A Pointer through Layers

Junyoung Jang
junyoung.jang@mail.mcgill.ca
McGill University

Joint Work with Frédéric Lahaie-Bertrand, Stefan Monnier, Brigitte Pientka

Modern Software Systems

- Rarely consist of a single component

Modern Software Systems

- Rarely consist of a single component
- Each component may use different languages

Modern Software Systems

- Rarely consist of a single component
- Each component may use different languages
- Communication between components, and therefore between languages, is unavoidable!

Modern Software Systems

- Rarely consist of a single component
- Each component may use different languages
- Communication between components, and therefore between languages, is unavoidable!

$\Longrightarrow$ (Programming Language) Interoperability

Two Languages for This Talk

Two Languages for This Talk

$\mathcal{L}_M$

Procedural

Unmanaged
(Manually managed)

Two Languages for This Talk

$\mathcal{L}_M$

Procedural

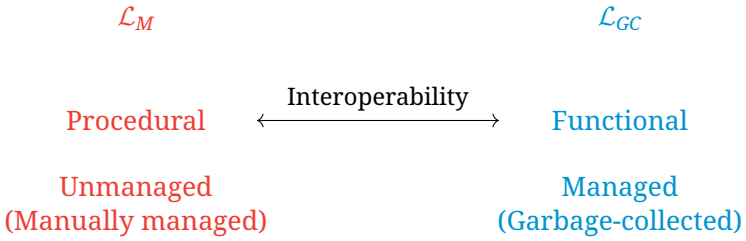
Unmanaged
(Manually managed)

$\mathcal{L}_{GC}$

Functional

Managed
(Garbage-collected)

Two Languages for This Talk



The Problem of Calling a Foreign Function

Numerical computation functions in $\mathcal{L}_M$

$$\texttt{vecAdd}_M \ : \ \mathit{Vec}_M \ \rightarrow \ \mathit{Vec}_M \ \rightarrow \ \mathit{Vec}_M$$
$$\texttt{vecDot}_M \ : \ \mathit{Vec}_M \ \rightarrow \ \mathit{Vec}_M \ \rightarrow \ \mathit{Int}_M$$

The Problem of Calling a Foreign Function

Numerical computation functions in $\mathcal{L}_M$

$$\texttt{vecAdd}_M : Vec_M \to Vec_M \to Vec_M$$
$$\texttt{vecDot}_M : Vec_M \to Vec_M \to Int_M$$

How can we call them from $\mathcal{L}_{GC}$?

Previous Work[1]

Functions converted via built-in conversions

[1] [Patterson et al.2017, Scherer et al.2018]

Previous Work[1]

Functions converted via built-in conversions

e.g., $\mathcal{L}_{GC}$ can use $\texttt{[vecAdd]}_{GC} : Vec_{GC} \to Vec_{GC} \to Vec_{GC}$

[1] [Patterson et al.2017, Scherer et al.2018]

Previous Work[1]

Functions converted via built-in conversions

e.g., $\mathcal{L}_{GC}$ can use $[\texttt{vecAdd}]_{GC} : Vec_{GC} \to Vec_{GC} \to Vec_{GC}$

Problem: Programmers *have no control* over conversions.

[1] [Patterson et al.2017, Scherer et al.2018]

A Consequence of No Control: Repeated Conversions

A Consequence of No Control: Repeated Conversions

Suppose we want to compute $(v_1 + v_2 + v_3) \cdot (u_1 + u_2 + u_3)$

A Consequence of No Control: Repeated Conversions

Suppose we want to compute $(v_1 + v_2 + v_3) \cdot (u_1 + u_2 + u_3)$

one might write

```
let ir1 = [vecAdd]GC v1  v2 in
let ir2 = [vecAdd]GC ir1 v3 in
let ir3 = [vecAdd]GC u1  u2 in
let ir4 = [vecAdd]GC ir3 u3 in
[vecDot]GC ir2 ir4
```

A Consequence of No Control: Repeated Conversions

Suppose we want to compute $(v_1 + v_2 + v_3) \cdot (u_1 + u_2 + u_3)$

one might write

$$
\begin{array}{l}
\mathit{let}\ \mathtt{ir}_1 = [\mathtt{vecAdd}]_{GC}\ \mathtt{v}_1\ \mathtt{v}_2\ \mathit{in} \\
\mathit{let}\ \mathtt{ir}_2 = [\mathtt{vecAdd}]_{GC}\ \mathtt{ir}_1\ \mathtt{v}_3\ \mathit{in} \\
\mathit{let}\ \mathtt{ir}_3 = [\mathtt{vecAdd}]_{GC}\ \mathtt{u}_1\ \mathtt{u}_2\ \mathit{in} \\
\mathit{let}\ \mathtt{ir}_4 = [\mathtt{vecAdd}]_{GC}\ \mathtt{ir}_3\ \mathtt{u}_3\ \mathit{in} \\
[\mathtt{vecDot}]_{GC}\ \mathtt{ir}_2\ \mathtt{ir}_4
\end{array}
$$

The $\mathtt{ir}_i$ values are repeatedly converted between Vec_M and Vec_{GC}!

Our Approach: Intermediate Type for Conversion

Our Approach: Intermediate Type for Conversion

d through intermediate *foreign pointer types* (inspired by adjoint logic[DeYoung and Pfenning20

$$\texttt{vecAdd}_{GC} \;:\; \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Vec}_M$$
$$\texttt{vecDot}_{GC} \;:\; \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Int}_M$$

along with *data conversion functions*

Our Approach: Intermediate Type for Conversion

d through intermediate *foreign pointer types* (inspired by adjoint logic[DeYoung and Pfenning20

$$\texttt{vecAdd}_{GC} \;:\; \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Vec}_M$$
$$\texttt{vecDot}_{GC} \;:\; \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{FPtr}\ \mathit{Int}_M$$

along with *data conversion functions*

$$\texttt{convFromVec}_{GC} \;:\; \mathit{Vec}_{GC} \to \mathit{FPtr}\ \mathit{Vec}_M$$
$$\texttt{convToVec}_{GC} \;:\; \mathit{FPtr}\ \mathit{Vec}_M \to \mathit{Vec}_{GC}$$

Only Desired Conversions

Only Desired Conversions

Again, $(v_1 + v_2 + v_3) \cdot (u_1 + u_2 + u_3)$

Only Desired Conversions

Again, $(v_1 + v_2 + v_3) \cdot (u_1 + u_2 + u_3)$

```
let ir1 = vecAdd_GC (convFromVec_GC v1) (convFromVec_GC v2) in
let ir2 = vecAdd_GC ir1                 (convFromVec_GC v3) in
let ir3 = vecAdd_GC (convFromVec_GC u1) (convFromVec_GC u2) in
let ir4 = vecAdd_GC ir3                 (convFromVec_GC u3) in
convToVec_GC (vecDot_GC ir2 ir4)
```

Only Desired Conversions

Again, $(v_1 + v_2 + v_3) \cdot (u_1 + u_2 + u_3)$

```
let ir1 = vecAdd_GC (convFromVec_GC v1) (convFromVec_GC v2) in
let ir2 = vecAdd_GC ir1                 (convFromVec_GC v3) in
let ir3 = vecAdd_GC (convFromVec_GC u1) (convFromVec_GC u2) in
let ir4 = vecAdd_GC ir3                 (convFromVec_GC u3) in
convToVec_GC (vecDot_GC ir2 ir4)
```

No redundant conversions are needed!

Definition of Intermediate Functions

Definition of Intermediate Functions

How do we obtain $\texttt{vecAdd}_{GC}$?

Definition of Intermediate Functions

How do we obtain $\texttt{vecAdd}_{GC}$?

```
vecAdd_GC vp_1 vp_2 =
    useF v_1 = vp_1 in
    useF v_2 = vp_2 in
    addrOfF(alloc(vecAdd_M *v_1 *v_2))
```

Definition of Intermediate Functions

How do we obtain $\texttt{vecAdd}_{GC}$?

```
vecAdd_GC vp1 vp2 =
    useF v1 = vp1 in
    useF v2 = vp2 in
    addrOfF(alloc(vecAdd_M *v1 *v2))
```

Here, *addrOfF* introduces foreign pointers
and *useF* eliminates them

The Problem of Data Conversions

Vec_M is a linked list with explicit pointers in $\mathcal{L}_M$
Vec_{GC} is a list represented as a recursive algebraic data type in $\mathcal{L}_{GC}$

The Problem of Data Conversions

Vec_M is a linked list with explicit pointers in $\mathcal{L}_M$
Vec_{GC} is a list represented as a recursive algebraic data type in $\mathcal{L}_{GC}$

How can we write conversions between them?

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

```
convFromVec_GC : List_GC Int_GC → FPtr (LinkList_M Int_M)
convFromVec_GC Nil_GC          = addrOfF (alloc(Null_M))
```

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

```
convFromVec_GC : List_GC Int_GC → FPtr (LinkList_M Int_M)

convFromVec_GC (Cons_GC h t) = useF h' = convFromInt_GC h in
                               useF t' = convFromVec_GC t in
                               addrOfF
                                 (alloc(alloc({ value_M = *h', next_M = *t' })))
```

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

```
convFromVec_GC : List_GC Int_GC → FPtr (LinkList_M Int_M)

convFromVec_GC (Cons_GC h t) = useF h' = convFromInt_GC h in
                               useF t' = convFromVec_GC t in
                               addrOfF
                                 (alloc(alloc({ value_M = *h', next_M = *t' })))
```

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

```
convFromVec_GC : List_GC Int_GC → FPtr (LinkList_M Int_M)

convFromVec_GC (Cons_GC h t) = useF h' = convFromInt_GC h in
                               useF t' = convFromVec_GC t in
                               addrOfF
                                 (alloc(alloc({ value_M = *h', next_M = *t' })))
```

where `convFromInt_GC : Int_GC → FPtr Int_M`

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

```
convFromVec_GC : List_GC Int_GC → FPtr (LinkList_M Int_M)

convFromVec_GC (Cons_GC h t) = useF h' = convFromInt_GC h in
                               useF t' = convFromVec_GC t in
                               addrOfF
                                 (alloc(alloc({ value_M = *h', next_M = *t' })))
```

where `convFromInt_GC : Int_GC → FPtr Int_M`

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

```
convFromVec_GC : List_GC Int_GC → FPtr (LinkList_M Int_M)

convFromVec_GC (Cons_GC h t) = useF h' = convFromInt_GC h in
                               useF t' = convFromVec_GC t in
                               addrOfF
                                 (alloc(alloc({ value_M = *h', next_M = *t' })))
```

where `convFromInt_GC : Int_GC → FPtr Int_M`

Example: Programmable Data Conversion From $\mathcal{L}_{GC}$ To $\mathcal{L}_{M}$

Conversion from a list in $\mathcal{L}_{GC}$ (without pointers) to a linked list in $\mathcal{L}_{M}$ (with explicit pointers)

```
convFromVec_GC : List_GC Int_GC → FPtr (LinkList_M Int_M)
convFromVec_GC Nil_GC          = addrOfF (alloc(Null_M))
convFromVec_GC (Cons_GC h t) = useF h' = convFromInt_GC h in
                               useF t' = convFromVec_GC t in
                               addrOfF
                                 (alloc(alloc({ value_M = *h', next_M = *t' })))
```

where `convFromInt_GC : Int_GC → FPtr Int_M`

Meta-properties

- Type safety: well-typed programs do not get stuck

Meta-properties

- Type safety: well-typed programs do not get stuck
- Memory safety: use-after-free errors occur only in $\mathcal{L}_M$
  i.e., if all $\mathcal{L}_M$ components are free of use-after-free errors, then so is the entire system.

Conclusion

Foreign pointer types allow us to

Conclusion

Foreign pointer types allow us to

- control conversions at a fine granularity

Conclusion

Foreign pointer types allow us to

- control conversions at a fine granularity
- call foreign functions

Conclusion

Foreign pointer types allow us to

- control conversions at a fine granularity
- call foreign functions
- compose foreign data conversions programmatically