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

Jan Midtgaard, Thomas P. Jensen

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Jan Midtgaard — Thomas P. Jensen

N° 6681

October 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6681--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

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

Jan Midtgaard*, Thomas P. Jensen†

Thème SYM — Systèmes symboliques
Équipe-Projet Lande

Rapport de recherche n° 6681 — October 2008 — 28 pages

Abstract: We derive a control-flow analysis for a simple higher-order functional language. Contrary to existing direct-style analyses, our analysis 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 call-stack, effectively approximating where function calls return. 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. From the analysis we extract an equivalent constraint-based formulation, thereby providing a rational reconstruction of a constraint-based CFA from abstract interpretation principles.

Key-words: control-flow analysis, abstract interpretation

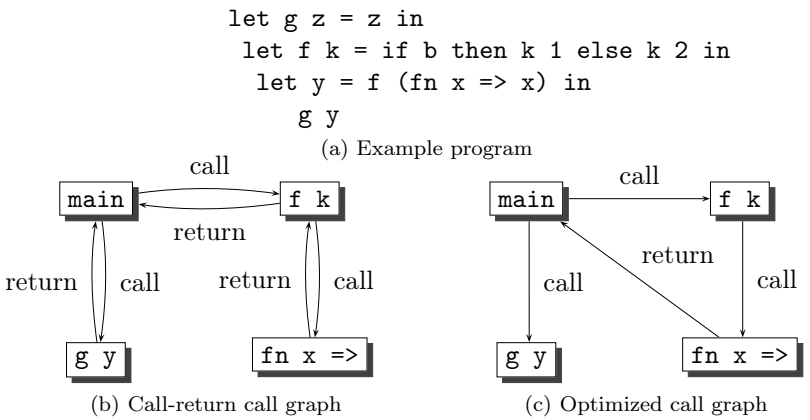
* INRIA Rennes – Bretagne Atlantique
† CNRS

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

—

Résumé : –

Mots-clés : –



(a) Example program

(b) Call-return call graph

(c) Optimized call graph

Figure 1: The corresponding call graphs

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 solely on calls: A CFA *"will determine where the flow of control may be transferred to in the case [...] of a function application."* [Nielson et al., 1999]. We argue that returns are an important but neglected control-flow mechanism and that returns should be given equal attention. To shift focus, we formulate a control-flow analysis that for each expression *"will determine where the flow of control may be transferred to in the case of a function return."* Our 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 Figure 1. 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 Figure 1b, 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 Figure 1c.

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.

1.1 Related work

We separate the discussion of related analyses in two: direct-style analyses and analyses based on continuation-passing style (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]. Surprisingly 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 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 a number of analyses have been formulated for CPS [Ashley and Dybvig, 1998, Might and Shivers, 2006]. In CPS all calls are tail calls, and even returns are encoded as calls to the *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.

Formulating CFA in the traditional abstract interpretation framework has been a recurring theme in the authors's previous work. 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, 2008], the authors systematically derived a 0-CFA for CPS from the collecting semantics of a stackless machine. While investigating how to derive a corresponding direct-style analysis we discovered the 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 two 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.

We intend not to enter a debate [Shivers, 2004, Flanagan et al., 2004, Kennedy, 2007] on the merits of one or another normal form (ANF or CPS) with this paper. Instead we intend to point out and correct the mismatch between the information computed by existing direct-style analyses and their CPS counterparts.

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. The rest of this article is organized as follows. In Section 2 we present the syntax and semantics of the language. In Section 3 we briefly recall basic principles of abstract interpretation. In Section 4 we formulate the collecting semantics of the analysis, which we systematically approximate into

an analysis in Section 5 and Section 6. In Section 7 we extract an equivalent constraint-based formulation. Section 8 explores applications of the analysis and Section 9 discusses our results, before we conclude.

2 Language and semantics

Our source language is a simple call-by-value core language in which all intermediate results are let bound, a language form which is known as *administrative normal form* (ANF). The grammar of ANF terms is given in Figure 2. Following Reynolds [1998], 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.

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

Figure 2: ANF grammar

The analysis is calculated from a simple operational semantics in the form of an abstract machine. We use the C_aEK abstract machine of Flanagan et al. [1993] given in Figure 3. The machine is environment-based, as such functional values are represented using *closures* [Landin, 1964], 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 $[\mathtt{x}, \mathtt{s}, e]$ 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 extended environment $e[\mathtt{x} \mapsto w]$ represents the rest of the computation in that stack frame. The empty stack is represented by `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 seman-*

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

(a) Values, environments, and stacks

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

(b) Helper function

$$\begin{aligned} \langle \texttt{t}, e, [\texttt{x}, \texttt{s}', e'] :: k' \rangle &\longrightarrow \langle \texttt{s}', e'[\texttt{x} \mapsto \mu(\texttt{t}, e)], k' \rangle \\ \langle \texttt{let x=t in s}, e, k \rangle &\longrightarrow \langle \texttt{s}, e[\texttt{x} \mapsto \mu(\texttt{t}, e)], k \rangle \\ \langle \texttt{t}_0\,\texttt{t}_1, e, k \rangle &\longrightarrow \langle \texttt{s}', e'[\texttt{x} \mapsto w], k \rangle \\ &\qquad \text{if } [\texttt{fn x => s}', e'] = \mu(\texttt{t}_0, e) \text{ and } w = \mu(\texttt{t}_1, e) \\ \langle \texttt{let x=t}_0\,\texttt{t}_1\,\texttt{in s}, e, k \rangle &\longrightarrow \langle \texttt{s}', e'[\texttt{y} \mapsto w], [\texttt{x}, \texttt{s}, e] :: k \rangle \\ &\qquad \text{if } [\texttt{fn y => s}', e'] = \mu(\texttt{t}_0, e) \text{ and } w = \mu(\texttt{t}_1, e) \end{aligned}$$

(c) Machine transitions

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

(d) Machine evaluation

Figure 3: The C_aEK abstract machine

tics 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 on 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.

Rather than insisting on simplifying the abstract domains into finite ones, an alternative *widening* technique permits infinite ones, while still ensuring termination. Abstract interpretation with widening [Cousot and Cousot, 1977] can be formulated as computing the limit of the sequence:

$$\begin{aligned} X_0 &= \bot \\ X_{i+1} &= X_i \triangledown T(X_i) \end{aligned}$$

where $\triangledown$ denotes the *widening operator*: an operator not decreasing in it's second argument, which must not give rise to an infinite, strictly increasing sequence: $X_0 \sqsubset X_1 \sqsubset \ldots$.

4 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.

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

First we formulate in Figure 4a 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 straight forwardly. This lemma enables us to express an equivalent collecting semantics based on μ_c, which appears in Figure 4.

Lemma 4.2. $\forall S : F(S) = F_c(S)$

Proof. By above lemma and unfolding the definitions. □

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

(a) Helper function

$$F_c : \wp(C \times Env \times K) \to \wp(C \times Env \times K)$$
$$F_c(S) = I_{\mathtt{p}}$$
$$\cup \bigcup_{\substack{\langle \mathtt{t},\, e,\, [\mathtt{x},\, \mathtt{s}',\, e']::k' \rangle \in S \\ w \in \mu_c(\mathtt{t}, \{e\})}} \{\langle \mathtt{s}',\ e'[\mathtt{x} \mapsto w],\ k' \rangle\}$$
$$\cup \bigcup_{\substack{\langle \mathtt{let\ x=t\ in\ s},\, e,\, k \rangle \in S \\ w \in \mu_c(\mathtt{t}, \{e\})}} \{\langle \mathtt{s},\ e[\mathtt{x} \mapsto w],\ k \rangle\}$$
$$\cup \bigcup_{\substack{\langle \mathtt{t_0\ t_1},\, e,\, k \rangle \in S \\ [\mathtt{fn\ x\ =>\ s}',\, e'] \in \mu_c(\mathtt{t_0}, \{e\}) \\ w \in \mu_c(\mathtt{t_1}, \{e\})}} \{\langle \mathtt{s}',\ e'[\mathtt{x} \mapsto w],\ k \rangle\}$$
$$\cup \bigcup_{\substack{\langle \mathtt{let\ x=t_0\ t_1\ in\ s},\, e,\, k \rangle \in S \\ [\mathtt{fn\ y\ =>\ s}',\, e'] \in \mu_c(\mathtt{t_0}, \{e\}) \\ w \in \mu_c(\mathtt{t_1}, \{e\})}} \{\langle \mathtt{s}',\ e'[\mathtt{y} \mapsto w],\ [\mathtt{x},\ \mathtt{s},\ e] :: k \rangle\}$$

(b) Transition function

Figure 4: Collecting semantics

5 Approximating the collecting semantics

The abstraction of the collecting semantics is staged in several steps. Figure 5 provides an overview. First we perform a Cartesian abstraction of the machine states, however keeping the relation between expressions and their calling context. Second we close 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 set, and (b) that any sub-stack is also contained in the expression-stack relation. Finally as a third step we approximate stacks by their top element and closure values by their lambda expression.

In the following sections we provide a detailed explanation of each abstraction in turn. In order to illustrate the systematic calculation and still remain of a manageable size, we only provide the calculations for the return case `t`. Since we calculate with Galois connections on complete lattices, the abstraction functions are complete join morphisms (CJMs), and hence distribute over each

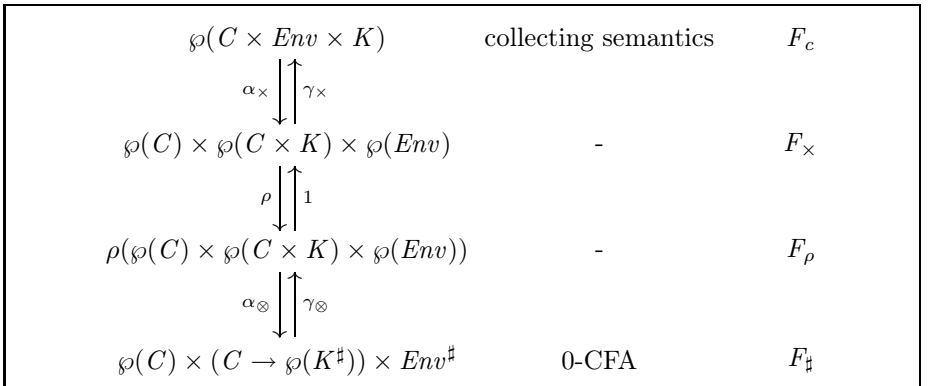


Figure 5: Overview of abstraction

element of a join, permitting us to do such case division. The remaining cases are proved similarly.

5.1 Projecting machine states

In CPS the relation between an expression and its corresponding return point is represented as a binding in the environment between the continuation variable and a continuation closure (read: stack frame). In the direct-style case the relation between an expression and its corresponding return point will have to be maintained elsewhere. In an attempt to do so, we extract expression-stack pairs from the set of reachable states.

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

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

Lemma 5.1. *$\alpha_\times$, $\gamma_\times$ 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 $\langle A,\ B,\ C\rangle$ containing 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 the partly-relational abstraction we now calculate a new transfer function. Let $\langle C,\ F,\ E\rangle \in \wp(C) \times \wp(C \times K) \times \wp(Env)$ be given.

$$\alpha_\times\Big(\bigcup_{\substack{\langle \mathtt{t},\ e,\ [\mathtt{x},\ \mathtt{s}',\ e']::k'\rangle \in \gamma_\times(\langle C,\ F,\ E\rangle) \\ w \in \mu_c(\mathtt{t},\{e\})}} \{\langle \mathtt{s}',\ e'[\mathtt{x} \mapsto w],\ k'\rangle\}\Big)$$

$$= \bigcup_{\substack{\times \\ \langle \mathtt{t}, e, [\mathtt{x}, \mathtt{s}', e']::k' \rangle \in \gamma_\times(\langle C, F, E \rangle) \\ w \in \mu_c(\mathtt{t}, \{e\})}} \alpha_\times(\{\langle \mathtt{s}', e'[\mathtt{x} \mapsto w], k' \rangle\}) \qquad (\alpha_\times \text{ a CJM})$$

$$= \bigcup_{\substack{\times \\ \langle \mathtt{t}, e, [\mathtt{x}, \mathtt{s}', e']::k' \rangle \in \gamma_\times(\langle C, F, E \rangle) \\ w \in \mu_c(\mathtt{t}, \{e\})}} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle \qquad (\text{def. } \alpha_\times)$$

$$= \bigcup_{\substack{\times \\ \alpha_\times(\{\langle \mathtt{t}, e, [\mathtt{x}, \mathtt{s}', e']::k' \rangle\}) \subseteq_\times \langle C, F, E \rangle \\ w \in \mu_c(\mathtt{t}, \{e\})}} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle \quad (\text{Galois conn.})$$

$$= \bigcup_{\substack{\times \\ \langle \{\mathtt{t}\}, \{\langle \mathtt{t}, [\mathtt{x}, \mathtt{s}', e']::k' \rangle\}, \{e\} \rangle \subseteq_\times \langle C, F, E \rangle \\ w \in \mu_c(\mathtt{t}, \{e\})}} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle \qquad (\text{def. } \alpha_\times)$$

The resulting transition function appears in Figure 6. By construction the transition function satisfies the following theorem.

Theorem 5.1.

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

$$F_\times : \wp(C) \times \wp(C \times K) \times \wp(Env) \to \wp(C) \times \wp(C \times K) \times \wp(Env)$$

$$F_\times(\langle C,\ F,\ E \rangle) = \langle \{\mathtt{p}\}, \{\langle \mathtt{p}, [\mathtt{x_r}, \mathtt{x_r}, \bullet] :: \mathtt{stop} \rangle\}, \{\bullet\} \rangle$$

$$\cup_\times \bigcup_{\substack{\times \\ \langle \{\mathtt{t}\}, \{\langle \mathtt{t}, [\mathtt{x}, \mathtt{s}', e']::k' \rangle\}, \{e\} \rangle \subseteq_\times \langle C, F, E \rangle \\ w \in \mu_c(\mathtt{t}, \{e\})}} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle$$

$$\cup_\times \bigcup_{\substack{\times \\ \langle \{\mathtt{let\ x=t\ in\ s}\}, \{\langle \mathtt{let\ x=t\ in\ s}, k \rangle\}, \{e\} \rangle \subseteq_\times \langle C, F, E \rangle \\ w \in \mu_c(\mathtt{t}, \{e\})}} \langle \{\mathtt{s}\}, \{\langle \mathtt{s}, k \rangle\}, \{e[\mathtt{x} \mapsto w]\} \rangle$$

$$\cup_\times \bigcup_{\substack{\times \\ \langle \{\mathtt{t_0\ t_1}\}, \{\langle \mathtt{t_0\ t_1}, k \rangle\}, \{e\} \rangle \subseteq_\times \langle C, F, E \rangle \\ [\mathtt{fn\ x\ =>\ s'}, e'] \in \mu_c(\mathtt{t_0}, \{e\}) \\ w \in \mu_c(\mathtt{t_1}, \{e\})}} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle$$

$$\cup_\times \bigcup_{\substack{\times \\ \langle \{\mathtt{let\ x=t_0\ t_1\ in\ s}\}, \{\langle \mathtt{let\ x=t_0\ t_1\ in\ s}, k \rangle\}, \{e\} \rangle \subseteq_\times \langle C, F, E \rangle \\ [\mathtt{fn\ y\ =>\ s'}, e'] \in \mu_c(\mathtt{t_0}, \{e\}) \\ w \in \mu_c(\mathtt{t_1}, \{e\})}} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', [\mathtt{x}, \mathtt{s}, e] :: k \rangle\}, \{e'[\mathtt{y} \mapsto w]\} \rangle$$

Figure 6: Abstract transition function

Keeping an explicit set of reachable expressions $\wp(C)$ 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 control-stack relation. However expressions may be in the control-stack-relation without ever being reached. An example hereof would be a diverging non-tail call.

5.2 A closure operator on machine states

Before continuing our calculation of the analysis, we recall a definition of a constituent relation due to Milner and Tofte [1991] and we formulate a structural order on expression-stack pairs.

Milner and Tofte's constituent relation 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.

An order on expression-stack pairs We furthermore need 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 frame of the first pair: $\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle \trianglerighteq \langle \mathtt{s}', k\rangle$. We write $\trianglerighteq^*$ 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 environments residing on the stack or nested within another environment, 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 5.1.

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

We need two helper lemmas relating the expression-stack ordering to the constituent relation. The first is proved by case analysis, and the second by structural induction (on the stack component).

Lemma 5.2. $\forall \langle \mathtt{s}, k\rangle, \langle \mathtt{s}', k'\rangle : \langle \mathtt{s}, k\rangle \trianglerighteq \langle \mathtt{s}', k'\rangle \implies k \succ k'$

Lemma 5.3. $\forall \langle \mathtt{s}, k\rangle, \langle \mathtt{s}', k'\rangle : \langle \mathtt{s}, k\rangle \trianglerighteq^* \langle \mathtt{s}', k'\rangle \implies k \succ^* k'$

Using the above lemmas, we can verify that ρ is a closure operator.

Lemma 5.4. *ρ is a closure operator*

We can now formulate an abstraction on the triples:

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

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:

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

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

Based on the closure operator-based Galois connection, we calculate a new intermediate transfer function. Now let $\langle C,\ F,\ E\rangle \in \rho(\wp(C) \times \wp(C \times K) \times \wp(Env))$ be given.

$$\rho\Big(\bigcup_{\substack{\langle\{\mathtt{t}\},\{\langle \mathtt{t},[\mathtt{x},\mathtt{s}',e']::k'\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ w\in\mu_c(\mathtt{t},\{e\})}}{}_{\times} \langle\{\mathtt{s}'\},\{\langle \mathtt{s}',k'\rangle\},\{e'[\mathtt{x}\mapsto w]\}\rangle\Big)$$

$$= \bigcup_{\substack{\langle\{\mathtt{t}\},\{\langle \mathtt{t},[\mathtt{x},\mathtt{s}',e']::k'\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ w\in\mu_c(\mathtt{t},\{e\})}}{}_{\rho} \rho(\langle\{\mathtt{s}'\},\{\langle \mathtt{s}',k'\rangle\},\{e'[\mathtt{x}\mapsto w]\}\rangle) \qquad (\rho \text{ a CJM})$$

$$= \bigcup_{\substack{\langle\{\mathtt{t}\},\{\langle \mathtt{t},[\mathtt{x},\mathtt{s}',e']::k'\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ w\in\mu_c(\mathtt{t},\{e\})}}{}_{\times} \rho(\langle\{\mathtt{s}'\},\{\langle \mathtt{s}',k'\rangle\},\{e'[\mathtt{x}\mapsto w]\}\rangle) \quad \text{(by observation)}$$

The resulting transfer function appears in Figure 7. This transfer function differs only minimally from the one in Figure 6, 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 5.2.

$$\forall C,F,E : \rho \circ F_\times \circ 1(\langle C,\ F,\ E\rangle) = F_\rho(\langle C,\ F,\ E\rangle)$$

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

$$F_\rho(\langle C,\ F,\ E\rangle) = \langle\{\mathtt{p}\},\ \{\langle \mathtt{p},\ [\mathtt{x_r},\ \mathtt{x_r},\ \bullet] :: \mathtt{stop}\rangle, \langle \mathtt{x_r},\ \mathtt{stop}\rangle\},\ \{\bullet\}\rangle$$

$$\cup_\times \bigcup_{\substack{\langle\{\mathtt{t}\},\{\langle \mathtt{t},[\mathtt{x},\mathtt{s}',e']::k'\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ w\in\mu_c(\mathtt{t},\{e\})}}{}_{\times} \rho(\langle\{\mathtt{s}'\},\{\langle \mathtt{s}',k'\rangle\},\{e'[\mathtt{x}\mapsto w]\}\rangle)$$

$$\cup_\times \bigcup_{\substack{\langle\{\mathtt{let\ x=t\ in\ s}\},\{\langle \mathtt{let\ x=t\ in\ s},k\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ w\in\mu_c(\mathtt{t},\{e\})}}{}_{\times} \rho(\langle\{\mathtt{s}\},\{\langle \mathtt{s},k\rangle\},\{e[\mathtt{x}\mapsto w]\}\rangle)$$

$$\cup_\times \bigcup_{\substack{\langle\{\mathtt{t_0\ t_1}\},\{\langle \mathtt{t_0\ t_1},k\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ [\mathtt{fn\ x\ =>\ s'},e']\in\mu_c(\mathtt{t_0},\{e\}) \\ w\in\mu_c(\mathtt{t_1},\{e\})}}{}_{\times} \rho(\langle\{\mathtt{s}'\},\{\langle \mathtt{s}',k\rangle\},\{e'[\mathtt{x}\mapsto w]\}\rangle)$$

$$\cup_\times \bigcup_{\substack{\langle\{\mathtt{let\ x=t_0\ t_1\ in\ s}\},\{\langle \mathtt{let\ x=t_0\ t_1\ in\ s},k\rangle\},\{e\}\rangle \subseteq_\times \langle C,F,E\rangle \\ [\mathtt{fn\ y\ =>\ s'},e']\in\mu_c(\mathtt{t_0},\{e\}) \\ w\in\mu_c(\mathtt{t_1},\{e\})}}{}_{\times} \rho(\langle\{\mathtt{s}'\},\{\langle \mathtt{s}',[\mathtt{x},\ \mathtt{s},\ e]::k\rangle\},\{e'[\mathtt{y}\mapsto w]\}\rangle)$$

Figure 7: The second abstract transition function

5.3 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 operating on expression-stack pairs.

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

$$\begin{aligned} @ &: C \times K \to C \times K^\sharp \\ @(\langle \mathtt{s}, \mathtt{stop} \rangle) &= \langle \mathtt{s}, \mathtt{stop} \rangle \\ @(\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k \rangle) &= \langle \mathtt{s}, [\mathtt{x}, \mathtt{s}'] \rangle \end{aligned}$$

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:

$$\langle \wp(C); \subseteq \rangle \underset{\alpha_@}{\overset{\gamma_@}{\leftrightarrows}} \langle \wp(A); \subseteq \rangle$$

$$\begin{aligned} \alpha_@(P) &= \{@(p) \mid p \in P\} \\ \gamma_@(Q) &= \{p \mid @(p) \in Q\} \end{aligned}$$

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

$$\langle \wp(A \times B); \subseteq \rangle \underset{\alpha_\omega}{\overset{\gamma_\omega}{\leftrightarrows\!\!\!\!\!\!\rightarrow}} \langle A \to \wp(B); \dot{\subseteq} \rangle$$

$$\begin{aligned} \alpha_\omega(r) &= \lambda a.\, \{b \mid \langle a, b \rangle \in r\} \\ \gamma_\omega(f) &= \{\langle a, b \rangle \mid b \in f(a)\} \end{aligned}$$

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

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

where $\alpha_{st} = \alpha_\omega \circ \alpha_@ = \lambda F. \dot{\bigcup}_{\langle \mathtt{s}, k \rangle \in F} \alpha_\omega(\{@(\langle \mathtt{s}, k \rangle)\})$ and $\gamma_{st} = \gamma_@ \circ \gamma_\omega$.

We can now prove a lemma relating the concrete and abstract expression-stack relations.

Lemma 5.5. *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.*

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

Proof. Assume $\{\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle\} \subseteq F$. Now $\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle \succ^* e$ and hence $e \in E$ by the assumption on E. Furthermore $\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle \succcurlyeq \langle \mathtt{s}', k\rangle$ hence $\{\langle \mathtt{s}', k\rangle\} \subseteq F$ by the assumption on F. For the last part we reason as follows:

$$
\begin{aligned}
&\implies \alpha_{st}(\{\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle\}) \mathbin{\dot\subseteq} \alpha_{st}(F) && (\alpha_{st} \text{ monotone})\\
&\iff \dot\bigcup_{\langle \mathtt{s}'', k''\rangle \in \{\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle\}} \alpha_\omega(\{@(\langle \mathtt{s}'', k''\rangle)\}) \mathbin{\dot\subseteq} \alpha_{st}(F) && (\text{def. } \alpha_{st})\\
&\iff \alpha_\omega(\{@(\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}', e] :: k\rangle)\}) \mathbin{\dot\subseteq} \alpha_{st}(F) && (\text{def. } \dot\cup)\\
&\iff \alpha_\omega(\{\langle \mathtt{s}, [\mathtt{x}, \mathtt{s}']\rangle\}) \mathbin{\dot\subseteq} \alpha_{st}(F) && (\text{def. } @)\\
&\iff \lambda\_.\, \emptyset[\mathtt{s} \mapsto \{[\mathtt{x}, \mathtt{s}']\}] \mathbin{\dot\subseteq} \alpha_{st}(F) && (\text{def. } \alpha_\omega)\\
&\iff \{[\mathtt{x}, \mathtt{s}']\} \subseteq \alpha_{st}(F)(\mathtt{s}) && (\text{def. } \dot\subseteq)
\end{aligned}
$$

□

5.4 Abstracting values

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

$$Val^\sharp \ni w^\sharp ::= c \mid [\mathtt{fn\ x\ =>\ s}] \qquad \text{(abstract values)}$$

$$
\begin{aligned}
@ &: Val \to Val^\sharp\\
@(c) &= c\\
@([\mathtt{fn\ x\ =>\ s}, e]) &= [\mathtt{fn\ x\ =>\ s}]
\end{aligned}
$$

5.5 Abstracting environments

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:

$$\langle \wp(D \to C); \subseteq\rangle \underset{\alpha_\Pi}{\overset{\gamma_\Pi}{\leftrightarrows}} \langle D \to C^\sharp; \dot\sqsubseteq\rangle$$

$$
\begin{aligned}
\alpha_\Pi(F) &= \lambda d.\, \alpha(\{f(d) \mid f \in F\})\\
\gamma_\Pi(A) &= \{f \mid \forall d : f(d) \in \gamma(A(d))\}
\end{aligned}
$$

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:

$$\langle \wp(C); \subseteq\rangle \underset{\alpha_\subset}{\overset{\gamma_\subset}{\leftrightarrows}} \langle \wp(A); \subseteq\rangle$$

$$\alpha_{\subset}(X) = X \cap A$$
$$\gamma_{\subset}(Y) = Y \cup (\neg A) \qquad \text{where } \neg A = C \setminus A$$

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 $c \in C$. Now consider environments $e \in \mathit{Var} \rightharpoonup \mathit{Val}$ to be total functions $\mathit{Var} \to (\mathit{Val} \cup \perp)$ using this idea. In this context the bottom element $\perp$ will denote variable lookup failure. Now compose a subset abstraction $\wp(\mathit{Val} \cup \perp) \underset{\alpha_{\subset}}{\overset{\gamma_{\subset}}{\leftrightarrows}} \wp(\mathit{Val})$ with the value abstraction from the previous section, and feed the result to the pointwise abstraction above. The result is a pointwise abstraction of a set of environments, that does not explicitly model variable lookup failure:

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

By considering only closed programs, we statically ensure against failure of variable-lookup, hence disregarding $\perp$ loses no information.

5.6 Abstracting the helper function

We calculate an abstract helper function, by "pushing α's" under the function definition, and reading off a resulting abstract definition.

Lemma 5.6. *Abstract helper function*

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

The resulting helper function reads:

$$\mu^\sharp : T \times \mathit{Env}^\sharp \to \wp(\mathit{Val}^\sharp)$$
$$\mu^\sharp(c, E^\sharp) = \{c\}$$
$$\mu^\sharp(\mathtt{x}, E^\sharp) = E^\sharp(\mathtt{x})$$
$$\mu^\sharp(\texttt{fn x => s}, E^\sharp) = \{[\texttt{fn x => s}]\}$$

where we write $\mathit{Env}^\sharp$ as shorthand for $\mathit{Var} \to \wp(\mathit{Val}^\sharp)$.

We will furthermore need a lemma relating the two helper function definitions on closed environments.

Lemma 5.7. *Helper function on closed environments (1)*
Let $\langle C, F, E\rangle \in \rho(\wp(C) \times \wp(C \times K) \times \wp(\mathit{Env}))$ 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 5.8. *Helper function on closed environments (2)*
Let $\langle C, F, E\rangle \in \rho(\wp(C) \times \wp(C \times K) \times \wp(\mathit{Env}))$ be given.

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

5.7 Abstracting the machine states

We abstract the triplet of sets by abstract triples using 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 use the notation $\cup_\otimes$ and $\subseteq_\otimes$ to denote 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 relation $\wp(C \times K)$ we use the expression-stack abstraction developed in Section 5.3. For the set of environments $\wp(Env)$ we use the environment abstraction developed in Section 5.5.

6 Calculating the analysis

We 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. For space-saving purposes the calculation is divided into a number of observations, on which the derivation relies. Let $\langle C, F, E \rangle \in \rho(\wp(C) \times \wp(C \times K) \times \wp(Env))$ be given. First observe that:

$$\begin{aligned}
&\{e \mid \exists \langle \mathtt{s}, k \rangle \in F : \langle \mathtt{s}, k \rangle \succ^* e \\
&\qquad \vee \ \exists e' \in (\bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}} \{e'[\mathtt{x} \mapsto w]\}) : e' \succ^* e\} \\
&= \{e \mid \exists \langle \mathtt{s}, k \rangle \in F : \langle \mathtt{s}, k \rangle \succ^* e \\
&\qquad \vee \ \exists e' \in E, w \in \mu_c(\mathtt{t}, E) : e'[\mathtt{x} \mapsto w] \succ^* e\} && \text{(def. } \cup\text{)} \\
&= \{e \mid \exists \langle \mathtt{s}, k \rangle \in F : \langle \mathtt{s}, k \rangle \succ^* e\} \\
&\qquad \cup \{e \mid \exists e' \in E, w \in \mu_c(\mathtt{t}, E) : e'[\mathtt{x} \mapsto w] \succ^* e\} && \text{(def. } \vee\text{)} \\
&\subseteq E \cup \{e \mid \exists e' \in E, w \in \mu_c(\mathtt{t}, E) : e'[\mathtt{x} \mapsto w] \succ^* e\} && \text{(assumption on } E\text{)} \\
&= E \cup \{e \mid \exists e' \in E, w \in \mu_c(\mathtt{t}, E) : e'[\mathtt{x} \mapsto w] = e \\
&\qquad \vee \ e' \succ^* e \ \vee \ w \succ^* e\} && \text{(case analysis)} \\
&= E \cup \{e \mid \exists e' \in E, w \in \mu_c(\mathtt{t}, E) : e'[\mathtt{x} \mapsto w] = e \\
&\qquad \vee \ e' \succ^* e\} && \text{(by Lemma 5.8)}
\end{aligned}$$

$$= E \cup \{ e \mid \exists e' \in E, w \in \mu_c(\mathtt{t}, E) : e'[\mathtt{x} \mapsto w] = e \} \qquad \text{(assumption on } E\text{)}$$
$$= E \cup \{ e'[\mathtt{x} \mapsto w] \mid e' \in E, w \in \mu_c(\mathtt{t}, E) \} \qquad \text{(def =)}$$

Secondly, observe that:

$$\bigcup_{\substack{\{\langle \mathtt{s}', k' \rangle\} \subseteq F \\ \{e'\} \subseteq E \quad \{e\} \subseteq E \\ w \in \mu_c(\mathtt{t}, \{e\})}}{}_{\times} \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle)$$
$$= \bigcup_{\substack{\{\langle \mathtt{s}', k' \rangle\} \subseteq F \\ \{e'\} \subseteq E \quad w \in \mu_c(\mathtt{t}, E)}}{}_{\times} \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle) \qquad \text{(def. } \mu_c\text{)}$$
$$= \bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}}{}_{\times} \rho(\bigcup_{\{\langle \mathtt{s}', k' \rangle\} \subseteq F}{}_{\times} \langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle) \qquad (\rho \text{ a CJM})$$
$$= \bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}}{}_{\times} \rho(\langle \{\mathtt{s}'\}, \bigcup_{\{\langle \mathtt{s}', k' \rangle\} \subseteq F} \{\langle \mathtt{s}', k' \rangle\}, \{e'[\mathtt{x} \mapsto w]\} \rangle) \qquad \text{(def. } \cup_\times\text{)}$$
$$\subseteq_\times \bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}}{}_{\times} \rho(\langle \{\mathtt{s}'\}, F, \{e'[\mathtt{x} \mapsto w]\} \rangle) \qquad \text{(def. } \cup\text{)}$$
$$= \rho(\bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}}{}_{\times} \langle \{\mathtt{s}'\}, F, \{e'[\mathtt{x} \mapsto w]\} \rangle) \qquad (\rho \text{ a CJM})$$
$$= \rho(\langle \{\mathtt{s}'\}, F, \bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}} \{e'[\mathtt{x} \mapsto w]\} \rangle) \qquad \text{(def. } \cup_\times\text{)}$$
$$= \left\langle \{\mathtt{s}'\}, \begin{array}{l} \{\langle \mathtt{s}, k \rangle \mid \exists \langle \mathtt{s}', k' \rangle \in F : \langle \mathtt{s}', k' \rangle \succcurlyeq^* \langle \mathtt{s}, k \rangle\}, \\ \{e \mid \exists \langle \mathtt{s}, k \rangle \in F : \langle \mathtt{s}, k \rangle \succ^* e \\ \qquad \vee \exists e' \in \bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}} \{e'[\mathtt{x} \mapsto w]\} : e' \succ^* e\} \end{array} \right\rangle \qquad \text{(def. } \rho\text{)}$$
$$= \left\langle \{\mathtt{s}'\}, F, \begin{array}{l} \{e \mid \exists \langle \mathtt{s}, k \rangle \in F : \langle \mathtt{s}, k \rangle \succ^* e \\ \qquad \vee \exists e' \in \bigcup_{\substack{\{e'\} \subseteq E \\ w \in \mu_c(\mathtt{t}, E)}} \{e'[\mathtt{x} \mapsto w]\} : e' \succ^* e\} \end{array} \right\rangle \qquad \text{(assumption on } F\text{)}$$
$$\subseteq_\times \langle \{\mathtt{s}'\}, F, E \cup \{ e'[\mathtt{x} \mapsto w] \mid e' \in E, w \in \mu_c(\mathtt{t}, E) \} \rangle \qquad \text{(First obs.)}$$

Thirdly, observe that:

$$\begin{array}{l} \alpha_\Pi(E \cup \{ e'[\mathtt{x} \mapsto w] \mid e' \in E, w \in \mu_c(\mathtt{t}, E) \}) \\ = \alpha_\Pi(E) \dot\cup \alpha_\Pi(\{ e'[\mathtt{x} \mapsto w] \mid e' \in E, w \in \mu_c(\mathtt{t}, E) \}) \qquad (\alpha_\Pi \text{ a CJM}) \\ = \alpha_\Pi(E) \dot\cup \alpha_\Pi(\{ \lambda \mathtt{y}.\, \text{if } \mathtt{y} = \mathtt{x} \text{ then } w \text{ else } e'(\mathtt{y}) \mid e' \in E, \\ \qquad\qquad w \in \mu_c(\mathtt{t}, E) \}) \qquad \text{(def. extend)} \\ = \alpha_\Pi(E) \dot\cup \lambda \mathtt{y}.\, \text{if } \mathtt{y} = \mathtt{x} \text{ then } \alpha_@(\{ w \mid w \in \mu_c(\mathtt{t}, E) \}) \\ \qquad\qquad \text{else } \alpha_@(\{ e'(\mathtt{y}) \mid e' \in E \}) \qquad \text{(def. } \alpha_\Pi\text{)} \\ = \alpha_\Pi(E) \dot\cup \lambda \mathtt{y}.\, \text{if } \mathtt{y} = \mathtt{x} \text{ then } \alpha_@(\mu_c(\mathtt{t}, E)) \text{ else } \alpha_\Pi(E)(\mathtt{y}) \qquad \text{(def. } \alpha_\Pi\text{)} \end{array}$$

$$= \alpha_\Pi(E) \dot\cup \lambda \mathtt{y}.\, \text{if } \mathtt{y} = \mathtt{x} \text{ then } \mu^\sharp(\mathtt{t}, \alpha_\Pi(E)) \text{ else } \alpha_\Pi(E)(\mathtt{y}) \quad \text{(by Lemma 5.6)}$$
$$= \alpha_\Pi(E) \dot\cup \alpha_\Pi(E)[\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, \alpha_\Pi(E))] \quad \text{(def. extend)}$$
$$= \alpha_\Pi(E) \dot\cup [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, \alpha_\Pi(E))] \quad \text{(def. } \dot\cup\text{)}$$

where we have written $[\mathtt{x} \mapsto \ldots]$ as shorthand for $\lambda\_.\, \emptyset[\mathtt{x} \mapsto \ldots]$. Now we can calculate the analysis:

$$\alpha_\otimes\Big(\bigcup_{\times\ \langle \{\mathtt{t}\}, \{\langle \mathtt{t}, [\mathtt{x}, \mathtt{s}', e'] :: k' \rangle\}, \{e\}\rangle \subseteq_\times \langle C, F, E\rangle \atop w \in \mu_c(\mathtt{t}, \{e\})} \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k'\rangle\}, \{e'[\mathtt{x} \mapsto w]\}\rangle)\Big)$$
$$= \alpha_\otimes\Big(\bigcup_{\times\ \{\mathtt{t}\} \subseteq C \atop \{\langle \mathtt{t}, [\mathtt{x}, \mathtt{s}', e'] :: k' \rangle\} \subseteq F \atop \{e\} \subseteq E \atop w \in \mu_c(\mathtt{t}, \{e\})} \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k'\rangle\}, \{e'[\mathtt{x} \mapsto w]\}\rangle)\Big) \quad \text{(def. } \subseteq_\times\text{)}$$
$$\subseteq_\otimes \alpha_\otimes\Big(\bigcup_{\times\ \{\mathtt{t}\} \subseteq C \atop \{[\mathtt{x}, \mathtt{s}']\} \subseteq \alpha_{st}(F)(\mathtt{t}) \atop \{\langle \mathtt{s}', k'\rangle\} \subseteq F \atop \{e'\} \subseteq E \quad \{e\} \subseteq E \atop w \in \mu_c(\mathtt{t}, \{e\})} \rho(\langle \{\mathtt{s}'\}, \{\langle \mathtt{s}', k'\rangle\}, \{e'[\mathtt{x} \mapsto w]\}\rangle)\Big) \quad \text{(by Lemma 5.5)}$$
$$\subseteq_\otimes \alpha_\otimes\Big(\bigcup_{\times\ \{\mathtt{t}\} \subseteq C \atop \{[\mathtt{x}, \mathtt{s}']\} \subseteq \alpha_{st}(F)(\mathtt{t})} \langle \{\mathtt{s}'\}, F, E \cup \{e'[\mathtt{x} \mapsto w] \mid e' \in E, w \in \mu_c(\mathtt{t}, E)\}\rangle\Big) \quad \text{(Second obs.)}$$
$$= \bigcup_{\otimes\ \{\mathtt{t}\} \subseteq C \atop \{[\mathtt{x}, \mathtt{s}']\} \subseteq \alpha_{st}(F)(\mathtt{t})} \alpha_\otimes(\langle \{\mathtt{s}'\}, F, E \cup \{e'[\mathtt{x} \mapsto w] \mid e' \in E, w \in \mu_c(\mathtt{t}, E)\}\rangle) \quad (\alpha_\otimes \text{ a CJM})$$
$$= \bigcup_{\otimes\ \{\mathtt{t}\} \subseteq C \atop \{[\mathtt{x}, \mathtt{s}']\} \subseteq \alpha_{st}(F)(\mathtt{t})} \langle \{\mathtt{s}'\}, \alpha_{st}(F), \alpha_\Pi(E \cup \{e'[\mathtt{x} \mapsto w] \mid e' \in E, w \in \mu_c(\mathtt{t}, E)\})\rangle \quad \text{(def. } \alpha_\otimes\text{)}$$
$$= \bigcup_{\times\ \{\mathtt{t}\} \subseteq C \atop \{[\mathtt{x}, \mathtt{s}']\} \subseteq \alpha_{st}(F)(\mathtt{t})} \langle \{\mathtt{s}'\}, \alpha_{st}(F), \alpha_\Pi(E) \dot\cup [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, \alpha_\Pi(E))]\rangle \quad \text{(Third obs.)}$$

The resulting analysis appears in Figure 8. As a corollary of the construction, the analysis satisfies the following, i.e., the analysis safely approximates the reachable states of the abstract machine.

Corollary 6.1. $\alpha_\otimes \circ \rho \circ \alpha_\times(\mathrm{lfp}\, F) \subseteq_\otimes \mathrm{lfp}\, F_\sharp$

6.1 A faster implementation

The analysis as formulated above will always terminate, as there are only a finite number of reachable expressions, variables and functions in a given program. Hence strictly speaking we do not need a widening operator. However to avoid computing redundant joins, one typically computes an equivalent sequence sharing the same fixed point:

$$X_0 = \langle \emptyset, \lambda\_.\, \emptyset, \lambda\_.\, \emptyset \rangle$$

$$F_\sharp : \wp(C) \times (C \to \wp(K^\sharp)) \times Env^\sharp \to \wp(C) \times (C \to \wp(K^\sharp)) \times Env^\sharp$$

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

Figure 8: The resulting analysis function

$$X_{i+1} = X_i \cup_\otimes F_\sharp(X_i)$$

where we use a join $\cup_\otimes$ as the widening operator.

6.2 Characteristics

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, Biswas, 1997, Gasser et al., 1997]. In our case they arise naturally from a projecting abstraction of a reachable states collecting semantics.

Second of all 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 literally captures this phenomenon in two approximation functions.

Third of all the analysis is *"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 *"caller-restore"* presentations [Palsberg, 1995, Nielson et al., 1999] where the caller propagates the obtained return values from the body of the function to the call site (typically formulated as *conditional constraints*).

7 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 below formulation 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 $\{u\}$, $F^\sharp(\mathtt{s})$, or $E^\sharp(\mathtt{x})$, right-hand sides *rhs* can be of the form C, $F^\sharp(\mathtt{s})$, 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 Figure 8 we directly read off the following constraints.

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

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

- For each return expression $\mathtt{t}$ and non-tail call $\mathtt{let\ x{=}t_0\ t_1\ in\ s'}$ in $\mathtt{p}$:

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

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

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

- For each tail call $\mathtt{t_0\ t_1}$ and function $\mathtt{fn\ x\ =>\ s'}$ in $\mathtt{p}$:

$$\{\mathtt{t_0\ t_1}\} \subseteq C \ \wedge \ \{[\mathtt{fn\ x\ =>\ s'}]\} \subseteq \mu_{sym}(\mathtt{t_0}, E^\sharp) \Rightarrow \begin{cases} \{\mathtt{s'}\} \subseteq C \ \wedge \\ F^\sharp(\mathtt{t_0\ t_1}) \subseteq F^\sharp(\mathtt{s'}) \ \wedge \\ \mu_{sym}(\mathtt{t_1}, E^\sharp) \subseteq E^\sharp(\mathtt{x}) \end{cases}$$

- For each non-tail call $\mathtt{s_{nt}} = \mathtt{let\ x{=}t_0\ t_1\ in\ s}$ and function $\mathtt{fn\ y\ =>\ s'}$ in $\mathtt{p}$:

$$\{\mathtt{s_{nt}}\} \subseteq C \ \wedge \ \{[\mathtt{fn\ y\ =>\ s'}]\} \subseteq \mu_{sym}(\mathtt{t_0}, E^\sharp) \Rightarrow \begin{cases} \{\mathtt{s'}\} \subseteq C \ \wedge \\ F^\sharp(\mathtt{s_{nt}}) \subseteq F^\sharp(\mathtt{s}) \ \wedge \\ \{[\mathtt{x},\ \mathtt{s}]\} \subseteq F^\sharp(\mathtt{s'}) \ \wedge \\ \mu_{sym}(\mathtt{t_1}, E^\sharp) \subseteq E^\sharp(\mathtt{y}) \end{cases}$$

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 to generate a singleton for constants and lambda expressions. The definition

of the symbolic helper function otherwise coincides with the abstract helper function $\mu^\sharp$:

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

The alert reader may have noticed that we generate constraints of the form $\{[\mathtt{fn\ x\ =>\ s}]\} \subseteq \{[\mathtt{fn\ y\ =>\ s'}]\}$, which are not covered by the above grammar. We therefore first pre-process the constraints in linear time:

- removing vacuously true inclusions $\{[\mathtt{fn\ x\ =>\ s}]\} \subseteq \{[\mathtt{fn\ x\ =>\ s}]\}$ from each constraint, and
- removing constraints with vacuously false preconditions $\{[\mathtt{fn\ x\ =>\ s}]\} \subseteq \{w^\sharp\}$, where $[\mathtt{fn\ y\ =>\ s'}] \neq w^\sharp$.

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 7.1. *A solution to the CFA constraints of program* $\mathtt{p}$ *is a safe approximation of the least fixpoint of the analysis function* $F_\sharp$ *induced by* $\mathtt{p}$*. Furthermore, the least solution to the CFA constraints is equal to the least fixpoint of* $F_\sharp$*.*

Proof. The first part of the theorem is proved by showing that a solution to the CFA constraints $\langle C, F, E\rangle$ is a post-fixpoint of $F_\sharp$, *i.e.*, that it satisfies $F_\sharp(\langle C, F, E\rangle) \subseteq_\otimes \langle C, F, E\rangle$ and then appeal to the Knaster-Tarski fixpoint theorem that the least fixpoint of a monotone operator $F_\sharp$ is the greatest lower bound of the set of post-fixpoints of $F_\sharp$. This reduces to showing that for each of the expressions defining $F_\sharp$ in Figure 8 we have that its value is already included in the solution $\langle C, F, E\rangle$. For example, for the expression

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

we must have, for all $\mathtt{t}$ satisfying $\{\mathtt{t}\} \subseteq C$ and $\mathtt{s'}$ satisfying $\{[\mathtt{x}, \mathtt{s'}]\} \subseteq F^\sharp(\mathtt{t})$, that

$$\{\mathtt{s'}\} \subseteq C \quad \text{and} \quad E^\sharp \dot\cup [\mathtt{x} \mapsto \mu^\sharp(\mathtt{t}, E^\sharp)] \dot\subseteq E^\sharp.$$

The latter inequality reduces to $\mu_{sym}(\mathtt{t}, E^\sharp) \subseteq E^\sharp(\mathtt{x})$. and we obtain exactly the constraints for return expressions. The other cases follow by similar reasoning.

For the equality of the least solution and the least fixpoint, it then suffices to prove that the fixpoint is a solution to the CFA constraints. The argumentation is again based on unfolding the definition of $F_\sharp$ and using reasoning similar to above. □

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

or label instead of the sub-expression itself, a single constraint takes only constant space. By generating $O(n^2)$ constraints, linear post-processing, and iteratively solving them using a well-known algorithm [Palsberg and Schwartzbach, 1995, Gasser et al., 1997, 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 caller-restore nature, i.e., propagation of return-flow from the function body is handled at the call-site. The reachability aspect of the extracted constraint is similar to Gasser et al. [1997] modulo an isomorphic encoding $\wp(C) \simeq C \to \wp(\{\mathbf{on}\})$ reminiscent of a characteristic set.

8 Applications of the analysis

In a compiler a 0-CFA can be used for a number of transformations and optimizations. As an example we can remove from the program any expression which is not reachable $\mathtt{s} \notin C$. In CPS where everything is a call, 0-CFA lends itself to a number of call optimizations. Fluet and Weeks [2001] coined the term *contification* for the transformation that turns a function into a continuation. In the words of Kennedy [2007]:

> Sometimes it is the case that a function can be transformed into a continuation, a process known as *contification*. This is possible exactly when the function always returns to the same place.

This condition is exactly the property that our analysis computes!

By appealing to the ANF-CPS isomorphism [Danvy, 1994] we formulate an equivalent condition for ANF: A function always returning to the same place can be transformed into a function representing *the rest of the computation*, i.e., turning non-tail calls into tail calls. Recall the example from the introduction:

```
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
```

Notice that the two calls to `k` are in tail-position: when either of the two calls return, control continues by binding the intermediate result to `y` and to the outer call to `g`.

When evaluated in some environment where `b` is bound, our analysis (straightforwardly extended with conditionals) determines that $F^\sharp(\mathtt{x}) = \{[\mathtt{y},\ \mathtt{g\ y}]\}$, i.e., the function `fn x => x` will always return to the same `let`-binding, and hence we can *inline* the rest of the computation in the function body:

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

The transformation lends itself to further optimizations: the let-binding can be eliminated, `g` can be inlined, etc.

Traditionally, a compiler may decide to inline a particular function call $\mathtt{t}_0\,\mathtt{t}_1$, if a 0-CFA can determine that only one particular lambda can be called: $\mu^\sharp(\mathtt{t}_0, E^\sharp) = \{[\mathtt{fn\ x\ =>\ s}]\}$, provided that the values of any free variables of the function are available in the lexical scope of the call-site. Dually, a compiler should be able to inline a particular function return `t`, if an analysis determines that it will always return to the same point: $F^\sharp(\mathtt{t}) = \{[\mathtt{x}, \mathtt{s}]\}$, provided that the values of any free variables of the rest of the computation are available in the lexical scope of the function-body. This idea is precisely the higher-order, direct-style version of the contification transformation described above.

Determining that two expressions always agree on the values of their free variables when evaluated is itself an interesting problem. A crude but correct condition is to prohibit inlining in the presence of free variables. A better approximation would be to allow inlining only when the values of any free variables are constant, e.g., when they denote top-level functions. More powerful flow analysis techniques have been pursued by Steckler and Wand [1997] and more recently by Might and Shivers [2006].

Alternatively, if a 0-CFA determines that only one particular lambda is called at a particular call-site, the compiler can generate a *direct call*, rather than an *indirect call* to an extracted lambda-expression of a closure. Dually, if our analysis determines that a particular function will always return to the same point, the compiler can generate a *direct return*, i.e., a direct jump and a call stack pop, rather than an *indirect return* through a code pointer stored on the call stack.

Debray and Proebsting [1997] list a number of applications of CFA: most notable the creation of interprocedural control-flow graphs, which in turn enable an optimization like interprocedural unboxing. An alternative optimization enabled by CFA is interprocedural basic block fusion, which bears a strong resemblance to direct-style contification as described above.

Formulating a CFA as traditional abstract interpretation furthermore allows us to integrate the CFA-domains with other domains and analyses. Hence it should be possible to formulate interval, polyhedra, or octagon analyses to higher-order functional programs using an approach similar to Nielson et al. [1999, Ch.3].

Fluet and Weeks [2001] defined the *contification* transformation for a first-order language. Furthermore they developed an optimal algorithm based on dominators. Kennedy [2007] formulated a *local contification* transformation for a higher-order language in CPS. Debray and Proebsting [1997] studied control-flow analysis for a tail-call optimized first-order language. They showed how the problem corresponds to traditional concepts from parsing theory. In this light, one can regard the current paper as a higher-order counterpart of Debray and Proebsting's first-order tail call-optimized 0-CFA.

9 Discussion

In this presentation we did not include an explicit construct for recursive functions. Since our source language is untyped, it is possible to encode recursion though fixed-point operators. Explicit recursion is typically modelled by circu-

lar environments. The current formulation extends straight-forwardly to handle those, because of our two-staged environment abstraction (closure operator and pointwise extended value abstraction).

The CFAs that we are aware of, all use a caller-restore strategy, typically mimicking the recursive nature of a corresponding interpreter, e.g., a big-step or denotational semantics. Such caller-restore CFAs need not abstract the call stack. In our case our starting point was a callee-restore machine with an explicit call stack. In hindsight it is perhaps less surprising that the "abstract interpreter" inherits this callee-restore strategy.

10 Conclusion

We have presented a control-flow analysis determining interprocedural control-flow of both calls and returns for a functional language in direct style. The analysis was developed systematically using calculational Galois connection-based abstract interpretation of a standard operational semantics for that language. Furthermore we extracted from the analysis an equivalent constraint-based formulation expressed in terms of conditional constraints. We have developed a prototype implementation of the resulting analysis in OCaml.[2]

Existing CFAs have focused on analysing which functions are called at a given call site. In contrast the systematic derivation leads to an analysis that provides extra information about where a function returns to at no additional cost. Such information enables both the creation of more precise call graphs in the presence of tail-call optimization, as well as the *contification* transformation of direct-style programs. The formulation of the analysis is furthermore pedagogically pleasing since monomorphism is made explicit through two Galois connections: one literally merges all bindings to the same variable and one literally merges all calling contexts of the same function.

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. It would also seem possible (though a substantial endeavour) to apply our derivational technique to an operational semantics of *e.g.*, Java byte code in order to provide a systematic (re-)construction of flow analyses for such a language.

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.

Acknowledgement: The authors thank Matthew Fluet for commenting on an earlier draft of this paper.

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

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.

S. K. Biswas. A demand-driven set-based analysis. In Jones, pages 372–385.

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, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998.

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, *Proceedings of the Sixth International Conference on Algebraic Methodology and Software Technology, AMAST '97*, volume 1349 of *Lecture Notes in Computer Science*, 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, *Proceedings 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. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In B. K. Rosen, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, Jan. 1979.

O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3): 183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

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, *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations (with retrospective). In McKinley [2004], pages 502–514.

M. Fluet and S. Weeks. Contification using dominators. In X. Leroy, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 2–13, Firenze, Italy, Sept. 2001.

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

N. Heintze. Set-based program analysis of ML programs. In C. L. Talcott, editor, *Proceedings 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 *Lecture Notes in Computer Science*, pages 114–128, Israel, July 1981. Springer-Verlag.

N. D. Jones, editor. *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997.

A. Kennedy. Compiling with continuations, continued. In N. Ramsey, editor, *Proceedings of the Twelfth ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 177–190, Freiburg, Germany, Oct. 2007.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

K. S. McKinley, editor. *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979–1999, A Selection*, 2004.

J. Midtgaard. Control-flow analysis of functional programs. Technical Report BRICS RS-07-18, DAIMI, Department of Computer Science, 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 *Lecture Notes in Computer Science*, pages 347–362, Valencia, Spain, July 2008. Springer-Verlag.

M. Might and O. Shivers. Environmental analysis via ΔCFA. In S. Peyton Jones, editor, *Proceedings 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, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

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.

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

J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

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

O. Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned (with retrospective). In McKinley [2004], pages 257–269.

O. Shivers. Control-flow analysis in Scheme. In M. D. Schwartz, editor, *Proceedings 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.

P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

A Underlying mathematical material

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

A partially ordered set (poset) $\langle S; \sqsubseteq \rangle$ is a set S equipped with a partial order $\sqsubseteq$. A complete lattice is a poset $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$, 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 $f \dot{\sqsubseteq} f' \iff \forall x. f(x) \sqsubseteq f'(x)$, and 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 (for all $c, c' \in C : c \sqsubseteq c' \implies \alpha(c) \leq \alpha(c')$ and for all $a, a' \in A : a \leq a' \implies \gamma(a) \sqsubseteq \gamma(a')$), (b) $\alpha \circ \gamma$ is reductive (for all $a \in A : \alpha \circ \gamma(a) \leq a$), and (c) $\gamma \circ \alpha$ is extensive (for all $c \in C : c \sqsubseteq \gamma \circ \alpha(c)$). Galois connections are typeset as $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \leq \rangle$. We omit the orderings when they are clear from the context. For a Galois connection between two complete lattices $\langle C; \sqsubseteq, \bot_c, \top_c, \sqcup, \sqcap \rangle$ and $\langle A; \leq, \bot_a, \top_a, \vee, \wedge \rangle$, α is a complete join-morphism (CJM) (for all $S_c \subseteq C : \alpha(\sqcup S_c) = \vee \alpha(S_c) = \vee \{\alpha(c) \mid c \in S_c\}$) and γ is a complete meet morphism (for all $S_a \subseteq A : \gamma(\wedge S_a) = \sqcap \gamma(S_a) = \sqcap \{\gamma(a) \mid a \in S_a\}$). 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 as: $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow}} \langle A; \leq \rangle$. Galois connections in which γ is surjective (or equivalently α is injective) are typeset as: $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftarrow\!\!\!\!\!\rightarrow}} \langle A; \leq \rangle$. 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 $\{\rho(s) \mid s \in S\}$. 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 $\langle \rho(C); \sqsubseteq, \rho(\bot), \top, \lambda X.\, \rho(\sqcup X), \sqcap \rangle$.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399