Is Continuation-Passing Useful for Data Flow Analysis?

Amr Sabry* Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892†

Abstract

The widespread use of the continuation-passing style (CPS) transformation in compilers, optimizers, abstract interpreters, and partial evaluators reflects a common belief that the transformation has a positive effect on the analysis of programs. Investigations by Nielson [13] and Burn/Filho [5, 6] support, to some degree, this belief with theoretical results. However, they do not pinpoint the source of increased abstract information and do not explain the observation of many people that continuation-passing confuses some conventional data flow analyses.

To study the impact of the CPS transformation on program analysis, we derive three canonical data flow analyzers for the core of an applicative higher-order programming language. The first analyzer is based on a direct semantics of the language, the second on a continuation-semantics of the language, and the last on the direct semantics of CPS terms. All analyzers compute the control flow graph of the source program and hence our results apply to a large class of data flow analyses. A comparison of the information gathered by our analyzers establishes the following points:

1. The results of a direct analysis of a source program are *incomparable* to the results of an analysis of the equivalent CPS program. In other words, the translation of the source program to a CPS version may increase or decrease static information. The gain of information occurs in non-distributive analyses and is solely due to the *duplication* of the analysis of the continuation. The loss of information is due to the confusion of distinct procedure returns.

2. The analyzer based on the continuation semantics produces more accurate results than both direct analyzers, but again only in non-distributive analyses due to the *duplication* of continuations along every execution path. However, when the analyzer explicitly accounts for looping constructs, the results of the semantic-CPS analysis are no longer computable.

In view of these results, we argue that, in practice, a direct data flow analysis that relies on some amount of duplication would be as satisfactory as a CPS analysis.

1 Compiling with CPS

Many compilers for higher-order applicative languages (Scheme, ML, Common Lisp) map source programs to programs in continuation-passing style (CPS) [1, 10, 11, 17]. Compiler writers believe that the intermediate representation based on CPS eases the production of code and facilitates optimizations. Numerous people also argue that the CPS transformation increases the precision of the data flow analysis that is necessary for advanced optimizations [2, 3, 5, 6, 16].[1]

Even though CPS programs are widely accepted as an advantageous intermediate representation, few compiler writers can pinpoint the advantages of the CPS representation over other intermediate representations.

*Supported in part by NSF grant CCR 89-17022 and by Texas ATP grant 91-003604014. Also partially supported by Arpa grant 8313, issued by ESD/AVS under Contract No. F196228-91-C-0168 under the direction of Robert Harper and Peter Lee.

†Temporary address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[1] Researchers often express this view in informal discussions as opposed to formal papers. We discussed and re-confirmed this idea with, among others, Charles Consel, and Olivier Danvy at LFP '92 [June 92], following the presentation of our paper on equational reasoning about programs in CPS [14]; in an email exchange with Geoffrey Burn and Juarez Filho [July 92]; in further discussions at POPL '93 with Daniel Weise; in email discussions with Kelsey [July 93] and Shivers [May 93]; and in discussions with Burn at FPCA '93 [June 93].

In an attempt to understand the principles of compiling with continuations and to determine its crucial properties, we recently found two important results:

1. The theory of equational manipulations of programs in CPS based on the β rule of the lambda calculus has a simple counterpart for source programs; in other words, whatever optimizations are expressible via β-steps on CPS programs, can equally well be formulated for source programs [14].

2. The code generation phase of a non-optimizing CPS compiler only requires that the intermediate representation be normalized according to simple transformations. It is unnecessary to perform a full CPS translation [7].

Prompted by conversations with G. Burn [July 92] and with H. Boehm [August 92], and by the observation that the CPS transformation obscures some obvious properties of programs, we started to investigate the exact effect of the CPS program representation on the data flow analysis of programs. Our answer rejects the common belief that the CPS transformation is useful for practical analyses.

The organization of the rest of this paper is as follows. The next section introduces the syntax and semantics of a typical higher-order language. Section 3 presents the CPS transformation. In Section 4, we derive various data flow analyzers from the standard semantics using the method of abstract interpretation. Section 5 presents all the formal theorems, which are discussed from a practical perspective in Section 6. For full proofs, we refer the reader to the technical report version of our paper [15].

2 Λ: Syntax and Semantics

Our source language is a simple extension of the language Λ of the λ-calculus. It corresponds to the core of typical higher-order languages like Scheme, Lisp, and ML. The set of terms includes values, applications, let-expressions, and conditionals; the set of syntactic values contains numerals ($n \in \mathbb{Z}$), variables ($x \in Var$), primitive procedures, and user-defined procedures:

$$M ::= V \mid (M\ M) \mid (\mathsf{let}\ (x\ M)\ M) \mid (\mathsf{if0}\ M\ M\ M)$$
$$V ::= n \mid x \mid \mathsf{add1} \mid \mathsf{sub1} \mid (\lambda x.M)$$

Informally, the semantics of the language is as follows. All procedures are call-by-value, the evaluation of $(\mathsf{let}\ (x\ M_1)\ M_2)$ computes the value of M_1 and binds it to x in M_2, and the conditional if0 branches to the second or third subexpression depending on whether the first subexpression evaluates to 0 or not. Though the language is overly simple, it is rich enough for the primary purpose of the paper.

The analysis of programs assumes that every intermediate result is named, and that all bound variables in a program are unique. The restricted subset is the following language:

$$\begin{aligned} M &::= V \\ &\mid (\mathsf{let}\ (x\ V)\ M) \\ &\mid (\mathsf{let}\ (x\ (V\ V))\ M) \\ &\mid (\mathsf{let}\ (x\ (\mathsf{if0}\ V\ M\ M))\ M) \\ V &::= n \mid x \mid \mathsf{add1} \mid \mathsf{sub1} \mid (\lambda x.M) \end{aligned}$$

The assumptions simplify the semantics and the derivation of the data flow analyzers without restricting the set of valid programs: every term in Λ has a semantically equivalent normal form in the restricted subset. For example, the code fragment $(f\ (\mathsf{let}\ (x\ 1)\ (g\ x)))$ becomes:

$$(\mathsf{let}\ (x_1\ 1)\ (\mathsf{let}\ (x_2\ (g\ x_1))\ (\mathsf{let}\ (x_3\ (f\ x_2))\ x_3))).$$

In general, the normalization process uses the reductions that we identified in previous work as the A-reductions [7, 14].[2]

The semantics of the restricted subset of Λ is specified by the two predicates $\mathcal{M}$ and *app* defined in Figure 1. It is straightforward to show that $\mathcal{M}$ is a partial function from terms, environments, and stores to answers. The environment is a finite table that maps the free variables to locations; the store is a finite table that maps locations to values.[3] An answer is a pair that consists of a run-time value and a store. The set of run-time values consists of numbers and closures. A *closure* is one of the procedure tags **inc** and **dec**, or a data-structure that contains the text of a user-defined procedure and the environment at the point of the creation of the closure. When applying a closure $\langle \mathbf{cl}\ x, M, \rho \rangle$ to a value u, we extend the enclosed environment at x with a *new* location and extend the store with the value u at the new location. Thus, the bound variable of a procedure or a block is related to different locations, one for each invocation of the procedure. The function *new* takes a variable x, a store

[2] The normalization process does not affect the results of the data flow analyzers. Intuitively, the first phase of A-normalization gives every subexpression a name to which the data flow analyzer can associate information about the expression. Without A-normalization, the analyzer would typically associate a "label" with every expression and attach the information about each expression at the corresponding label [8, 13, 16]. The two treatments are identical but the replacement of labels by variables simplifies the analyzers. The second phase of A-normalization re-orders the expressions to reflect the order in which the interpreters will traverse them. For example, an expression (add1 (let (x V) 0)) would be rewritten as (let (x V) (add1 0)). Again, the change is transparent to the (abstract) interpreters since they evaluate both expressions in the same manner.

[3] The formulation of the semantics does not require a store, but the presence of the store simplifies the derivation of data flow analyzers.

Domains:

$$\begin{array}{rcl} Ans & = & Val \times Sto \\ Env & = & Var \xrightarrow{\circ} Loc \\ Sto & = & Loc \xrightarrow{\circ} Val \\ Val & = & Num + Clo \\ Clo & = & (Var \times \Lambda \times Env) + \mathbf{inc} + \mathbf{dec} \end{array}$$

Auxiliary Function:

$$\begin{array}{rcl} \phi : \Lambda(V) \times Env \times Sto & \rightarrow & Val \\ \phi(n, \rho, s) & = & n \\ \phi(x, \rho, s) & = & s(\rho(x)) \\ \phi(\mathsf{add1}, \rho, s) & = & \mathbf{inc} \\ \phi(\mathsf{sub1}, \rho, s) & = & \mathbf{dec} \\ \phi((\lambda x.M), \rho, s) & = & \langle \mathbf{cl}\ x, M, \rho \rangle \end{array}$$

$\mathcal{M} : (\Lambda \times Env \times Sto) \rightarrow Ans$

$$\frac{u = \phi(V, \rho, s)}{\langle V, \rho, s \rangle\ \mathcal{M}\ \langle u, s \rangle} \qquad \frac{u = \phi(V, \rho, s) \quad \langle M, \rho[x := new(x)], s[new(x) := u] \rangle\ \mathcal{M}\ A}{\langle (\mathsf{let}\ (x\ V)\ M), \rho, s \rangle\ \mathcal{M}\ A}$$

$$\frac{u_1 = \phi(V_1, \rho, s) \quad u_2 = \phi(V_2, \rho, s) \quad \langle u_1, u_2, s \rangle\ app\ \langle u_3, s_3 \rangle \quad \langle M, \rho[x := new(x)], s_3[new(x) := u_3] \rangle\ \mathcal{M}\ A}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \rho, s \rangle\ \mathcal{M}\ A}$$

$$\frac{u_0 = \phi(V_0, \rho, s) \quad \langle M_i, \rho, s \rangle\ \mathcal{M}\ \langle u_1, s_1 \rangle \quad \langle M, \rho[x := new(x)], s_1[new(x) := u_1] \rangle\ \mathcal{M}\ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \rho, s \rangle\ \mathcal{M}\ A} \quad i = 1 \text{ if } u_0 = 0,\ i = 2 \text{ otherwise.}$$

$app : (Val \times Val \times Sto) \rightarrow Ans$

$$\frac{}{\langle \mathbf{inc}, n, s \rangle\ app\ \langle (n+1), s \rangle} \qquad \frac{}{\langle \mathbf{dec}, n, s \rangle\ app\ \langle (n-1), s \rangle} \qquad \frac{\langle M, \rho[x := new(x)], s[new(x) := u] \rangle\ \mathcal{M}\ A}{\langle \langle \mathbf{cl}\ x, M, \rho \rangle, u, s \rangle\ app\ A}$$

Figure 1: Direct (Store) Interpreter

s, and returns a new location ℓ from which it is possible to recover x, *i.e.*, $new(x, s) = \ell \notin dom(s)$ and $x = new^{-1}(\ell)$. (For brevity, we will often omit the second argument to new.)

3 Continuation-Passing Style

A continuation-passing style transformation may be applied to the interpreter or to the source program. We distinguish the two approaches by referring to the first transformation as the *semantic*-CPS transformation and to the second as the *syntactic*-CPS transformation. We discuss both possibilities in this section.

3.1 Continuations

A continuation is the control state of an evaluator. For example, during the evaluation of the procedure call $((\lambda a.N_1)\ 5)$ in $(\mathsf{let}\ (x\ ((\lambda a.N_1)\ 5))\ N_2)$, the evaluator must remember the *evaluation context* $(\mathsf{let}\ (x\ [\,])\ N_2)$ of the call as well as the environment ρ in which to evaluate N_2. Typically, this information is packaged in a frame and added to the continuation prior to the procedure call. The evaluation of the body of the procedure N_1 may itself push frames on the control stack. Thus the continuation κ can in general be represented as a list of frames where each frame consists of an evaluation context and an environment [7]:

$\kappa = \langle E_1, \rho_1 \rangle :: \cdots :: \mathbf{nil}$ where $E_i = (\mathsf{let}\ (x_i\ [\,])\ M_i)$

3.2 Semantic-CPS Transformation

The CPS interpreter (see Figure 2) maps expressions, environments, continuations, and stores to answers. It employs two auxiliary functions: $appk$, which is the CPS counterpart of app, and $appr$, which corresponds to the "return" operation of an abstract machine. The latter operation binds the return value to a variable, restores the environment, pops the control stack, and jumps to the next instruction.

The direct interpreter and the semantic-CPS interpreter produce the same output.

Lemma 3.1 *Let* $M \in \Lambda$, *then* $\langle M, \rho, s \rangle\ \mathcal{M}\ A$ *iff* $\langle M, \rho, \mathbf{nil}, s \rangle\ \mathcal{C}\ A$.

3.3 Syntactic-CPS Transformation

The transformation of a source program to a program in CPS uses two mutually recursive functions: $\mathcal{F}$ to transform terms and $\mathcal{V}$ to transform values. The function $\mathcal{F}$ takes an additional argument k, which is a variable that represents the current continuation.

Definition 3.2. (*Syntactic-CPS Transformation*) Let $cps(\Lambda)$ be the language:

$$\begin{array}{rcl} P & ::= & (k\ W) \\ & | & (\mathsf{let}\ (x\ W)\ P) \\ & | & (W\ W\ (\lambda x.P)) \\ & | & (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W\ P\ P)) \\ W & ::= & n \mid x \mid \mathsf{add1k} \mid \mathsf{sub1k} \mid (\lambda x k.P) \end{array}$$

Domains:

$$\begin{array}{rcl@{\qquad}rcl} Ans & = & Val \times Sto & Con & = & (\Lambda(E) \times Env) :: Con + \mathbf{nil} \\ Env & = & Var \to\!\!\!\!\circ\; Loc & Val & = & Num + Clo \\ Sto & = & Loc \to\!\!\!\!\circ\; Val & Clo & = & (Var \times \Lambda \times Env) + \mathbf{inc} + \mathbf{dec} \end{array}$$

$\mathcal{C} : (\Lambda \times Env \times Con \times Sto) \to Ans$

$$\frac{u = \phi(V, \rho, s) \quad \langle \kappa, \langle u, s\rangle\rangle \ appr\ A}{\langle V, \rho, \kappa, s\rangle \ \mathcal{C}\ A} \qquad \frac{u = \phi(V, \rho, s) \quad \langle M, \rho[x := new(x)], \kappa, s[new(x) := u]\rangle \ \mathcal{C}\ A}{\langle (\mathsf{let}\ (x\ V)\ M), \rho, \kappa, s\rangle \ \mathcal{C}\ A}$$

$$\frac{u_1 = \phi(V_1, \rho, s) \quad u_2 = \phi(V_2, \rho, s) \quad \langle u_1, u_2, \langle (\mathsf{let}\ (x\ [\,])\ M), \rho\rangle :: \kappa, s\rangle \ appk\ A}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \rho, \kappa, s\rangle \ \mathcal{C}\ A}$$

$$\frac{u_0 = \phi(V_0, \rho, s) \quad \langle M_i, \rho, \langle (\mathsf{let}\ (x\ [\,])\ M), \rho\rangle :: \kappa, s\rangle \ \mathcal{C}\ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \rho, \kappa, s\rangle \ \mathcal{C}\ A} \quad i = 1 \text{ if } u_0 = 0,\ i = 2 \text{ otherwise.}$$

$appk : (Val \times Val \times Con \times Sto) \to Ans$

$$\frac{\langle \kappa, \langle (n+1), s\rangle\rangle \ appr\ A}{\langle \mathbf{inc}, n, \kappa, s\rangle \ appk\ A} \qquad \frac{\langle \kappa, \langle (n-1), s\rangle\rangle \ appr\ A}{\langle \mathbf{dec}, n, \kappa, s\rangle \ appk\ A} \qquad \frac{\langle M, \rho[x := new(x)], \kappa, s[new(x) := u]\rangle \ \mathcal{C}\ A}{\langle \langle \mathbf{cl}\ x, M, \rho\rangle, u, \kappa, s\rangle \ appk\ A}$$

$appr : (Con \times Ans) \to Ans$

$$\frac{\langle M, \rho[x := new(x)], \kappa, s[new(x) := u]\rangle \ \mathcal{C}\ A}{\langle \langle (\mathsf{let}\ (x\ [\,])\ M), \rho\rangle :: \kappa, \langle u, s\rangle\rangle \ appr\ A} \qquad \frac{}{\langle \mathbf{nil}, A\rangle \ appr\ A}$$

Figure 2: Semantic-CPS Interpreter

where $x \in Vars$, $k \in KVars$ and $KVars \cap Vars = \emptyset$. The CPS transformation uses the functions $\mathcal{F}$ and $\mathcal{V}$:

$\mathcal{F} : \Lambda \to cps(\Lambda)$
$\mathcal{F}_k[\![V]\!] = (k\ \mathcal{V}[\![V]\!])$
$\mathcal{F}_k[\![(\mathsf{let}\ (x\ V)\ M)]\!] = (\mathsf{let}\ (x\ \mathcal{V}[\![V]\!])\ \mathcal{F}_k[\![M]\!])$
$\mathcal{F}_k[\![(\mathsf{let}\ (x\ (V_1\ V_2))\ M)]\!] = (\mathcal{V}[\![V_1]\!]\ \mathcal{V}[\![V_2]\!]\ \lambda x.\mathcal{F}_k[\![M]\!])$
$\mathcal{F}_k[\![(\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M)]\!] =$
$\quad (\mathsf{let}\ (k'\ \lambda x.\mathcal{F}_k[\![M]\!])\ (\mathsf{if0}\ \mathcal{V}[\![V_0]\!]\ \mathcal{F}_{k'}[\![M_1]\!]\ \mathcal{F}_{k'}[\![M_2]\!]))$

$\mathcal{V} : \Lambda(V) \to cps(\Lambda)(W)$
$\mathcal{V}[\![n]\!] = n$
$\mathcal{V}[\![x]\!] = x$
$\mathcal{V}[\![\mathsf{add1}]\!] = \mathsf{add1k}$
$\mathcal{V}[\![\mathsf{sub1}]\!] = \mathsf{sub1k}$
$\mathcal{V}[\![(\lambda x.M)]\!] = (\lambda x k.\mathcal{F}_k[\![M]\!])$

■

The evaluation for CPS programs is defined by $\mathcal{M}_c$, a specialized version of the direct interpreter $\mathcal{M}$ [7]. It handles procedures of two arguments and manipulates a larger set of run-time values than the direct interpreter that includes continuations of the form $\langle \mathbf{co}\ x, P, \rho\rangle$ (see Figure 3). The larger set of run-time values reflects the salient aspect of the CPS transformation: it reifies the continuation of the evaluator to an object that the program explicitly manipulates.[4]

[4] In principle, we could use the direct interpreter $\mathcal{M}$ to evaluate CPS programs. However, this choice forces continuations to be represented as procedures, which is (unrealistic and) unnecessarily confusing for data flow analyzers.

To establish the formal relationship between the behavior of a direct term and the behavior of its CPS-transform, we define the function δ that relates direct run-time values to their CPS counterparts:

$$\begin{array}{rcl} \delta(n) & = & n \\ \delta(\mathbf{inc}) & = & \mathbf{inck} \\ \delta(\mathbf{dec}) & = & \mathbf{deck} \\ \delta(\langle \mathbf{cl}\ x, M, \rho\rangle) & = & \langle \mathbf{cl}\ xk, \mathcal{F}_k[\![M]\!], \rho\rangle \end{array}$$

We extend δ to work on stores by applying it to the value at each location and to answers by applying it to both the value component and the store component.

The following lemma describes the precise relationship between the semantic-CPS interpreter and the syntactic-CPS interpreter. The interpreters yield answers related by δ; the store resulting from the syntactic-CPS interpreter will contain additional entries that correspond to continuations.

Lemma 3.3 *Let $M \in \Lambda$, then:*

$\langle M, \rho, \mathbf{nil}, s\rangle \ \mathcal{C}\ \langle u_1, s_1\rangle$ *iff*
$\quad \langle \mathcal{F}_k[\![M]\!], \rho[k := new(k)], \delta(s)[new(k) := \mathbf{stop}]\rangle$
$\mathcal{M}_c\ \langle \delta(u_1), \delta(s_1)[new(k_1) := \kappa_1, new(k_2) := \kappa_2, \ldots]\rangle.$

Together with Lemma 3.1, this result also relates the syntactic-CPS interpreter to the direct one.

Domains:

$$\begin{array}{rcl} Ans & = & Val \times Sto \\ Env & = & Var \rightarrow\!\!\!\!\circ \; Loc \\ Sto & = & Loc \rightarrow\!\!\!\!\circ \; Val \\ Val & = & Num + Clo + Con \\ Clo & = & (Var \times KVar \times cps(\Lambda) \times Env) \\ & & + \mathbf{inck} + \mathbf{deck} \\ Con & = & (Var \times cps(\Lambda) \times Env) + \mathbf{stop} \end{array}$$

Auxiliary Function:

$$\begin{array}{rcl} \phi_c : cps(\Lambda)(W) \times Env \times Sto & \rightarrow & Val \\ \phi_c(n, \rho, s) & = & n \\ \phi_c(x, \rho, s) & = & s(\rho(x)) \\ \phi_c(\mathsf{add1k}, \rho, s) & = & \mathbf{inck} \\ \phi_c(\mathsf{sub1k}, \rho, s) & = & \mathbf{deck} \\ \phi_c((\lambda x k.P), \rho, s) & = & \langle \mathbf{cl}\ x k, P, \rho \rangle \end{array}$$

$\mathcal{M}_c : (cps(\Lambda) \times Env \times Sto) \rightarrow Ans$

$$\frac{\kappa = s(\rho(k)) \quad u = \phi_c(W, \rho, s) \quad \langle \kappa, \langle u, s \rangle \rangle \ appr_c\ A}{\langle (k\ W), \rho, s \rangle \ \mathcal{M}_c\ A} \qquad \frac{u = \phi_c(W, \rho, s) \quad \langle P, \rho[x := new(x)], s[new(x) := u] \rangle \ \mathcal{M}_c\ A}{\langle (\mathsf{let}\ (x\ W)\ P), \rho, s \rangle \ \mathcal{M}_c\ A}$$

$$\frac{u_1 = \phi_c(W_1, \rho, s) \quad u_2 = \phi_c(W_2, \rho, s) \quad \langle u_1, u_2, \langle \mathbf{co}\ x, P, \rho \rangle, s \rangle \ app_c\ A}{\langle (W_1\ W_2\ (\lambda x.P)), \rho, s \rangle \ \mathcal{M}_c\ A}$$

$$\frac{u_0 = \phi_c(W_0, \rho, s) \quad \langle P_i, \rho[k := new(k)], s[new(k) := \langle \mathbf{co}\ x, P, \rho \rangle] \rangle \ \mathcal{M}_c\ A}{\langle (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W_0\ P_1\ P_2)), \rho, s \rangle \ \mathcal{M}_c\ A} \quad i = 1 \text{ if } u_0 = 0,\ i = 2 \text{ otherwise.}$$

$app_c : (Val \times Val \times Val \times Sto) \rightarrow Ans$

$$\frac{\langle \kappa, \langle (n+1), s \rangle \rangle \ appr_c\ A}{\langle \mathbf{inck}, n, \kappa, s \rangle \ app_c\ A} \qquad \frac{\langle \kappa, \langle (n-1), s \rangle \rangle \ appr_c\ A}{\langle \mathbf{deck}, n, \kappa, s \rangle \ app_c\ A}$$

$$\frac{\langle P, \rho[x := new(x), k := new(k)], s[new(x) := u, new(k) := \kappa] \rangle \ \mathcal{M}_c\ A}{\langle \langle \mathbf{cl}\ x k, P, \rho \rangle, u, \kappa, s \rangle \ app_c\ A}$$

$appr_c : (Val \times Ans) \rightarrow Ans$

$$\frac{\langle P, \rho[x := new(x)], s[new(x) := u] \rangle \ \mathcal{M}_c\ A}{\langle \langle \mathbf{co}\ x, P, \rho \rangle, \langle u, s \rangle \rangle \ appr_c\ A} \qquad \frac{}{\langle \mathbf{stop}, A \rangle \ appr_c\ A}$$

Figure 3: Syntactic-CPS Interpreter

4 Constant Propagation by Abstract Interpretation

Using well-known ideas from the area of abstract interpretation [4, 8, 13, 16], we now derive a data flow analyzer from each of the three interpreters. The first step in the derivation is to associate *one* location with each variable that holds the potentially infinite set of values to which the variable is bound during the evaluation of the program. Second, we approximate these sets of values so that each label is associated with a finite number of values. Finally, we modify the interpreters to detect and recover from all loops when computing over the universe of approximate values.

4.1 Abstracting Procedures

The first step in the derivation of the data flow analyzers is to limit the number of locations that can be created during the evaluation of a given program. One of the simple approximations, known as 0CFA analysis [16], is to associate *one* location for each variable and to collect *all* the values to which the variable is bound at that location. Formally, we approximate environments ρ to $\overline{\rho}$ and stores s to $\overline{s}$ as follows:

- Since each variable is associated with exactly one location, we can choose that location to be the variable itself. Thus, if $\rho = \{x_1 \mapsto new(x_1), \ldots\}$, $\overline{\rho} = \{x_1 \mapsto x_1, \ldots\}$. This approximation of the environment does not provide any information, that is, we can drop it completely. Thus, a closure $\langle \mathbf{cl}\ x, M, \rho \rangle$ becomes an abstract closure $\langle \mathbf{cl}^{\ominus}\ x, M \rangle$, a continuation $\langle E_1, \rho_1 \rangle :: \ldots :: \mathbf{nil}$ becomes $E_1 :: \ldots :: \mathbf{nil}$, and a continuation $\langle \mathbf{co}\ x, M, \rho \rangle$ becomes an abstract continuation $\langle \mathbf{co}^{\ominus}\ x, M \rangle$. It follows that, for each source program, the sets of abstract closures and abstract continuations are finite.

- For stores $s = \{new(x_1) \mapsto u_1, \ldots\}$, we first recover the variable associated with each of the locations: $\{x_1 \mapsto u_1, \ldots\}$. Then, to obtain the store

$\overline{s}$, we merge all entries of the form $x \mapsto u_1, x \mapsto u_2, \ldots$ for some x into one entry $x \mapsto \{u_1, u_2, \ldots\}$.

A "collecting semantics" like the above associates a set with each variable. Intuitively, the larger the set the less information is available at compile time about the variable. To formalize this notion of "precision," we note that the sets of collected values form a complete lattice ordered by set-inclusion; the least upper bound operation is set-union. Thus, the relation "is more precise than" coincides with the lattice ordering.

4.2 Abstracting Integers

Despite the approximations of environments, stores, closures, and continuations, the "collecting semantics" still associates an unbounded set of values with a variable because the lattice of collected values contains infinite chains of elements of decreasing precision, *e.g.*, $\emptyset \subseteq \{0\} \subseteq \{0,1\} \subseteq \{0,1,2\} \ldots$. Since these infinite chains may cause the analysis to diverge, we approximate sets of numbers to abstract numbers [9]:

$$\overline{\emptyset} = \bot, \quad \overline{\{n\}} = n, \quad \text{and} \quad \overline{\{n_1, n_2, \ldots\}} = \top.$$

At this point, the universe of abstract values consists of abstract numbers and abstract closures. It remains to impose an order $\sqsubseteq$ on the abstract values similar to the order $\subseteq$ on collected values that coincides with the relation "is more precise than." For the direct and semantic-CPS interpreters, we organize the abstract values in a lattice that is the product of two lattices: the first is the traditional lattice $\mathbb{N}_\bot^\top$ for constant propagation [9], and the second is the power set of abstract closures (ordered by the subset relation) for control flow analysis [16]. The ordering relation $\sqsubseteq$ and the least upper bound operation $\sqcup$ are defined component-wise. It is easy to check that, if S_1 and S_2 are sets of collected values, then $S_1 \subseteq S_2$ implies $\overline{S_1} \sqsubseteq \overline{S_2}$.

For the syntactic-CPS collecting interpreter, the sets of values include abstract continuations as well. In that case, we use a lattice of abstract values that consists of the product of three lattices: the constant propagation lattice, the power set of abstract closures, and the power set of abstract continuations.

The ordering of abstract values induces an ordering on stores. If σ_1 and σ_2 are abstract stores, then $\sigma_1 \sqsubseteq \sigma_2$ if for every variable x in the domain of σ_1, $\sigma_1(x) \sqsubseteq \sigma_2(x)$. The latter ordering induces a component-wise ordering on abstract answers.

We can now specify collecting interpreters that manipulate abstract values. The interpreters replace *add1* and *sub1* by $add1^\ominus$ and $sub1^\ominus$:

$$\begin{array}{lcllcl} add1^\ominus(\bot) & = & \bot & sub1^\ominus(\bot) & = & \bot \\ add1^\ominus(n) & = & (n+1) & sub1^\ominus(n) & = & (n-1) \\ add1^\ominus(\top) & = & \top & sub1^\ominus(\top) & = & \top \end{array}$$

Figures 4, 5, and 6 contain the direct abstract collecting interpreters, the semantic-CPS abstract collecting interpreter, and the syntactic-CPS abstract collecting interpreter respectively.

4.3 Correctness

The correctness criterion of an abstract collecting interpreter is that its results approximate the actual execution of the program. For example, if the variable x gets bound to 5 along any actual execution path, the abstract collecting interpreter should associate an abstract value $u \sqsupseteq \langle 5, \bot \rangle$ with the variable x.

Lemma 4.1 *If $\overline{s} \sqsubseteq \sigma$, $\overline{\{u_1\}} \sqsubseteq u_2$ and $\overline{s_1} \sqsubseteq \sigma_2$, then:*

1. *If $\langle M, \rho, s\rangle \ \mathcal{M}\ \langle u_1, s_1\rangle$, then $\langle M, \sigma\rangle \ \mathcal{M}^\ominus\ \langle u_2, \sigma_2\rangle$.*
2. *If $\langle M, \rho, \kappa, s\rangle \ \mathcal{C}\ \langle u_1, s_1\rangle$, then $\langle M, \overline{\kappa}, \sigma\rangle \ \mathcal{C}^\ominus\ \langle u_2, \sigma_2\rangle$.*
3. *If $\langle P, \rho, s\rangle \ \mathcal{M}_c\ \langle u_1, s_1\rangle$, then $\langle P, \sigma\rangle \ \mathcal{M}_c^\ominus\ \langle u_2, \sigma_2\rangle$.*

4.4 Termination

Interpreted naïvely, the specifications for the abstract collecting interpreters define partial functions that diverge on some inputs, and hence are not data flow algorithms. However, this is not a problem, since it is possible to detect all loops in the derivations. More precisely, assume we have the following fragment of a derivation tree:

$$\begin{array}{l} \ldots \\ \langle M_j, \sigma_j\rangle \ \mathcal{M}^\ominus\ ? \\ \ldots \\ \langle M_i, \sigma_i\rangle \ \mathcal{M}^\ominus\ ? \\ \ldots \\ \langle M_1, \sigma_1\rangle \ \mathcal{M}^\ominus\ ? \end{array}$$

The evaluation of M_1 in store σ_1 requires the value of M_i in store σ_i, which in turn requires the value of M_j in store σ_j, and so on. If the above fragment of the derivation is indeed infinite, then one of the M's must be repeated infinitely often as the abstract syntax tree of the program has only a finite number of subtrees. Thus, without loss of generality, let $M_1 = M_i = M_j = M$. By inspection of the direct abstract collecting interpreter, $\sigma_1 \ldots \sqsubseteq \sigma_i \ldots \sqsubseteq \sigma_j \ldots$. As the lattice of abstract stores does not have any infinite ascending chains, one of the σ's in the sequence must be repeated: $\sigma_i = \sigma_j = \sigma$. Thus, all loops will result in two identical proof goals.

Having detected a loop, we return the least precise value paired with the current store. Thus, if the arguments $\langle M, \sigma\rangle$ have already been considered, the direct interpreter returns the answer $\langle\langle \top, CL^\top\rangle, \sigma\rangle$ where $CL^\top$ is the set of all abstract closures in the program. Similarly, the semantic-CPS interpreter returns

Domains:

$$\begin{array}{rcl} \overline{Ans} & = & \overline{Val} \times \overline{Sto} \\ \overline{Sto} & = & \overline{Var} \stackrel{\circ}{\rightarrow} \overline{Val} \\ \overline{Val} & = & \overline{Num} \times \mathcal{P}(\overline{Clo}) \\ \overline{Clo} & = & (Var \times \Lambda) + \mathbf{inc} + \mathbf{dec} \end{array}$$

Auxiliary Functions:

$$\begin{array}{rcl} \phi^{\ominus} : \Lambda(V) \times \overline{Sto} & \rightarrow & \overline{Val} \\ \phi^{\ominus}(n, \sigma) & = & \langle n, \emptyset \rangle \\ \phi^{\ominus}(x, \sigma) & = & \sigma(x) \\ \phi^{\ominus}(\mathsf{add1}, \sigma) & = & \langle \bot, \{\mathbf{inc}\} \rangle \\ \phi^{\ominus}(\mathsf{sub1}, \sigma) & = & \langle \bot, \{\mathbf{dec}\} \rangle \\ \phi^{\ominus}((\lambda x.M), \sigma) & = & \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x, M \rangle\} \rangle \end{array}$$

$\mathcal{M}^{\ominus} : (\Lambda \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{u = \phi^{\ominus}(V, \sigma)}{\langle V, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u, \sigma \rangle} \qquad \frac{u = \phi^{\ominus}(V, \sigma) \qquad \langle M, \sigma[x := \sigma(x) \sqcup u] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ V)\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A}$$

$$\frac{u_1 = \phi^{\ominus}(V_1, \sigma) \quad u_2 = \phi^{\ominus}(V_2, \sigma) \quad \langle u_1, u_2, \sigma \rangle\ app^{\ominus}\ \langle u_3, \sigma_3 \rangle \quad \langle M, \sigma_3[x := \sigma_3(x) \sqcup u_3] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A}$$

$$\frac{u_0 = \phi^{\ominus}(V_0, \sigma) \quad \langle M_i, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_1, \sigma_1 \rangle \quad \langle M, \sigma_1[x := \sigma_1(x) \sqcup u_1] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A} \quad i = 1 \text{ if } u_0 = \langle 0, \emptyset \rangle,\ i = 2 \text{ if } \langle 0, \emptyset \rangle \not\sqsubseteq u_0.$$

$$\frac{\langle 0, \emptyset \rangle \sqsubset u_0 = \phi^{\ominus}(V_0, \sigma) \quad \langle M_1, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_1, \sigma_1 \rangle \quad \langle M_2, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_2, \sigma_2 \rangle \quad \langle M, (\sigma_1 \sqcup \sigma_2)[x := (\sigma_1 \sqcup \sigma_2)(x) \sqcup (u_1 \sqcup u_2)] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A}$$

$app^{\ominus} : (\overline{Val} \times \overline{Val} \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{\langle cl_1, u, \sigma \rangle\ {app_1}^{\ominus}\ A_1 \quad \ldots \quad \langle cl_n, u, \sigma \rangle\ {app_1}^{\ominus}\ A_n}{\langle \langle n, \{cl_1, \ldots, cl_n\} \rangle, u, \sigma \rangle\ app^{\ominus}\ \bigsqcup_{i=1,n} A_i}$$

${app_1}^{\ominus} : (\overline{Clo} \times \overline{Val} \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{u = \langle {add1}^{\ominus}(n), \emptyset \rangle}{\langle \mathbf{inc}, \langle n, CL \rangle, \sigma \rangle\ {app_1}^{\ominus}\ \langle u, \sigma \rangle} \qquad \frac{u = \langle {sub1}^{\ominus}(n), \emptyset \rangle}{\langle \mathbf{dec}, \langle n, CL \rangle, \sigma \rangle\ {app_1}^{\ominus}\ \langle u, \sigma \rangle} \qquad \frac{\langle M, \sigma[x := \sigma(x) \sqcup u] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle \langle \mathbf{cl}^{\ominus}\ x, M \rangle, u, \sigma \rangle\ {app_1}^{\ominus}\ A}$$

Figure 4: Direct Abstract Collecting Interpreter

$\langle \langle \top, CL^{\top} \rangle, \sigma \rangle$ to the continuation κ. If the syntactic-CPS interpreter detects that the arguments $\langle P, \sigma \rangle$ have already been considered, it returns $\langle \langle \top, CL^{\top}, K^{\top} \rangle, \sigma \rangle$ where $K^{\top}$ is the set of all abstract continuations $\langle \mathbf{co}^{\ominus}\ x, P \rangle$ in the program.

In the remainder of the paper, we will use "abstract collecting interpreter" or "data flow analysis" to refer to the terminating versions of the interpreters that detect loops as above.

5 Formal Relationships

After deriving the data flow analyzers, we turn our attention to the relationship between them. For the connection between the direct and syntactic-CPS analyses, we need an abstract version of the function δ that maps abstract direct values to abstract CPS values:

$$\begin{array}{l} \delta^{\ominus}(\langle n, \{cl_1, \ldots, cl_i\} \rangle) = \langle n, \{\mathcal{V}^{\ominus}(cl_1), \ldots, \mathcal{V}^{\ominus}(cl_i)\}, \emptyset \rangle \\ \mathcal{V}^{\ominus}(\langle \mathbf{cl}^{\ominus}\ x_1, M_1 \rangle) = \langle \mathbf{cl}^{\ominus}\ x_1 k_1, \mathcal{F}_{k_1}[\![M_1]\!] \rangle \\ \mathcal{V}^{\ominus}(\mathbf{inc}) = \mathbf{inck} \\ \mathcal{V}^{\ominus}(\mathbf{dec}) = \mathbf{deck} \end{array}$$

The application of $\delta^{\ominus}$ to stores and answers is pointwise and component-wise respectively.

5.1 Direct vs Syntactic-CPS

The first theorem establishes that the direct analysis of M may be *more* precise than the analysis of $\mathcal{F}_k[\![M]\!]$.

Theorem 5.1 *There exists $M \in \Lambda$ and $\sigma \in Sto$ such that:*

- $\langle M, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_1, \sigma_1 \rangle$,
- $\langle \mathcal{F}_k[\![M]\!], \delta^{\ominus}(\sigma)[k := \langle \bot, \emptyset, \{\mathbf{stop}\} \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ \langle u_2, \sigma_2 \rangle$,

Domains:

$$\begin{array}{rcl} \overline{Ans} & = & \overline{Val} \times \overline{Sto} \\ \overline{Sto} & = & Var \rightarrow\!\!\!\!\circ\; \overline{Val} \end{array} \qquad \begin{array}{rcl} \overline{Con} & = & \Lambda(E) :: \overline{Con} + \mathbf{nil} \\ \overline{Val} & = & \overline{Num} \times \mathcal{P}(\overline{Clo}) \\ \overline{Clo} & = & (Var \times \Lambda) + \mathbf{inc} + \mathbf{dec} \end{array}$$

$\mathcal{C}^{\ominus} : (\Lambda \times \overline{Con} \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{u = \phi^{\ominus}(V,\sigma) \quad \langle \kappa, \langle u, \sigma\rangle\rangle \ appr^{\ominus}\ A}{\langle V, \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A} \qquad \frac{u = \phi^{\ominus}(V,\sigma) \quad \langle M, \kappa, \sigma[x := \sigma(x) \sqcup u]\rangle \ \mathcal{C}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ V)\ M), \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u_1 = \phi^{\ominus}(V_1,\sigma) \quad u_2 = \phi^{\ominus}(V_2,\sigma) \quad \langle u_1, u_2, (\mathsf{let}\ (x\ [\,])\ M) :: \kappa, \sigma\rangle \ appk^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u_0 = \phi^{\ominus}(V_0,\sigma) \quad \langle M_i, (\mathsf{let}\ (x\ [\,])\ M) :: \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A} \quad i = 1 \text{ if } u_0 = \langle 0, \emptyset\rangle,\ i = 2 \text{ if } \langle 0, \emptyset\rangle \not\sqsubseteq u_0.$$

$$\frac{\langle 0, \emptyset\rangle \sqsubset u_0 = \phi^{\ominus}(V_0,\sigma) \quad \langle M_1, (\mathsf{let}\ (x\ [\,])\ M) :: \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A_1 \quad \langle M_2, (\mathsf{let}\ (x\ [\,])\ M) :: \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A_2}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ A_1 \sqcup A_2}$$

$appk^{\ominus} : (\overline{Val} \times \overline{Val} \times \overline{Con} \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{\langle cl_1, u, \kappa, \sigma\rangle \ appk_1{}^{\ominus}\ A_1 \quad \ldots \quad \langle cl_n, u, \kappa, \sigma\rangle \ appk_1{}^{\ominus}\ A_n}{\langle \langle n, \{cl_1, \ldots, cl_n\}\rangle, u, \kappa, \sigma\rangle \ appk^{\ominus} \bigsqcup_{i=1,n} A_i}$$

$appk_1{}^{\ominus} : (\overline{Clo} \times \overline{Val} \times \overline{Con} \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{u = \langle add1^{\ominus}(n), \emptyset\rangle \quad \langle \kappa, \langle u, \sigma\rangle\rangle \ appr^{\ominus}\ A}{\langle \mathbf{inc}, \langle n, CL\rangle, \kappa, \sigma\rangle \ appk_1{}^{\ominus}\ A} \qquad \frac{u = \langle sub1^{\ominus}(n), \emptyset\rangle \quad \langle \kappa, \langle u, \sigma\rangle\rangle \ appr^{\ominus}\ A}{\langle \mathbf{dec}, \langle n, CL\rangle, \kappa, \sigma\rangle \ appk_1{}^{\ominus}\ A} \qquad \frac{\langle M, \kappa, \sigma[x := \sigma(x) \sqcup u]\rangle \ \mathcal{C}^{\ominus}\ A}{\langle \langle \mathbf{cl}^{\ominus}\ x, M\rangle, u, \kappa, \sigma\rangle \ appk_1{}^{\ominus}\ A}$$

$appr^{\ominus} : (\overline{Con} \times \overline{Ans}) \rightarrow \overline{Ans}$

$$\frac{\langle M, \kappa, \sigma[x := \sigma(x) \sqcup u]\rangle \ \mathcal{C}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ [\,])\ M) :: \kappa, \langle u, \sigma\rangle\rangle \ appr^{\ominus}\ A} \qquad \frac{}{\langle \mathbf{nil}, A\rangle \ appr^{\ominus}\ A}$$

Figure 5: Semantic-CPS Abstract Collecting Interpreter

- *$\delta^{\ominus}(u_1) \sqsubseteq u_2$, and for each variable in the domain of σ_1, $\delta^{\ominus}(\sigma_1(x)) \sqsubseteq \sigma_2(x)$.*

Proof. Let M be $(\mathsf{let}\ (a_1\ (f\ 1))\ (\mathsf{let}\ (a_2\ (f\ 2))\ a_2))$, and let:

$$\begin{array}{rcl} \sigma & = & \{a_1 \mapsto \langle \bot, \emptyset\rangle, \\ & & \ a_2 \mapsto \langle \bot, \emptyset\rangle, \\ & & \ f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x, x\rangle\}\rangle, \\ & & \ x \mapsto \langle \bot, \emptyset\rangle\}. \end{array}$$

It is straightforward to calculate that the result of the direct abstract collecting interpreter is $A_1 = \langle u_1, \sigma_1\rangle$ where:

$$\begin{array}{rcl} u_1 & = & \langle \top, \emptyset\rangle \\ \sigma_1 & = & \{a_1 \mapsto \langle 1, \emptyset\rangle, \\ & & \ a_2 \mapsto \langle \top, \emptyset\rangle, \\ & & \ f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x, x\rangle\}\rangle, \\ & & \ x \mapsto \langle \top, \emptyset\rangle\}. \end{array}$$

For the analysis of the CPS version, we have that:

$$\begin{array}{rcl} \mathcal{F}_k[\![M]\!] & = & (f\ 1\ (\lambda a_1.(f\ 2\ (\lambda a_2.(k\ a_2)))))) \\ \sigma' & = & \{a_1 \mapsto \langle \bot, \emptyset, \emptyset\rangle, \\ & & \ a_2 \mapsto \langle \bot, \emptyset, \emptyset\rangle, \\ & & \ f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ xk_1, (k_1\ x)\rangle\}, \emptyset\rangle, \\ & & \ x \mapsto \langle \bot, \emptyset, \emptyset\rangle, \\ & & \ k \mapsto \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle\}. \end{array}$$

The syntactic-CPS abstract collecting interpreter produces the answer $A_2 = \langle u_2, \sigma_2\rangle$ where:

$$\begin{array}{rcl} u_2 & = & \langle \top, CL^{\top}, K^{\top}\rangle \\ \sigma_2 & = & \{a_1 \mapsto \langle \top, \emptyset, \emptyset\rangle, \\ & & \ a_2 \mapsto \langle \top, \emptyset, \emptyset\rangle, \end{array}$$

Domains:

$$\begin{array}{rcl}
\overline{Ans} & = & \overline{Val} \times \overline{Sto} \\
\overline{Sto} & = & Var \twoheadrightarrow \overline{Val} \\
\overline{Val} & = & \overline{Num} \times \mathcal{P}(\overline{Clo}) \times \mathcal{P}(\overline{Con}) \\
\overline{Clo} & = & (Var \times KVar \times cps(\Lambda)) \\
 & & + \mathbf{inck} + \mathbf{deck} \\
\overline{Con} & = & (Var \times cps(\Lambda)) + \mathbf{stop}
\end{array}$$

Auxiliary Function:

$$\begin{array}{rcl}
\phi_c^{\ominus} : cps(\Lambda)(W) \times \overline{Sto} & \rightarrow & \overline{Val} \\
\phi_c^{\ominus}(n, \sigma) & = & \langle n, \emptyset, \emptyset \rangle \\
\phi_c^{\ominus}(x, \sigma) & = & \sigma(x) \\
\phi_c^{\ominus}(\mathsf{add1k}, \sigma) & = & \langle \bot, \{\mathbf{inck}\}, \emptyset \rangle \\
\phi_c^{\ominus}(\mathsf{sub1k}, \sigma) & = & \langle \bot, \{\mathbf{deck}\}, \emptyset \rangle \\
\phi_c^{\ominus}((\lambda x k.P), \sigma) & = & \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x k, P \rangle\}, \emptyset \rangle
\end{array}$$

$\mathcal{M}_c^{\ominus} : (cps(\Lambda) \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{\kappa = \sigma(k) \quad u = \phi_c^{\ominus}(W, \sigma) \quad \langle \kappa, \langle u, \sigma \rangle \rangle\ appr_c^{\ominus}\ A}{\langle (k\ W), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A} \qquad \frac{u = \phi_c^{\ominus}(W, \sigma) \quad \langle P, \sigma[x := \sigma(x) \sqcup u] \rangle\ \mathcal{M}_c^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ W)\ P), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A}$$

$$\frac{u_1 = \phi_c^{\ominus}(W_1, \sigma) \quad u_2 = \phi_c^{\ominus}(W_2, \sigma) \quad \langle u_1, u_2, \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle, \sigma \rangle\ app_c^{\ominus}\ A}{\langle (W_1\ W_2\ (\lambda x.P)), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A}$$

$$\frac{u_0 = \phi_c^{\ominus}(W_0, \sigma) \quad \langle P_i, \sigma[k := \sigma(k) \sqcup \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ A}{\langle (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W_0\ P_1\ P_2)), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A} \quad i = 1 \text{ if } u_0 = \langle 0, \emptyset, \emptyset \rangle,\ i = 2 \text{ if } \langle 0, \emptyset, \emptyset \rangle \not\sqsubseteq u_0.$$

$$\frac{\begin{array}{c}\langle 0, \emptyset, \emptyset \rangle \sqsubset u_0 = \phi_c^{\ominus}(W_0, \sigma) \\ \langle P_1, \sigma[k := \sigma(k) \sqcup \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ A_1 \qquad \langle P_2, \sigma[k := \sigma(k) \sqcup \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ A_2\end{array}}{\langle (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W_0\ P_1\ P_2)), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A_1 \sqcup A_2}$$

$app_c^{\ominus} : (\overline{Val} \times \overline{Val} \times \overline{Val} \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{\langle cl_1, u, \kappa, \sigma \rangle\ app_{1c}^{\ominus}\ A_1 \quad \ldots \quad \langle cl_n, u, \kappa, \sigma \rangle\ app_{1c}^{\ominus}\ A_n}{\langle \langle n, \{cl_1, \ldots, cl_n\}, K \rangle, u, \kappa, \sigma \rangle\ app_c^{\ominus}\ \bigsqcup_{i=1,n} A_i}$$

$app_{1c}^{\ominus} : (\overline{Clo} \times \overline{Val} \times \overline{Val} \times \overline{Sto}) \rightarrow \overline{Ans}$

$$\frac{u = \langle add1^{\ominus}(n), \emptyset, \emptyset \rangle \quad \langle \kappa, \langle u, \sigma \rangle \rangle\ appr_c^{\ominus}\ A}{\langle \mathbf{inck}, \langle n, CL, K \rangle, \kappa, \sigma \rangle\ app_{1c}^{\ominus}\ A} \qquad \frac{u = \langle sub1^{\ominus}(n), \emptyset, \emptyset \rangle \quad \langle \kappa, \langle u, \sigma \rangle \rangle\ appr_c^{\ominus}\ A}{\langle \mathbf{deck}, \langle n, CL, K \rangle, \kappa, \sigma \rangle\ app_{1c}^{\ominus}\ A}$$

$$\frac{\langle P, \sigma[x := \sigma(x) \sqcup u, k := \sigma(k) \sqcup \kappa] \rangle\ \mathcal{M}_c^{\ominus}\ A}{\langle \langle \mathbf{cl}^{\ominus}\ x k, P \rangle, u, \kappa, \sigma \rangle\ app_{1c}^{\ominus}\ A}$$

$appr_c^{\ominus} : (\overline{Val} \times \overline{Ans}) \rightarrow \overline{Ans}$

$$\frac{\langle \kappa_1, \langle u, \sigma \rangle \rangle\ appr_{1c}^{\ominus}\ A_1 \quad \ldots \quad \langle \kappa_n, \langle u, \sigma \rangle \rangle\ appr_{1c}^{\ominus}\ A_n}{\langle \langle n, CL, \{\kappa_1, \ldots, \kappa_n\} \rangle, \langle u, \sigma \rangle \rangle\ appr_c^{\ominus}\ \bigsqcup_{i=1,n} A_i}$$

$appr_{1c}^{\ominus} : (\overline{Con} \times \overline{Ans}) \rightarrow Ans$

$$\frac{\langle P, \sigma[x := \sigma(x) \sqcup u] \rangle\ \mathcal{M}_c^{\ominus}\ A}{\langle \langle \mathbf{co}^{\ominus}\ x, P \rangle, \langle u, \sigma \rangle \rangle\ appr_{1c}^{\ominus}\ A} \qquad \frac{}{\langle \mathbf{stop}, A \rangle\ appr_{1c}^{\ominus}\ A}$$

Figure 6: Syntactic-CPS Abstract Collecting Interpreter

$$\begin{array}{l}
f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x k_1, (k_1\ x) \rangle\}, \emptyset \rangle, \\
x \mapsto \langle \top, \emptyset, \emptyset \rangle, \\
k_1 \mapsto \langle \bot, \emptyset, \{\ \langle \mathbf{co}^{\ominus}\ a_1, (f\ 2\ (\lambda a_2.(k\ a_2))) \rangle, \\
\qquad\qquad \langle \mathbf{co}^{\ominus}\ a_2, (k\ a_2) \rangle\} \rangle \\
k \mapsto \langle \bot, \emptyset, \{\mathbf{stop}\} \rangle]
\end{array}$$

The analysis of the source program is more precise since it determines that the variable a_1 is constant (=1), while the analysis of the CPS program fails to produce any information about a_1. ∎

Our second theorem states that the direct analysis of a program may also give *less* information than the syntactic-CPS analysis. Together with the previous theorem, the result establishes that the direct analysis of a source program is *incomparable* to the syntactic-CPS analysis.

Theorem 5.2 *There exists a term $M \in \Lambda$ such that:*

- $\langle M, \sigma\rangle \ \mathcal{M}^{\ominus} \ \langle u_1, \sigma_1\rangle$,
- $\langle \mathcal{F}_k[\![M]\!], \delta^{\ominus}(\sigma)[k := \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle]\rangle \ \mathcal{M}_c^{\ominus} \ \langle u_2, \sigma_2\rangle$,
- $\delta^{\ominus}(u_1) \sqsupseteq u_2$, *and for each variable x in the domain of σ_1, $\delta^{\ominus}(\sigma_1(x)) \sqsupseteq \sigma_2(x)$.*

Proof. To illustrate the principle, we present two cases in which the analysis of the CPS program yields more information than the direct analysis of the source program.

For the first case, take:

$$\begin{aligned} M &= (\mathsf{let}\ (a_1\ (\mathsf{if0}\ x\ 0\ 1)) \\ &\qquad (\mathsf{let}\ (a_2\ (\mathsf{if0}\ a_1\ (+\ a_1\ 3)\ (+\ a_1\ 2))) \\ &\qquad\ a_2)) \\ \sigma &= \{a_1 \mapsto \langle \bot, \emptyset\rangle,\ a_2 \mapsto \langle \bot, \emptyset\rangle,\ x \mapsto \langle \top, \emptyset\rangle\} \end{aligned}$$

where $(+\ a_1\ 3)$ and $(+\ a_1\ 2)$ are the obvious abbreviations. The direct analysis of the term, not knowing which branch to take, merges the abstract values of 0 and 1 at the variable a_1 and hence loses all information about a_2. In contrast, the analysis of:

$$\begin{aligned} &\mathcal{F}_k[\![M]\!] = \\ &(\mathsf{let}\ (k'\ \lambda a_1.\mathcal{F}_k[\![(\mathsf{let}\ (a_2\ (\mathsf{if0}\ a_1\ (a_1+3)\ (a_1+2)))\ a_2)]\!]) \\ &\ (\mathsf{if0}\ x\ (k'\ 0)\ (k'\ 1))) \end{aligned}$$

analyzes both $(k'\ 0)$ and $(k'\ 1)$ in a store that maps $k' \mapsto \langle \mathbf{co}^{\ominus}\ a_1, \ldots\rangle$. The analysis of each execution path determines that the abstract value of a_2 is $\langle 3, \emptyset, \emptyset\rangle$, which improves on the direct analysis.

For the second case, take M:

$$\begin{aligned} M &= (\mathsf{let}\ (a_1\ (f\ 3)) \\ &\qquad (\mathsf{let}\ (a_2\ (\mathsf{if0}\ a_1\ 5\ (\mathsf{if0}\ (\mathsf{sub1}\ a_1)\ 5\ 6))) \\ &\qquad\ a_2)) \\ \sigma &= \{a_1 \mapsto \langle \bot, \emptyset\rangle \\ &\qquad a_2 \mapsto \langle \bot, \emptyset\rangle, \\ &\qquad f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ d_0, 0\rangle, \langle \mathbf{cl}^{\ominus}\ d_1, 1\rangle\}\rangle \\ &\qquad d_0 \mapsto \langle \bot, \emptyset\rangle, \\ &\qquad d_1 \mapsto \langle \bot, \emptyset\rangle\} \end{aligned}$$

where we have not named the results of $(\mathsf{sub1}\ a_1)$ and $(\mathsf{if0}\ (\mathsf{sub1}\ a_1)\ 5\ 6)$ to avoid clutter. The direct analysis of M begins by applying both closures bound to f to the abstract value of 3. The analysis then combines the results of these two applications associating $\langle 0, \emptyset\rangle \sqcup \langle 1, \emptyset\rangle = \langle \top, \emptyset\rangle$ with a_1. As a consequence, the analysis loses all information about the value of a_2. In contrast, the analysis of the CPS version:

$$(f\ 3\ (\lambda a_1.\mathcal{F}_k[\![(\mathsf{let}\ (a_2\ (\mathsf{if0}\ a_1\ 5\ (\mathsf{if0}\ (\mathsf{sub1}\ a_1)\ 5\ 6)))\ a_2)]\!]))$$

duplicates the continuation $(\lambda a_1 \ldots.)$ when evaluating the application of each of the closures bound to f. The analysis determines that the value of a_2 is $\langle 5, \emptyset, \emptyset\rangle$ along each execution path and hence improves on the analysis of the source program. ∎

5.2 Direct vs Semantic-CPS

The character of the relationship between the direct and the semantic-CPS analysis depends on a key property of analyses.

Definition 5.3. **(*Distributivity*)** An analysis is distributive[5] if for all κ, A_i, and n:

$$\begin{aligned} &\langle \kappa, \textstyle\bigsqcup_{i=1,n} A_i\rangle \ appr^{\ominus}\ A_f \quad \text{iff} \\ &\qquad \langle \kappa, A_1\rangle \ appr^{\ominus}\ B_1 \ \ldots\ \langle \kappa, A_n\rangle \ appr^{\ominus}\ B_n \\ &\quad \text{and } A_f = \textstyle\bigsqcup_{i=1,n} B_n. \end{aligned}$$

∎

When *Distributivity* does not hold, *e.g.*, for constant propagation [9], the semantic-CPS data flow analyzer may gain information by duplicating the continuation along every execution path as in the right hand side of the condition. Otherwise, the results of the analyses are identical.

Theorem 5.4 *Let $M \in \Lambda$, then $\langle M, \kappa, \sigma\rangle \ \mathcal{C}^{\ominus} \ A_1$ if and only if:*

- $\langle M, \sigma\rangle \ \mathcal{M}^{\ominus} \ A_2$, *and* $\langle \kappa, A_2\rangle \ appr^{\ominus} \ A_3$, *and* $A_1 \sqsubseteq A_3$, *or*
- *if the Distributivity condition holds,* $\langle M, \sigma\rangle \ \mathcal{M}^{\ominus} \ A_2$, *and* $\langle \kappa, A_2\rangle \ appr^{\ominus} \ A_1$.

5.3 Syntactic-CPS vs Semantic-CPS

The semantic-CPS analyzer may yield more precise results than the syntactic-CPS analyzer since the latter may confuse the continuations collected at a given label.

Theorem 5.5 *Let $M \in \Lambda$, then:*

$$\begin{aligned} &\langle M, \mathbf{nil}, \sigma\rangle \ \mathcal{C}^{\ominus} \ A_1 \quad \textit{iff} \\ &\langle \mathcal{F}_k[\![M]\!], \delta^{\ominus}(\sigma)[k := \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle]\rangle \ \mathcal{M}_c^{\ominus} \ A_2 \end{aligned}$$

where $\delta^{\ominus}(A_1) \sqsubseteq A_2$.

[5] In the traditional framework, the lattice is usually inverted and the *Distributivity* condition is stated using the greatest lower bound $\sqcap$. In our case, the condition should be called *Continuity* but, to avoid confusion, we use the standard terminology.

6 Discussion of the Results

In summary, our theorems show that:

1. The syntactic-CPS analyzer may confuse some continuations, *i.e.*, may analyze an infeasible path, and also duplicates the analysis of continuations along every execution path, *i.e.*, may gather more information than the direct analyzer in non-distributive analyses.

2. The semantic-CPS analyzer does not suffer from the false return problem and increases the collected information in non-distributive analyses by the duplication of the analysis of continuations.

In the remainder of this section, we discuss each of the properties of the CPS analyzers in detail.

6.1 False Returns

In practice, many analyses do indeed confuse continuations when applied to CPS programs. For example, Shivers's 0CFA analysis of CPS programs [16] merges distinct control paths unnecessarily. Shivers did not relate the problem to CPS but his example [16:p.33] is essentially the example for Theorem 5.1.

Given our result, we can explain how the CPS transformation confuses some data flow analyzers that associate (approximate) information with program points. Because the CPS transformation reifies the continuation to a value that the program manipulates explicitly, the analysis of a CPS program is obligated to collect, at each variable k, the set of continuations that k may refer to during the execution of the program. Thus, when considering a return, *i.e.*, a call $(k\ W)$, the analysis applies each of the continuations bound to k and merges the results. In contrast, the analysis of the source program and the semantic-CPS analysis do not collect continuations, but only consider the *current* continuation at any program point.

6.2 Duplication

The gain of information in semantic-CPS analyzers is folklore knowledge. Nielson [13] proved that, for a small imperative language, the semantic-CPS analysis computes the MOP (meet over all paths) solution and the direct analysis computes the less precise MFP (maximum fixed point) solution; Filho and Burn [6] improved the abstract interpretations of typed call-by-name languages using the CPS transformation. Our result shows that the gain in all cases is entirely due to the duplication of the analysis of the continuation along different execution paths. In the remainder of this section, we consider the impact of this duplication on the computability and cost of the analysis.

Intuitively, the difference between the direct semantics and the CPS analyses is that the former merges all the values of an expression before analyzing the continuation and the latter apply the continuation to each of the values of an expression and merge the results. Therefore, the duplication of the analysis of the continuation depends on the number of values an expression may have.

Thus far, every expression in our language had only a finite number of values: the analysis of a conditional expression may proceed along two paths, and the analysis of a procedure call may proceed along some finite number of paths, one for each abstract closure that the term in function position evaluates to. Consequently, at each conditional and at each call site, the continuation may be duplicated along each of the possible paths, at an overall *exponential* cost in the analysis.

In a realistic language, the duplication of the continuation causes the computation of the result of the CPS analysis to become undecidable. To illustrate this point, we assume an extension of the language with an explicit looping construct and a sufficiently rich set of primitives.

Let the construct loop be an infinite loop whose exact collecting semantics returns the infinite set of values $\{0, 1, 2, \ldots\}$.[6] The extensions of the direct and semantic-CPS analyzers are:

$$\frac{u_i = \langle i, \emptyset\rangle \quad A_i = \langle u_i, \sigma\rangle}{\langle \mathsf{loop}, \sigma\rangle\ \mathcal{M}^{\ominus}\ (\bigsqcup_{i=0,\infty} A_i)}$$

$$\frac{u_i = \langle i, \emptyset\rangle \quad A_i = \langle u_i, \sigma\rangle \quad \langle \kappa, A_i\rangle\ appr^{\ominus}\ B_i}{\langle \mathsf{loop}, \kappa, \sigma\rangle\ \mathcal{C}^{\ominus}\ (\bigsqcup_{i=0,\infty} B_i)}$$

In the direct interpreter, each u_i is an abstract number $\langle i, \emptyset\rangle$ and the least upper bound of the set $\{u_i \mid i \geq 0\}$ is $\langle \top, \emptyset\rangle$. In the semantic-CPS case, the computation of $\bigsqcup_{i=0,\infty} B_i$ is undecidable. The proof is an adaptation of Kam and Ullman's proof [9] that it is undecidable to compute the MOP solution for a general program in an arbitrary monotone framework.

6.3 Conclusion

In conclusion, a *practical* analysis based on the CPS transformation should not perform any duplication when the analysis is distributive since the duplication would not yield more precise answers. In non-distributive cases, a CPS analysis should limit the amount of duplication for both computability and efficiency reasons. When the analysis of a CPS program does not perform any duplication, the net effect of transforming the program to CPS is to obscure the fact that there is only *one* control stack at any point

[6] The construct loop corresponds to the following program fragment '$x := 0$; **while** true $x := x + 1$'.

during a computation. Hence a more practical alternative is to combine heuristic in-lining algorithms with a direct-style analysis.

Acknowledgements

We thank Hans Boehm for discussions about the undecidability of the semantic-CPS analysis, and Geoffrey Burn, Bruce Duba, Juarez Filho, Cormac Flanagan, John Greiner, Bob Harper, Nevin Heintze, Peter Lee, and Frank Pfenning for comments on an earlier version of this paper and for discussions of the results.

References

[1] Appel, A. *Compiling with Continuations.* Cambridge University Press (1992).

[2] Bondorf, A. Improving binding times without explicit CPS-conversion. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1992) 1–10.

[3] Consel, C. and Danvy, O. For a better support of static data flow. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1991) 496–519.

[4] Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (1977) 238–252.

[5] Filho, J. Muylaert. Improving abstract interpretations with CPS-translation. (1993). Unpublished Manuscript.

[6] Filho, J. Muylaert and Burn, G. Continuation passing transformation and abstract interpretation. In Burn, G., Gay, S., and Ryan, M., editors, *Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods* (1993).

[7] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1993) 237–247.

[8] Hudak, P. and Young, J. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13, 2 (April 1991) 269–290.

[9] Kam, J.B. and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica*, 7 (1977) 305–317.

[10] Kelsey, R. and Hudak, P. Realistic compilation by program transformation. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages* (1989) 281–292.

[11] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM Sigplan Symposium on Compiler Construction*, Sigplan Notices, 21, 7 (1986) 219–233.

[12] Marlowe, T.J. and Ryder, B.G. Properties of data flow frameworks: A unified model. *Acta Informatica* (1990).

[13] Nielson, F. A denotational framework for data flow analysis. *Acta Informatica*, 18 (1982) 265–287.

[14] Sabry, A. and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6, 3/4 (1993) 289–360. Also in Proceedings of the ACM Conference on Lisp and Functional Programming, 1992, and Technical Report 92-180, Rice University.

[15] Sabry, A. and Felleisen, M. *Is Continuation-Passing Useful for Data Flow Analysis?* Technical Report TR94-223, Rice University (1994).

[16] Shivers, O. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, Carnegie Mellon University (1991).

[17] Steele, G. L. *Rabbit: A Compiler for Scheme.* MIT AI Memo 474, Massachusetts Institute of Technology (1978).