Control Structure Transformations in the Program Multiviewer

Jüri Kiho and Dmitri Brodski and Alexey Podlipski

Institute of Computer Science, University of Tartu
Liivi str.2, Tartu, Estonia
e-mail: kiho/dmitri_b/alekseip@cs.ut.ee

Abstract. A program multiviewer is under development at the University of Tartu. The most sophisticated program structure transformations in the multiviewer are related to processes of structure reducing and switching from textual representation to graphics and vice verse. For each kind of transformations a set of rules is given and application of the rules in corresponding algorithms is outlined.

1 Introduction

The idea of developing a program multiviewer [3] was proposed and corresponding project *AMADEUS*, as a students' group work, was started in 1995 at the University of Tartu. *AMADEUS* aims at combining both textual and graphic editors for handling source programs. A given program can be viewed and edited both as a text and as a two-dimensional diagram. For the latter, the sketchy representation technique [1] has been chosen. A sketchy program describes program structure by means of simple (nested) graphical blocks, called the sketches, containing textual elements – elementary declarations, statements, procedure calls and tests (Boolean expressions) from a textual host language, and determining control flow by graphical means instead of keywords. The layout of algorithms in this paper also demonstrates such representation techniques. *AMADEUS* is supposed to support a wide spectrum of host languages. At present stage it includes Pascal and C-family (incl. *JAVA*) programming languages.

Beside usual text/graphics processing facilities and some others [3], the most specific features of the program multiviewer are the transformations between textual and graphic (sketchy) views, and structure reduction of sketchy programs. The user may input, handle, develop and output programs in any of the two modes, possibly alternatively. For instance, he or she can start with a program text, then switch on to the graphical mode, continue editing, and finally switch back to the text mode etc. While developing software the user needs not care about the maximal simplicity of introduced control structures, since at any moment he or she can have sketchy program structure reduced by calling corresponding component (*REDUCER*) of the multiviewer.

In the current paper, the transformations performed in processes of switching from text to graphics (sketchifying), from graphics to text (normalizing) and structure simplification (reducing) are considered, and corresponding algorithms are outlined. Given a sketchy program, *NORMALIZER* uses certain transformations to represent it exclusively by sketches directly substitutable by some structural statements in target host language. In particular, normalizing for Pascal is considered. To perform its task, *SKETCHIFIER* applies also some structure transformations for solving **goto**s [4] inherited from the source textual program.

The program multiviewer can be used as a tool for software re-engineering, enabling to analyze and rewrite older software components. For instance, to get rid of **goto**s in a C-source, aiming at rewriting it in *JAVA*, the given C-program could be sketchified and then normalized for *JAVA*.

For each kind of transformations a set of rules is given. A transformation rule is represented as

$\Pi \Rightarrow \Pi'$, condition.

The rule states that Π and Π' are equivalent, if Π satisfies the given condition. At present stage, all transformations in *AMADEUS* are made under very strong equivalence. It means that given a sketchy program Π_1 and another sketchy program Π_2, obtained from the first one by applying some rules (for subsketches), both the static and dynamic order of host language elements for Π_1 and Π_2 are identical. (Dynamic orders are identical, if given the same arbitrary input the execution histories – sequences of executed host language elements – for Π_1 and Π_2 are identical.) To be precise, the orders may actually differ by the number of trivial tests 'TRUE?', and tests 't?', '$\neg t$?' should be considered as identical. The latter is reasonable, since evaluations of these tests may be thought of as $test(t)$: $compare(eval(t), TRUE)$ and $test(\neg t)$: $compare(eval(t), FALSE)$.

2 Notions and Notations

Control structures in the sketchy program are represented by *sketches*.

$sketch ::= simple\_sketch \mid loop\_sketch \mid switch\_sketch$

$loop\_sketch ::= free\_loop \mid head\_loop$

The sketch is a group (called *the segment*) of *members*.

$segment ::= empty \mid member \mid segment\, member$

$member ::= primitive \mid condition \mid arrow \mid sketch$

$arrow ::= strong\_arrow \mid weak\_arrow$

We use Π to denote segments, α to denote arbitrary arrows, and $|\Pi|$ is the number of members in Π.

Example 1. $\Pi = \Pi_1\alpha_1\Pi_2\alpha_2\Pi_3$ means that Π is a sequence of segments Π_1, Π_2, Π_3 separated by arrows α_1 and α_2; $|\Pi| = |\Pi_1| + |\Pi_2| + |\Pi_3| + 2$.

The primitive is a host language elementary statement (e.g. assignment, procedure call, declaration). A condition consists of a host language Boolean expression, followed by '?'. The conditions and arrows are a means for describing the control flow in the sketchy program [1]. A sketch may have member sketches i.e. *subsketches*. In the real sketchy programs the sketches are represented as (nested) two-dimensional diagrams [1], which are self-explaining and easily understandable. Although, for the purpose of theoretical analysis we use the following *scientific notation*.

$simple\_sketch ::= [^{nesting\_level} segment]$
$free\_loop ::= [:^{nesting\_level}\ segment\ :]$
$head\_loop ::= [*:^{nesting\_level}\ segment\ :]$
$strong\_arrow ::= \uparrow^{endpoint\_level}$
$weak\_arrow ::= \uparrow^{(endpoint\_level)}$
$nesting\_level ::= empty \mid unsigned\_integer$
$endpoint\_level ::= unsigned\_integer$

From the point of view of structure transformations, there is no need to introduce a notation for the switch sketch, since in this context each "case-cell" may be considered as a simple sketch (which cannot be opened or expanded during transformations). Similarly, here we explicitly describe only the body-segment of the head loop, omitting its heading. The minimal value of level is 1. There are some restrictions for endpoint levels (described below).

Example 2. A sketchy program (a simple sketch of level 1), which consists of a head loop containing, in turn, a free loop and a simple sketch:
$[[*: \Pi_1 [: \Pi_2 :][\Pi_3]\Pi_4 :]]$.

Any sketch (in the scientific notation) may be also considered as a sequence of lexems. An arbitrary subsequence of lexems is called the *fragment*. We use Φ to denote fragments.

Example 3. $\Pi = \Phi \uparrow^l \Phi'$ expresses the fact that Π contains the arrow $\uparrow^l$, not necessarily as a member of Π. For instance, $[:^3\ y := f(x)[^4 y < \epsilon ? \uparrow^3 x := y]\ :] = [:^l\ \Phi \uparrow^3 \Phi'\ :]$, where Φ is '$y := f(x)[^4 y < \epsilon ?$' and Φ' is '$x := y]$'.

A segment is called *arrowless* (*conditionless*), if it does not contain any member arrow (member condition). A condition $t?$ in a segment $\Pi = \Pi_1 t? \Pi_2$ is called *open*, if the tail segment Π_2 is arrowless. Otherwise, the condition is called *closed*. A segment Π is *condition-closed*, if it does not contain any open member conditions, and *linear* if it is arrowless and conditionless.

We use also the following generic notations:
$\{\Pi\}$ denotes a sketch of any kind, i.e. $[\Pi]$ or $[: \Pi :]$ or $[*: \Pi :]$;
$\{: \Pi\}$ denotes $[\Pi]$ or $[: \Pi :]$;
$\{*: \Pi\}$ denotes $[: \Pi :]$ or $[*: \Pi :]$.

In any (legal) sketch of level l, the endpoint level of a member arrow is greater than 0 or less than or equal to l, i.e $\{^l \Pi_1 \uparrow^e \Pi_2\}$ where $e < 1$ or $e > l$ is illegal. Moreover, the endpoint level of a weak arrow matches the level of an outer loop sketch, i.e. $[^l \Phi \uparrow^l \Phi']$ is illegal.

Semantics of arrows: execution of the strong arrow in $\{^l \Phi \uparrow^l \Phi'\}$ breaks the sketch; execution of the weak arrow in $\{*:^l\ \Phi \uparrow^{(l)} \Phi'\}$ continues the execution of the loop sketch (starts the next iteration).

Semantics of conditions: evaluation of a condition causes a (forward) jump iff it evaluates to FALSE.

Let us consider a condition $t?$ in a sketch Π. If t is closed, then $\Pi = \{\Pi_1\ t?\ \Pi_2 \alpha \Pi_3\}$ where Π_2 is arrowless, and evaluation t to FALSE jumps to Π_3. If $t?$ is open, then $\Pi = \{\!:^l\ \Pi_1\ t?\Pi_2\}$ executes as $\Pi' = \{\!:^l\ \Pi_1\ [\neg t?\uparrow^l]\Pi_2\}$ and $\Pi = [\!*:^l\ \Pi_1\ t?\Pi_2\ :\!]$ executes as $\Pi'' = [\!*:^l\ \Pi_1\ [\neg t?\uparrow^{(l)}]\Pi_2\ :\!]$. Note, that the condition $\neg t?$ in the inner sketch is closed.

Let Π be a segment. By Π^- we denote the segment obtained form Π by omitting its last member; if Π is empty Π^- is also empty.

For a segment Π we denote by $\Pi^{|l}$ the segment, obtained from Π by decreasing by 1 endpoint levels e of all arrows having $e \geq l$, and by $\Pi^{||l}$ the segment, obtained from Π by increasing by 1 endpoint levels e of all arrows having $e \geq l$. These operations are related to transformations involving opening/enclosing subsketches/subsegments.

3 Rules for Reducing

Here the list of reducing rules used by *REDUCER* is given. The reducing algorithm is straightforward: given a sketchy program, its structure is reduced until no more rules apply.

Each rule consists of a pair of sketches (separated by $\Rightarrow$) followed by sufficient conditions (if any) for equivalence of the two sketches. Most of the rules are proved in [3], the others may be proved similarly using the quasi-structured program model [2].

In certain cases some "unreducing" or *expanding* of sketches is desirable. The given set of reducing rules provides a basis for proving such expanding rules. To prove an expanding rule $\Pi_1 \Rightarrow \Pi_2$, it is sufficient to show that Π_2 is reducible to Π_1, by some reducing rule(s). In the following sections we introduce several expanding rules; for each rule we provide a reference to the corresponding reducing rule (cf.Ri^0) or rules, which can be used to prove the expanding rule.

The last four rules in the list are added as auxiliary ones. They do not describe actual reducing, but may be used in the process of transformations.

Remove empty simple sketch:

$R1^0$ $[\ ] \Rightarrow\ \ .$

Reduce sketch type:

$R2^0$ $[\!*:^l\ \Pi\ :\!] \Rightarrow [\!:^l\ \Pi \uparrow^{(l)} :\!]$, if the heading of the given head loop is empty;

$R3^0$ $[\!:^l\ \Pi \uparrow^l :\!] \Rightarrow [^l \Pi]$ if Π is condition-closed and does not contain any arrow $\uparrow^{(l)}$.

Reduce arrow:

$R4^0$ $[^l \Pi \uparrow^l] \Rightarrow [^l \Pi]$;

$R5^0$ $[\!:^l\ \Pi \uparrow^{(l)} :\!] \Rightarrow [\!:^l\ \Pi\ :\!]$, if Π is condition-closed;

$R6^0$ $[\!*:^l\ \Pi \uparrow^{(l)} :\!] \Rightarrow [\!*:^l\ \Pi\ :\!]$;

$R7^0$ $\{\Pi_1[\Pi_2\alpha_1]\alpha_2\Pi_3\} \Rightarrow \{\Pi_1[\Pi_2]\alpha_1\Pi_3\}$, if Π_2 is condition-closed;

$R8^0$ $[^l \Pi\{\Phi \uparrow^l \Phi'\}] \Rightarrow [^l \Pi\{\Phi \uparrow^{l+1} \Phi'\}]$;

$R9^0$ $\{\!*:^l\ \Pi\{\Phi \uparrow^{(l)} \Phi'\}\} \Rightarrow \{\!*:^l\ \Pi\{\Phi \uparrow^{l+1} \Phi'\}\}$;

$R10^0$ $[:^l\ [:\ \Phi \uparrow^{(l)} \Phi'\ :]\Pi\ :] \Rightarrow [:^l\ [:\ \Phi \uparrow^{(l+1)} \Phi'\ :]\Pi\ :]$.

Open the last subsketch:

$R11^0$ $[\Pi_1[^l\Pi_2]] \Rightarrow [\Pi_1\Pi_2^{|l}]$, if Π_1 is condition-closed or if Π_2^- does not contain member arrows;

$R12^0$ $P = \{*{:}\ \Pi_1[^l\Pi_2]\} \Rightarrow \{*{:}\ \Pi_1\Pi_2'^{|l} \uparrow^{(l)}\}$ where Π_2' is obtained from Π_2 by replacing all arrows $\uparrow^l$ by $\uparrow^{(l)}$, if Π_1 is condition-closed or if P is a head loop and Π_2^- does not contain member arrows.

Decrease inner subsketch:

$R13^0$ $\{\Pi_1[^l\Pi_2\Pi_3]\Pi_4\} \Rightarrow \{^l\Pi_1\Pi_2^{|l}[^l\Pi_3]\Pi_4\}$, if Π_1 and Π_2 are condition-closed and Π_2 does not contain any arrow $\uparrow^l$.

$R14^0$ $\{\Pi_1[^l\Pi_2\Pi_3]\Pi_4\} \Rightarrow \{^l\Pi_1[^l\Pi_2]\Pi_3^{|l}\Pi_4\}$, if Π_2 is condition-closed and $\Pi_2\Pi_3$ does not contain any arrow $\uparrow^l$ and either Π_1, Π_3 are condition closed or Π_3 is linear.

Open inner subsketch:

$R15^0$ $\{\Pi_1[^l\Pi_2]\Pi_3\} \Rightarrow \{^l\Pi_1\Pi_2^{|l}\Pi_3\}$, if Π_2 does not contain any arrow $\uparrow^l$ and either Π_1 and Π_2 are condition-closed or Π_2 is linear;

$R16^0$ $\{\Pi_1[^l\Pi_2]\alpha\Pi_3\} \Rightarrow \{\Pi_1\ \Pi_2'^{|l}\ \alpha\ \Pi_3\}$ where Π_2' is obtained from Π_2 by replacing all arrows $\uparrow^l$ by α, if Π_2 is linear; in particular, $\{^l\Pi_1[\Pi_2] \uparrow^l \Pi_3\} \Rightarrow \{^l\Pi_1\ \Pi_2^{|l} \uparrow^l \Pi_3\}$, if Π_2 is linear.

Reduce the first loop in a loop:

$R17^0$ $[:\ [:^l\ \Pi_1\ :]\Pi_2\ :] \Rightarrow [:\ [^l\Pi_1' \uparrow^{(l-1)}]\Pi_2\ :]$ where Π_1' is obtained from Π_1 by replacing all arrows $\uparrow^{(l)}$ by $\uparrow^{(l-1)}$.

Auxiliary rules for closing conditions and segments:

$R18^0$ $\{:^l\ \Pi_1 t?\ \Pi_2\} \Rightarrow \{:^l\ \Pi_1\neg t?\uparrow^l\ \Pi_2\}$, if Π_1 is condition-closed and Π_2 is arrowless;

$R19^0$ $[*:^l\ \Pi_1 t?\ \Pi_2\ :] \Rightarrow [*:^l\ \Pi_1\neg t?\uparrow^{(l)}\ \Pi_2\ :]$, if Π_1 is condition-closed and Π_2 is arrowless.

$R20^0$ $\{:^l\ \Pi\} \Rightarrow \{:^l\ \Pi^c\}$, where Π^c is condition-closed and is obtained from Π by negating all open conditions and inserting arrows $\uparrow^l$ after them;

$R21^0$ $[*:^l\ \Pi\ :] \Rightarrow [*:^l\ \Pi^c\ :]$, where Π^c is condition-closed and is obtained from Π by negating all open conditions and inserting arrows $\uparrow^{(l)}$ after them.

4 Solve goto

The problem of solving **goto**s arises in the process of sketchifying. Given a source textual program, *SKETCHIFIER* first builds the corresponding sketchy program, leaving **goto**s unresolved yet. The obtained sketchy program contains **goto**s and labels as sketch members (host language primitives). Attempts to solve **goto**s are made on the second stage of sketchifying, where a pair *goto* L and L is considered at a time. The program structure is transformed (if possible) to a form, where the given *goto* L may be replaced by an arrow, which transfers the control to the corresponding L.

Given a pair *goto* L and L, we denote:

$\Pi(g\_L)$ – a segment, containing *goto* L;

$\Pi(L)$ – a segment, containing the label L.

Definition 1. The minimal sketch containing both *goto* L and L is called the *cover* for the pair *goto* L and L.

Without loss of generality we may assume that **goto**s and labels are enclosed into trivial simple sketches $[gotoL]$ and $[L]$ (cf.$R15^0$). Therefore, the cover for a forward *goto* L has the form $\{\Pi_1\{\Pi_2(g\_L)\}\Pi_3\{\Pi_4(L)\}\Pi_5\}$ and the cover for a backward *goto* L has the form $\{\Pi_1\{\Pi_2(L)\}\Pi_3\{\Pi_4(g\_L)\}\Pi_5\}$.

Definition 2. Let $\Pi = \Phi L \Phi'$ be a segment containing a label L. Label L is *attainable* in Π if the fragment Φ consists of zero or more opening brackets '[' and '[:'.

An attainable label L could be moved to the first place in the segment $\Phi L \Phi'$, since the control transfer to the label performs in the same way both for $\Phi L \Phi'$ and $L \Phi \Phi'$.

4.1 Solve Forward goto

To solve a forward *goto* L in a sketchy program the latter must be given the form, where in the cover for *goto* L and L, $\{\Pi_1\{^l\Pi_2(g\_L)\}\Pi_3\{\Pi_4(L)\}\Pi_5\}$, Π_3 is empty and L is attainable in $\{\Pi_4(L)\}$. Thereafter the final solution is obtained by replacing the occurrence of *goto* L in $\Pi_2(g\_L)$ by the arrow $\uparrow^l$.

Cover transformations used for solving forward gotos:

$F1^0$ $\{\Pi_1\{^l\Pi_2(g\_L)\}\Pi_3\{\Pi_4(L)\}\Pi_5\} \Rightarrow \{\Pi_1[^l\{^{l+1}\Pi_2^{||l}(g\_L)\}\Pi_3^{||l}]\{\Pi_4(L)\}\Pi_5\}$, if Π_1 and Π_3 are condition-closed (cf. $R15^0$);

$F2^0$ $\{\Pi_1[^l\Pi_2(g\_L)]\Pi_3\{\Pi_4(L)\}\Pi_5\} \Rightarrow \{\Pi_1[^l\Pi_2(g\_L)\Pi_3^{||l}]\{\Pi_4(L)\}\Pi_5\}$, if Π_2 is condition-closed and Π_3 is linear;

$F3^0$ $\{\Pi_1'\Pi_1''\{^l\Pi_2(g\_L)\}\Pi_3[^l\Pi_4(L)]\Pi_5\} \Rightarrow \{\Pi_1'[^l\Pi_1''\{^{l+1}\Pi_2^{||l}(g\_L)\}\Pi_3^{||l}\Pi_4(L)]\Pi_5\}$, if Π_1' is condition closed and $\Pi_1''\Pi_3$ is condition-closed.

Note, that the expanded simple sketch in $F3^0$ becomes a new cover for the pair *goto* L and L.

The algorithm is outlined in Figure 1. The subfunction *newcover*() establishes new cover by $F3^0$ and so changes values for $\Pi_1 \dots \Pi_5$. The splitting operation $split(\Pi_1)$ determines the largest initial condition-closed part Π_1' of the segment Π_1 and assigns the rest of Π_1 to Π_1''.

4.2 Solve Backward goto

To solve a backward *goto* L in a sketchy program the latter must be given the form, where in the cover for *goto* L and L, $[:\ \Pi_1\{^l\Pi_2(L)\}\Pi_3\{\Pi_4(g\_L)\}\Pi_5\ :]$, Π_1 is empty and L is attainable in $\{\Pi_2(L)\}$. Thereafter the final solution is obtained by replacing the occurrence of *goto* L in $\Pi_4(g\_L)$ by the arrow $\uparrow^{(l)}$.

Cover transformations used for solving backward gotos:

$B1^0$ $[^l\Pi] \Rightarrow [:^l\ \Pi[\uparrow^l]\ :]$;

This rule is proved by the following equivalence transformations:

(a) $[^l\Pi] \Rightarrow [^l\Pi^c]$ (by $R20^0$);

$SolveForward(Cover)$
--- Given: a cover sketch for a pair *goto_L* and L,
$Cover = \{\Pi_1\{\Pi_2(g\_L)\}\Pi_3\{\Pi_4(L)\}\Pi_5\}$
--- Result: *goto* L resolved if possible (returns 1), otherwise returns 0

$Coversave := Cover$
--- loop
L is attainable in $\{\Pi_4\}$?
$\longleftarrow$ --- break loop
$\{\Pi_4(L)\}$ is not a simple sketch?
$Cover := Coversave$; --- restore
$\longleftarrow$ (0)
$\Pi_1'\Pi_1'' := split(\Pi_1)$
$\Pi_1''\Pi_3$ is condition-closed?
$Cover := newcower(Cover)$; --- by $F3^0$
--- continue loop:
$\leftarrow$----
$Cover := Coversave$; --- restore
$\longleftarrow$ (0)
Π_3 is empty?
$\longleftarrow$ (1)
Π_1 and Π_3 are condition-closed?
transform $Cover$ by $F1^0$
$\longleftarrow$ (1)
$\{\Pi_2(g\_L)\}$ is a simple sketch?
Π_2 is condition-closed and Π_3 is linear?
transform $Cover$ by $F2^0$
$\longleftarrow$ (1)
$Cover := Coversave$; --- restore
$\longleftarrow$ (0)

Fig. 1. Transformations for a forward goto

(b) $[^l\Pi^c] \Rightarrow [\!:^l\ \Pi^c \uparrow^l :\!]$ (cf.$R3^0$);
(c) $[\!:^l\ \Pi^c \uparrow^l :\!] \Rightarrow [\!:^l\ \Pi^c[\uparrow^l]\ :\!]$ (cf.$R15^0$).
Opening the conditions (cf. $R18^0$), which were closed at step (a), gives $[\!:^l\ \Pi[\uparrow^l]\ :\!]$.
$B2^0$ $[\!*\!: \Pi\ :\!] \Rightarrow [\!*\!: [\Pi]\ :\!]$ (cf. $R6^0$);
$B3^0$ $\{\Pi_1\{^l\Pi_2(L)\}\Pi_3\{\Pi_4(g\_L)\}\Pi_5'\Pi_5''\} \Rightarrow$
$\Rightarrow \{\Pi_1[^l\{\Pi_2^{\|l}(L)\}\Pi_3^{\|l}\{\Pi_4^{\|l}(g\_L)\}\Pi_5'^{\|l}]\Pi_5''\}$, if Π_1 and $\Pi_3\Pi_5'$ are condition-closed or $\Pi_3\Pi_5'$ is linear (cf. $R15^0$);
$B4^0$ $\{\Pi_1[^l\Pi_2(L)]\Pi_3\{\Pi_4(g\_L)\}\Pi_5'\Pi_5''\} \Rightarrow \{\Pi_1[^l\Pi_2^{\|l}(L)\Pi_3^{\|l}\{\Pi_4^{\|l}(g\_L)\}\Pi_5'^{\|l}]\Pi_5''\}$, if Π_2 is condition-closed and Π_2 does not contain any arrow $\uparrow^l$ and either Π_1 and $\Pi_3\Pi_5'$ are condition-closed or $\Pi_3\Pi_5'$ is linear (cf. $R14^0$).

The algorithm is outlined in Figure 2. The splitting function $propersplit(\Pi_5)$

determines the largest initial condition-closed part Π_5' of the segment Π_5, assigns the rest of Π_5 to Π_5'' and returns true iff Π_5' is terminated by an arrow.

5 Normalizing Sketchy Programs for Pascal

For a segment Π we denote by $P(\Pi)$ its Pascal equivalent.

In particular, $P(\mathit{Pascal\text{-}primitive}) = \mathit{Pascal\text{-}primitive}$ and for a sequence of linear segments $S_1, S_2, \ldots, S_n$, $P(S_1 S_2 \ldots S_n) = P(S_1); P(S_2); \ldots; P(S_n)$.

Also, for a linear segment S we define $\bar{P}(S)$: $\bar{P}(S) = BEGIN\ P(S)\ END$, if $|S| > 1$ and $\bar{P}(S) = P(S)$, otherwise.

A sketch Π is *normalized* (for Pascal) if it falls into one of the following categories:

1) $\Pi = [S]$, where S is linear; in this case $P(\Pi) = \mathrm{BEGIN}P(S)\mathrm{END}$;

2) $\Pi = [^l t_1?S_1 \uparrow^l t_2?S_2 \uparrow^l \ldots \uparrow^l t_n?S_n]$, where $S_1, S_2, \ldots S_n$ $(n \geq 1)$ are linear; in this case $P(\Pi)$=

IF t_1 THEN
  $\bar{P}(S_1)$
ELSE IF t_2 THEN
  $\bar{P}(S_2)$
...
ELSE IF t_n THEN (ELSE, if $t_n \Leftrightarrow$ TRUE)
  $\bar{P}(S_n)$

3) $\Pi = [:^l\ t?\uparrow^l\ S\ :]$, where S is linear; in this case $P(\Pi) =$WHILE NOT(t) DO$\bar{P}(S)$

4) $\Pi = [:^l\ St?:]$, where S is linear; in this case $P(\Pi) =$REPEAT $P(S)$ UNTIL NOT(t).

Normalizing transformations:

$N1^0$ $[:^l\ t?\uparrow^l\ \Pi\ :] \Rightarrow [:^l\ t?\uparrow^l\ [\Pi^{\|l+1}]\ :]$, if Π is condition-closed (cf. $R15^0$);

$N2^0$ $[:^l\ t?\Pi\ :] \Rightarrow [:^l\ \neg t?\uparrow^l\ \Pi\ :]$, if Π is arrowless (by $R18^0$);

$N3^0$ $[:^l\ \Pi t?\uparrow^l :] \Rightarrow [:^l\ \Pi\neg t?:]$, if Π is condition-closed (cf. $R18^0$);

$N4^0$ $[:^l\ \Pi t?:] \Rightarrow [:^l\ [(\Pi^c)^{\|l+1}]t?:]$, where Π^c is obtained from Π by negating all open conditions and inserting arrows $\uparrow^l$ after them (by $R20^0$, cf. $R18^0$, $R15^0$);

$N5^0$ $[:^l\ \Pi\ :] \Rightarrow [:^l\ \Pi\mathrm{TRUE}?:]$, by semantics of conditions (TRUE? may be inserted everywhere as a sketch member);

$N6^0$ $[^l \Pi_1 t?\Pi_2] \Rightarrow [^l \Pi_1[t?\Pi_2^{\|l}]]$, if Π_1 is condition-closed or if Π_2 does not contain arrows $\uparrow^l$ (cf. $R11^0$);

$N7^0$ $[^l \Pi_1 \alpha \Pi_2] \Rightarrow [^l[\Pi_1^{\|l}[\alpha]]\Pi_2]$ if Π_1 is arrowless (cf. $R11^0$, $R15^0$);

$N8^0$ $[^l \Pi_1 \uparrow^e \Pi_2] \Rightarrow [^l \Pi_1[\uparrow^e] \uparrow^l \Pi_2]$ (cf. $R7^0$, $R1^0$).

Transformations $N1^0 \ldots N7^0$ are used to normalize free loops and segments in simple sketches (see Figure 3). The subfunction $tru(\Pi)$ in $NormalizeSegment()$ inserts condition TRUE? in front of every member arrow in Π, which is not preceded by a condition. In some cases, occurrence of such arrows informs about unreachable subsegments: in $\{\Pi_1 \alpha \Pi_2\}$ the tail segment Π_2 is unreachable if Π_1 is condition-closed (not proved here).

$SolveBackward(Cover)$
- - - Given: a cover sketch for a pair $goto\_L$ and L,
$Cover = \{\Pi_1\{^l\Pi_2(L)\}\Pi_3\{\Pi_4(g\_L)\}\Pi_5\}$
- - - Result: *goto* L resolved if possible (returns 1), otherwise returns 0

$Coversave := Cover$

$Cover$ is a head loop?
transform $Cover$ by $B2^0$

L is attainable in $\{\Pi_2\}$?

$\{\Pi_2(L)\}$ is not a simple sketch?

Π_2 is not condition-closed or contains $\uparrow^l$?

Π_3 is linear?

Π_1 is not condition-closed?

Π_3 is condition-closed?

$\neg propersplit(\Pi_5)$?

transform $Cover$ by $B4^0$

Π_1 is empty?

$Cover$ is a simple sketch?
transform $Cover$ by $B1^0$

(1)

Π_3 is linear?

Π_1 is not condition-closed?

Π_3 is condition-closed?

$\neg propersplit(\Pi_5)$?

transform $Cover$ by $B3^0$
transform $Cover$ by $B1^0$
(1)

$Cover := Coversave$; - - - restore
(0)

Fig. 2. Transformations for a backward goto

$NormalizeFreeLoop(\Pi)$
--- Given: a free loop $\Pi = [^l\ \Pi_0\ ]$
--- Result: Π is normalized

(Π_0 starts with $t?{\uparrow}^l$, i.e. $\Pi_0 = t?{\uparrow}^l\ \Pi_1$)**?**
Π_1 is linear**?**
Π_1 is condition-closed**?**
transform Π by $N1^0$
(Π_0 starts with $t?$, i.e. $\Pi_0 = t?\Pi_1$)**?**
Π_1 is linear**?**
transform Π by $N2^0$
(Π_0 ends with $t?$, i.e. $\Pi_0 = \Pi_1 t?$)**?**
(Π_0 ends with $t?{\uparrow}^l$, i.e. $\Pi_0 = \Pi_1 t?{\uparrow}^l$)**?**
Π_1 is condition-closed**?**
transform Π by $N3^0$
transform Π by $N5^0$; transform Π by $N4^0$
Π_1 is not linear**?**
transform Π by $N4^0$

$NormalizeSegment(\Pi)$
--- Given: a segment Π of level l
--- Result: returns corresponding normalized segment Π' such that $[^l \Pi']$ is normalized

Π is linear**?**
(Π)
let $\Pi = \Pi_1\Pi_2$, where Π_1 is the maximal conditionless initial part in Π
$\Pi := tru(\Pi_1)\Pi_2$
the first member in Π is not a condition**?**
let $\Pi = \Pi_1 t?\Pi_2$, where Π_1 is linear (and nonempty)
$(\Pi_1[t?\Pi_2])$ --- $N6^0$
--- the first member in Π is a condition
let $\Pi = t?\Pi_1\Pi_2$, where Π_1 is the maximal linear part after the condition in Π
Π_2 is empty**?**
(Π)
$(\Pi_2 = \alpha t'?\Pi_2')$**?**
$(t?\Pi_1\alpha\ NormalizeSegment(t'?\Pi_2'\ ))$
t is open in Π**?**
$(t?\Pi_1[t'?\Pi_2'^{\,|||l}]\ )$ --- $N6^0$
let $\Pi = t?\Pi_1 t'?\Pi_2\alpha\Pi_3$, where Π_1 is linear and Π_2 is arrowless
$(NormalizeSegment([t?\Pi_1^{|||l} t'?\Pi_2^{|||l}[\alpha]]\Pi_3\ ))$ --- $N7^0$

Fig. 3. Normalizing transformations

Normalizing a given Pascal-based sketchy program consists of the following steps. First, arrows are shortened (by $R8^0$) as much as possible and then (applying $N8^0$) every prolonged member arrow α is replaced by two members, $[\alpha]$ and $\uparrow^l$, where l is the level of the sketch containing α as a member (cf.R) (if α is not already the single member of a simple sketch). Trivial sketches $[\alpha]$ are called the *hiding sketches*. After that, all free loops are normalized by $NormalizeFreeLoop()$. Since these transformations may introduce new prolonged arrows (as members of simple sketches), the first step to shorten arrows and "hide" prolonged arrows is performed again. Finally, simple sketches $[\Pi]$ (except hiding sketches) are normalized by $NormalizeSegment(\Pi)$ until all nontrivial simple sketches are normalized.

To obtain corresponding Pascal-code, the hiding sketches are replaced by **goto**-statements (also inserting necessary labels) and every sketch Π is recursively replaced by $P(\Pi)$.

6 Further Research

The given set of reducing rules is obviously implementation-oriented. It contains some dependent rules, which can be derived from the others, and is probably incomplete. Because reducing rules serve as proofs for all expanding rules, it would be of theoretical interest to find out a basic, complete and pair-wise independent set of reducing rules.

One way to make transformation algorithms more powerful is to use more widely condition-closing operations $R18^0 \ldots R21^0$. Also, the problem of solving **goto**s may be given a better solution, aimed at minimizing the number of unresolved **goto**s.

References

1. Kiho, J.: Diagramming techniques and sketchy programming. Proc. 2nd Symposium on Programming Languages and Software Tools Pirkkala Finland 1991 78–82.
2. Kiho, J.: Quasi-structured programs. Proc. 4th Symposium on Programming Languages and Software Tools Visegrád Hungary 1995 June 8-14 67–78
3. Kiho, J., Oja, A.: Designing a Program Multiviewer. Proc. 7th Nordic Workshop on Programming Theory Göteborg 1995 Nov. 1-3 165–184
4. Ramshaw, L.: Eliminating goto's while preserving program structure. J.ACM **35** 4 1988 893–920

This article was processed using the LaTeX macro package with LLNCS style