HAL
open science

Control-flow analysis of function call and returns by abstract interpretation

Jan Midtgaard, Thomas Jensen

To cite this version:

Jan Midtgaard, Thomas Jensen. Control-flow analysis of function call and returns by abstract interpretation. ACM International Conference on Functional Programming, Sep 2009, Edinburgh, United Kingdom. 10.1145/1596550.1596592 . inria-00574944

HAL Id: inria-00574944

https://inria.hal.science/inria-00574944v1

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 interprocedural 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 function calls return across optimized tail calls. The analysis is systematically calculated by abstract interpretation of the stack-based C_aEK 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 abstract interpretation of a stack-less CPS machine, and *2)* extract an equivalent constraint-based formulation, thereby providing a rational 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

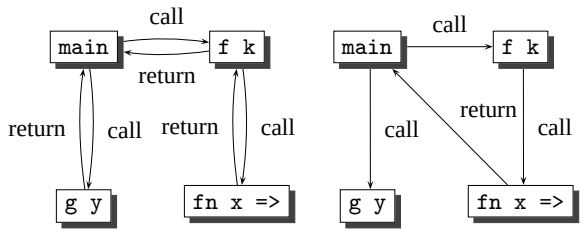
Keywords Control flow analysis, abstract interpretation, tail-call optimization, continuation-passing style, direct style, constraint-based analysis

1. Introduction

The control flow of a functional program is expressed in terms of function calls and returns. As a result, iteration in functional programs 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*. Consequently functions do not necessarily return control to their caller. Control-flow analysis (CFA) has long been a staple of program optimization and verification. Surprisingly, research on control-flow analysis has focused on calls: A textbook CFA *"will determine where the flow of control may be transferred to in the case [...] of a function application."* [Nielson et al., 1999]. Our systematic approximation of a known operational semantics leads to a CFA 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.

```
let g z = z in
 let f k = if b then k 1 else k 2 in
  let y = f (fn x => x) in
     g y
```

(a) Example program



(b) Call-return call graph (c) Optimized call graph

Figure 1: The corresponding call graphs

Consider the example program in Fig. 1(a). The program contains three functions: two named function `g` and `f` and an anonymous function `fn x => x`. A standard direct-style CFA can determine that the applications of `k` in each branch of the conditional will call the anonymous function `fn x => x` at run time. Building 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 addition 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_aEK 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 itself 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 simulation between the two, reducing the proof to a fixpoint induction over the abstract interpretations.

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.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08. . . $10.00

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 : exists $p, C_1(p) \nsim C_2(cps(p))$

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

exists analysis C : for all $p, C(p) \sim C(cps(p))$

Present paper, Theorem 5.1:

exists analyses C_1, C_2 : for all $p, C_1(p) \sim C_2(cps(p))$

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 Nielson and Nielson [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 [Midtgaard 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 Nielson and Nielson [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 [Midtgaard 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_aEK 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

$$P \ni \mathtt{p} ::= \mathtt{s} \qquad \text{(programs)}$$
$$T \ni \mathtt{t} ::= c \mid \mathtt{x} \mid \mathtt{fn\ x\ =>\ s} \qquad \text{(trivial expressions)}$$
$$C \ni \mathtt{s} ::= \mathtt{t} \mid \mathtt{let\ x{=}t\ in\ s} \mid \mathtt{t_0\,t_1} \mid \mathtt{let\ x{=}t_0\,t_1\ in\ s} \qquad \text{(serious expressions)}$$

(a) ANF grammar

$$\mathit{Val} \ni w ::= c \mid [\mathtt{fn\ x\ =>\ s}, e]$$
$$\mathit{Env} \ni e ::= \bullet \mid e[\mathtt{x} \mapsto w]$$
$$K \ni k ::= \mathtt{stop} \mid [\mathtt{x}, \mathtt{s}, e] :: k$$

(b) Values, environments, and stacks

$$\mu : T \times \mathit{Env} \rightharpoonup \mathit{Val}$$
$$\mu(c, e) = c$$
$$\mu(\mathtt{x}, e) = e(\mathtt{x})$$
$$\mu(\mathtt{fn\ x\ =>\ s}, e) = [\mathtt{fn\ x\ =>\ s}, e]$$

(c) Helper function

$$\langle \mathtt{t}, e, [\mathtt{x}, \mathtt{s}', e'] :: k' \rangle \longrightarrow \langle \mathtt{s}', e'[\mathtt{x} \mapsto \mu(\mathtt{t}, e)], k' \rangle$$
$$\langle \mathtt{let\ x{=}t\ in\ s}, e, k \rangle \longrightarrow \langle \mathtt{s}, e[\mathtt{x} \mapsto \mu(\mathtt{t}, e)], k \rangle$$
$$\langle \mathtt{t_0\,t_1}, e, k \rangle \longrightarrow \langle \mathtt{s}', e'[\mathtt{x} \mapsto w], k \rangle \quad \text{if } [\mathtt{fn\ x\ =>\ s}', e'] = \mu(\mathtt{t_0}, e) \text{ and } w = \mu(\mathtt{t_1}, e)$$
$$\langle \mathtt{let\ x{=}t_0\,t_1\ in\ s}, e, k \rangle \longrightarrow \langle \mathtt{s}', e'[\mathtt{y} \mapsto w], [\mathtt{x}, \mathtt{s}, e] :: k \rangle \quad \text{if } [\mathtt{fn\ y\ =>\ s}', e'] = \mu(\mathtt{t_0}, e) \text{ and } w = \mu(\mathtt{t_1}, e)$$

(d) Machine transitions

$$\mathit{eval}(\mathtt{p}) = w \text{ iff } \langle \mathtt{p}, \bullet, [\mathtt{x_r}, \mathtt{x_r}, \bullet] :: \mathtt{stop} \rangle \longrightarrow^* \langle \mathtt{x_r}, \bullet[\mathtt{x_r} \mapsto w], \mathtt{stop} \rangle$$

(e) Machine evaluation

Figure 2: The C_aEK abstract machine

environment and a control stack. The control stack is composed of elements ("stack frames") of the form $[\mathtt{x}, \mathtt{s}, e]$ where $\mathtt{x}$ is the variable receiving the return value w of the current function call, and $\mathtt{s}$ is a serious expression whose evaluation in the environment $e[\mathtt{x} \mapsto w]$ represents the rest of the computation in that stack frame. The empty stack is represented by $\mathtt{stop}$. The machine has a helper function μ 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 $\mathtt{x_r}$ 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. Given a transition function T defined as: $T(\Sigma) = I \cup \{\sigma \mid \exists \sigma' \in \Sigma : \sigma' \to \sigma\}$, we can compute the reachable states of T as the least fixed-point $\mathrm{lfp}\,T$ 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 α and γ (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 $T^\sharp \circ \alpha$, from which the analysis function $T^\sharp$ 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_aEK collecting semantics

As our collecting semantics we consider the reachable states of the C_aEK machine, expressed as the least fixed point $\mathrm{lfp}\,F$ of the following transition function.

$$F : \wp(C \times \mathit{Env} \times K) \to \wp(C \times \mathit{Env} \times K)$$
$$F(S) = I_\mathtt{p} \cup \{s \mid \exists s' \in S : s' \longrightarrow s\}$$
$$\text{where } I_\mathtt{p} = \{\langle \mathtt{p}, \bullet, [\mathtt{x_r}, \mathtt{x_r}, \bullet] :: \mathtt{stop} \rangle\}$$

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

Lemma 4.1. $\forall \mathtt{t}, e : \{\mu(\mathtt{t}, e)\} = \mu_c(\mathtt{t}, \{e\})$

The equivalence of the two helper functions follow straightforwardly. This lemma enables us to express an equivalent collecting semantics based on μ_c, which appears in Fig. 3. The equivalence of F and F^c 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

$$\mu_c : T \times \wp(Env) \to \wp(Val)$$
$$\mu_c(c, E) = \{c\}$$
$$\mu_c(\mathtt{x}, E) = \{w \mid \exists e \in E : w = e(\mathtt{x})\}$$
$$\mu_c(\mathtt{fn\ x\ =>\ s}, E) = \{[\mathtt{fn\ x\ =>\ s}, e] \mid \exists e \in E\}$$

(a) Helper function

$$F^c : \wp(C \times Env \times K) \to \wp(C \times Env \times K)$$
$$\begin{aligned} F^c(S) = I_{\mathrm{p}} & \\ & \cup \bigcup_{\substack{\langle \mathtt{t}, e, [\mathtt{x}, \mathtt{s}', e'] :: k' \rangle \in S \\ w \in \mu_c(\mathtt{t}, \{e\})}} \{\langle \mathtt{s}', e'[\mathtt{x} \mapsto w], k' \rangle\} \\ & \cup \bigcup_{\substack{\langle \mathtt{let\ x=t\ in\ s}, e, k \rangle \in S \\ w \in \mu_c(\mathtt{t}, \{e\})}} \{\langle \mathtt{s}, e[\mathtt{x} \mapsto w], k \rangle\} \\ & \cup \bigcup_{\substack{\langle \mathtt{t}_0\ \mathtt{t}_1, e, k \rangle \in S \\ [\mathtt{fn\ x\ =>\ s}', e'] \in \mu_c(\mathtt{t}_0, \{e\}) \\ w \in \mu_c(\mathtt{t}_1, \{e\})}} \{\langle \mathtt{s}', e'[\mathtt{x} \mapsto w], k \rangle\} \\ & \cup \bigcup_{\substack{\langle \mathtt{let\ x=t_0\ t_1\ in\ s}, e, k \rangle \in S \\ [\mathtt{fn\ y\ =>\ s}', e'] \in \mu_c(\mathtt{t}_0, \{e\}) \\ w \in \mu_c(\mathtt{t}_1, \{e\})}} \{\langle \mathtt{s}', e'[\mathtt{y} \mapsto w], [\mathtt{x}, \mathtt{s}, e] :: k \rangle\} \end{aligned}$$

(b) Transition function

Figure 3: Collecting semantics

$$\begin{array}{ccc} \wp(C \times Env \times K) & \text{coll. sem.} & F^c \\ \alpha_\times \downarrow \uparrow \gamma_\times & & \\ \wp(C) \times \wp(C \times K) \times \wp(Env) & - & F^\times \\ \rho \downarrow \uparrow 1 & & \\ \rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) & - & F^\rho \\ \alpha_\otimes \downarrow \uparrow \gamma_\otimes & & \\ \wp(C) \times (C/_\equiv \to \wp(K^\sharp)) \times Env^\sharp & \text{0-CFA} & F^\sharp \end{array}$$

Figure 4: Overview of abstraction

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, we 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.

$$\wp(C \times Env \times K) \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} \wp(C) \times \wp(C \times K) \times \wp(Env)$$
$$\alpha_\times(S) = \langle \pi_1 S, \{\langle \mathtt{s}, k \rangle \mid \exists e : \langle \mathtt{s}, e, k \rangle \in S\}, \pi_2 S \rangle$$
$$\gamma_\times(\langle C, F, E \rangle) = \{\langle \mathtt{s}, e, k \rangle \mid \mathtt{s} \in C \ \wedge\ \langle \mathtt{s}, k \rangle \in F \ \wedge\ e \in E\}$$

Lemma 4.2. $\alpha_\times$, $\gamma_\times$ is a Galois connection.

The above Galois connection and the proof hereof closely resembles the independent attributes abstraction, which is a known Galois connection. We use the notation $\cup_\times$ and $\subseteq_\times$ for the componentwise join and componentwise inclusion of triples.

As traditional [Cousot and Cousot, 1979, 1992a, 1994], we will assume that the abstract product domains throughout this article have been *reduced*, i.e., all triples $\langle A, B, C \rangle$ with a bottom component ($A = \bot_a \ \vee\ B = \bot_b \ \vee\ C = \bot_c$) have been eliminated and replaced by a single bottom element $\langle \bot_a, \bot_b, \bot_c \rangle$.

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

Theorem 4.1.

$$\forall C, F, E : \alpha_\times(F^c(\gamma_\times(\langle C, F, E \rangle))) = F^\times(\langle C, F, E \rangle)$$

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 $\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k \rangle$ we also want to retain that $\mathtt{s}'$ 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 component x_i of a tuple $\langle x_0, \ldots, x_n \rangle$ is a *constituent* of the tuple, written $\langle x_0, \ldots, x_n \rangle \succ x_i$. For a partial function[1] $f = [x_0 \mapsto w_0, \ldots, x_n \mapsto w_n]$, 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: $\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k \rangle \gtrdot \langle \mathtt{s}', k \rangle$. We write $\gtrdot^*$ for the reflexive, transitive closure of the expression-stack pair ordering.

Next, we consider an operator ρ, defined in terms of the constituent relation and the expression-stack pair ordering. The operator ρ 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.

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

We need to relate the expression-stack ordering to the constituent relation. By case analysis one can prove that $\forall \langle \mathtt{s}, k \rangle, \langle \mathtt{s}', k' \rangle : \langle \mathtt{s}, k \rangle \gtrdot \langle \mathtt{s}', k' \rangle \implies k \succ k'$. By structural induction (on the stack component) it now follows that $\forall \langle \mathtt{s}, k \rangle, \langle \mathtt{s}', k' \rangle : \langle \mathtt{s}, k \rangle \gtrdot^* \langle \mathtt{s}', k' \rangle \implies k \succ^* k'$. Based on these results we can verify that ρ is a closure operator and formulate an abstraction on the triples:

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

[1] Milner and Tofte define the constituent relation for finite functions.

$$F^{\times} : \wp(C) \times \wp(C \times K) \times \wp(Env) \to \wp(C) \times \wp(C \times K) \times \wp(Env)$$

$$\begin{aligned}
F^{\times}(\langle C, F, E\rangle) = \langle \{\mathtt{p}\}, \{\langle \mathtt{p}, [\mathtt{x_r}, \mathtt{x_r}, \bullet] :: \mathtt{stop}\rangle\}, \{\bullet\}\rangle \\
\cup_{\times} \bigcup_{\substack{\langle\{\mathtt{t}\},\{\langle \mathtt{t},[\mathtt{x},\mathtt{s}',e']::k'\rangle\},\{e\}\rangle \subseteq_{\times} \langle C,F,E\rangle \\ w \in \mu_c(\mathtt{t},\{e\})}}{}_{\times} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k'\rangle\}, \{e'[\mathtt{x} \mapsto w]\}\rangle \\
\cup_{\times} \bigcup_{\substack{\langle\{\mathtt{let\ x{=}t\ in\ s}\},\{\langle \mathtt{let\ x{=}t\ in\ s},k\rangle\},\{e\}\rangle \subseteq_{\times} \langle C,F,E\rangle \\ w \in \mu_c(\mathtt{t},\{e\})}}{}_{\times} \langle \{\mathtt{s}\}, \{\langle \mathtt{s}, k\rangle\}, \{e[\mathtt{x} \mapsto w]\}\rangle \\
\cup_{\times} \bigcup_{\substack{\langle\{\mathtt{t_0\,t_1}\},\{\langle \mathtt{t_0\,t_1},k\rangle\},\{e\}\rangle \subseteq_{\times} \langle C,F,E\rangle \\ [\mathtt{fn\ x\ =>\ s}',e'] \in \mu_c(\mathtt{t_0},\{e\}) \\ w \in \mu_c(\mathtt{t_1},\{e\})}}{}_{\times} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k\rangle\}, \{e'[\mathtt{x} \mapsto w]\}\rangle \\
\cup_{\times} \bigcup_{\substack{\langle\{\mathtt{let\ x{=}t_0\,t_1\ in\ s}\},\{\langle \mathtt{let\ x{=}t_0\,t_1\ in\ s},k\rangle\},\{e\}\rangle \subseteq_{\times} \langle C,F,E\rangle \\ [\mathtt{fn\ y\ =>\ s}',e'] \in \mu_c(\mathtt{t_0},\{e\}) \\ w \in \mu_c(\mathtt{t_1},\{e\})}}{}_{\times} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', [\mathtt{x}, \mathtt{s}, e] :: k\rangle\}, \{e'[\mathtt{y} \mapsto w]\}\rangle
\end{aligned}$$

Figure 5: Abstract transition function

We use the notation $\cup_\rho$ for the join operation $\lambda X.\,\rho(\cup_\times X)$ on the closure operator-induced complete lattice. First observe that in our case:

$$\cup_\rho = \lambda X.\,\rho(\bigcup_i{}_{\times} X_i) = \lambda X.\,\bigcup_i{}_{\times}\, \rho(X_i) = \lambda X.\,\bigcup_i{}_{\times}\, X_i = \cup_\times$$

Based on the closure operator-based Galois connection, we can calculate a new intermediate transfer function F^ρ. 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 $\langle \mathtt{x_r}, \mathtt{stop}\rangle$, 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. $\forall C, F, E : \rho \circ F^{\times} \circ 1(\langle C, F, E\rangle) = F^\rho(\langle C, F, E\rangle)$

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 stacks and an elementwise operator $@ : C \times K \to C \times K^\sharp$ operating on expression-stack pairs.

$$K^\sharp \ni k^\sharp ::= \mathtt{stop} \mid [\mathtt{x}, \mathtt{s}]$$
$$@(\langle \mathtt{s}, \mathtt{stop}\rangle) = \langle \mathtt{s}, \mathtt{stop}\rangle$$
$$@(\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle) = \langle \mathtt{s}, [\mathtt{x}, \mathtt{s}']\rangle$$

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

Elementwise abstraction [Cousot and Cousot, 1997]: A given elementwise operator $@ : C \to A$ induces a Galois connection:

$$\langle \wp(C); \subseteq\rangle \underset{\alpha_@}{\overset{\gamma_@}{\leftrightarrows}} \langle \wp(A); \subseteq\rangle$$
$$\alpha_@(P) = \{@(p) \mid p \in P\} \qquad \gamma_@(Q) = \{p \mid @(p) \in Q\}$$

Notice how some expressions share the same return point (read: same stack): the expressions $\mathtt{let\ x{=}t\ in\ s}$ and $\mathtt{s}$ share the same return point, and $\mathtt{let\ x{=}t_0\,t_1\ in\ s}$ and $\mathtt{s}$ share the same return point. In order to eliminate such redundancy we define an equivalence relation on serious expressions grouping together expressions sharing the same return point. We define the smallest equivalence relation $\equiv$ satisfying:

$$\mathtt{let\ x{=}t\ in\ s} \equiv \mathtt{s}$$
$$\mathtt{let\ x{=}t_0\,t_1\ in\ s} \equiv \mathtt{s}$$

Based hereon we define a second elementwise operator $@' : C \times K^\sharp \to C/_\equiv \times K^\sharp$ mapping the first component of an expression-stack pair to a representative of its corresponding equivalence class:

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

We can choose the outermost expression as a representative for each equivalence class by a linear top-down traversal of the input program.

Pointwise coding of a relation [Cousot and Cousot, 1994]: A relation can be isomorphically encoded as a set-valued function by a Galois connection:

$$\langle \wp(A \times B); \subseteq\rangle \underset{\alpha_\omega}{\overset{\gamma_\omega}{\leftrightarrows}} \langle A \to \wp(B); \dot{\subseteq}\rangle$$
$$\alpha_\omega(r) = \lambda a.\,\{b \mid \langle a, b\rangle \in r\} \qquad \gamma_\omega(f) = \{\langle a, b\rangle \mid b \in f(a)\}$$

By composing the three above Galois connections we obtain our abstraction of the expression-stack relation:

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

where $\alpha_{st} = \alpha_\omega \circ \alpha_{@'} \circ \alpha_@ = \lambda F.\,\dot{\bigcup}_{\langle \mathtt{s}, k\rangle \in F} \alpha_\omega(\{@' \circ @(\langle \mathtt{s}, k\rangle)\})$ and $\gamma_{st} = \gamma_@ \circ \gamma_{@'} \circ \gamma_\omega$. We can now prove a lemma relating the concrete and abstract expression-stack relations.

Lemma 4.3. Control stack and saved environments
Let $\langle C, F, E\rangle \in \rho(\wp(C) \times \wp(C \times K) \times \wp(Env))$ be given.

$$\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle \in F \implies e \in E \wedge \{\langle \mathtt{s}', k\rangle\} \subseteq F$$
$$\wedge\ \{[\mathtt{x}, \mathtt{s}']\} \subseteq \alpha_{st}(F)([\mathtt{s}]_\equiv)$$

Proof. The first half follows from the assumptions. The second half follows from monotonicity of α_{st}, and the definitions of α_{st}, $\dot{\cup}$, $@$, $@'$, α_ω, and $\dot{\subseteq}$. □

4.4 Abstracting environments

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

$$F^\rho : \rho(\wp(C) \times \wp(C \times K) \times \wp(Env)) \to \rho(\wp(C) \times \wp(C \times K) \times \wp(Env))$$

$$\begin{aligned}
F^\rho(\langle C, F, E\rangle) = {} & \langle \{\mathtt{p}\}, \{\langle \mathtt{p}, [\mathtt{x_r}, \mathtt{x_r}, \bullet] :: \mathtt{stop}\rangle, \langle \mathtt{x_r}, \mathtt{stop}\rangle\}, \{\bullet\}\rangle \\
& \cup_\times \bigcup_{\substack{\langle\{\mathtt{t}\},\{\langle \mathtt{t},[\mathtt{x},\mathtt{s}',e']::k'\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ w \in \mu_c(\mathtt{t},\{e\})}}_\times \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k'\rangle\}, \{e'[\mathtt{x} \mapsto w]\}\rangle) \\
& \cup_\times \bigcup_{\substack{\langle\{\mathtt{let\ x{=}t\ in\ s}\},\{\langle \mathtt{let\ x{=}t\ in\ s},k\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ w \in \mu_c(\mathtt{t},\{e\})}}_\times \rho(\langle \{\mathtt{s}\}, \{\langle \mathtt{s}, k\rangle\}, \{e[\mathtt{x} \mapsto w]\}\rangle) \\
& \cup_\times \bigcup_{\substack{\langle\{\mathtt{t_0\ t_1}\},\{\langle \mathtt{t_0\ t_1},k\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ [\mathtt{fn\ x\ =>\ s}',e'] \in \mu_c(\mathtt{t_0},\{e\}) \\ w \in \mu_c(\mathtt{t_1},\{e\})}}_\times \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k\rangle\}, \{e'[\mathtt{x} \mapsto w]\}\rangle) \\
& \cup_\times \bigcup_{\substack{\langle\{\mathtt{let\ x{=}t_0\ t_1\ in\ s}\},\{\langle \mathtt{let\ x{=}t_0\ t_1\ in\ s},k\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ [\mathtt{fn\ y\ =>\ s}',e'] \in \mu_c(\mathtt{t_0},\{e\}) \\ w \in \mu_c(\mathtt{t_1},\{e\})}}_\times \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', [\mathtt{x}, \mathtt{s}, e] :: k\rangle\}, \{e'[\mathtt{y} \mapsto w]\}\rangle)
\end{aligned}$$

Figure 6: The second abstract transition function

operator $@ : Val \to Val^\sharp$ mapping concrete to abstract values.

$$Val^\sharp \ni w^\sharp ::= c \mid [\mathtt{fn\ x\ =>\ s}]$$
$$@(c) = c$$
$$@([\mathtt{fn\ x\ =>\ s}, e]) = [\mathtt{fn\ x\ =>\ s}]$$

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 $\langle \wp(C); \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle C^\sharp; \sqsubseteq\rangle$ induces a Galois connection on a set of functions:

$$\langle \wp(D \to C); \subseteq\rangle \underset{\alpha_\Pi}{\overset{\gamma_\Pi}{\leftrightarrows}} \langle D \to C^\sharp; \dot{\sqsubseteq}\rangle$$
$$\alpha_\Pi(F) = \lambda d.\, \alpha(\{f(d) \mid f \in F\})$$
$$\gamma_\Pi(A) = \{f \mid \forall d : f(d) \in \gamma(A(d))\}$$

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:

$$\langle \wp(C); \subseteq\rangle \underset{\alpha_\subset}{\overset{\gamma_\subset}{\leftrightarrows}} \langle \wp(A); \subseteq\rangle$$
$$\alpha_\subset(X) = X \cap A \qquad \gamma_\subset(Y) = Y \cup (C \setminus A)$$

A standard trick is to think of partial functions $r : D \rightharpoonup C$ as total functions $r_\perp : D \to (C \cup \perp)$ where $\perp \sqsubseteq \perp \sqsubseteq c$, for all $c \in C$. Consider environments $e \in Var \rightharpoonup Val$ to be total functions $Var \to (Val \cup \perp)$ using this idea. In this context the bottom element $\perp$ will denote variable lookup failure. Now compose a subset abstraction $\wp(Val \cup \perp) \underset{\alpha_\subset}{\overset{\gamma_\subset}{\leftrightarrows}} \wp(Val)$ 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: $\wp(Env) \underset{\alpha_\Pi}{\overset{\gamma_\Pi}{\leftrightarrows}} Var \to \wp(Val^\sharp)$. 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 α's" under the function definition, and reading off a resulting abstract definition.

Lemma 4.4. $\forall \mathtt{t}, E : \alpha_@(\mu_c(\mathtt{t}, E)) = \mu^\sharp(\mathtt{t}, \alpha_\Pi(E))$

The resulting helper function $\mu^\sharp : T \times Env^\sharp \to \wp(Val^\sharp)$ reads:

$$\mu^\sharp(c, E^\sharp) = \{c\}$$
$$\mu^\sharp(\mathtt{x}, E^\sharp) = E^\sharp(\mathtt{x})$$
$$\mu^\sharp(\mathtt{fn\ x\ =>\ s}, E^\sharp) = \{[\mathtt{fn\ x\ =>\ s}]\}$$

where we write $Env^\sharp$ as shorthand for $Var \to \wp(Val^\sharp)$. We shall need a lemma relating the two helper function definitions on closed environments.

Lemma 4.5. Helper function on closed environments (1)
Let $\langle C, F, E\rangle \in \rho(\wp(C) \times \wp(C \times K) \times \wp(Env))$ be given.

$$\{[\mathtt{fn\ x\ =>\ s}, e]\} \subseteq \mu_c(\mathtt{t}, E) \implies e \in E$$
$$\wedge\ \{[\mathtt{fn\ x\ =>\ s}]\} \subseteq \mu^\sharp(\mathtt{t}, \alpha_\Pi(E))$$

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 $\langle C, F, E\rangle \in \rho(\wp(C) \times \wp(C \times K) \times \wp(Env))$ be given.

$$\{w\} \subseteq \mu_c(\mathtt{t}, E) \wedge w \succ^* e'' \implies e'' \in E$$

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: $\wp(C_i) \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} A_i$ for $i \in \{1, \ldots, n\}$, their componentwise composition induces a Galois connection on tuples:

$$\langle \wp(C_1) \times \ldots \times \wp(C_n); \subseteq_\times\rangle \underset{\alpha_\otimes}{\overset{\gamma_\otimes}{\leftrightarrows}} \langle A_1 \times \ldots \times A_n; \subseteq_\otimes\rangle$$
$$\alpha_\otimes(\langle X_1, \ldots, X_n\rangle) = \langle \alpha_1(X_1), \ldots, \alpha_n(X_n)\rangle$$
$$\gamma_\otimes(\langle x_1, \ldots, x_n\rangle) = \langle \gamma_1(x_1), \ldots, \gamma_n(x_n)\rangle$$

We write $\cup_\otimes$ and $\subseteq_\otimes$ for componentwise join and inclusion, respectively.

For the set of expressions $\wp(C)$ we use the identity abstraction consisting of two identity functions. For the expression-stack

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

$$\begin{aligned}
F^\sharp_{\mathtt{p}}(\langle C, F^\sharp, E^\sharp\rangle) = \langle &\{\mathtt{p}\}, [[\mathtt{p}]_\equiv \mapsto \{[\mathtt{x_r}, \mathtt{x_r}]\}, [\mathtt{x_r}]_\equiv \mapsto \{\mathtt{stop}\}], \lambda_.\emptyset\rangle \\
&\cup_\otimes \bigcup_{\substack{\{\mathtt{t}\}\subseteq C \\ \{[\mathtt{x},\mathtt{s}']\}\subseteq F^\sharp([\mathtt{t}]_\equiv)}}^{\otimes} \langle\{\mathtt{s}'\}, F^\sharp, E^\sharp \dot\cup [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle \\
&\cup_\otimes \bigcup_{\{\mathtt{let\ x{=}t\ in\ s}\}\subseteq C}^{\otimes} \langle\{\mathtt{s}\}, F^\sharp, E^\sharp \dot\cup [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)]\rangle \\
&\cup_\otimes \bigcup_{\substack{\{\mathtt{t_0\,t_1}\}\subseteq C \\ \{[\mathtt{fn\ x\ {=}{>}\ s'}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}}^{\otimes} \langle\{\mathtt{s}'\}, F^\sharp \dot\cup [[\mathtt{s}']_\equiv \mapsto F^\sharp([\mathtt{t_0\,t_1}]_\equiv)], E^\sharp \dot\cup [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle \\
&\cup_\otimes \bigcup_{\substack{\{\mathtt{let\ x{=}t_0\,t_1\ in\ s}\}\subseteq C \\ \{[\mathtt{fn\ y\ {=}{>}\ s'}]\}\in\mu^\sharp(\mathtt{t_0},E^\sharp)}}^{\otimes} \langle\{\mathtt{s}'\}, F^\sharp \dot\cup [[\mathtt{s}']_\equiv \mapsto \{[\mathtt{x}, \mathtt{s}]\}], E^\sharp \dot\cup [\mathtt{y} \mapsto \mu^\sharp(\mathtt{t_1}, E^\sharp)]\rangle
\end{aligned}$$

Figure 7: The resulting analysis function

relation $\wp(C \times K)$ we use the expression-stack abstraction α_{st} developed in Section 4.3. For the set of environments $\wp(Env)$ we use the environment abstraction α_Π developed in Section 4.4.

Using the alternative "recipe" we can calculate the analysis by "pushing α's" under the intermediate transition function: $\alpha_\otimes(F^\rho(\langle C, F, E\rangle)) \subseteq_\otimes F^\sharp(\langle C, \alpha_{st}(F), \alpha_\Pi(E)\rangle)$ from which the final definition of $F^\sharp$ 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. $\alpha_\otimes \circ \rho \circ \alpha_\times(\mathrm{lfp}\, F) \subseteq_\otimes \mathrm{lfp}\, F^\sharp$

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 characteristics are well known, 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*).

$CProg \ni \mathtt{p} ::= \mathtt{fn\ k\ {=}{>}\ e}$	(CPS programs)
$SExp \ni \mathtt{e} ::= \mathtt{t_0\,t_1\,c} \mid \mathtt{c\,t}$	(serious CPS expressions)
$TExp \ni \mathtt{t} ::= \mathtt{x} \mid \mathtt{v} \mid \mathtt{fn\ x,k\ {=}{>}\ e}$	(trivial CPS expressions)
$CExp \ni \mathtt{c} ::= \mathtt{fn\ v\ {=}{>}\ e} \mid \mathtt{k}$	(continuation expressions)

Figure 8: BNF of CPS language

5. Analysis equivalence

In previous work [Midtgaard 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 $\mathtt{x} \in X$, from intermediate variables $\mathtt{v} \in V$ and continuation variables $\mathtt{k} \in K$. We assume the three classes are non-overlapping. Their union constitute the domain of CPS variables $Var = X \cup V \cup K$.

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 $\mathtt{k}$ is provided in the initial call to $\mathscr{F}$. A fresh $\mathtt{k}$ is generated in $\mathscr{V}$'s lambda abstraction case. To ease the expression of the relation, we choose $\mathtt{k}$ unique to the serious expression $\mathtt{s}$ — $\mathtt{k_s}$. It follows that we only need one $\mathtt{k}$ per lambda abstraction in the original program + an additional $\mathtt{k}$ in the initial case.

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

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

$$\mathscr{C} : P \to CProg$$
$$\mathscr{C}[\mathtt{p}] = \mathtt{fn}\ \mathtt{k_p}\ \mathtt{=>}\ \mathscr{F}_{\mathtt{k_p}}[\mathtt{p}]$$

$$\mathscr{F} : K \to C \to SExp$$
$$\mathscr{F}_{\mathtt{k}}[\mathtt{t}] = \mathtt{k}\,\mathscr{V}[\mathtt{t}]$$
$$\mathscr{F}_{\mathtt{k}}[\mathtt{let\ x{=}t\ in\ s}] = (\mathtt{fn\ x\ =>}\ \mathscr{F}_{\mathtt{k}}[\mathtt{s}])\,\mathscr{V}[\mathtt{t}]$$
$$\mathscr{F}_{\mathtt{k}}[\mathtt{t_0\,t_1}] = \mathscr{V}[\mathtt{t_0}]\,\mathscr{V}[\mathtt{t_1}]\,\mathtt{k}$$
$$\mathscr{F}_{\mathtt{k}}[\mathtt{let\ x{=}t_0\,t_1\ in\ s}] = \mathscr{V}[\mathtt{t_0}]\,\mathscr{V}[\mathtt{t_1}]\,(\mathtt{fn\ x\ =>}\ \mathscr{F}_{\mathtt{k}}[\mathtt{s}])$$

$$\mathscr{V} : T \to TExp$$
$$\mathscr{V}[\mathtt{x}] = \mathtt{x}$$
$$\mathscr{V}[\mathtt{fn\ x\ =>\ s}] = \mathtt{fn\ x,k_s\ =>}\ \mathscr{F}_{\mathtt{k_s}}[\mathtt{s}]$$

(a) CPS transformation

$$\mathscr{D} : CProg \to P$$
$$\mathscr{D}[\mathtt{fn\ k\ =>\ e}] = \mathscr{U}[\mathtt{e}]$$

$$\mathscr{U} : SExp \to C$$
$$\mathscr{U}[\mathtt{k\,t}] = \mathscr{P}[\mathtt{t}]$$
$$\mathscr{U}[(\mathtt{fn\ v\ =>\ e})\,\mathtt{t}] = \mathtt{let\ v{=}}\mathscr{P}[\mathtt{t}]\ \mathtt{in}\ \mathscr{U}[\mathtt{e}]$$
$$\mathscr{U}[\mathtt{t_0\,t_1\,k}] = \mathscr{P}[\mathtt{t_0}]\,\mathscr{P}[\mathtt{t_1}]$$
$$\mathscr{U}[\mathtt{t_0\,t_1\,(fn\ v\ =>\ e)}] = \mathtt{let\ v{=}}\mathscr{P}[\mathtt{t_0}]\,\mathscr{P}[\mathtt{t_1}]\ \mathtt{in}\ \mathscr{U}[\mathtt{e}]$$

$$\mathscr{P} : TExp \to T$$
$$\mathscr{P}[\mathtt{x}] = \mathtt{x}$$
$$\mathscr{P}[\mathtt{v}] = \mathtt{v}$$
$$\mathscr{P}[\mathtt{fn\ x,k\ =>\ e}] = \mathtt{fn\ x\ =>}\ \mathscr{U}[\mathtt{e}]$$

(b) Direct-style transformation

Figure 9: Transformations to and from CPS

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 `fn k => e` as the direct-style transformation of its body $\mathscr{U}[\mathtt{e}]$.

Transforming a program, a serious expression, or a trivial 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. $\mathscr{D}[\mathscr{C}[\mathtt{p}]] = \mathtt{p} \ \wedge\ \mathscr{U}[\mathscr{F}_{\mathtt{k}}[\mathtt{s}]] = \mathtt{s} \ \wedge\ \mathscr{P}[\mathscr{V}[\mathtt{t}]] = \mathtt{t}$

5.2 CPS analysis

We recall the CPS analysis of Midtgaard and Jensen [2008a] in Fig. 10. It is defined as the least fixed point of a program specific transfer function $T^\sharp_{\mathtt{p}}$. The definition relies on two helper functions $\mu^\sharp_t$ and $\mu^\sharp_c$ 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 values can be either the initial continuation `stop`, function closures $[\mathtt{fn\ x,k\ =>\ e}]$, or continuation closures $[\mathtt{fn\ v\ =>\ e}]$.

The definition relies on two special variables $\mathtt{k}_r$ and $\mathtt{v}_r$, the first of which names the initial continuation and the second of which names the result of the program. To ensure the most precise analysis result, variables in the source program can be renamed to be distinct as is traditional in control-flow analysis [Nielson et al., 1999].

5.3 Analysis equivalence

Before formally stating the equivalence of the two analyses we will study an example run. As our example we use the ANF program: $\mathtt{let\ f{=}fn\ x\ =>\ x\ in\ let\ a_1{=}f\ cn1\ in\ let\ a_2{=}f\ cn2\ in\ a_2}$, taken from Sabry and Felleisen [1994] where we have Church encoded the integer literals. We write `cn1` for $\mathtt{fn\ s\ =>\ fn\ z\ =>\ s\ z}$ and `cn2` for $\mathtt{fn\ s\ =>\ fn\ z\ =>\ let\ t_1{=}s\ z\ in\ s\ t_1}$. The analysis trace appears in the left column of Table 1.

Similarly we study the CPS analysis of the CPS transformed program. The analysis trace appears in the right column of Table 1 where we have written `ccn1` for $\mathscr{V}[\mathtt{cn1}]$ and `ccn2` for $\mathscr{V}[\mathtt{cn2}]$. Contrary to Sabry and Felleisen [1994] both the ANF and the CPS analyses achieve the same precision on the example, determining that $\mathtt{a_1}$ will be bound to one of the two integer literals.

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 $\langle C, F^\sharp, E^\sharp\rangle = \mathrm{lfp}\,F^\sharp_{\mathtt{p}}$ and $\langle Q^\sharp, R^\sharp\rangle = \mathrm{lfp}\,T^\sharp_{\mathscr{C}[\mathtt{p}]}$. Then

$$\mathtt{s} \in C \iff \mathscr{F}_{\mathtt{k_s}}[\mathtt{s}] \in Q^\sharp \ \wedge$$
$$[\mathtt{x},\mathtt{s}'] \in F^\sharp(\mathtt{s}) \iff [\mathtt{fn\ x\ =>}\ \mathscr{F}_{\mathtt{k_{s'}}}[\mathtt{s}']] \in R^\sharp(\mathtt{k_s}) \ \wedge$$
$$\mathtt{stop} \in F^\sharp(\mathtt{s}) \iff \mathtt{stop} \in R^\sharp(\mathtt{k_s}) \ \wedge$$
$$[\mathtt{fn\ x\ =>\ s}] \in E^\sharp(\mathtt{y}) \iff [\mathtt{fn\ x,k_s\ =>}\ \mathscr{F}_{\mathtt{k_s}}[\mathtt{s}]] \in R^\sharp(\mathtt{y})$$

For the purpose of the equivalence we equate the special variables $\mathtt{x_r}$ and $\mathtt{v}_r$ 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 $\mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}}$ that relates ANF analysis triples to CPS analysis pairs.

Definition 5.2. $\langle C, F^\sharp, E^\sharp\rangle\ \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}}\ \langle Q^\sharp, R^\sharp\rangle$ iff $\forall \mathtt{s}$:

$$\mathtt{s} \in C \implies \mathscr{F}_{\mathtt{k_s}}[\mathtt{s}] \in Q^\sharp \ \wedge$$
$$[\mathtt{x},\mathtt{s}'] \in F^\sharp(\mathtt{s}) \implies [\mathtt{fn\ x\ =>}\ \mathscr{F}_{\mathtt{k_{s'}}}[\mathtt{s}']] \in R^\sharp(\mathtt{k_s}) \ \wedge$$
$$\mathtt{stop} \in F^\sharp(\mathtt{s}) \implies \mathtt{stop} \in R^\sharp(\mathtt{k_s}) \ \wedge$$
$$[\mathtt{fn\ x\ =>\ s}] \in E^\sharp(\mathtt{y}) \implies [\mathtt{fn\ x,k_s\ =>}\ \mathscr{F}_{\mathtt{k_s}}[\mathtt{s}]] \in R^\sharp(\mathtt{y})$$

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

Lemma 5.2.

$$[\mathtt{fn\ x\ =>\ s}] \in \mu^\sharp(\mathtt{t},E^\sharp) \ \wedge\ \langle C, F^\sharp, E^\sharp\rangle\ \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}}\ \langle Q^\sharp, R^\sharp\rangle$$
$$\implies [\mathtt{fn\ x,k_s\ =>}\ \mathscr{F}_{\mathtt{k_s}}[\mathtt{s}]] \in \mu^\sharp_t(\mathscr{V}[\mathtt{t}], R^\sharp)$$

$$Env^\sharp = Var \to \wp(Val^\sharp) \qquad \text{(abstract environment)}$$
$$Val^\sharp \ni w^\sharp ::= \texttt{stop} \mid [\texttt{fn x,k => e}] \mid [\texttt{fn v => e}] \qquad \text{(abstract values)}$$

(a) Abstract domains

$$T^\sharp : CProg \to \wp(SExp) \times Env^\sharp \to \wp(SExp) \times Env^\sharp$$
$$\begin{aligned} T^\sharp_{\texttt{fn k => e}}(\langle Q^\sharp, R^\sharp\rangle) = \langle \{\texttt{e}\}, [\texttt{k}_r \mapsto \{\texttt{stop}\}, \texttt{k} \mapsto \{[\texttt{fn v}_r \texttt{ => k}_r \texttt{ v}_r]\}]\rangle \\ \cup_\otimes \bigcup_{\substack{\texttt{t}_0\,\texttt{t}_1\,\texttt{c} \in Q^\sharp \\ [\texttt{fn x,k}' \texttt{ => e}'] \in \mu^\sharp_t(\texttt{t}_0, R^\sharp)}}{}_{\otimes} \langle \{\texttt{e}'\}, R^\sharp \,\dot\cup\, [\texttt{x} \mapsto \mu^\sharp_t(\texttt{t}_1, R^\sharp), \texttt{k}' \mapsto \mu^\sharp_c(\texttt{c}, R^\sharp)]\rangle \\ \cup_\otimes \bigcup_{\substack{\texttt{c}\,\texttt{t} \in Q^\sharp \\ [\texttt{fn v => e}'] \in \mu^\sharp_c(\texttt{c}, R^\sharp)}}{}_{\otimes} \langle \{\texttt{e}'\}, R^\sharp \,\dot\cup\, [\texttt{v} \mapsto \mu^\sharp_t(\texttt{t}, R^\sharp)]\rangle \end{aligned}$$

(b) Abstract transition function

$$\mu^\sharp_t : TExp \times Env^\sharp \to \wp(Val^\sharp)$$
$$\mu^\sharp_t(\texttt{x}, R^\sharp) = R^\sharp(\texttt{x})$$
$$\mu^\sharp_t(\texttt{v}, R^\sharp) = R^\sharp(\texttt{v})$$
$$\mu^\sharp_t(\texttt{fn x,k => e}, R^\sharp) = \{[\texttt{fn x,k => e}]\}$$
$$\mu^\sharp_c : CExp \times Env^\sharp \to \wp(Val^\sharp)$$
$$\mu^\sharp_c(\texttt{k}, R^\sharp) = R^\sharp(\texttt{k})$$
$$\mu^\sharp_c(\texttt{fn v => e}, R^\sharp) = \{[\texttt{fn v => e}]\}$$

(c) Abstract helper functions

Figure 10: CPS analysis

The relation is preserved by the transfer functions.

Theorem 5.2.

$$\begin{aligned} &\langle C, F^\sharp, E^\sharp\rangle \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; \langle Q^\sharp, R^\sharp\rangle \\ &\qquad \Longrightarrow F^\sharp_{\mathtt{p}}(\langle C, F^\sharp, E^\sharp\rangle) \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; T^\sharp_{\mathscr{C}[\mathtt{p}]}(\langle Q^\sharp, R^\sharp\rangle) \end{aligned}$$

Proof. First we name the individual triples of the union in the function body of $F^\sharp$. We name the first triple of results as initial: $\langle C_I, F^\sharp_I, E^\sharp_I\rangle = \langle \{\mathtt{p}\}, [\mathtt{p} \mapsto \{[\mathtt{x_r}, \mathtt{x_r}]\}, \mathtt{x_r} \mapsto \{\mathtt{stop}\}], \lambda_.\emptyset\rangle$. The results of the second, third, fourth, and fifth joined triples corresponding to return, binding, tail call, and non-tail call are named $\langle C_{ret}, F^\sharp_{ret}, E^\sharp_{ret}\rangle$, $\langle C_{bind}, F^\sharp_{bind}, E^\sharp_{bind}\rangle$, $\langle C_{tc}, F^\sharp_{tc}, E^\sharp_{tc}\rangle$ and $\langle C_{ntc}, F^\sharp_{ntc}, E^\sharp_{ntc}\rangle$, respectively. Similarly we name the first result pair in the function body of the CPS analysis as initial: $\langle Q^\sharp_I, R^\sharp_I\rangle = \langle \{\mathtt{e}\}, [\mathtt{k}_r \mapsto \{\mathtt{stop}\}, \mathtt{k} \mapsto \{[\mathtt{fn\ v}_r \mathtt{\ =>\ k}_r\ \mathtt{v}_r]\}]\rangle$. The results of the second and third joined pair corresponding to call and return are named $\langle Q^\sharp_{call}, R^\sharp_{call}\rangle$ and $\langle Q^\sharp_{ret}, R^\sharp_{ret}\rangle$, respectively.

The proof proceeds by verifying five relations:

$$\langle C_I, F^\sharp_I, E^\sharp_I\rangle \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; \langle Q^\sharp_I, R^\sharp_I\rangle \tag{1}$$
$$\langle C_{ret}, F^\sharp_{ret}, E^\sharp_{ret}\rangle \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; \langle Q^\sharp_{ret}, R^\sharp_{ret}\rangle \tag{2}$$
$$\langle C_{bind}, F^\sharp_{bind}, E^\sharp_{bind}\rangle \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; \langle Q^\sharp_{ret}, R^\sharp_{ret}\rangle \tag{3}$$
$$\langle C_{tc}, F^\sharp_{tc}, E^\sharp_{tc}\rangle \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; \langle Q^\sharp_{call}, R^\sharp_{call}\rangle \tag{4}$$
$$\langle C_{ntc}, F^\sharp_{ntc}, E^\sharp_{ntc}\rangle \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; \langle Q^\sharp_{call}, R^\sharp_{call}\rangle \tag{5}$$

Realizing that the union of related triples and pairs are related we obtain the desired result. □

After realizing that the bottom elements are related by the above relation, it follows by fixed point induction that their least fixed points (and hence the analyses) are related.

Corollary 5.1. $\mathrm{lfp}\, F^\sharp_{\mathtt{p}} \; \mathrm{R}^{\mathrm{ANF}}_{\mathrm{CPS}} \; \mathrm{lfp}\, T^\sharp_{\mathscr{C}[\mathtt{p}]}$

5.5 CPS-ANF equivalence

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

Definition 5.3. $\langle Q^\sharp, R^\sharp\rangle \; \mathrm{R}^{\mathrm{CPS}}_{\mathrm{ANF}} \; \langle C, F^\sharp, E^\sharp\rangle$ iff $\forall \mathtt{e}$:

$$\begin{aligned} \mathtt{e} \in Q^\sharp &\Longrightarrow \mathscr{U}[\mathtt{e}] \in C \;\wedge \\ [\mathtt{fn\ x\ =>\ e}] \in R^\sharp(\mathtt{k_s}) &\Longrightarrow [\mathtt{x}, \mathscr{U}[\mathtt{e}]] \in F^\sharp(\mathtt{s}) \;\wedge \\ \mathtt{stop} \in R^\sharp(\mathtt{k_s}) &\Longrightarrow \mathtt{stop} \in F^\sharp(\mathtt{s}) \;\wedge \\ [\mathtt{fn\ x, k_s\ =>\ e}] \in R^\sharp(\mathtt{y}) &\Longrightarrow [\mathtt{fn\ x\ =>}\ \mathscr{U}[\mathtt{e}]] \in E^\sharp(\mathtt{y}) \end{aligned}$$

We again need a helper lemma relating the helper functions.

Lemma 5.3.

$$\begin{aligned} &[\mathtt{fn\ x, k_s\ =>\ e}] \in \mu^\sharp_t(\mathtt{t}, R^\sharp) \;\wedge\; \langle Q^\sharp, R^\sharp\rangle \; \mathrm{R}^{\mathrm{CPS}}_{\mathrm{ANF}} \; \langle C, F^\sharp, E^\sharp\rangle \\ &\qquad \Longrightarrow [\mathtt{fn\ x\ =>}\ \mathscr{U}[\mathtt{e}]] \in \mu^\sharp(\mathscr{P}[\mathtt{t}], E^\sharp) \end{aligned}$$

This relation is also preserved by the transfer functions.

Theorem 5.3.

$$\begin{aligned} &\langle Q^\sharp, R^\sharp\rangle \; \mathrm{R}^{\mathrm{CPS}}_{\mathrm{ANF}} \; \langle C, F^\sharp, E^\sharp\rangle \\ &\qquad \Longrightarrow T^\sharp_{\mathscr{C}[\mathtt{p}]}(\langle Q^\sharp, R^\sharp\rangle) \; \mathrm{R}^{\mathrm{CPS}}_{\mathrm{ANF}} \; F^\sharp_{\mathtt{p}}(\langle C, F^\sharp, E^\sharp\rangle) \end{aligned}$$

Proof. The proof follows a similar structure to the earlier proof. □

The bottom elements are related by the relation and it follows by fixed point induction that their least fixed points (and hence the analyses) are related.

Corollary 5.2. $\mathrm{lfp}\, T^\sharp_{\mathscr{C}[\mathtt{p}]} \; \mathrm{R}^{\mathrm{CPS}}_{\mathrm{ANF}} \; \mathrm{lfp}\, F^\sharp_{\mathtt{p}}$

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 [Palsberg and Schwartzbach, 1995, Gasser et al., 1997]:

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

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

i	ANF trace: $\langle C_i, F^\sharp_i, E^\sharp_i\rangle$	CPS trace: $\langle Q^\sharp_i, R^\sharp_i\rangle$
0	$\{\texttt{let f=fn x => x in let a}_1\texttt{=f cn1 in let a}_2\texttt{=f cn2 in a}_2\}$ $\left[\begin{array}{l}[\texttt{x}_\texttt{r}]_\equiv \mapsto \{\texttt{stop}\},\\ [\texttt{let f=fn x => x in let a}_1\texttt{=f cn1 in let a}_2\texttt{=f cn2 in a}_2]_\equiv \mapsto \{[\texttt{x}_\texttt{r}, \texttt{x}_\texttt{r}]\}\end{array}\right]$ $\lambda_.\emptyset$	$\{(\texttt{fn f => f ccn1 (fn a}_1\texttt{ => f ccn2 (fn a}_2\texttt{ => k}_\texttt{p}\ \texttt{a}_2)))\,(\texttt{fn x, k}_\texttt{x}\texttt{ => k}_\texttt{x}\ \texttt{x})\}$ $\left[\begin{array}{l}\texttt{k}_r \mapsto \{\texttt{stop}\},\\ \texttt{k}_\texttt{p} \mapsto \{[\texttt{fn v}_r\texttt{ => k}_r\ \texttt{v}_r]\}\end{array}\right]$
1	$C_0 \cup \{\texttt{let a}_1\texttt{=f cn1 in let a}_2\texttt{=f cn2 in a}_2\}$ $F^\sharp_0$ $E^\sharp_0 \dot\cup \left[\texttt{f} \mapsto \{[\texttt{fn x => x}]\}\right]$	$Q^\sharp_0 \cup \{\texttt{f ccn1 (fn a}_1\texttt{ => f ccn2 (fn a}_2\texttt{ => k}_\texttt{p}\ \texttt{a}_2))\}$ $R^\sharp_0 \dot\cup \left[\texttt{f} \mapsto \{[\texttt{fn x, k}_\texttt{x}\texttt{ => k}_\texttt{x}\ \texttt{x}]\}\right]$
2	$C_1 \cup \{\texttt{x}\}$ $F^\sharp_1 \dot\cup \left[[\texttt{x}]_\equiv \mapsto \{[\texttt{a}_1, \texttt{let a}_2\texttt{=f cn2 in a}_2]\}\right]$ $E^\sharp_1 \dot\cup \left[\texttt{x} \mapsto \{\texttt{cn1}\}\right]$	$Q^\sharp_1 \cup \{\texttt{k}_\texttt{x}\ \texttt{x}\}$ $R^\sharp_1 \dot\cup \left[\begin{array}{l}\texttt{k}_\texttt{x} \mapsto \{[\texttt{fn a}_1\texttt{ => f ccn2 (fn a}_2\texttt{ => k}_\texttt{p}\ \texttt{a}_2)]\}\\ \texttt{x} \mapsto \{\texttt{ccn1}\}\end{array}\right]$
3	$C_2 \cup \{\texttt{let a}_2\texttt{=f cn2 in a}_2\}$ $F^\sharp_2$ $E^\sharp_2 \dot\cup \left[\texttt{a}_1 \mapsto \{\texttt{cn1}\}\right]$	$Q^\sharp_2 \cup \{\texttt{f ccn2 (fn a}_2\texttt{ => k}_\texttt{p}\ \texttt{a}_2)\}$ $R^\sharp_2 \dot\cup \left[\texttt{a}_1 \mapsto \{\texttt{ccn1}\}\right]$
4	C_3 $F^\sharp_3 \dot\cup \left[[\texttt{x}]_\equiv \mapsto \{[\texttt{a}_1, \texttt{let a}_2\texttt{=f cn2 in a}_2], [\texttt{a}_2, \texttt{a}_2]\}\right]$ $E^\sharp_3 \dot\cup \left[\texttt{x} \mapsto \{\texttt{cn1}, \texttt{cn2}\}\right]$	$Q^\sharp_3$ $R^\sharp_3 \dot\cup \left[\begin{array}{l}\texttt{k}_\texttt{x} \mapsto \{[\texttt{fn a}_1\texttt{ => f ccn2 (fn a}_2\texttt{ => k}_\texttt{p}\ \texttt{a}_2)], [\texttt{fn a}_2\texttt{ => k}_\texttt{p}\ \texttt{a}_2]\}\\ \texttt{x} \mapsto \{\texttt{ccn1}, \texttt{ccn2}\}\end{array}\right]$
5	$C_4 \cup \{\texttt{a}_2\}$ $F^\sharp_4$ $E^\sharp_4 \dot\cup \left[\begin{array}{l}\texttt{a}_1 \mapsto \{\texttt{cn1}, \texttt{cn2}\}\\ \texttt{a}_2 \mapsto \{\texttt{cn1}, \texttt{cn2}\}\end{array}\right]$	$Q^\sharp_4 \cup \{\texttt{k}_\texttt{p}\ \texttt{a}_2\}$ $R^\sharp_4 \dot\cup \left[\begin{array}{l}\texttt{a}_1 \mapsto \{\texttt{ccn1}, \texttt{ccn2}\}\\ \texttt{a}_2 \mapsto \{\texttt{ccn1}, \texttt{ccn2}\}\end{array}\right]$
6	$C_5 \cup \{\texttt{x}_\texttt{r}\}$ $F^\sharp_5$ $E^\sharp_5 \dot\cup \left[\texttt{x}_\texttt{r} \mapsto \{\texttt{cn1}, \texttt{cn2}\}\right]$	$Q^\sharp_5 \cup \{\texttt{k}_r\ \texttt{v}_r\}$ $R^\sharp_5 \dot\cup \left[\texttt{v}_r \mapsto \{\texttt{ccn1}, \texttt{ccn2}\}\right]$
7	$C_6 \quad F^\sharp_6 \quad E^\sharp_6$	$Q^\sharp_6 \quad R^\sharp_6$

Table 1: Analysis traces of $\texttt{let f=fn x => x in let a}_1\texttt{=f cn1 in let a}_2\texttt{=f cn2 in a}_2$ and its CPS transformed counterpart

- For the program $\texttt{p}$:

$$\{\texttt{p}\} \subseteq C \qquad \{[\texttt{x}_\texttt{r}, \texttt{x}_\texttt{r}]\} \subseteq F^\sharp([\texttt{p}]_\equiv) \qquad \{\texttt{stop}\} \subseteq F^\sharp([\texttt{x}_\texttt{r}]_\equiv)$$

- For each return expression $\texttt{t}$ and non-tail call $\texttt{let x=t}_0\ \texttt{t}_1\ \texttt{in s}'$ in $\texttt{p}$:

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

- For each let-binding $\texttt{let x=t in s}$ in $\texttt{p}$:

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

- For each tail call $\texttt{t}_0\ \texttt{t}_1$ and function $\texttt{fn x => s}'$ in $\texttt{p}$:

$$\begin{array}{c}\{\texttt{t}_0\ \texttt{t}_1\} \subseteq C \wedge \\ \{[\texttt{fn x => s}']\} \subseteq \mu_{sym}(\texttt{t}_0, E^\sharp)\end{array} \Rightarrow \begin{cases}\{\texttt{s}'\} \subseteq C \wedge \\ F^\sharp([\texttt{t}_0\ \texttt{t}_1]_\equiv) \subseteq F^\sharp([\texttt{s}']_\equiv) \wedge \\ \mu_{sym}(\texttt{t}_1, E^\sharp) \subseteq E^\sharp(\texttt{x})\end{cases}$$

- For each non-tail call $\texttt{let x=t}_0\ \texttt{t}_1\ \texttt{in s}$ and function $\texttt{fn y => s}'$ in $\texttt{p}$:

$$\begin{array}{l}\{\texttt{let x=t}_0\ \texttt{t}_1\ \texttt{in s}\} \subseteq C \wedge \\ \{[\texttt{fn y => s}']\} \subseteq \mu_{sym}(\texttt{t}_0, E^\sharp)\end{array} \Rightarrow \begin{cases}\{\texttt{s}'\} \subseteq C \wedge \\ \{[\texttt{x}, \texttt{s}]\} \subseteq F^\sharp([\texttt{s}']_\equiv) \wedge \\ \mu_{sym}(\texttt{t}_1, E^\sharp) \subseteq E^\sharp(\texttt{y})\end{cases}$$

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 $\mu^\sharp$.

We may generate constraints $\{[\texttt{fn x => s}]\} \subseteq \{[\texttt{fn y => s}']\}$ of a form not covered by the above grammar. We therefore first preprocess the constraints in linear time, removing vacuously true inclusions $\{[\texttt{fn x => s}]\} \subseteq \{[\texttt{fn x => s}]\}$ from each constraint, and removing constraints containing vacuously false preconditions $\{[\texttt{fn x => s}]\} \subseteq \{w^\sharp\}$, where $[\texttt{fn y => s}'] \neq w^\sharp$.

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 $F^\sharp$ induced by p. Furthermore, the least solution to the CFA constraints is equal to the least fixpoint of $F^\sharp$.

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 $O(n^3)$ time.

The extracted constraints bear similarities to existing constraint-based analyses in the literature. Consider, e.g., calls $\mathtt{t_0}\,\mathtt{t_1}$, which usually gives rise to two conditional constraints [Palsberg, 1995, Nielson et al., 1999]: (1) $\{[\mathtt{fn\ x\ =>\ s'}]\} \subseteq \widehat{C}(\mathtt{t_0}) \Rightarrow \widehat{C}(\mathtt{t_1}) \subseteq \widehat{E}(\mathtt{x})$ and (2) $\{[\mathtt{fn\ x\ =>\ s'}]\} \subseteq \widehat{C}(\mathtt{t_0}) \Rightarrow \widehat{C}(\mathtt{s'}) \subseteq \widehat{C}(\mathtt{t_0}\,\mathtt{t_1})$. 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 $\wp(C) \simeq C \to \wp(\{\mathbf{on}\})$ 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_aEK 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]

[2] available at http://www.brics.dk/~jmi/ANF-CFA/

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 (γ) 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

J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, 1998.

A. E. Ayers. Efficient closure analysis with reachability. In M. Billaud, P. Castéran, M.-M. Corsini, K. Musumbu, and A. Rauzy, editors, *Actes WSA'92 Workshop on Static Analysis*, Bigre, pages 126–134, Bordeaux, France, Sept. 1992. Atelier Irisa, IRISA, Campus de Beaulieu.

D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1): 56–78, 2005.

W. D. Clinger. Proper tail recursion and space efficiency. In K. D. Cooper, editor, *Proc. of the ACM SIGPLAN 1998 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998.

C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Proc. of the Fifth ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *LNCS*, pages 496–519, Cambridge, Massachusetts, Aug. 1991. Springer-Verlag.

P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

P. Cousot. Semantic foundations of program analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

P. Cousot and R. Cousot. Abstract interpretation of algebraic polynomial systems. In M. Johnson, editor, *Proc. of the Sixth International Conference on Algebraic Methodology and Software Technology, AMAST '97*, volume 1349 of *LNCS*, pages 138–154, Sydney, Australia, Dec. 1997. Springer-Verlag.

P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In H. Bal, editor, *Proc. of the Fifth IEEE International Conference on Computer Languages*, pages 95–112, Toulouse, France, May 1994.

P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992a.

P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992b.

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

posium on Principles of Programming Languages, pages 269–282, San Antonio, Texas, Jan. 1979.

D. Damian and O. Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. *Journal of Functional Programming*, 13(5):867–904, 2003. A preliminary version was presented at the 2000 ACM SIGPLAN International Conference on Functional Programming.

O. Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, Dec. 1991.

B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, England, second edition, 2002.

S. K. Debray and T. A. Proebsting. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Transactions on Programming Languages and Systems*, 19(4):568–585, 1997.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In D. W. Wall, editor, *Proc. of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In M. Tofte, editor, *Proc. of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 38–51, Amsterdam, The Netherlands, June 1997.

R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.

N. Heintze. Set-based program analysis of ML programs. In C. L. Talcott, editor, *Proc. of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, pages 306–317, Orlando, Florida, June 1994.

N. D. Jones. Flow analysis of lambda expressions (preliminary version). In S. Even and O. Kariv, editors, *Automata, Languages and Programming, 8th Colloquium, Acre (Akko)*, volume 115 of *LNCS*, pages 114–128, Israel, July 1981. Springer-Verlag.

J. Midtgaard. Control-flow analysis of functional programs. Technical Report BRICS RS-07-18, Dept. of Comp. Sci., University of Aarhus, Aarhus, Denmark, Dec. 2007. Accepted for publication in *ACM Computing Surveys*.

J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In M. Alpuente and G. Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008*, volume 5079 of *LNCS*, pages 347–362, Valencia, Spain, July 2008a. Springer-Verlag.

J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. Rapport de Recherche RR-6681, INRIA Rennes – Bretagne Atlantique, Oct. 2008b.

M. Might and O. Shivers. Environmental analysis via ΔCFA. In S. Peyton Jones, editor, *Proc. of the 33rd Annual ACM Symposium on Principles of Programming Languages*, pages 127–140, Charleston, South Carolina, Jan. 2006.

R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In N. D. Jones, editor, *Proc. of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 332–345, Paris, France, Jan. 1997.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

H. R. Nielson and F. Nielson. Flow logic: a multi-paradigmatic approach to static analysis. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 223–244. Springer-Verlag, 2002.

J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, 1995.

J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

J. Palsberg and M. Wand. CPS transformation of flow information. *Journal of Functional Programming*, 13(5):905–923, 2003.

D. Pichardie. *Interprétation abstraite en logique intuitioniste: extraction d'analyseurs Java certifiés*. PhD thesis, Université de Rennes 1, Sept. 2005.

A. Sabry and M. Felleisen. Is continuation-passing useful for data flow analysis? In V. Sarkar, editor, *Proc. of the ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation*, pages 1–12, Orlando, Florida, June 1994.

P. Sestoft. Replacing function parameters by global variables. In J. E. Stoy, editor, *Proc. of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, Sept. 1989.

O. Shivers. Control-flow analysis in Scheme. In M. D. Schwartz, editor, *Proc. of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation*, pages 164–174, Atlanta, Georgia, June 1988.

F. Spoto and T. P. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Transactions on Programming Languages and Systems*, 25(5):578–630, 2003.

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

A Galois connection is a pair of functions α, γ between two posets $\langle C; \sqsubseteq \rangle$ and $\langle A; \leq \rangle$ such that for all $a \in A, c \in C : \alpha(c) \leq a \iff c \sqsubseteq \gamma(a)$. Equivalently a Galois connection can be defined as a pair of functions satisfying (a) α and γ are monotone, (b) $\alpha \circ \gamma$ is reductive, and (c) $\gamma \circ \alpha$ is extensive. Galois connections are typeset $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \leq \rangle$. We omit the orderings when they are clear from the context. For a Galois connection between two complete lattices α is a complete join-morphism (CJM) and γ is a complete meet morphism. The composition of two Galois connections $\langle C; \sqsubseteq \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle B; \subseteq \rangle$ and $\langle B; \subseteq \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle A; \leq \rangle$ is itself a Galois connection $\langle C; \sqsubseteq \rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle A; \leq \rangle$. Galois connections in which α is surjective (or equivalently γ is injective) are typeset $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftarrows}} \langle A; \leq \rangle$. Galois connections in which γ is surjective (or equivalently α is injective) are typeset $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\rightarrows}} \langle A; \leq \rangle$. When both α and γ are surjective, the two domains are isomorphic.

A(n upper) closure operator ρ is map $\rho : S \to S$ on a poset $\langle S; \sqsubseteq \rangle$, that is (a) monotone: (for all $s, s' \in S : s \sqsubseteq s' \implies \rho(s) \sqsubseteq \rho(s')$), (b) extensive (for all $s \in S : s \sqsubseteq \rho(s)$), and (c) idempotent, (for all $s \in S : \rho(s) = \rho(\rho(s))$). A closure operator ρ induces a Galois connection $\langle S; \sqsubseteq \rangle \underset{\rho}{\overset{1}{\leftrightarrows}} \langle \rho(S); \sqsubseteq \rangle$, writing $\rho(S)$ for $\{\rho(s) \mid s \in S\}$ and 1 for the identity function. Furthermore the image of a complete lattice $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ by an upper closure operator is itself a complete lattice $\langle \rho(C); \sqsubseteq, \rho(\bot), \top, \lambda X. \rho(\sqcup X), \sqcap \rangle$.