ISSN 0105-8517

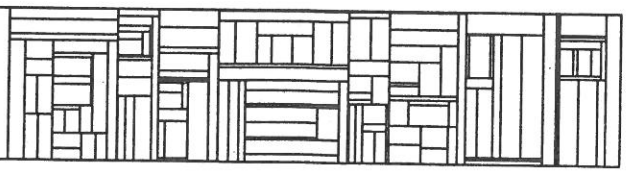
PROGRAM TRANSFORMATIONS IN A DENOTATIONAL SETTING

by

Flemming Nielson

DAIMI PB-140
November 1981

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



Contents

1. INTRODUCTION 1

2. PRELIMINARIES 3

3. PROGRAM TRANSFORMATIONS AND FORWARD DATA FLOW ANALYSES.... 5
   Toy Language 5
   Collecting Semantics 8
   Program Transformations 13

4. PROGRAM TRANSFORMATIONS AND BACKWARD DATA FLOW ANALYSES... 20
   Future Semantics 20
   Liveness Semantics 25

5. CONCLUSION 29

References 30

PROGRAM TRANSFORMATIONS IN A DENOTATIONAL SETTING

Abstract

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

Strong equivalence is obtained for transformations that exploit forward data flow information, whereas weak equivalence is obtained for transformations that exploit backward data flow information. 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.

Keywords

program transformations, denotational semantics, correctness proof, forward data flow analysis, backward data flow analysis, live variables analysis.

1. INTRODUCTION

In this paper we consider a class of program transformations, where a program is transformed into another in the same language (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 the 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 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:

```
... y := 2 ... (no y's) ... x := y + (1+1) ... (no x's) ... x := 0 ...
```

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 a *forward* data flow analysis (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 replace x := y + (1+1) by a dummy statement (or eliminate it). 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 a *backward* data flow analysis (live variables analysis [1]). The meanings of x := y + (1+1) and a dummy statement are different so this transformation is also not so 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 forward data flow information (section 3) and 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 some 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 [11] and [7] but the domains are cpo's (as in [8]) rather than complete lattices. Below we explain some fundamental notions and non-standard notation (≫, ==, -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 non-empty 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* will be used for cpo's and elements of some domain S are denoted s, s', s_1 etc. A domain is *flat* if any chain contains at most 2 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* $S_1 + \ldots + S_n$. This is a domain with a new least element and injection functions $in S_i$, enquiry functions $\sqsubseteq S_i$ and projection functions $|S_i$. The *cartesian product* $S_1 x \ldots x S_n$ is a domain with selection functions $\downarrow i$. The domain S* of *lists* is $\{< >\} + S + (S \times S) + \ldots$ Function # yields the length of a list, function $\dagger i$ removes the first i elements and § concatenates lists. By $\mathcal{P}(S)$ 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 ∈ S -t> S' is *additive* (a *complete-⊔-morphism*) if $f(\bigsqcup S'') = \bigsqcup\{f(s) \mid s \in S''\}$ 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 ∈ S is a *fixed point* of f ∈ S -t> S if f(s) = s. When S is partially ordered it 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 ∈ S -c> S the least fixed point always exists and is given by $FIX(f) = \bigsqcup\{f^n(\bot) \mid n \geqslant 0\}$. We shall frequently write $\bigsqcup_{n=0}^{\infty} f^n(\bot)$ instead of $\bigsqcup\{f^n(\bot) \mid n \geqslant 0\}$.

For any domain S we use the symbol == as a continuous equality predicate (S x S -c> T), whereas = is reserved for true equality. So (true == ⊥) will be ⊥ whereas (true = ⊥) is false. When S is of finite height it is assumed that s_1 == s_2 is ⊥ if one of s_1, s_2 is non-maximal and equals $s_1 = s_2$ otherwise. We write ⩾⩾ 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 ⊥ depending on whether t is true, false or ⊥. 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 forward data flow information. 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 ::= cmd_1;cmd_2 | ide := exp | IF exp THEN cmd_1 ELSE cmd_2 FI
| WHILE exp DO cmd OD | WRITE exp | READ ide
exp ::= exp_1 ope exp_2 | ide | bas

We do not specify the syntax of identifiers (Ide), basic values (Bas) and operators (Ope). The semantics is given by tables 1 and 2. Table 2 defines some domains and auxiliary functions as well as an associative combinator (*) used for sequencing. Table 1 defines a single semantic function $\mathcal{T}$ that ascribes meaning to both commands and expressions. It simplifies some notation to be used later that only one semantic function is used. The semantic function is in direct style because continuations are not needed in 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]. 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> $\mathcal{T}$ verifies whether the argument state is on a special form. Only if this is the case, the state will 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 acquainted with [7] should have no trouble in supplying the definitions.

TABLE 1: Semantic Function

$T \in$ Syn -c> G

T[[cmd_1;cmd_2]] = T[[cmd_1]] * T[[cmd_2]]

T[[ide := exp]] = T[[exp]] * assign[[ide]]

T[[IF exp THEN cmd_1 ELSE cmd_2 FI]] =
T[[exp]] * cond(T[[cmd_1]] , T[[cmd_2]])

T[[WHILE exp DO cmd OD]] =
FIX(λg. T[[exp]] * cond(T[[cmd]] * g , λsta.sta inR))

T[[WRITE exp]] = T[[exp]] * write

T[[READ ide]] = read * assign[[ide]]

T[[exp_1 ope exp_2]] = T[[exp_1]] * T[[exp_2]] * apply[[ope]]

T[[ide]] = content[[ide]]

T[[bas]] = push[[bas]]

Table 2: Store Semantics

Domains

```
G = Sta -c> R
R = Sta + {"error"}

Sta = Env x Inp x Out x Tem        states
Env = Ide -c> Val                  environments
Inp = Val*                         inputs
Out = Val*                         outputs
Tem = Val*                         temporary result stacks
Val = T + N + ... + {"nil"}        values
```

Combinator

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

$g_1 * g_2 = \lambda sta.g_1(sta) \in Sta \to g_2(g_1(sta) \mid Sta), g_1(sta)$

Functions

```
cond ∈ G x G -c> G
cond(g1,g2) = λsta.Vcond(sta) → (Scond(sta) → g1,g2)(Bcond(sta)),
                                                    "error" inR
   Vcond<env,inp,out,tem> =  #tem ≥≥ 1 → tem ↓1 ∈ T, false
   Bcond<env,inp,out,tem> =  <env,inp,out,tem †1>
   Scond<env,inp,out,tem> =   tem ↓1 | T

apply[[ ope]] ∈ G
apply[[ ope]] = λsta.Vapply[[ ope]] (sta) → Bapply[[ ope]] (sta) inR,
                                                    "error" inR
      Vapply[[ ope]] <env,inp,out,tem> = #tem ≥≥ 2
      Bapply[[ ope]] <env,inp,out,tem> = <env,inp,out,<0[[ ope]]
                                          <tem ↓2, tem ↓1>>§(tem†2)>

assign[[ ide]] ∈ G
assign[[ ide]] = λsta.Vassign[[ ide]] (sta) → Bassign[[ ide]] (sta) inR,
                                                    "error" inR
      Vassign[[ ide]] <env,inp,out,tem> = #tem ≥≥ 1
      Bassign[[ ide]] <env,inp,out,tem> = <env[tem ↓1/ide],
                                           inp,out,tem †1>

content[[ ide]] ∈ G
content[[ ide]] = λsta.Vcontent[[ ide]] (sta)→ Bcontent[[ ide]](sta)inR,
                                                    "error" inR
      Vcontent[[ ide]] <env,inp,out,tem> = true
      Bcontent[[ ide]] <env,inp,out,tem> = <env,inp,out,
                                           <env[[ ide]] >§tem>
```

push[[bas]] ∈ G, read ∈ G, write ∈ G are defined similarly.

It is not difficult to give a standard semantics that is equivalent to the store semantics of tables 1 and 2. The main reason for using a store semantics is that it becomes easier to define the collecting semantics below. A consequence is that the collecting semantics is the continuation removed version of that in [9]. Semantic functions $\mathcal{O}$ (and $\mathcal{B}$) used in apply (and push) are not defined here, and we also omit the proofs of correctness of the functionalities stated in the tables. The lemma below is needed in later proofs. It says that the iterates in a WHILE loop either give no information or full information.

Lemma 1 Let $g[\bar{g}] = \mathcal{T}[\![\text{exp}]\!] * \text{cond}(\mathcal{T}[\![\text{cmd}]\!] * \bar{g}, \lambda \text{sta}.\text{sta inR})$.† Then $(\lambda\bar{g}.g[\bar{g}])^n \perp \text{sta}$ is either $\perp$ or $\mathcal{T}[\![\text{WHILE exp DO cmd OD}]\!]$ sta. □

Proof It suffices to show that $\forall \text{sta}: [(\lambda\bar{g}.g[\bar{g}])^n \perp \text{sta} \neq \perp \Rightarrow \forall g_o: (\lambda\bar{g}.g[\bar{g}])^n g_o \text{ sta} = (\lambda\bar{g}.g[\bar{g}])^n \perp \text{sta}]$. This is because $(\lambda\bar{g}.g\ \bar{g}\ )^{n+1} \perp = (\lambda\bar{g}.g[\bar{g}])^n g_o$ when $g_o = g[\perp]$. The proof is by induction in n and since the case n = 0 is obvious consider the inductive step. It is easy to see that $(\lambda\bar{g}.g[\bar{g}])^{n+1} g_o$ sta independently of g_o is $\perp$, "error" inR, sta' inR or $(\lambda\bar{g}.g[\bar{g}])^n g_o$ sta" where sta' and sta" are independent of g_o. 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 $\in$ Inp). Instead of specifying the result of this execution the purpose of the collecting semantics is to associate each program point with the states

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

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 semi-adjoined [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 shall identify a program with a parse-tree and to each node we associate an occurrence (a member of Occ = N*). The root has occurrence < > and the i'th son of a node with occurrence occ has occurrence occ§<i>. A program point will be represented by a tuple <occ,q> ∈ Pla = Occ x Q. The quotation q is used to indicate whether the program point is to the left (q = "L") or to the right (q = "R") of the node (figure 1). From the description above it follows that only maximal elements of Pla will be used.

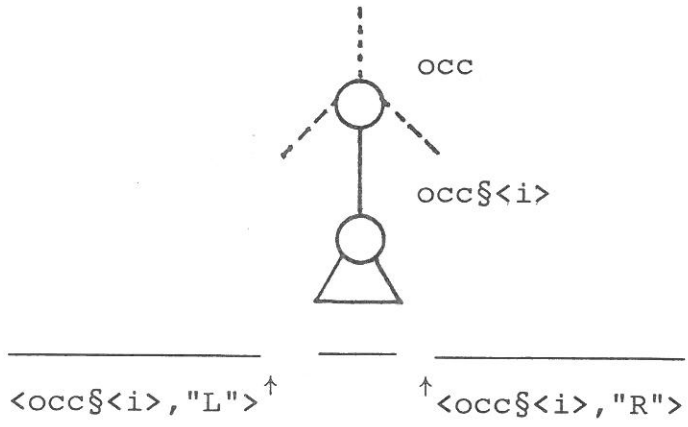


FIGURE 1

According to the usual view of parse-trees the occurrence associated with a node 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 (e.g. attach <occ,"L">) that associate information with program points. Table 3 sketches the result of performing these changes. The systematic placement of attach is useful later.

The collecting semantics is specified by tables 3 and 4. Domain A = Pla -c> $\mathcal{P}$(Sta) is used to associate each program point with those states that control can be in when reaching that point. The associative combinator * is continuous in its right argument (but not the left [9]) so FIX (in table 3) is only applied to continuous functions. To distinguish between the collecting semantics and the store semantics we use suffixes col and sto, so e.g. $\mathcal{T}$col is the semantic function of the collecting semantics.

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 pertain) 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 $\in$ Syn, occ $\in$ Occ be maximal and sta $\in$ Sta. Then $\mathcal{T}$col[[syn]] occ sta $\downarrow$1 = $\mathcal{T}$sto[[syn]] sta.

□

Proof of Property Ca is by a straight-forward structural induction and is omitted.

□

TABLE 3: Modified Semantic Function

$T \in$ Syn -c> Occ -c> G

```
T[[ IF exp THEN cmd1 ELSE cmd2 FI]]  occ =
            attach <occ,"L"> *
            T[[ exp]] occ§<1> *
            cond(T[[ cmd1]] occ§<2> * attach <occ,"R">
               , T[[ cmd2]] occ§<3> * attach <occ,"R">)
T[[ WHILE exp DO cmd OD]] occ =
            attach <occ,"L"> *
            FIX(λg.T[[ exp]] occ§<1> *
                 cond(T[[ cmd]] occ§<2> * g
                    , attach <occ,"R">))
T[[ exp1 ope exp2]] occ =
            attach <occ,"L"> *
            T[[ exp1]] occ§<1> *
            T[[ exp2]] occ§<3> *
            apply[[ ope]] *
            attach <occ,"R">
```

remaining clauses changed similarly to the one for exp_1 ope exp_2.

TABLE 4: Collecting Semantics

Domains

```
G = Sta -t> (R x A)
R = Sta + {"error"}
A = Pla -c> P(Sta)

Occ = N*                          occurrences
Pla = Occ x Q                     places
```

remaining domains as in table 2.

Combinator

$* \in G \times G \xrightarrow{t} G$ (continuous in second argument)

$g_1 * g_2 = \lambda sta.\langle g_1(sta)\downarrow 1 \in Sta \to g_2(g_1(sta)\downarrow 1 \mid Sta)\downarrow 1, g_1(sta)\downarrow 1$
$, [g_1(sta)\downarrow 1 \in Sta \to g_2(g_1(sta)\downarrow 1 \mid Sta)\downarrow 2, \perp] \sqcup g_1(sta)\downarrow 2\rangle$

Functions

$attach \in Pla \xrightarrow{c} G$

$attach(pla) = \lambda sta.\langle sta\ inR, \perp[\{sta\}/pla]\rangle$

$cond \in G \times G \xrightarrow{c} G$

$cond(g_1,g_2) = \lambda sta.Vcond(sta) \to$
$(Scond(sta) \to g_1,g_2)\ (Bcond(sta)),$
$\langle "error"\ inR, \perp\rangle$

Vcond, Scond, Bcond as in table 2.

$apply[\![ope]\!] \in G$

$apply[\![ope]\!] = \lambda sta.\langle Vapply[\![ope]\!](sta) \to (Bapply[\![ope]\!](sta))\ inR,$
$"error"\ inR$
$, \perp\rangle$

Vapply[[ope]] , Bapply[[ope]] as in table 2.

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

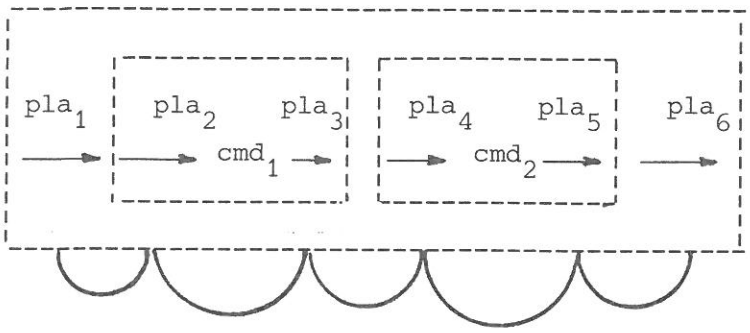
Program Transformations

We now consider how to validate program transformations like the one mentioned in the introduction where 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 = occ§occ' for some maximal occ' and that pla↓2 ∈ {"L", "R"}. Define the additive function filter from $\mathcal{P}$(Sta + {"error"}) to $\mathcal{P}$(Sta) by filter (R) = {(r | Sta) | r ∈ R ∧ (r ∈ Sta) = true}. Furthermore abbreviate

condtrue = λsta.<Vcond(sta) → Scond(sta) → (Bcond(sta)) inR,
"error" inR, "error" inR, ⊥> and

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 dotted rectangles correspond 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.



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">

FIGURE 2

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↓2. Then a-col<occ§occ', "R"> =

filter {𝒯col[[syn at occ']] < > sta'↓1 | 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↓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"> U
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) $\mathcal{T}$col[[syn]] occ sta ↓2 pla ≠ ∅ ⇒ pla is a descendant of occ
(ii) $\mathcal{T}$col[[syn]] occ sta ↓2 <occ,"L"> = {sta}
(iii) $\mathcal{T}$col[[syn]] occ sta ↓2 <occ,"R"> = filter {$\mathcal{T}$col[[syn]] occ sta ↓1}

□

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

Proof of Properties Cb, Cc and Cd

The proof is by structural induction and we omit the suffix col. It is convenient to define a combinator

Δ ∈ G x R -t> R x A

by

g Δ r = r ∈ Sta → g(r | Sta), <r,⊥>

Then g(sta) = g Δ (sta inR) and $(g_1 * g_2)$ Δ r = $\langle g_2$ Δ $(g_1$ Δ r↓1) ↓1, g_2 Δ $(g_1$ Δ r↓1) ↓2 ⊔ g_1 Δ r↓2> as well as cond(g_1,g_2) Δ r↓2 = g_1 Δ (condtrue Δ r↓1) ↓2 ⊔ g_2 Δ (condfalse Δ r↓1) ↓2.

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

g[$\bar{g}$] = $\mathcal{T}$[[exp]] occ§<1> * cond($\mathcal{T}$[[cmd]] occ§<2> * $\bar{g}$, attach <occ,"R">)
iter = $\mathcal{T}$[[exp]] occ§<1> * condtrue * $\mathcal{T}$[[cmd]] occ§<2>
iter*^{0} = λsta.<sta inR, ⊥> and iter*$^{(n+1)}$ = iter * iter*^{n} =
iter*^{n} * iter (by * associative)
g_1 = λsta.$\mathcal{T}$[[exp]] occ§<1> sta
g_2 = λsta.$\mathcal{T}$[[cmd]] occ§<2> Δ (condtrue Δ ($\mathcal{T}$[[exp]] occ§<1> sta ↓1) ↓1)
and let pla be different from <occ,"R">.

Calculations show

$$(\lambda\bar{g}.g[\bar{g}])^{n+1}\ \bot\ (sta)\ \downarrow 2\ (pla) = g_1(sta)\ \downarrow 2\ (pla) \sqcup g_2(sta)\ \downarrow 2\ (pla) \sqcup (\lambda\bar{g}.g[\bar{g}])^{n}\ \bot\ \Delta\ (iter(sta)\ \downarrow 1)\ \downarrow 2(pla)$$

so that it can be proved (by induction in n) that

$$(\lambda\bar{g}.g[\bar{g}])^{n+1}\bot\ (sta)\ \downarrow 2\ (pla) = \bigsqcup_{m=0}^{n}\bigsqcup_{i=1}^{2} g_i\ \Delta\ (iter^{*m}(sta)\ \downarrow 1)\ \downarrow 2(pla)$$

Hence, $\mathcal{T}$[[WHILE exp DO cmd OD]] occ sta ↓2(pla)

$$= attach\langle occ,"L"\rangle\ sta\ \downarrow 2(pla) \sqcup \bigsqcup_{n=0}^{\infty} g_1\ \Delta\ (iter^{*n}(sta)\ \downarrow 1)\ \downarrow 2(pla) \sqcup \bigsqcup_{n=0}^{\infty} g_2\ \Delta\ (iter^{*n}(sta)\ \downarrow 1)\ \downarrow 2(pla)$$

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

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

The second step follows because

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

as can be shown by induction in n. The third step follows because filter $\{\bot\} = \emptyset$ and $(\lambda\bar{g}.g[\bar{g}])^{n}\ \bot$ sta ↓1 is ⊥ or $\mathcal{T}$[[WHILE exp DO cmd OD]] occ sta ↓1. The latter result is proved similarly to lemma 1 (or use lemma 1 and property Ca).

The proof of Cb is by cases of occ'. If occ' = < > the result follows by Cd, because $\mathcal{T}$[[syn]] occ" sta ↓1 is independent of occ". If occ' = <1>§occ" or occ' = <2>§occ" the result follows by the hypotheses of the structural induction, the above expression for $\mathcal{T}$[[WHILE exp DO cmd OD]] occ sta ↓2(pla) and the additivity of filter.

The proof of Cc is also by cases of occ'. Assume occ' = < > and consider the first result. It follows from

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

Next consider the second result. Abbreviate

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

$\mathcal{T}[\![\ \text{WHILE exp DO cmd OD}]\!]\ \text{occ sta}\downarrow 2\ \langle \text{occ}\S\langle 2\rangle,\text{"L"}\rangle =$
$$\bigcup_{n=0}^{\infty} \text{filter}\ \{\text{condtrue}\ \Delta\ r_n\downarrow 1\}\ \text{and}$$
$\mathcal{T}[\![\ \text{WHILE exp DO cmd OD}]\!]\ \text{occ sta}\downarrow 2\ \langle \text{occ}\S\langle 1\rangle,\text{"R"}\rangle =$
$$\bigcup_{n=0}^{\infty} \text{filter}\ \{r_n\}.$$

The result follows by the additivity of filter.

If occ' = <1>§occ" or occ' = <2>§occ" the proof is by cases of syn <u>at</u> occ'. In all cases the result follows from the induction hypothesis and "additivity" (i.e. if x = H(y) is to be proved then H is additive).

□

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

<u>Theorem 1</u> ("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 = $\mathcal{T}$col[[syn]] < > sta ↓2 be the result of data flow analysing syn.

If
- syn' is of the same category as syn <u>at</u> occ, and
- syn' behaves the same as syn <u>at</u> occ on each state (sta') possible before syn <u>at</u> 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">:
$\mathcal{T}$sto[[syn at occ']] (sta') = $\mathcal{T}$sto[[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 from syn at occ'§<3-i> = syn[occ ← syn'] at occ'§<3-i>.

To show P(occ') abbreviate

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

We first show sta ∈ a-col <occ'§<1>,"L"> ⇒ $(\lambda\bar{g}.g\text{-sto}[\bar{g}])^n \perp$ sta = $(\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp$ sta. The proof is by induction in n and since the result is trivial for n=0, consider the case n+1. Let sta ∈ a-col <occ'§<1>,"L"> so $\mathcal{T}$sto[[exp]] (sta) = $\mathcal{T}$sto[[exp']] (sta) by P(occ'§<1>). If the common value is ⊥ or "error" inR the result is immediate, so assume it is sta' inR. Then sta' ∈ a-col<occ'§<1>,"R"> follows by properties Ca and Cb. Unless Vcond(sta') = true and Scond(sta') ∈ {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') ↓1 | Sta is in a-col<occ'§<2>,"L"> by property Cc. Then $\mathcal{T}$sto[[cmd]] (Bcond(sta')) = $\mathcal{T}$sto[[cmd']] (Bcond(sta')) by P(occ'§<2>). Again the result is immediate unless the common value is sta" inR. From properties Ca, Cb and Cc we have sta" ∈ a-col <occ'§<1>,"L"> 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 <occ',"L"> so that sta $\in$ a-col <occ'§<1>,"L"> by property Cc. The above result then gives $\mathcal{T}$sto[[WHILE exp DO cmd OD]] (sta) = $\bigsqcup_{n=0}^{\infty}$ $(\lambda\bar{g}.g\text{-sto}[\bar{g}])^n \perp$ (sta) = $\bigsqcup_{n=0}^{\infty}$ $(\lambda\bar{g}.g'\text{-sto}[\bar{g}])^n \perp$ (sta) = $\mathcal{T}$sto[[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 formulae are "sound". Theorem 1 above expresses *strong* equivalence with respect to a store semantics (that 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 backward data flow information. An example is the tranformation 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 (by also data flow analysing the transformed program). In a special case we are able to obtain strong equivalence even when only the original program is data flow analysed.

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 it seems natural to 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 = Sta -c> (Out+{"error"}) and the obvious "final" (or initial [11]) continuation is λsta.sta ↓3 in(Out+{"error"}).

The future semantics is given by tables 3 and 5. Domain C = Sta -c> R is the domain of continuations. As in the previous section domain A = Pla -c> C is used to associate each program point with the desired information (here a continuation). Combinator * is associative and auxiliary functions cond' and ⊕ satisfy that 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). We shall use suffix fut for the future semantics.

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

Property Fa Let syn $\in$ Syn, occ $\in$ Occ be maximal and c $\in$ C. Then $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 1 = \mathcal{T}sto[[syn\ ]] \oplus c$. □

Proof of Property Fa is by (an omitted) structural induction. □

The second component of the semantic function applied to some continuation (i.e. $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 2$) 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.

Property Fb Let syn $\in$ Syn, occ $\in$ Occ be maximal, c $\in$ C and occ' $\in$ Occ point into syn and abbreviate a-fut = $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 2$. Then
a-fut<occ§occ', "L"> = $\mathcal{T}fut[[syn$ at occ' $]]$ < > (a-fut<occ§occ', "R">) $\downarrow$ □

Property Fc Let syn $\in$ Syn, occ $\in$ Occ be maximal, c $\in$ C and occ' $\in$ Occ point into syn and abbreviate a-fut = $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 2$.
If syn at occ' is WHILE exp DO cmd OD
then a-fut<occ§occ'§<2>, "R"> = a-fut<occ§occ', "L">
      a-fut<occ§occ'§<1>, "R"> = cond' (a-fut<occ§occ'§<2>, "L">,
            a-fut<occ§occ', "R">)
For the remaining constructs there are more or less similar properties. □

Property Fd Let syn $\in$ Syn, occ $\in$ Occ be maximal, c $\in$ C and pla $\in$ Pla
Then
(i) $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 2$ pla $\neq \perp \Rightarrow$ pla is a descendant of occ
(ii) $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 2$<occ, "R"> = c
(iii) $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 2$<occ, "L"> = $\mathcal{T}fut[[syn\ ]]$ occ c $\downarrow 1$ □

TABLE 5: Future Semantics

Domains

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

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

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

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

remaining domains as in tables 2 and 4 .

Combinator

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

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

Functions

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

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

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

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

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

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

and Vcond, Scond, Bcond are as in table 2.

$apply\ [[ope]] \in G$

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

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

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

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

Proof of Properties Fb, Fc and Fd is by structural induction and is omitted. □

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 $\mathcal{T}$sto[[syn']] ⊕ 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 = $\mathcal{T}$fut[[syn]] < > c ↓2 be the result of data flow analysing syn. If

- syn' is of the same category as syn at occ,

and

- for the continuation c' holding after syn at occ (c' = a-fut <occ, "R">) that syn' followed by c' is less defined than syn at occ followed by c'

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

Proof Let P(occ') mean that for c' = a-fut<occ', "R">:
$\mathcal{T}$sto[[syn at occ']] ⊕ c' ⊒ $\mathcal{T}$sto[[syn [occ ←syn'] at occ']] ⊕ c'.
The assumption is P(occ) and the result follows from P(< >) by property Fd. The proof consists in 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 ∈ {1,2} and syn[occ←syn'] at occ' is WHILE exp' DO cmd' OD and we have both P(occ'§<1>) and P(occ'§<2>).

Define two abbreviations

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

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-fut<occ', "R">. The proof is by induction in n and the result is easy for n = 0, so consider n + 1. By Fa, Fb and Fc we have

a-fut<occ'§<2>,"R"> = FIX($\lambda\bar{g}$.g-sto[$\bar{g}$]) $\oplus$ c' so
a-fut<occ'§<2>,"L"> = $\mathcal{T}$sto[[cmd]] $\oplus$ (FIX($\lambda\bar{g}$.g-sto[$\bar{g}$]) $\oplus$ c')
$\sqsupseteq$ $\mathcal{T}$sto[[cmd']] $\oplus$ (($\lambda\bar{g}$.g'-sto[$\bar{g}$])^n $\perp$ $\oplus$ c') by Fa, Fb, P(occ'§<2>)
and $\oplus$ continuous. Proceeding in this way a-fut<occ'§<1>,"R">
= cond'($\mathcal{T}$sto[[cmd]] $\oplus$ (FIX($\lambda\bar{g}$.g-sto[$\bar{g}$]) $\oplus$ c'), c')
$\sqsupseteq$ cond'($\mathcal{T}$sto[[cmd']] $\oplus$ (($\lambda\bar{g}$.g'-sto[$\bar{g}$])^n $\perp$ $\oplus$ c'), c') and
$\mathcal{T}$sto[[exp]] $\oplus$ cond'($\mathcal{T}$sto[[cmd]] $\oplus$ (FIX($\lambda\bar{g}$.g-sto[$\bar{g}$]) $\oplus$ c'), c')
$\sqsupseteq$ $\mathcal{T}$sto[[exp']] $\oplus$ cond'($\mathcal{T}$sto[[cmd']] $\oplus$ (($\lambda\bar{g}$.g'-sto[$\bar{g}$])^n $\perp$ $\oplus$ c'), c')
i.e. g-sto[FIX($\lambda\bar{g}$.g-sto[$\bar{g}$])] $\oplus$ c' $\sqsupseteq$ ($\lambda\bar{g}$.g'-sto[$\bar{g}$])$^{n+1}$ $\perp$ $\oplus$ c'.
Then $\mathcal{T}$sto[[WHILE exp DO cmd OD]] $\oplus$ c' = g-sto[FIX($\lambda\bar{g}$.g-sto[$\bar{g}$])] $\oplus$ c'
$\sqsupseteq$ $\bigsqcup_{n=0}^{\infty}$ (($\lambda\bar{g}$.g'-sto[$\bar{g}$])^n $\perp$ $\oplus$ c') = $\mathcal{T}$sto[[WHILE exp' DO cmd' OD]] $\oplus$ c'.

□

Even if we assume that (in the notation of the theorem) syn' followed by c' is equal to syn <u>at</u> 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 = λsta.sta ↓3 inR that simply emits the output. The continuation c' holding immediately before OD is $\mathcal{T}$sto[[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 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 analysed. Since syn = (syn[occ ← syn']) [occ ← syn <u>at</u> 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 = λsta. sta ↓3 inR), but it is slightly unsatisfactory that also the transformed program has to be data flow analysed.

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' = $\mathcal{T}$fut[[syn]] < > c ↓2 pla) then the set of states not leading to an error is {sta ∈ Sta | c'(sta) ≠ "error" 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 to be live at some program point (pla) then the continuation holding there (c' = $\mathcal{T}$fut[[syn]] < > c ↓2 pla) must produce the same output (c'(sta_1) = c'(sta_2)) for any two states differing only on that identifier (sta_1 ↓1 = sta_2 ↓1[sta_1 ↓1[[ide]] /ide] and sta_1 ↓i = sta_2 ↓i for i ≠ 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 analysed, 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 analysed. The liveness semantics (tables 3 and 6) operates in essentially the same way as the future semantics. The most interesting functions are assign[[ide]] and content[[ide]] . It is easy to see that * is associative.

Property La below expresses the connection between the store semantics and the first component of the liveness semantics. For this we need a predicate 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,i.e. sta_1 = <env_1, inp,out, tem> ⇒ ∃env_2:[sta_2 = <env_2, inp, out, tem> ∧ ide ∈ l ⇒ env_1[[ide]] = env_2[[ide]]].

TABLE 6: Liveness Semantics

Domains

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

$L = \mathcal{P}(Ide)$

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

remaining domains as in tables 2 and 4.

Combinator

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

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

Functions

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

$attach(pla) = \lambda l.\langle l,\ \bot[l/pla] \rangle$

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

$cond(g_1, g_2) = g_1 \sqcup g_2$

$apply[[ope]] \in G$

$apply[[ope]] = \lambda l.\langle l,\ \bot \rangle$

$assign[[ide]] \in G$

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

$content[[ide]] \in G$

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

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

Also define syn_1 to be <ll, lr> - related to syn_2 when r_i = Tsto[[syn_i]] sta_i satisfies that if sta_1 is ll-similar to sta_2 then $r_1 = r_2$ or $r_i = sta_i'$ inR with sta_1' lr-similar to sta_2'.

Property La Let syn ∈ Syn, occ ∈ Occ be maximal, lr ∈ L and ll = T liv[[syn]] occ lr ↓1. Then syn is <ll, lr> - related to syn.

□

Proof of Property La is 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 these 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 ↓2 be the result of data flow analysing 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 is 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 we clearly have that syn[occ←syn'] followed by c = λsta.sta ↓3 inR equals syn followed by c .

Hopefully 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 identifiers 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 $\langle l_1, l_2 \rangle$ - related). Perhaps the future semantics should be augmented with (suitable generalizations of) such information.

5. CONCLUSION

We have shown that it is possible to validate program transformations that exploit data flow information. We have aimed at using an abstract formulation of data flow information in order to factor out the details of approximate analyses. For the "forward" transformations this has been completely successful: By only data flow analysing the original program we obtained strong equivalence (theorem 1). For practical purposes the use of the collecting semantics can be replaced by a more approximate data flow analysis [9]. Theorem 1 on "forward" replacements can of course be combined with theorems 2 and 3 on "backward" replacements.

For the use of backward data flow information the abstract formulation (the future semantics) is less satisfactory since only weak equivalence is obtained. To obtain strong equivalence both the original and the transformed program must be data flow analysed. In the special case of transformations exploiting liveness information we were able to dispense with the data flow analysis of the transformed program. If this special case can be generalized it will be worth-while to characterize backward data flow analyses with respect to the abstract formulation (in the spirit of [2, 3, 9]).

Acknowledgement

Neil Jones, Brian Mayoh and Hanne Riis commented upon a previous version.

REFERENCES

[1] A.V. Aho and J.D. Ullman, Principles of Compiler Design, Addison-Wesley, London, 1977.

[2] P. Cousot and R. Cousot, Systematic design of program analysis frameworks, Proc. 6th ACM Symposium on Principles of Programming Languages, 1979, pp. 269-282.

[3] P. Cousot, Semantic foundations of program analysis, Program Flow Analysis: Theory and Applications, S.S. Muchnick and N.D. Jones, Eds., Prentice-Hall, New Jersey, 1981, pp. 303-342.

[4] S.L. Gerhart, Correctness-preserving program transformations, Proc. 2nd ACM Symposium on Principles of Programming Languages, 1975, pp. 54-66.

[5] G. Huet and B. Lang, Proving and applying program transformations expressed with second-order patterns, Acta Informat., 11 (1978), pp. 31-55.

[6] D.B. Loveman, Program improvement by source-to-source transformations, J. Assoc. Comput. Mach., 24 (1977), pp. 121-145.

[7] R. Milne and C. Strachey, A Theory of Programming Language Semantics, Chapman and Hall, London, 1976.

[8] R. Milner, Program semantics and mechanized proof, Foundations of Computer Science II, K.R. Apt and J.W. de Bakker, Eds., Mathematical Centre Tracts 82, Amsterdam, 1976, pp. 3-44.

[9] F. Nielson, A denotational framework for data flow analysis, Report No. PB-135, Aarhus University, Denmark, 1981.

[10] B.K. Rosen, Tree-manipulating systems and Church-Rosser theorems, J. Assoc. Comput. Mach., 20 (1973), pp. 160-187.

[11] J.E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, MA, 1977.