines objects

inside ρ_O or ρ_Y

Use $\exists \rho \in \Delta.\sigma$ to hide

the region from the mutator

Generations

$$\begin{aligned}\sigma ::= & \ \texttt{int} \\ & \mid \sigma_1 \times^{\rho} \sigma_2 \\ & \mid \sigma_1 \to 0 \mid \exists^{\rho} t.\sigma \\ & \mid \forall t.\sigma \mid \forall \rho.\sigma \\ & \mid \mathtt{M}_{\rho_Y,\rho_O}(\tau) \\ & \mid \exists \rho \in \Delta.\sigma\end{aligned}$$

$\mathtt{M}_{\rho_Y,\rho_O}(\tau)$ confines objects

inside ρ_O or ρ_Y

Use $\exists \rho \in \Delta.\sigma$ to hide

the region from the mutator

Copying the young region to the old region

$$\texttt{fun}\ copy[\rho_Y, \rho_O][t](x : \mathtt{M}_{\rho_Y,\rho_O}(t)) : \mathtt{M}_{\rho_O,\rho_O}(t)$$

Enforcing the age barrier

$\mathtt{M}$ must prevent references from the old generation to the young:

$$\mathtt{M}_{\rho_Y,\rho_O}(\tau_1) \times^{\rho_Y} \mathtt{M}_{\rho_Y,\rho_O}(\tau_2)$$

$$\mathtt{M}_{\rho_O,\rho_O}(\tau_1) \times^{\rho_O} \mathtt{M}_{\rho_O,\rho_O}(\tau_2)$$

Enforcing the age barrier

$\mathtt{M}$ must prevent references from the old generation to the young:

$$\mathtt{M}_{\rho_Y,\rho_O}(\tau_1) \times^{\rho_Y} \mathtt{M}_{\rho_Y,\rho_O}(\tau_2)$$

$$\mathtt{M}_{\rho_O,\rho_O}(\tau_1) \times^{\rho_O} \mathtt{M}_{\rho_O,\rho_O}(\tau_2)$$

The two cases are unified using an existential package:

$$\begin{aligned}
\mathtt{M}_{\rho_Y,\rho_O}(\mathtt{Int}) &\rightsquigarrow \mathtt{int} \\
\mathtt{M}_{\rho_Y,\rho_O}(\tau_1 \times \tau_2) &\rightsquigarrow \exists \rho \in \{\rho_Y, \rho_O\}.\mathtt{M}_{\rho,\rho_O}(\tau_1) \times^{\rho} \mathtt{M}_{\rho,\rho_O}(\tau_2) \\
\mathtt{M}_{\rho_Y,\rho_O}(\exists t.\tau) &\rightsquigarrow \exists \rho \in \{\rho_Y, \rho_O\}.\exists^{\rho} t.\mathtt{M}_{\rho,\rho_O}(\tau) \\
\mathtt{M}_{\rho_Y,\rho_O}(\tau \to 0) &\rightsquigarrow \forall[\rho, \rho'].\mathtt{M}_{\rho,\rho'}(\tau) \to 0
\end{aligned}$$

Conclusion

Important steps towards a realistic type safe GC

Clean and generic framework to express mutator/GC interface

Soundness proofs, including for forwarding pointers

Simple generational type safe GC

Still naive w.r.t cycles, side-effects, tag representation, ...