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 $A \rightarrow \alpha$ as answer type

**Solution:** restrict answer types to $\bot\bot$-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 \( \neg \neg \neg \) 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]
Jouvelot and Gifford, 1988: goto $\rho$ and comefrom $\rho$ effects. Simplification: we use just $\rho$ for both of them.

Idea: Peirce’s law $((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha$ refined with regions.

\[
\Gamma \vdash_{\text{call/cc}} \forall \rho.((\alpha \rightarrow \beta) \rightarrow \alpha) \rho^{\cup e} \rightarrow \alpha ! \emptyset
\]

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

Control effects
Effect systems

\[
\frac{
\Gamma \vdash c \ M : A \xrightarrow{e_1} B ! e_2 \quad \Gamma \vdash c \ N : A ! e_3
}{
\Gamma \vdash c \ MN : B ! e_1 \cup e_2 \cup e_3
}
\]

\[
\frac{
\Gamma, x : A \vdash c \ M : B ! e
}{
\Gamma \vdash c \ \lambda x. M : A \xrightarrow{e} B ! \emptyset
}
\]

\[
\frac{
\Gamma \vdash c \ V : A ! \emptyset
}{
\Gamma \vdash c \ V : \forall \alpha. A ! \emptyset
}
\quad \alpha \notin \text{Tyvar}(\Gamma)
\]

\[
(Masking) \quad \frac{
\Gamma \vdash c \ M : A ! \rho
}{
\Gamma \vdash c \ \text{newreg} M : A ! \emptyset
}
\quad \rho \notin (\text{Reg}(A) \cup \text{Reg}(\Gamma))
\]
CPS transform on types and effects

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

\[
\begin{align*}
A \rightarrow^\emptyset B & = \forall \alpha. (\overline{B} \rightarrow \alpha) \rightarrow (\overline{A} \rightarrow \alpha) \quad \text{where } \alpha \text{ is fresh} \\
A \rightarrow^\rho B & = (\overline{B} \rightarrow \alpha_\rho) \rightarrow (\overline{A} \rightarrow \alpha_\rho) \\
\forall \alpha. A & = \forall \alpha. \overline{A} \\
\forall \rho. A & = \forall \alpha_\rho. \overline{A} \\
\overline{\alpha} & = \alpha
\end{align*}
\]

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 $\Gamma \vdash c \ M : A \ ! \rho$ then

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

If $\Gamma \vdash c \ M : A \ ! \emptyset$ then

$$\overline{\Gamma} \vdash \overline{M} : \forall \alpha. (\overline{A} \rightarrow \alpha) \rightarrow \alpha$$ where $\alpha$ is fresh
Example: no control effects

\[ \vdash_c \lambda z.\lambda x.x : B \not\rightarrow (A \not\rightarrow A) \not\rightarrow \emptyset \]

is transformed

\[ \lambda z.\lambda x.x = \lambda k_1.k_1(\lambda k_2 z. k_2(\lambda k_3 x.k_3 x)) \]

\[ B \not\rightarrow (A \not\rightarrow A) \]

\[ = \forall \alpha_2.((\forall \alpha_3. (\overline{A} \rightarrow \alpha_3) \rightarrow (\overline{A} \rightarrow \alpha_3)) \rightarrow \alpha_2) \rightarrow (\overline{B} \rightarrow \alpha_2) \]

Each \( \not\rightarrow \) has its own answer type.
“Upward continuation” example

\[
\lambda z. \text{call/cc}(\lambda k. \lambda x. k(\lambda y. x))
\]

\[
= \lambda k_1. k_1(\lambda k_2 z. k_2(\lambda k_3 x. k_2(\lambda k_4 y. k_4 x)))
\]

\[
\forall \rho. B \overset{\rho}{\rightarrow} (A \overset{\rho}{\rightarrow} A)
\]

\[
= \forall \alpha_\rho.(((\overline{A} \rightarrow \alpha_\rho) \rightarrow (\overline{A} \rightarrow \alpha_\rho)) \rightarrow \alpha_\rho) \rightarrow (\overline{B} \rightarrow \alpha_\rho)
\]

Control effects enforce answer type sharing.
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; completeness result for λμ-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 construct an isomorphism between two CPS transforms.
Visualizing the isomorphism

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

\[
(MN)^* = \lambda k. M^*(N^*, k) \quad (P, Q)
\]

\[
MN = \lambda k. M (\lambda m.mNk) \quad \lambda p.pPQ
\]

\[
(\lambda x.M)^* = \lambda (x, q). M^*q \quad \pi_j P
\]

\[
\lambda x.M = \lambda k. k(\lambda xq.Mq) \quad P(\lambda x_1x_2.x_j)
\]
Frame Rules from Answer Types for Code Pointers

Hayo Thielecke

University of Birmingham, UK
Separation logic and the frame rule

Separating conjunction $P \ast Q$
Split the heap into disjoint parts, satisfying $P$ and $Q$.
[Reynolds, O’Hearn et. al.]

Frame rule in separation Hoare logic
The conjunction $\ast$ allows local reasoning:

\[
\{P\} \ c \ \{Q\} \\
\{P \ast R\} \ c \ \{Q \ast R\}
\]

But what about code pointers?

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

Solution: Only instantiate answer types to types that are “well-behaved”: $\perp \perp$-closed.
Setting: a typed assembly language fragment

Syntax

\[ c ::= \begin{align*}
    &\text{jmp } f \\
| &\text{jmp } [p] \\
| &\text{movc } f \ p; c \\
| &\text{movh } p \ q; c
\end{align*} \]

Operational semantics

Machine state: \( \langle \text{current code block } c, \text{heap } h, \text{code segment } s \rangle \)

\[ \langle \text{jmp } f \ | \ h \ | \ s \rangle \rightsquigarrow \langle s(f) \ | \ h \ | \ s \rangle \quad \text{where } f \not\in \{\text{exit}, \text{error}\} \]

\[ \langle \text{jmp } [p] \ | \ h \ | \ s \rangle \rightsquigarrow \langle s(h(p)) \ | \ h \ | \ s \rangle \]

\[ \langle \text{movc } f \ p; c \ | \ h \ | \ s \rangle \rightsquigarrow \langle c \ | \ h[p \mapsto f] \ | \ s \rangle \]

\[ \langle \text{movh } p \ q; c \ | \ h \ | \ s \rangle \rightsquigarrow \langle c \ | \ h[q \mapsto h(p)] \ | \ s \rangle \]
Typing

Three kinds of types

| $A$ ::= $p \mapsto B$ | $A \star A$ | $A \land A$ | $\text{emp}$ | $\text{true}$ | (heap types) |
| $B$ ::= $A \rightarrow B$ | $A \rightarrow B$ | $\forall \alpha. B$ | $\alpha$ | (behaviour types) |
| $C$ ::= $\alpha$ | $A \rightarrow C$ | $\forall \alpha. C$ | (closed types) |

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

\[
\frac{\Gamma, f \triangleright B, \Gamma' \mid p \mapsto B \vdash c : C}{\Gamma, f \triangleright B, \Gamma' \mid p \mapsto B' \vdash \text{movc } f \ p; c : C} \quad \text{(MovCode)}
\]

Indirect jump via a code pointer

\[
\frac{\Gamma \mid p \mapsto \forall \alpha.((p \mapsto \alpha) \to* C) \vdash \text{jmp } [p] : C}{\Gamma \mid p \mapsto \forall \alpha.((p \mapsto \alpha) \to* C) \vdash \text{jmp } [p] : C} \quad \text{(INDJMP)}
\]

(\text{where } \alpha \text{ not free in } C)

Arrow introduction and elimination are silent

\[
\begin{align*}
\frac{\Gamma \mid A' \star A \vdash c : B}{\Gamma \mid A' \vdash c : A \to* B} \\
\frac{\Gamma \mid A' \land A \vdash c : B}{\Gamma \mid A' \vdash c : A \to B}
\end{align*}
\]
Avoiding unsoundness: Restrict answer types to \(\bot\)-closed

**Orthogonality**

Inspired by Pitts, Krivine, Melliès, Vouillon. There: term \(\bot\) evaluation context; here: machine state \(\bot\) more state

**Definition**

We write \(\langle c \mid h \mid s \rangle \perp \langle h', s' \rangle\) if \(h \not\approx h'\) and \(s \not\approx s'\) implies \(\langle c \mid h \ast h' \mid s \cup s' \rangle\) 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 $(-)^\bot$ and $\rightarrow^*$, e.g.

$$(A_1 \ast A_2)^\bot = A_1 \rightarrow^* A_2^\bot$$

**Lemma**

$\bot\bot$-closed types are closed under $A \rightarrow^* (-)$; that is, if $C$ is closed, then so is $A \rightarrow^* C$.

**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

\[
\begin{align*}
\text{jmp } f &: A_1 \Rightarrow_r A_2 \\
\text{jmp } f &: \forall \alpha. (r \mapsto (\forall \alpha_r. (r \mapsto \alpha_r) \rightarrow A_2 \rightarrow \alpha)) \rightarrow A_1 \rightarrow \alpha \\
\text{jmp } f &: (r \mapsto (\forall \alpha_r. (r \mapsto \alpha_r) \rightarrow A_2 \rightarrow A' \rightarrow \alpha')) \rightarrow A_1 \rightarrow A' \rightarrow \alpha' \\
\text{jmp } f &: \forall \alpha'. (r \mapsto (\forall \alpha_r. (r \mapsto \alpha_r) \rightarrow (A_1 \rightarrow A') \rightarrow \alpha')) \rightarrow (A_1 \rightarrow A') \rightarrow \alpha' \\
\text{jmp } f &: (A_1 \star A') \Rightarrow_r (A_2 \star A')
\end{align*}
\]
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 $f$. ⇒ need recursive types.

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

A possible rule for general jumping
$$\Gamma \mid \mu \phi. p \mapsto (\phi \to C) \vdash \text{jmp } [p] : C$$ ($\mu$JMP)

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

## Summary

**Answer types**

Should not type $A$-accepting continuations as $A \rightarrow 0$; $A \rightarrow \alpha$ gives more information, together with $\forall\alpha$.

**Hoare typing**

Need to be more careful, but $A \rightarrow^* \alpha$ 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)