# Frame Rules from Answer Types for Code Pointers

Hayo Thielecke

University of Birmingham, UK

# Frame rules from answer types for code pointers

**Background:** typed low-level languages, e.g. assembly
separation logic connectives $*$, $-\!*$ ("magic wand")

**Aim:** frame law in the presence of code pointers
reasoning using the type (or spec),
without having to know the code itself

**Idea:** use answer type polymorphism
in Continuation Passing Style (CPS)
[POPL'03,ESOP'04]
Here: instantiate answer to $A -\!* \alpha$

**Complication:** unsoundness for $A \to \alpha$ as answer type

**Solution:** restrict answer types to $\perp\perp$-closed types
adapted from Pitts, Krivine et. al.

# Answer types background

- Basic questions about continuations: what is the answer type?
- Universal type? Sierpinski? Free type variable?
  Nothing, just negation? Pietism and $\neg \dashv \neg$ self-adjointness.
- Using a single answer type loses precision.
- Quantified answer types in CPS gives us a type-theoretic control flow analysis.
- Where the quantifiers are depends on where the term can jump.
- See *From Control Effects to Typed Continuation Passing* [POPL'03]

# Control effects

Jouvelot and Gifford, 1988: $\texttt{goto}\,\rho$ and $\texttt{comefrom}\,\rho$ effects.
Simplification: we use just $\rho$ for both of them.
Idea: Peirce's law $((\alpha \to \beta) \to \alpha) \to \alpha$ refined with regions.

$$\frac{}{\Gamma \vdash_{\mathsf{c}} \texttt{call/cc} : \forall \rho.((\alpha \xrightarrow{\rho} \beta) \xrightarrow{e} \alpha) \xrightarrow{\rho \cup e} \alpha \; ! \; \emptyset}$$

Further simplification: at most singleton effects; $\rho$ or $\emptyset$.

# Effect systems

$$\frac{\Gamma \vdash_{\mathsf{c}} M : A \xrightarrow{e_1} B \mathbin{!} e_2 \qquad \Gamma \vdash_{\mathsf{c}} N : A \mathbin{!} e_3}{\Gamma \vdash_{\mathsf{c}} MN : B \mathbin{!} e_1 \cup e_2 \cup e_3}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{c}} M : B \mathbin{!} e}{\Gamma \vdash_{\mathsf{c}} \lambda x.M : A \xrightarrow{e} B \mathbin{!} \emptyset}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} V : A \mathbin{!} \emptyset}{\Gamma \vdash_{\mathsf{c}} V : \forall \alpha.A \mathbin{!} \emptyset} \quad \alpha \notin \mathsf{Tyvar}(\Gamma)$$

$$(\mathrm{Masking})\frac{\Gamma \vdash_{\mathsf{c}} M : A \mathbin{!} \rho}{\Gamma \vdash_{\mathsf{c}} \mathtt{newreg}\, M : A \mathbin{!} \emptyset} \quad \rho \notin (\mathsf{Reg}(A) \cup \mathsf{Reg}(\Gamma))$$

# CPS transform on types and effects

For each region $\rho$, we assume a fresh type variable $\alpha_\rho$.

$$
\begin{array}{rcll}
\overline{A \xrightarrow{\emptyset} B} & = & \forall\alpha.(\overline{B} \to \alpha) \to (\overline{A} \to \alpha) & \text{where } \alpha \text{ is fresh} \\
\overline{A \xrightarrow{\rho} B} & = & (\overline{B} \to \alpha_\rho) \to (\overline{A} \to \alpha_\rho) & \\
\overline{\forall\alpha.A} & = & \forall\alpha.\overline{A} & \\
\overline{\forall\rho.A} & = & \forall\alpha_\rho.\overline{A} & \\
\overline{\alpha} & = & \alpha &
\end{array}
$$

If a region $\rho$ is not free in $A$, then $\alpha_\rho$ is not free in $\overline{A}$.

# From regions to answer type polymorphism

If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$ then

$$\overline{\Gamma} \vdash \overline{M} : (\overline{A} \to \alpha_\rho) \to \alpha_\rho$$

If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset$ then

$$\overline{\Gamma} \vdash \overline{M} : \forall \alpha.(\overline{A} \to \alpha) \to \alpha \qquad \text{where } \alpha \text{ is fresh}$$

# Example: no control effects

$$\vdash_{\mathsf{c}} \lambda z.\lambda x.x : B \xrightarrow{\emptyset} (A \xrightarrow{\emptyset} A) \mathrel{!} \emptyset$$

is transformed

$$\begin{aligned} & \overline{\lambda z.\lambda x.x} \\ = \quad & \lambda k_1.k_1(\lambda k_2 z.k_2(\lambda k_3 x.k_3 x)) \end{aligned}$$

$$\begin{aligned} & \overline{B \xrightarrow{\emptyset} (A \xrightarrow{\emptyset} A)} \\ = \quad & \forall \alpha_2.((\forall \alpha_3.(\overline{A} \to \alpha_3) \to (\overline{A} \to \alpha_3)) \to \alpha_2) \to (\overline{B} \to \alpha_2) \end{aligned}$$

Each $\xrightarrow{\emptyset}$ has its own answer type.

# “Upward continuation” example

$$
\begin{array}{rl}
& \overline{\lambda z.\,\texttt{call/cc}(\lambda k.\lambda x.k(\lambda y.x))} \\
= & \lambda k_1.k_1(\lambda k_2 z.k_2(\lambda k_3 x.k_2(\lambda k_4 y.k_4 x)))
\end{array}
$$

$$
\begin{array}{rl}
& \overline{\forall\rho.B \xrightarrow{\rho} (A \xrightarrow{\rho} A)} \\
= & \forall\alpha_\rho.(((\overline{A} \to \alpha_\rho) \to (\overline{A} \to \alpha_\rho)) \to \alpha_\rho) \to (\overline{B} \to \alpha_\rho)
\end{array}
$$

Control effects enforce answer type sharing.

# Answer types in call-by-name

- Plotkin (1975) published two Continuation Passing Style (CPS) transforms: call-by-value and call-by-name.
- The call-by-value one has been studied extensively, and is the basis of CPS compilers.
- Streicher (in the mid-1990) invented a new call-by-name transform. Hofmann and Streicher; completenes result for $\lambda\mu$-calculus.

**Theorem**

*The Plotkin (with answer type polymorphism) and Streicher CPS transforms are isomorphic (assuming parametricity)*

**Proof**:
Start from the isomorphism for the encoding of pairs,
lift it up to function types,
using Meyer and Wand's technique,
to construct an isomorphism $A^* \cong A^\circ$,
show that the CPS transforms commute with the isomorphism.
Meyer and Wand used back-and-forth translations to construct a retraction between direct and continuation semantics.
We use similar translations to contruct an isomorphism between two CPS transforms.

# Visualizing the isomorphism

Plotkin transform arises from the Streicher one by coding of pairs.

$$\begin{array}{rcl} (MN)^* & = & \lambda k.M^*(N^*, k) \\ \underline{MN} & = & \lambda k.\underline{M}\ (\lambda m.m\underline{N}k) \end{array} \qquad \begin{array}{c} (P, Q) \\ \downarrow \\ \lambda p.pPQ \end{array}$$

$$\begin{array}{rcl} (\lambda x.M)^* & = & \lambda(x, q).\quad M^* q \\ \underline{\lambda x.M} & = & \lambda k.k(\lambda xq.\underline{M}q) \end{array} \qquad \begin{array}{c} \pi_j\ P \\ \downarrow \\ P\,(\lambda x_1 x_2.x_j) \end{array}$$

# Frame Rules from Answer Types for Code Pointers

Hayo Thielecke

University of Birmingham, UK

# Separation logic and the frame rule

## Separating conjunction $P * Q$

Split the heap into disjoint parts, satisfying $P$ and $Q$.
[Reynolds, O'Hearn et. al.]

## Frame rule in separation Hoare logic

The conjunction $*$ allows local reasoning:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

## But what about code pointers?

$$p \mapsto \text{some code} \vdash \texttt{jmp}\ [p] : ???$$

# Answer type polymorphism

## Frame via answer type

In CPS, a type $A$ becomes:

$$\forall \alpha.(A \to \alpha) \to \alpha$$

instantiate answer type:

$$\forall \alpha'.(A \to R \to \alpha') \to R \to \alpha'$$

then uncurry:

$$\forall \alpha'.(((A \times R) \to \alpha') \times R) \to \alpha'$$

Consistently adds $\times R$ to **all** continuations.

Problem: With Hoare typing, assignment changes the types
That makes the above unsound;
like a frame rule with $\wedge$ instead of $*$.

Solution: Only instantiate answer types to types
that are "well-behaved": $\perp\perp$-closed.

# Setting: a typed assembly language fragment

## Syntax

$$
\begin{array}{rcl}
c & ::= & \texttt{jmp}\ f \\
& | & \texttt{jmp}\ [p] \\
& | & \texttt{movc}\ f\ \ p; c \\
& | & \texttt{movh}\ p\ q; c
\end{array}
$$

## Operational semantics

Machine state: $\langle$ current code block $c$, heap $h$, code segment $s$ $\rangle$

$$
\begin{array}{rcll}
\langle \texttt{jmp}\ f \mid h \mid s \rangle & \rightsquigarrow & \langle s(f) \mid h \mid s \rangle & \quad \text{where } f \notin \{\text{exit}, \text{error}\} \\
\langle \texttt{jmp}\ [p] \mid h \mid s \rangle & \rightsquigarrow & \langle s(h(p)) \mid h \mid s \rangle & \\
\langle \texttt{movc}\ f\ \ p; c \mid h \mid s \rangle & \rightsquigarrow & \langle c \mid h[p \mapsto f] \mid s \rangle & \\
\langle \texttt{movh}\ p\ q; c \mid h \mid s \rangle & \rightsquigarrow & \langle c \mid h[q \mapsto h(p)] \mid s \rangle &
\end{array}
$$

# Typing

## Three kinds of types

$$
\begin{array}{lcll}
A & ::= & p \mapsto B \mid A * A \mid A \wedge A \mid \texttt{emp} \mid \texttt{true} & \text{(heap types)} \\
B & ::= & A \twoheadrightarrow B \mid A \to B \mid \forall\alpha.B \mid \alpha & \text{(behaviour types)} \\
C & ::= & \alpha \mid A \twoheadrightarrow C \mid \forall\alpha.C & \text{(closed types)}
\end{array}
$$

$p \mapsto B$: pointer $p$ points to code with type $B$

$A \twoheadrightarrow B$: code that expects fresh heap satisfying $A$ and then behaves according to $B$

## Judgements for code blocks

$\Gamma \mid A \vdash c : B$ means that

if code segment has type $\Gamma$

and the current heap satisfies $A$

then code $c$ behaves as specified by $B$.

# Some typing rules

## Storing code pointers in the heap

$$\frac{\Gamma, f \triangleright B, \Gamma' \mid p \mapsto B \vdash c : C}{\Gamma, f \triangleright B, \Gamma' \mid p \mapsto B' \vdash \texttt{movc}\ f\ p; c : C} \ (\textsc{MovCode})$$

## Indirect jump via a code pointer

$$\frac{}{\Gamma \mid p \mapsto \forall \alpha.((p \mapsto \alpha) \twoheadrightarrow C) \vdash \texttt{jmp}\ [p] : C} \ (\textsc{IndJmp})$$

(where $\alpha$ not free in $C$)

## Arrow introduction and elimination are silent

$$\frac{\Gamma \mid A' * A \vdash c : B}{\Gamma \mid A' \vdash c : A \twoheadrightarrow B} \qquad \frac{\Gamma \mid A' \wedge A \vdash c : B}{\Gamma \mid A' \vdash c : A \to B}$$

# Avoiding unsoundness: Restrict answer types to $\perp\perp$-closed

## Orthogonality

Inspired by Pitts, Krivine, Melliés, Vouillon.
There: term $\perp$ evaluation context; here: machine state $\perp$ more state

## Definition

We write $\langle c \mid h \mid s \rangle \perp \langle h', s' \rangle$ if $h \# h'$ and $s \sharp s'$ implies $\langle c \mid h * h' \mid s \cup s' \rangle$ terminates or loops (but does not cause an error).

## Orthogonal and $\perp\perp$-closure

Define $S^{\perp} = \{y \mid x \perp y \text{ for all } x \in S\}$.
$(-)^{\perp\perp}$ is a closure operator. $S$ is closed iff $S^{\perp\perp} \subseteq S$.

## $\perp\perp$-closed means well-behaved

Intuitively, closed types are characterized by interaction with disjoint state.

# Soundness

We have some algebraic laws for $(-)^\perp$ and $\multimap$, e.g.

$$(\mathcal{A}_1 * \mathcal{A}_2)^\perp \;\; = \;\; \mathcal{A}_1 \mathrel{-\!\!*} \mathcal{A}_2^\perp$$

### Lemma

*$\perp\perp$-closed types are closed under $\mathcal{A} \mathrel{-\!\!*} (-)$; that is, if $\mathcal{C}$ is closed, then so is $\mathcal{A} \mathrel{-\!\!*} \mathcal{C}$.*

### Theorem

*The type system is sound for a realizability semantics.*

Hence there are enough closed types for framing...

# Frame laws via instantiation of answer types

## Functions in CPS

In continuation-passing style, functions are an idiom

$$A_1 \Rightarrow_r A_2$$

for a jump expecting a return address in $r$.

## Derived frame rule for functions

$$
\begin{array}{ll}
\texttt{jmp}\ f : A_1 \Rightarrow_r A_2 & \\
\hline
\texttt{jmp}\ f : \forall\alpha.(r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) \mathrel{-\!\!*} A_2 \mathrel{-\!\!*} \alpha)) \mathrel{-\!\!*} A_1 \mathrel{-\!\!*} \alpha & \text{by def. } (\Rightarrow_r) \\
\hline
\texttt{jmp}\ f : (r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) \mathrel{-\!\!*} A_2 \mathrel{-\!\!*} A' \mathrel{-\!\!*} \alpha')) \mathrel{-\!\!*} A_1 \mathrel{-\!\!*} A' \mathrel{-\!\!*} \alpha' & (\forall\mathrm{E}) \\
\hline
\texttt{jmp}\ f : (r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) \mathrel{-\!\!*} (A_2 * A') \mathrel{-\!\!*} \alpha')) \mathrel{-\!\!*} (A_1 * A') \mathrel{-\!\!*} \alpha' & (\equiv) \\
\hline
\texttt{jmp}\ f : \forall\alpha'.(r \mapsto (\forall\alpha_r.(r \mapsto \alpha_r) \mathrel{-\!\!*} (A_1 * A') \mathrel{-\!\!*} \alpha')) \mathrel{-\!\!*} (A_2 * A') \mathrel{-\!\!*} \alpha' & \\
\hline
\texttt{jmp}\ f : (A_1 * A') \Rightarrow_r (A_2 * A') &
\end{array}
$$

# Recursion via jumping

## Knots in the store

Code pointers allow one to create recursion dynamically.
E.g. code at $f$ is indirect jump to $p$; we update $p$ to point to $f$.
$\Rightarrow$ need recursive types.

## Recursive heaps

$$\mu\phi.A \equiv A[(\mu\phi.A)/\phi]$$

## A possible rule for general jumping

$$\frac{}{\Gamma \mid \mu\phi.p \mapsto (\phi \multimap C) \vdash \texttt{jmp}\ [p] : C}\ (\mu\text{JMP})$$

Semantically challenging (recursion *and* polymorphism), but fits into the overall picture.

# Conclusions

## Summary

**Answer types** Should not type $A$-accepting continuations as $A \to \mathbf{0}$; $A \to \alpha$ gives more information, together with $\forall\alpha$.

**Hoare typing** Need to be more careful, but $A \multimap \alpha$ well behaved.

**Frame rules** arise as idioms in continuation passing.

## Further work

- Extend to a richer assembly language
- Hypothetical and higher-order frame rules
- Recursion through the store (sketched in the paper)