Is Continuation-Passing Useful for Data Flow Analysis?

Amr Sabry   Matthias Felleisen

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

Summary

The widespread use of the continuation-passing style (CPS) transformation in compilers, partial evaluators, and abstract interpretations suggests that it is useful for program analysis. Indeed, there is some theoretical and practical evidence that the CPS framework may be used to improve the precision of program analyses.

In order to formalize the exact impact of CPS on program analysis, we derive three canonical data flow analyzers for a typical higher-order applicative language. The first analyzer is based on the direct semantics of the language, the second is based on a continuation-semantics of the language, and the last is based on the semantics of CPS terms. All analyzers compute the control flow graph of the source program and hence the results apply to a large class of data flow analyses. We then study the information gathered by the analyzers and establish that:

1. The direct analysis of a source program is *incomparable* to the analysis of the CPS-transformed program. In other words, the transformation of the source program to CPS 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, again only in non-distributive analyses and by *duplicating* continuations along every execution path. The analysis based on the continuation-semantics also gets more accurate results than the analysis of CPS programs by not confusing distinct returns.

In summary, CPS only improves the precision of non-distributive data flow analyses by duplication of the analysis of the continuation: unbounded duplication of the continuation yields an undecidable analysis, controlled duplication may be achieved by alternative methods like cloning, in-lining, and loop unrolling, and no duplication eliminates the advantage of CPS. In the latter case, the net effect of transforming a program to CPS is to potentially confuse calls to the continuation.

[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 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 CPS 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 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 [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 question whether the CPS program representation affects the data flow analysis of programs and, if so, whether it improves the precision of the results. Our answer rejects the common belief that the latter is the case. More precisely, we can show that the most common data flow analyses for higher-order languages produce *incomparable* results when applied to a source program and its CPS counterpart. That is, there is a program whose analysis produces better results than the analysis of its CPS version, and *vice versa*. On the other hand, we can prove that applying an analyzer *based on a continuation semantics* to the *source program* yields more information than the original analyzer in both cases. The latter is in the spirit of a result by Nielson [14] who considered a small imperative language, and a result by Filho and Burn [6] who considered a typed call-by-name language.

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 in a more practical setting in Section 6. The appendix includes two of the most involved proofs.

2 Λ: Syntax and Semantics

Our source language is a simple extension of the language Λ of the λ-calculus. It is the core of typical higher-order languages like Scheme, Lisp, and ML, and contains all the essential ingredients for the comparison of the direct and continuation-passing formulations of data flow analysis. The set of terms includes values, applications, let-expressions, and conditionals; the set of syntactic values

[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.

contains numerals ($n \in \mathbb{Z}$), variables ($x \in \mathit{Var}$), primitive procedures,[3] 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 $\mathsf{if0}$ branches to the second or third subexpression depending on whether the first subexpression evaluates to 0 or not.

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 we consider for analysis is the following:

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

The assumptions simplify the semantics and the derivation of the data flow analyzers without restricting the set of valid programs: every term in Λ is normalizable to 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 earlier identified as the A-reductions [7, 15]. It is important to note that the normalization process does not affect the results of the data flow analyzers. The first part 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 $(\mathsf{add1}\ (\mathsf{let}\ (x\ V)\ 0))$ would be rewritten as $(\mathsf{let}\ (x\ V)\ (\mathsf{add1}\ 0))$. Again, the change is transparent to the data flow analyzers since they traverse both expressions in exactly the same manner.

The semantics of programs in the restricted subset of Λ is specified by the two predicates $\mathcal{M}$ and app defined in Figure 1. It is straightforward to show by induction on the height of the trees that if $\langle M, \rho, s\rangle$ is $\mathcal{M}$-related to answers A_1 and 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 semantic value and a store. The set of semantic 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 state with the value u at the new location. Thus, the bound variable of a procedure 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 ℓ from which it is possible to recover x, *i.e.*, $\mathit{new}(x, s) = \ell \notin \mathit{dom}(s)$ and $x = \mathit{new}^{-1}(\ell)$. (For brevity, we will usually omit the second argument to *new*.)

[3]The restriction of the set of primitive procedures to add1 and sub1 is only for convenience and does not affect the generality of the language since other operations on integers, *e.g.*, multiplication, are definable via recursive procedures. The latter procedures are expressible using the call-by-value Y-combinator.

Domains:

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

Auxiliary Function:

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

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

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

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

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

where $i = 1$ if $u_0 = 0$ and $i = 2$ otherwise.

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

$$\frac{}{\langle \mathbf{inc}, n, s \rangle \ app \ \langle (n+1), s \rangle} \qquad \frac{}{\langle \mathbf{dec}, n, s \rangle \ app \ \langle (n-1), s \rangle}$$

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

Figure 1: Direct Interpreter

3 Continuation-Passing Style

The continuation-passing style transformation may be applied to the interpreter (if we think of the implementation of the interpreter as a program) or to the source program. To distinguish the two approaches we will refer 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

Given an evaluator for Λ, the continuation is an abstraction of the control state of the evaluator. For example, let M be the program $(\mathsf{let}\ (x\ ((\lambda a.N_1)\ 5))\ N_2)$. In a conventional implementation, the evaluator would push a frame on the control stack before performing the procedure call $((\lambda a.N_1)\ 5)$. Conceptually, the frame would include the *evaluation context* $(\mathsf{let}\ (x\ [\,])\ N_2)$, and the environment ρ in which to evaluate N_2. Of course, 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]:

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

3.2 Semantic-CPS Transformation

When applied to the interpreter, the CPS transformation adds an additional continuation argument to the interpretation of every subexpression (see Figure 2). The new function *appr* corresponds to the "return" operation of an abstract machine: it binds the return value to a variable, restores the environment, pops the control stack, and jumps to the next instruction.

Domains:

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

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

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

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

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

where $i = 1$ if $u_0 = 0$ and $i = 2$ otherwise.

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

$$\frac{\langle \kappa, \langle (n+1), s\rangle\rangle \ appr \ A}{\langle \mathbf{inc}, n, \kappa, s\rangle \ appk \ A} \qquad \frac{\langle \kappa, \langle (n-1), s\rangle\rangle \ appr \ A}{\langle \mathbf{dec}, n, \kappa, s\rangle \ appk \ A}$$

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

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

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

Figure 2: Semantic-CPS Interpreter

The formal relationship between the direct and semantic-CPS interpreters is subject of the following lemma.

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

Proof Idea. By induction on the height of the derivation of A. The statement of the lemma must be strengthened to take continuations other than **nil** into account. ∎

3.3 Syntactic-CPS Transformation

The transformation of a source program to 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 will assume that the continuation variables are drawn from a set $KVar = \{k, k_1, k', \ldots\}$ that is disjoint from Var.

Definition 3.2. (*The Syntactic-CPS Transformation*)

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

$$
\begin{array}{rcl}
\mathcal{V}[\![n]\!] & = & n \\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\mathsf{add1}]\!] & = & \mathsf{add1k} \\
\mathcal{V}[\![\mathsf{sub1}]\!] & = & \mathsf{sub1k} \\
\mathcal{V}[\![(\lambda x.M)]\!] & = & (\lambda xk.\mathcal{F}_k[\![M]\!])
\end{array}
$$

∎

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

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

The semantics of CPS programs is defined using a specialized version $\mathcal{M}_c$ 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 semantic values that includes continuations of the form $\langle \mathbf{co}\ x, P, \rho\rangle$. The larger set of semantic values reflects the salient aspect of the CPS transformation: it reifies the continuation of the evaluator to an object that the program text may manipulate.

In order to establish the formal relationship between the semantics of a direct term and the semantics of its CPS-transform, we define the function δ that relates direct semantic values to their CPS counterparts:

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

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

The following lemma gives the precise connection between the semantic-CPS interpreter and the syntactic-CPS interpreter. Given Lemma 3.1, it also relates the syntactic-CPS interpreter to the direct one.

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

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

Proof. See Appendix A. ∎

4 Constant Propagation by Abstract Interpretation

Exploiting 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 transform the standard semantics into a collecting semantics. While the former attaches the current value of each variable to the corresponding location, the latter associates the variable with the (potentially infinite) set of values to which it is bound during the evaluation of the program. Second, we approximate the infinite sets of values, and modify the collecting semantics to detect and recover from all loops when computing over the universe of approximate values.

Domains:

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

Auxiliary Function:

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

$\mathcal{M}_c : (cps(\Lambda) \times Env \times Sta) \to Ans$

$$\frac{\kappa = s(\rho(k)) \quad u = \phi_c(W, \rho, s) \quad \langle \kappa, \langle u, s \rangle \rangle\ appr_c\ A}{\langle (k\ W), \rho, s \rangle\ \mathcal{M}_c\ A}$$

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

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

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

where $i = 1$ if $u_0 = 0$ and $i = 2$ otherwise.

$app_c : (Val \times Val \times Val \times Sta) \to Ans$

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

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

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

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

Figure 3: Syntactic-CPS Interpreter

4.1 Collecting Semantics

The standard interpreters may diverge on some inputs and hence may not be used as static analysis tools. One of the sources of divergence is that the interpreters create a *new* location at each procedure call. The first step in the derivation of the data flow analyzers is to bound 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 $\rho = \{x_1 \mapsto new(x_1), x_2 \mapsto new(x_2), \ldots\}$, then the abstract environment is $\overline{\rho} = \{new^{-1}(new(x_1)) \mapsto x_1, new^{-1}(new(x_2)) \mapsto x_2, \ldots\} = \{x_1 \mapsto x_1, x_2 \mapsto$

$x_2, \ldots\}$. As the approximation of the environment does not provide any information, we totally 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 $\langle \mathbf{co}^{\ominus}\ x, M\rangle$. We will use an "overline" to denote all these approximations.

- 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 $x \mapsto \{u_1, u_2, \ldots\}$.

In summary, the collecting semantics associates a set with each variable. Intuitively, the larger the set, the less information is available at static 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 Abstract Collecting Interpreters

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, and hence is not computable. We therefore need to approximate sets of numbers to abstract numbers. A common approximation [10] is:

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

At this point, the universe of abstract values consists of abstract numbers, abstract closures, and possibly abstract continuations. It remains to impose an order $\sqsubseteq$ on the abstract 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 (finite) 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 point-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 $\overline{S_1} \sqsubseteq \overline{S_2}$. In other words, the "overline" approximation preserves the relation "is more precise than".

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 (finite) power set of abstract closures, and the (finite) power set of abstract continuations.

The ordering of abstract values induces an ordering on states. If σ_1 and σ_2 are states that map variables to abstract values, then $\sigma_1 \sqsubseteq \sigma_2$ if for every variable x in the domain of σ_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 states, computed point-wise.

We can now specify collecting interpreters that manipulate abstract values. The interpreters use the abstract primitive operations $add1^{\ominus}$ and $sub1^{\ominus}$ instead of the precise ones:

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

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 Figures 6 and 7.

Domains:

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

Auxiliary Functions:

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

$\mathcal{M}^{\ominus} : (\Lambda \times \overline{Sta}) \to \overline{Ans}$

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

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

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

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

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

$app^{\ominus} : (\overline{Val} \times \overline{Val} \times \overline{Sta}) \to \overline{Ans}$

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

$app_1^{\ominus} : (\overline{Clo} \times \overline{Val} \times \overline{Sta}) \to \overline{Ans}$

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

Figure 4: Direct Abstract Collecting Interpreter

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 *Let* $\overline{s} \sqsubseteq \sigma$*:*

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

Domains:

$$\overline{Ans} = \overline{Val} \times \overline{Sta} \qquad \overline{Sta} = Var \twoheadrightarrow \overline{Val}$$

$$\overline{Con} = \Lambda(E) :: \overline{Con} + \mathbf{nil} \qquad \overline{Val} = Num \times \mathcal{P}(\overline{Clo}) \qquad \overline{Clo} = (Var \times \Lambda) + \mathbf{inc} + \mathbf{dec}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 5: Semantic-CPS Abstract Collecting Interpreter

Proof Idea. The proofs proceed by induction on the height of the derivation of $\langle u_1, s_1\rangle$. For the direct interpreter, the base case is when M is a value V. Then $u_1 = \phi(V, \rho, s)$, $s_1 = s$, $u_2 = \phi^{\ominus}(V, \sigma)$, and $\sigma_2 = \sigma$. It suffices to check that $\overline{\{u_1\}} \sqsubseteq u_2$ which is straightforward. We only show one inductive case when $M = (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ N)$. Let $u_0 = \phi(V_0, \rho, s)$, then as before we have that $\overline{\{u_0\}} \sqsubseteq \phi^{\ominus}(V_0, \sigma)$. There are two cases:

1. $u_0 = 0$, then $\langle 0, \emptyset\rangle \sqsubseteq \phi^{\ominus}(V_0, \sigma)$, *i.e.*, either $\langle 0, \emptyset\rangle = \phi^{\ominus}(V_0, \sigma)$ or $\langle 0, \emptyset\rangle \sqsubset \phi^{\ominus}(V_0, \sigma)$. The abstract collecting interpreter proceeds with the evaluation of M_1 or with the evaluation of both M_1 and M_2. The result follows by induction and the fact that the results of the two branches are combined.

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

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

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

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

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

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

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

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

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

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

Figure 6: Syntactic-CPS Abstract Collecting Interpreter (I)

2. $u_0 \neq 0$, then u_0 can be any number n other than 0 or a closure. It follows that $\phi^{\ominus}(V_0, \sigma) \sqsupseteq \langle n, CL \rangle$ or $\phi^{\ominus}(V_0, \sigma) \sqsupseteq \langle \top, CL \rangle$ where CL is an arbitrary set of closures. Either M_2 is evaluated or both branches are evaluated, and the result follows as in the first case. ∎

4.4 Termination

The abstract collecting interpreters may 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.

More precisely, assume we have the following (potentially infinite) 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 state σ_1 requires the value of M_i in state σ_i which in turn requires the

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

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

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

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

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

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

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

Figure 7: Syntactic-CPS Abstract Collecting Interpreter (II)

value of M_j in state 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, we have $\sigma_1 \ldots \sqsubseteq \sigma_i \ldots \sqsubseteq \sigma_j \ldots$. As the lattice of abstract states does not have any infinite ascending chains, one of the σ's in the sequence must be repeated, say $\sigma_i = \sigma_j = \sigma$. Thus, all loops must manifest themselves in two calls to the direct abstract collecting interpreter with the same arguments $\langle M, \sigma\rangle$. 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 state. 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 $\langle \mathbf{cl}^{\ominus}\ x, M\rangle$ in the program. Similarly, the semantic-CPS interpreter returns $\langle\langle \top, CL^{\top}\rangle, \sigma\rangle$ 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

Given the direct and CPS-based data flow analyzers, we turn our attention to the connection between the analyses. The proofs follow the same pattern of the proofs for the standard interpreters (Lemmas 3.1 and 3.3). 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:

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

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

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 a term $M \in \Lambda$ such that:*

- $\langle M, \sigma\rangle\ \mathcal{M}^{\ominus}\ A_1$,
- $\langle \mathcal{F}_k[\![M]\!], \delta^{\ominus}(\sigma)[k := \langle \perp, \emptyset, \{\mathbf{stop}\}\rangle]\rangle\ \mathcal{M}_c^{\ominus}\ A_2$,
- *and* $\delta^{\ominus}(A_1) \sqsubseteq A_2$.

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

$$\sigma = \{a_1 \mapsto \langle \perp, \emptyset\rangle,\ a_2 \mapsto \langle \perp, \emptyset\rangle,\ f \mapsto \langle \perp, \{\langle \mathbf{cl}^{\ominus}\ x, x\rangle\}\rangle,\ x \mapsto \langle \perp, \emptyset\rangle\}.$$

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

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

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

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

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

$$\begin{aligned}
u_2 &= \langle \top, CL^{\top}, K^{\top}\rangle \\
\sigma_2 &= \{a_1 \mapsto \langle \top, \emptyset, \emptyset\rangle, \\
&\quad\ a_2 \mapsto \langle \top, \emptyset, \emptyset\rangle, \\
&\quad\ f \mapsto \langle \perp, \{\langle \mathbf{cl}^{\ominus}\ xk_1, (k_1\ x)\rangle\}, \emptyset\rangle, \\
&\quad\ x \mapsto \langle \top, \emptyset, \emptyset\rangle, \\
&\quad\ k_1 \mapsto \langle \perp, \emptyset, \{\ \langle \mathbf{co}^{\ominus}\ a_1, (f\ 2\ (\lambda a_2.(k\ a_2)))\rangle, \langle \mathbf{co}^{\ominus}\ a_2, (k\ a_2)\rangle\}\rangle \\
&\quad\ k \mapsto \langle \perp, \emptyset, \{\mathbf{stop}\}\rangle]
\end{aligned}$$

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 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. 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:

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

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 $\mathcal{F}_k[\![M]\!]$:

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

analyzes both $(k'\ 0)$ and $(k'\ 1)$ in a state 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:

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

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 whether the following condition holds.

Definition 5.3. (*Distributivity*) We call an analysis distributive[4] if for all κ, A_i, and n:

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

∎

When the *Distributivity* condition does not hold, *e.g.*, 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* $\langle \kappa, A_2 \rangle \; appr^{\ominus} \; A_3$, *and* $A_1 \sqsubseteq A_3$.
- *or if the Distributivity condition holds* $\langle M, \sigma \rangle \; \mathcal{M}^{\ominus} \; A_2$, *and* $\langle \kappa, A_2 \rangle \; appr^{\ominus} \; A_1$,

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. ∎

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 (cf. Theorem 5.1).

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

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

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

Proof. See Appendix B. ∎

6 Discussion of the Results

Intuitively, the previous theorems show that:

1. The syntactic-CPS analyzer:
   - may confuse some continuations, and hence may consider a path that corresponds to a "false return",
   - duplicates the analysis of continuations along every execution path, and hence 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 but still duplicates the analysis of continuations.

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

[4] 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 we prefer to use the standard terminology.

6.1 False Returns

All our data flow analyzers collect (an approximation of) the set of values that may be bound to a variable and associate this set with the variable. By reifying the continuation to a value that the program may manipulate, the syntactic-CPS analyzer 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 call $(k\ W)$, the analyzer applies each of the continuations that may be bound to k and merges the results. In contrast, both the direct analyzer and the semantic-CPS analyzer do not collect continuations and thus consider only the current continuation at this point.

In practice, we have found that most analyses do indeed confuse continuations when applied to CPS programs. For example, in his OCFA analysis of CPS programs, Shivers observes that the analysis causes distinct control paths to merge unnecessarily [16]. Although Shivers did not relate the problem to CPS, his example [16:p.33] is essentially the example we used in Theorem 5.1. As another example, Heintze's "set based analysis" [8] also confuses continuations when applied to CPS programs. (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.)

6.2 Duplication

The other aspect of CPS analyzers is that they may gain information by duplicating the analysis of the continuation along different execution paths. Mathematically speaking, the gain comes from the fact that for any n:

$$A \sqsupseteq (\bigsqcup_{i=1,n} B_i) \qquad \text{where } \langle \kappa, \bigsqcup_{i=1,n} A_i \rangle\ appr^{\ominus}\ A \qquad \text{and } \langle \kappa, A_i \rangle\ appr^{\ominus}\ B_i$$

In other words, merging all the possible values of an expression before calling the continuation is less precise than calling the continuation on each of the possible values and then merging the results.

In the previous sections, we have limited ourselves to cases where each expression can only have a finite number of values: a conditional may have two values, and a variable in function position may refer to a finite number of abstract closures. In that case, the extra precision comes at an exponential cost in the analysis since there is a potential duplication of the continuation at every program point.

Some special treatments of looping constructs associate an infinite set of values with an expression [14]. In that case, the unbounded duplication of the continuation causes the computation of the least upper bound operation to become undecidable. To illustrate this point, we extend our language with a construct loop whose exact collecting semantics returns the infinite set of values $\{0, 1, 2, \ldots\}$. Although the construct is already expressible in the language via the Y-combinator, the following special treatment of the loop will cause the semantic-CPS analysis to be undecidable:

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

In the direct interpreter, each u_i is a 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 [10] that, given an arbitrary monotone framework, it is undecidable to compute the MOP (meet over all paths) solution for each program:

Given a Turing machine H, we construct the continuation κ such that:

$$\langle \kappa, \langle \langle n, \emptyset \rangle, \sigma \rangle \rangle\ appr^{\ominus}\ \langle \langle n', \emptyset \rangle, \sigma \rangle$$

where:

- $n' = n$ if n is $\bot$ or $\top$, and
- if $n = i$ for some natural number i, then the continuation simulates the Turing machine H for i steps and returns $n' = 0$ if H halts in less than i steps and $n' = 1$ if H fails to terminate after i steps.

Given the above construction, each B_i in the set whose least upper bound we want to compute is either $\langle\langle 0, \emptyset\rangle, \sigma\rangle$ (indicating H halts in less than i steps), or $\langle\langle 1, \emptyset\rangle, \sigma\rangle$ (indicating H does not halt in less than i steps). Therefore, the least upper of the set is either $\langle\langle 0, \emptyset\rangle, \sigma\rangle$, $\langle\langle 1, \emptyset\rangle, \sigma\rangle$, or $\langle\langle \top, \emptyset\rangle, \sigma\rangle$. In the first case, all entries are $\langle\langle 0, \emptyset\rangle, \sigma\rangle$ indicating that H terminates in 0 steps. In the second case, all entries are $\langle\langle 1, \emptyset\rangle, \sigma\rangle$ indicating that H does not halt in any number of steps. In the last case, some of the entries (the first j entries) are $\langle\langle 1, \emptyset\rangle, \sigma\rangle$ and some (all remaining entries) are $\langle\langle 0, \emptyset\rangle, \sigma\rangle$ indicating that H halts after j steps. Thus, we can determine whether the Turing machine H halts or not by computing the least upper bound of the set $\{B_i \mid i \geq 0\}$, which implies that the latter operation must be undecidable.

6.3 Summary and Conclusion

To summarize, a practical analysis based on the CPS transformation should not perform any duplication when the analysis is distributive and should limit the amount of duplication otherwise. 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.

When the analysis of a CPS program does not perform any duplication, then 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.

Finally, the results above are applicable for all data flow analyses that are expressible in the traditional framework using a control flow graph and a lattice. The reason is that part of our analysis implicitly constructs a flow graph, and the precision of the control flow graph directly affects the precision of other data flow analyses based on it.

Acknowledgements

We thank Hans Boehm, Geoffrey Burn, Juarez Filho, Bob Harper, Nevin Heintze, Peter Lee, and Frank Pfenning for comments on an earlier version of this paper and for discussions of the results.

A Proof of Lemma 3.3

The proof requires a more general statement of the lemma.

Lemma A.1 *Let $M \in \Lambda$, and let:*

$$\begin{aligned}
E_i &= (\mathsf{let}\ (u_i\ [\,])\ N_i) \\
\kappa(h, i, \ldots, j, \top) &= \langle E_h, \rho_h\rangle :: \langle E_i, \rho_i\rangle :: \ldots :: \langle E_j, \rho_j\rangle :: \mathbf{nil} \\
\mathbf{co}(i, j) &= \langle \mathbf{co}\ u_i, \mathcal{F}_{k_j}[\![N_i]\!], \rho_i[k_j := new(k_j)]\rangle \\
s(h, i, \ldots, j, \top) &= s[new(k_h) := \mathbf{co}(h, i), \ldots, new(k_j) := \mathbf{co}(j, \top), new(k_\top) := \mathbf{stop}]\rangle
\end{aligned}$$

then:

$$\begin{aligned}
&\langle M, \rho, \kappa(h, \ldots, \top), s\rangle\ \mathcal{C}\ \langle u_1, s_1\rangle \quad \textit{iff} \\
&\langle \mathcal{F}_{k_h}[\![M]\!], \rho[k_h := new(k_h)], \delta(s)(q, \ldots, h, \ldots, \top)\rangle\ \mathcal{M}_c\ \langle \delta u_1, \delta(s_1)(p, \ldots, q, \ldots, h, \ldots, \top)\rangle
\end{aligned}$$

Proof. The proof is by induction on the height of the derivation of $\langle u_1, s_1\rangle$. For the base case (height $= 1$), we must have that M is a value V and the continuation is the initial one:

$$\frac{u = \phi(V,\rho,s) \quad \langle \mathbf{nil}, \langle u, s\rangle\rangle \ appr\ \langle u, s\rangle}{\langle V, \rho, \kappa(\top), s\rangle\ \mathcal{C}\ \langle u, s\rangle}$$

$$\frac{\kappa = \mathbf{stop} \quad u' = \phi_c(\mathcal{V}[\![V]\!], \rho[k_\top := new(k_\top)], s') \quad \langle \mathbf{stop}, \langle u', s'\rangle\rangle\ appr_c\ \langle u', s'\rangle}{\langle (k_\top\ \mathcal{V}[\![V]\!]), \rho[k_\top := new(k_\top)], s'\rangle\ \mathcal{M}_c\ \langle u', s'\rangle}$$

where $s' = \delta(s)(q, \ldots, \top)$. The result follows since $u' = \delta(u)$.

For the inductive case (height > 1), we proceed by cases of M:

- $M = V$:

$$\frac{u = \phi(V,\rho,s) \quad \dfrac{\langle N_h, \rho_h[u_h := new(u_h)], \kappa(i, \ldots, \top), s[new(u_h) := u]\rangle\ \mathcal{C}\ A}{\langle \kappa(h, i, \ldots, \top), \langle u, s\rangle\rangle\ appr\ A}}{\langle V, \rho, \kappa(h, i, \ldots, \top), s\rangle\ \mathcal{C}\ A}$$

$$\frac{u' = \delta(u) \quad \dfrac{\langle \mathcal{F}_{k_i}[\![N_h]\!], \rho_h[u_h := new(u_h)][k_i := new(k_i)], s'[new(u_h) := u']\rangle\ \mathcal{M}_c\ A'}{\langle \mathbf{co}(h,i), \langle u', s'\rangle\rangle\ appr_c\ A'}}{\langle (k_h\ \mathcal{V}[\![V]\!]), \rho[k_h := new(k_h)], s'\rangle\ \mathcal{M}_c\ A'}$$

  where $s' = \delta(s)(q, \ldots, h, i, \ldots, \top)$. The result follows since A and A' are correctly related by induction.

- $M = (\mathsf{let}\ (x\ V)\ N)$:

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

$$\frac{u' = \delta(u) \quad \langle \mathcal{F}_{k_h}[\![N]\!], \rho[k_h := new(k_h)][x := new(x)], s'[new(x) := u']\rangle\ \mathcal{M}_c\ A'}{\langle (\mathsf{let}\ (x\ \mathcal{V}[\![V]\!])\ \mathcal{F}_{k_h}[\![N]\!]), \rho[k_h := new(k_h)], s'\rangle\ \mathcal{M}_c\ A'}$$

  where $s' = \delta(s)(q, \ldots, h, \ldots, \top)$. The result follows since A and A' are correctly related by induction.

- $M = (\mathsf{let}\ (u_g\ (V_1\ V_2))\ N_g)$:

$$\frac{u_1 = \phi(V_1,\rho_g,s) \quad u_2 = \phi(V_2,\rho_g,s) \quad \langle u_1, u_2, \kappa(g, h, \ldots, \top), s\rangle\ appk\ A}{\langle (\mathsf{let}\ (u_g\ (V_1\ V_2))\ N_g), \rho_g, \kappa(h, \ldots, \top), s\rangle\ \mathcal{C}\ A}$$

$$\frac{u_1' = \delta(u_1) \quad u_2' = \delta(u_2) \quad \langle u_1', u_2', \mathbf{co}(g,h), s'\rangle\ app_c\ A'}{\langle (\mathcal{V}[\![V_1]\!]\ \mathcal{V}[\![V_2]\!]\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])), \rho_g[k_h := new(k_h)], s'\rangle\ \mathcal{M}_c\ A'}$$

  where $s' = \delta(s)(q, \ldots, h, \ldots, \top)$. We proceed by cases of u_1:

  - $u_1 = \mathbf{inc}$ and $u_2 = n$:

$$\frac{\dfrac{\langle N_g, \rho_g[u_g := new(u_g)], \kappa(h, \ldots, \top), s[new(u_g) := (n+1)]\rangle\ \mathcal{C}\ A}{\langle \kappa(g, h, \ldots, \top), \langle (n+1), s\rangle\rangle\ appr\ A}}{\langle \mathbf{inc}, n, \kappa(g, h, \ldots, \top), s\rangle\ appk\ A}$$

$$\frac{\dfrac{\langle \mathcal{F}_{k_h}[\![N_g]\!], \rho_g[k_h := new(k_h)][u_g := new(u_g)], s'[new(u_g) := (n+1)]\rangle\ \mathcal{M}_c\ A'}{\langle \mathbf{co}(g,h), \langle (n+1), s'\rangle\rangle\ appr_c\ A'}}{\langle \mathbf{inck}, n, \mathbf{co}(g,h), s'\rangle\ app_c\ A'}$$

    The result follows since A and A' are correctly related by induction.

- $u_1 = \mathbf{dec}$: similar to the previous case.
- $u_1 = \langle \mathbf{cl}\ z, N_z, \rho_z \rangle$:

$$\frac{\langle N_z, \rho_z[z := new(z)], \kappa(g, h, \ldots, \top), s[new(z) := u_2] \rangle\ \mathcal{C}\ A}{\langle \langle \mathbf{cl}\ z, N_z, \rho_z \rangle, u_2, \kappa(g, h, \ldots, \top), s \rangle\ appk\ A}$$

$$\frac{\langle \mathcal{F}_{k_g}[\![N_z]\!], \rho_z[z := new(z)][k_g := new(k_g)], s''[new(z) := \delta(u_2)] \rangle\ \mathcal{M}_c\ A'}{\langle \langle \mathbf{cl}\ z k_g, \mathcal{F}_{k_g}[\![N_z]\!], \rho_z \rangle, \delta(u_2), \mathbf{co}(g, h), s' \rangle\ app_c\ A'}$$

where $s'' = s'[new(k_g) := \mathbf{co}(g, h)]$. The result follows since A and A' are correctly related by induction.

- $M = (\mathsf{let}\ (u_g\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ N_g)$:

$$\frac{u_0 = \phi(V_0, \rho_g, s) = 0 \quad \langle M_1, \rho_g, \kappa(g, h, \ldots, \top), s \rangle\ \mathcal{C}\ A}{\langle (\mathsf{let}\ (u_g\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ N_g), \rho_g, \kappa(h, \ldots, \top), s \rangle\ \mathcal{C}\ A}$$

$$\frac{u_0' = \delta(u_0) \quad \langle \mathcal{F}_{k_g}[\![M_1]\!], \rho_g[k_h := new(k_h)][k_g := new(k_g)], s'' \rangle\ \mathcal{M}_c\ A'}{\langle (\mathsf{let}\ k_g\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])\ (\mathsf{if0}\ \mathcal{V}[\![V_0]\!]\ \mathcal{F}_{k_g}[\![M_1]\!]\ \mathcal{F}_{k_g}[\![M_2]\!])), \rho_g[k_h := new(k_h)], s' \rangle\ \mathcal{M}_c\ A'}$$

where $s' = \delta(s)(q, \ldots, h, \ldots, \top)$ and $s'' = s'[new(k_g) := \mathbf{co}(g, h)]$. The result follows since A and A' are correctly related by induction. The case when the test is false is similar. ∎

B Proof of Theorem 5.5

Intuitively, the proof is the abstract version of the proof in the previous section. The crucial point is that no new location is created for each continuation. Thus, the locations $k_1, k_2, \ldots$ may refer to more than one continuation.

Theorem B.1 *Let $M \in \Lambda$, and let:*

$$\begin{array}{rcl} E_i & = & (\mathsf{let}\ (u_i\ [\ ])\ N_i) \\ \kappa(h, i, \ldots, j, \top) & = & E_h :: E_i :: \ldots :: E_j :: \mathbf{nil} \\ \mathbf{co}(i, j) & = & \langle \bot, \emptyset, \{\langle \mathbf{co}^{\ominus}\ u_i, \mathcal{F}_{k_j}[\![N_i]\!] \rangle\} \rangle \\ \sigma(h, i, \ldots, j, \top) & = & s[k_h := \mathbf{co}(h, i), \ldots, k_j := \mathbf{co}(j, \top), k_\top := \langle \bot, \emptyset, \{\mathbf{stop}\} \rangle] \end{array}$$

then:

$$\langle M, \kappa(h, \ldots, \top), \sigma \rangle\ \mathcal{C}^{\ominus}\ \langle u_1, \sigma_1 \rangle \quad \mathit{iff} \quad \langle \mathcal{F}_{k_h}[\![M]\!], \sigma' \rangle\ \mathcal{M}_c^{\ominus}\ \langle u_2, \sigma_2 \rangle$$

where $\delta^{\ominus}(\sigma)(h, \ldots, \top) \sqsubseteq \sigma'$, $\delta^{\ominus}(u_1) \sqsubseteq u_2$ and $\delta^{\ominus}(\sigma_1)(h, \ldots, \top) \sqsubseteq \sigma_2$.

Proof. The proof proceeds by induction on the height of the derivation of A_1. For the base case, M is a value V and the continuation is the initial one:

$$\frac{u = \phi^{\ominus}(V, \sigma) \quad \langle \mathbf{nil}, \langle u, \sigma \rangle \rangle\ appr^{\ominus}\ \langle u, \sigma \rangle}{\langle V, \kappa(\top), \sigma \rangle\ \mathcal{C}^{\ominus}\ \langle u, \sigma \rangle}$$

$$\frac{u' = \phi_c^{\ominus}(\mathcal{V}[\![V]\!], \sigma') \quad \dfrac{\ldots}{\langle (K_\top \sqcup \langle \bot, \emptyset, \{\mathbf{stop}\} \rangle), \langle u', \sigma' \rangle \rangle\ appr_c^{\ominus}\ \langle u' \sqcup u'', \sigma' \sqcup \sigma'' \rangle}}{\langle (k_\top\ \mathcal{V}[\![V]\!]), \sigma' \rangle\ \mathcal{M}_c^{\ominus}\ \langle u' \sqcup u'', \sigma' \sqcup \sigma'' \rangle}$$

The result follows since $\delta^{\ominus}(u) \sqsubseteq u'$.

For the inductive case, we proceed by cases of M:

- $M = V$:

$$\frac{u = \phi^{\ominus}(V,\sigma) \quad \dfrac{\langle N_h, \kappa(i,\ldots,\top), \sigma[u_h := \sigma(u_h) \sqcup u]\rangle\ \mathcal{C}^{\ominus}\ A}{\langle \kappa(h,i,\ldots,\top), \langle u,\sigma\rangle\rangle\ \mathit{appr}^{\ominus}\ A}}{\langle V, \kappa(h,i,\ldots,\top), \sigma\rangle\ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u' \sqsupseteq \delta^{\ominus}(u) \quad \dfrac{\ldots \quad \langle \mathcal{F}_{k_i}[\![N_h]\!], \sigma'[u_h := \sigma'(u_h) \sqcup u']\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle K_h \sqcup \mathbf{co}(h,i), \langle u',\sigma'\rangle\rangle\ \mathit{appr}_c^{\ominus}\ A' \sqcup A''}}{\langle (k_h\ \mathcal{V}[\![V]\!]), \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ A' \sqcup A''}$$

  The result follows by induction.

- $M = (\mathsf{let}\ (x\ V)\ N)$:

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

$$\frac{u' \sqsupseteq \delta^{\ominus}(u) \quad \langle \mathcal{F}_{k_h}[\![N]\!], \sigma'[x := \sigma'(x) \sqcup u']\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle (\mathsf{let}\ (x\ \mathcal{V}[\![V]\!])\ \mathcal{F}_{k_h}[\![N]\!]), \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ A'}$$

  The result follows by induction.

- $M = (\mathsf{let}\ (u_g\ (V_1\ V_2))\ N_g)$:

$$\frac{u_1 = \phi^{\ominus}(V_1,\sigma) \quad u_2 = \phi^{\ominus}(V_2,\sigma) \quad \langle u_1, u_2, \kappa(g,h,\ldots,\top), \sigma\rangle\ \mathit{appk}^{\ominus}\ A}{\langle (\mathsf{let}\ (u_g\ (V_1\ V_2))\ N_g), \kappa(h,\ldots,\top), \sigma\rangle\ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u_1' \sqsupseteq \delta^{\ominus}(u_1) \quad u_2' \sqsupseteq \delta^{\ominus}(u_2) \quad \langle u_1', u_2', \mathbf{co}(g,h), \sigma'\rangle\ \mathit{app}_c^{\ominus}\ A'}{\langle (\mathcal{V}[\![V_1]\!]\ \mathcal{V}[\![V_2]\!]\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])), \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ A'}$$

  Let the closure component of the abstract value u_1 be CL. Since $u_1' \sqsupseteq \delta^{\ominus}(u_1)$, the closure component of u_1' must at least include CL. Since the results of all the calls are combined, it suffices to establish the result for each closure $cl \in CL$ separately:

  - $cl = \mathbf{inc}$, $u_2 = \langle n, \ldots\rangle$:

$$\frac{\dfrac{\langle N_g, \kappa(h,\ldots,\top), \sigma[u_g := \sigma(u_g) \sqcup \langle \mathit{add1}^{\ominus}(n), \emptyset\rangle]\rangle\ \mathcal{C}^{\ominus}\ A}{\langle \kappa(g,h,\ldots,\top), \langle\langle \mathit{add1}^{\ominus}(n), \emptyset\rangle, \sigma\rangle\rangle\ \mathit{appr}^{\ominus}\ A}}{\langle \mathbf{inc}, u_2, \kappa(g,h,\ldots,\top), \sigma\rangle\ \mathit{appk}^{\ominus}\ A}$$

$$\frac{\dfrac{\langle \mathcal{F}_{k_h}[\![N_g]\!], \sigma'[u_g := \sigma'(u_g) \sqcup \langle \mathit{add1}^{\ominus}(n), \emptyset, \emptyset\rangle]\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle \mathbf{co}(g,h), \langle\langle \mathit{add1}^{\ominus}(n), \emptyset, \emptyset\rangle, \sigma'\rangle\rangle\ \mathit{appr}_c^{\ominus}\ A'}}{\langle \mathbf{inck}, u_2', \mathbf{co}(g,h), \sigma'\rangle\ \mathit{app}_c^{\ominus}\ A'}$$

    The result follows by induction.

  - $cl = \mathbf{dec}$: similar to the previous case.
  - $cl = \langle \mathbf{cl}^{\ominus}\ z, N_z\rangle$:

$$\frac{\langle N_z, \kappa(g,h,\ldots,\top), \sigma[z := \sigma(z) \sqcup u_2]\rangle\ \mathcal{C}^{\ominus}\ A}{\langle\langle \mathbf{cl}^{\ominus}\ z, N_z\rangle, u_2, \kappa(g,h,\ldots,\top), \sigma\rangle\ \mathit{appk}^{\ominus}\ A}$$

$$\frac{\langle \mathcal{F}_{k_g}[\![N_z]\!], \sigma'[z := \sigma'(z) \sqcup u_2', k_g := \sigma'(k_g) \sqcup \mathbf{co}(g,h)]\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle\langle \mathbf{cl}^{\ominus}\ zk_g, \mathcal{F}_{k_g}[\![N_z]\!]\rangle, u_2', \mathbf{co}(g,h), \sigma'\rangle\ \mathit{app}_c^{\ominus}\ A'}$$

    The result follows by induction.

* $M = (\mathsf{let}\ (u_g\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ N_g)$:

$$\frac{u_0 = \phi^{\ominus}(V_0, \sigma) = \langle 0, \emptyset \rangle \quad \langle M_1, \kappa(g, h, \ldots, \top), \sigma \rangle\ \mathcal{C}^{\ominus}\ A}{\langle (\mathsf{let}\ (u_g\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ N_g), \kappa(h, \ldots, \top), \sigma \rangle\ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u_0' = \langle 0, \emptyset, \emptyset \rangle \quad \langle \mathcal{F}_{k_g}[\![M_1]\!], \sigma'[k_g := \sigma'(k_g) \sqcup \mathbf{co}(g, h)] \rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle (\mathsf{let}\ k_g\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])\ (\mathsf{if0}\ \mathcal{V}[\![V_0]\!]\ \mathcal{F}_{k_g}[\![M_1]\!]\ \mathcal{F}_{k_g}[\![M_2]\!])), \sigma' \rangle\ \mathcal{M}_c^{\ominus}\ A'}$$

Since in general, $u_0' \sqsupseteq \delta^{\ominus}(u_0)$, the syntactic-CPS interpreter may also evaluate both branches. Both cases follow by induction. The cases when the semantic-CPS interpreter evaluates M_2 or when it evaluates both branches are similar. ∎

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] 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).