# Is Continuation-Passing Useful for Data Flow Analysis?

Amr Sabry Matthias Felleisen

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891[1]

sabry@cs.cmu.edu
(412) 268-7670

*Summary*

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 [14] and Burn/Filho [5, 6] support, to some degree, this belief with theoretical results. However, these investigations 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 understand the exact 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 the direct semantics of the language, the second on a continuation-semantics of the language, and the last on the 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 the analyzers establishes two 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 only 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 returns (calls to the continuation).

2. The analyzer based on the continuation semantics gets more accurate results than the direct analyzer, but again only in non-distributive analyses and by *duplicating* continuations along every execution path. However, when the idea continuation-passing is applied to 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, *e.g.*, cloning, would be as satisfactory as a CPS analysis.

[1] Permanent Address: Department of Computer Science, Rice University, Houston, TX 77251-1892.

# 1 Data Flow Analysis of Programs in CPS

Many compilers for higher-order applicative languages (Scheme, ML, Common Lisp) map source programs to programs in continuation-passing style (CPS) [1, 11, 12, 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].[2]

Even though CPS programs are widely accepted as an advantageous intermediate representation, few compiler writers can pinpoint what distinguishes the CPS representation from other intermediate representations or precisely why it is better. 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 $\beta$ rule of the lambda calculus has a simple counterpart for source programs; in other words, whatever optimizations are expressible via $\beta$-steps on CPS programs, can equally well be formulated for source programs [15].

2. The code generation phase of non-optimizing CPS compilers only requires that the intermediate representation be normalized according to simple transformations that are present in most compilers. In particular, it is unnecessary to perform a full CPS translation in the front end of the compiler [7].

Prompted by conversations with G. Burn [July 92] and with H. Boehm [August 92], we also started to investigate the exact effect of the CPS program representation on the data flow analysis of programs. Our new results reject the common belief that the CPS representation is useful for practical data flow analyses.

The organization of the rest of this paper is as follows. The next section introduces the syntax and semantics of a typical functional language. Section 3 presents the CPS transformation. In Section 4, we derive various data flow analyzers from the standard semantics. Section 5 presents all the formal theorems, which are then discussed from a practical perspective in Section 6.

# 2 $\Lambda$: Syntax and Semantics

Our source language is a simple extension of the language $\Lambda$ of the $\lambda$-calculus. 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:

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

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.

**Note:** The language is overly simple but contains enough ingredients for the comparison of the direct and CPS formulations of data flow analysis. The undecidability of the CPS analysis in the

[2] Researchers often express this view in informal discussions as opposed to formal papers. We discussed and reconfirmed 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 [15]; 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]. We suspect that many others who use the CPS transformation subscribe to the same conjecture.

last section assumes a richer set of primitives, *e.g.*, multiplication, division, strings and associated operations, etc. The treatment of the additional operations is similar to the treatment of integers, add1, and sub1. **End Note.**

---

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:

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

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

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

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

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

where $i = 1$ if $u_0 = 0$ and $i = 2$ 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 Interpreter

---

For the analysis of programs, we assume that every intermediate result is named, and that all the bound variables are unique, *e.g.*, if $(\lambda x_1.M_1)$ and $(\lambda x_2.M_2)$ are two distinct procedures, then $x_1 \neq x_2$. The restricted subset is the following language:

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

Every term in $\Lambda$ is normalizable to the restricted subset. For example, $(f\ (\mathsf{let}\ (x\ 1)\ (g\ x)))$ 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, 15].[3]

The semantics of the restricted subset of $\Lambda$ is specified by the two relations $\mathcal{M}$ and *app* defined in Figure 1. It is straightforward to show by induction on the height of the trees that if $\langle M, \rho, s \rangle\ \mathcal{M}\ A_1$ and $\langle M, \rho, s \rangle\ \mathcal{M}\ A_2$, then $A_1 = A_2$. Consequently, $\mathcal{M}$ 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. An answer is a pair that consists of a

[3] The normalization process does not affect the results of the data flow analyzers and is only for convenience. 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 [9, 14, 16]. The two treatments are identical but the elimination of the labels 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 data flow analyzers since they traverse both expressions in exactly the same manner.

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 $s$, and returns a new location $\ell$ 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 usually omit the second argument to *new* since it is easy to reconstruct from the context.)

# 3 Continuation-Passing Style

Given an evaluator for $\Lambda$, the continuation is an abstraction of the control state of the evaluator. For example, let $M$ be the program $(\mathsf{let}\ (x\ ((\lambda a.N_1)\ 5))\ N_2)$. During the evaluation of the procedure call $((\lambda a.N_1)\ 5)$, the evaluator must remember the *evaluation context* $(\mathsf{let}\ (x\ [\,])\ N_2)$ of the call as well as the environment $\rho$ in which to evaluate $N_2$. Typically, this information would be packaged in a frame and pushed on the control stack prior to the procedure call. Of course, the evaluation of the body of the procedure $N_1$ may itself push frames on the control stack. Thus the continuation $\kappa$ can in general be represented as a list of frames where each frame consists of an evaluation context and an environment [7]:

$$\kappa = \langle E_1, \rho_1\rangle :: \langle E_2, \rho_2\rangle :: \ \cdots\ :: \mathbf{nil} \qquad \text{where } E_i = (\mathsf{let}\ (x_i\ [\,])\ M_i).$$

As its name implies, the continuation-passing style transformation modifies the program such that it manipulates the continuation explicitly. The transformation may be applied to the interpreter (*semantic*-CPS transformation) or to the source program (*syntactic*-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 *ret*, 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 transformation of a source program to an equivalent 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. We assume that the continuation variables are drawn from a set $KVar = \{k, k_1, k', \ldots\}$ that is disjoint from *Var*:

$$\begin{aligned}
\mathcal{F}_k[\![V]\!] &= (k\ \mathcal{V}[\![V]\!])\\
\mathcal{F}_k[\![(\mathsf{let}\ (x\ V)\ M)]\!] &= (\mathsf{let}\ (x\ \mathcal{V}[\![V]\!])\ \mathcal{F}_k[\![M]\!])\\
\mathcal{F}_k[\![(\mathsf{let}\ (x\ (V_1\ V_2))\ M)]\!] &= (\mathcal{V}[\![V_1]\!]\ \mathcal{V}[\![V_2]\!]\ \lambda x.\mathcal{F}_k[\![M]\!])\\
\mathcal{F}_k[\![(\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M)]\!] &= (\mathsf{let}\ (k'\ \lambda x.\mathcal{F}_k[\![M]\!])\ (\mathsf{if0}\ \mathcal{V}[\![V_0]\!]\ \mathcal{F}_{k'}[\![M_1]\!]\ \mathcal{F}_{k'}[\![M_2]\!]))
\end{aligned}$$

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

The output of the CPS transformation is the language $cps(\Lambda)$:

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

The semantics of CPS programs is defined by $\mathcal{M}_c$, a specialized version of the direct interpreter $\mathcal{M}$ [7]. The evaluator for CPS terms (see Figure 3) handles procedures of two arguments and manipulates a larger set of run-time values than the direct interpreter $\mathcal{M}$ that includes continuations of the form $\langle \mathbf{co}\ x, P, \rho\rangle$. The enlarged set of run-time values reflects the salient aspect of the CPS

Domains:

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

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

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

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

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

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

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

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

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

Figure 2: Semantic-CPS Interpreter

transformation: it reifies the continuation of the evaluator to an object that the program text may manipulate.[4]

# 4 Constant Propagation

Based on well-known ideas from the area of abstract interpretation [4, 9, 14, 16], we derive a data flow analyzer from each of the previous interpreters. The first step in the derivation is to associate *one* location with each variable that holds the set of all values to which the variable is bound during the evaluation of the program. Second, we approximate the infinite sets of values, and 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 with each variable and to collect *all* the values to which the variable is bound at that location. Formally, we approximate environments $\rho$ 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 $\rho = \{x_1 \mapsto new(x_1), x_2 \mapsto new(x_2), \ldots\}$, then the abstract environment

[4] In theory, 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) confusing for data flow analyzers.

Domains:

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

Auxiliary Function:

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

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

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

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

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

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

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

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

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

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

Figure 3: Syntactic-CPS Interpreter

is $\overline{\rho} = \{x_1 \mapsto new^{-1}(new(x_1)), x_2 \mapsto new^{-1}(new(x_2)), \ldots\} = \{x_1 \mapsto x_1, x_2 \mapsto x_2, \ldots\}$. As the approximation of the environment does not provide any information, we dispense with the environment in the remainder of the paper. The approximation (and subsequent elimination) of environments immediately entails an approximation of closures, the continuations of the semantic-CPS interpreter, and the continuations of the syntactic-CPS interpreter. A closure $\langle \mathbf{cl}\ x, M, \rho \rangle$ becomes an abstract closure $\langle \mathbf{cl}^{\ominus}\ x, M \rangle$, a continuation $\langle E_1, \rho_1 \rangle :: \langle E_2, \rho_2 \rangle :: \ldots :: \mathbf{nil}$ becomes $E_1 :: E_2 :: \ldots :: \mathbf{nil}$, and a continuation $\langle \mathbf{co}\ x, M, \rho \rangle$ becomes an abstract continuation $\langle \mathbf{co}^{\ominus}\ x, M \rangle$. It follows that abstract closures and abstract continuations need not be approximated any further since, given a source program, the sets of abstract closures and abstract continuations are finite.

- For stores $s = \{new(x_1) \mapsto u_1, new(x_2) \mapsto u_2, \ldots\}$, we first recover the variable associated with each of the locations: $\{x_1 \mapsto u_1, x_2 \mapsto u_2, \ldots\}$. 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 $x \mapsto \{u_1, u_2, \ldots\}$.

The above "collecting" semantics 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

and whose least upper bound 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" may still associate an infinite set of values, *e.g.*, $\{0, 1, 2, \ldots\}$, with a given variable. The problem is that 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 divergence, we need to approximate sets of numbers to abstract numbers [10]:

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

At this point, the universe of abstract values consists of abstract numbers, abstract closures, and possibly abstract continuations for the syntactic-CPS interpreter. 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 $\mathbb{N}_\bot^\top$ used for constant propagation [10], and the second is the power set of abstract closures (ordered by the subset relation) used to approximate the control flow [16]. The ordering relation $\sqsubseteq$, as well as the least upper bound operation $\sqcup$, of the lattice of abstract values 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 $\sigma_1$ and $\sigma_2$ are stores that map variables to abstract values, then $\sigma_1 \sqsubseteq \sigma_2$ if for every variable $x$ in the domain of $\sigma_1$, we have that $\sigma_1(x) \sqsubseteq \sigma_2(x)$. The latter ordering also induces an ordering on abstract answers, *i.e.*, pairs of abstract values and abstract stores, computed component-wise.

We can now specify the collecting interpreters that manipulate abstract values. The direct abstract collecting interpreter is in Figure 4. The semantic-CPS abstract collecting interpreter is in Figure 5. The syntactic-CPS abstract collecting interpreter is in Figure 6. The interpreters use the abstract primitive operations $add1^{\ominus}$ and $sub1^{\ominus}$ instead of the precise ones:

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

## 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 $x$. We omit the proofs of correctness from this abstract.

## 4.4 Termination

When interpreted naïvely, the specifications for the abstract collecting interpreters may be perceived as partial functions that diverge on some inputs, and hence are not quite the desired data flow analyses. This is not a problem, since it is possible to detect (and hence avoid) all loops.

Domains:

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

Auxiliary Functions:

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

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

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

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

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

where $i = 1$ if $u_0 = \langle 0, \emptyset \rangle$ and $i = 2$ if $\langle 0, \emptyset \rangle \not\sqsubseteq u_0$.

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

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

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

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

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

Figure 4: Direct Abstract Collecting Interpreter

More precisely, assume we have the following fragment of a derivation tree:

$$\langle M_1, \sigma_1 \rangle\ \mathcal{M}^{\ominus}\ ? \quad \ldots \quad \langle M_i, \sigma_i \rangle\ \mathcal{M}^{\ominus}\ ? \quad \ldots \quad \langle M_j, \sigma_j \rangle\ \mathcal{M}^{\ominus}\ ? \quad \ldots$$

The evaluation of $M_1$ in store $\sigma_1$ requires the value of $M_i$ in store $\sigma_i$ which in turn requires the value of $M_j$ in store $\sigma_j$ 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, we have $\sigma_1 \ldots \sqsubseteq \sigma_i \ldots \sqsubseteq \sigma_j \ldots$. As the lattice of abstract stores does not have any infinite ascending chains, one of the $\sigma$'s in the sequence must be repeated, say $\sigma_i = \sigma_j = \sigma$. Thus, all loops will result in two calls to the direct abstract collecting interpreter with the same arguments. The same reasoning applies to the other interpreters.

By modifying the abstract collecting interpreters to compare their arguments with a history of all the previous arguments, we can therefore detect all loops. Having detected a loop, we return the least informative value paired with the current store. In the remainder of the paper, we will use "abstract collecting interpreter" or "data flow analysis" to refer to the terminating versions of the

Domains:

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

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

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

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

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

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

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

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

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

$$\frac{u = \langle add1^{\ominus}(n), \emptyset\rangle \quad \langle \kappa, \langle u, \sigma\rangle\rangle \ ret^{\ominus} \ A}{\langle \mathbf{inc}, \langle n, CL\rangle, \kappa, \sigma\rangle \ {appk_1}^{\ominus} \ A} \qquad \frac{u = \langle sub1^{\ominus}(n), \emptyset\rangle \quad \langle \kappa, \langle u, \sigma\rangle\rangle \ ret^{\ominus} \ A}{\langle \mathbf{dec}, \langle n, CL\rangle, \kappa, \sigma\rangle \ {appk_1}^{\ominus} \ A}$$

$$\frac{\langle M, \kappa, \sigma[x := \sigma(x) \sqcup u]\rangle \ \mathcal{C}^{\ominus} \ A}{\langle \langle \mathbf{cl}^{\ominus} \ x, M\rangle, u, \kappa, \sigma\rangle \ {appk_1}^{\ominus} \ A}$$

$ret^{\ominus} : (\overline{Con} \times \overline{Ans}) \to \overline{Ans}$

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

Figure 5: Semantic-CPS Abstract Collecting Interpreter

interpreters that detect loops as above.[5]

# 5 Formal Relationships

Given the direct and CPS-based data flow analyzers, we turn our attention to the connection between the analyses. For the connection between the direct and syntactic-CPS analyses, we need to map abstract direct values to abstract CPS values:

$$\mathcal{V}^{\ominus}(\langle \mathbf{cl}^{\ominus} \ x_1, M_1\rangle) = \langle \mathbf{cl}^{\ominus} \ x_1 k_1, \mathcal{F}_{k_1}[\![M_1]\!]\rangle, \quad \mathcal{V}^{\ominus}(\mathbf{inc}) = \mathbf{inck}, \quad \mathcal{V}^{\ominus}(\mathbf{dec}) = \mathbf{deck}$$

We also extend $\delta^{\ominus}$ to work on stores by applying it to the value at each location, and to answers by applying it to both components.

[5] The termination argument of our data flow analyzers is not based on computing the fixed-point of a function by iteration.

Domains:

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

Auxiliary Function:

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

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

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

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

$$\frac{u_0 = \phi_c^{\ominus}(W_0, \sigma) \quad \langle P_i, \sigma[k := \sigma(k) \sqcup \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ A}{\langle (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W_0\ P_1\ P_2)), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A}$$

where $i = 1$ if $u_0 = \langle 0, \emptyset, \emptyset \rangle$ and $i = 2$ if $\langle 0, \emptyset, \emptyset \rangle \not\sqsubseteq u_0$.

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

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

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

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

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

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

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

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

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

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

Figure 6: Syntactic-CPS Abstract Collecting Interpreter

The first theorem establishes that the direct analysis may be *more* precise than the syntactic-CPS analysis.

**Theorem 5.1** *There exists a term $M \in \Lambda$ such that:*

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

**Proof**. Take $M = (\mathsf{let}\ (f\ \lambda x.x)\ (\mathsf{let}\ (a_1\ (f\ 1))\ (\mathsf{let}\ (a_2\ (f\ 2))\ a_2)))$ and $\sigma$ be the initial store that maps every variable to $\langle \bot, \emptyset\rangle$. By straightforward calculations, the analysis of the source program determines that the variable $a_1$ is the constant 1, while the analysis of the CPS program determines that $a_1$ can be any integer ($\top$). ∎

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* $\sigma_1$, $\delta^{\ominus}(\sigma_1(x)) \sqsupseteq \sigma_2(x)$.

**Proof**. Take $M = (\mathsf{let}\ (a_1\ (\mathsf{if0}\ x\ 0\ 1))\ (\mathsf{let}\ (a_2\ (\mathsf{if0}\ a_1\ (+\ a_1\ 3)\ (+\ a_1\ 2)))\ a_2))$. The analysis of the CPS program determines that $a_1$ is the constant 3 while the analysis of the source program determines that $a_1$ can be any integer. ∎

The character of the relationship between the direct and the semantic-CPS analysis depends on whether the following condition holds.

**Definition 5.3.** (*Distributivity*) We call an analysis distributive[6] if for all $\kappa$, $A_i$, and $n$:

$$\langle \kappa, \bigsqcup_{i=1,n} A_i\rangle \ ret^{\ominus} \ A_f \quad \text{iff} \quad \langle \kappa, A_1\rangle \ ret^{\ominus} \ B_1 \ \ldots \ \langle \kappa, A_n\rangle \ ret^{\ominus} \ B_n \ \text{ and } A_f = \bigsqcup_{i=1,n} B_n.$$

∎

When the *Distributivity* condition does not hold, *e.g.*, for constant propagation [10, 13], 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 semantic-CPS analysis is identical to the direct analysis of the program.

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

**Proof Idea**. The proof is by induction on the height of the derivation of $A_1$. The *Distributivity* condition is used only at conditionals when both branches are evaluated, and at $appk^{\ominus}$ when more than one closure is applied. ∎

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:*

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

[6] 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, the theorems show that the syntactic-CPS analyzer may confuse some continuations, *i.e.*, may consider a path that corresponds to a "false return", and that it duplicates the analysis of continuations along every execution path, *i.e.*, may gather more information than the direct analyzer in non-distributive analyses. 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.

## 6.1 False Returns

In practice, we have found that most analyses do indeed confuse continuations when applied to CPS programs. For example, in his 0CFA analysis of CPS programs, Shivers [16] observes that the analysis causes distinct control paths to merge unnecessarily. Although Shivers did not relate the problem to CPS, his example [16:p.33] is essentially the example we used in Theorem 5.1. Similarly, Heintze's "set based analysis" [8] also confuses continuations when applied to CPS programs. In particular, the set based analysis of our example in Theorem 5.1 associates the set $\{1\}$ with $a_1$ in the direct case and the set $\{1, 2\}$ in the CPS case.

The above examples prove the observation of many people that the conversion to CPS confuses some data flow analyses. Given our result, we can explain how this observation holds for *any* data flow analysis that associates (approximate) information with program points. Because the CPS transformation reifies the continuation to a value that the program may manipulate, 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 that may be bound to $k$ and merges the results. In contrast, the analysis of the source program and the semantic-CPS analysis do not collect continuations, and thus only consider the *current* continuation at any program point.

## 6.2 Duplication

The gain of information in semantic-CPS analyzers is folklore knowledge; Nielson [14] 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] improve the abstract interpretations of typed call-by-name languages using the CPS transformation. Since the gain in all cases is entirely due to the duplication of the analysis of the continuation along different execution paths, we consider the impact of this duplication on the cost and computability of the analysis.

Intuitively, the difference between the direct semantics and the CPS analyses is that the direct semantics merges all the values of an expression before calling the continuation and the CPS analyses apply the continuation to each of the values of an expression and then merge the results. Therefore, the number of times the continuation is duplicated depends on the maximum number of values that an expression may have.

So far, every expression in our language had only a finite number of values: the analysis of a conditional expression may proceed along two paths (the "then" path and the "else" path), 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.

More importantly, in a realistic language, the duplication of the continuation causes the computation of the result of the CPS analysis to become undecidable. This point was not apparent so far since our language does not have any explicit looping constructs, but instead expresses loops using

self-application of procedures, *e.g.*, the $Y$-combinator. If we extend the language with a built-in looping construct and assume a sufficiently rich set of primitives, the undecidability of the semantic-CPS analysis becomes apparent.

Let the construct loop be an infinite loop whose exact collecting semantics returns the infinite set of values $\{0, 1, 2, \ldots\}$.[7] The extensions of the direct and semantic-CPS analyzers are:

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

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, each element of the set whose least upper bound we want to compute is the result of applying the continuation to an abstract number $\langle i, \emptyset\rangle$. In a sufficiently expressive language, we can construct a continuation that, given $\langle i, \emptyset\rangle$, simulates a Turing machine $H$ for $i$ steps returning $\langle 0, \emptyset\rangle$ if $H$ halts in less than $i$ steps and $\langle 1, \emptyset\rangle$ otherwise. Thus, the least upper bound of the set is $\langle 1, \emptyset\rangle$ if and only if $H$ does not halt in any number of steps, which implies that the computation of the least upper bound is undecidable.

The above argument is an adaptation of Kam and Ullman's [10] proof that, given an arbitrary monotone framework, it is undecidable to compute the MOP solution for each program. It is easy to modify the argument to prove the undecidability of the semantic-CPS analysis for other languages.

### 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. In the case 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 during a computation. Alternatively, a direct analysis that allows some duplication, *e.g.*, in-lining some procedures in flow-sensitive analyses, would be as satisfactory as a practical CPS-based analysis.

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

[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] Heintze, N. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University (1992).

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

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

[11] 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.

[12] 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.

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

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

[15] Sabry, A. and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6, 3/4 (1993) 287–358. Preliminary version in Proceedings of the ACM Conference on Lisp and Functional Programming, 1992. Also Technical Report 92-180, Rice University.

[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).