Control-flow analysis of function call and returns by abstract interpretation
Jan Midtgaard, Thomas Jensen

To cite this version:

HAL Id: inria-00574944
https://inria.hal.science/inria-00574944
Submitted on 9 Mar 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation

Jan Midtgaard
Roskilde University
jmid@ruc.dk

Thomas P. Jensen
CNRS
thomas.jensen@irisa.fr

Abstract
We derive a control-flow analysis that approximates the interproce-
dural control-flow of both function calls and returns in the presence
of first-class functions and tail-call optimization. In addition to an
abstract environment, our analysis computes for each expression
an abstract control stack, effectively approximating where func-
tion calls return across optimized tail calls. The analysis is sys-
tematically calculated by abstract interpretation of the stack-based
C_{\alpha}E_{K} abstract machine of Flanagan et al. using a series of Galois
connections. Abstract interpretation provides a unifying setting in
which we 1) prove the analysis equivalent to the composition of a
continuation-passing style (CPS) transformation followed by an ab-

tract interpretation of a stack-less CPS machine, and 2) extract an
equivalent constraint-based formulation, thereby providing a ratio-
nal reconstruction of a constraint-based control-flow analysis from
abstract interpretation principles.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis

General Terms Languages, Theory, Verification

Keywords Control flow analysis, abstract interpretation, tail-call optimization, continuation-passing style, direct style, constraint-based analysis

1. Introduction
The control flow of a functional program is expressed in terms of
function calls and returns. As a result, iteration in functional pro-
grams is expressed using recursive functions. In order for this
approach to be feasible, language implementations perform tail-call
optimization of function calls [Clinger, 1998], by not pushing a
stack frame on the control stack at call sites in tail position. Con-
sequently functions do not necessarily return control to their caller.
Control-flow analysis (CFA) has long been a staple of program op-
timization and verification. Surprisingly, research on control-flow
analysis has focused on calls: A textbook CFA [Nielson et al., 1999].

Permission to make digital or hard copies of all or part of this work for personal or
classroom use is granted without fee provided that copies are not made or distributed
for profit or commercial advantage and that copies bear this notice and the full citation
on the first page. To copy otherwise, to republish, to post on servers or to redistribute
to lists, requires prior specific permission and/or a fee.

Figure 1: The corresponding call graphs

that “will determine where the flow of control may be transferred
to in the case of a function return.” The resulting analysis thereby
approximates both call and return information for a higher-order,
direct-style language. Interestingly it does so by approximating the
control stack.

Consider the example program in Fig. 1(a). The program con-
tains three functions: two named function g and f and an anony-
umous function fn x => x. A standard direct-style CFA can deter-
mine that the applications of k in each branch of the conditional
will call the anonymous function fn x => x at run time. Build-

ing a call-graph based on this output gives rise to Fig. 1(b), where
we have named the main expression of the program main. In addi-
tion to the above resolved call, our analysis will determine that
the anonymous function returns to the let-binding of y in main upon
completion, rather than to its caller. The analysis hence gives rise
to the call graph in Fig. 1(c).

On a methodological level, we derive the analysis systematically
by Cousot-Cousot-style abstract interpretation. The analysis
approximates the reachable states of an existing abstract machine
from the literature: the C_{\alpha}E_{K} machine of Flanagan et al. [1993].
We obtain the analysis as the result of composing the collecting
semantics induced by the abstract machine with a series of Galois
connections that each specifies one aspect of the abstraction in
the analysis.

We show how the abstract interpretation formulation lends it-
self to a lock-step equivalence proof between our analysis and a
previously derived CPS-based CFA. More precisely, we define a
relation between the abstract domains of the analyses that is a sim-
ulation between the two, reducing the proof to a fixpoint induction
over the abstract interpretations.
To sum up, the main contributions of this article are:

- An abstract interpretation-derivation of a CFA for a higher-order functional language from a well-known operational semantics,
- a resulting CFA with reachability which computes both call and return control-flow,
- a proof of equivalence of the analysis of programs in direct style and the CPS analysis of their CPS counterparts,
- an equivalent constraint-based analysis extracted from the above.

1.1 Related work

We separate the discussion of related analyses in two: direct-style analyses and analyses based on CPS.

**Direct-style CFA** has a long research history. Jones [1981] initially developed methods for approximating the control flow of lambda terms. Since then Sestoft [1989] conceived the related closure analysis. Palsberg [1995] simplified the analysis and formulated an equivalent constraint-based analysis. At the same time Heintze [1994] developed a related set-based analysis formulated in terms of set constraints. For a detailed account of related work, we refer to a recent survey of the area [Midtgaard, 2007]. It is worth emphasizing that all of the above analyses focus on calls, in that they approximate the source lambdas being called at each call site. As such they do not directly determine return flow for programs in direct style.

**CPS-based CFA** was pioneered by Shivers [1988] who formulated control-flow analysis for Scheme. Since then several analyses have been formulated for CPS [Ayers, 1992, Ashley and Dybvig, 1998, Might and Shivers, 2006]. In CPS all calls are tail calls, and even returns are encoded as calls to the current continuation. By determining “call flow” and hence the receiver functions of such continuation calls, a CPS-based CFA thereby determines return flow without additional effort.

**The impact of CPS transformation on static analyses** originates in binding-time analysis, for which the transformation is known to have a positive effect [Consel and Danvy, 1991, Damian and Danvy, 2003]. As to the impact of CPS transformation on CFA we separate the previous work on the subject in two:

1. results relating an analysis specialized to the source language to an analysis specialized to the target language (CPS), and
2. results relating the analysis of a program to the same analysis of the CPS transformed program.

Sabry and Felleisen [1994] designed and compared specialized analyses and hence falls into the first category as does the present paper. Damian and Danvy [2003] related the analysis of a program and its CPS counterpart for a standard flow-logic CFA (as well as for two binding-time analyses), and Palsberg and Wand [2003] related the analysis of a program and its CPS counterpart for a standard conditional constraint CFA. Hence the latter two fall into the second category.

We paraphrase the relevant theorems of Sabry and Felleisen [1994], of Damian and Danvy [2003], of Palsberg and Wand [2003], and of the present paper in order to underline the difference between the contributions (C refers to non-trivial, 0-CFA-like analyses defined in the cited papers, \( p \) ranges over direct-style programs, \( cps \) denotes CPS transformation, and \( \sim \) denotes analysis equivalence). Our formulations should not be read as a formal system, but only as a means for elucidating the difference between the contributions.

**Sabry and Felleisen [1994]:**

exists analyses \( C_1, C_2 : \forall p, C_1(p) \sim C_2(cps(p)) \)

**Damian and Danvy [2003], Palsberg and Wand [2003]:**

exists analysis \( C : \forall p, C(p) \sim C(cps(p)) \)

**Present paper, Theorem 5.1:**

exists analyses \( C_1, C_2 : \forall p, C_1(p) \sim C_2(cps(p)) \)

Our work relates to all of the above contributions. The disciplined derivation of specialized CPS and direct-style analyses results in comparable analyses, contrary to Sabry and Felleisen [1994]. Furthermore our equivalence proof extends the results of Damian and Danvy [2003] and Palsberg and Wand [2003] in that we relate both call flow, return flow, and reachability, contrary to their relating only the call flow of standard CFAs. In addition, the systematic abstract interpretation-based approach suggests a strategy for obtaining similar equivalence results for other CFAs derived in this fashion.

**Formulating CFA in the traditional abstract interpretation framework** was stated as an open problem by Nielsen and Nielsen [1997]. It has been a recurring theme in the work of the present authors. In an earlier paper Spoto and Jensen [2003] investigated class analysis of object-oriented programs as a Galois connection-based abstraction of a trace semantics. In a recent article [Midtgard and Jensen, 2008a], the authors systematically derived a CPS-based CFA from the collecting semantics of a stack-less machine. While investigating how to derive a corresponding direct-style analysis we discovered a mismatch between the computed return information. As tail calls are identified syntactically, the additional information could also have been obtained by a subsequent analysis after a traditional direct-style CFA. However we view the need for such a subsequent analysis as a strong indication of a mismatch between the direct-style and CPS analysis formulations. Debray and Proebsting [1997] have investigated such a “return analysis” for a first-order language with tail-call optimization. This paper builds a semantics-based CFA that determines such information, and for a higher-order language.

**The systematic design of constraint-based analyses** is a goal shared with the flow logic framework of Nielsen and Nielsen [2002]. In flow logic an analysis specification can be systematically transformed into a constraint-based analysis. The present paper instead extracts a constraint-based analysis from an analysis developed in the original abstract interpretation framework.

The idea of CFA by control stack approximation, applies equally well to imperative or object-oriented programs, but it is beyond the scope of this paper to argue this point. Due to space limitations most calculations and proofs are also omitted. We refer the reader to the accompanying technical report [Midtgard and Jensen, 2008b].

2. Language and semantics

Our source language is a simple call-by-value core language known as *administrative normal form* (ANF). The grammar of ANF terms is given in Fig. 2(a). Following Reynolds, the grammar distinguishes serious expressions, i.e., terms whose evaluation may diverge, from trivial expressions, i.e., terms without risk of divergence. Trivial expressions include constants, variables, and functions, and serious expressions include returns, let-bindings, tail calls, and non-tail calls. Programs are serious expressions.

The analysis is calculated from a simple operational semantics in the form of an abstract machine. We use the environment-based \( C,EK \) abstract machine of Flanagan et al. [1993] given in Fig. 2 in which functional values are represented using closures, i.e., pairs of a lambda-expression and an environment. The environment component captures the (values of the) free variables of the lambda. Machine states are triples consisting of a serious expression, an
environment and a control stack. The control stack is composed of elements ("stack frames") of the form \([x, a, e]\) where \(x\) is the variable receiving the return value \(w\) of the current function call, and \(a\) is a serious expression whose evaluation in the environment \(e[x \mapsto w]\) represents the rest of the computation in that stack frame. The empty stack is represented by \texttt{stop}. The machine has a helper function \(\mu\) for evaluation of trivial expressions. The machine is initialized with the input program, with an empty environment, and with an initial stack, that will bind the result of the program to a special variable \(x_0\) before halting. Evaluation follows by repeated application of the machine transitions.

3. Abstract interpretation basics

We assume some familiarity with the basic mathematical facts recalled in Appendix A. Canonical abstract interpretation approximates the collecting semantics of a transition system [Cousot, 1981]. A standard example of a collecting semantics is the reachable states from a given set of initial states \(I\). A given transition function \(T\) defined as: \(T(S) = I \cup \{p \mid \exists p' \in S:: p' \rightarrow s \}\), we can compute the reachable states of \(T\) as the least fixed-point \(lfp T\) of \(T\). The collecting semantics is ideal, in that it is the most precise analysis. Unfortunately it is in general uncomputable. Abstract interpretation therefore approximates the collecting semantics, by instead computing a fixed-point over an alternative and perhaps simpler domain. For this reason, abstract interpretation is also referred to as a theory of fixed-point approximation.

Abstractions are formally represented as Galois connections which connect complete lattices through a pair of adjoint functions \(\alpha\) and \(\gamma\) (see Appendix A). Galois connection-based abstract interpretation suggests that one may derive an analysis systematically by composing the transition function with these adjoints: \(\alpha \circ T \circ \gamma\). In this setting Galois connections allow us to gradually refine the collecting semantics into a computable analysis function by mere calculation. An alternative "recipe" consists in rewriting the composition of the abstraction function and transition function \(\alpha \circ T\) into something of the form \(T^\circ \circ \alpha\), from which the analysis function \(T^\circ\) can be read off [Cousot and Cousot, 1992a]. Cousot [1999] has shown how to systematically construct a static analyser for a first-order imperative language using calculational abstract interpretation.

4. Approximating the \(C_a E\)K abstract machine

As our collecting semantics we consider the reachable states of the \(C_a E\)K machine, expressed as the least fixed point \(lfp F\) of the following transition function.

\[
F: \mathcal{P}(C \times Env \times K) \rightarrow \mathcal{P}(C \times Env \times K)
\]

\[
F(S) = I_p \cup \{s \mid \exists s' \in S:: s' \rightarrow s\}
\]

where \(I_p = \{\{p, e, x, x_0, e\}:: \text{stop}\}\)

First we formulate in Fig. 3(a) an equivalent helper function \(\mu_k\) extended to work on sets of environments.

**Lemma 4.1.** \(\forall t, e:: (\mu (t, e)) = \mu_k (t, [e])\)

The equivalence of the two helper functions follow straightforwardly. This lemma enables us to express an equivalent collecting semantics based on \(\mu_k\), which appears in Fig. 3. The equivalence of \(F\) and \(F^\circ\) follows from the lemma and by unfolding the definitions.

The abstraction of the collecting semantics is staged in several steps. Figure 4 provides an overview. Intuitively, the analysis extracts three pieces of information from the set of reachable states.

1. An approximation of the set of reachable expressions.
2. A relation between expressions and control stacks that represents where the values of expressions are returned to.
3. An abstract environment mapping variables to the expressions that may be bound to that variable. This is standard in CFA and allows to determine which functions are called at a given call site.

Keeping an explicit set of reachable expressions is more precise than leaving it out, once we further approximate the expression-stack pairs. Alternatively the reachable expressions would be approximated by the expressions present in the expression-stack relation. However expressions may be in the expression-stack relation without ever being reached. An example hereof would be a diverging non-tail call.

To formalize this intuition, we first perform a Cartesian abstraction of the machine states, however keeping the relation between expressions and their corresponding control stacks. The second step in the approximation consists in closing the triples by a closure operator, to ensure that (a) any saved environment on the stack or nested within another environment is itself part of the environment.
Finally as a third step we approximate stacks by their top element,
above is defined formally as follows. The mapping that extracts the three kinds of information described
ther down in a control stack are also contained in the expression-
set, and (b) that all expression-control stack pairs that appear further
down in a control stack are also contained in the expression-
stack relation. We explain this in more detail below (Section 4.2).
Finally as a third step we approximate stacks by their top element,
merge expressions with the same return point into equivalence
classes, and we approximate closure values by their lambda expression.
In the following sections we provide a detailed explanation of
each abstraction in turn.

4.1 Projecting machine states
The mapping that extracts the three kinds of information described
above is defined formally as follows.

\[
\begin{align*}
\alpha_S & : \mathcal{S} \times \mathcal{F} \to \mathcal{S} \\
\gamma_i & : \mathcal{F} \to \mathcal{S} \\
\gamma_e & : \mathcal{S} \times \mathcal{F} \\
\end{align*}
\]

**Lemma 4.2.** \(\alpha_c, \gamma_c\) is a Galois connection.

The above Galois connection and the proof hereof closely resem-
bles the independent attributes abstraction, which is a known
Galos connection. We use the notation \(\cup_c\) and \(\subseteq_c\) for the compon-
entwise join and componentwise inclusion of triples.

As traditional [Coustous and Coustot, 1979, 1992a, 1994], we will
assume that the abstract product domains throughout this article
have been reduced, i.e., all triples \((A, B, C)\) with a bottom com-
ponent \((A = \bot_a, B = \bot_b, C = \bot_c)\) have been eliminated and
replaced by a single bottom element \((\bot_a, \bot_b, \bot_c)\).

Based on this abstraction we can now calculate a new transfer
function \(\mathcal{F}^\times\). The resulting transition function appears in Fig. 5. By
construction, the transition function satisfies the following theorem.

**Theorem 4.1.**

\[\forall C, F : \alpha_c : (\mathcal{F}^\times(\langle C, F, E \rangle)) = \mathcal{F}^\times(\langle C, F, E \rangle)\]

4.2 A closure operator on machine states
For the final analysis, we are only interested in an abstraction of
the information present in an expression-stack pair. More precisely,
we aim at only keeping track of the link between an expression
and the top stack frame in effect during its evaluation, throwing
away everything below. However, we need to make this information
explicit for all expressions appearing on the control stack, i.e.,
for a pair \((a, [x, a', e] :: k)\) we also want to retain that \(a'\) will
be evaluated with control stack \(k\). Similarly, environments can be
stored on the stack or inside other environments and will have to be
extracted. We achieve this by defining a suitable closure operator
on these nested structures.

For environments, we adapt the definition of a constituent
relation due to Milner and Tofte [1991] We say that each com-
ponent \(x_i\) of a tuple \((x_0, \ldots, x_n)\) is a constituent of the tuple, written
\([x_0, \ldots, x_n] \succ x_i\). For a partial function \(f = [x_0 \mapsto w_0, \ldots, x_n \mapsto w_n]\), we say that each \(w_i\) is a constituent of the function, written
\(f \succ w_i\). We write \(\succ\) for the reflexive, transitive closure of the
constituent relation.

To deal with the control stack, we define an order on expression-
stack pairs. Two pairs are ordered if (a) the stack component of
the second is the tail of the first’s stack component, and (b) the
expression component of the second, resides on the top stack frame
of the first pair:

\[\langle a, [x, a', e] :: k \rangle \succ \langle a', k' \rangle\]

We write \(\succ\) for the reflexive, transitive closure of the expression-stack pair ordering.

Next, we consider an operator \(\rho\), defined in terms of the constituent
relation and the expression-stack pair ordering. The opera-
tor \(\rho\) ensures that all constituent environments will themselves
belong to the set of environments, and that any structurally smaller
expression-stack pairs are also contained in the expression-stack relation.

**Definition 4.1.**

\[\rho(\langle C, F, E \rangle) = \langle C, \{\{a, k\} | \exists (a', k') \in F: (a', k') \succ (a, k)\}, \{e | \exists (a, k) \in F: (a, k) \succ^* e \land \exists e' \in E: e' \succ^* e\}\} \]

We need to relate the expression-stack ordering to the con-
stituent relation. By case analysis one can prove that \(\forall (a, k), (a', k') : (a, k) \succ (a', k') \iff k \succ k'\). By structural induction (on the
stack component) it now follows that \(\forall (a, k), (a', k') : (a, k) \succ^* (a', k') \iff k \succ^* k'\). Based on these results we can verify that \(\rho\) is a
closure operator and formulate an abstraction on the triples:

\[\rho(\langle C, F, E \rangle) \times \rho(\langle C, F, E \rangle) \times \rho(\langle C, F, E \rangle) \times \rho(\langle C, F, E \rangle) \]

Milner and Tofte define the constituent relation for finite functions.
We use the notation $\cup_p$ for the join operation $\lambda x.\rho(\cup_p x)\ X$ on
the closure operator-induced complete lattice. First observe that in our
case:

$$\cup_p = \lambda x.\rho(\cup_i X_i) = \lambda x.\rho(x) = \lambda x.\bigcup_i X_i = \bigcup_i x$$

Based on the closure operator-based Galois connection, we can calculate a
new intermediate transfer function $F^\rho$. The resulting transfer function appears in Fig. 6. This transfer function differs
only minimally from the one in Fig. 5, in that (a) the signature has changed, (b) the set of initial states has been “closed” and
now contains the structurally smaller pair $(x, \text{stop})$, and (c) the four indexed joins now each join “closed” triples in the image
of the closure operator. By construction, the new transition function satisfies the following theorem.

**Theorem 4.2.** Let $(C, F, E, \rho) \in (C \times K \lambda x.\rho(\cup_p x)\ X) = F^\rho((C, F, E)) \to F^\rho((C, F, E))$

### 4.3 Abstracting the expression-stack relation

Since stacks can grow unbounded (for non-tail recursive programs), we need to approximate the stack component and hereby the
expression-stack relation. We first formulate a grammar of abstract
states and an elementwise transfer function $\rho : C \lambda x.\rho(\cup_p x)\ X \to C \lambda x.\rho(\cup_p x)\ X$
operating on expression-stack pairs.

$$K^2 \ni \cup_i X_i :\lambda x.\rho(\cup_i X_i) = \lambda x.\rho(x) = \lambda x.\bigcup_i X_i = \bigcup_i x$$

Based on the elementwise operator we can now use an elementwise abstraction.

**Elementwise abstraction** [Cousot and Cousot, 1997]: A given

$$\rho : C \lambda x.\rho(\cup_p x)\ X \to C \lambda x.\rho(\cup_p x)\ X$$

### 4.4 Abstracting environments

We also abstract values using an elementwise abstraction. Again we formulate a grammar of abstract values and an elementwise

$$\begin{align*}
\rho(C) \ni \cup_i X_i :\lambda x.\rho(\cup_i X_i) = \lambda x.\rho(x) = \lambda x.\bigcup_i X_i = \bigcup_i x
\end{align*}$$

Notice how some expressions share the same return point (read: same
stack): the expressions let $x = t$ in $s$ and $s$ share the return
point, and let $x = t \text{ stop}$ in $s$ and $s$ share the same return point.
In order to eliminate such redundancy we define an equivalence relation
that abstracts such expressions.

The resulting $\rho(C) \ni \cup_i X_i :\lambda x.\rho(\cup_i X_i) = \lambda x.\rho(x) = \lambda x.\bigcup_i X_i = \bigcup_i x$

Based hereon we define a second elementwise operator $\rho' : C \lambda x.\rho(\cup_p x)\ X \to C \lambda x.\rho(\cup_p x)\ X$

**Proof.** The first half follows from the assumptions. The second half follows from monotonicity of $\alpha_{\rho'}$, and the definitions of $\alpha_{\rho'}$, and $\cup_i x$.

**4.4 Abstracting environments**
By considering only closed programs, we statically ensure a gainst

We can calculate an abstract helper function, by “pushing

Subset abstraction [Cousot and Cousot, 1997]:

A Galois connection:

The abstraction of environments, which are partial functions, can be composed by a series of well-known Galois connections.

Pointwise abstraction of a set of functions [Cousot and Cousot, 1994]: A given Galois connection on the co-domain \( \rho(C) ; \subseteq \)

\[ \frac{\alpha}{\gamma} (C; \subseteq) \] induces a Galois connection on a set of functions:

\[ \langle \rho(D \rightarrow C) ; \subseteq \rangle \cong \frac{\gamma}{\alpha} (D \rightarrow C^\subseteq; \subseteq) \]

\[ \alpha_\ominus(F) = \lambda d. \alpha(\{ f(d) | f \in F \}) \]

\[ \gamma_\ominus(A) = \{ f | \forall d : f(d) \in \gamma(A(d)) \} \]

Subset abstraction [Cousot and Cousot, 1997]: Given a set \( C \) and a strict subset \( A \subset C \) hereof, the restriction to the subset induces a Galois connection:

\[ \langle \rho(C) ; \subseteq \rangle \cong \frac{\gamma}{\alpha} (\rho(A) ; \subseteq) \]

\[ \alpha_{\ominus}(X) = X \cap A \]

A standard trick is to think of partial functions \( r : D \rightarrow C \) as total functions \( r : D \rightarrow (C \cup \perp) \) where \( \perp \subseteq \perp \subseteq \perp \), for all \( c \in C \). Consider environments \( e \in \text{Var} \rightarrow \text{Val} \) to be total functions \( \text{Var} \rightarrow (\text{Val} \cup \perp) \) using this idea. In this context the bottom element \( \perp \) will denote variable lookup failure. Now compose a subset abstraction \( \rho(\text{Val} \cup \perp) \cong \frac{\gamma}{\alpha} (\rho(\text{Val}) \) with the above value abstraction, and feed the result to the pointwise abstraction above. The result is a pointwise abstraction of a set of environments, not explicitly modelling variable lookup failure: \( \rho(\text{Env}) \cong \frac{\gamma}{\alpha} (\rho(\text{Val}) \). By considering only closed programs, we statically ensure against failure of variable-lookup, hence disregarding \( \perp \) loses no information.

4.5 Abstracting the helper function

We can calculate an abstract helper function, by “pushing \( \alpha_\ominus \)’s” under the function definition, and reading off a resulting abstract definition.

Lemma 4.4. \( \forall t, E : \alpha_\ominus(\mu_{\ominus}(t, E)) = \mu_\ominus^\ominus(t, \alpha_\ominus(E)) \)

The resulting helper function \( \mu_\ominus^\ominus(\text{t} \times \text{Env}^\ominus) \rightarrow \rho(\text{Val}) \) reads:

\[ \mu_\ominus^\ominus(\text{c}, \text{E}^\ominus) = \{ c \} \]

\[ \mu_\ominus^\ominus(\text{f}, \text{E}^\ominus) = \text{E}^\ominus(\text{x}) \]

where we write \( \text{Env}^\ominus \) as shorthand for \( \text{Var} \rightarrow \rho(\text{Val}) \). We shall need a lemma relating the two helper function definitions on closed environments.

Lemma 4.5. Helper function on closed environments (1)

Let \( \langle C, F, E \rangle \in \rho(\rho(C) \times \rho(C \times K) \times \rho(\text{Env})) \) be given.

\[ \{ \text{fn x =⇒ s, e} \} \subseteq \mu_{\ominus}(t, E) \implies e \in E \]

\[ \land \{ \{ \text{fn x =⇒ s} \} \subseteq \mu_\ominus^\ominus(t, \alpha_\ominus(\text{E})) \} \]

The above lemma is easily extended to capture nested environments in all values returned by the helper function:

Lemma 4.6. Helper function on closed environments (2)

Let \( \langle C, F, E \rangle \in \rho(\rho(C) \times \rho(C \times K) \times \rho(\text{Env})) \) be given.

\[ \{ w \} \subseteq \mu_{\ominus}(t, E) \land w \not\supseteq e^\ominus \implies e^\ominus \in E \]

4.6 Abstracting the machine states

We abstract the triplet of sets into abstract Triples by a componentwise abstraction.

Componentwise abstraction [Cousot and Cousot, 1994]: Assuming a series of Galois connections: \( \rho(C_i) \cong \frac{\gamma}{\alpha} (\text{A}_i) \) for \( i \in \{1, \ldots, n\} \), their componentwise composition induces a Galois connection on tuples:

\[ \rho(C_1) \times \ldots \times \rho(C_n) ; \subseteq \times \cong \frac{\gamma}{\alpha} \{(A_1 \times \ldots \times A_n) ; \subseteq \} \]

\[ \alpha_\ominus((x_1, \ldots, x_n)) = (\alpha_1(x_1), \ldots, \alpha_n(x_n)) \]

\[ \gamma_\ominus((x_1, \ldots, x_n)) = (\gamma_1(x_1), \ldots, \gamma_n(x_n)) \]

We write \( \cup_\ominus \) and \( \subseteq_\ominus \) for componentwise join and inclusion, respectively.

For the set of expressions \( \rho(\text{C}) \) we use the identity abstraction consisting of two identity functions. For the expression-stack
relation $\rho(C \times K)$ we use the expression-stack abstraction $\alpha_x$ developed in Section 4.3. For the set of environments $\rho(\text{Env})$ we use the environment abstraction $\alpha_{\text{env}}$ developed in Section 4.4.

Using the alternative “recipe” we can calculate the analysis by “pushing $\alpha_{\text{env}}$’s” under the intermediate transition function: $\alpha_{\text{env}}(F^3(\langle C, F, E \rangle)) \subseteq F^3(\langle C, \alpha_{\text{env}}(F), \alpha_{\text{env}}(E) \rangle)$ from which the final definition of $F^3$ can be read off. The resulting analysis appears in Fig. 7. The alert reader may have noticed that this final abstraction is not complete in that the above equation contains an inequality. Completeness is a desirable goal in an abstract interpretation but unfortunately it is not possible in general without refining the abstract domain [Giacobazzi et al., 2000]. Consider for example the addition operator over the standard sign-domain: $0 = \alpha(1 + (-1)) \sqsubseteq \alpha(1) + \alpha(-1) = \top$. As traditional [Cousot, 1999], we instead limit upward judgements to a minimum.

As a corollary of the construction, the analysis safely approximates the reachable states of the abstract machine.

**Corollary 4.1.** $\alpha_{\text{env}} \circ \rho \circ \alpha_x(\text{let } F) \subseteq \text{lfp } F^3$

### 4.7 Characteristics of the analysis

First of all the analysis incorporates reachability: it computes an approximate set of reachable expressions and will only analyse those reachable program fragments. Reachability analyses have previously been discovered independently [Ayers, 1992; Palsberg and Schwartzbach, 1995, Gasser et al., 1997]. In our case they arise naturally from a projecting abstraction of a reachable states collecting semantics.

Second the formulation materializes monomorphism into two mappings: (a) one mapping merging all bindings to the same variable, and (b) one mapping merging all calling contexts of the same function. Both wellknown facts are well presented, but our presentation is novel in that it literally captures this phenomenon in two approximation functions.

Third the analysis handles returns inside-out (“callee-restore”), in that the called function restores control from the approximate control stack and propagates the obtained return values. This differs from the traditional presentations [Palsberg, 1995; Nielson et al., 1999] that handle returns outside-in (“caller-restore”) where the caller propagates the obtained return values from the body of the function to the call site (typically formulated as conditional constraints).

$$F^2 : p \rightarrow \rho(C) \times (C/\equiv \rightarrow \rho(K^2)) \times \text{En} \rightarrow \rho(C) \times (C/\equiv \rightarrow \rho(K^2)) \times \text{En}^2$$

$$F^3(\langle C, F, E \rangle) = (\langle p \rangle, [p]_\equiv \mapsto ([x], [x]_\equiv \mapsto \{\text{stop}\}), \lambda, \emptyset)$$

- $\cup_{\text{let } x=t \text{ in } s} C \subseteq \{[s], F^1, E^1 \cup [x \mapsto \mu(t, E^1)]\}$
- $\cup_{\text{let } x=t \text{ in } s} C \subseteq \{[s], F^1, E^1 \cup [x \mapsto \mu(t, E^1)]\}$
- $\cup_{\text{let } x=t \text{ in } s} C \subseteq \{[s], F^1, E^1 \cup [x \mapsto \mu(t, E^1)]\}$

**Figure 7:** The resulting analysis function

---

5. **Analysis equivalence**

In previous work [Midggaard and Jensen, 2008a] we derived an initial CFA with reachability for a CPS language from the stack-less CE-machine [Flanagan et al., 1993]. In this section we show that the present ANF analysis achieves the same precision as obtained by first transforming a program into CPS and then using the CPS analysis. This is done by defining a relation that captures how the direct-style analysis and the CPS analysis operate in lock-step.

The grammar of CPS terms is given in Fig. 8. The grammar distinguishes variables in the original source program $x \in X$, from intermediate variables $v \in V$ and continuation variables $k \in K$. We assume the three classes are non-overlapping. Their union constitute the domain of CPS variables $\text{Var} = X \cup V \cup K$

| $CProg \triangleright p \ ::= \text{fn } k \mapsto e$ | (CPS programs) |
| $SExp \triangleright e \ ::= t_{0} t_{1} c \ | \ c t$ | (serious CPS expressions) |
| $TExp \triangleright t \ ::= x \ | \ v \ | \ \text{fn } x, k \mapsto e$ | (trivial CPS expressions) |
| $CEExp \triangleright c \ ::= \text{fn } v \mapsto e \ | \ k$ | (continuation expressions) |

**Figure 8:** BNF of CPS language

5.1 **CPS transformation and back again**

In order to state the relation between the ANF and CPS analyses we first recall the relevant program transformations. The below presentation is based on Danvy [1991], Flanagan et al. [1993], and Sabry and Felleisen [1994].

**The CPS transformation** given in Fig. 9(a) is defined by two mutually recursive functions — one for serious and trivial expressions. A continuation variable $k$ is provided in the initial call to $\mathcal{F}$. A fresh $k$ is generated in $\mathcal{F}$’s lambda abstraction case. To ease the expression of the original, we choose $k$ unique to the serious expression $a \mapsto k_a$. It follows that we only need one $k$ per lambda abstraction in the original program $\mathcal{F}$ + additional $k$ in the initial case.

It is immediate from the definition of $\mathcal{F}$ that the CPS transformation of a let-binding $\text{let } x=t \text{ in } s$ and the CPS transformation of its body $s$ share the same continuation identifier — and similarly for non-tail calls. Hence we shall equate the two:

**Definition 5.1.** $k_s \equiv k_{s'}$ iff $s \equiv s'$

---

"traits"

"function to the call site (typically formulated as conditional constraints)."
The direct-style transformation given in Fig. 9(b) is defined by two mutually recursive functions over serious and trivial CPS expressions. We define the direct-style transformation of a program \( \text{fn } k \Rightarrow e \) as the direct-style transformation of its body \( \mathcal{W}[e] \).

Transforming a program, a serious or an expression to CPS and back to direct style yields the original expression, which can be confirmed by (mutual) structural induction on trivial and serious expressions.

Lemma 5.1. \( \mathcal{P}[\mathcal{W}[p]] = p \land \mathcal{W}[\mathcal{F}_k[s]] = s \land \mathcal{P}[\mathcal{V}[t]] = t \)

5.2 CPS analysis

We recall the CPS analysis of Midgaard and Jensen [2008a] in Fig. 10. It is defined as the least fixed point of a program specific transfer function \( T^p_k \). The definition relies on two helper functions \( \mu^t_x \) and \( \mu^t_y \) for trivial and continuation expressions, respectively. The analysis computes a pair consisting of (a) a set of serious expressions (the reachable expressions) and (b) an abstract environment. Abstract environments map variables to abstract values. Abstract environments and function closures.

\( \mathcal{P} : CProg \rightarrow P \)

\( \mathcal{P}[\text{fn } k \Rightarrow e] = \mathcal{W}[e] \)

\( \mathcal{W} : \text{SExp} \rightarrow C \)

\( \mathcal{W}[k t] = \mathcal{P}[t] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t = \text{let } v = \mathcal{P}[t] \text{ in } \mathcal{W}[e] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 k = \mathcal{P}[t_0] \mathcal{P}[t_1] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 (\text{fn } v \Rightarrow e) = \text{let } v = \mathcal{P}[t_0] \mathcal{P}[t_1] \text{ in } \mathcal{W}[e] \)

\( \mathcal{P} : TExp \rightarrow T \)

\( \mathcal{P}[x] = x \)

\( \mathcal{P}[v] = v \)

\( \mathcal{P}[\text{fn } x, k \Rightarrow e] = \text{fn } x \Rightarrow \mathcal{W}[e] \)

(b) Direct-style transformation

\( \mathcal{P} : CProg \rightarrow P \)

\( \mathcal{P}[\text{fn } k \Rightarrow e] = \mathcal{W}[e] \)

\( \mathcal{W} : \text{SExp} \rightarrow C \)

\( \mathcal{W}[k t] = \mathcal{P}[t] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t = \text{let } v = \mathcal{P}[t] \text{ in } \mathcal{W}[e] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 k = \mathcal{P}[t_0] \mathcal{P}[t_1] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 (\text{fn } v \Rightarrow e) = \text{let } v = \mathcal{P}[t_0] \mathcal{P}[t_1] \text{ in } \mathcal{W}[e] \)

\( \mathcal{P} : TExp \rightarrow T \)

\( \mathcal{P}[x] = x \)

\( \mathcal{P}[v] = v \)

\( \mathcal{P}[\text{fn } x, k \Rightarrow e] = \text{fn } x \Rightarrow \mathcal{W}[e] \)

(b) Direct-style transformation

\( \mathcal{P} : CProg \rightarrow P \)

\( \mathcal{P}[\text{fn } k \Rightarrow e] = \mathcal{W}[e] \)

\( \mathcal{W} : \text{SExp} \rightarrow C \)

\( \mathcal{W}[k t] = \mathcal{P}[t] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t = \text{let } v = \mathcal{P}[t] \text{ in } \mathcal{W}[e] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 k = \mathcal{P}[t_0] \mathcal{P}[t_1] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 (\text{fn } v \Rightarrow e) = \text{let } v = \mathcal{P}[t_0] \mathcal{P}[t_1] \text{ in } \mathcal{W}[e] \)

\( \mathcal{P} : TExp \rightarrow T \)

\( \mathcal{P}[x] = x \)

\( \mathcal{P}[v] = v \)

\( \mathcal{P}[\text{fn } x, k \Rightarrow e] = \text{fn } x \Rightarrow \mathcal{W}[e] \)

(b) Direct-style transformation

\( \mathcal{P} : CProg \rightarrow P \)

\( \mathcal{P}[\text{fn } k \Rightarrow e] = \mathcal{W}[e] \)

\( \mathcal{W} : \text{SExp} \rightarrow C \)

\( \mathcal{W}[k t] = \mathcal{P}[t] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t = \text{let } v = \mathcal{P}[t] \text{ in } \mathcal{W}[e] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 k = \mathcal{P}[t_0] \mathcal{P}[t_1] \)

\( \mathcal{W}[\text{fn } v \Rightarrow e] t_0 t_1 (\text{fn } v \Rightarrow e) = \text{let } v = \mathcal{P}[t_0] \mathcal{P}[t_1] \text{ in } \mathcal{W}[e] \)

\( \mathcal{P} : TExp \rightarrow T \)

\( \mathcal{P}[x] = x \)

\( \mathcal{P}[v] = v \)

\( \mathcal{P}[\text{fn } x, k \Rightarrow e] = \text{fn } x \Rightarrow \mathcal{W}[e] \)

(b) Direct-style transformation

We are now in position to state our main theorem relating the ANF analysis to the CPS analysis. Intuitively the theorem relates:

- reachability in ANF to CPS reachability
- abstract stacks in ANF to CPS continuation closures
- abstract stack bottom in ANF to CPS initial continuation
- ANF closures to CPS function closures

Theorem 5.1. Let \( p \) be given. Let \( \langle C, F^t, E^t \rangle = \text{lfp} F^p_k \) and \( \langle Q^t, R^t \rangle = \text{lfp} T^p_k \). Then

\( s \in C \iff \mathcal{F}_k[s] \in Q^t \land \mathcal{V}[s] \in R^t \land \mathcal{V}^t(x) \land \mathcal{V}^t(y) \land \mathcal{V}[s] \in R^t(y) \)

For the purpose of the equivalence we equate the special variables \( x \) and \( v \), both naming the result of the computations. We prove the theorem by combining an implication in each direction with the identity from Lemma 5.1. We formulate both implication as relations and prove that both relations are preserved by the transfer functions.

5.4 ANF-CPS equivalence

We formally define a relation \( R^{\text{ANF}}_{\text{CPS}} \) that relates ANF analysis triples to CPS analysis pairs.

Definition 5.2. \( \langle C, F^t, E^t \rangle R^{\text{ANF}}_{\text{CPS}} \langle Q^t, R^t \rangle \) iff \( \forall a: \)

\( s \in C \implies \mathcal{F}_k[s] \in Q^t \land \mathcal{V}[s] \in R^t \land \mathcal{V}^t(x) \land \mathcal{V}^t(y) \land \mathcal{V}[s] \in R^t(y) \)

First we need a small lemma relating the ANF helper function to one of the CPS helper functions.

Lemma 5.2. \( \langle C, F^t, E^t \rangle R^{\text{ANF}}_{\text{CPS}} \langle Q^t, R^t \rangle \) if \( \forall a: \)

\( [f x \Rightarrow s] \in \mu^t(x, E^t) \land \langle C, F^t, E^t \rangle R^{\text{ANF}}_{\text{CPS}} \langle Q^t, R^t \rangle \)

\( \implies [f x, k a] \Rightarrow \mathcal{F}_k[s] \in \mu^t(y[t], R^t) \)
The relation is preserved by the transfer functions.

**Theorem 5.2.**
\[ (C, F^2, E^2) \text{ RANF}_{\text{CPS}} (Q^2, R^2) \]
\[ \implies F^2_{\text{CPS}} (C, F^2, E^2) \text{ RANF}_{\text{CPS}} T^{\sharp}_{\text{CPS}} (Q^2, R^2) \]

**Proof.** First we name the individual triples of the union in the function body of \( F^2 \). We name the first triple of results as initial:
\[ (C_1, F^2_1, E^2_1) = (\{ x \mapsto \{ x, x \}, x \leadsto \{ \text{stop} \}, \lambda \mapsto 0) \]

The results of the second, third, fourth, and fifth joined triples corresponding to return, binding, tail call, and non-tail call are named \( (C_{\text{ret}}, F^2_{\text{ret}}, E^2_{\text{ret}}), (C_{\text{bind}}, F^2_{\text{bind}}, E^2_{\text{bind}}), (C_{\text{call}}, F^2_{\text{call}}, E^2_{\text{call}}) \) and \( (C_{\text{inc}}, F^2_{\text{inc}}, E^2_{\text{inc}}) \), respectively. Similarly we name the first result pair in the function body of the CPS analysis as initial:
\[ (Q^2, R^2) = (\{ x \mapsto \{ \text{stop} \}, \lambda \mapsto 0) \]

The results of the second and third joined pair corresponding to call and return are named \( (Q^2_{\text{call}}, R^2_{\text{call}}) \) and \( (Q^2_{\text{ret}}, R^2_{\text{ret}}) \), respectively.

The proof proceeds by verifying five relations:
\[ (C_1, F^2_1, E^2_1) \text{ RANF}_{\text{CPS}} (Q^2_1, R^2_1) \]
\[ (C_{\text{ret}}, F^2_{\text{ret}}, E^2_{\text{ret}}) \text{ RANF}_{\text{CPS}} (Q^2_{\text{ret}}, R^2_{\text{ret}}) \]
\[ (C_{\text{bind}}, F^2_{\text{bind}}, E^2_{\text{bind}}) \text{ RANF}_{\text{CPS}} (Q^2_{\text{bind}}, R^2_{\text{bind}}) \]
\[ (C_{\text{call}}, F^2_{\text{call}}, E^2_{\text{call}}) \text{ RANF}_{\text{CPS}} (Q^2_{\text{call}}, R^2_{\text{call}}) \]
\[ (C_{\text{inc}}, F^2_{\text{inc}}, E^2_{\text{inc}}) \text{ RANF}_{\text{CPS}} (Q^2_{\text{inc}}, R^2_{\text{inc}}) \]

Realizing that the union of related triples and pairs are related we obtain the desired result. \( \square \)

6. Extracting constraints

The resulting analysis may appear complex at first glance. However we can express the analysis in the popular constraint formulation, extracted from the obtained definition. The formulation shown below is in terms of program-specific conditional constraints. Constraints have a (possibly empty) list of preconditions and a conclusion [Falsberg and Schwartzbach, 1995, Gasser et al., 1997]:
\[ \{ u_1 \} \subseteq \text{rhs}_1 \land \ldots \land \{ u_n \} \subseteq \text{rhs}_n \Rightarrow \text{lhs} \subseteq \text{rhs} \]

The constraints operate on the same three domains as the above analysis. Left-hand sides \( \text{lhs} \) can be of the form \( \{ u \}, F^2([s]_\text{var}) \), or \( E^2(x) \), right-hand sides \( \text{rhs} \) can be of the form \( C, F^2([s]_\text{var}) \), or \( E^2(x) \), and singleton elements \( u \) can be of the form \( a, c, \{ \text{fn x} \mapsto a \}, \) or \( [x, a] \). From Fig. 7 we directly read off the following constraints.

**Corollary 5.2.** \( \text{Ifp} T^{\sharp}_{\text{CPS}} \text{ RANF}_{\text{CPS}} \text{ Ifp} F^2_{\text{CPS}} \)

5.5 CPS-ANF equivalence

Again we formally define a relation now relating CPS analysis pairs to ANF analysis triples.

**Definition 5.3.**
\[ (Q^2, R^2) \text{ RANF}_{\text{ANF}} (C, F^2, E^2) \]
\[ \forall e : e \in Q^2 \Rightarrow \text{Var}[e] \in C \land
\[ \text{fn x} \mapsto e \in R^2(k_a) \Rightarrow [x, \text{Var}[e]] \in F^2(a) \land
\[ \text{stop} \in R^2(k_a) \Rightarrow \text{stop} \in F^2(a) \land
\[ \text{fn x, k_a} \mapsto e \in R^2(y) \Rightarrow \text{fn x} \mapsto \text{Var}[e] \in E^2(y) \]

We again need a helper lemma relating the helper functions.

**Lemma 5.3.**
\[ \text{fn x, k_a} \mapsto e \in \mu^2_1(\text{ret}, R^2) \land \mu^2_1(\text{call}, R^2) \text{ RANF}_{\text{ANF}} (C, F^2, E^2) \]
\[ \Rightarrow \text{fn x} \mapsto \text{Var}[e] \in \mu^2_1(\text{ret}, E^2) \]

This relation is also preserved by the transfer functions.

**Theorem 5.3.**
\[ (Q^2, R^2) \text{ RANF}_{\text{CPS}} (C, F^2, E^2) \]
\[ \Rightarrow T^{\sharp}_{\text{CPS}} (Q^2, R^2) \text{ RANF}_{\text{CPS}} F^2_{\text{CPS}} (C, F^2, E^2) \]

**Proof.** The proof follows a similar structure to the earlier proof. \( \square \)
<table>
<thead>
<tr>
<th>i</th>
<th>ANF trace: ( (C_1, F^i, E^i) )</th>
<th>CPS trace: ( (Q^i, R^i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( { \text{let } t = \text{fn } x \Rightarrow x \text{ in let } a_1 = \text{fn c}n \text{ in let } a_2 = \text{fn c}n \text{ in } a_2 } )</td>
<td>( { \text{fn } f \Rightarrow \text{fn } k \text{c}n \text{ in let } a_1 = \text{fn } k \text{c}n \text{ in let } a_2 = \text{fn } k \text{c}n \text{ in } a_2 } )</td>
</tr>
<tr>
<td>1</td>
<td>( C_0 \cup { a_1 = \text{fn c}n \text{ in } a_2 } )</td>
<td>( Q_0 \cup { \text{fn } a_1 = \text{fn } k \text{c}n \text{ in } a_2 } )</td>
</tr>
<tr>
<td>2</td>
<td>( C_1 \cup { x } )</td>
<td>( Q_1 \cup { \text{fn } x \Rightarrow x } )</td>
</tr>
<tr>
<td>3</td>
<td>( C_2 \cup { a_1 = \text{cn}1 } )</td>
<td>( Q_2 \cup { k \text{c}n } )</td>
</tr>
<tr>
<td>4</td>
<td>( C_3 \cup { a_1 = \text{cn}1, \text{cn}2 } )</td>
<td>( Q_3 \cup { k \text{c}n } )</td>
</tr>
<tr>
<td>5</td>
<td>( C_4 \cup { a_1 = \text{cn}1, \text{cn}2 } )</td>
<td>( Q_4 \cup { k, a_2 } )</td>
</tr>
<tr>
<td>6</td>
<td>( C_5 \cup { a_1 = \text{cn}1, \text{cn}2 } )</td>
<td>( Q_5 \cup { k, v } )</td>
</tr>
</tbody>
</table>

Table 1: Analysis traces of \texttt{let } t = \texttt{fn } x \Rightarrow x \text{ in let } a_1 = \text{fn c}n \text{ in let } a_2 = \text{fn c}n \text{ in } a_2 \text{ and its CPS transformed counterpart}

- For the program \( p \):
  \[ \{ p \} \subseteq C \cap \{ [x_r, x_r] \} \subseteq F^i([p]_E) \cap \{ \text{stop} \} \subseteq F^i([x_r]_E) \]

- For each return expression \( t \) and non-tail call \texttt{let } x = \texttt{t0 } t_1 \text{ in } s' \) in \( p \):
  \[ \{ t \} \subseteq C \cap \{ [x, s'] \} \subseteq F^i([t]_E) \Rightarrow \begin{cases} \{ s' \} \subseteq C \\cap \mu_{\text{sym}}(t, E^i) \subseteq E^i(x) \end{cases} \]

- For each let-binding \texttt{let } x = \texttt{t } t_1 \text{ in } s \text{ in } p:
  \[ \{ \text{let } x = \text{t } t_1 \text{ in } s \} \subseteq C \Rightarrow \begin{cases} \{ s \} \subseteq C \\cap \mu_{\text{sym}}(t, E^i) \subseteq E^i(x) \end{cases} \]

- For each tail call \texttt{t0 } t_1 \text{ and function } \texttt{fn } x \Rightarrow s' \text{ in } p:
  \[ \{ \text{fn } x \Rightarrow s' \} \subseteq \mu_{\text{sym}}(t_0, E^i) \Rightarrow \begin{cases} \{ s' \} \subseteq C \\cap \mu_{\text{sym}}(t_0, E^i) \subseteq E^i(x) \end{cases} \]

- For each non-tail call \texttt{let } x = \texttt{t0 } t_1 \text{ in } s \text{ and function } \texttt{fn } x \Rightarrow s' \text{ in } p:
  \[ \{ \text{let } x = \text{t0 } t_1 \text{ in } s \} \subseteq C \cap \{ [x, s'] \} \subseteq \mu_{\text{sym}}(t_0, E^i) \Rightarrow \begin{cases} \{ s' \} \subseteq C \\cap \mu_{\text{sym}}(t_0, E^i) \subseteq E^i(x) \end{cases} \]

where we partially evaluate the helper function, i.e., interpret the helper function symbolically at constraint-generation time, to generate a lookup for variables, and a singleton for constants and lambda expressions. The definition of the symbolic helper function otherwise coincides with the abstract helper function \( \mu^f \).

We may generate constraints \( \{ [f n x \Rightarrow s'] \} \subseteq \{ [f n y \Rightarrow s'] \} \) from each constraint, and removing constraints containing vacuously false preconditions \( \{ [f n x \Rightarrow s'] \} \subseteq \{ w^f \} \), where \( [f n y \Rightarrow s'] \neq w^f \).

The resulting constraint system is formally equivalent to the control flow analysis in the sense that all solutions yield correct control flow information and that the best (smallest) solution of
the constraints is as precise as the information computed by the analysis. More formally:

**Theorem 6.1.** A solution to the CFA constraints of program \( p \) is a safe approximation of the least fixpoint of the analysis function \( F^2 \) induced by \( p \). Furthermore, the least solution to the CFA constraints is equal to the least fixpoint of \( F^2 \).

Implemented naively, a single constraint may take \( O(n) \) space alone. However by using pointers or by labelling each sub-expression and using the pointer/label instead of the sub-expression itself, a single constraint takes only constant space. By linearly determining a representative for each sub-expression, by generating \( O(n^2) \) constraints, linear post-processing, and iteratively solving them using a well-known algorithm [Palsberg and Schwartzbach, 1995, Nielson et al., 1999], we can compute the analysis in worst-case \( O(n^2) \) time.

The extracted constraints bear similarities to existing constraint-based analyses in the literature. Consider, e.g., calls \( t_0 \ t_1 \), which usually gives rise to two conditional constraints [Palsberg, 1995, Nielson et al., 1999]:

1. \( \{ \text{fn } \ x \rightarrow \ s' \} \subseteq \hat{C}(t_0) \Rightarrow \hat{C}(t_1) \subseteq \hat{E}(x) \)
2. \( \{ \text{fn } \ x \rightarrow \ s' \} \subseteq \hat{C}(t_0) \Rightarrow \hat{C}(s') \subseteq \hat{C}(t_2 \ t_1) \).

The first constraint resembles our third constraint for tail calls. The second “return constraint” differs in that it has a inside-out (or caller-restore) nature, i.e., propagation of return-flow from the function body is handled at the call site. The extracted reachability constraints are similar to Gasser et al. [1997] (modulo an isomorphic encoding \( \varphi(C) \cong C \rightarrow \varphi(\text{on}) \) of powersets).

### 7. Conclusion

We have presented a control-flow analysis determining interprocedural control-flow of both calls and returns for a direct-style language. Existing CFAs have focused on analysing which functions are called at a given call site. In contrast, the systematic derivation of our CFA has lead to an analysis that provides extra information about where a function returns to at no additional cost. In the presence of tail-call optimization, such information enables the creation of more precise call graphs.

The analysis was developed systematically using Galois connection-based abstract interpretation of a standard operational semantics for that language: the \( C_{\text{FL}} \) abstract machine of Flanagan et al. In addition to being more principled, such a formulation of the analysis is pedagogically pleasing since monomorphism of the analysis is made explicit through two Galois connections: one literally merges all bindings to the same variable and one merges all calling contexts of the same function.

The analysis has been shown to provide a result equivalent to what can be obtained by first CPS transforming the program and then running a control-flow analysis derived from a CPS-based operational semantics. This extends previous results obtained by Damian and Danvy, and Palsberg and Wand. The close correspondence between the way that the analyses operate (as illustrated by the analysis trace in Table 1) leads us to conjecture that such equivalence results can be obtained for other CFAs derived using abstract interpretation.

The functional, derived by abstract interpretation, that defines the analysis may appear rather complex at first glance. As a final result, we have shown how to extract from the analysis an equivalent constraint-based formulation expressed in terms of the more familiar conditional constraints. Nevertheless, we stress that the derived functional can be used directly to implement the analysis. We have developed a prototype implementation of the resulting analysis in OCaml.²

The analysis has been developed for a minimalistic functional language in order to be able to focus on the abstraction of the control structure induced by function calls and returns. An obvious extension is to enrich the language with numerical operators and study how our Galois connections interact with abstractions such as the interval or polyhedral abstraction of numerical entities.

The calculations involved in the derivation of a CFA are lengthy and would benefit enormously from some form of machine support. **Certified abstract interpretation** [Pichardie, 2005, Cachera et al., 2005] has so far focused on proving the correctness of the analysis inside a proof assistant by using the concretization (\( \gamma \)) component of the Galois connection to prove the correctness of an already defined analysis. Further work should investigate whether proof assistants such as Coq are suitable for conducting the kind of reasoning developed in this paper in a machine-checkable way.

Acknowledgments

The authors thank Matthew Fluet, Amr Sabry, Mitchell Wand, Daniel Damian, Olivier Danvy, and the anonymous referees for comments on earlier versions. Part of this work was done with the support of the Carlsberg Foundation.

References


P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In B. K. Rosen, editor, *Proc. of the Sixth Annual ACM Sym-

² available at [http://www.brics.dk/~jml/ABF-CFA/](http://www.brics.dk/~jml/ABF-CFA/)


### A. Underlying mathematical material

This section is based on known material [Cousot and Cousot, 1979, Cousot, 1981, Cousot and Cousot, 1992b, 1994, Davey and Priestley, 2002].

A complete lattice is a partially ordered set \( (\mathcal{C}, \sqsubseteq, \top, \bot) \) (poset), such that the least upper bound \( \sqcup \) and the greatest lower bound \( \sqcap \) exists for every subset \( S \subseteq \mathcal{C} \). \( \bot = \bot \mathcal{C} \) denotes the infimum of \( \mathcal{C} \) and \( \top = \top \mathcal{C} \) denotes the supremum of \( \mathcal{C} \). The set of total functions \( D \to \mathcal{C} \), whose domain is a complete lattice \( \mathcal{L} \in \mathcal{C} \), is itself a complete lattice \( \langle D \to \mathcal{C}, \sqsubseteq, \bot, \top \rangle \) under the pointwise ordering \( f \sqsubseteq f' \iff \forall x.f(x) \sqsubseteq f'(x) \) with bottom, top, join, and meet extended similarly. The powersets \( \mathcal{P}(S) \) of a set \( S \) ordered by set inclusion is a complete lattice \( \langle \mathcal{P}(S), \subseteq \rangle \).

A Galois connection is a pair of functions \( \alpha, \gamma \) between two posets \( \langle \mathcal{C}, \sqsubseteq \rangle \) and \( \langle \mathcal{A}, \sqsubseteq \rangle \) such that for all \( a \in A \in C : \alpha(c) \leq a \iff c \sqsubseteq \gamma(a) \). Equivalently a Galois connection can be defined as a pair of functions satisfying (a) \( \alpha \) and \( \gamma \) are monotone, (b) \( \alpha \circ \gamma \) is reductive, and (c) \( \gamma \circ \alpha \) is extensive. Galois connections are type sets \( \langle \mathcal{C}, \subseteq \rangle \). We omit the orderings when they are clear from the context. For a Galois connection between two complete lattices \( \alpha \) is a complete join-morphism (CJM) and \( \gamma \) is a complete meet morphism. The composition of two Galois connections \( \langle \mathcal{C}, \subseteq \rangle \circ \alpha : \mathcal{B} \to \mathcal{A} \) and \( \beta : \mathcal{A} \to \mathcal{A} \) is itself a Galois connection \( \langle \mathcal{C}, \subseteq \rangle \circ \alpha : \mathcal{B} \to \mathcal{A} \). Galois connections in which \( \alpha \) is surjective (or equivalently \( \gamma \) is injective) are type sets \( \langle \mathcal{C}, \subseteq \rangle \). Galois connections in which \( \gamma \) is surjective (or equivalently \( \alpha \) is injective) are type sets \( \langle \mathcal{C}, \subseteq \rangle \).

When both \( \alpha \) and \( \gamma \) are surjective, the two domains are isomorphic.

A(n upper) closure operator \( \mu \) is map \( \mu : \mathcal{S} \to \mathcal{S} \) on a poset \( \langle \mathcal{S}, \subseteq \rangle \), that is (a) monotone: (for all \( s, s' \in \mathcal{S} : s \subseteq s' \implies \mu(s) \subseteq \mu(s') \)), (b) extensive (for all \( s \in \mathcal{S} : s \subseteq \mu(s) \)), and (c) idempotent (for all \( s \in \mathcal{S} : \mu(s) = \mu(\mu(s)) \)). A closure operator \( \mu \) induces a Galois connection \( \langle \mathcal{S}, \subseteq \rangle \circ \mu : \mathcal{S} \to \mathcal{S} \), writing \( \mu(s) \) for \( \{ \mu(s) \mid s \in \mathcal{S} \} \) and \( 1 \) for the identity function. Furthermore the image of a complete lattice \( \langle \mathcal{C}, \subseteq, \bot, \top, \mu \rangle \) by an upper closure operator is itself a complete lattice \( \langle \mathcal{C}, \subseteq, \mu, \bot, \top, \mu.1 \rangle \).