Specifying and Checking Stateful Software Interfaces (Lecture 2)

Manuel Fähndrich maf@microsoft.com
Microsoft Research

2005 Summer School on Reliable Computing
Eugene, Oregon
Lecture 1 recap

- **Goal:** Specify and check stateful interfaces
- **Techniques**
  - Linear type systems
  - Type system based on capabilities (permissions)
- **Modeling**
  - allocation/deallocation
  - type state protocols
  - locking
Lecture 2

- Frame axiom
- Type-states using capabilities
- Vault: W2K driver case study
- Recursive data structures
- Unifying non-linear data structures and linear data
Lambda abstraction

\[ \tau ::= \ldots | \forall[\Delta].(C; \sigma) \rightarrow (\sigma; C) \]

- We can abstract allocation sequence

\[
\text{init} : \forall[\rho].(\{\rho \leftrightarrow \langle s(0), s(0) \rangle\}, \text{pt}(\rho)) \rightarrow (\text{unit}, \{\rho \leftrightarrow t\})
\]

- pre-heap

- post-heap
Recall examples

- Function taking a list argument (but not consuming it!)
  \[ \text{length} : \forall [\rho]. (C_{\text{List}}(\rho), \text{pt}(\rho)) \rightarrow (\text{int}, C_{\text{List}}(\rho)) \]

- Function freeing entire list
  \[ \text{freeAll} : \forall [\rho]. (C_{\text{List}}(\rho), \text{pt}(\rho)) \rightarrow (\text{unit}, \cdot) \]

- Application rule

\[
\begin{align*}
A; C_1 \vdash e_1 : (C_3, \sigma_1) \rightarrow (\sigma_2, C_4); C_2 \\
A; C_2 \vdash e_2 : \sigma_1; C_3 \\
\hline
A, C_1 \vdash e_1 \ e_2 : \sigma_2; C_4
\end{align*}
\]
The frame rule

\[ C_1 = C_r \otimes C_2 \]

\[
\begin{align*}
A; C_2 \vdash e : \sigma; C_3
\end{align*}
\]

\[
\frac{}{A, C_1 \vdash e : \sigma; C_r \otimes C_3}
\]

[frame]

- Example

\[ x : \text{pt}(\rho_x), \ y : \text{pt}(\rho_y) \]

freeAll(y);

int z = length(x);

freeAll(x);

- Modifications?
Specification tasks

- Allocation/Deallocation ⇒
- Memory initialization ⇒
- Locks ⇒
- Events
- Type states ⇒
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom

Let’s look again at type-states.
Type-states with capabilities

- Still one type per type-state

openR : \( \forall [\rho].(\{\rho \mapsto AFile\}, pt(\rho)) \rightarrow (\text{unit}, \{\rho \mapsto RFile\}) \)

openW : \( \forall [\rho].(\{\rho \mapsto AFile\}, pt(\rho)) \rightarrow (\text{unit}, \{\rho \mapsto WFile\}) \)

promote : \( \forall [\rho].(\{\rho \mapsto RFile\}, pt(\rho)) \rightarrow (\text{unit}, \{\rho \mapsto WFile\}) \)

close : \( \forall [\rho].(\{\rho \mapsto RFile \cup WFile\}, pt(\rho)) \rightarrow (\text{unit}, \{\rho \mapsto CFile\}) \)

free : \( \forall [\rho].(\{\rho \mapsto CFile\}, pt(\rho)) \rightarrow (\text{unit}, \cdot) \)
Observation about type states

- A type state is just a type!
- Type = Predicate over values and heap fragments
- A physical block of memory can have different types, thus different states/properties at different times.
Heavy notation?

- Vault programming language
  - Try to make capabilities available to programmers
  - Type-states as family of some base type
    File@A, File@R, File@W, File@C

```c
void openR( tracked(\rho) File file ) [ \rho@A ! R ];
openR : \forall[\rho].({\rho \leftrightarrow AFile}, pt(\rho)) \rightarrow (unit, {\rho \leftrightarrow RFile})

void closeR( tracked(\rho) File file ) [ --\rho@A ];
closeR : \forall[\rho].({\rho \leftrightarrow RFile}, pt(\rho)) \rightarrow (unit, {\rho \leftrightarrow CFile})
```
Case Study: Windows Drivers

- Driver handles requests from the kernel
  - e.g. start, read, write, shutdown, ...
  - driver exports a function for each request type
  - lifetime of request $\neq$ lifetime of function call
- Request is encapsulated in a data structure
  - I/O Request Packet (IRP)
  - Driver handles request by side-effecting IRP
  - IRP ownership and lifetime are important
Request often lives across calls

```
KERNEL

Read(Device,Irp)

IFun

IFun

DRIVER

on interrupt call IFun
IoMarkIrpPending(Irp)

read memory

read memory

IoCompleteRequest(Irp)
```
Drivers form a stack

- Kernel sends IRP to top driver in stack
- Driver may...
  - handle IRP itself
  - pass IRP down
  - pass new IRP(s) down
IoCompleteRequest

IoCompleteRequest indicates the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O Manager.

Parameters

Irp Points to the IRP to be completed.

PriorityBoost Specifies a system-defined constant by which to increment the runtime priority of the original thread that requested the operation. This value is IO_NO_INCREMENT if the original thread requested an operation the driver could complete quickly (so the requesting thread is not compensated for its assumed wait on I/O) or if the IRP is completed with an error. Otherwise, the set of PriorityBoost constants are device-type-specific. See ntddk.h or wdm.h for these constants.

Comments

When a driver has finished all processing for a given IRP, it calls IoCompleteRequest. The I/O Manager checks the IRP to determine whether any higher-level drivers have set up an IoCompletion routine for the IRP. If so, each IoCompletion routine is called, in turn, until every layered driver in the chain has completed the IRP.

When all drivers have completed a given IRP, the I/O Manager returns status to the original requestor of the operation. Note that a higher-level driver that sets up a driver-created IRP must supply an IoCompletion routine to release the IRP it created.

Callers of IoCompleteRequest must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoSetCompletionRoutine

"IoCompleteRequest indicates the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O Manager."
IRP Ownership

IoCompleteRequest

```c
VOID
IoCompleteRequest(
    IN PIRP Irp,
    IN CCHAR PriorityBoost);
```

IoCompleteRequest indicates the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O Manager.

**Parameters**

- **Irp** Points to the IRP to be completed.
- **PriorityBoost** Specifies a system-defined constant by which to increment the runtime priority of the original thread that requested the operation. This value is IO_NO_INCREMENT if the original thread requested an operation the driver could complete quickly (so the requesting thread is not compensated for its assumed wait on I/O) or if the IRP is completed with an error. Otherwise, the set of PriorityBoost constants are device-type-specific. See ntddk.h or wdm.h for these constants.

**Comments**

When a driver has finished all processing for a given IRP, it calls IoCompleteRequest. The I/O Manager checks the IRP to determine whether any higher-level drivers have set up an IoCompletion routine for the IRP. If so, each IoCompletion routine is called, in turn, until every layered driver in the chain has completed the IRP.

When all drivers have completed a given IRP, the I/O Manager returns status to the original requestor of the operation. Note that a higher-level driver that sets up a driver-created IRP must

```c
void IoCompleteRequest( tracked(I) IRP Irp, CHAR Boost) [ -I ];
```
IRP Ownership

IoCallDriver

NTSTATUS

IoCallDriver(
    IN PDEVICE_OBJECT DeviceObject,
    IN OUT PIRP Irp);

IoCallDriver sends an IRP to the next-lower-level driver after the caller has set up the I/O stack location in the IRP for that driver.

Parameters

DeviceObject Points to the next-lower driver's device object, representing the target device for the requested I/O operation.

Irp Points to the IRP.

Return Value

IoCallDriver returns the NTSTATUS value that a lower driver set in the I/O status block for the given request or STATUS_PENDING if the request was queued for additional processing.

Comments

IoCallDriver assigns the DeviceObject input parameter to the device object field of the IRP stack location for the next lower driver.

An IRP passed in a call to IoCallDriver becomes inaccessible to the higher-level driver, unless the higher-level driver has set up its IoCompletion routine for the IRP with IoSetCompletionRoutine. If it does, the IRP input to the driver-supplied IoCompletion routine has its I/O status block set by the lower driver(s) and all lower-level driver(s)' I/O stack locations filled with zeros.

Drivers must not use IoCallDriver to pass power IRPs (IRP_MJ_POWER). Use PoCallDriver instead.

Callers of IoCallDriver must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateIrp, IoBuildAsynchronousFsdRequest, IoBuildDeviceIoControlRequest, IoBuildSynchronousFsdRequest, IoSetCompletionRoutine, PoCallDriver
IoCallDriver sends an IRP to the next-lower-level driver after the caller has set up the I/O stack location in the IRP for that driver.

Parameters

- **DeviceObject**
  Points to the next-lower driver's device object, representing the target device for the requested I/O operation.

- **Irp**
  Points to the IRP.

Return Value

IoCallDriver returns the NTSTATUS value that a lower driver set in the I/O status block for the given request or STATUS_PENDING if the request was queued for additional processing.

Comments

IoCallDriver assigns the DeviceObject input parameter to the device object field of the IRP stack location for the next lower driver.

An IRP passed in a call to IoCallDriver becomes inaccessible to the higher-level driver, unless the higher-level driver has set up its IoCompletion routine for the IRP with IoSetCompletionRoutine. If it does, the IRP input to the driver-supplied IoCompletion routine has its I/O status block set by the lower driver(s) and all lower-level driver(s)' I/O stack locations filled with zeros.

Drivers must not use IoCallDriver to pass power IRPs (IRP_MJ_POWER). Use PoCallDriver instead.

Callers of IoCallDriver must be running at IRQL <= DISPATCH_LEVEL.

See Also

- IoAllocateIrp
- IoBuildAsynchronousFsdRequest
- IoBuildDeviceIoControlRequest
- IoBuildSynchronousFsdRequest
- IoSetCompletionRoutine
- PoCallDriver
Example: Driver request

```c
NTSTATUS Read(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [ -I ] {

    if (GetRequestLength(Irp) == 0) {
        NTSTATUS status = `STATUS_SUCCESS(`TransferBytes(0));
        IoCompleteRequest(Irp, status);
        return status;
    } else
        return IoCallDriver(NextDriver,Irp);
}
```
Example: Driver request

```c
NTSTATUS Read(DEVICES_OBJECT Dev, tracked(I) IRP Irp) {
    if (GetRequestLength(Irp) == 0) {
        NTSTATUS status = STATUS_SUCCESS(TransferBytes(0));
        IoCompleteRequest(Irp, status);
        return status;
    } else
        return IoCallDriver(NextDriver, Irp);
}
```
IRP completion routines

- Getting IRP ownership back
  - driver A hands IRP to B and wants it back after B is done
  - driver A sets “completion routine” on IRP

```c
void IoSetCompletionRoutine(tracked(K) IRP Irp,
                          COMPLETION_ROUTINE<K> Fun) [K];

type COMPLETION_ROUTINE<key K> =
    tracked COMPLETION_RESULT<K>(DEVICE_OBJECT Dev,
                                  tracked(K) IRP Irp) [-K];

tracked variant COMPLETION_RESULT<key K> [``MoreProcessingRequired |
                                           `Finished(NTSTATUS) {K} ];
```
type KEVENT<key R>;

KEVENT<E> KeInitializeEvent<type T> (tracked(E) T Obj) [ E ];

NTSTATUS KeSignalEvent(KEVENT<E> Event) [ -E ];

NTSTATUS KeWaitForEvent(KEVENT<E> Event) [ +E ];
Completion routine example

```c
NTSTATUS PlugPlay(DEVICE_OBJECT Dev, tracked(R) IRP Irp) [-R] {
    KEVENT<R> DoneEvent = KeInitializeEvent(Irp);

    tracked COMPLETION_RESULT<I>
    CompletePnP(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [-I] {
        KeSignalEvent(DoneEvent);
        return `MoreProcessingRequired;
    }

    IoSetCompletionRoutine(Irp, CompletePnP<R>);
    CALL_RESULT<R> result = IoCallDriver(lowerDriver, Irp);
    KeWaitForEvent(DoneEvent);

    ...
}
```
Specification tasks

- Allocation/Deallocation
- Memory initialization
- Locks
- Events
- Type states
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom

- Non-tree data structures?
Non-tree data structures?

- arbitrary finite graphs and
- a form of regular recursive graphs via existential abstraction over pointer names and heap fragments
Recursive data structures

- Consider a linear list
  \[ \text{List}^2 = \text{Nil} \cup \text{Cons of int} \times \text{List}^2 \]
  “Each Cons cell *owns* the rest of the list”

- Using capabilities:
  - Use \( \text{pt}(0) \) for Nil
  - Package a heap fragment with non-zero pointer
  - Abstract over the pointer value

\[ \text{List}^2, \ 9[\rho \mapsto \text{ListH}.\rho\text{i}].\text{pt}(\rho) \]
\[ \text{C}_{\text{List}}.\rho\text{i} = (\rho=0) \cup \{ \rho \mapsto \text{ListH} \} \]
\[ \text{ListH} = 9[\rho \mapsto \text{ListH}.\rho\text{i}].\text{h int, pt}(\rho) \text{i} \]
Linear list unpacking and packing

\( \rho_0 \) 

\[
\begin{array}{c|c|c}
\text{int} & \rho_1 & \text{CList}(\rho_1) \\
\end{array}
\]
Linear list unpacking and packing

\[ \rho_0 \quad \text{int } \rho_1 \quad (\rho_1 = 0) \lor \{ \rho_1 \leftrightarrow \text{ListH} \} \]
Linear list unpacking and packing

\[ \rho_0 \]

\[ \text{int} \rho_1 \{ \rho_1 \leftrightarrow \text{ListH} \} \]
Linear list unpacking and packing

\[ \{ \rho_1 \mapsto \exists [\rho_2 \mid \text{CList}(\rho_2)]. \langle \text{int}, \text{pt}(\rho_2) \rangle \} \]
Linear list unpacking and packing

\[ \rho_0 \]

\[ \text{int} \]

\[ \rho_1 \]

\[ \text{int} \]

\[ \rho_2 \]

\[ \text{CList}(\rho_2) \]
Linear list unpacking and packing

\[ \rho_0 \]

\[ \text{int} \]

\[ \rho_1 \]

\[ \text{int} \quad \rho_2 \quad (\rho_2 = 0) \lor \{\rho_2 \leftrightarrow \text{ListH}\} \]
Linear list unpacking and packing

\[ \rho_0 \]

\[
\begin{array}{c}
\text{int} \\
\rho_1 \\
\text{int} \\
\rho_2 \\
\text{int} \rho_3 \end{array}
\]

\( (\rho_3 = 0) \lor \{\rho_3 \leftrightarrow \text{ListH} \} \)

\[ \ldots \]
Packing and Unpacking

\[
A; C_1 \vdash e : \exists[\Delta \mid C].\sigma; C_2 \quad \text{[unpack]}
\]

\[
A; C_1 \vdash \text{unpack} \ e : \sigma; C \otimes C_2
\]

\[
A; C_1 \vdash e : S(\sigma); S(C) \otimes C_2
\]

\[
S = [\Delta'/\Delta]
\]

\[
A; C_1 \vdash \text{pack} [\Delta \mid C].e : \exists[\Delta \mid C].\sigma; C_2 \quad \text{[pack]}
\]
Summary of Capability Type Systems

- Capabilities are single-threaded in type system (heap is single-threaded in dynamic semantics)
  - Linear treatment of capabilities
- Splitting and joining of heap fragments
- Relaxed single pointer requirement
  - Single heap fragment invariant
- Natural imperative programming style
  - Can use pointers as often as we like
    - as long as we can prove suitable capability is present
  - Explicit treatment of dangling pointers
Programming languages

- Based on capabilities or similar concepts
  - Vault: resource management and type states
  - Fugue: object type states
  - Sing#: resource management and channels
  - Cyclone: safe C replacement with regions
  - Clay: low-level memory management (GC)
  - ATS: low-level memory management
  - ...

PL Characteristics

- Dichotomy between precisely tracked data and non-linear data (exception Clay)
- Surface specification language vs. internal specification language
  - Has to be concise, otherwise it’s a calculus
  - Difficult to find good trade-off between expressiveness and conciseness
- How much is inferred, how much is explicit?
  - Coercions, Instantiations, Proof terms
Arbitrary data structures?

- Arbitrary graphs are difficult to express, but not impossible
  - O’Hearn et.al. have done specifications and hand proofs of complicated graph algorithms
    - graph copying and freeing
  - But automated systems with such expressive power are still under development
    - Clay (Hawblitzel et. al) and ATS (Xi et.al.) come close.
- Different domains require different expressiveness
  - Specifying and checking copying GC
  - Application program dealing with sockets and files
Non-linear data structures

- Mere mortals need way to express data structures with less detailed capability specifications
  - Who owns the observer in the view-observer pattern?
  - Who owns call-back closures on GUI elements?
    - Where is the permission?
    - How is it threaded to place of use?
- Require some way to abstract over individual permissions
- Necessary evil
Specifications

- Allocation/Deallocation ➞
- Memory initialization ➞
- Locks ➞
- Events ➞
- Type states ➞
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom

- Use ≠ Consume ➞
- Non-tree data structures ➞
Regions

- Rather than handling individual capabilities for individual objects, need a mechanism to abstract over the capabilities for a set of objects.

- Well-known abstraction: Regions
  - A region is a named subset of the heap
  - Objects are individually allocated from a region
  - A region is deallocated as a whole
    - Common lifetime for all objects within region
  - $\rho BT$ denotes an object of type $T$ in region $\rho$
Regions

- A region has type pt(\(\rho\)), where \(\{\rho \mapsto \text{Region}\}\)
- An object in a region has type \(\rho B T\)
- Can define specialized type rules

\[
A; C_0 \vdash e_0 : pt(\rho); C_1 \\
A; C_i \vdash e_i : \tau_i; C_{i+1}
\]

\[
C_{n+1} = \{\rho \mapsto \text{Region}\} \otimes -
\]

\[
A; C_0 \vdash \text{alloc}(e_0)\langle e_1..e_n \rangle : \rho \triangleright \langle \tau_1..\tau_k \rangle; C_{n+1}
\]

\[
A; C_1 \vdash e : \rho \triangleright \langle \tau_1..\tau_n \rangle; C_2
\]

\[
C_2 = \{\rho \mapsto \text{Region}\} \otimes C_3
\]

\[
A; C_1 \vdash e.k : \tau_k; C_2
\]
void main() {
    tracked(R) region reg = Region.create();
    R:point pt = new(reg) point {x=4, y=2};
    int y;
    if (pt.x > 0) {
        Region.delete(reg);
        y = 0;
    } else {
        y = pt.x;
        Region.delete(reg);
    }
    }   post condition!
Bug 1: Dangling reference

```java
void main() {
    tracked(R) region reg = Region.create();
    R:point pt = new(reg) point {x=4, y=2};
    int y;
    if (pt.x > 0) {
        Region.delete(reg);
        y = 0;
    } else {
        Region.delete(reg);
        y = pt.x;
    }
}
```
void main() {
    tracked(R) region reg = Region.create();
    R:point pt = new(reg) point {x=4, y=2};
    int y;
    if (pt.x > 0) {
        y = 0;
    } else {
        y = pt.x;
    }
}

{R} bug! leaking key R
Discussion of Regions

- Different objects in region can have the same type
  \[ \text{R:point} \ x; \ \text{R:point} \ y; \ \ldots \]
- Non-region pointers and pointers into regions have distinct types
  \[ \text{pt}(\rho) \text{ with } \{\rho \mapsto T\} \text{ vs. } \rho BT \]
- Decision for what kind of object is used is done at allocation, and fixed throughout
  - Can’t do incremental initialization e.g.
- Component restriction of linear types:
  - can’t have linear components in region types
Motivating example

Dictionary example:

- map keys to resizable arrays
- sharing of cells suggests cells and contents in a region $\rho Brefh \rho Bint[] i$
- But, resize can’t free old array
- ref<int[]>?  

![Diagram showing a hierarchical structure and a pointer to a resizable array.](image-url)
Generalizing the region idea

- Goal: Uniform object model
  - Birth and death as linear objects
  - Switch from linear to non-linear and back
  - Switch from non-linear to linear and back
- Any resource can serve as a region (a lifetime delimiter)
- Call such a resource an adopter $\rho_a$
- For adoptee:
  - use type $\text{pt}(\rho_1)$
  - non-linear predicate (adoption fact): $\{\rho_a \mapsto T_a\} \land \rho_1 : T_1$
    - “given cap $\{\rho_a \mapsto T_a\}$, can deduce $\rho_1$ is a pointer to $T_1$”
    - delegates permissions
Adoption (Freezing)

- Explicit act to introduce adoption fact

\[ \{\rho_0 : \tau_0\} \text{ B } \rho_1 : \tau_1 \text{ from } \{\rho_1 \mapsto \tau_1\} \]

\[ \begin{align*}
A; C_i &\vdash e_i : \text{pt}(\rho_i); C_{i+1} & i = 0, 1 \\
C_2 &= \{\rho_0 \mapsto h_0\} \otimes \{\rho_1 \mapsto h_1\} \otimes C_3 \\
C_4 &= \{\rho_0 \mapsto h_0\} \otimes C_3 \otimes \{\rho_0 \mapsto h_0\} \triangleright \rho_1 : h_1 \end{align*} \]

\[ A; C_0 \vdash \text{adopt } e_1 \text{ by } e_0 : \text{pt}(\rho_1); C_4 \]

- Abbreviation
  - CBT, 9[\rho | C B \rho : T].\text{pt}(\rho)

- Linear components in non-linear objects
  - T arbitrary
  - But, cannot access linear T's via non-linear permission
Adoption graphically

Before:

\[
\begin{align*}
\rho_1 & \rightarrow h_1 \\
\rho_0 & \rightarrow h_0
\end{align*}
\]

Capability

\[
\{\rho_0 \mapsto h_0\} - \{\rho_1 \mapsto h_1\}
\]

After:

\[
\begin{align*}
\rho_1 & \rightarrow h_1 \\
\rho_0 & \rightarrow h_0
\end{align*}
\]

\[
\{\rho_0 \mapsto h_0\} - \{\rho_0 \mapsto h_0\} B_{\rho_1 : h_1}
\]
Data lifetime model (types)

### alloc pt(\(\rho\))

- \(\{\rho \rightarrow h\}\)
- \(\{\rho_a \rightarrow h_a\} B \rho:h\)
- \(\{\rho \rightarrow h\}\)

### adoption

### focus

### unfocus

### unadoption

### free pt(\(\rho\))
Example Adoption

ACellPh<sub>ρ_D</sub>i newCell( pt(ρ_D) Dict d) {
    pt(ρ_c) Cell c = new Cell;
c.data = new int[];
return(adopt c by d);
}

Cell = \langle \exists \rho' \mid \{ \rho' \mapsto \text{int[]} \}\rangle . pt(\rho')

ACellP<sub>ρ_D</sub> = \exists \rho' \mid \{ \rho_D \mapsto \text{Dict}\} \triangleright \rho' : \text{Cell}. pt(\rho')
Adoption is related to \textbf{let!}

- Wadler 90:
  \begin{verbatim}
  let! (x) y = e_1 in e_2
  \end{verbatim}
  linear type of x is non-linear during \(e_1\).

- Problems:
  - Scoped
  - How to enforce escaping of components of x
  - Unsound with mutability:
    Consider \(\text{ref<int[]>} \rightarrow \text{ref<int[]>}\)
Focus

focus $e_1$ in $e_2$

Restore capability for $\rho_1 \{ \rho_1 \rightarrow h_1 \}$

Fact to restore $\rho_A$

$\{ \rho_1 \rightarrow h_1 \} \ ( \{ \rho_a \rightarrow h_a \}$

Stateful Software Interfaces
void resize(ACellPhρ_D c) {
    focus c {
        free c.data;
        c.data = new int[];
    }
}
Unfocus

- $\{\rho_1 \mapsto h_1\} (\{\rho_A \mapsto h_A\})$
- Can be seen as an implication or coercion function
- Explicit implication allows for non-lexical scopes
  - Right to unfocus can be passed up/down to other functions
  - Useful for inferring scopes locally
Unadoption

Before:

free $\rho_a$

After:

$h_a$ is connected to $\rho_1$ through $\rho_a$, and now $h_n$ is disconnected from $\rho_1$. The state $h_a$ is free of $\rho_a$. 
Generalizations

- Adoption facts: \( C \ B \ \rho:\tau \)
- Abstract over capabilities (symbolic cap \( G \))
- Resize function does not need to know details of adopter
- \( 8[\rho, G]. \ ( G - (G B \ \rho:Cell), \ pt(\rho) ) \) ! ( void, G )
- Temporary view of non-adopted pointer as adopted
  - \( \{ \rho \rightarrow h \} \) ! \( \{ \rho \rightarrow h \} - \{ \rho \rightarrow h \} B \rho:h \)
  - can write functions that work over adopted and non-adopted data!
Generalizations (cont)

- Can handle interior pointers
  \( \{ \rho \mapsto h \} \)
  \[ h = h \ T_1, T_2 \ i \]
  Want \( pt(\rho_1) \) to 1st field of type \( T_1 \)
  \( \{ \rho \mapsto h \} B_{\rho_1 : T_1} \)

- Pointers to the stack
Lecture 3

- Permission sharing
- Type states for objects
- Techniques for message based systems
Backups
Locking (3)

- Lingering problems
  - Release wrong lock

\[
\tau ::= \ldots \ j \ \text{Lock} h_\rho, \ \sigma i \ j \ \text{RToken} h_\rho i
\]

acquire : \( \forall [\rho].(\cdot, \text{Lock} \langle \rho, \sigma \rangle) \rightarrow \exists [\rho'](\sigma, \{\rho' \mapsto \text{RToken} \langle \rho \rangle\}) \)
release : \( \forall [\rho, \rho'].(\{\rho' \mapsto \text{RToken} \langle \rho \rangle\}, \text{Lock} \langle \rho, \sigma \rangle, \sigma) \rightarrow (\text{unit}, \cdot) \)

- Code looks as expected:
  - token not passed explicitly

\[
T \ x = \text{acquire}(\text{lock});
\ldots
release(\text{lock}, \ x);
\]
Packing and unpacking of h

\[
A; C_1 \vdash e : \text{pt}(i); C_2 \\
C_2 = \{i \mapsto S(h)\} \otimes S(C) \otimes C_3 \\
S = [\Delta'/\Delta] \\
\frac{A; C_1 \vdash \text{pack}[\Delta \mid C].e : \text{pt}(i); \{i \mapsto \exists[\Delta \mid C].h\} \otimes C_3}{[\text{pack-h}]} \\

A; C_1 \vdash e : \text{pt}(i); C_2 \\
C_2 = \{i \mapsto \exists[\Delta \mid C].h\} \otimes C_3 \\
\frac{A; C_1 \vdash \text{unpack} e : \text{pt}(i); \{i \mapsto h\} \otimes C \otimes C_3}{[\text{unpack-h}]}