Case studies: Control-Flow Analysis and Abstract Debugging

Jan Midtgaard

Week 6, Abstract Interpretation

Aarhus University, Q4 - 2012

Last time

A catalogue of abstractions

- □ Toolbox abstractions
- □ Structural abstractions: sums, pairs/tuples, . . .
- □ Numerical abstractions: constants, intervals, congruences, polyhedra, . . .
- □ Concretization-based abstract interpretation, briefly

A retrospective on the 3 counter machine analysis, incl. constraint extraction

Today

Based on two research articles:

- *Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation*, Midtgaard and Jensen, IC'12 (ICFP'09)
- *Abstract Debugging of Higher-Order Imperative Languages*, Bourdoncle, PLDI'93

Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation

What: control-flow analysis

Control-flow analysis (CFA) is a static analysis for determining inter-procedural control-flow:

for each call-site predict which function is called

What: control-flow analysis

Control-flow analysis (CFA) is a static analysis for determining inter-procedural control-flow:

for each call-site predict which function is called

Predictions are conservative due to our failure to solve the Halting problem

What: control-flow analysis

Control-flow analysis (CFA) is a static analysis for determining inter-procedural control-flow:

for each call-site predict which function is called

Predictions are conservative due to our failure to solve the Halting problem

CFA has been the subject of much research:
Jones:ICALP81, Rozas:BSc84, Shivers:PLDI88, Sestoft:FPCA89, Bondorf:SCP91, Henglein:TR92, Heintze:LFP94, Palsberg:TOPLAS95, Jagannathan-Wright:SAS95, Nielson-Nielson:POPL97, Ashley-Dybvig:TOPLAS98, Might-Shivers:POPL06 . . . (just to name a few)

Why CFA?

CFA is useful for program transformers, and other static analyses in compilers, program verification, etc.

As such CFA can be considered an "*analysis primitive*"

The analysis is relevant to all languages with some form of *procedural parameters* (C, C#, JavaScript, ...) hence not just functional languages (ML, Scheme, Haskell, ...)

Textbook control-flow analysis

A textbook constraint-based CFA emits constraints of the form:

$$\{u_1\} \subseteq rhs_1 \ \wedge \ \ldots \ \wedge \ \{u_n\} \subseteq rhs_n \implies lhs \subseteq rhs$$

Textbook control-flow analysis

A textbook constraint-based CFA emits constraints of the form:

$$\{u_1\} \subseteq rhs_1 \;\wedge\; \ldots \;\wedge\; \{u_n\} \subseteq rhs_n \implies lhs \subseteq rhs$$

For all $\texttt{(}\lambda\ \texttt{(x) e)}$ in the program we generate:

$$\{\texttt{(}\lambda\ \texttt{(x) e)}\} \subseteq [\![\texttt{(}\lambda\ \texttt{(x) e)}]\!]$$

and for all $\texttt{(}\lambda\ \texttt{(x) e)}$ and $\texttt{(e}_0\,\texttt{e}_1\texttt{)}$ we generate:

$$\{\texttt{(}\lambda\ \texttt{(x) e)}\} \subseteq [\![\texttt{e}_0]\!] \implies [\![\texttt{e}_1]\!] \subseteq [\![\texttt{x}]\!] \;\wedge\; [\![\texttt{e}]\!] \subseteq [\![\texttt{(e}_0\,\texttt{e}_1\texttt{)}]\!]$$

which are then solved iteratively by a constraint solver.

Abstract Interpretation

Canonical (Galois connection-based) abstract interpretation is presented as (Cousot:MJ81):

- □ a collecting semantics (e.g., reachable states) of a
- □ transition system,
- □ systematically approximated through Galois connections

Lots of variations (Cousot-Cousot:JLC92):
trace-based collecting semantics, concretization-only, abstraction-only, . . .

Two takes to correctness

- Standard CFAs are verified a posteriori (semantics, analysis, correctness proof), whereas
- Abstract interpretation-based analyses are correct by construction (Galois connection-based calculation)

Two takes to correctness

- Standard CFAs are verified a posteriori (semantics, analysis, correctness proof), whereas
- Abstract interpretation-based analyses are correct by construction (Galois connection-based calculation)

Which CFA do we obtain by taking the Cousot-route?

(Cliffhanger)

Two takes to correctness

- Standard CFAs are verified a posteriori (semantics, analysis, correctness proof), whereas
- Abstract interpretation-based analyses are correct by construction (Galois connection-based calculation)

Which CFA do we obtain by taking the Cousot-route?

How is the result related to constraint-based CFA?

(Another cliffhanger)

Enough cliffhanging: Three contributions

- ☐ Derivation of a call-return CFA for ANF by abstract interpretation
- ☐ Extraction of an equivalent constraint-based CFA
- ☐ Lock-step equivalence proof to earlier derived CPS-based CFA

Outline

Source language: ANF

ANF grammar:

$$
\begin{array}{llr}
P \ni \mathtt{p} ::= \mathtt{s} & & \text{(programs)} \\
T \ni \mathtt{t} ::= c \mid \mathtt{x} \mid \mathtt{(}\lambda\ \mathtt{(x)\ s)} & & \text{(trivial expressions)} \\
C \ni \mathtt{s} ::= \mathtt{t} & & \text{(serious expressions)} \\
\quad \mid \mathtt{(let\ ((x\ t))\ s)} & & \\
\quad \mid \mathtt{(t_0\ t_1)} & & \\
\quad \mid \mathtt{(let\ ((x\ (t_0\ t_1)))\ s)} & &
\end{array}
$$

Following Reynolds, the grammar distinguishes serious and trivial expressions.

Wanted: transition system

Let's use the C_aEK abstract machine!

The C_aEK (Flanagan-al:PLDI93) is a simple three component machine:

- **C** – the code component (a serious expression)
- **E** – the environment component
- **K** – the stack component

Furthermore (as we shall see) the machine is tail-call optimized

C_aEK semantics

Values, environments, and stacks:

$$Val \ni w ::= c \mid [\texttt{(}\lambda\ \texttt{(x)}\ \texttt{s)}, e]$$
$$Env \ni e ::= \bullet \mid e[\texttt{x} \mapsto w]$$
$$K \ni k ::= \texttt{stop} \mid [\texttt{x}, \texttt{s}, e] :: k$$

Helper function:

$$\mu : T \times Env \rightharpoonup Val$$
$$\mu(c, e) = c$$
$$\mu(\texttt{x}, e) = e(\texttt{x})$$
$$\mu(\texttt{(}\lambda\ \texttt{(x)}\ \texttt{s)}, e) = [\texttt{(}\lambda\ \texttt{(x)}\ \texttt{s)}, e]$$

Machine transitions:

$$\langle \texttt{t}, e, [\texttt{x}, \texttt{s}', e'] :: k' \rangle \longrightarrow \langle \texttt{s}', e'[\texttt{x} \mapsto \mu(\texttt{t}, e)], k' \rangle$$
$$\langle \texttt{(let ((x t)) s)}, e, k \rangle \longrightarrow \langle \texttt{s}, e[\texttt{x} \mapsto \mu(\texttt{t}, e)], k \rangle$$
$$\langle \texttt{(t}_0\,\texttt{t}_1\texttt{)}, e, k \rangle \longrightarrow \langle \texttt{s}', e'[\texttt{x} \mapsto w], k \rangle$$
$$\text{if } [\texttt{(}\lambda\ \texttt{(x)}\ \texttt{s}'\texttt{)}, e'] = \mu(\texttt{t}_0, e) \text{ and } w = \mu(\texttt{t}_1, e)$$
$$\langle \texttt{(let ((x (t}_0\,\texttt{t}_1\texttt{))) s)}, e, k \rangle \longrightarrow \langle \texttt{s}', e'[\texttt{y} \mapsto w], [\texttt{x}, \texttt{s}, e] :: k \rangle$$
$$\text{if } [\texttt{(}\lambda\ \texttt{(y)}\ \texttt{s}'\texttt{)}, e'] = \mu(\texttt{t}_0, e) \text{ and } w = \mu(\texttt{t}_1, e)$$

Next step: Collecting semantics

We choose a standard reachable states collecting semantics (Cousot:MJ81) defined in terms of the transition function:

$$F : \wp(C \times Env \times K) \to \wp(C \times Env \times K)$$
$$F(S) = I_{\mathrm{p}} \cup \{s \mid \exists s' \in S : s' \longrightarrow s\}$$

where $I_{\mathrm{p}} = \{\langle \mathtt{p}, \bullet, [\mathtt{x}_{\mathtt{r}}, \mathtt{x}_{\mathtt{r}}, \bullet] :: \mathtt{stop}\rangle\}$

Next step: Collecting semantics

We choose a standard reachable states collecting semantics (Cousot:MJ81) defined in terms of the transition function:

$$F : \wp(C \times Env \times K) \to \wp(C \times Env \times K)$$
$$F(S) = I_{\mathrm{p}} \cup \{s \mid \exists s' \in S : s' \longrightarrow s\}$$

where $I_{\mathrm{p}} = \{\langle \mathtt{p}, \bullet, [\mathtt{x}_{\mathtt{r}}, \mathtt{x}_{\mathtt{r}}, \bullet] :: \mathtt{stop}\rangle\}$

The reachable states can now be expressed as $\mathrm{lfp}\, F$.

Next step: Collecting semantics

We choose a standard reachable states collecting semantics (Cousot:MJ81) defined in terms of the transition function:

$$
\begin{aligned}
F &: \wp(C \times Env \times K) \rightarrow \wp(C \times Env \times K) \\
F(S) &= I_{\mathrm{p}} \cup \{s \mid \exists s' \in S : s' \longrightarrow s\}
\end{aligned}
$$

where $I_{\mathrm{p}} = \{\langle \mathrm{p}, \bullet, [\mathtt{x}_{\mathtt{r}}, \mathtt{x}_{\mathtt{r}}, \bullet] :: \mathtt{stop} \rangle\}$

The reachable states can now be expressed as $\mathrm{lfp}\, F$.

The collecting semantics is ideal in terms of precision

It is also as hard as running the program hence we need to approximate it.

Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$

and the "pushing of α's":

$$\alpha(F(S)) = \ldots = \ldots \subseteq_{\otimes} \ldots$$

Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$
$$\ldots \text{calculate} \ldots$$

and the "pushing of α's":

$$\alpha(F(S)) = \ldots = \ldots \subseteq_{\otimes} \ldots$$

Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$

$$\ldots \text{calculate} \ldots$$

$$\ldots \text{calculate} \ldots$$

and the "pushing of α's":

$$\alpha(F(S)) = \ldots = \ldots \subseteq_{\otimes} \ldots$$

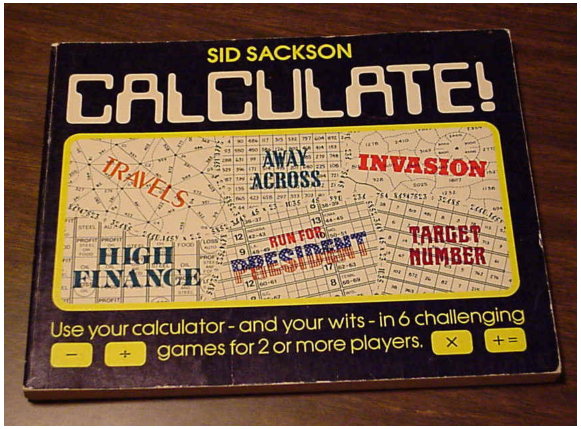
Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$

and the "pushing of α's":

$$\alpha(F(S)) = \ldots = \ldots \subseteq_{\otimes} \ldots$$



Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$

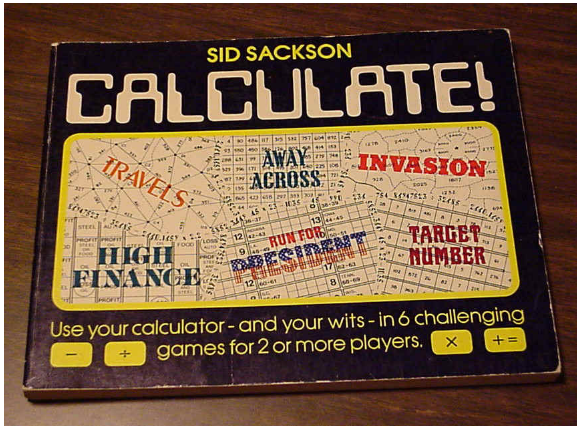
$$\ldots \text{calculate} \ldots$$

$$\ldots \text{calculate} \ldots$$

$$\ldots \text{calculate} \ldots = F^\sharp(S)$$

and the "pushing of α's":

$$\alpha(F(S)) = \ldots = \ldots \subseteq_\otimes \ldots$$



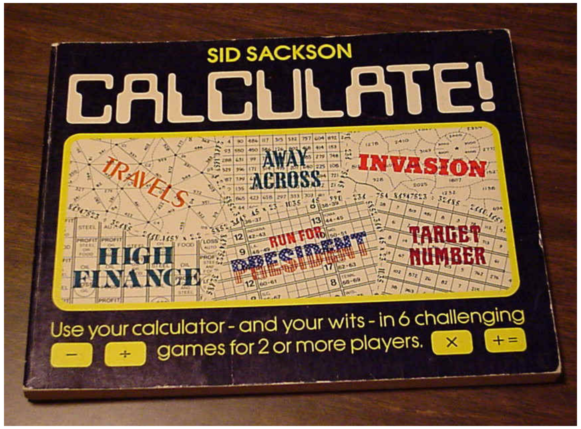
Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots = F^{\sharp}(S)$$

and the "pushing of α's":

$$\alpha(F(S)) = \ldots = \ldots \subseteq_{\otimes} \ldots$$
$$\ldots \text{calculate} \ldots$$



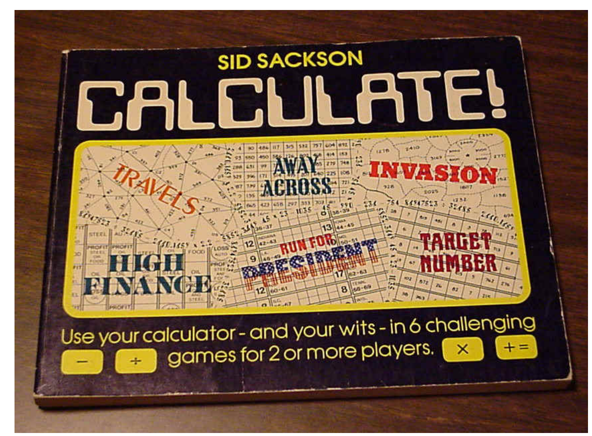
Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$
\begin{aligned}
\alpha(F(\gamma(S))) &= \ldots = \ldots = \ldots \\
&\ldots \text{calculate} \ldots \\
&\ldots \text{calculate} \ldots \\
&\ldots \text{calculate} \ldots = F^\sharp(S)
\end{aligned}
$$

and the "pushing of α's":

$$
\begin{aligned}
\alpha(F(S)) &= \ldots = \ldots \subseteq_\otimes \ldots \\
&\ldots \text{calculate} \ldots \\
&\ldots \text{calculate} \ldots
\end{aligned}
$$

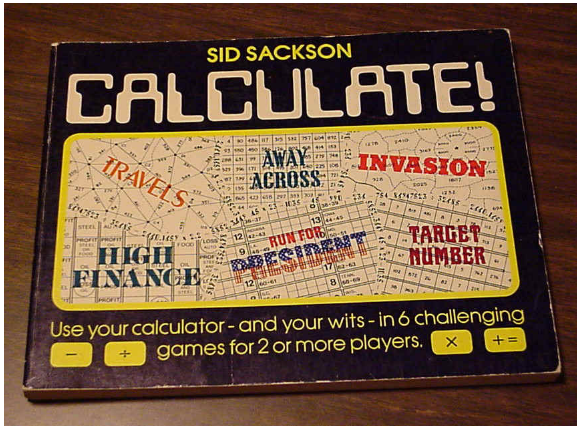


Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots = F^{\sharp}(S)$$

and the “pushing of α’s”:



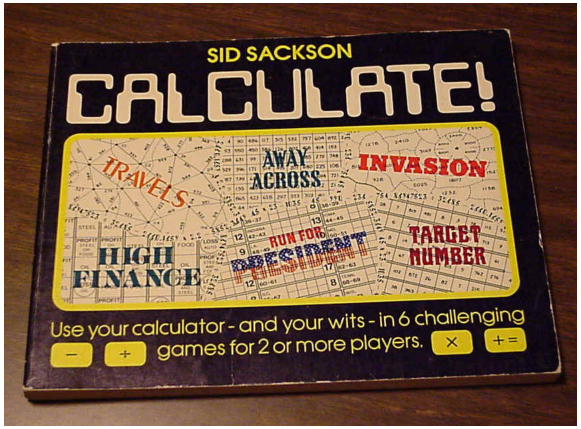
$$\alpha(F(S)) = \ldots = \ldots \subseteq_{\otimes} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$

Approximating the collecting semantics

We calculate abstract transfer functions using well-known strategies:

$$\alpha(F(\gamma(S))) = \ldots = \ldots = \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots = F^{\sharp}(S)$$

and the "pushing of α's":



$$\alpha(F(S)) = \ldots = \ldots \subseteq_{\otimes} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots$$
$$\ldots \text{calculate} \ldots = F^{\sharp}(\alpha(S))$$

Derivation outline

$$\begin{tikzcd}
\wp(C \times Env \times K) \arrow[d, shift right, "\alpha_\times"'] & \text{collecting semantics} \\
\wp(C) \times \wp(C \times K) \times \wp(Env) \arrow[u, shift right, "\gamma_\times"'] \arrow[d, shift right, "\rho"'] & \text{-} \\
\rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) \arrow[u, shift right, "1"'] \arrow[d, shift right, "\alpha_\otimes"'] & \text{-} \\
\wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp \arrow[u, shift right, "\gamma_\otimes"'] & \text{analysis}
\end{tikzcd}$$

Derivation outline

- ☐ Project reachable expressions
- ☐ Preserve expr-stack relation
- ☐ Merge environments

$$\begin{tikzcd}
\wp(C \times Env \times K) \arrow[d, shift right, "\alpha_\times"'] \\
\wp(C) \times \wp(C \times K) \times \wp(Env) \arrow[u, shift right, "\gamma_\times"'] \arrow[d, shift right, "\rho"'] \\
\rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) \arrow[u, shift right, "1"'] \arrow[d, shift right, "\alpha_\otimes"'] \\
\wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp \arrow[u, shift right, "\gamma_\otimes"']
\end{tikzcd}$$

-

analysis

Step 1: Projecting machine states

The first abstraction projects off a set of expressions and a set of environments:

$$\wp(C \times Env \times K) \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} \wp(C) \times \wp(C \times K) \times \wp(Env)$$

where

$$\alpha_\times(S) = \langle \pi_1 S, \{\langle \mathrm{s}, k\rangle \mid \exists e : \langle \mathrm{s}, e, k\rangle \in S\}, \pi_2 S\rangle$$
$$\gamma_\times(\langle C, F, E\rangle) = \{\langle \mathrm{s}, e, k\rangle \mid \mathrm{s} \in C \wedge \langle \mathrm{s}, k\rangle \in F \wedge e \in E\}$$

We calculate a new transition function using the first strategy:

$$\alpha_\times \circ F^c \circ \gamma_\times = \cdots = F^\times$$

Derivation outline

$$
\begin{tikzcd}
\wp(C \times Env \times K) \arrow[d, shift right, "\alpha_\times"'] & \text{collecting semantics} \\
\wp(C) \times \wp(C \times K) \times \wp(Env) \arrow[u, shift right, "\gamma_\times"'] \arrow[d, shift right, "\rho"'] & \text{-} \\
\rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) \arrow[u, shift right, "1"'] \arrow[d, shift right, "\alpha_\otimes"'] & \text{-} \\
\wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp \arrow[u, shift right, "\gamma_\otimes"'] & \text{analysis}
\end{tikzcd}
$$

Derivation outline

$$
\begin{tikzcd}
\wp(C \times Env \times K) \arrow[d, shift right, "\alpha_\times"'] \\
\wp(C) \times \wp(C \times K) \times \wp(Env) \arrow[u, shift right, "\gamma_\times"'] \arrow[d, shift right, "\rho"'] \\
\rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) \arrow[u, shift right, "1"'] \arrow[d, shift right, "\alpha_\otimes"'] \\
\wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp \arrow[u, shift right, "\gamma_\otimes"']
\end{tikzcd}
$$

collecting semantics

Ensure expr-stack relation and environment are global

-

analysis

Step 2: A closure operator on machine states

A closure operator ensures that

- all expr-stack pairs are part of the global set, and
- all sub-environments are part of the global environment.

However we first need two “sub-component” orderings:

- an order $\succ$ on environments
- an order $\gtrdot$ on expr-stack pairs

with $\succ^*$ and $\gtrdot^*$ being the reflexive transitive closures of the two.

Step 2: A closure operator on machine states

Now we can formulate the closure operator:

$$\wp(C)\times\wp(C \times K)\times\wp(Env) \underset{\rho}{\overset{1}{\leftrightarrows}} \rho(\wp(C)\times\wp(C \times K)\times\wp(Env))$$

where

$$\begin{aligned}\rho(\langle C,\, F,\, E\rangle) = \langle C, \{\langle \mathrm{s}\,,\, k\rangle \mid \exists\langle \mathrm{s}',\, k'\rangle \in F : \langle \mathrm{s}',\, k'\rangle \gtrdot^* \langle \mathrm{s}\,,\, k\rangle\},& \\ \{e \mid \exists\langle \mathrm{s}\,,\, k\rangle \in F : \langle \mathrm{s}\,,\, k\rangle \succ^* e \;\vee\; \exists e' \in E : e' \succ^* e\}\rangle&\end{aligned}$$

Again we calculate a new transition function using the first strategy:

$$\rho \circ F^{\times} \circ 1 = \cdots = F^{\rho}$$

Derivation outline

$$\begin{tikzcd}
\wp(C \times Env \times K) \arrow[d, shift right, "\alpha_\times"'] \\
\wp(C) \times \wp(C \times K) \times \wp(Env) \arrow[u, shift right, "\gamma_\times"'] \arrow[d, shift right, "\rho"'] \\
\rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) \arrow[u, shift right, "1"'] \arrow[d, shift right, "\alpha_\otimes"'] \\
\wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp \arrow[u, shift right, "\gamma_\otimes"']
\end{tikzcd}$$

collecting semantics

-

-

analysis

Derivation outline

$$\begin{array}{c}
\wp(C \times Env \times K) \\
\alpha_\times \Big\downarrow \Big\uparrow \gamma_\times \\
\wp(C) \times \wp(C \times K) \times \wp(Env) \\
\rho \Big\downarrow \Big\uparrow 1 \\
\rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) \\
\alpha_\otimes \Big\downarrow \Big\uparrow \gamma_\otimes \\
\wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp
\end{array}$$

collecting semantics

-

- ☐ Approximate closures by their lambda
- ☐ Approximate stacks by their top frame
- ☐ Merge expressions with same return point

Step 3a: Abstracting the expression-stack relation (1/2)

We abstract stacks to the top-of-stack:

$$K^\sharp \ni k^\sharp ::= \texttt{stop} \mid [\mathrm{x},\ \mathrm{s}] \qquad \text{(abstract stacks)}$$

using an elementwise operator:

$$\begin{aligned} @(\langle \mathrm{s},\ \texttt{stop}\rangle) &= \langle \mathrm{s},\ \texttt{stop}\rangle \\ @(\langle \mathrm{s},\ [\mathrm{x},\ \mathrm{s}',\ e] :: k\rangle) &= \langle \mathrm{s},\ [\mathrm{x},\ \mathrm{s}']\rangle \end{aligned}$$

which induces a Galois connection:

$$\wp(C \times K) \underset{\alpha_@}{\overset{\gamma_@}{\leftrightarrows}} \wp(C \times K^\sharp)$$

where $\alpha_@(F) = \{@(\langle \mathrm{s},\ k\rangle) \mid \langle \mathrm{s},\ k\rangle \in F\}$.

Step 3a: Abstracting the expression-stack relation (2/2)

Some expressions share their return points, e.g., `(let ((x t)) s)` and `s`, which induces an equivalence relation $\equiv$:

$$\texttt{(let ((x t)) s)} \equiv \texttt{s}$$
$$\texttt{(let ((x (t}_0\,\texttt{t}_1\texttt{))) s)} \equiv \texttt{s}$$

and another elementwise operator (and corresponding Galois connection):

$$@'(\langle \texttt{s}, k^\sharp \rangle) = \langle [\texttt{s}]_\equiv, k^\sharp \rangle$$

By composing the above with a *pointwise coding* we get:

$$\wp(C \times K) \underset{\alpha_{st}}{\overset{\gamma_{st}}{\leftrightarrows}} C/_\equiv \to \wp(K^\sharp)$$

Step 3b: Abstracting values and envs

We abstract values to abstract values

$$Val^\sharp \ni w^\sharp ::= c \mid \big[\texttt{(}\lambda\ \texttt{(x)}\ \texttt{s)}\big]$$

using yet another elementwise abstraction:

$$\begin{aligned} @(c) &= c \\ @([\texttt{(}\lambda\ \texttt{(x)}\ \texttt{s)}, e]) &= \big[\texttt{(}\lambda\ \texttt{(x)}\ \texttt{s)}\big] \end{aligned}$$

Based on the value abstraction, we can do a *pointwise abstraction of a set of functions*:

$$\wp(Env) \underset{\alpha_\Pi}{\overset{\gamma_\Pi}{\leftrightarrows}} Var \to \wp(Val^\sharp)$$

The third and final calculation

We calculate the final transition function using the second strategy:

$$\alpha_{\otimes} \circ F^{\rho} = \ldots \subseteq_{\otimes} F^{\sharp} \circ \alpha_{\otimes}$$

Note: this is not a complete abstraction

The resulting analysis

By the fixed-point transfer theorem the analysis of a program p is $\mathrm{lfp}\, F^\sharp_{\mathrm{p}}$, where

$$F^\sharp : P \to \wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp \to \wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp$$

$$F^\sharp_{\mathrm{p}}(\langle C,\, F^\sharp,\, E^\sharp\rangle) =$$

$$\langle\{\mathtt{p}\},\, [[\mathtt{p}]_\equiv \mapsto \{[\mathtt{x_r},\, \mathtt{x_r}]\}, [\mathtt{x_r}]_\equiv \mapsto \{\mathbf{stop}\}],\, \lambda\_.\,\emptyset\rangle$$

$$\cup_\otimes \bigcup_{\substack{\{\mathtt{t}\}\subseteq C \\ \{[\mathtt{x},\mathtt{s'}]\}\subseteq F^\sharp([\mathtt{t}]_\equiv)}}{}_{\otimes} \langle\{\mathtt{s'}\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\{\mathtt{(let\ ((x\ t))\ s)}\}\subseteq C}{}_{\otimes} \langle\{\mathtt{s}\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\substack{\{\mathtt{(t_0\ t_1)}\}\subseteq C \\ \{[\mathtt{(\lambda\ (x)\ s')}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}}{}_{\otimes} \langle\{\mathtt{s'}\},\, F^\sharp \,\dot\cup\, [[\mathtt{s'}]_\equiv \mapsto F^\sharp([\mathtt{(t_0\ t_1)}]_\equiv)],\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\substack{\{\mathtt{(let\ ((x\ (t_0\ t_1)))\ s)}\}\subseteq C \\ \{[\mathtt{(\lambda\ (y)\ s')}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}}{}_{\otimes} \langle\{\mathtt{s'}\},\, F^\sharp \,\dot\cup\, [[\mathtt{s'}]_\equiv \mapsto \{[\mathtt{x},\, \mathtt{s}]\}],\, E^\sharp \,\dot\cup\, [\mathtt{y} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle$$

Analysis characteristics

- We obtain a CFA with reachability (Ayers:WSA92, Palsberg-Schwartzbach:IAC95, Biswas:POPL97, Gasser-Nielson-Nielson:ICFP97, . . .)
- It predicts both calls *and returns* (in the presence of tail-call optimization!)
- Think of it as "*CFA by control-stack approximation*"

[Demo]

(fold your hands, please)

Outline

Recall the analysis

$$F^\sharp_{\mathtt{p}}(\langle C,\, F^\sharp,\, E^\sharp\rangle) =$$

$$\langle\{\mathtt{p}\},\, [[\mathtt{p}]_\equiv \mapsto \{[\mathtt{x_r},\, \mathtt{x_r}]\}, [\mathtt{x_r}]_\equiv \mapsto \{\mathbf{stop}\}],\, \lambda\_.\,\emptyset\rangle$$

$$\cup_\otimes \bigcup_{\otimes\, \substack{\{\mathtt{t}\}\subseteq C \\ \{[\mathtt{x},\,\mathtt{s'}]\}\subseteq F^\sharp([\mathtt{t}]_\equiv)}} \langle\{\mathtt{s'}\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\otimes\, \{\mathtt{(let\ ((x\ t))\ s)}\}\subseteq C} \langle\{\mathtt{s}\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\otimes\, \substack{\{\mathtt{(t_0\, t_1)}\}\subseteq C \\ \{[\mathtt{(\lambda\ (x)\ s')}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}} \langle\{\mathtt{s'}\},\, F^\sharp \,\dot\cup\, [[\mathtt{s'}]_\equiv \mapsto F^\sharp([\mathtt{(t_0\, t_1)}]_\equiv)],\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\otimes\, \substack{\{\mathtt{(let\ ((x\ (t_0\, t_1)))\ s)}\}\subseteq C \\ \{[\mathtt{(\lambda\ (y)\ s')}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}} \langle\{\mathtt{s'}\},\, F^\sharp \,\dot\cup\, [[\mathtt{s'}]_\equiv \mapsto \{[\mathtt{x},\, \mathtt{s}]\}],\, E^\sharp \,\dot\cup\, [\mathtt{y} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle$$

Recall the analysis and read off constraints

$F^\sharp_{\mathtt{p}}(\langle C,\, F^\sharp,\, E^\sharp\rangle) =$

$$\langle\{\mathtt{p}\},\, [[\mathtt{p}]_\equiv \mapsto \{[\mathtt{x_r},\, \mathtt{x_r}]\}, [\mathtt{x_r}]_\equiv \mapsto \{\mathtt{stop}\}],\, \lambda\_.\,\emptyset\rangle$$

For program $\mathtt{p}$:

$$\{\mathtt{p}\} \subseteq C \qquad \{[\mathtt{x_r},\, \mathtt{x_r}]\} \subseteq F^\sharp([\mathtt{p}]_\equiv) \qquad \{\mathtt{stop}\} \subseteq F^\sharp([\mathtt{x_r}]_\equiv)$$

$$\cup_\otimes \bigcup_{\substack{\otimes \\ \{\mathtt{(t_0\;t_1)}\}\subseteq C \\ \{[\mathtt{(\lambda\;(x)\;s')}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}} \langle\{\mathtt{s'}\},\, F^\sharp\,\dot\cup\,[[\mathtt{s'}]_\equiv \mapsto F^\sharp([\mathtt{(t_0\;t_1)}]_\equiv)],\, E^\sharp\,\dot\cup\,[\mathtt{x} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\substack{\otimes \\ \{\mathtt{(let\;((x\;(t_0\;t_1)))\;s)}\}\subseteq C \\ \{[\mathtt{(\lambda\;(y)\;s')}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}} \langle\{\mathtt{s'}\},\, F^\sharp\,\dot\cup\,[[\mathtt{s'}]_\equiv \mapsto \{[\mathtt{x},\, \mathtt{s}]\}],\, E^\sharp\,\dot\cup\,[\mathtt{y} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle$$

Recall the analysis and read off constraints

$$F^\sharp_{\mathtt{p}}(\langle C,\, F^\sharp,\, E^\sharp\rangle) =$$

$$\langle\{\mathtt{p}\},\, [[\mathtt{p}]_\equiv \mapsto \{[\mathtt{x_r},\, \mathtt{x_r}]\}, [\mathtt{x_r}]_\equiv \mapsto \{\mathtt{stop}\}],\, \lambda\_.\,\emptyset\rangle$$

$$\cup_\otimes \quad \bigcup_{\substack{\{\mathtt{t}\}\subseteq C \\ \{[\mathtt{x},\,\mathtt{s}']\}\subseteq F^\sharp([\mathtt{t}]_\equiv)}}{}_{\otimes} \quad \langle\{\mathtt{s}'\},\, F^\sharp,\, E^\sharp \mathbin{\dot\cup} [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t},E^\sharp)]\rangle$$

For each $\mathtt{t}$ and $\mathtt{(let\ ((x\ (t_0\, t_1)))\ s')}$ in $\mathtt{p}$:

$$\{\mathtt{t}\} \subseteq C \ \wedge\ \{[\mathtt{x},\, \mathtt{s}']\} \subseteq F^\sharp([\mathtt{t}]_\equiv) \Rightarrow \begin{cases} \{\mathtt{s}'\} \subseteq C \ \wedge \\ \mu_{sym}(\mathtt{t},E^\sharp) \subseteq E^\sharp(\mathtt{x}) \end{cases}$$

(where we partially evaluate the call to μ_{sym})

Recall the analysis and read off constraints

$$F^\sharp_{\mathrm{p}}(\langle C,\, F^\sharp,\, E^\sharp \rangle) =$$

$$\langle \{\mathtt{p}\},\, [[\mathtt{p}]_\equiv \mapsto \{[\mathtt{x_r},\, \mathtt{x_r}]\},\, [\mathtt{x_r}]_\equiv \mapsto \{\mathtt{stop}\}],\, \lambda\_.\, \emptyset \rangle$$

$$\cup_\otimes \bigcup_{\substack{\{\mathtt{t}\} \subseteq C \\ \{[\mathtt{x},\, \mathtt{s}']\} \subseteq F^\sharp([\mathtt{t}]_\equiv)}}^{\otimes} \langle \{\mathtt{s}'\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)] \rangle$$

$$\cup_\otimes \bigcup_{\{\mathtt{(let\ ((x\ t))\ s)}\} \subseteq C}^{\otimes} \langle \{\mathtt{s}\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)] \rangle$$

$\cup_\otimes$

$\{[(\lambda$ (

$\cup_\otimes$

$\{\mathtt{(let}$ (

$\{[(\lambda$ (

For each $\mathtt{(let\ ((x\ t))\ s)}$ in $\mathtt{p}$:

$$\{\mathtt{(let\ ((x\ t))\ s)}\} \subseteq C \Rightarrow \begin{cases} \{\mathtt{s}\} \subseteq C \ \wedge \\ \mu_{sym}(\mathtt{t}, E^\sharp) \subseteq E^\sharp(\mathtt{x}) \end{cases}$$

(where we partially evaluate the call to μ_{sym})

Recall the analysis and read off constraints

$F^\sharp_{\mathtt{p}}(\langle C,\, F^\sharp,\, E^\sharp\rangle) =$

$$\langle \{\mathtt{p}\},\ [[\mathtt{p}]_\equiv \mapsto \{[\mathtt{x_r},\, \mathtt{x_r}]\}, [\mathtt{x_r}]_\equiv \mapsto \{\mathtt{stop}\}],\ \lambda\_.\,\emptyset\rangle$$

$$\cup_\otimes \bigcup_{\substack{\{\mathtt{t}\}\subseteq C \\ \{[\mathtt{x},\, \mathtt{s}']\}\subseteq F^\sharp([\mathtt{t}]_\equiv)}}{}_{\otimes} \langle \{\mathtt{s}'\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle$$

$$\cup_\otimes \bigcup_{\{\mathtt{(let\ ((x\ t))\ s)}\}\subseteq C}{}_{\otimes} \langle \{\mathtt{s}\},\, F^\sharp,\, E^\sharp \,\dot\cup\, [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle$$

$\cup_\otimes$ … $E^\sharp)]\rangle$

$\{[(\lambda\ (\mathtt{x}$ …

Yeah yeah, we get the idea. . .

$$\cup_\otimes \bigcup_{\substack{\{\mathtt{(let\ ((x\ (t_0\ t_1)))\ s)}\}\subseteq C \\ \{[\mathtt{(}\lambda\ \mathtt{(y)\ s')}]\}\in\mu^\sharp(\mathtt{t_0}, E^\sharp)}}{}_{\otimes} \langle \{\mathtt{s}'\},\, F^\sharp \cup [[\mathtt{s}']_\equiv \mapsto \{[\mathtt{x},\, \mathtt{s}]\}],\, E^\sharp \cup [\mathtt{y} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle$$

Analysis equivalence

The resulting constraint-based CFA is equivalent:

Theorem:

A solution to the CFA constraints of $\mathtt{p}$ is a safe approximation of the least fixed point of the analysis function $F^\sharp$. Furthermore, the least solution to the CFA constraints is equal to the least fixed point of $F^\sharp$.

Outline

Deriving a CPS analysis from the CE-machine

For CPS terms it's the same story:

$$\begin{tikzcd}
\wp(SExp \times Env) \arrow[d, shift right, "\alpha_\times"'] & \text{collecting semantics} \\
\wp(SExp) \times \wp(Env) \arrow[u, shift right, "\gamma_\times"'] \arrow[d, shift right, "\alpha_\otimes"'] & \text{-} \\
\wp(SExp) \times Env^\sharp \arrow[u, shift right, "\gamma_\otimes"'] & \text{CPS CFA}
\end{tikzcd}$$

(previously derived in Midtgaard-Jensen:SAS08)

Deriving a CPS analysis from the CE-machine

For CPS terms it's the same story:

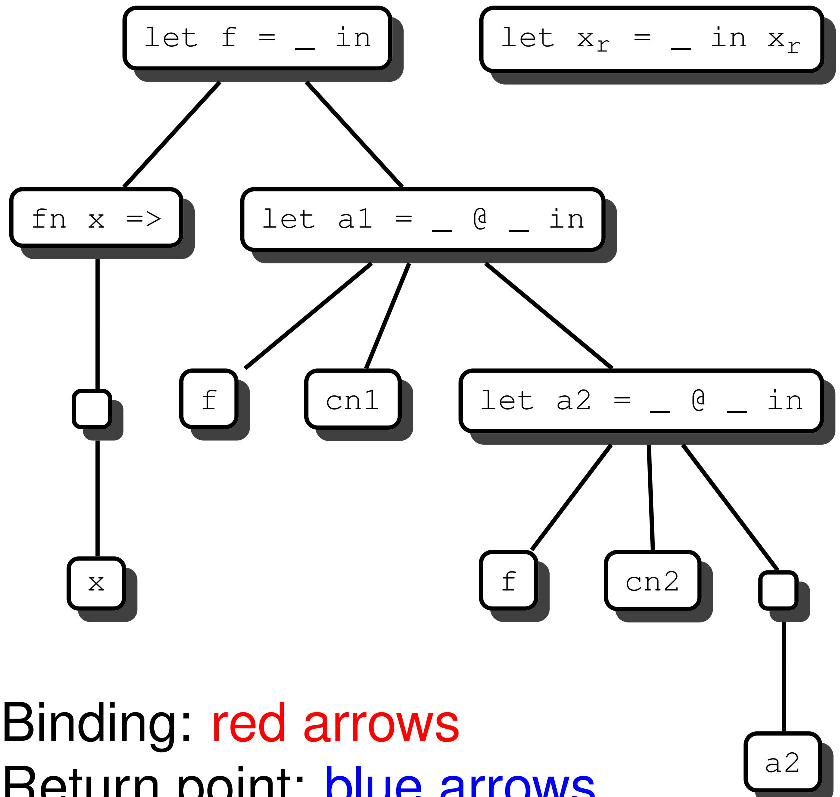
$$\begin{tikzcd}
\wp(SExp \times Env) \arrow[d, shift right=2, "\alpha_\times"'] & \text{collecting semantics} \\
\wp(SExp) \times \wp(Env) \arrow[u, shift right=2, "\gamma_\times"'] \arrow[d, shift right=2, "\alpha_\otimes"'] \arrow[d, bend left, "\alpha_\Pi \circ \rho"] & \text{-} \\
\wp(SExp) \times Env^\sharp \arrow[u, shift right=2, "\gamma_\otimes"'] & \text{CPS CFA}
\end{tikzcd}$$

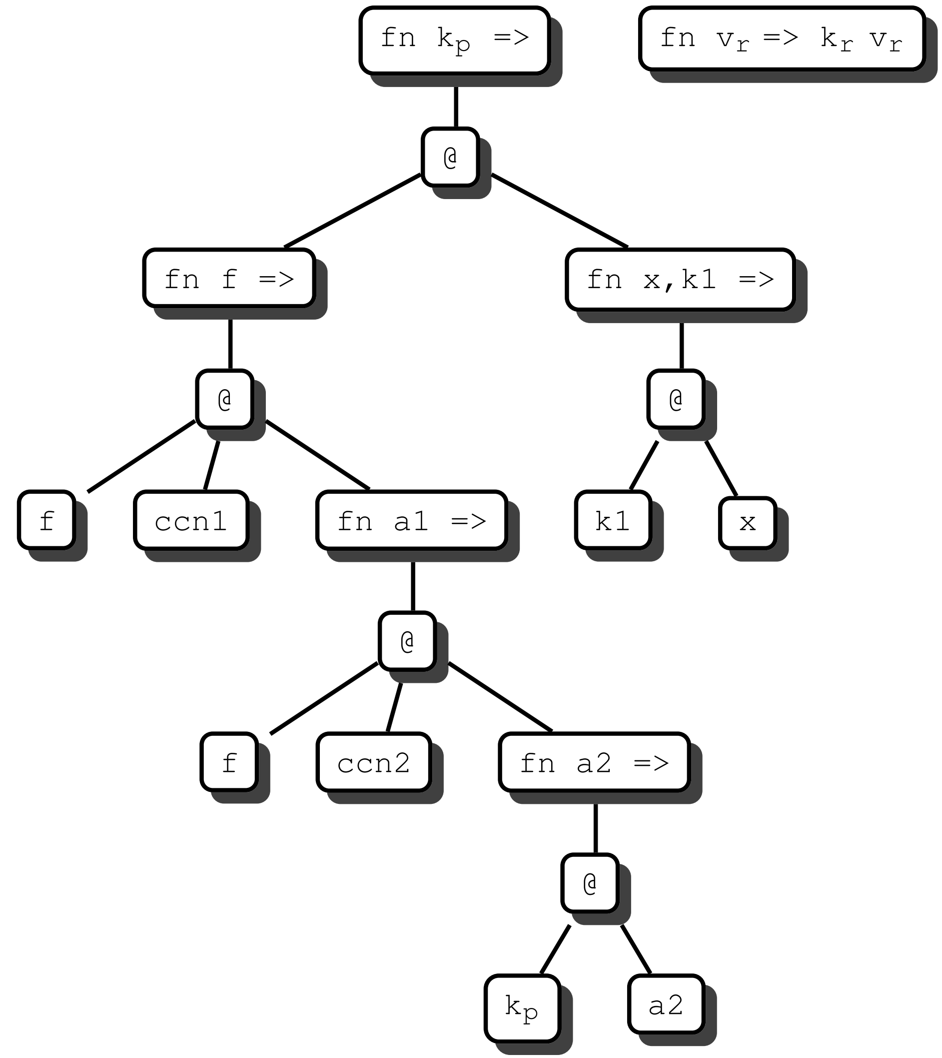
(previously derived in Midtgaard-Jensen:SAS08)

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```



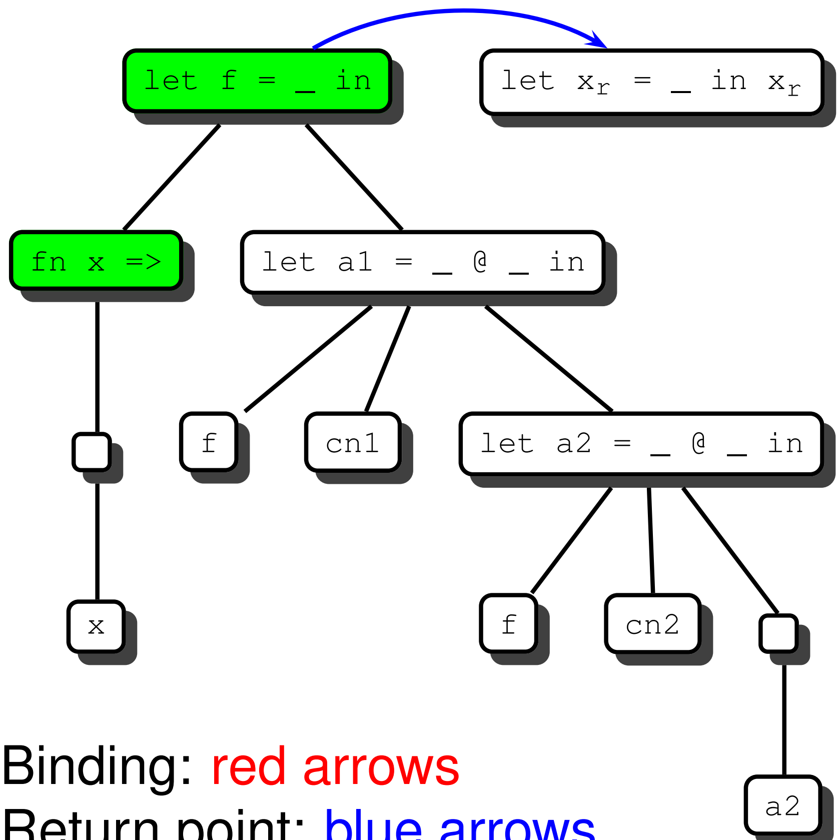


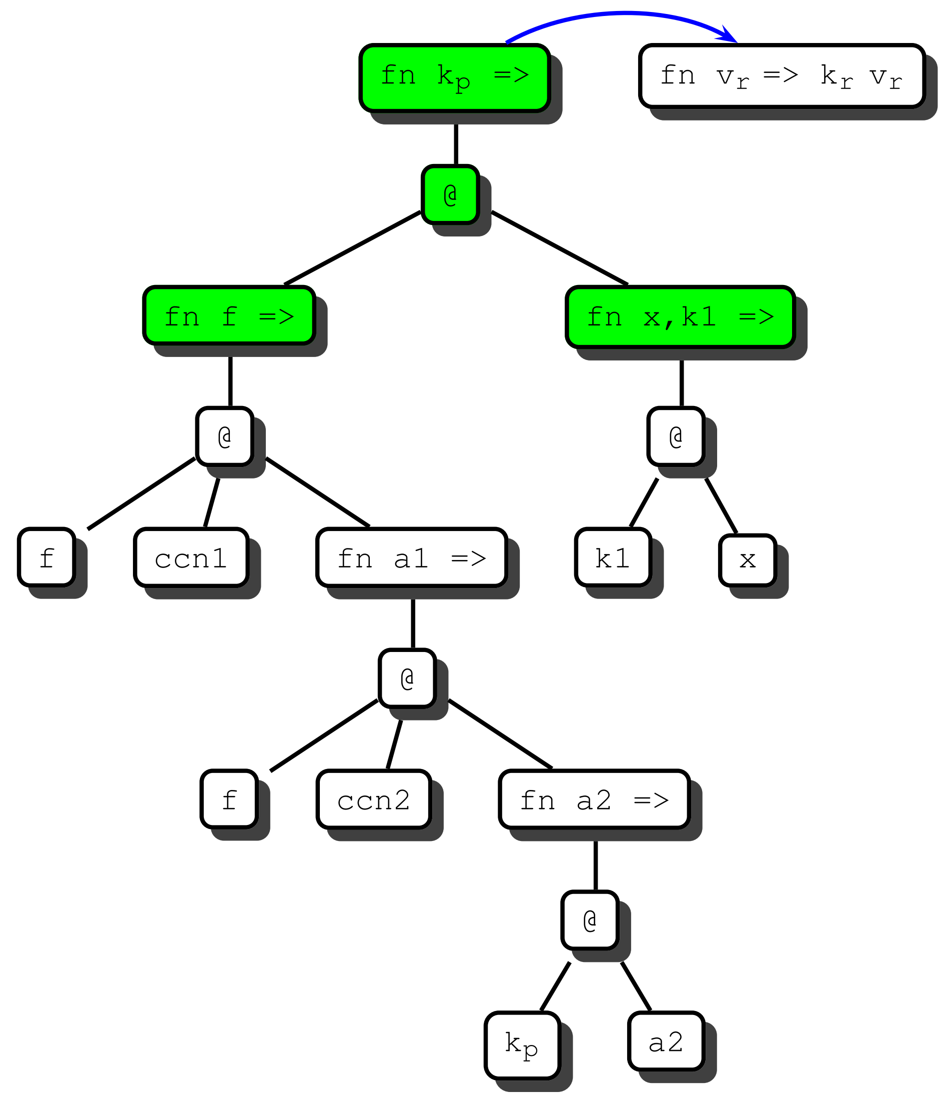
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```





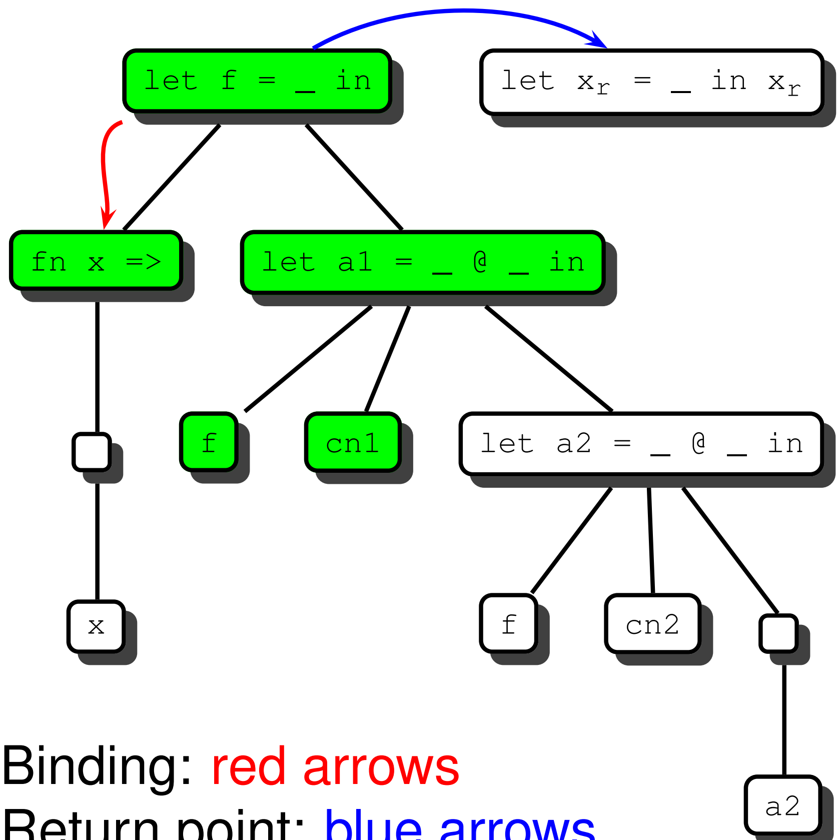
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

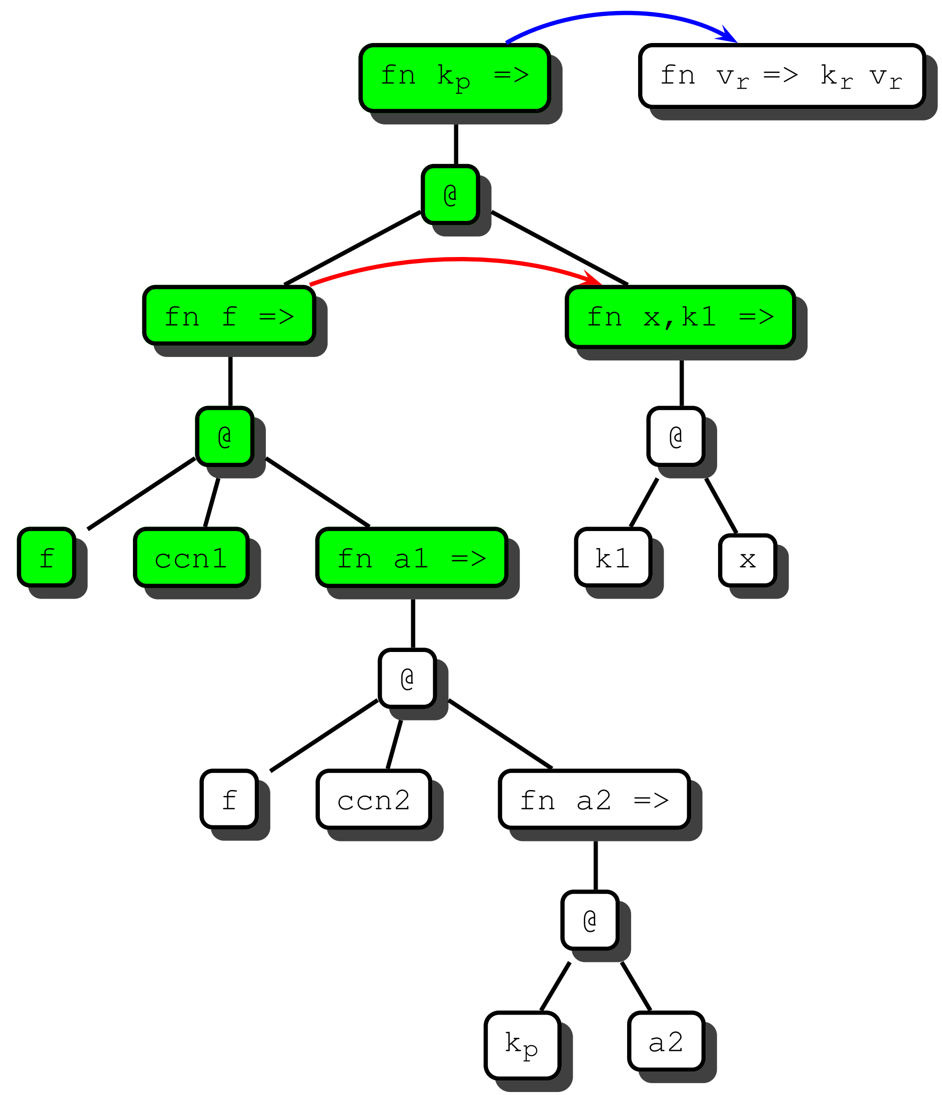
Iteration 1

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```





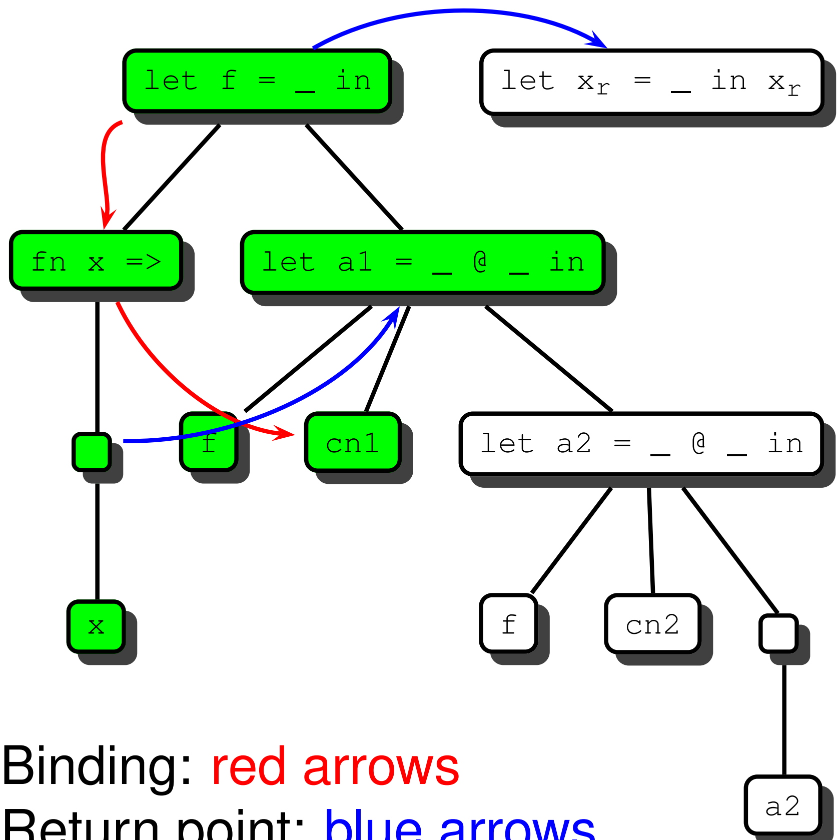
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

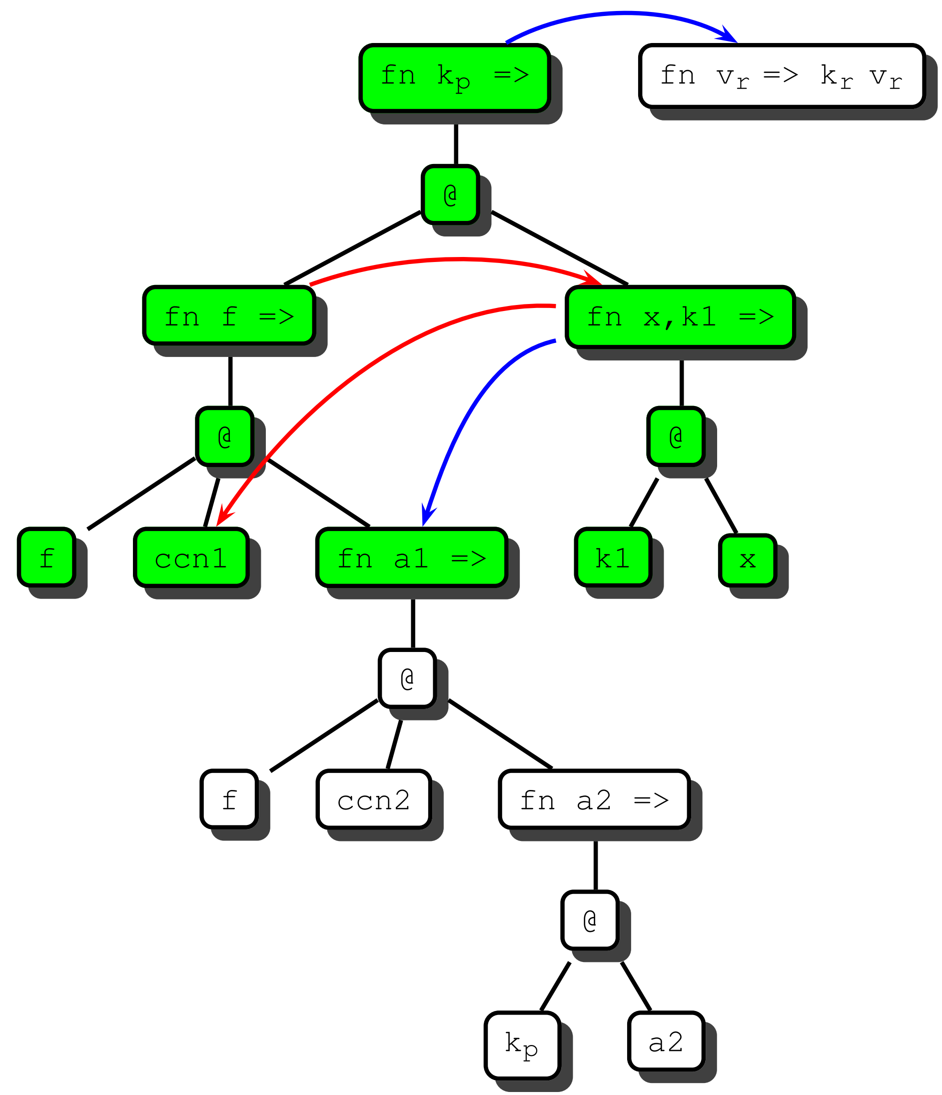
Iteration 2

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```





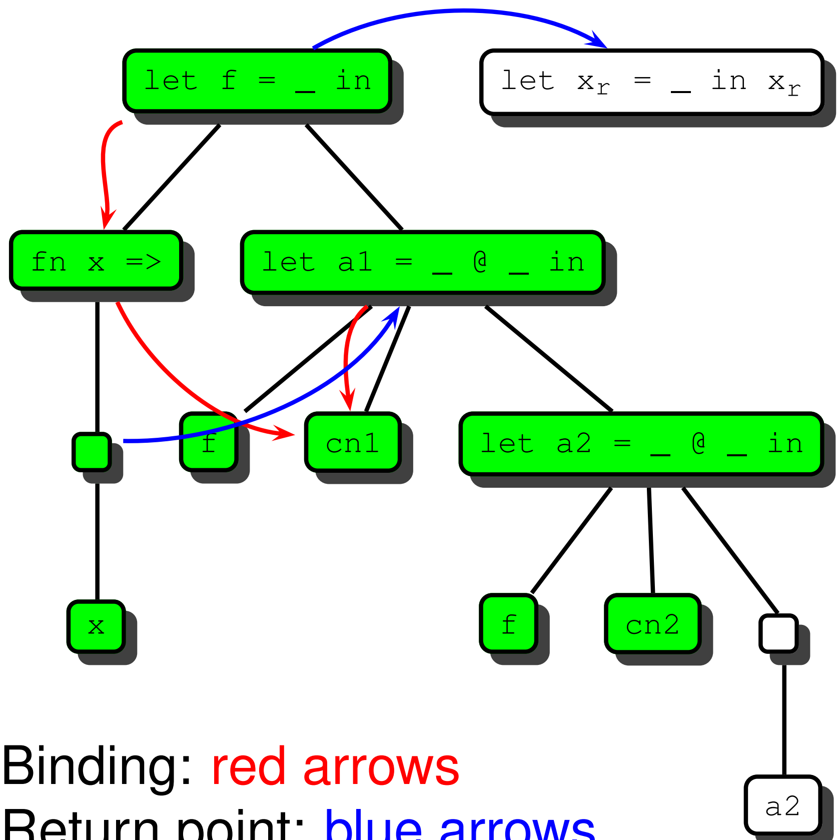
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

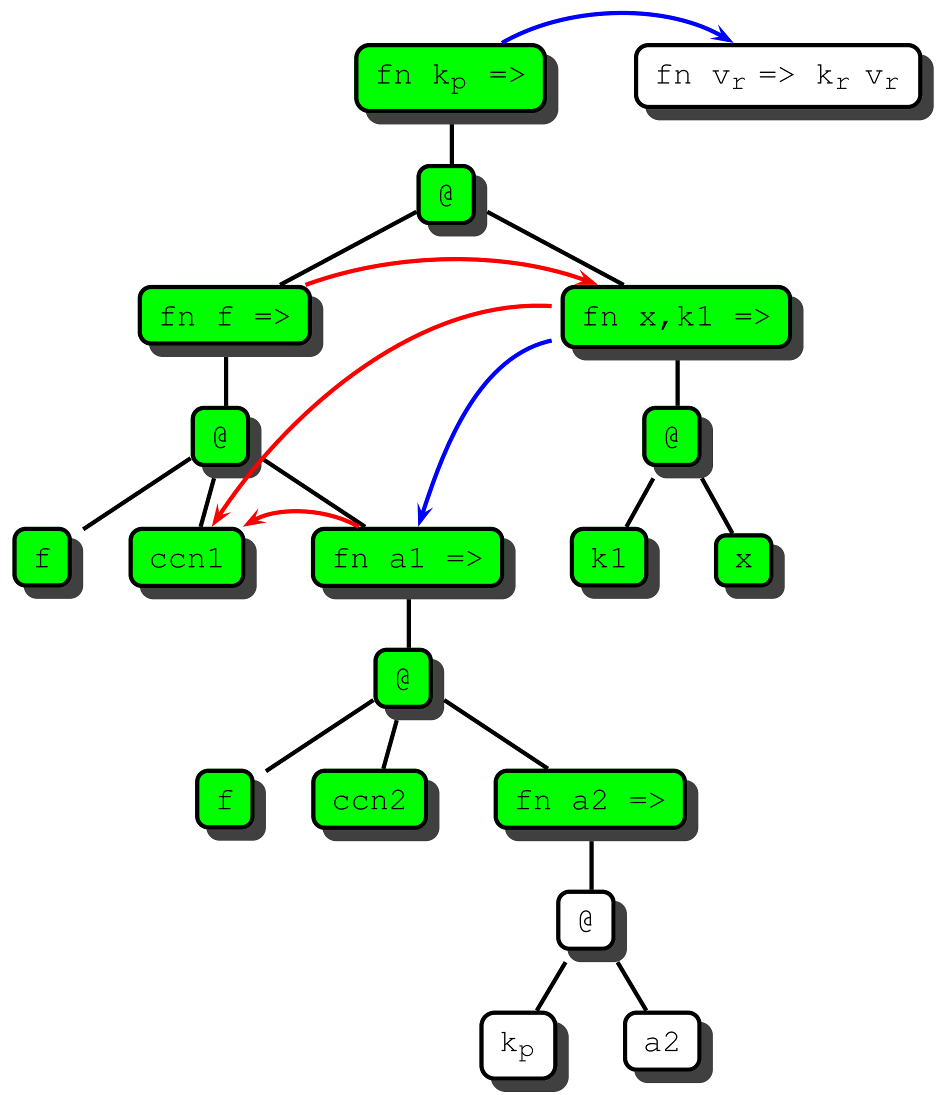
Iteration 3

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```





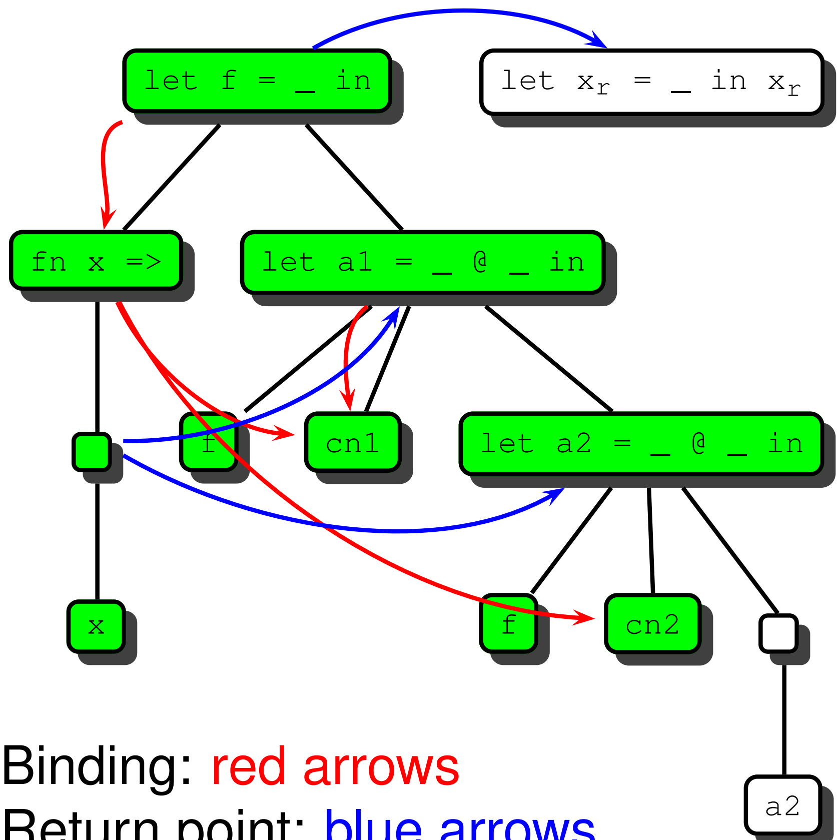
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

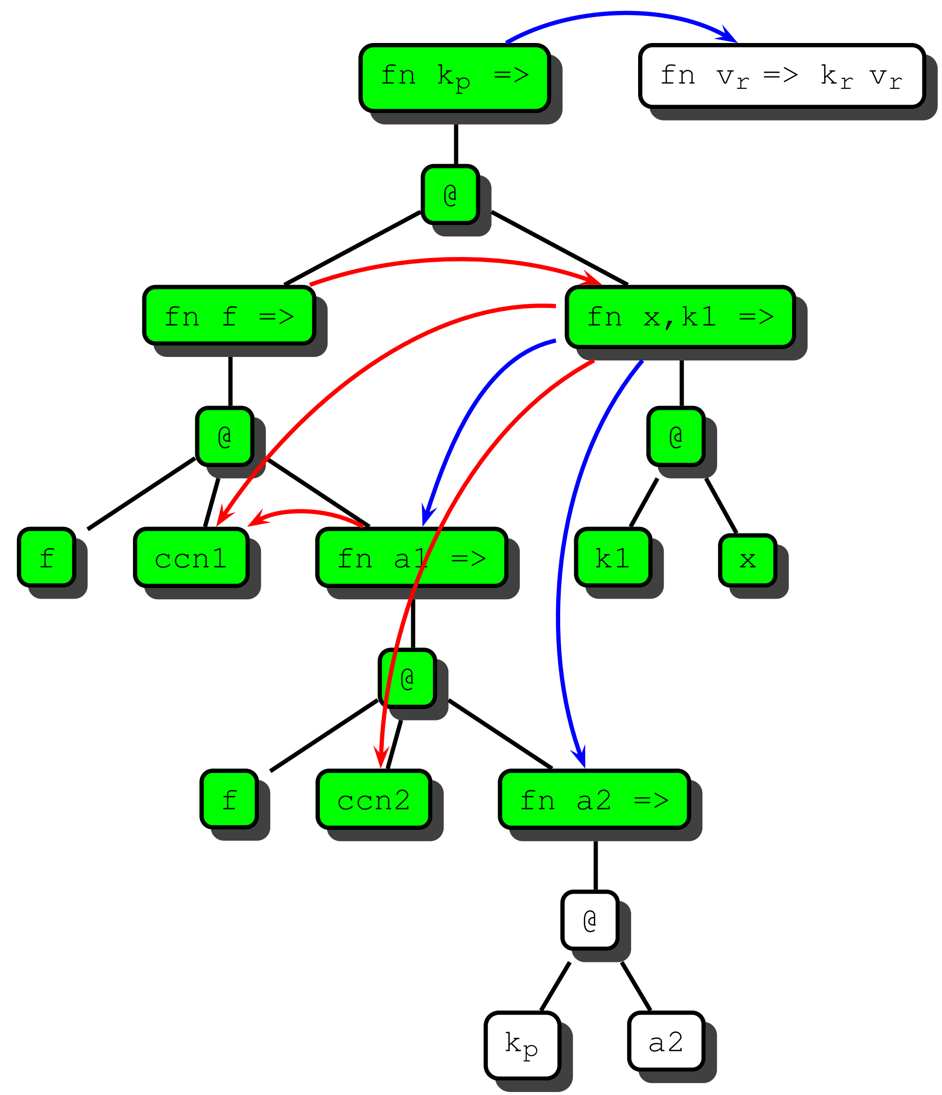
Iteration 4

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```





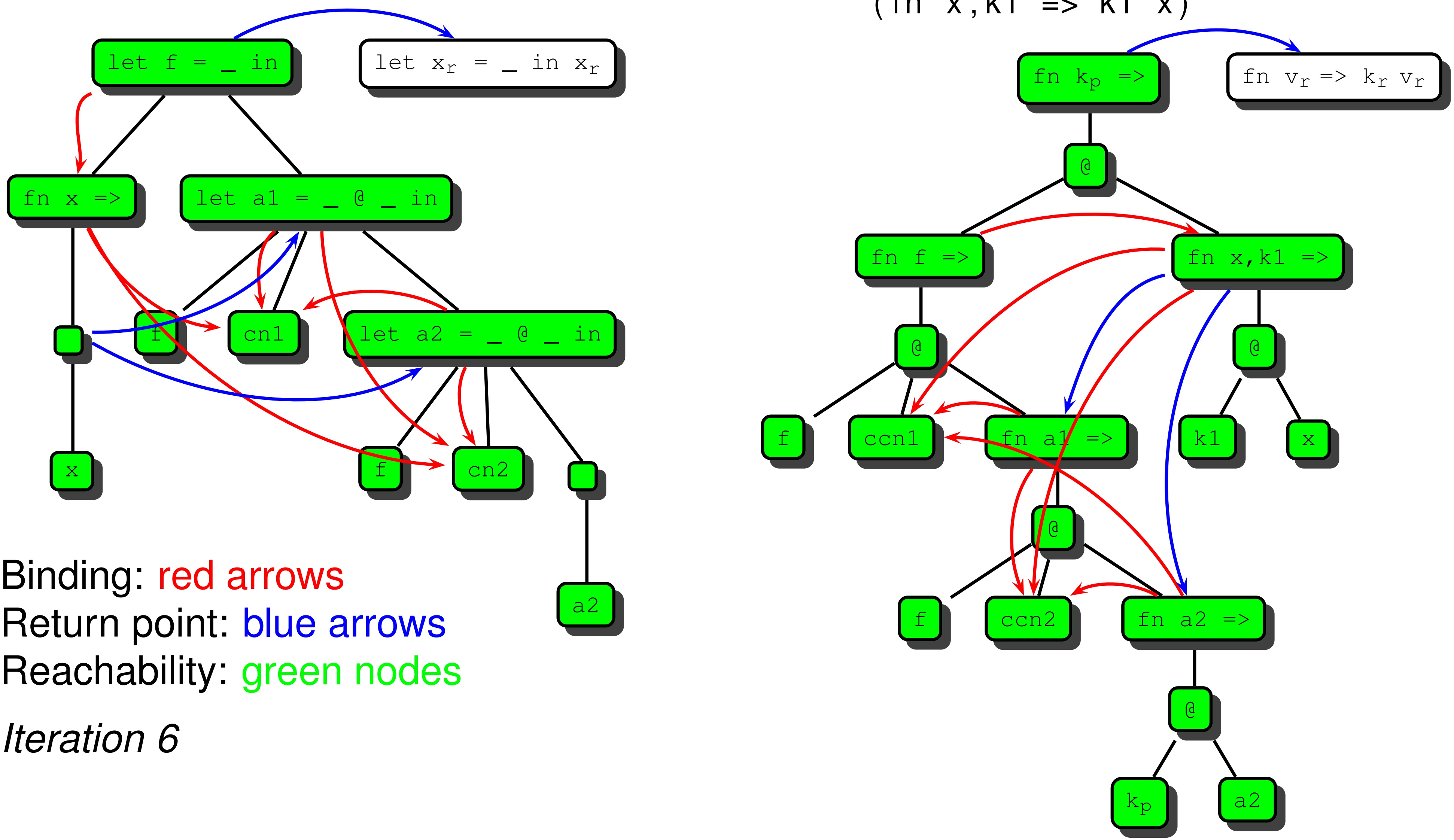
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

Iteration 5

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```



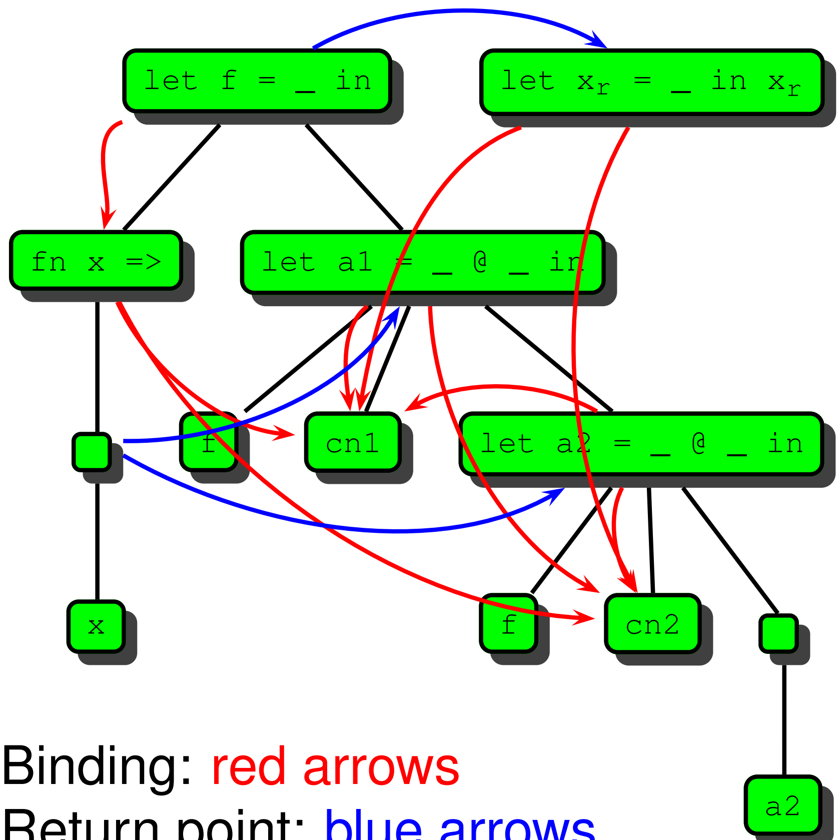
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

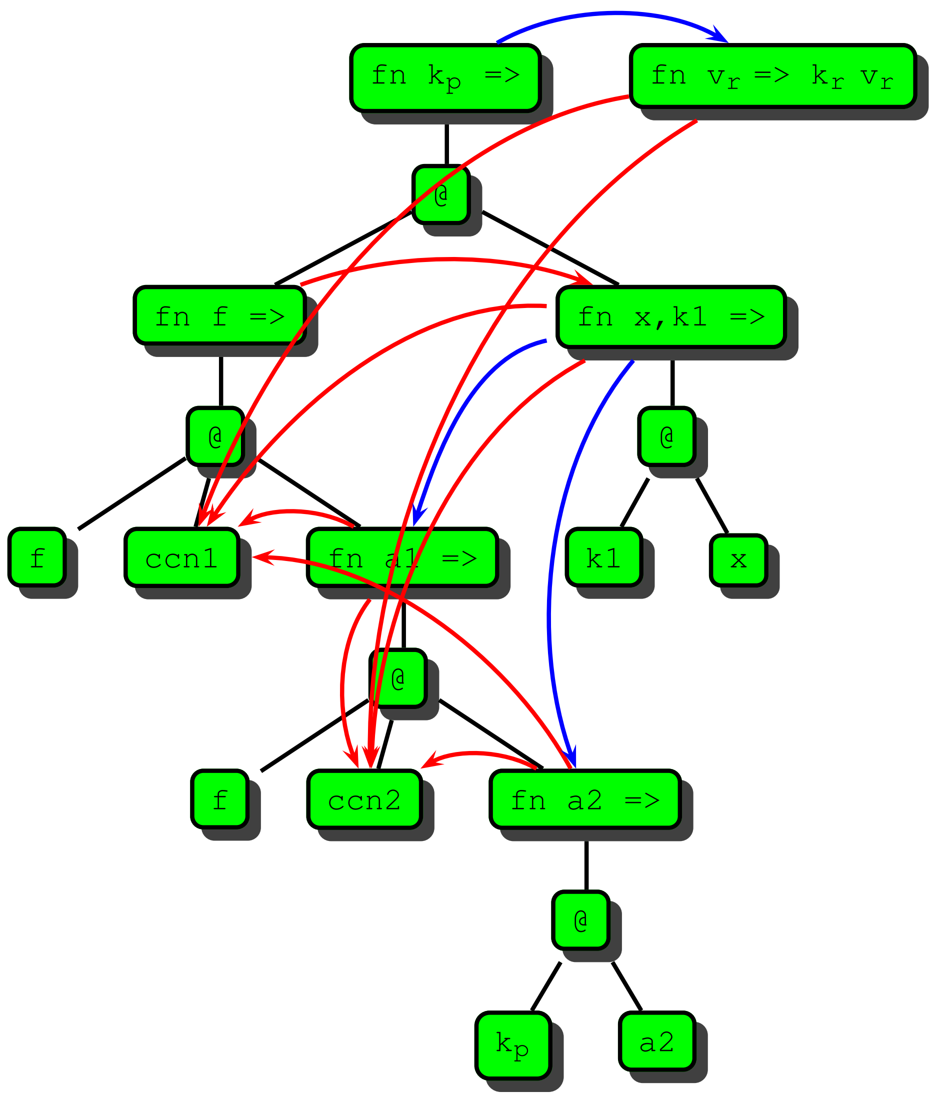
Iteration 6

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```





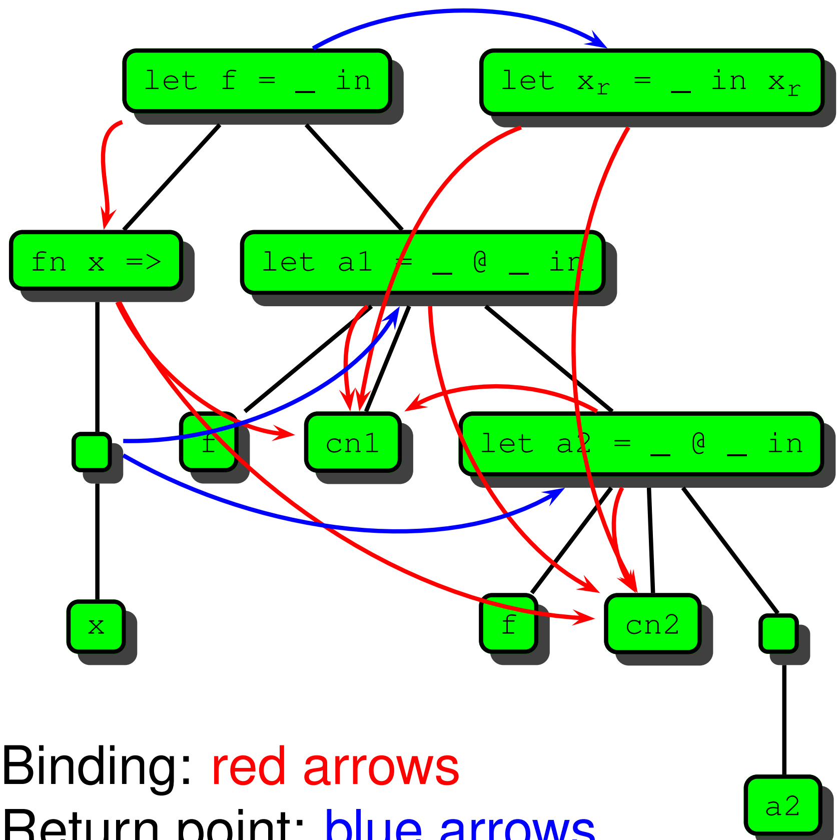
Binding: red arrows
Return point: blue arrows
Reachability: green nodes

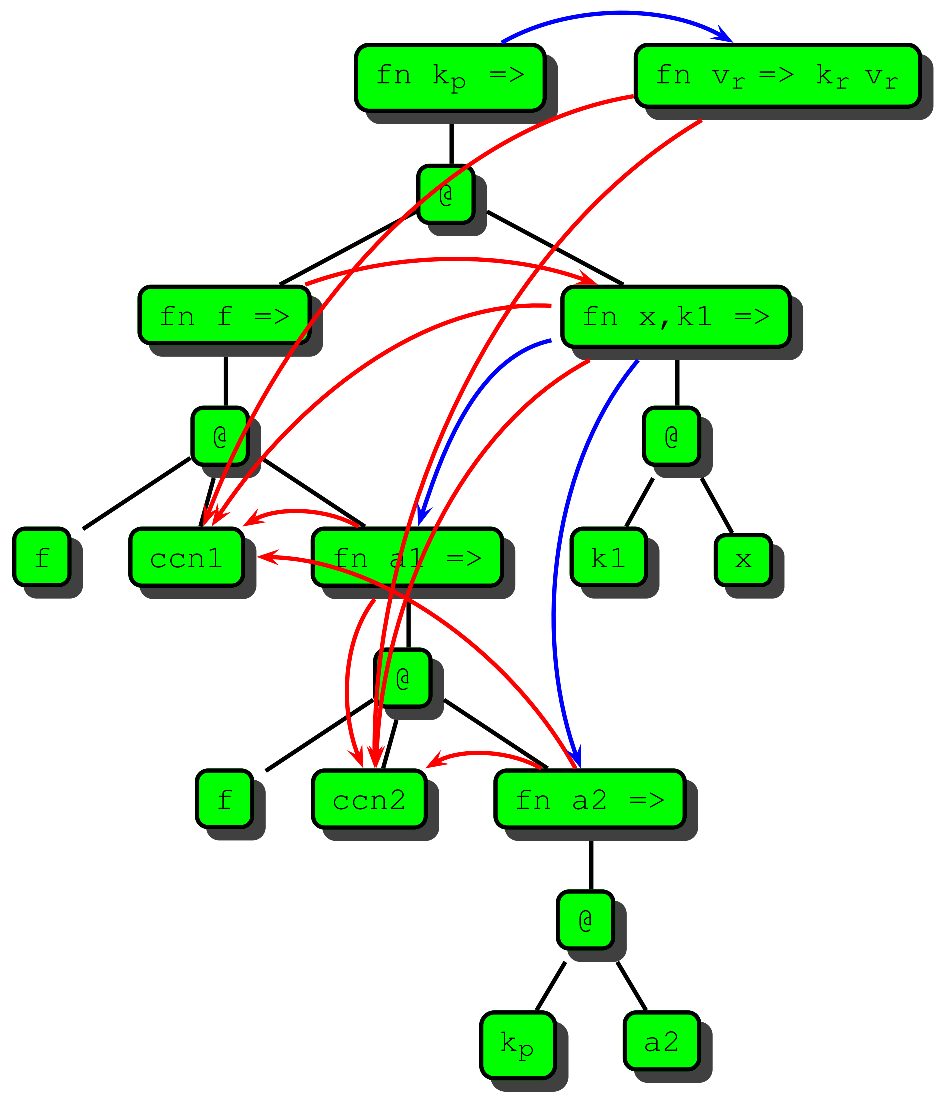
Iteration 7

Lock-step equivalence to CPS analysis

```
let f = (fn x => x) in
  let a1 = f cn1 in
    let a2 = f cn2 in a2
```

```
fn kp => (fn f =>
  f ccn1 (fn a1 =>
    f ccn2 (fn a2 => kp a2)))
      (fn x,k1 => k1 x)
```





Binding: red arrows
Return point: blue arrows
Reachability: green nodes

Iteration 8 - fixpoint

Analysis equivalence

The formal equivalence result relates:

ANF reachability	$\longleftrightarrow$	CPS reachability
ANF abstract stacks	$\longleftrightarrow$	CPS continuation closures
ANF closures	$\longleftrightarrow$	CPS function closures

(See paper for details)

Why should you care?

CFA has developed in two camps:

- □ Direct style CFA camp: Jones'81, Sestoft:FPCA89, Bondorf:SCP91, Palsberg:TOPLAS95, . . .
- □ CPS-based CFA camp: Shivers:PLDI88, Ayers:WSA92, Ashley-Dybvig:TOPLAS98, . . .

The resulting analyses are not necessarily comparable (Sabry-Felleisen:PLDI94, Mossin:PhD97, . . .)

Why should you care?

CFA has developed in two camps:

- □ Direct style CFA camp: Jones’81, Sestoft:FPCA89, Bondorf:SCP91, Palsberg:TOPLAS95, . . .
- □ CPS-based CFA camp: Shivers:PLDI88, Ayers:WSA92, Ashley-Dybvig:TOPLAS98, . . .

Conclusion

Summary and Conclusion

- Traditional abstract interpretation provides guidelines for disciplined analysis development
- It enabled us to derive a CFA predicting both calls and returns
- We have illustrated how to read off an equivalent constraint-based analysis
- Furthermore the resulting analysis is lock-step equivalent to a CPS analysis
- The top-down AI approach allows us to systematically break and preserve relations present in the collecting semantics

Abstract Debugging of Higher-Order Imperative Languages

Basic idea and context

Instead of ’dataflow analysis’ or ’program verification’, an analysis is used for ’abstract debugging’,

i.e. using abstract interpretation to locate the cause of bugs statically (without running the program!)

Achieved through a cool combination of forwards/backwards analysis

Basic idea and context

Instead of ’dataflow analysis’ or ’program verification’, an analysis is used for ’abstract debugging’,

i.e. using abstract interpretation to locate the cause of bugs statically (without running the program!)

Achieved through a cool combination of forwards/backwards analysis

Historic context: *"...applicable to languages such as Pascal, Modula-2, Modula-3, C or C++"*

The paper is from 1993 — Java wasn’t invented until 1995. All examples are given in Pascal

A crash course in Pascal (enough to parse examples)

- imperative programming language
  - types: integers, arrays, ...
  - statement-based: assignment (`:=`), `if`-`then`-`else`, loops (`while`, `for`, `repeat`-`until`)
- lexically scoped, variables are declared with `var`
- blocks are written `begin...end` (instead of `{...}`)
- `read(n)` reads input from stdin and assigns result to variable `n`
- `write(n)` outputs variable `n` to stdout

Pascal peculiarity 1

A function returns its result by "assigning it to the function's name":

```
function Fac(n: integer): integer;
begin
  if n = 0 then
     Fac := 1
  else
     Fac := n * Fac(n-1)
end;
```

Pascal peculiarity 2

Arrays are indexed as indicated by their declaration:

```
program For;
  var i, n : integer;
      T    : array [1..100] of integer;
  begin
      read(n);
      for i := 0 to n do
          read(T[i])
  end.
```

Problem 1: for `i=0` the statement `read(T[i])` indexes the array out-of-bounds

Problem 2: for this program, the input `n` also has to be < 101

Two types of assertions

The debugger is driven by two types of assertions:

Invariant assertions these are similar to normal `assert` statements: properties that must always hold at this point.

Example: `x > 0` at some program point

Intermittent assertions these are different: properties that eventually hold at this point

Example: `false` (i.e. bottom) at program exit (meaning end of program not reachable)

Properties in collecting semantics

Semantically, these properties can be expressed as a combination of forward/backward/lfp/gfp:

- □ Descendants of a set of states Σ (forward):

$$\mathrm{lfp}(\lambda X.\, \Sigma \cup post[\tau](X))$$

- □ Ascendants of a set of states Σ (backward):

$$\mathrm{lfp}(\lambda X.\, \Sigma \cup pre[\tau](X))$$

- □ Ascendants not leading to error in S_{err} (backward):

$$\mathrm{gfp}(\lambda X.\, pre[\tau](X) \backslash S_{err})$$

Assertion properties, more generally

For a property $\Pi \in \wp(S)$, that will eventually hold:

$$\textbf{eventually}(\Pi) = \mathrm{lfp}(\lambda X.\, \Pi \cup \mathit{pre}[\tau](X))$$

with the corresponding Kleene sequence:

$$\textbf{eventually}(\Pi) = \Pi \cup \mathit{pre}[\tau](\Pi) \cup \mathit{pre}^2[\tau](\Pi) \cup \ldots$$

For a property $\Pi \in \wp(S)$, that must always hold:

$$\textbf{always}(\Pi) = \mathrm{gfp}(\lambda X.\, \Pi \cap \mathit{pre}[\tau](X))$$

with the corresponding Kleene sequence:

$$\textbf{always}(\Pi) = \Pi \cap \mathit{pre}[\tau](\Pi) \cap \mathit{pre}^2[\tau](\Pi) \cap \ldots$$

Assertions as always/eventually properties

Programs are modeled using $PC \times Memory$ pairs.

Property π_k always holds at point c_k (for all $k \in K_a$)

$$\Pi_a = \{\langle c, m\rangle \in S \mid \forall k \in K_a : c = c_k \implies m \in \pi_k\}$$

(invariant ass.)

at all other points c, the memory m is true (anything)

Property π_k eventually holds at point c_k (for some $k \in K_e$)

$$\Pi_e = \{\langle c, m\rangle \in S \mid \exists k \in K_e : c = c_k \ \wedge \ m \in \pi_k\}$$

(intermittent ass.)

at all other points c, the memory m is false (non-existing)

Fixed point computation, (coll.) semantically

Semantically we seek the limit I of the sequence

$$S = I_0 \supseteq I_1 \supseteq I_2 \supseteq I_3 \supseteq \dots$$

where

- $I_{k+1} = \mathrm{lfp}(\lambda X.\, I_k \cap (S_{in} \cup post[\tau](X)))$
- $I_{k+2} = \mathrm{gfp}(\lambda X.\, I_{k+1} \cap \Pi_a \cap pre[\tau](X))$
- $I_{k+3} = \mathrm{lfp}(\lambda X.\, I_{k+2} \cap (\Pi_e \cup pre[\tau](X)))$

The fixed point computation continues to propagate forwards ($k+1$), backwards ($k+2$), backwards ($k+3$)

Error detection from fixed point result

- All $s \in S_{in} \backslash I$ break one of the programmer's invariants, since s is not in Π_a or will not lead to a state in Π_e.
- All $s \in post^*[\tau](I) \backslash I$ also break an invariant, since s follows from the forwards flow from I, but not from the backwards flow.

Hence such states can be reported to the programmer.

From fixed point semantics to analysis

The analysis is similar, except it performs fixed point computations over an abstract domain.

The analysis and semantics are (of course) connected by Galois connections.

It is expressed as forward and backward "semantic equations".

These equations are similar to the IMP semantics from week 2

(and to the constraints we extracted last week).

Forward equation example

$$
\begin{array}{ll}
\color{blue}{0:} & x_0 = \top \\
\color{blue}{1:}\ \texttt{read(i);} & x_1 = [\![\texttt{read(i)}]\!](x_0) \\
\color{blue}{2:}\ \texttt{while (i} \leq \texttt{100) do} & x_2 = [\![i \leq 100]\!](x_1) \sqcup [\![i \leq 100]\!](x_3) \\
\color{blue}{3:}\ \quad\quad \texttt{i} := \texttt{i} + \texttt{1} & x_3 = [\![i := i + 1]\!](x_2) \\
\color{blue}{4:} & x_4 = [\![i > 100]\!](x_1) \sqcup [\![i > 100]\!](x_3)
\end{array}
$$

where

- $[\![-]\!]$ abstract the primitive operations, and
- the x_i's represent an abstract memory per program point

Backward intermittent equation example

$$
\begin{array}{lll}
0: & & x_0 = [\![\texttt{read(i)}]\!]^{-1}(x_1) \\
1: & \texttt{read(i);} & x_1 = [\![i \le 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4) \\
2: & \texttt{while (i} \le \texttt{100) do} & x_2 = \alpha(\{10\}) \sqcup [\![i := i + 1]\!]^{-1}(x_3) \\
3: & \qquad \texttt{i := i + 1} & x_3 = [\![i \le 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4) \\
4: & & x_4 = x_4
\end{array}
$$

where

- the intermittent assertion `i = 10` has been inserted (to mimic join with Π_e), and
- $[\![-]\!]^{-1}$ abstract the backwards primitive operations.

Backward invariant equation example

$$
\begin{array}{ll}
0: & x_0 = [\![\texttt{read(i)}]\!]^{-1}(x_1) \\
1: \texttt{read(i);} & x_1 = [\![i \leq 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4) \\
2: \texttt{while (i} \leq \texttt{100) do} & x_2 = \alpha(\{0, 1, 2, \ldots\}) \sqcap [\![i := i + 1]\!]^{-1}(x_3) \\
3: \quad\quad \texttt{i} := \texttt{i} + \texttt{1} & x_3 = [\![i \leq 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4) \\
4: & x_4 = x_4
\end{array}
$$

where

- the invariant assertion $\texttt{i} \geq \texttt{0}$ has been inserted (to mimic meet with Π_a), and
- $[\![-]\!]^{-1}$ abstract the backwards primitive operations.

Minimal use of widening

To speed up convergence or guarantee termination the analysis uses widening/narrowing operators.

Widening (and narrowing) represent information loss, so we want to minimize the number of widenings.

Only loops (cycles) can lead to infinite chains in the analysis.

Convergence is guaranteed by at least one widening operator per cycle in the equation dependency graph.

Forward equations with widening

```
read(i);
while (i ≤ 100) do
    i := i + 1
```

$$
\begin{aligned}
x_0 &= \top \\
x_1 &= [\![\texttt{read(i)}]\!](x_0) \\
x_2 &= x_2 \,\nabla\, ([\![i \le 100]\!](x_1) \sqcup [\![i \le 100]\!](x_3)) \\
x_3 &= [\![i := i + 1]\!](x_2) \\
x_4 &= [\![i > 100]\!](x_1) \sqcup [\![i > 100]\!](x_3)
\end{aligned}
$$

where

- the widening operator breaks the x_2–x_3–x_2 dependency cycle of the above equations

Interval analysis

The analysis prototype uses an interval lattice that correctly models underflow/overflow:

$$l, u \in [-2^{b-1}; 2^{b-1} - 1]$$

of finite height 2^b. However Bourdoncle still uses widening to speed up convergence.

For strictly increasing upper bounds, interval widening jumps to top ($2^{b-1} - 1$)

and for strictly decreasing lower bounds, interval widening jumps to bottom (-2^{b-1})

Hence the resulting analysis converges in at most 4 iterations

Analysis complexity

One can simply solve the equations by Kleene fixed point iteration.

However there are more clever approaches based on chaotic iteration.

Bourdoncle combines two strategies:

- First compute intraprocedural fixed points, based on the dependency graph,
- then compute interprocedural fixed points, based on the call graph

The resulting algorithm is quadratic in the program size (assuming the number of variables is constant).

Prototype

The prototype implementation consists of

- approx. 20000 lines of C
- incl. 4000 lines of X-window GUI

It first extracts semantic equations, which are subsequently solved.

The prototype is configurable. By default it performs

- a forward analysis,
- two backward analyses, and
- a final forward analysis

McCarthy's 91 function

Bourdoncle analyses (a generalization of) the following benchmark program:

$$MC(n) = \begin{cases} n - 10 & \text{if } \; n > 100 \\ MC(MC(n+11)) & \text{if } \quad \leq 100 \end{cases}$$

which is functionally equivalent to:

$$MC(n) = \begin{cases} n - 10 & \text{if } \; n > 100 \\ 91 & \text{if } \quad \leq 100 \end{cases}$$

It is interesting for static analysis, because the constant 91 does not appear anywhere in the source text.

McCarthy's 91 function, generalized

Bourdoncle analyses the following generalized benchmark program:

$$MC_k(n) = \begin{cases} n - 10 & \text{if } \ n > 100 \\ {MC_k}^k(n + 10k - 9) & \text{if } \ \le 100 \end{cases}$$

which is still functionally equivalent to:

$$MC_k(n) = \begin{cases} n - 10 & \text{if } \ n > 100 \\ 91 & \text{if } \ \le 100 \end{cases}$$

But now MC_k contains k recursive calls.

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (invariant) assert $\mathtt{n} \leq 101$ here,

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
                 MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (invariant) assert $\texttt{n} \leq 101$ here,

the analysis proves $\texttt{m} = 91$ here

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (intermittent) assert $\mathtt{m} = 91$ here,

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

the analysis finds that $\mathtt{n} \leq 101$ is a necessary condition here

If we (intermittent) assert $\mathtt{m} = 91$ here,

MC_9 in Pascal, buggy

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 71)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

MC_9 in Pascal, buggy

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 71)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (intermittent) assert $true$ here,

MC_9 in Pascal, buggy

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 71)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

the analysis finds that `n` ≥ 101 is a necessary termination condition here

If we (intermittent) assert *true* here,

Syntox tool for download

Bourdoncle keeps a binary executable for download:

`http://web.me.com/fbourdoncle/page18/page6/page6.html`

It is however restricted to

- sparc (Suns),
- solaris (Sun + Solaris) or
- mips (MIPS/Ultrix DECStation)

Let me know if you find a machine (or an emulator) able to run it.

Summary and conclusion

A very nice application of abstract interpretation machinery.

Overall the basic techniques are very well presented.

Hence they are directly applicable to an "abstract 3CM debugger" (which would be a very cool project).

For more complex features (reference parameters with aliasing, recursive function calls, ...) more details are swept under the rug.

Summary

Summary

Two case studies based on research articles:

- *Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation*, Midtgaard and Jensen, ICFP'09
- *Abstract Debugging of Higher-Order Imperative Languages*, Bourdoncle, PLDI'93