Program Transformations in a Denotational Setting

FLEMMING NIELSON
Aarhus University

Program transformations are frequently performed by optimizing compilers, and the correctness of applying them usually depends on data flow information. For language-to-same-language transformations, it is shown how a denotational setting can be useful for validating such program transformations.

Strong equivalence is obtained for transformations that exploit information from a class of forward data flow analyses, whereas only weak equivalence is obtained for transformations that exploit information from a class of backward data flow analyses. To obtain strong equivalence, both the original and the transformed program must be data flow analysed, but consideration of a transformation-exploiting liveness of variables indicates that a more satisfactory approach may be possible.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters, optimization, translator writing systems, compiler generators*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: Data flow analysis, abstract interpretation, program transformation

1. INTRODUCTION

In this paper we consider a class of program transformations, where a program is transformed into another in the same language (language-to-same-language transformations or source-to-source transformations). Such transformations are useful for "high-level optimization" in optimizing compilers (see e.g., [6]). The meaning of the transformed program must equal that of the original one. The two programs may differ in other respects, such as running time, but this will not be considered here, although it is generally such differences that motivate program transformations. The correctness of transforming a program may depend on data flow information. Even though this is frequently the case in practice, the literature contains, to our knowledge, no satisfactory framework for proving the

Author's current address: Institute of Electronic Systems, Aalborg University Center, Strandvejen 19, DK-9000 Aalborg C, Denmark.

correctness of such transformations. Here we address this problem in a denotational setting.

To give examples of program transformations, consider the following fragment of a program

$$\cdots y := 2 \cdots (\text{no } ys) \cdots x := y + (1 + 1) \cdots (\text{no } xs) \cdots x := 0 \cdots$$

One transformation is to replace $x := y + (1 + 1)$ by $x := y + 2$. It is easy to validate this transformation because the meaning of $x := y + (1 + 1)$ equals that of $x := y + 2$, so no data flow information is needed. Another transformation is to replace $x := y + (1 + 1)$ by $x := 4$ (constant folding [1]). This transformation is valid because the value of y immediately before $x := y + (1 + 1)$ is always 2, as can be determined by the forward data flow analysis called constant propagation [1]. It is not so easy to validate this transformation because the meanings of $x := y + (1 + 1)$ and $x := 4$ are not identical. A third transformation is to eliminate $x := y + (1 + 1)$ or (for technical reasons) replace it with a dummy statement. This transformation is valid because the value of x is not used until after x is assigned the value 0, as can be determined by the backward data flow analysis called live variables analysis [1]. The meanings of $x := y + (1 + 1)$ and a dummy statement are different, so this transformation is also not easy to validate.

Transformations that do not exploit data flow information (as replacing $x := y + (1 + 1)$ by $x := y + 2$) are considered in [5]. We consider transformations that exploit a class of forward data flow information (Section 3) and a class of backward data flow information (Section 4). In order to factor out the details of actual data flow analyses, we mostly consider abstract formulations of data flow information. In [9], it is shown how the ideas of [2] can be used to relate a large class of forward data flow analyses to the formulation used here. We sketch how a similar connection may be possible for backward data flow analyses. The framework for validating program transformations is compared to that of [4] and is claimed to be better. Section 5 contains the conclusions.

2. PRELIMINARIES

In defining semantic equations we use the notation of [7] and [11], but the domains are cpo's (as in [8]), rather than complete lattices. Below we explain some fundamental notions and nonstandard notation ($\gg$, $==$, $-t>$, $-c>$).

A *partially ordered* set $(S, \sqsubseteq)$ is a set S with partial order $\sqsubseteq$ (i.e., $\sqsubseteq$ is a reflexive, antisymmetric, and transitive relation on S). For $S' \subseteq S$, there may exist a (necessarily unique) *least upper bound* $\sqcup S'$ in S such that $\forall s \in S$: $(s \sqsupseteq \sqcup S' \Leftrightarrow \forall s' \in S'$: $s \sqsupseteq s')$. When $S' = \{s_1, s_2\}$, one often writes $s_1 \sqcup s_2$ instead of $\sqcup S'$. A nonempty subset $S' \subseteq S$ is a *chain* if S' is countable and $s_1, s_2 \in S' \Rightarrow (s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1)$. An element $s \in S$ is *maximal* if $\forall s' \in S$: $(s' \sqsupseteq s \Rightarrow s' = s)$, and it is *least* if $\forall s' \in S$: $s' \sqsupseteq s$. A partially ordered set is a *cpo* if it has a least element ($\perp$) and any chain has a least upper bound. The word *domain* is used for cpo's, and elements of some domain S are denoted s, s', s_1, and so on. A domain is *flat* if any chain contains at most two elements, and it is of *finite height* if any chain is finite.

Domains N, Q, and T are flat domains of natural numbers, quotations, and truth values. From domains $S_1, \ldots, S_n$ one can construct the *separated sum*

$S_1 + \cdots + S_n$. This is a domain with a new least element and injection functions in S_i, enquiry functions $\mathsf{E}\ S_i$, and projection functions $|\ S_i$. The *cartesian product* $S_1 \times \cdots \times S_n$ is a domain with selection functions $\downarrow i$. The domain S^* of *lists* is $\{\langle\ \rangle\} + S + (S \times S) + \ldots$. Function # yields the length of a list, function $\dagger i$ removes the first i elements, and § concatenates lists. By $P(S)$ is meant the *power set* of S with set inclusion as partial order. Sometimes a set is regarded as a partially ordered set whose partial order is equality.

All functions are assumed to be total. For partially ordered sets S and S', the set of (total) functions from S to S' is denoted $S - t > S'$. A function $f \in S - t > S'$ is *continuous* if $f(\sqcup S'') = \sqcup \{f(s) \mid s \in S''\}$ holds for any chain $S'' \subseteq S$ whose least upper bound exists. The set of continuous functions from S to S' is denoted by $S - c > S'$. A function $f \in S - t > S'$ is *additive* (a *complete*-$\sqcup$-*morphism*) if $f(\sqcup S'') = \sqcup\{f(s) \mid s \in S''\}$ for any subset $S'' \subseteq S$ whose least upper bound exists. Both $S - t > S'$ and $S - c > S'$ are partially ordered by $f_1 \sqsubseteq f_2 \Leftrightarrow \forall s \in S: f_1(s) \sqsubseteq f_2(s)$. If S' is a domain, the same holds for $S - t > S'$ and $S - c > S'$.

An element $s \in S$ is a *fixed point* of $f \in S - t > S$ if $f(s) = s$. When S is partially ordered, s is the *least fixed point* provided it is a fixed point and $s' = f(s') \Rightarrow s' \sqsupseteq s$. If S is a domain and $f \in S - c > S$, the least fixed point always exists and is given by $\mathrm{FIX}(f) = \sqcup\{f^n(\perp) \mid n \geq 0\}$. We frequently write $\sqcup_{n=0}^{\infty} f^n(\perp)$ instead of $\sqcup\{f^n(\perp) \mid n \geq 0\}$.

For any domain S, we use the symbol $==$ as a continuous equality predicate $(S \times S - c > T)$, whereas $=$ is reserved for true equality. So (true $== \perp$) will be $\perp$, whereas (true $= \perp$) is false. When S is of finite height, it is assumed that $s_1 == s_2$ is $\perp$ if one of s_1, s_2 is nonmaximal, and equals $s_1 = s_2$ otherwise. We write $\gg$ for the continuous extension of $>$ (the predicate "greater than or equal to" on the integers). The conditional $t \rightarrow s_1, s_2$ is s_1, s_2 or $\perp$, depending on whether t is true, false, or $\perp$. By $f[y/x]$ is meant $\lambda z.z == x \rightarrow y, f(z)$.

3. PROGRAM TRANSFORMATIONS AND FORWARD DATA FLOW ANALYSES

In this section we show how to validate program transformations that exploit information from a class of data flow analyses. First, we define a toy language. Then we give an abstract way of specifying forward data flow information by means of a collecting semantics. Finally, we consider program transformations.

Toy Language

The toy language consists of commands (syntactic category Cmd) and expressions (Exp). It is convenient to let Syn be the union of Cmd and Exp. The syntax of commands and expressions is

```
cmd :: = cmd₁; cmd₂ | ide := exp | IF exp THEN cmd₁ ELSE cmd₂ FI
       | WHILE exp DO cmd OD | WRITE exp | READ ide
exp  :: = exp₁ ope exp₂ | ide | bas
```

We do not specify the syntax of identifiers (Ide), basic values (Bas), and operators (Ope). The semantics is given by Tables I and II, and is explained below. Table II defines some domains and auxiliary functions as well as an associative

Table I. Semantic Function

$$
\begin{array}{l}
T \in \mathrm{Syn} - c > G \\
T[[\ \mathrm{cmd}_1;\ \mathrm{cmd}_2]] = T[[\ \mathrm{cmd}_1\ ]] * [[\ \mathrm{cmd}_2\ ]] \\
T[[\ \mathrm{ide} := \mathrm{exp}\ ]] = T[[\ \mathrm{exp}\ ]] * \mathrm{assign}[[\ \mathrm{ide}]] \\
T[[\ \mathrm{IF\ exp\ THEN\ cmd}_1\ \mathrm{ELSE\ cmd}_2\ \mathrm{FI}]] = \\
\qquad\qquad T[[\ \mathrm{exp}]] * \mathrm{cond}(T[[\ \mathrm{cmd}_1]],\ T[[\ \mathrm{cmd}_2]]) \\
T[[\ \mathrm{WHILE\ exp\ DO\ cmd\ OD}]] = \\
\qquad\qquad \mathrm{FIX}(\lambda g.\ T[[\ \mathrm{exp}]] * \mathrm{cond}(T[[\ \mathrm{cmd}]] * g,\ \lambda \mathrm{sta.sta\ inR})\ ) \\
T[[\ \mathrm{WRITE\ exp}]] = T[[\ \mathrm{exp}]] * \mathrm{write} \\
T[[\ \mathrm{READ\ ide}]] = \mathrm{read} * \mathrm{assign}[[\ \mathrm{ide}]] \\
T[[\ \mathrm{exp}_1\ \mathrm{ope}\ \mathrm{exp}_2]] = T[[\ \mathrm{exp}_1]] * T[[\ \mathrm{exp}_2]] * \mathrm{apply}[[\ \mathrm{ope}]] \\
T[[\ \mathrm{ide}]] = \mathrm{content}[[\ \mathrm{ide}]] \\
T[[\ \mathrm{bas}]] = \mathrm{push}[[\ \mathrm{bas}]]
\end{array}
$$

combinator (*) used for sequencing. The semantic functions O for operators and B for basic values have been left unspecified, just as the corresponding syntactic categories. Table I defines a single semantic function T that ascribes meaning to both commands and expressions. It simplifies some notation to be used later in that only one semantic function is used. The semantic function is in direct style because continuations are not needed for the development.

A state (element of Sta) consists of an environment, current input and output, and a stack of temporary results. The presence of the stack of temporary results (stack of witnessed values [7]) indicates that the semantics is a store semantics [7, 11]. The stack is used to hold the values of subexpressions during the evaluation of expressions. The functions apply[[ope]], content[[ide]], and assign[[ide]] illustrate how this is done. As an example, consider the definition of apply[[ope]]. The function Vapply[[ope]] $\in$ Sta $- c >$ T verifies whether the argument state is of a special form. Only if this is the case, will the state be transformed as described by Bapply[[ope]] $\in$ Sta $- c >$ Sta (B for "body"). The definitions of read, write, and push[[bas]] are similar, and the reader should have little trouble in supplying the definitions (they are along the lines of [7]).

Example. Consider the program WRITE 2 + 3. Define the states

$$
\begin{array}{l}
\mathrm{sta} \ = \langle \mathrm{env}, \langle\,\rangle, \langle\,\rangle, \langle\,\rangle\rangle \\
\mathrm{sta}' = \langle \mathrm{env}, \langle\,\rangle, \langle 5\mathrm{inVal}\rangle, \langle\,\rangle\rangle
\end{array}
$$

where, for instance, env = λide. "nil"inVal. We then have

$$\mathrm{T}[[2+3]](\mathrm{sta}) = \langle \mathrm{env}, \langle\,\rangle, \langle\,\rangle, \langle 5\mathrm{inVal}\rangle\rangle\ \mathrm{inR}$$

so that

$$T[[\mathrm{WRITE}\ 2+3]]\ (\mathrm{sta}) = \mathrm{sta}'\mathrm{inR}$$

The notion of a store semantics was defined in [7] as a more implementation-oriented semantics than the more usual standard semantics [7, 11]. In the store semantics there is virtually no difference between the behavior of commands and expressions, and this is convenient when defining the collecting semantics below (as is further discussed in [9]). In the standard semantics, commands and

Table II. Store Semantics

Domains

$G = \text{Sta} - c > R$

$R = \text{Sta} + \{\text{“error”}\}$

$\text{Sta} = \text{Env} \times \text{Inp} \times \text{Out} \times \text{Tem}$	states
$\text{Env} = \text{Ide} - c > \text{Val}$	environments
$\text{Inp} = \text{Val}^*$	inputs
$\text{Out} = \text{Val}^*$	outputs
$\text{Tem} = \text{Val}^*$	temporary result stacks
$\text{Val} = T + N + \cdots + \{\text{“}nil\text{”}\}$	values

Combinator

$* \in G \times G - c > G$

$g_1 * g_2 = \lambda \text{sta}.g_1(\text{sta}) \mathrel{\mathsf{E}} \text{Sta} \longrightarrow g_2(g_1(\text{sta}) \mid \text{Sta}), g_1(\text{sta})$

Functions

$\text{cond} \in G \times G - c > G$

$$\text{cond}(g_1, g_2) = \lambda \text{sta}.\text{Vcond}(\text{sta}) \longrightarrow (\text{Scond}(\text{sta}) \longrightarrow g_1, g_2)(\text{Bcond}(\text{sta})),\ \text{“error” inR}$$

$\text{Vcond}\langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \#\text{tem} \gg 1 \longrightarrow \text{tem} \downarrow 1 \mathrel{\mathsf{E}} T, \text{false}$

$\text{Bcond}\langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \langle \text{env}, \text{inp}, \text{out}, \text{tem} \dagger 1\rangle$

$\text{Scond}\langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \text{tem} \downarrow 1 \mid T$

$\text{apply}[[\text{ ope}]] \in G$

$$\text{apply}[[\text{ ope}]] = \lambda \text{sta}.\text{Vapply}[[\text{ ope}]]\ (\text{sta}) \longrightarrow \text{Bapply}[[\text{ ope}]]\ (\text{sta}) \text{ inR},\ \text{“error” inR}$$

$\text{Vapply}[[\text{ ope}]]\ \langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \#\text{tem} \gg 2$

$$\text{Bapply}[[\text{ ope}]]\ \langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \langle \text{env}, \text{inp}, \text{out}, \langle\ O[[\text{ ope}]]\ \langle \text{tem} \downarrow 2, \text{tem} \downarrow 1\rangle\ \rangle\ \S(\text{tem}\dagger 2)\rangle$$

$\text{assign}[[\text{ ide}]] \in G$

$$\text{assign}[[\text{ ide}]] = \lambda \text{sta}.\text{Vassign}[[\text{ ide}]]\ (\text{sta}) \longrightarrow \text{Bassign}[[\text{ ide}]]\ (\text{sta}) \text{ inR},\ \text{“error” inR}$$

$\text{Vassign}[[\text{ ide }]]\ \langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \#\text{tem} \gg 1$

$$\text{Bassign}[[\text{ ide}]]\ \langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \langle \text{env}[\text{tem} \downarrow 1/\text{ide}], \text{inp}, \text{out}, \text{tem} \dagger 1\rangle$$

$\text{content}\ [[\text{ ide}]] \in G$

$$\text{content}[[\text{ ide}]] = \lambda \text{sta}.\text{Vcontent}[[\text{ ide}]]\ (\text{sta}) \longrightarrow \text{Bcontent}[[\text{ ide}]](\text{sta}) \text{ inR},\ \text{“error” inR}$$

$\text{Vcontent}[[\text{ ide}]]\ \langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \text{true}$

$$\text{Bcontent}[[\text{ ide}]]\ \langle \text{env}, \text{inp}, \text{out}, \text{tem}\rangle = \langle \text{env}, \text{inp}, \text{out}, \langle \text{env}[[\text{ ide}]]\ \rangle\ \S\text{tem}\rangle$$

$\text{push}[[\text{ bas}]] \in G$, $\text{read} \in G$, $\text{write} \in G$ are defined similarly

expressions behave rather differently because there is no stack of temporary results. (See, e.g., an arbitrary semantics in [11] for an example.) It is well known how to transform a standard semantics into a store semantics [7], and this is why we have chosen the store semantics as our starting place.

Finally, the following lemma is needed in later proofs. It says that the iterates in a WHILE loop either give no information or give full information.

LEMMA 1. *Let* $g[\bar{g}] = T[[exp]] * cond(T[[cmd]] * \bar{g}, \lambda sta.sta\ inR)$.[1] *Then* $(\lambda\bar{g}.g[\bar{g}])^n\bot$ *sta is either* $\bot$ *or* $T[[$ *WHILE exp DO cmd OD*$]]$ *sta.*

PROOF. It suffices to show that $\forall$sta: $[(\lambda\bar{g}.g[\bar{g}])^n\bot \text{ sta} \neq \bot \Rightarrow \forall g_0\text{: } (\lambda\bar{g}.g[\bar{g}])^n g_0$ sta $= (\lambda\bar{g}.g[\bar{g}])^n\bot$ sta]. This is because $(\lambda\bar{g}.g[\bar{g}])^{n+1}\bot = (\lambda\bar{g}.g[\bar{g}])^n g_0$ when $g_0 = g[\bot]$. The proof is by induction on n and, since the case $n = 0$ is obvious, consider the inductive step. It is easy to see that $(\lambda g.g[\bar{g}])^{n+1}g_0$ sta independently of g_0 is $\bot$, "error" inR, sta′ inR, or $(\lambda\bar{g}.g[\bar{g}])^n g_0$ sta″ where sta′ and sta″ are independent of g_0. In the first three cases the result is immediate and in the last case it follows by the induction hypothesis. □

Collecting Semantics

We now define a collecting semantics [9] that gives an abstract way of specifying (some types of) forward data flow information. Like the store semantics, the collecting semantics executes the program for one particular initial state (e.g., sta = ⟨λide."nil" inVal, inp, ⟨ ⟩, ⟨ ⟩⟩ for some input inp ∈ Inp). Instead of specifying the result of this execution, the purpose of the collecting semantics is to associate with each program point the *set of states* in which control can be when that point is reached. The data flow information specified by the collecting semantics is in a rather abstract form that is suitable for the subsequent development. In practice, more approximate data flow analyses will be used (to assure computability), and [9] uses the ideas of [2] to relate approximate analyses to the collecting semantics. This is done by formulating an induced semantics (specified by a pair of adjoined [2] or semiadjoined [9] functions) that executes the program on an (approximate) description of a set of states. The data flow analysis "constant propagation" can be specified this way.

We identify a program with a parse tree. The usual way to address nodes in such a tree is by lists of integers called occurrences. In our formulation this means a member of Occ = N^*. The root has occurrence ⟨ ⟩ and the ith son (counting from the left) of a node with occurrence occ has occurrence occ§⟨i⟩. We then represent a program point by a tuple ⟨occ, q⟩ ∈ Pla = Occ × Q. The tuple ⟨occ, "L"⟩ represents the point immediately before the syntactic construct pointed to by occ. Similarly, ⟨occ, "R"⟩ represents the point immediately after. This is illustrated in Figure 1. Note that Pla contains elements that cannot reasonably be viewed as representing program points: only the maximal elements (in the sense of Section 2) can be viewed in this way. This is just a facet of the common situation in denotational semantics that domains intuitively contain too many elements.

The occurrence associated with a node in a parse tree is not part of the node itself, so to be able to "mention" program points in the semantic equations we supply the semantic function with an occurrence as an additional parameter. Furthermore, the semantic equations are augmented with functions like attach⟨occ, "L"⟩ that are to associate information with the program points mentioned. Table III sketches the result of performing these changes.

[1] For typographical reasons, we write $g[\bar{g}]$ instead of $g_{\bar{g}}$, so $g[\bar{g}] \in G$ for any fixed $\bar{g} \in G$.

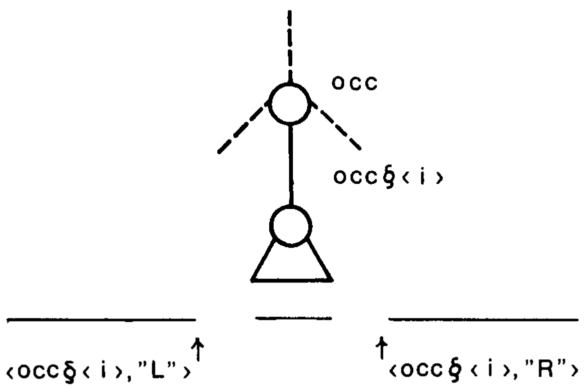


Figure 1.

Table III. Modified Semantic Function

```
$T \in$ Syn $- c >$ Occ $- c > G$
$T$[[ IF exp THEN cmd$_1$ ELSE cmd$_2$ FI]] occ =
  attach ⟨occ, "$L$"⟩ *
  $T$[[ exp]] occ§⟨1⟩ *
  cond($T$[[ cmd$_1$]] occ§⟨2⟩ * attach ⟨occ, "$R$"⟩
    , $T$[[ cmd$_2$]] occ§⟨3⟩ * attach ⟨occ, "$R$"⟩)
$T$[[ WHILE exp DO cmd OD]] occ =
  attach ⟨occ, "$L$"⟩ *
  FIX($\lambda g.T$[[ exp]] occ§⟨1⟩ *
    cond($T$[[ cmd]] occ§⟨2⟩ * $g$
      , attach ⟨occ, "$R$"⟩))
$T$[[ exp$_1$ ope exp$_2$]] occ =
  attach ⟨occ, "$L$"⟩ *
  $T$[[ exp$_1$]] occ§⟨1⟩ *
  $T$[[ exp$_2$]] occ§⟨3⟩ *
  apply[[ ope]] *
  attach ⟨occ, "$R$"⟩
remaining clauses changed similarly to the one for exp$_1$ ope, exp$_2$,
```

The collecting semantics is specified by Tables III and IV. (It is similar to the one in [9], except that continuations have been removed.) Domain A = Pla $- c > P$(Sta) is used to associate each program point with the set of those states that control can be in when that point is reached. The associative combinator * is continuous in its right argument (but not the left [9]), so FIX (in Table III) is applied to continuous functions only. To distinguish between the collecting semantics and the store semantics, we use suffixes col and sto, so, for example, Tcol is the semantic function of the collecting semantics.

Table IV. Collecting Semantics

Domains

$G = \text{Sta} - t > (R \times A)$
$R = \text{Sta} + \{\text{"error"}\}$
$A = \text{Pla} - c > P(\text{Sta})$

$\text{Occ} = N^*$	occurrences
$\text{Pla} = \text{Occ} \times Q$	places

remaining domains as in Table II.

Combinator

$* \in G \times G - t > G$ (continuous in second argument)

$$g_1 * g_2 = \lambda\text{sta}.\langle g_1(\text{sta}) \downarrow 1 \text{ E Sta} \rightarrow g_2(g_1(\text{sta}) \downarrow 1 \mid \text{Sta}) \downarrow 1, g_1(\text{sta}) \downarrow 1,$$
$$[g_1(\text{sta}) \downarrow 1 \text{ E Sta} \rightarrow g_2(g_1(\text{sta}) \downarrow 1 \mid \text{Sta}) \downarrow 2, \bot] \sqcup g_1(\text{sta}) \downarrow 2\rangle$$

Functions

$\text{attach} \in \text{Pla} - c > G$
$\text{attach (pla)} = \lambda\text{sta}.\langle \text{sta inR}, \bot[\{\text{sta}\}/\text{pla}]\rangle$

$\text{cond} \in G \times G - c > G$
$\text{cond } (g_1, g_2) = \lambda\text{sta.Vcond(sta)} \rightarrow$
$\quad (\text{Scond(sta)} \rightarrow g_1, g_2)\ (\text{Bcond(sta)}),$
$\quad \langle \text{"error" inR}, \bot\rangle$

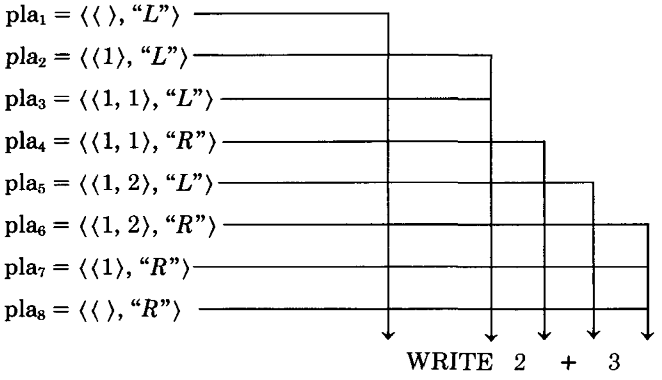
Vcond, Scond, Bcond as in Table II.

$\text{apply}[[\text{ ope}]] \in G$
$\text{apply}[[\text{ ope}]] = \lambda\text{sta}.\langle \text{Vapply}[[\text{ ope}]]\ (\text{sta}) \rightarrow (\text{Bapply}[[\text{ ope}]]\ (\text{sta}))\ \text{inR},$
$\quad \text{"error" inR}$
$\quad , \bot\rangle$

Vapply[[ope]], Bapply[[ope]] as in Table II.

assign[[ide]], content[[ide]], push[[bas]], read, write are defined similarly to apply[[ope]].

Example. Considering the program WRITE 2 + 3, we may identify the following program points which "arise" in its collecting semantics:


$\text{pla}_1 = \langle\langle\ \rangle, \text{"}L\text{"}\rangle$
$\text{pla}_2 = \langle\langle 1\rangle, \text{"}L\text{"}\rangle$
$\text{pla}_3 = \langle\langle 1, 1\rangle, \text{"}L\text{"}\rangle$
$\text{pla}_4 = \langle\langle 1, 1\rangle, \text{"}R\text{"}\rangle$
$\text{pla}_5 = \langle\langle 1, 2\rangle, \text{"}L\text{"}\rangle$
$\text{pla}_6 = \langle\langle 1, 2\rangle, \text{"}R\text{"}\rangle$
$\text{pla}_7 = \langle\langle 1\rangle, \text{"}R\text{"}\rangle$
$\text{pla}_8 = \langle\langle\ \rangle, \text{"}R\text{"}\rangle$
WRITE 2 + 3

The plentitude of program points is due to the very systematic nature in which the attach functions have been placed; this is technically convenient for the development, but would be remedied in practical applications.

Turning to the collecting semantics, we have

$T\mathrm{col}[[\text{ WRITE } 2 + 3]]\langle\ \rangle\mathrm{sta} = \langle \mathrm{sta}'\mathrm{inR}, a\rangle$

where sta′ is as in the previous example, and, for example:

$a(\mathrm{pla}_1) = \{\mathrm{sta}\}$

$a(\mathrm{pla}_4) = \{\langle \mathrm{env}, \langle\ \rangle, \langle\ \rangle, \langle 2\mathrm{inVal}\rangle\rangle\}$

$a(\mathrm{pla}_7) = \{\langle \mathrm{env}, \langle\ \rangle, \langle\ \rangle, \langle 5\mathrm{inVal}\rangle\rangle\}$

$a(\mathrm{pla}_8) = \{\mathrm{sta}'\}$

Note that $a(\mathrm{pla}_7)$ is not equal to $a(\mathrm{pla}_8)$, even though pla_7 is "very close" to pla_8. We return to this below.

The collecting semantics cannot be proved correct with respect to the store semantics because two programs that look different (and to which different data flow information pertains) may have the same meaning in the store semantics. A partial relationship between the collecting semantics and the store semantics is given by the following property, which says, intuitively, that the store semantics is embedded in the collecting semantics.

PROPERTY Ca. *Let syn ∈ Syn, occ ∈ Occ be maximal and sta ∈ Sta. Then Tcol[[syn]] occ sta ↓1 = Tsto[[syn]] sta.*

PROOF OF PROPERTY Ca. By straightforward structural induction, and is omitted. □

Program Transformations

We now consider how to validate program transformations such as the one mentioned in the Introduction when $x := y + (1 + 1)$ was replaced by $x := 4$. This is achieved by Theorem 1 below. To specify program transformations we need some operations upon parse trees. Rather than giving formal definitions using concepts from tree replacement systems [10], we give informal explanations. Let "occ *points into* syn" mean that there is a node in syn that has occurrence occ and is of syntactic category Cmd or Exp. In that case, "syn *at* occ" denotes the subtree of syn with that node as the root. Let occ point into syn and suppose syn *at* occ and syn′ belong to the same syntactic category. Then syn [occ ← syn′] denotes the parse tree that is syn with syn *at* occ replaced by syn′.

We also need some notation to state properties of the collecting semantics. Let "pla *is a descendant* of occ" mean that pla↓1 equals occ§occ′ for some maximal occ′ and that pla↓2 ∈ {"L", "R"}. This means that the program point pla is in the subprogram pointed to by occ. Define the additive function "filter" from $P(\mathrm{Sta} + \{\text{"error"}\})$ to $P(\mathrm{Sta})$ by

$\mathrm{filter}(R) = \{(r \mid \mathrm{Sta}) \mid r \in R \wedge (r \mathrel{\mathrm{E}} \mathrm{Sta}) = \mathrm{true}\}$

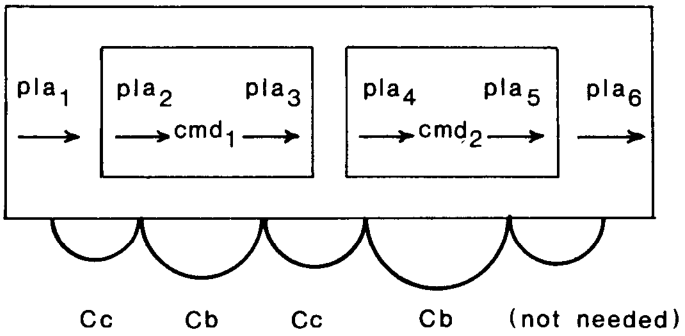


Figure 2. pla_1 = ⟨occ, "L"⟩, pla_2 = ⟨occ§⟨1⟩, "L"⟩, pla_3 = ⟨occ§⟨1⟩, "R"⟩, pla_4 = ⟨occ§⟨2⟩, "L"⟩, pla_5 = ⟨occ§⟨2⟩, "R"⟩, pla_6 = ⟨occ, "R"⟩.

The purpose is to single out exactly those results (in the set R) that give rise to an actual computation in the remainder of the program. Furthermore, abbreviate

```
condtrue = λsta.⟨ Vcond(sta) →
                  Scond(sta) → (Bcond(sta))inR, "error"inR
                             , "error"inR
                ,⊥⟩
condfalse = λsta.⟨ Vcond(sta) →
                  Scond(sta) → "error"inR, (Bcond(sta))inR
                             , "error"inR
                ,⊥⟩
```

The proof of Theorem 1 uses properties Ca, Cb, Cc, and Cd. Property Cb relates data flow information for program points on each side of a syntactic subphrase. Property Cc relates adjacent program points (e.g., ⟨occ, "L"⟩ and ⟨occ§⟨1⟩, "L"⟩ which often denote the same program point). It is stated by cases of the syntactic construct. For the construct cmd_1; cmd_2, the properties Cb and Cc are sketched in Figure 2 (arrows correspond to places and rectangles to syntactic constructs). Property Cd is used in the proof of properties Cb and Cc. Among other things it says that subphrases can only supply data flow information for program points contained in them.

PROPERTY Cb. *Let syn ∈ Syn, occ ∈ Occ be maximal, sta ∈ Sta and occ′ ∈ Occ point into syn and abbreviate a–col = Tcol*[[*syn*]] *occ sta*$\downarrow$2. *Then, a–col*⟨*occ*§*occ*′, "*R* "⟩ =

filter {*Tcol*[[*syn at occ*′]] <> *sta*′$\downarrow$1 | *sta*′ ∈ *a-col*⟨*occ*§*occ*′, "*L*"⟩}

Given that Ca holds, one may reformulate property Cb as stating

a-col⟨occ§occ′, "R"⟩ =
  filter{Tsto[[syn *at* occ′]](sta′) | sta′ ∈ *a*-col⟨occ§occ′, "L"⟩}

PROPERTY Cc. *Let syn ∈ Syn, occ ∈ Occ be maximal, sta ∈ Sta and occ′ ∈ Occ point into syn and abbreviate a-col = Tcol*[[*syn*]] *occ sta*$\downarrow$2.

If syn *at* occ′ is exp_1 ope exp_2
then a-col⟨occ§occ′§⟨1⟩, "L"⟩ = a-col⟨occ§occ′, "L"⟩
a-col⟨occ§occ′§⟨3⟩, "L"⟩ = a-col⟨occ§occ′§⟨1⟩, "R"⟩
If syn *at* occ′ is IF exp THEN cmd_1 ELSE cmd_2 FI
then a-col⟨occ§occ′§⟨1⟩, "L"⟩ = a-col⟨occ§occ′, "L"⟩
a-col⟨occ§occ′§⟨2⟩, "L"⟩ = filter {condtrue(sta′)↓1 | sta′ ∈ a-col⟨occ§occ′§⟨1⟩, "R"⟩}
a-col⟨occ§occ′§⟨3⟩, "L"⟩ = filter {condfalse(sta′)↓1 | sta′ ∈ a-col⟨occ§occ′§⟨1⟩, "R"⟩}
If syn *at* occ′ is WHILE exp DO cmd OD
then a-col⟨occ§occ′§⟨1⟩, "L"⟩ = a-col⟨occ§occ′, "L"⟩ ∪ a-col⟨occ§occ′§⟨2⟩, "R"⟩
a-col⟨occ§occ′§⟨2⟩, "L"⟩ = filter {condtrue(sta′)↓1 | sta′ ∈ a-col⟨occ§occ′§⟨1⟩, "R"⟩}

For the remaining constructs there are properties "similar" to the one for exp_1 ope exp_2.

PROPERTY Cd. *Let syn ∈ Syn, occ ∈ Occ be maximal, sta ∈ Sta and pla ∈ Pla. Then*

(*i*) *Tcol*[[*syn*]] *occ sta* ↓2 *pla* ≠ ∅ ⟹ *pla is a descendant of occ*
(*ii*) *Tcol*[[*syn*]] *occ sta* ↓2 ⟨*occ*, "*L*"⟩ = {*sta*}
(*iii*) *Tcol*[[*syn*]] *occ sta* ↓2 ⟨*occ*, "*R*"⟩ = *filter* {*Tcol*[[*syn*]] *occ sta* ↓1}

It is possible to prove property Cd first and then Cb, Cc in either order, but it is easier to prove the three properties together.

Proof of Cb, Cc, and Cd. The proof is by structural induction, and we omit the suffix col. It is convenient to be able to view the semantic functions in the collecting semantics as operating upon elements of R = Sta + {"error"}, rather than just elements of Sta. To facilitate this, define the combinator

$$\Delta \in G \times R - t > R \times A$$

by

$$g \,\Delta\, r = r \mathrel{\mathsf{E}} \text{Sta} \rightarrow g(r \mid \text{Sta}),\ \langle r, \bot\rangle.$$

Then $g(\text{sta}) = g\ \Delta\ (\text{sta inR})$ and $(g_1 * g_2)\ \Delta\ r = \langle g_2\ \Delta\ (g_1\ \Delta\ r \downarrow 1) \downarrow 1, g_2\ \Delta\ (g_1\ \Delta\ r \downarrow 1) \downarrow 2 \sqcup g_1\ \Delta\ r \downarrow 2\rangle$, as well as $\text{cond}(g_1, g_2)\ \Delta\ r \downarrow 2 = g_1\ \Delta\ (\text{condtrue}\ \Delta\ r \downarrow 1) \downarrow 2 \sqcup g_2\ \Delta\ (\text{condfalse}\ \Delta\ r \downarrow 1) \downarrow 2$.

For the structural induction, we only consider the case where syn is WHILE exp DO cmd OD. Abbreviate

$g[\bar{g}] = T$[[exp]] occ§⟨1⟩ * cond(T[[cmd]] occ§⟨2⟩ * $\bar{g}$, attach ⟨occ, "R"⟩)

corresponding to the body used in the fixed point (in the semantics of WHILE),

iter = T[[exp]] occ§⟨1⟩ * condtrue * T[[cmd]] occ§⟨2⟩

corresponding to executions where the expression evaluates to true,

$\text{iter}^{*^0} = \lambda\text{sta}.\langle\text{sta inR}, \bot\rangle$
$\text{iter}^{*^{(n+1)}} = \text{iter} * \text{iter}^{*^n} = \text{iter}^{*^n} * \text{iter}$ (by * associative)

Furthermore, let

$$g_1 = \lambda\text{sta}.T[[\text{ exp}]]\ \text{occ}\S\langle 1\rangle\ \text{sta}$$
$$g_2 = \lambda\text{sta}.T[[\text{ cmd}]]\ \text{occ}\S\langle 2\rangle\ \Delta\ (\text{condtrue}\ \Delta\ (T[[\text{ exp}]]\ \text{occ}\S\langle 1\rangle\ \text{sta} \downarrow 1) \downarrow 1)$$

so that iter $= g_1 \sqcup g_2$.

Our first task is to show Cd. For this, let pla be different from $\langle$occ, "R"$\rangle$, and calculate

$$\begin{aligned}(\lambda\bar{g}.g[\bar{g}])^{n+1} \perp (\text{sta}) \downarrow 2\ (\text{pla}) &= g_1(\text{sta}) \downarrow 2\ (\text{pla}) \sqcup g_2(\text{sta}) \downarrow 2\ (\text{pla})\\ &\quad \sqcup (\lambda\bar{g}.g[\bar{g}])^{n} \perp \Delta\ (\text{iter}(\text{sta}) \downarrow 1) \downarrow 2(\text{pla})\end{aligned}$$

Induction on n may be used to show that this may be rewritten to

$$\begin{aligned}&(\lambda\bar{g}.g[\bar{g}])^{n+1} \perp (\text{sta}) \downarrow 2\ (\text{pla})\\ &\quad = \textstyle\bigsqcup_{m=0}^{n} \bigsqcup_{i=1}^{2} g_i\ \Delta\ (\text{iter}^{*m}(\text{sta}) \downarrow 1) \downarrow 2(\text{pla})\end{aligned}$$

Hence,

$$\begin{aligned}&T[[\text{ WHILE exp DO cmd OD}]]\ \text{occ sta} \downarrow 2(\text{pla})\\ &= \text{attach}\langle\text{occ}, \text{“}L\text{”}\rangle\ \text{sta} \downarrow 2(\text{pla})\\ &\quad \sqcup \textstyle\bigsqcup_{n=0}^{\infty} g_1\ \Delta\ (\mathit{iter}^{*n}(\text{sta}) \downarrow 1) \downarrow 2(\text{pla})\\ &\quad \sqcup \textstyle\bigsqcup_{n=0}^{\infty} g_2\ \Delta\ (\text{iter}^{*n}(\text{sta}) \downarrow 1) \downarrow 2(\text{pla})\end{aligned}$$

Then, Cd(i) and Cd(ii) are immediate. For Cd(iii), we have (the steps are justified below):

$$\begin{aligned}&T[[\text{ WHILE exp DO cmd OD}]]\ \text{occ sta} \downarrow 2\ \langle\text{occ}, \text{“}R\text{”}\rangle\\ &= \textstyle\bigsqcup_{n=0}^{\infty} ((\lambda\bar{g}.g[\bar{g}])^{n} \perp (\text{sta}) \downarrow 2\ \langle\text{occ}, \text{“}R\text{”}\rangle)\\ &= \textstyle\bigcup_{n=0}^{n} \text{filter}\ \{(\lambda\bar{g}.g[\bar{g}])^{n} \perp (\text{sta}) \downarrow 1\}\\ &= \text{filter}\ \{T[[\text{ WHILE exp DO cmd OD}]]\ \text{occ sta} \downarrow 1\}\end{aligned}$$

The second steps follows because

$$\forall\text{sta}: [(\lambda\bar{g}.g[\bar{g}])^{n} \perp \text{sta} \downarrow 2\ \langle\text{occ}, \text{“}R\rangle = \text{filter}\ \{(\lambda\bar{g}.g[\bar{g}])^{n} \perp \text{sta} \downarrow 1)\}]$$

as can be shown by induction on n. The third step follows because filter($\{\perp\}$) = $\varnothing$ and $(\lambda\bar{g}.g[\bar{g}])^{n} \perp \text{sta} \downarrow 1$ is $\perp$ or $T[[$ WHILE exp DO cmd OD$]]$ occ Sta $\downarrow 1$. The latter result follows from Lemma 1 and property Ca.

The proof of Cb is by cases of occ′. If occ′ = $\langle\ \rangle$, the result follows by Cd because $T[[$ syn$]]$ occ″ sta $\downarrow 1$ is independent of occ″. If occ′ = $\langle 1\rangle\S$occ″ or occ′ = $\langle 2\rangle\S$occ″, the result follows by the hypotheses of the structural induction, the above expression for $T[[$ WHILE exp DO cmd OD$]]$ occ sta $\downarrow 2$(pla), and the additivity of filter.

The proof of Cc is also by cases of occ′, and suppose first that occ′ = $\langle\ \rangle$. The first result then follows from

$$\begin{aligned}&g_1\Delta(\text{iter}^{*0}(\text{sta}) \downarrow 1) \downarrow 2\ \langle\text{occ}\S\langle 1\rangle, \text{“}L\text{”}\rangle = \{\text{sta}\}\\ &g_1\Delta(\text{iter}^{*(n+1)}\text{sta} \downarrow 1) \downarrow 2\ \langle\text{occ}\S\langle 1\rangle, \text{“}L\text{”}\rangle, = \text{filter}\ \{\text{iter}^{*(n+1)}\text{sta} \downarrow 1\}\\ &= g_2\Delta(\text{iter}^{*n}(\text{sta}) \downarrow 1) \downarrow 2\ \langle\text{occ}\S\langle 2\rangle, \text{“}R\text{”}\rangle\end{aligned}$$

Next consider the second result, and let

$$r_n = T[[\text{exp}]]\ \text{occ}\S\langle 1\rangle\ \Delta\ (\text{iter}^{*n}(\text{sta}) \downarrow 1) \downarrow 1$$

abbreviate the state resulting from the n + 1st evaluation of the expression (when the loop is entered with sta). Then the set of states possible after evaluation of the expression naturally is

T[[WHILE exp DO cmd OD]] occ sta $\downarrow 2$ ⟨occ§⟨1⟩, "R"⟩

$$= \bigcup_{n=0}^{\infty} \text{filter } \{r_n\}$$

and the set of states possible before the body of the loop is

T[[WHILE exp DO cmd OD]] occ sta $\downarrow 2$ ⟨occ§⟨2⟩, "L"⟩

$$= \bigcup_{n=0}^{\infty} \text{filter } \{\text{condtrue } \Delta\ r_n \downarrow 1\}$$

The result then follows by the additivity of the filter.

If occ′ = ⟨1⟩§occ″ or occ′ = ⟨2⟩§occ″, the proof is by cases of syn *at* occ′. In all cases the result follows from the induction hypothesis and additivity (i.e., $x = H(y)$ follows because H is additive). □

Using properties Ca, Cb, Cc, and Cd, we can prove the following replacement theorem. As was said previously, applications of this theorem will in practice use an approximate data flow analysis and descriptions of sets of states [2, 9], rather than the collecting semantics and a single initial state.

THEOREM 1. *("Forward" replacement theorem). Consider some program syn ∈ Syn and occurrence occ that points into syn. Let sta ∈ Sta be an initial state and let a-col = I col[[syn]] ⟨ ⟩ sta* $\downarrow 2$ *be the result of data flow analyzing syn. If*

- *syn′ is of the same category as syn* at *occ, and*
- *syn′ behaves the same as syn* at *occ on each state (sta′) possible before syn* at *occ (sta′ ∈ a-col ⟨occ, "L"⟩)*

then syn [occ ← syn′] behaves the same as syn on the initial state (sta).

PROOF. Let P(occ′) be

∀sta′ ∈ a-col ⟨occ′, "L"⟩:
    Tsto[[syn *at* occ′]] (sta′)
    = Tsto[[syn[occ ← syn′] at occ′]] (sta′)

The theorem assumes P(occ), and by property Cd the result follows from P(⟨ ⟩). The proof amounts to showing P(occ′§⟨i⟩) ⇒ P(occ′) by cases of syn *at* occ′ for (occ′§⟨i⟩ a prefix of occ). We only consider the case where syn *at* occ′ is WHILE exp DO cmd OD. Then, $i = 1$ or $i = 2$ and syn[occ ← syn′] *at* occ′ is WHILE exp′ DO cmd′ OD. We have both P(occ′§⟨1⟩) and P(occ′§⟨2⟩): P(occ′§⟨i⟩) is by assumption and P(occ′§⟨3 − i⟩) follows, since syn *at* occ′§⟨3 − i⟩ equals syn[occ ← syn′] *at* occ′§⟨3 − i⟩.

To show P(occ′), let

g-sto[$\bar{g}$] = Tsto[[exp]] ∗ cond(Tsto[[cmd]] ∗ $\bar{g}$, λsta.sta inR)
$g′$-sto[$\bar{g}$] = Tsto[[exp′]] ∗ cond(Tsto[[cmd′]] ∗ $\bar{g}$, *λsta.sta inR*)

abbreviate the bodies used in the fixed points.

We first show

sta ∈ a-col ⟨occ′§⟨1⟩, "L"⟩ ⇒
  $(\lambda\bar{g}.g\text{-sto}[\bar{g}])^n \perp \text{sta} = (\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp \text{sta}$

The proof is by induction on n, and since the result is trivial for $n = 0$, consider the case $n + 1$. The plan of the proof is to monitor one iteration of both programs and show that either the computations terminate with the same result or they lead to more iterations starting in equal states (and then the inductive hypothesis applies).

Let us begin by choosing sta $\in$ a-col $\langle$occ′§$\langle 1\rangle$, "L"$\rangle$ so Tsto [[exp]] (sta) = Tsto [[exp′]] (sta) by P(occ′§$\langle 1\rangle$). If the common value is $\perp$ or "error" inR, the result is immediate, so assume it is sta′ inR. Then sta′ $\in$ a-col$\langle$occ′§$\langle 1\rangle$, "R"$\rangle$ follows by properties Ca and Cb. Unless Vcond(sta′) = true and Scond(sta′) $\in$ {true,false}, the result is immediate. If Vcond(sta′) = true and Scond(sta′) = false, then $(\lambda\bar{g}.g\text{-sto}[\bar{g}])^{n+1} \perp$ sta = (Bcond(sta′)) inR = $(\lambda\bar{g}.g'\text{-sto}[\bar{g}])^{n+1} \perp$ sta. If Vcond(sta′) = true = Scond(sta′), then Bcond(sta′) = condtrue(sta′) $\downarrow 1 \mid$ Sta is in a-col$\langle$occ′§$\langle 2\rangle$, "L"$\rangle$ by property Cc. Then, Tsto [[cmd]] (Bcond(sta′)) = Tsto [[cmd′]] (Bcond(sta′)) by P(occ′§$\langle 2\rangle$). Again, the result is immediate unless the common value is sta″ inR. From properties Ca, Cb, and Cc we have sta″ $\in$ a-col $\langle$occ′§$\langle 1\rangle$, "L"$\rangle$, so $(\lambda\bar{g}.g\text{-sto}[\bar{g}])^{n+1} \perp$ sta = $(\lambda\bar{g}.g\text{-sto}[\bar{g}])^{n} \perp$ sta″ = $(\lambda\bar{g}.g'\text{-sto}[\bar{g}])^{n} \perp$ sta″ = $(\lambda\bar{g}.g'\text{-sto}[\bar{g}])^{n+1} \perp$ sta follows by the induction hypothesis.

We now show P(occ′). Let sta $\in$ a-col$\langle$occ′, "L"$\rangle$ so that sta $\in$ a-col $\langle$occ′§$\langle 1\rangle$, "L"$\rangle$ by property Cc. The above result then gives Tsto [[WHILE exp DO cmd OD]] (sta) = $\sqcup_{n=0}^{\infty} (\lambda\bar{g}.g\text{-sto}[\bar{g}])^{n} \perp$ (sta) = $\sqcup_{n=0}^{\infty} (\lambda\bar{g}.g'\text{-sto}[\bar{g}])^{n} \perp$ (sta) = Tsto [[WHILE exp′ DO cmd′ OD]] (sta). □

Theorem 1 can be compared with the results achieved in [4], where forward (and backward, See Section 4) "data flow information" is exploited to guarantee that transformations preserve the *partial* correctness of programs with respect to input and output assertions. In [4], the semantics is not considered explicitly, but is merely assumed to be such that some constructed verification formulas are "sound." Theorem 1 expresses *strong* equivalence with respect to a store semantics (which can easily be converted to a standard semantics). For the method of [4] to be applicable, any loop of a program must be augmented with relevant "data flow information" (to be proved correct by theorem-proving methods). In the present approach, data flow analysis is used to "automatically" compute (approximations to) the required information.

4. PROGRAM TRANSFORMATIONS AND BACKWARD DATA FLOW ANALYSES

In this section we show how to validate program transformations that exploit information from a class of backward data flow analyses. An example is the transformation mentioned in the introduction where $x := y + (1 + 1)$ was replaced by a dummy statement. The intention is to specify the backward data flow information in an abstract way (using a so-called future semantics) similar to the collecting semantics of the previous section. It is possible to relate data flow analyses like "live variables analysis" [1] and "states that do not lead to an error" [3] to the future semantics, and the replacement theorem guarantees weak equivalence. Strong equivalence can be obtained by applying the replacement theorem twice (also by data flow analyzing the transformed program). In a special

Table V. Future Semantics

Domains

$G = C - c > (C \times A)$

$C = \text{Sta} - c > R$

$R = \text{Out} + \{\text{"error"}\}$

$A = \text{Pla} - c > C$

remaining domains as in Tables II and IV.

Combinator

$* \in G \times G - c > G$

$g_1 * g_2 = \lambda c.\ \langle g_1(g_2 c \downarrow 1) \downarrow 1, g_1(g_2\ c \downarrow 1) \downarrow 2 \sqcup g_2\ c \downarrow 2 \rangle$

Functions

$\text{attach} \in \text{Pla} - c > G$

$\text{attach(pla)} = \lambda c.\ \langle c, \bot[c/\text{pla}] \rangle$

$\text{cond} \in G \times G - c > G$

$\text{cond}(g_1, g_2) = \lambda c.\ \langle \text{cond}'(g_1\ c \downarrow 1, g_2\ c \downarrow 1), g_1\ c \downarrow 2 \sqcup g_2\ c \downarrow 2 \rangle$

where $\text{cond}' \in C \times C - c > C$

is $\text{cond}'(c_1, c_2) = \lambda \text{sta}.\ \text{Vcond(sta)} \rightarrow$ $(\text{Scond(sta)} \rightarrow c_1, c_2)\ (\text{Bcond(sta)}),$ "error" inR

and Vcond, Scond, Bcond, are as in Table II.

apply [[ope]] $\in G$

apply [[ope]] $= \lambda c.\ \langle \text{apply-sto[[ope]]} \oplus c, \bot \rangle$

where $\oplus \in G\text{-sto} \times C - c > C$

is $g\text{-sto} \oplus c = \lambda \text{sta}.g\text{-sto(sta)}\ \mathsf{E}\ \text{Sta} \rightarrow c(g\text{-}sto(sta) \mid \text{Sta}),$ "error" inR

assign[[ide]], content[[ide]], push[[bas]], read, write are defined similarly to apply[[ope]].

case we are able to obtain strong equivalence even when only the original program is data flow analyzed.

Future Semantics

The purpose of the future semantics is to associate each program point with the meaning of the remainder of the program. The *dynamic effect of the remainder of the program* can be given by a *continuation* [11], so we shall associate a continuation with each program point. The continuations to be used are those that would naturally be used in a continuation-style store semantics (e.g., members of $C = \text{Sta} - c\rangle\ (\text{Out}+\{\text{"error"}\})$ and the obvious "final" (or initial [11]) continuation is $\lambda\text{sta.sta} \downarrow 3$ in $(\text{Out}+\{\text{"error"}\})$).

The future semantics is given by Tables III and V. Domain $C = \text{Sta} - c\rangle\ R$ is the domain of continuations. As in the previous section, domain $A = \text{Pla} - c\rangle\ C$ is used to associate each program point with the desired information (here a continuation). We use the suffix fut for the future semantics.

Example. Define the continuations

$c = \lambda\text{sta.}\ \text{sta}\downarrow 3\text{in} \cdots$

$c' = \lambda\text{sta.}\ (\text{sta}\downarrow 3) \S \langle 5\text{inVal} \rangle \text{in} \cdots$

For the program WRITE 2 + 3, we then get

$T\text{fut}[[\text{WRITE } 2 + 3]] \langle\ \rangle\ c = \langle c', a \rangle$

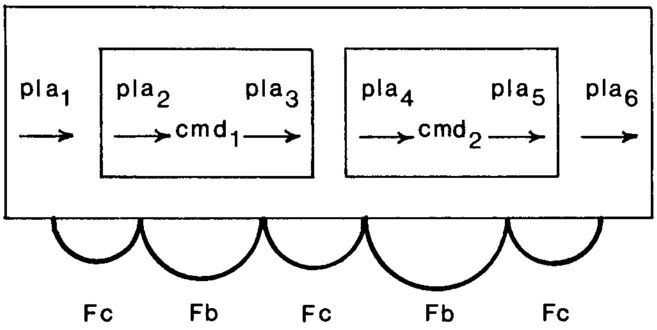


Figure 3. pla_1 = ⟨occ, "*L*"⟩, pla_2 = ⟨occ§⟨1⟩, "*L*"⟩, pla_3 = ⟨occ§⟨1⟩, "*R*"⟩, pla_4 = ⟨occ§⟨2⟩, "*L*"⟩, pla_5 = ⟨occ§⟨2⟩, "*R*"⟩, pla_6 = ⟨occ, "*R*"⟩.

where, for example,

$a(\text{pla}_1) = c'$
$a(\text{pla}_7) = \lambda\text{sta}.\ (\text{sta}\downarrow 3)\S\langle \text{sta}\downarrow 4\downarrow 1\rangle\text{in}\cdots$
$a(\text{pla}_8) = c$

The first component of the semantic function (i.e., $\lambda c.T\text{fut}[[\text{syn}]]\text{occ}\ c \downarrow 1$) is an ordinary continuation-style store semantics. The store semantics of Section 3 is a version of this with the continuations removed. This is formally expressed by the following property (an analogue of Ca):

PROPERTY Fa. *Let syn ∈ Syn, occ ∈ Occ be maximal and c ∈ C. Then Tfut*[[*syn*]] *occ c* ↓1 = *Tsto*[[*syn*]] ⊕ *c.*

PROOF OF Fa. By (an omitted) structural induction using the auxiliary functions cond′ and ⊕ that satisfy cond′(g_1-sto ⊕ *c*, g_2-sto ⊕ *c*) = (cond-sto(g_1-sto, g_2-sto)) ⊕ *c* and (g_1-sto ∗ g_2-sto) ⊕ *c* = g_1-sto ⊕ (g_2-sto ⊕ *c*). □

The second component of the semantic function applied to some continuation (i.e., *T*fut [[syn]] occ *c* ↓2) maps a program point to the continuation corresponding to the remainder of the program. This gives an abstract way of specifying backward data flow information that is similar to the collecting semantics. To obtain a replacement theorem, we need to state some properties (Fb, Fc, and Fd) of the future semantics. These properties correspond closely to Cb, Cc, and Cd of Section 3, except that intuitively information now flows from right to left rather than left to right. In particular, in Figure 3 it is shown that the purpose of Fb and Fc closely mirror Figure 2.

PROPERTY Fb. *Let syn ∈ Syn, occ ∈ Occ by maximal, c ∈ C and occ′ ∈ Occ point into syn and abbreviate a-fut = Tfut*[[*syn*]] *occ c* ↓2. *Then a-fut*⟨*occ*§*occ′*, "*L*"⟩ = *Tfut*[[*syn* at *occ′*]] ⟨ ⟩ (*a-fut*⟨*occ*§*occ′*, "*R*"⟩)↓1.

PROPERTY Fc. *Let syn ∈ Syn, occ ∈ Occ be maximal, c ∈ C and occ′ ∈ Occ point into syn and abbreviate a-fut = Tfut*[[*syn*]] *occ c* ↓2. *If syn* at *occ′ is WHILE exp DO cmd OD, then a-fut*⟨*occ*§*occ′*§⟨2⟩, "*R*"⟩ = *a-fut*⟨*occ*§*occ′*, "*L*"⟩, *and a-fut*⟨*occ*§*occ′*§⟨1⟩,"*R*"⟩ = *cond′* (*a-fut*⟨*occ*§*occ′*§⟨2⟩, "*L*"⟩, *and a-fut*⟨*occ*§*occ′*, "*R*"⟩).

For the remaining constructs, there are more or less similar properties.

PROPERTY Fd. *Let syn ∈ Syn, occ ∈ Occ be maximal, c ∈ C and pla ∈ Pla. Then*

(*i*) $Tfut[[syn]]\ occ\ c \downarrow 2\ pla \neq \perp \Rightarrow$ *pla is a descendant of occ*
(*ii*) $Tfut[[syn]]\ occ\ c \downarrow 2\langle occ, \text{“}R\text{”}\rangle = c$
(*iii*) $Tfut[[syn]]\ occ\ c \downarrow 2\langle occ, \text{“}L\text{”}\rangle = Tfut[[syn]]\ occ\ c \downarrow 1.$

PROOF OF Fb, Fc, and Fd. By an omitted structural induction. □

Using properties Fa, Fb, Fc, and Fd we can prove the following replacement theorem, which expresses weak equivalence. The statement of the theorem makes use of the phrase “syn′ *followed by c′* ”, which means $T\text{sto}[[\text{syn}']] \oplus c'$.

THEOREM 2. (*“Backward” replacement theorem*). *Consider some program syn ∈ Syn and occurrence occ that points into syn. Let c ∈ C be a “final” continuation and let a-fut* $= Tfut[[syn]]\langle\ \rangle\ c \downarrow 2$ *be the result of data flow analyzing syn. If*

- *syn′ is of the same syntactic category as syn at occ, and*
- *the continuation c′ holding after syn* at *occ* ($c' = a\text{-}fut\langle occ, \text{“}R\text{”}\rangle$) *satisfies that syn′ followed by c′ is less defined than syn* at *occ followed by c′*

then syn[occ ← syn′] followed by c (the final continuation) is less defined than syn followed by c.

PROOF. This proof follows the same overall plan as that of Theorem 1. Let $P(\text{occ}')$ mean that

$T\text{sto}[[\text{syn } at \text{ occ}']] \oplus c' \sqsupseteq T\text{sto}[[\text{syn } [\text{occ} \leftarrow \text{syn}'] at \text{ occ}']] \oplus c'$ where $c' = a\text{-fut}\langle\text{occ}', \text{“R”}\rangle$.

The assumption is $P(\text{occ})$ and the result follows from $P(\langle\ \rangle)$ by property Fd. The proof consists in showing $P(\text{occ}'\S\langle i\rangle) \Rightarrow P(\text{occ}')$ by cases of syn *at* occ′ (for occ′ § ⟨*i*⟩, a prefix of occ). We only consider the case where syn *at* occ′ is WHILE exp DO cmd OD. Then $i \in \{1, 2\}$ and syn[occ←syn′] *at* occ′ is WHILE exp′ DO cmd′ OD, and we have both $P(\text{occ}'\S\langle 1\rangle)$ and $P(\text{occ}'\S\langle 2\rangle)$.

Let

$$g\text{-sto}[\bar{g}] = T\text{sto}[[\text{ exp}]] * \text{cond}(T\text{sto}[[\text{ cmd}]] * \bar{g}, \lambda\text{sta.sta inR})$$
$$g'\text{-sto}[\bar{g}] = T\text{sto}[[\text{ exp}']] * \text{cond}(T\text{sto}[[\text{ cmd}']] * \bar{g}, \lambda\text{sta.sta inR})$$

abbreviate the bodies used in the fixed points.

We first show $\text{FIX}(\lambda\bar{g}.g\text{-sto}[\bar{g}]) \oplus c' \sqsupseteq (\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp \oplus c'$ for $c' = a\text{-fut}\langle\text{occ}', \text{“}R\text{”}\rangle$. The proof is by induction on n, and the result is easy for $n = 0$, so consider $n + 1$. By Fa, Fb, and Fc we have

$a\text{-fut}\langle\text{occ}'\S\langle 2\rangle, \text{“}R\text{”}\rangle = \text{FIX}(\lambda\bar{g}.g\text{-sto}[\bar{g}]) \oplus c'$, so $a\text{-fut}\langle\text{occ}'\S\langle 2\rangle, \text{“}L\text{”}\rangle = T\text{sto}[[\text{ cmd}]] \oplus (\text{FIX}(\lambda\bar{g}.g\text{-sto}[\bar{g}]) \oplus c') \sqsupseteq T\text{sto}[[\text{ cmd}'\ ]] \oplus ((\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp \oplus c')$ by Fa, Fb, $P(\text{occ}'\S\langle 2\rangle)$ and ⊕ continuous.

Proceeding in this way, $a\text{-fut}\langle\text{occ}'\S\langle 1\rangle, \text{“}R\text{”}\rangle$

$= \text{cond}'(T\text{sto}[[\text{ cmd}]] \oplus (\text{FIX}(\lambda\bar{g}.g\text{-sto}[\bar{g}]) \oplus c'), c') \sqsupseteq \text{cond}'(T\text{sto}[[\text{ cmd}']] \oplus ((\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp \oplus c'), c')$ and $T\text{sto}[[\text{ exp}]] \oplus \text{cond}'(T\text{sto}[[\text{ cmd}]] \oplus (\text{FIX}(\lambda\bar{g}.g\text{-sto}[\bar{g}]) \oplus c'), c') \sqsupseteq T\text{sto}[[\text{ exp}']] \oplus \text{cond}'(T\text{sto}[[\text{ cmd}']] \oplus ((\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp \oplus c'), c')$ i.e., $g\text{-sto}[\text{FIX}(\lambda\bar{g}.g\text{-sto}[\bar{g}])] \oplus c' \sqsupseteq (\lambda\bar{g}.g'\text{-sto}[\bar{g}])^{n+1} \perp \oplus c'$.

Then

$T\text{sto}[[\text{ WHILE exp DO cmd OD}]] \oplus c' = g\text{-sto}[\text{FIX}(\lambda\bar{g}.g\text{-sto}[\bar{g}])] \oplus c' \sqsupseteq \bigsqcup_{n=0}^{\infty} ((\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp \oplus c') = T\text{sto}[[\text{ WHILE exp}' \text{ DO cmd}' \text{ OD}]] \oplus c'$. □

Even if we assume that (in the notation of the theorem) syn′ followed by c' is equal to syn *at* occ followed by c', we cannot obtain that syn[occ ← syn′] followed by c is equal to syn followed by c. The following example shows that this must be so. Consider the program

READ(x) ; WHILE $x > 0$ DO $x := 0 - x$ OD ; WRITE(0)

followed by the final continuation $c = \lambda\text{sta}.\text{sta}{\downarrow}3\text{inR}$ that simply emits the output. The continuation c' holding immediately before OD is

Tsto[[WHILE $x > 0$ DO $x := 0 - x$ OD ; WRITE(0)]] $\oplus\ c$

so that $x := 0 + x$ followed by c' is equal to $x := 0 - x$ followed by c'. But the above program always terminates, whereas the transformed program READ(x); WHILE $x > 0$ DO $x := 0 + x$ OD; WRITE(0) loops on some inputs. Intuitively, this is because the continuation holding before OD is affected by the transformation. So as in [4], only weak equivalence is obtained, but even then there are advantages of using the present approach. We consider that a formal (store) semantics and WHILE loops need not be augmented with assertions.

By applying Theorem 2 twice, we can obtain strong equivalence. First apply it to syn and then to syn[occ ← syn′], so that both syn and syn[occ ← syn′] are data flow analyzed. Since syn = (syn[occ ← syn′]) [occ ← syn *at* occ] this gives conditions for when syn followed by some final continuation (c) equals syn[occ ← syn′] followed by the same continuation. This is the desired result, since only the output of a program is important (i.e., $c = \lambda\text{sta}.\text{sta}\ {\downarrow}3\ \text{inR}$), but it is slightly unsatisfactory that the transformed program also has to be data flow analyzed.

Liveness Semantics

Many backward data flow analyses can be related to the future semantics and viewed as approximating it. One example is the determination of states which do not lead to an error [3]. Consider some program (syn) and final continuation (c). If c' is the continuation holding at some program point pla ($c' = T\text{fut}[[\text{syn }]]\ \langle\ \rangle\ c\ {\downarrow}2\ \text{pla}$), then the set of states not leading to an error is $\{\text{sta} \in \text{Sta} \mid c'(\text{sta}) \neq$ "error" $\text{inR}\}$. Another example is "live variables analysis" [1] that is a syntactic way of associating each program point with a set of live identifiers. Correctness of "live variables analysis" implies that if some identifier (ide) is deemed not be be live at some program point (pla), then the continuation holding there ($c' = T\text{fut}[[\text{syn }]]\ \langle\ \rangle\ c\ {\downarrow}2\ \text{pla}$) must produce the same output ($c'\ (\text{sta}_1) = c'\ (\text{sta}_2)$) for any two states differing only on that identifier ($\text{sta}_1\ {\downarrow}1 = \text{sta}_2\ {\downarrow}1[\text{sta}_1\ {\downarrow}1[[\text{ide}]]\ /\ \text{ide}]$ and $\text{sta}_1\ {\downarrow}i = \text{sta}_2\ {\downarrow}i$ for $i \neq 1$).

By the above correctness condition for "live variables analysis," we can validate program transformations exploiting liveness information. But both the original and the transformed program has to be data flow analyzed, contrary to what is done in practice. We therefore define a liveness semantics (suffix liv) that computes "live variables" and we sketch how to obtain strong equivalence when only the original program is data flow analyzed. The liveness semantics (Tables

Table VI. Liveness Semantics

Domains

$G = L - c > (L \times A)$

$L = P(\text{ide})$

$A = \text{Pla} - c > L$

remaining domains as in Tables II and IV.

Combinator

$* \in G \times G - c > G$

$g_1 * g_2 = \lambda 1.\ \langle g_1(g_2\ 1 \downarrow 1) \downarrow 1, g_1(g_2\ 1 \downarrow 1) \downarrow 2 \sqcup g_2\ 1 \downarrow 2\rangle$

Functions

$\text{attach} \in \text{Pla} - c > G$

$\text{attach(pla)} = \lambda 1.\langle 1, \bot[1/\text{pla}]\rangle$

$\text{cond} \in G \times G - c > G$

$\text{cond}(g_1, g_2) = g_1 \sqcup g_2$

$\text{apply}[[\text{ope}]] \in G$

$\text{apply}[[\text{ope}]] = \lambda 1.\langle 1, \bot\rangle$

$\text{assign}[[\text{ide}]] \in G$

$\text{assign}[[\text{ide}]] = \lambda 1.\langle 1 - \{\text{ide}\}, \bot\rangle$

$\text{content}[[\text{ide}]] \in G$

$\text{content}[[\text{ide}]] = \lambda 1.\langle 1 \cup \{\text{ide}\}, \bot\rangle$

push[[bas]], read, write are defined similarly to apply[[ope]].

III and VI) operates in essentially the same way as the future semantics. The most interesting functions are assign[[ide]] and content[[ide]]. They simply correspond to the usual transfer functions [1] associated with the basic blocks of a program represented as a flowchart. So assign[[ide]] records that ide has been killed and content[[ide]] records that ide has now been generated.

Example. For the program WRITE 2 + 3, we get

$\mathcal{T}\text{liv}[[\text{WRITE } 2 + 3]]\ \langle\ \rangle\ l = \langle l, a\rangle$

where, for example, $a(\text{pla}_1) = a(\text{pla}_7) = a(\text{pla}_8) = l$.

Property La below expresses the connection between the store semantics and the first component of the liveness semantics. For this we need a predicate called l-similar such that sta_1 is *l-similar* to sta_2 if sta_1 and sta_2 differ only on identifiers not in the set l of live identifiers. Formally,

$$\begin{aligned}&\text{sta}_1 = \langle \text{env}_1, \text{inp}, \text{out}, \text{tem}\rangle \Rightarrow\\ &\quad \exists\ \text{env}_2{:}[\text{sta}_2 = \langle \text{env}_2, \text{inp}, \text{out}, \text{tem}\rangle \wedge \text{ide} \in l\\ &\qquad \Rightarrow \text{env}_1[[\text{ide}]] = \text{env}_2[[\text{ide}]]\ ]\end{aligned}$$

So, for example, the continuation $\lambda\text{sta.sta}\downarrow 3\text{inR}$ will give the same result on any two states that are l-similar for some l (e.g., $l = \varnothing$). Furthermore, Ide-similarity means equality.

Define syn_1 to be *$\langle ll, lr\rangle$-related* to syn_2 where $r_i = \mathcal{T}\text{sto}[[\text{syn}_i]]\ \text{sta}_i$ satisfies that if sta_1 is ll-similar to sta_2 then $r_1 = r_2$ or $r_i = \text{sta}_i'$ inR with sta_1' lr-similar to sta_2'. When $\text{syn}_1 = \text{syn}_2$, this means that the final values of variables in lr only depend on the initial values of the variables in ll.

PROPERTY La. *Let syn ∈ Syn, occ ∈ Occ be maximal, lr ∈ L and ll = Tliv*[[*syn*]] *occ lr* $\downarrow$1. *Then syn is* ⟨*ll, lr*⟩*-related to syn.*

PROOF OF La. By structural induction. Lemma 1 is used when syn is WHILE exp DO cmd OD. We omit the details. □

We omit stating properties Lb, Lc, and Ld that are analogues of Fb, Fc, and Fd. Using the latter, we can prove the following replacement theorem, guaranteeing "strong equivalence", using only one data flow analysis.

THEOREM 3. *Consider some program syn ∈ Syn and occurrence occ that points into syn. Let l ∈ L be a set of live identifiers and let a-liv = Tliv*[[*syn*]] ⟨ ⟩ *l* $\downarrow$2 *be the result of data flow analyzing syn. If*

- *syn′ is of the same category as syn* at *occ, and*
- *for the sets ll and lr of live identifiers before and after syn* at *occ* (*ll = a-liv* ⟨*occ,* "*L*"⟩ *and lr = a-liv* ⟨*occ,* "*R*"⟩) *that syn′ is* ⟨*ll, lr*⟩*-related to syn* at *occ then syn*[*occ* ← *syn′*] *is* ⟨*Ide, l*⟩*-related to syn.*

PROOF. Similar to that of the previous theorems. For the WHILE case, Lemma 1 is used. We omit the details. □

When syn[occ←syn′] is ⟨Ide, *l*⟩-related to syn (for some *l*, e.g., $l = \varnothing$), we clearly have that syn[occ←syn′] followed by c = λsta.sta$\downarrow$3inR equals syn followed by *c*. Hence, the program transformation has not affected the overall meaning of the program.

It is hoped that the above development can be generalized so that the liveness semantics is replaced by a more abstract formulation. The future semantics gives information about program points (the effect of the remainder of the program) and the liveness semantics also does so: If *l* is the set of identifers live at some program point, then any two *l*-similar states produce the same output. Additionally, the liveness semantics gives information about program pieces (the concept of ⟨l_1, l_2⟩-related). Perhaps the future semantics should be augmented with (suitable generalizations of) such information.

5. CONCLUSION

It has been shown that it is possible to validate program transformations that exploit data flow information. The formulations of the data flow analyses were chosen on purpose to be rather abstract, so that the result would be applicable for a large class of data flow analyses. For the forward data flow analyses, a so-called collecting semantics was used and strong equivalence (Theorem 1) was obtained by data flow analyzing only the original program. For the backward data flow analyses, a so-called future semantics was defined, but only weak equivalence (Theorem 2) could be obtained, unless the transformed program was also data flow analyzed. It was possible to overcome this deficiency in a treatment of live variables analysis, but it remains to generalize the method to the general setting.

To explain why Theorem 2 is weaker than Theorem 1, we introduce the following classification of data flow analyses. Call a data flow analysis *first-order* if the data flow properties describe sets of states, and call it *second-order* if this is not so. (We shall not at this point be more precise about what a second-order property actually is.) Then, constant propagation is a first-order data flow analysis and live variables analysis is a second-order data flow analysis. Not all forward data flow analyses are first-order, for example, reaching definitions [1]. Neither are all backward data flow analyses second-order, for example, the detection of the set of states that do not lead to an error [3].

Theorem 1 is applicable to all first-order forward analyses because the collecting semantics intuitively is as precise as possible. It is not clear how to incorporate second-order forward analyses because it is not clear what to use instead of the collecting semantics. It would seem that Theorem 2 is applicable to both first-order and second-order backward analyses. Concerning first-order backward analyses, one may envisage a version of the collecting semantics (or, rather, its companion, the static semantics [9]) for backward analyses so that a theorem giving strong equivalence could be obtained. The difference between strong versus weak equivalence thus seems to be due to the first-order versus second-order distinction, because in the latter case the program transformation affects the abstract data flow information.

REFERENCES

1. AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design.* Addison-Wesley, London, 1977.
2. COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Proceedings 6th ACM Symposium on Principles of Programming Languages* (1979), ACM, New York, 269–282.
3. COUSOT, P. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds., Prentice-Hall, Englewood Cliffs, N.J., 1981, 303–342.
4. GERHART, S. L. Correctness-preserving program transformations. In *Proceedings 2nd ACM Symposium on Principles of Programming Languages* (1975), ACM, New York, 54–66.
5. HUET, G., AND LANG, B. Proving and applying program transformations expressed with second-order patterns. *Acta Inf. 11* (1978), 31–55.
6. LOVEMAN, D. B. Program improvement by source-to-source transformations. *J. ACM 24* (1977), 121–145.
7. MILNE, R., AND STRACHEY, C. *A Theory of Programming Language Semantics.* Chapman and Hall, London, 1976.
8. MILNER, R. Program semantics and mechanized proof. In *Foundations of Computer Science II*, K. R. Apt and J. W. de Bakker, Eds., Mathematical Centre Tracts 82, Amsterdam, 1976, 3–44.
9. NIELSON, F. A denotational framework for data flow analysis. *Acta Inf. 18* (1982), 265–287.
10. ROSEN, B. K. Tree-manipulating systems and Church–Rosser theorems. *J. ACM 20* (1973), 160–187.
11. STOY, J. E. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Mass., 1977.

Received June 1984; revised January 1985; accepted April 1985