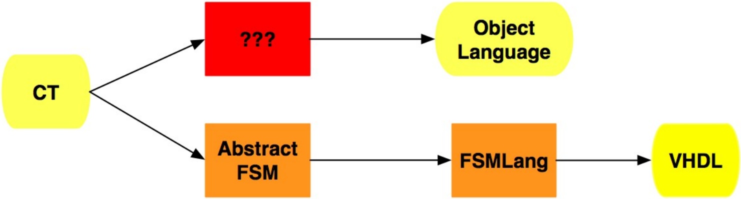
Taking the 'defunct' out of 'defunctionalization'

And putting the 'fun' back in 'defunct'

Benjamin Schulz

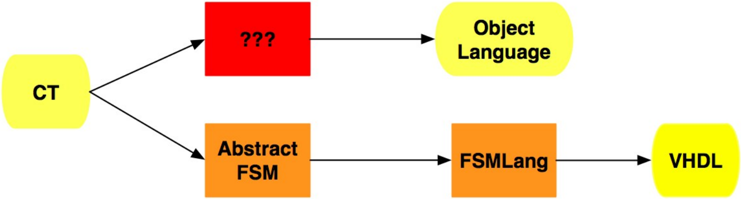
August 13, 2010

Architecture: Major Passes of the CT Compiler



What's the Big Picture? What's the Problem?

Architecture: Major Passes of the CT Compiler



... and architecture of what needs to be done

- Rigorous definition of semantics for CT
- Rigorous semantics of intermediate form
- Construct well-defined relation between the two
- Define compilation rules based on relation between CT and intermediate form
- *Overall, make design decisions that facilitate verification and correct code generation*

Intermediate Language: Design Objectives

- Specification – one we can stick with!
- Implementation
  - Write a code generation module that reflects the simulation relation between CT semantics and the OPSEM
  - Construct a language to serve as a reasonable jumping-off point for generation of Microblaze-ready code, e.g. C or MB ASM
- Verification
  - Preserve the resumption-monadic semantics of the CT source
  - Establish a formal relation between a CT program and its intermediate form
- Application
  - Produce working examples of secure, verifiable embedded system kernels

Previous Attempts

- In the beginning: TIC [2]
  - i.e. the Typed Interrupt Calculus
  - a language of interrupts and conditional atomicity
- ETIC
  - C with 'pthreads' primitive
  - TIC+?
- OESS
  - CT without monad operations
  - ... and with jumps, status registers
  - ETIC+?
- The CHEAP Machine
  - stab at an operational semantics
  - closer to a typed assembly language
  - ETIC++?

Previous Attempts

- In the beginning: TIC [2]
  - i.e. the Typed Interrupt Calculus
  - a language of interrupts and conditional atomicity
- ETIC
  - C with 'pthreads' primitive
  - TIC+?
- OESS
  - CT without monad operations
  - ... and with jumps, status registers
  - ETIC+?
- The CHEAP Machine
  - stab at an operational semantics
  - closer to a typed assembly language
  - ETIC++?
- *Where's the 0xDEADBEEF?*

First Attempt: ETIC

ETIC: Closest thing so far to code generation in action

```
else
{
   if ((!(__isdone(r1)) && ((True && (True && ((r1.__sig).__ctor__ == InEnv__enum))) && True)))
   {
      rho2 = (((r1.__sig).__instance__).__altinstance__4).__structfield__0;
      phi1 = (((r1.__sig).__instance__).__altinstance__4).__structfield__1;
      r3 = r1.__rts;
      stack<stack<pair<string,int>>> __r_59 = push(rho2,stk0);
      __r_60 = go0(__r_59, phi1);
      __r_61 = InEnvRe(__r_60);
      r_SIGNAL = __r_61;
      __ret = __run(r3);
      __TCB __r_63 = __slice(__r_62);
      __r_54 = __r_63;
```

Virtues

- Working implementation of generation to MB
- Running kernel examples exist
- Easy to compile to C

Vices

- Code generation (:: CT $\rightarrow$ ETIC) is b0rk3d
- Informal semantics, difficulty squaring with CT
- Syntax obscures essential features

Second Attempt: OESS

The Operational Execution Stream Semantics (OESS) Grammar

$$
\begin{array}{lcl}
S & ::= & A \ ; \ S \ \mid \ \mathsf{ifzero}\ S\ S\ S \ \mid \ \mathsf{jsr}\ L\ S \\
 & \mid & \mathsf{loop}\ S\ S \ \mid \ \mathsf{done}\ X \\
 & & \\
A & ::= & \mathsf{atom}(K) \\
 & \mid & \mathsf{tcreate}_{kernel}\ S \ \mid \ \mathsf{tcreate}_{user}\ T \\
 & \mid & \mathsf{kill}\ X \ \mid \ \mathsf{catch}\ X\ L \\
 & \mid & \mathsf{switch}_A\ X \ \mid \ \mathsf{switch}_Z\ X \ \mid \ \mathsf{break} \\
 & & \\
T & ::= & \mathsf{cont}(K)\ ;\ T \ \mid \ \mathsf{throw}(X)\ ;\ T \\
 & \mid & \mathsf{ifzero}\ T\ T\ T \ \mid \ \mathsf{jsr}\ L\ T \\
 & \mid & \mathsf{loop}\ T\ T \ \mid \ \mathsf{done}\ X
\end{array}
\qquad
\begin{array}{lcl}
M & ::= & R := X \ \mid \ \mathsf{tst}\ X \ \mid \ \mathsf{nop} \\
K & ::= & M\ ;\ K \ \mid \ \mathsf{ifzero}\ K\ K\ K \ \mid \ \mathsf{jsr}\ L\ K \mid M \\
 & & \\
X & ::= & R \ \mid \ L \\
 & \mid & \text{-}X \ \mid \ X + X \ \mid X \text{ - } X \\
 & \mid & X * X \ \mid X \ / \ X \\
 & \mid & !X \ \mid \ X \ \&\& \ X \ \mid \ X \ || \ X \\
 & \mid & Int \ \mid \ \mathsf{nil} \\
 & & \\
R & ::= & Var \ \mid \ Var[X] \\
 & \mid & \mathsf{r}_{ret} \ \mid \mathsf{r}_{pc} \mid \ \mathsf{r}_{signal} \ \mid \mathsf{r}_{tct} \mid \mathsf{r}_Z \\
 & & \\
L & ::= & Label
\end{array}
$$

A typical transliteration CT → OESS

$$
\begin{array}{l}
step_R\ k \bigstar_R \lambda v. \\
step_R\ (k'\ v) \bigstar_R \\
f
\end{array}
\quad \Longrightarrow_{compile} \quad
$$

```
atom(jsr __k; v := ret);
atom(
__k0_param0 := v; jsr __k0;
__f_param0 := ret);
jsr __f;
done ret
```

Third Attempt: CHEAP Abstract Machines

Abstract machine consisting of:

A quintuple $\langle C, H, E, A, P \rangle$ incorporating:

- a *Context* of variable bindings
- a *Heap* of labeled code blocks
- an active *Execution stream*
- a stack of suspended *execution-Abstractions*
- a *Pool* of threads indexed by unique identifiers

Ticking the program counter

$$\frac{H(l) = E,\ \ C(\mathrm{r}_{tid}) = n}{(C, H, E, A, P) \rightarrow (C, H, E, A, P[< n, C\prime[\mathrm{r}_{pc} \mapsto l], E\prime, A\prime >])}\ (\textit{thread update})$$

An operation of context switching

$$\frac{P(t) =< n\prime, C\prime, E\prime, A\prime >\ ,\ \ C(\mathrm{r}_{tid}) = n}{(C, H, \mathrm{switch}_A\ \ t;\ \ E, A, P) \rightarrow (C\prime[\mathrm{r}_{parent} \mapsto n, \mathrm{r}_{tid} \mapsto n\prime], H, E\prime, A\prime, P)}\ (\textit{unconditional switch})$$

Lessons Learned, Solutions Proposed

Don't do this ...

- Model the CT intermediate form on C
- Make threads (in the usual sense) primitive
- Try to construct an abstract machine from a low-level system perspective
- Plan on retroactively fitting the intermediate form to the source semantics

Do this instead

- Construct the semantics of the intermediate form in express relation to those of the CT source denotation
- Model the intermediate form on typed assembly language
- Derive threads, processes, handlers from resumptions and their types, not the other way around

Lessons Learned, Solutions Proposed

Don't do this ...

- Model the CT intermediate form on C
- Make threads (in the usual sense) primitive
- Try to construct an abstract machine from a low-level system perspective
- Plan on retroactively fitting the intermediate form to the source semantics

Do this instead

- Construct the semantics of the intermediate form in express relation to those of the CT source denotation
- Model the intermediate form on typed assembly language
- Derive threads, processes, handlers from resumptions and their types, not the other way around

How do these nice-sounding ideas work in practice?

Back to the Past: A Definitional Evaluator for CT

Definition: for a language L with a set of expressible values V, an *evaluator* is a function $eval :: L \rightarrow V$

Example from CT: evaluating atomicity

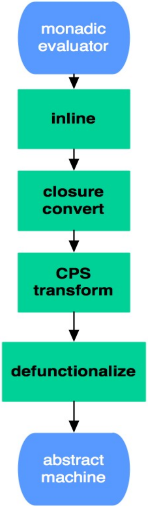
$eval_R :: Env \rightarrow CT \rightarrow ResT\ (StateT\ Mem\ Id)\ Val$

$eval_R\ e\ (\mathtt{step}_R\ \phi) = Pause((eval_K\ e\ \phi) \bigstar_R (\eta_K \circ \eta_R))$

- CT has *first-class resumptions*, meaning resumptions are expressible values
- Programs evaluate to monadic terms reflecting the specified effects, i.e. concurrency and state
- *How can we use a definitional evaluator for CT?*

The Ager-Danvy-Midtgaard (ADM) Transformations

...defunctionalized continuation-passing evaluators are abstract machines.[1]

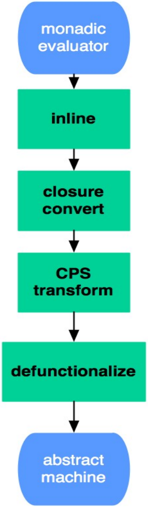


Producing an abstract machine from a monadic evaluator

- Inline all monadic operations to make computational effects explicit in evaluation
- Closure convert data types to reduce higher-order data to first-order values
- CPS transform the evaluator to materialize its control flow as functions "to the rest of the computation" i.e. continuations
- Defunctionalize continuations to produce a transition system

The Ager-Danvy-Midtgaard (ADM) Transformations

...defunctionalized continuation-passing evaluators are abstract machines.[1]



Producing an abstract machine from the CT evaluator

- Inlining the monadic operations of $R\ a$ simply produces sequential composition
- Closure conversion is unnecessary because CT has no higher-order data
- CPS is implicit in the definition of expressible values in CT, i.e. the monad $ResT\ (StateT\ MemId)\ Val$
- Defunctionalization follows directly, with states delineated by applications of the *Pause* constructor

ADM in Practice: a CT Operational Semantics via DFun

Exposing control flow with resumptions

$$step_R\ k_0 \bigstar_R step_R\ k_1 \bigstar_R step_R \bigstar_R step_R\ k_2 \equiv$$
$$Pause(k_0 \bigstar_K \eta_K(Pause(k_1 \bigstar_K \eta_K(Pause(k_2 \bigstar_K (\eta_K \circ Done))))))$$

- k_i are blocks of imperative code
- *Pause* delineates blocks from one another
- When the behavior of a system satisfies this semantics, the argument to η_K specifies the remainder of the computation

Transition system from explicit control flow

$$Pause(k_0 \bigstar_K \eta_K(Pause(k_1 \bigstar_K \eta_K(Pause(k_2 \bigstar_K (\eta_K \circ Done))))))$$
$$\sim$$
$$\lceil \texttt{\_\_k0: stmt0;...stmtN; jmp \_\_k1} \rceil$$
$$\lceil \texttt{\_\_k1: stmt0;...stmtM; jmp \_\_k2} \rceil$$
$$\lceil \texttt{\_\_k2: stmt0;...stmtP; exit ret} \rceil$$

The RKI abstract machine

r	$\Longrightarrow$	$_{init}$	$\langle r, E_{init}, \text{nil}\rangle_R$
$\langle Pause\ \ k, E, R\rangle$	$\Longrightarrow$	$_R$	$\langle k, R, E\rangle_K$
$\langle BZ\ \ b\ \ r\ \ r\prime,\ \ E, R\rangle$	$\Longrightarrow$	$_R$	$\langle (r, r\prime), b, R, E\rangle_I$
$\langle Loop\ \ r\ \ r\prime, E, R\rangle$	$\Longrightarrow$	$_R$	$\langle r, E, (r, r\prime) :: R\rangle_R$
$\langle Done\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow$	$_R$	$\langle r, E[ret \mapsto v], (r, r\prime) :: R\rangle_R$
$\langle Break\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow$	$_R$	$\langle r\prime, E[ret \mapsto v], R\rangle_R$
$\langle get_x\ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow$	$_K$	$\langle k, E[ret \mapsto \lceil x \rceil]\rangle_K$
$\langle put_x\ \ e\ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow$	$_K$	$\langle k, E[x \mapsto \lceil e \rceil,\ \ ret \mapsto \text{nil}]\rangle_K$
$\langle \lambda x.k, E\rangle$	$\Longrightarrow$	$_K$	$\langle k, E[x \mapsto \lceil ret \rceil]\rangle_K$
$\langle \eta_K\ \ r, R, E\rangle$	$\Longrightarrow$	$_K$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), x, R, E\rangle$	$\Longrightarrow$	$_I$	$\langle (r, r\prime), \lceil x \rceil, R, E\rangle_I$
$\langle (r, r\prime), 1, R, E\rangle$	$\Longrightarrow$	$_I$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), 0, R, E\rangle$	$\Longrightarrow$	$_I$	$\langle r\prime, E, R\rangle_R$
$\langle Done\ \ x, E, \text{nil}\rangle$	$\Longrightarrow$	$_{halt}$	x

The RKI abstract machine

r	$\Longrightarrow_{init}$	$\langle r, E_{init}, \text{nil}\rangle_R$
$\langle \mathit{Pause}\ \ k, E, R\rangle$	$\Longrightarrow_R$	$\langle k, R, E\rangle_K$
$\langle \mathit{BZ}\ \ b\ \ r\ \ r\prime,\ \ E, R\rangle$	$\Longrightarrow_R$	$\langle (r, r\prime), b, R, E\rangle_I$
$\langle \mathit{Loop}\ \ r\ \ r\prime, E, R\rangle$	$\Longrightarrow_R$	$\langle r, E, (r, r\prime) :: R\rangle_R$
$\langle \mathit{Done}\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow_R$	$\langle r, E[\mathit{ret} \mapsto v], (r, r\prime) :: R\rangle_R$
$\langle \mathit{Break}\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow_R$	$\langle r\prime, E[\mathit{ret} \mapsto v], R\rangle_R$
$\langle \mathit{get}_x\ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow_K$	$\langle k, E[\mathit{ret} \mapsto \lceil x\rceil]\rangle_K$
$\langle \mathit{put}_x\ \ e\ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow_K$	$\langle k, E[x \mapsto \lceil e\rceil,\ \ \mathit{ret} \mapsto \text{nil}]\rangle_K$
$\langle \lambda x.k, E\rangle$	$\Longrightarrow_K$	$\langle k, E[x \mapsto \lceil \mathit{ret}\rceil]\rangle_K$
$\langle \eta_K\ \ r, R, E\rangle$	$\Longrightarrow_K$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), x, R, E\rangle$	$\Longrightarrow_I$	$\langle (r, r\prime), \lceil x\rceil, R, E\rangle_I$
$\langle (r, r\prime), 1, R, E\rangle$	$\Longrightarrow_I$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), 0, R, E\rangle$	$\Longrightarrow_I$	$\langle r\prime, E, R\rangle_R$
$\langle \mathit{Done}\ \ x, E, \text{nil}\rangle$	$\Longrightarrow_{halt}$	x

Atomic state delimiter

From RPS to a CT Abstract Machine

The RKI abstract machine

r	$\Longrightarrow_{init}$	$\langle r, E_{init}, \text{nil}\rangle_R$
$\langle Pause\ \ k, E, R\rangle$	$\Longrightarrow_R$	$\langle k, R, E\rangle_K$
$\langle BZ\ \ b\ \ r\ \ r\prime,\ \ E, R\rangle$	$\Longrightarrow_R$	$\langle (r, r\prime), b, R, E\rangle_I$
$\langle Loop\ \ r\ \ r\prime, E, R\rangle$	$\Longrightarrow_R$	$\langle r, E, (r, r\prime) :: R\rangle_R$
$\langle Done\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow_R$	$\langle r, E[ret \mapsto v], (r, r\prime) :: R\rangle_R$
$\langle Break\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow_R$	$\langle r\prime, E[ret \mapsto v], R\rangle_R$
$\langle get_x\ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow_K$	$\langle k, E[ret \mapsto \lceil x\rceil]\rangle_K$
$\langle put_x\ \ e\ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow_K$	$\langle k, E[x \mapsto \lceil e\rceil,\ \ ret \mapsto \text{nil}]\rangle_K$
$\langle \lambda x.k, E\rangle$	$\Longrightarrow_K$	$\langle k, E[x \mapsto \lceil ret\rceil]\rangle_K$
$\langle \eta_K\ \ r, R, E\rangle$	$\Longrightarrow_K$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), x, R, E\rangle$	$\Longrightarrow_I$	$\langle (r, r\prime), \lceil x\rceil, R, E\rangle_I$
$\langle (r, r\prime), 1, R, E\rangle$	$\Longrightarrow_I$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), 0, R, E\rangle$	$\Longrightarrow_I$	$\langle r\prime, E, R\rangle_R$
$\langle Done\ \ x, E, \text{nil}\rangle$	$\Longrightarrow_{halt}$	x

Imperative actions

The RKI abstract machine

r	$\Longrightarrow_{init}$	$\langle r, E_{init}, \text{nil}\rangle_R$
$\langle Pause\ \ k, E, R\rangle$	$\Longrightarrow_R$	$\langle k, R, E\rangle_K$
$\langle BZ\ \ b\ \ r\ \ r\prime,\ \ E, R\rangle$	$\Longrightarrow_R$	$\langle (r, r\prime), b, R, E\rangle_I$
$\langle Loop\ \ r\ \ r\prime, E, R\rangle$	$\Longrightarrow_R$	$\langle r, E, (r, r\prime) :: R\rangle_R$
$\langle Done\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow_R$	$\langle r, E[ret \mapsto v], (r, r\prime) :: R\rangle_R$
$\langle Break\ \ v, E, (r, r\prime) :: R\rangle$	$\Longrightarrow_R$	$\langle r\prime, E[ret \mapsto v], R\rangle_R$
$\langle get_x\ \ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow_K$	$\langle k, E[ret \mapsto \lceil x\rceil]\rangle_K$
$\langle put_x\ \ e\ \bigstar_K\ \ k, R, E\rangle$	$\Longrightarrow_K$	$\langle k, E[x \mapsto \lceil e\rceil,\ \ ret \mapsto \text{nil}]\rangle_K$
$\langle \lambda x.k, E\rangle$	$\Longrightarrow_K$	$\langle k, E[x \mapsto \lceil ret\rceil]\rangle_K$
$\langle \eta_K\ \ r, R, E\rangle$	$\Longrightarrow_K$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), x, R, E\rangle$	$\Longrightarrow_I$	$\langle (r, r\prime), \lceil x\rceil, R, E\rangle_I$
$\langle (r, r\prime), 1, R, E\rangle$	$\Longrightarrow_I$	$\langle r, E, R\rangle_R$
$\langle (r, r\prime), 0, R, E\rangle$	$\Longrightarrow_I$	$\langle r\prime, E, R\rangle_R$
$\langle Done\ x, E, \text{nil}\rangle$	$\Longrightarrow_{halt}$	x

Iteration

Example: Defunctionalizing a Trivial Kernel

Dumb Little Kernel

```
 main =   loop_R
          (\ r ->
                   case r of
                   Pause x -> step_R x >>= \r0 -> return r0
                   Done v -> break_R v
          ) phi
phi =
step_R(get G >>= \v -> put G (v + 1) >> return v) >>= \v ->
return v
```

Dumb Little Assembly Fragment

```
__main:
r := __phi
s := 1
__loop:
tst s
bz __case1
__case0:
jsr r
jmp __loop

__case1:
ret := ret
jmp __loop_exit
__loop_exit:
halt

__phi:
s := 1
ret := G
v := ret
G := v + 1
ret := v
r := __phi0
rts

__phi0:
ret := v
s := 0
rts
```

Ideal Target: A Typed Assembly Language (TAL)

A typed assembly language of linear continuations [3]

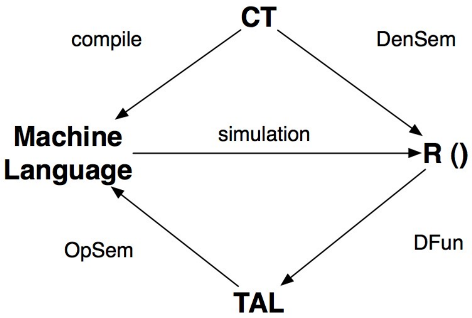
$$
\begin{array}{rrcl}
\text{Security Labels} & \ell, \mathsf{pc} & \in & \mathcal{L} \\
\text{Base Types} & \tau & ::= & \mathsf{int} \mid 1 \mid \sigma\ \mathsf{ref} \mid [\mathsf{pc}](\sigma, \kappa) \to 0 \\
\text{Security Types} & \sigma & ::= & \tau_\ell \\
\text{Linear Types} & \kappa & ::= & \sigma \to 0 \\
\\
\text{Base Values} & bv & ::= & n \mid \langle\rangle \mid L^\sigma \mid \lambda[\mathsf{pc}] f(x{:}\sigma, y{:}\kappa).\, e \\
\text{Values} & v & ::= & x \mid bv_\ell \\
\text{Linear Values} & lv & ::= & y \mid \lambda\langle\mathsf{pc}\rangle(x{:}\sigma).\, e \\
\text{Primitives} & prim & ::= & v \mid v \oplus v \mid \mathtt{deref}(v) \\
\\
\text{Expressions} & e & ::= & \mathtt{let}\ x = prim\ \mathtt{in}\ e \\
& & \mid & \mathtt{let}\ x = \mathtt{ref}_\ell^\sigma\ v\ \mathtt{in}\ e \\
& & \mid & \mathtt{set}\ v := v\ \mathtt{in}\ e \\
& & \mid & \mathtt{letlin}\ y = lv\ \mathtt{in}\ e \\
& & \mid & \mathtt{if0}\ v\ \mathtt{then}\ e\ \mathtt{else}\ e \\
& & \mid & \mathtt{goto}\ v\ v\ lv \\
& & \mid & \mathtt{lgoto}\ lv\ v
\end{array}
$$

Next steps

- Memory safety via low-level type system
- Operationalizing memory maps

Verification Objective: A Simulation Relation Between CT and TAL

Goal: implement this relation



- CT denotes resumptions, i.e. explicit action sequences
- R defunctionalizes to transition systems with mutable state
- TAL realizes an operational semantics, simulates resumptions
- That's model-driven engineering

"With just one line of code, HASK Lab rips all these signed, big-budget motherf*ckers." Questions?

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard.
A functional correspondence between monadic evaluators and abstract machines for languages with computational effects.
Theor. Comput. Sci., 342(1):149–172, 2005.

Jens Palsberg and Di Ma.
A typed interrupt calculus.
In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 291–310, London, UK, 2002. Springer-Verlag.

Steve Zdancewic and Andrew C. Myers.
Secure information flow via linear continuations.
Higher Order Symbol. Comput., 15(2-3):209–234, 2002.