The Complexity of $k$CFA

David Van Horn
& Harry Mairson
Van Horn and Mairson, ICFP’07, ICFP’08, SAS’08
0CFA and PTIME
What is in the intersection?
CHAPTER 4. LINEAR LOGIC AND STATIC ANALYSIS

4.1 Sharing Graphs for Static Analysis

In general, the sharing graph of a term will consist of a distinguished root wire from which the rest of the term's graph "hangs." Graphs consist of ternary abstraction (\(\text{term} \rightarrow \text{term} \rightarrow \text{term}\)), apply (@), sharing (O) nodes, and unary weakening (\(\text{term} \rightarrow \text{term}\)) nodes. Each node has a distinguished principal port. For unary nodes, this is the only port. The ternary nodes have two auxiliary ports, distinguished as the white and black ports.

- A variable occurrence is represented simply as a wire from the root to the free occurrence of the variable.

\[
\text{fv}(M) \cap \{x\}
\]

\(M\)
CHAPTER 4. LINEAR LOGIC AND STATIC ANALYSIS

4.1 Sharing Graphs for Static Analysis

In general, the sharing graph of a term will consist of a distinguished root wire from which the rest of the term's graph "hangs." At the bottom of the graph, the dangling wires represent free variables and connect to occurrences of the free variable within in term.

Graphs consist of ternary abstraction, apply (@), sharing (Ω) nodes, and unary weakening (−) nodes. Each node has a distinguished principal port. For unary nodes, this is the only port. The ternary nodes have two auxiliary ports, distinguished as the white and black ports.

A variable occurrence is represented simply as a wire from the root to the free occurrence of the variable.

\[ M^{fv}(M) \cap \{x\} \]
the abstraction $x.M$ is formed as, $\lambda \cdot fv(M)$ \ $\cap \{x\}$

Supposing $x$ does not occur in $M$, the weakening node $(\cdot)$ is used to "plug" the variable wire.

Given graphs for $M$ and $N$, $\lambda \cdot (fv(M) \ \cap \ {x})$ $\lambda \cdot (fv(M) \ \cap \ {x})$
the abstraction $x.M$ is formed as, $x.M$, $x^2 fv(M)$

Supposing $x$ does not occur in $M$, the weakening node $(\_ \_ \_)$ is used to “plug” the variable wire.

Given graphs for $M$ and $N$, $M N fv(M) fv(N)$,
Figure 4.1: CFA virtual wire propagation rules.

The application $MN$ is formed as,

$$<M@N> \cap \text{fv}(M) \cap \text{fv}(N) \setminus \text{fv}(M) \cap \text{fv}(N) \setminus \text{fv}(M)$$

An application node is introduced. The operator $M$ is connected to the function port and the operand $N$ is connected to the argument port. The continuation wire becomes the root wire for the application. Free variables shared between both $M$ and $N$ are fanned out with sharing nodes.

4.2 Graphical OCF

We now describe an algorithm for performing control flow analysis that is based on the graph coding of terms. The graphical formulation consists of generating a set of virtual paths for a program graph. Virtual paths describe an approximation of the real paths that will arise during program execution.

Figure 4.1 defines the virtual path propagation rules. Note that a wire can be
Algorithm constructs answers that satisfy the acceptability relation specifying the analysis. Moreover, this algorithm constructs least solutions according to the partial order given in section 2.3.

Lemma 5. \[ \beta C_0, \hat{r}_0 | e \] implies \[ \beta C, \hat{r}_v \] for \[ \beta C, \hat{r} \] constructed for \[ e \] as described above.

We now consider an example of use of the algorithm. Consider the labeled program:

\[
(f.f f (y.y)) (x.x)
\]

Figure 4.2 shows the graph coding of the program and the corresponding CFA graph. The CFA graph is constructed by adding virtual wires 10 and \( f 9 \), induced by the actual -redex on wire 7. Adding the virtual path \( f 9 \) to the graph creates a virtual -redex via the route 1 \( f \) (through the sharing node), and \( f 9 \) (through the virtual wire). This induces \( 3 8 \) and \( 8 2 \). There is now a virtual -redex via \( 3 8 \), so wires 6 and 8 are added. This addition creates another virtual redex via \( 3 8 2 5 \), which induces virtual wires 6 4 and 4 5. No further wires can be added, so the...
Similarly, the textual information can be used to distinguish the copies and give restricted to those occurring in a no—makes the analysis insensitive to the output size of any redex from reductions to take place. "to what values can a subexpression evaluate?"

\[ \lambda x . \lambda x \]

\[ \Rightarrow \text{cfa} \]
defined according to the following clauses:

A more precise answers to these questions.

Figure 2 defines the virtual path propagation rules. The left hand rule states that a virtual wire is added from the continuation wire to the apply and lambda node connected by an apply and lambda node connected by a virtual path between a virtual path and a lambda. An acceptable control flow analysis for an expression is based on the graph coding of terms. The graphical for-
The ability relation is given by the greatest fixed point of the functional
bound to that variable during evaluation.

Similarly, the value, which represents the set of (textual) values that may be
reached from the principal port of a sharing node, or 4) written
to the body wire and from the variable wire to the argument wire of
a virtual path. There is a virtual path in the CFA graph. The CFA graph is constructed by adding vir-
ual wires: 1) if a redex, the right hand rule states analogous wires are added
to the continuation wire to
2) from the virtual path from a reachable apply node.

We follow Nielson et al. (1999) and say the result of 0CFA is
the set of labels—i.e., a program point. More precisely:

Given expressions x, a redex. The right hand rule states analogous wires are added

The graphical analogue of this problem point to a
—node and

A decision problem—a question that can be answered with a yes
or a no—makes the analysis insensitive to the output size of any

The correspondence between 0CFA and

Experimenting with CFA and abstract caches

control flow analysis. After describing the problem here, subse-
described above.

Lemma 1.
Every variable occurs once.
The ability relation is given by the greatest fixed point of the functional equation:

\[ e^\lambda \]

Similarly, the textual information can be used to distinguish the copies and give rise to what values can a subexpression evaluate? These questions provide ways of answering the more general question of "what is the result of 0CFA?"

We follow Nielson et al. (1999) and say the result of 0CFA is denoted by some lambda expression or application in the program. This function get applied to this argument?, where the function is particular function applied at this particular call site? or does the virtual path connects to the graph creates a virtual path. There is a virtual apply and lambda node connected by virtual wires. In particular, wires are added only when there is a virtual flow into Clements and label a virtual path. There is a virtual flow into the labeled program:

\[ \lambda f \]

An acceptable control flow analysis for an expression is based on the graph coding of terms. The graphical formulation consists of generating a set of restrictions to those occurring in the abstract cache.

The decision problem for a program is to what values can a subexpression evaluate? It is easy to see that a redex via the route \[ e \]

is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.

An apply node is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.

An apply node is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.

An apply node is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.

An apply node is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.

An apply node is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.

An apply node is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.

An apply node is reachable if it is on the spine of the program, i.e., if there is a redex on the spine of the program.
defined according to the following clauses:

bound to that variable during evaluation.

value, which represents the set of (textual) values that may be

Similarly, the

of lambda expressions, which represents the set of (textual) values

...
tractable fragment by limiting the depth of
cal reasons are in the formal definition of System-
different instances of a type variable, it gets the same fresh name. A
instances are chosen such that if the same expansion is applied to
To meaningfully talk about solutions, we need to specify what it
The expansion variables come to use when doing type inference.
where the term is decorated with expansion variables to form a so-
represent System-
An acceptable control flow analysis for an expression
We follow Nielson et al. (1999) and say the result of 0CFA is
Some care must be taken to ensure leastness when propagat-
for the whole
virtual
wires
maps a program label to an
Term
CFA virtual wire propagation rules.
Proposition 2.1. Proposition 2.1.
Figure 2 defines the virtual path propagation rules. The left hand
structural least solutions according to the partial order

\[ \tau \setminus \sigma \setminus \{z\} = \{\} \]

\[ \tau \rightarrow \sigma \setminus \{z\} = \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]

\[ \tau \rightarrow \sigma \setminus \{\} \]

\[ \tau \setminus \sigma \setminus \{\} \]
Boolean logic

\[
\begin{align*}
TT & \equiv \lambda p. \text{let } \langle x, y \rangle = p \text{ in } \langle x, y \rangle & \text{True} & \equiv \langle \text{TT}, \text{FF} \rangle \\
\text{FF} & \equiv \lambda p. \text{let } \langle x, y \rangle = p \text{ in } \langle y, x \rangle & \text{False} & \equiv \langle \text{FF}, \text{TT} \rangle \\
\text{Copy} & \equiv \lambda b. \text{let } \langle u, v \rangle = b \text{ in } \langle u \langle \text{TT}, \text{FF} \rangle, v \langle \text{FF}, \text{TT} \rangle \rangle \\
\text{Implies} & \equiv \lambda b_1. \lambda b_2. \\
& \quad \text{let } \langle u_1, v_1 \rangle = b_1 \text{ in } \\
& \quad \text{let } \langle u_2, v_2 \rangle = b_2 \text{ in } \\
& \quad \text{let } \langle p_1, p_2 \rangle = u_1 \langle u_2, \text{TT} \rangle \text{ in } \\
& \quad \text{let } \langle q_1, q_2 \rangle = v_1 \langle \text{FF}, v_2 \rangle \text{ in } \\
& \quad \langle p_1, q_1 \circ p_2 \circ q_2 \circ \text{FF} \rangle
\end{align*}
\]
Every variable occurs once.
Every variable occurs once.
Every variable occurs once.
The diagram illustrates relationships between different programming concepts:

- **Sub0CFA**: A subset of CFA concepts focused on the evaluation of expressions.
- **0CFA**: A foundational concept in CFA that deals with simple closures.
- **PTIME**: A computational complexity class that represents problems solvable in polynomial time.
- **Eval**: The process of evaluating expressions within a computational framework.

The diagram suggests that Sub0CFA, 0CFA, and PTIME are interrelated, with Eval overlapping with Sub0CFA, indicating a specific subset of Sub0CFA that falls within the Eval domain.
functions or continuations be in tail position. Adapting this approach to this approach since the direct-style transformation requires all calls to shift. 9

tions flow into the weakening node of the analysis. For example, in Figure 5, we can compute which continu-
as seen in the example. In the same way that 0CFA considers all was introduced through non-linearity of bound variables, approxi-
to the reduction strategy. Note that whereas before, approximation
the fact that the program will return then control returns "normally" and the value
was applied. If was applied. If
then control returns "normally" and the value

Figure 5. The left side of Figure 5 gives the CFA graph for the program:

The right side of Figure 5 gives the CFA graph for the program:

Examining this graph, we can read of an interpretation of
for languages with
programs into continuation passing style (CPS). They observed that
grams with control operators such as
and Mairson 2000) and our graphical formulation of control flow analysis carries over without modification.

Languages such as (Danvy and Filinski 1990), are not immediately amenable
under call-by-value. Thus, the analysis is indifferent
under a call-by-name reduction
b
1
l
1)(= 3
2
)

The left side of Figure 5 gives the CFA graph for the program:

The right side of Figure 5 gives the CFA graph for the program:

Examining this graph, we can read of an interpretation of
for languages with
programs into continuation passing style (CPS). They observed that
grams with control operators such as
and Mairson 2000) and our graphical formulation of control flow analysis carries over without modification.

Languages such as (Danvy and Filinski 1990), are not immediately amenable
under call-by-value. Thus, the analysis is indifferent
under a call-by-name reduction
b
1
l
1)(= 3
2
)

The left side of Figure 5 gives the CFA graph for the program:

The right side of Figure 5 gives the CFA graph for the program:

Examining this graph, we can read of an interpretation of
for languages with
programs into continuation passing style (CPS). They observed that
grams with control operators such as
and Mairson 2000) and our graphical formulation of control flow analysis carries over without modification.

Languages such as (Danvy and Filinski 1990), are not immediately amenable
under call-by-value. Thus, the analysis is indifferent
under a call-by-name reduction
b
1
l
1)(= 3
2
)

The left side of Figure 5 gives the CFA graph for the program:

The right side of Figure 5 gives the CFA graph for the program:

Examining this graph, we can read of an interpretation of
for languages with
programs into continuation passing style (CPS). They observed that
grams with control operators such as
and Mairson 2000) and our graphical formulation of control flow analysis carries over without modification.

Languages such as (Danvy and Filinski 1990), are not immediately amenable
under call-by-value. Thus, the analysis is indifferent
under a call-by-name reduction
b
1
l
1)(= 3
2
)

The left side of Figure 5 gives the CFA graph for the program:

The right side of Figure 5 gives the CFA graph for the program:

Examining this graph, we can read of an interpretation of
for languages with
programs into continuation passing style (CPS). They observed that
grams with control operators such as
and Mairson 2000) and our graphical formulation of control flow analysis carries over without modification.

Languages such as (Danvy and Filinski 1990), are not immediately amenable
under call-by-value. Thus, the analysis is indifferent
under a call-by-name reduction
b
1
l
1)(= 3
2
)}
1CFA and EXPTIME
Datalog-style programming with analysis.
\[ \hat{C} \in \text{Cache} = \text{Lab} \times \text{Lab}^{\leq k} \rightarrow \mathcal{P}(\text{Term} \times \text{Env}) \]

\[ A[(t^{l_1} t^{l_2})^l]^{ce}_\delta = A[t^{l_1}]^{ce}_\delta ; A[t^{l_2}]^{ce}_\delta ; \]

\text{foreach} \ \langle \lambda x.t^{l_0}, ce' \rangle \in \hat{C}(l_1, \delta) :

\begin{align*}
\hat{r}(x, \left\lceil \delta l \right\rceil_k) & \leftarrow \hat{C}(l_2, \delta); \\
A[t^{l_0}]^{ce'}[x \mapsto \left\lceil \delta l \right\rceil_k]; \\
\hat{C}(l, \delta) & \leftarrow \hat{C}(l_0, \left\lceil \delta l \right\rceil_k)
\end{align*}
Hardness of $\kappa$CFA relies on two insights:

1. Program points are approximated by an exponential number of closures.

2. *Inexactness* of analysis engenders *reevaluation* which provides *computational power*. 
Many closures can flow to a single program point:

$$\left( \lambda w. wx_1 x_2 \ldots x_n \right)$$

- $n$ free variables
- an exponential number of possible associated environments mapping these variables to program points (contours of length 1 in 1CFA).
Consider the following *non-linear* example

\[
(\lambda f.(f \text{ True})(f \text{ False}))
\]

\[
(\lambda x. \\
(\lambda p.p(\lambda u.p(\lambda v.(\text{Implies } u v)))))(\lambda w.wx)
\]

Q: What does \text{Implies } u v evaluate to? 
A: \text{True}: it is equivalent to \text{Implies } x x, a tautology.

Q: What flows out of \text{Implies } u v? 
A: both \text{True} and \text{False}: \text{Not true evaluation!}
\( (\lambda f_1. (f_1 \text{ True})(f_1 \text{ False}))\)
\( (\lambda x_1. \)
\( (\lambda f_2. (f_2 \text{ True})(f_2 \text{ False}))\)
\( (\lambda x_2. \)
\( (\lambda f_3. (f_3 \text{ True})(f_3 \text{ False}))\)
\( (\lambda x_3. \)

\[ \cdots \]

\( (\lambda f_n. (f_n \text{ True})(f_n \text{ False}))\)
\( (\lambda x_n. \)
\[ E[(\lambda v. \phi v)(\lambda w. w x_1 x_2 \cdots x_n)](\cdots))] \)
The idea:

★ Break machine ID into an exponential number of pieces
★ Do piecemeal transitions on pairs of puzzle pieces

\[\langle T, S, H, C, b \rangle\]

“At time \(T\), machine is in state \(S\), the head is at cell \(H\), and cell \(C\) holds symbol \(b\)”
\( \langle T, S, H, C, b \rangle \): “At time \( T \), machine is in state \( S \), the head is at cell \( H \), and cell \( C \) holds symbol \( b \)”

1) Compute:
\[
\delta \langle T, S, H, H, b \rangle \langle T, S', H', C', b' \rangle = \langle T + 1, \delta Q(S, b), \delta LR(S, H, b), H, \delta \Sigma(S, b) \rangle
\]

2) Communicate:
\[
\delta \langle T + 1, S, H, C, b \rangle \langle T, S', H', C', b' \rangle = \langle T + 1, S, H, C', b' \rangle \quad (H' \neq C')
\]

3) Otherwise:
\[
\delta \langle T, S, H, C, b \rangle \langle T', S', H', C', b' \rangle = \langle \text{some goofy null value} \rangle \quad (T \neq T' \text{ and } T \neq T' + 1)
\]
Setting up initial ID, iterator, and test:

\((\lambda f_1. (f_1 \ 0) (f_1 \ 1))\)

\((\lambda z_1.\)

\((\lambda f_2. (f_2 \ 0) (f_2 \ 1))\)

\((\lambda z_2.\)...

\((\lambda f_N. (f_N \ 0) (f_N \ 1))\)

\((\lambda z_N.\)

(let \( \Phi = \text{coding of transition function of TM} \) in

\(\text{Widget[Extract}(Y \ \Phi (\lambda w. w \ 0 \ldots 0 Q_0 H_0 z_1 z_2 \ldots z_N 0))]) \ldots)\)

\(\langle T, S, H, C, b \rangle\)
Similar precision, better performance
Precision

kCFA

1CFA

0CFA

Sub0CFA

Simple closure

EXPTIME

PTIME
Precision

kCFA  mCFA
   :   :
1CFA  1CFA
0CFA
Sub0CFA
Simple closure
   :   :

PTIME  EXPTIME  PTIME

PTIME