# Catamorphic Approach to Program Analysis

Mizuhito OGAWA[1,2], Zhenjiang HU[1,2], Isao SASANO[1], Masato TAKEICHI[1]

[1] The University of Tokyo
[2] PRESTO 21, Japan Science and Technology Corporation
{mizuhito,hu,sasano,takeichi}@ipl.t.u-tokyo.ac.jp

**Abstract.** This paper proposes a new framework for program analysis that regards them as maximum marking problems: mark the codes of a program under a certain condition such that the number of marked nodes is maximum. We show that if one can develop an efficient *checking* program (in terms of a finite *catamorphism*) whether marked programs are correctly marked, then an efficient and incremental *marking* algorithm to analyze control flow of structured programs (say in C or Java with limited numbers of `gotos`) is obtained *for free.* To describe catamorphisms on control flow graphs, we present an algebraic construction, called *SP Term*, for graphs with bounded tree width. The transformation to SP Terms is done efficiently in linear time. We demonstrate that our framework is powerful to describe various kinds of control flow analyses. Especially, for analyses in which some optimality is required (or expected), our approach has a big advantage.

**Keywords**: Catamorphism, Control Flow Graphs, Control Flow Analysis, Functional Programming, Linear Time Algorithm, Program Optimization, Bounded Tree Width

## 1 Introduction

While program transformation plays an important role in optimizing compiler and software engineering, implementing program transformation requires various kinds of program analyses to extract program fragments satisfying a certain property; these fragments will then be replaced by equivalent but more efficient constructs.

As a simple example, consider the program in Fig. 1 from which we want to eliminate all dead codes, which assign a value that will be never used. Here, the task of the analysis is to find *all* possible dead codes; for this example, the assignments in the marked lines are dead codes which may be replaced later by the skip statement.

It is not so trivial to develop an *efficient* and *incremental* analysis algorithm

$$marking :: Prog \rightarrow Prog^*$$

for transforming the initial program to that with marks on *all* dead codes. But, it turns out to be rather easier if we are just to design an efficient algorithm

$$checking :: Prog^* \rightarrow Bool$$

for *checking* whether a marked programs have correct marking so that marked assignments are dead codes.

In this paper, we show the derivation of control flow analyses such that if one can develop an efficient *checking* program in terms of finite *catamorphisms*, then an efficient *marking* algorithm is obtained for free.

The first interesting point of our approach is that *many program analyses can be considered as a maximum marking problems*; mark the codes of a control flow graph under a certain condition such that the number of marked nodes is maximum. In fact, our approach is greatly inspired by the optimization theorem [31, 30, 6] for solving maximum marking problems, which says that marking

```
read X;
while X > 1 do
Z := X * 3 + 1;
C := X % 2;
if C = 0
   then
   X := X / 2;
   else
   X := Z;
fi;
Z := X * 2;          *  // dead code
od;
write X;
```

**Fig. 1.** A simple program piece where all dead codes are marked

as many nodes as possible in a data structure can be implemented efficiently in linear time, if the condition for such marking is described in terms of an efficient finite catamorphism.

One major difficulty in applying the optimization theorem to control flow analyses, is the discovery of an algebraic construction of control flow graphs such that properties are described as catamorphisms. It is well known that general graphs are difficult to define constructively [19], but is less known that

> *Control flow graphs of programs like C or Pascal are rather well-structured; that is, their tree widths are at most 6 if the program does not contains* `goto` *[37]. The control flow graphs of Java programs has unbounded tree width in general, but the empirical study shows that most Java programs has tree width at most* 3 *[20].*

This hints us that we may solve the problem of program analyses as a maximum marking problems, providing that the tree width of control flow graphs is bounded.

## Our Contributions

The purpose of this paper is to give a new framework for program analyses by treating them as maximum marking problems. Our main contributions are summarized as follows.

– We provide a new abstract way of using *functions* (instead of logical formulae) to specify control flow analyses so that one needs only to develop the *checking* algorithms in terms of finite catamorphisms. This is much easier than developing the *marking* algorithms (program analyses). Our approach guarantees to produce *efficient* and *incremental* algorithms for program analyses, provided that the user's specifications are efficient.
– We present an algebraic construction, called *SP Term*, for control flow graphs with bounded tree width. This enables us to define catamorphisms on control flow graphs. In addition, we show that the transformation to SP Terms is done efficient in linear time.
– We demonstrate that our framework is powerful to describe various kinds of control flow analyses; forward and backward analyses or even complicated combination of both. Especially, for analyses providing optimality like "all dead code detection", our approach has a big advantage in obtaining efficient algorithms due to the optimization theorem in [31, 30, 6].

The rest of this paper is organized as follows. We start by explaining our basic idea though an example of the dead-code analysis on a simple flowchart program in Section 2. Then in Section 3, we step to generalize our idea to analyze control graphs. The example class is *series-parallel graphs*, which represents the simplest nontrivial control structures. We give a formal study of our approach in Section 4, discuss related work in Section 5, and draw conclusion and highlight future work in Section 6. Throughout the paper, we will use Haskell-like notations.

## 2 Analyzing Flowchart Programs: A Tour

We briefly explain our idea through an example of the dead code analysis. To simplify the construction of control flow graphs, we use the following flowchart scheme. At the end of the whole program, the `exit` sentence is added.

| | | |
|---|---|---|
| $Prog :=$ | $x := e$ | assignment |
| $\mid$ | $\texttt{read}\ x$ | read sentence |
| $\mid$ | $\texttt{write}\ x$ | write sentence |
| $\mid$ | $Prog;\ Prog$ | sequence |
| $\mid$ | $\textbf{if}\ e\ \textbf{then}\ Prog\ \textbf{else}\ Prog\ \textbf{fi}$ | conditional sentence |
| $\mid$ | $\textbf{while}\ e\ \textbf{do}\ Prog\ \textbf{od}$ | while loop |

The point of our approach is to regard a control flow analysis as a maximum marking problem. So dead code analysis requires marking the assignments in a program such that (1) the marked ones are dead code, and (2) the number of marks is maximum. From the optimization theorem for solving the maximum marking problem mechanically [31, 30] (see Section A), we can reduce it to a simpler problem of checking whether a program is correctly marked in the sense that all the marked assignments are really dead codes.

It should be noted that the condition (2) defines sort of optimality; finding as many dead codes as possible. Compared to just detecting a dead code, finding optimal solution generally makes analysis more difficult to implement efficiently. In our context, obtaining such optimality is the direct consequence of the optimization theorem [31, 30, 6].

### 2.1 Checking Dead Codes: Specification

The property whether the marked assignments in a program are really dead is recursively defined over the language constructs. The function *checking* checks that all the variable that is defined in marked assignment node are dead codes, i.e., an assignment node is not marked unless it is really a dead code.

$$
\begin{array}{ll}
\mathit{checking} & ::\ \mathit{Prog}^* \rightarrow \mathit{Bool} \\
\mathit{checking}\ p & = \mathit{dead}\ p\ \ [\,] \\
\\
\mathit{dead}\ n@(x := e)\ vs & = \text{if } \mathit{marked}\ n \text{ then } x \notin vs \text{ else } \mathit{True} \\
\mathit{dead}\ n@(\texttt{read}\ x)\ vs & = \text{if } \mathit{marked}\ n \text{ then } x \notin vs \text{ else } \mathit{True} \\
\mathit{dead}\ n@(\texttt{write}\ x)\ vs & = \mathit{not}\ (\mathit{marked}\ n) \\
\mathit{dead}\ (p_1; p_2)\ vs & = \mathit{dead}\ p_1\ (\mathit{live}\ p_2\ vs)\ \wedge\ \mathit{dead}\ p_2\ vs \\
\mathit{dead}\ (\textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \textbf{fi})\ vs & = \mathit{dead}\ p_1\ vs\ \wedge\ \mathit{dead}\ p_2\ vs \\
\mathit{dead}\ (\textbf{while}\ e\ \textbf{do}\ p\ \textbf{od})\ vs & = \mathit{dead}\ p\ (\mathit{live}\ (\textbf{while}\ e\ \textbf{do}\ p\ \textbf{od})\ vs)
\end{array}
$$

The function *dead* takes a marked program $p$ and a set of variables $vs$ that may be used later, and returns *True* if each marked assignment in the marked program $p$ is really a dead code. For the assignment $x := e$, it means that it will never be used later. For the sequence $p_1; p_2$, it first checks whether the assignments in $p_2$ may be used in $vs$, and then checks $p_1$ whether the assignments in $p_1$ may be used in either $p_2$ or $vs$, which is computed by the function *live*. We omit explanation for other cases.

The function *live* takes a program $p$ and a set of variables $vs$ that may be used after $p$, and returns a set of variables that are alive in the start point of $p$.

$$
\begin{array}{ll}
\mathit{live}\ (x := e)\ vs & = (vs \setminus \{x\})\ \cup\ FV(e) \\
\mathit{live}\ (\texttt{read}\ x)\ vs & = vs \setminus \{x\} \\
\mathit{live}\ (\texttt{write}\ x)\ vs & = vs\ \cup\ \{x\} \\
\mathit{live}\ (p_1; p_2)\ \ vs & = \mathit{live}\ p_1\ (\mathit{live}\ p_2\ vs) \\
\mathit{live}\ (\textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \textbf{fi})\ vs & = FV(e)\ \cup\ \mathit{live}\ p_1\ vs\ \cup\ \mathit{live}\ p_2\ vs \\
\mathit{live}\ (\textbf{while}\ e\ \textbf{do}\ p\ \textbf{od})\ vs & = FV(e)\ \cup\ vs\ \cup\ \mathit{live}\ p\ (FV(e)\ \cup\ vs)
\end{array}
$$

For the case of assignment $x := e$, it firstly deletes $x$ from $vs$, because the value of $x$ is changed here, and secondly adds used variables in the expression $e$, i.e., free variables $FV(e)$ in $e$. For the case of sequential $p_1; p_2$, it first adds used variables in $p_2$, and then adds used variables in $p_1$. We omit explanation for other cases.

Note that our *checking* is a straightforward recursive definition.

## 2.2 Marking All Dead Codes: Analysis

From the above definition of *checking*, we can systematically develop a definition of function *marking*, which detects dead codes as many as possible. The key is the *optimization theorem* for solving the *maximum marking problem* [31, 30, 6]. If function *checking* is a finite mutumorphism (mutually recursive version of catamorphism), it is straightforward to apply the theorem. Otherwise, we need to decompose *checking* into a composition $f \circ g$ such that $f$ is a finite mutumorphism.

Informally, a *catamorphism* on $Prog^*$ is a function defined in a *bottom-up* manner on the structure of $Prog^*$. More specifically, a function $f$ is said to be catamorphism if it is defined in the following form:

$$\begin{array}{ll}
f\ (x := e) & = g_a\ x\ e \\
f\ (\texttt{read}\ x) & = g_r\ x \\
f\ (\texttt{write}\ x) & = g_t\ x \\
f\ (p_1; p_2) & = g_p\ (f\ p_1)\ (f\ p_2) \\
f\ (\textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \textbf{fi}) & = g_i\ e\ (f\ p_1)\ (f\ p_2) \\
f\ (\textbf{while}\ e\ \textbf{do}\ p\ \textbf{od}) & = g_w\ e\ (f\ p)
\end{array}$$

where $g_a, g_r, g_t, g_p, g_i, g_w$ are any user defined functions that do not call $f$. If the range of function $f$ is finite, we say that the catamorphism is finite. *Mutumorphism* is a natural extension of catamorphism where a set of functions $f_1, \ldots, f_n$ are mutually defined in a bottom-up manner.

Although *checking* (in Section 2.1) is not a finite catamorphic form, we can apply the decomposition transformation to decompose it into a composition:

$$checking \;=\; checking' \;\circ\; pre$$

such that $checking'$ is a finite mutumorphism. For our example, $pre$ turns out to be a function to precompute the values of $vs$ and attaches them to each assignment node in program $p$ as follows:

$$\begin{array}{l}
pre :: Prog^* \to Prog'^* \\
pre\ p = addLive\ p\ [\,]
\end{array}$$

where

$$\begin{array}{ll}
addLive\ n@(x := e)\ vs & = (n, vs) \\
addLive\ n@(\texttt{read}\ x)\ vs & = (n, vs) \\
addLive\ n@(\texttt{write}\ x) & = (n, vs) \\
addLive\ (p_1; p_2)\ vs & = addLive\ p_1\ (live\ p_2\ vs)\ ;\ addLive\ p_2\ vs \\
addLive\ (\textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \textbf{fi})\ vs & = \textbf{if}\ e\ \textbf{then}\ addLive\ p_1\ vs\ \textbf{else}\ addLive\ p_2\ vs\ \textbf{fi} \\
addLive\ (\textbf{while}\ e\ \textbf{do}\ p\ \textbf{od})\ vs & = \textbf{while}\ e\ \textbf{do}\ addLive\ p\ (live\ (\textbf{while}\ e\ \textbf{do}\ p\ \textbf{od})\ vs)\ \textbf{od}
\end{array}$$

Two remarks are worth making on efficient computation of *pre*. First, *addLive* calls *live* in the definition, and both of them traverse the program structure. In fact, we can apply the tupling calculation [21] to eliminate this multiple traversals by defining a new function $tup\ p\ vs = (addLive\ p\ vs, live\ p\ vs)$. Second, *pre* contains many operations on sets, which are efficiently implemented by using bit vectors as in many compiler textbooks.

Consequently, $checking'$ is simplified to be the following finite catamorphism.

$$
\begin{array}{ll}
checking'\ n@(x := e,\ vs) & = \text{if } marked\, n \text{ then } x \notin vs \text{ else } True \\
checking'\ n@(\texttt{read}\ x,\ vs) & = \text{if } marked\, n \text{ then } x \notin vs \text{ else } True \\
checking'\ n@(\texttt{write}\ x,\ vs) & = not\ (marked\, n) \\
checking'\ (p_1; p_2) & = checking'\ p_1\ \wedge\ checking'\ p_2 \\
checking'\ (\textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \textbf{fi}) & = checking'\ p_1\ \wedge\ checking'\ p_2 \\
checking'\ (\textbf{while}\ e\ \textbf{do}\ p\ \textbf{od}) & = checking'\ p
\end{array}
$$

Applying the optimization theorem [31] (see Section A) immediately gives the following efficient solution:

```
$marking\ ::\ Prog \to Prog^*$
$marking\ p = opt_{Prog'}\ accept\ \phi_a\ \phi_r\ \phi_t\ \phi_s\ \phi_i\ \phi_w\ (pre\ p)$
  **where**
    $accept = id$
    $\phi_a\ v@((x, e), vs) = \text{if } marked\, v \text{ then } x \notin vs \text{ else } True$
    $\phi_r\ v@(x, vs) = \text{if } marked\, v \text{ then } x \notin vs \text{ else } True$
    $\phi_t\ v@(x, vs) = not\ (marked\, v)$
    $\phi_s\ c1\ c2 = c1\ \wedge\ c2$
    $\phi_i\ e\ c1\ c2 = c1\ \wedge\ c2$
    $\phi_w\ e\ c = c$
```

where the function $opt_{Prog'}$ is a straightforward modification of the generic function *opt* in [31].

Note that the above *marking* function is an executable Haskell program. After the computation of *pre p*, which is efficiently implemented by bit vector techniques and a tupling transformation on *addLive* and *live*, *marking* is a linear time algorithm in the size of the input program.

## 3 Analyzing Series-Parallel Graphs: Towards a General Study

For flowchart programs, it is quite easy to write down the specification of the checking directly on program syntax. However, this becomes intricate if we consider more complex programming languages. To be more general and language-independent, we will show that the catamorphic framework in Section 2 can be applied on control flow graphs by the use of the algebraic construction, called SP Term.

Before entering to the general and formal study in Section 4, this section introduces the simple case study, *series-parallel graph* [35]. This class of graphs corresponds to the control flow graphs of structured programs (in strict sense) i.e., programs consist of single-entry and single-exit blocks.

### 3.1 SP Terms for Representing Structured Control Flow Graphs

#### Algebraic Construction of Series-Parallel Graphs

A control flow graphs of flowchart scheme are *series-parallel graphs* [35], which are graphs with tree width at most 2. We give the transformation from the control flow graph to a *SP Term*. Note that this definition is somewhat simplified compared to the definition in Section 4.3 for general cases.

**Definition 1.** An SP Term is a pair of a ground term $t$ and a tuple $(l_1, l_2)$ of labels, defined as the following.[1]

[1] In Section 4.3, $e^+$, $e^-$, $S$, and $P$ are denoted by $e_2(1, 2)$, $e_2(2, 1)$, $S_2$, and $P_2$, respectively.

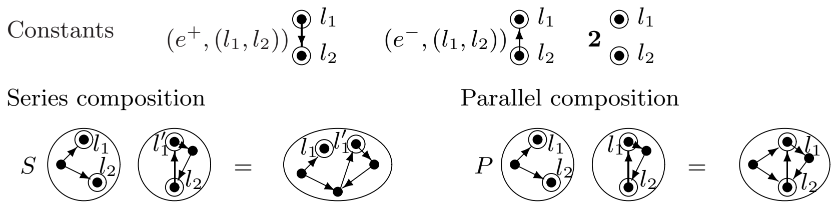


**Fig. 2.** Interpretation of $e^+$, $e^-$, **2**, $S$, and $P$.

$$
\begin{aligned}
SP_2 :=\ & (e^+,\ (l_1, l_2)) \\
\mid\ & (e^-,\ (l_1, l_2)) \\
\mid\ & (\mathbf{2},\ (l_1, l_2)) \\
\mid\ & S\ SP_2\ SP_2 \\
\mid\ & P\ SP_2\ SP_2
\end{aligned}
$$

■

An SP Term is interpreted as a pair of a 2-terminal series-parallel digraph and a tuple of 2-labels; a 2-terminal digraph is a digraph with a tuple consists of two vertices. Labels $(l_1, l_2)$ are interpreted as the identifiers of vertices. The interpretation of each function symbol and constant is described in Fig. 2; a terminal is presented as the double circle, and labels are associated to terminals.

**Definition 2.** Let $match(l, l')$ be the function that returns *true* if either $l = l'$, $l = *$, or $l' = *$ (i.e., accept the special label $*$ as a wild card). The series composition $S\ (t_1, (l_1, l_2))\ (t_2, (l'_1, l'_2))$ fuses the second terminals in $t_1$ and $t_2$ if $match(l_2, l'_2)$, and renumber the first terminal in $t_1$ as the first and the first terminal in $t_2$ as the second. The parallel composition $P\ (t_1, (l_1, l_2))\ (t_2, (l'_1, l'_2))$ fuses each first and second terminal in $t_1$ and $t_2$ if $match(l_1, l'_1)$ and $match(l_2, l'_2)$. ■

We prepare the function $chT$ that exchanges the order of the two terminals of a graph.

$$
\begin{aligned}
& chT && ::\ SP_2 \to SP_2 \\
& chT\ (e^+,\ (l_1, l_2)) && = (e^-,\ (l_2, l_1)) \\
& chT\ (e^-,\ (l_1, l_2)) && = (e^+,\ (l_2, l_1)) \\
& chT\ (\mathbf{2},\ (l_1, l_2)) && = (\mathbf{2},\ (l_2, l_1)) \\
& chT\ (S\ x\ y) && = S\ y\ x \\
& chT\ (P\ x\ y) && = P\ (chT\ x)\ (chT\ y)
\end{aligned}
$$

**Translation**

Before defining the translation *trans* to an SP Term, as a preprocessing, we add `return` *var* (where *var* presents the output) at the end of a program, and give labels to identify vertices in a control flow graph (except for `then`, `else`, `fi`, and `od`). We denote the set of such labeled programs by *LProg*.

The implementation *trans* of the transformation is given below.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{trans}\ ::\ \mathit{LProg} \rightarrow \mathit{SP}_2} \\
\mathit{trans}\ (\mathtt{l} : x := e) & = (e^+, (\mathtt{l}, *)) \\
\mathit{trans}\ (\mathtt{l} : \mathtt{read}\ x) & = (e^+, (\mathtt{l}, *)) \\
\mathit{trans}\ (\mathtt{l} : \mathtt{write}\ x) & = (e^+, (\mathtt{l}, *)) \\
\mathit{trans}\ (p_1; p_2) & = S\ (\mathit{trans}\ p_1)\ (\mathit{chT}\ (\mathit{trans}\ p_2)) \\
\mathit{trans}\ (\mathtt{l} : \mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2\ \mathbf{fi}) & = P\ (S\ (e^+, (\mathtt{l}, *))\ (\mathit{chT}\ (\mathit{trans}\ p_1))) \\
 & \quad\ (S\ (e^+, (\mathtt{l}, *))\ (\mathit{chT}\ (\mathit{trans}\ p_2))) \\
\mathit{trans}\ (\mathtt{l} : \mathbf{while}\ e\ \mathbf{do}\ p\ \mathbf{od}) & = P\ (e^+, (\mathtt{l}, *)) \\
 & \quad\ (S\ (P\ (e^+, (l, *))\ (\mathit{chT}\ (\mathit{trans}\ p))) \\
 & \qquad (\mathbf{2}, (*, \mathtt{l} + \mathtt{1})))
\end{array}
$$

For instance, the translation of **while**-sentence proceeds as in Fig. 3. Intuition behind the wild character label "`*`" is; for each fragment of a program, the first label in the tuple of an SP Term denotes the entry of the fragment, and the second label, which is always "`*`" during transformation, denotes the next control point. Note that each program fragment has the unique node with "$*$". At the end, `*` is replaced with the label for the `exit` sentence, i.e., the end of the program.

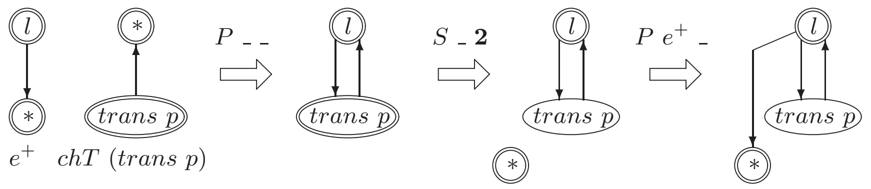
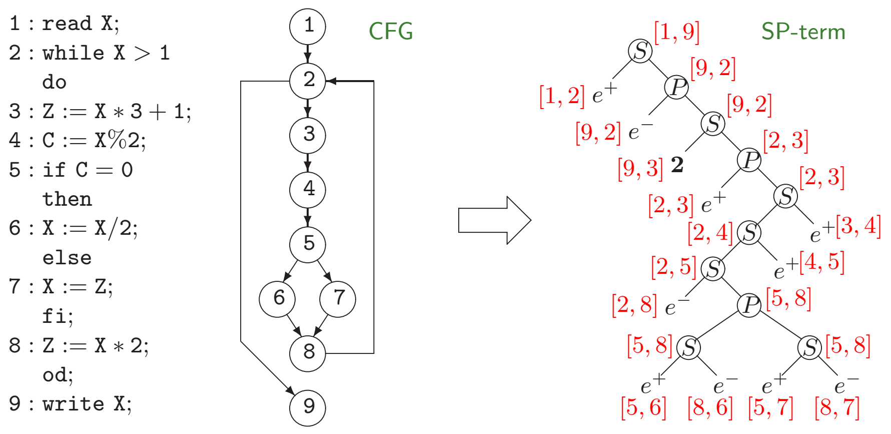


**Fig. 3.** Translation of `while` sentence to an SP Term.

Fig. 4 describes the control flow graph of the example program (in Section 1) and its transformation to an SP Term by *trans*. In Fig. 4, a tuple associated to each subtree is a tuple of terminals at the interpretation of the subtree. Note that the SP Term is computed in linear time; consequently, its size is linear to the size of a program.



```
1 : read X;
2 : while X > 1
    do
3 : Z := X * 3 + 1;
4 : C := X%2;
5 : if C = 0
    then
6 : X := X/2;
    else
7 : X := Z;
    fi;
8 : Z := X * 2;
    od;
9 : write X;
```

**Fig. 4.** An example of control flow graph and its transformation to SP Term

### 3.2 Checking on SP Terms

Now we show how the idea in Section 2 can be brought here for analyzing on SP Terms by checking on marked SP Term (a SP Term with some labels being marked). We use the same example of dead code detection for comparison.

To extract information from a node labeled $l$, we define the following functions.

$$\begin{aligned}
\mathit{defv}\ l &= x && \text{if the node is an assignment } x := e \text{ or a read sentence } \texttt{read}\ x.\\
&= \_ && \text{if the node is just an expression.}\\
\mathit{usev}\ l &= FV(e) && \text{if the node is either an assignment } x := e \text{ or an expression } e.\\
&= \{x\} && \text{if the node is a write sentence } \texttt{write}\ x.
\end{aligned}$$

So the function for checking whether the marked nodes in the marked SP term are dead codes can be defined by

$$\begin{aligned}
\mathit{checking}\ &::\ SP_2^* \to \mathit{Bool}\\
\mathit{checking}\ g\ &=\ \mathit{dead}\ g\ \ []\ \ []
\end{aligned}$$

where $\mathit{dead}\ g\ vs_1\ vs_2$ is to check the program with two additional arguments; the first denotes the live variables outgoing from the terminal 1 of $g$ and the second denotes the live variables outgoing from the terminal 2.

$$\begin{aligned}
\mathit{dead}\ (e^+,\ (l_1,l_2))\ vs_1\ vs_2 &= (\mathit{marked}\ l_1\ \Rightarrow\ \mathit{defv}\ l_1\ \notin\ vs_1 \cup (\mathit{usev}\ l_2 \cup (vs_2 \setminus \mathit{defv}\ l_2)))\ \wedge\\
&\quad\ (\mathit{marked}\ l_2\ \Rightarrow\ \mathit{defv}\ l_2\ \notin\ vs_2)\\
\mathit{dead}\ (e^-,\ (l_1,l_2))\ vs_1\ vs_2 &= \mathit{dead}\ (e^+,\ (l_2,l_1))\ vs_2\ vs_1\\
\mathit{dead}\ (\mathbf{2},\ (l_1,l_2))\ vs_1\ vs_2 &= (\mathit{marked}\ l_1\ \Rightarrow\ \mathit{defv}\ l_1 \notin vs_1)\ \wedge\\
&\quad\ (\mathit{marked}\ l_2\ \Rightarrow\ \mathit{defv}\ l_2 \notin vs_2)\\
\mathit{dead}\ (S\ x\ y)\ vs_1\ vs_2 &= \mathit{dead}\ x\ vs_1\ (\mathit{live}_2\ y\ vs_2\ [])\ \wedge\ \mathit{dead}\ y\ vs_2\ (\mathit{live}_2\ x\ vs_1\ [])\\
\mathit{dead}\ (P\ x\ y)\ vs_1\ vs_2 &= \mathit{dead}\ x\ (vs_1\ \cup\ \mathit{live}_1\ y\ vs_1\ vs_2)\ (vs_2\ \cup\ \mathit{live}_2\ y\ vs_1\ vs_2)\ \wedge\\
&\quad\ \mathit{dead}\ y\ (vs_1\ \cup\ \mathit{live}_1\ x\ vs_1\ vs_2)\ (vs_2\ \cup\ \mathit{live}_2\ x\ vs_1\ vs_2)
\end{aligned}$$

For the graph of $(e^+,\ (l_1,l_2))$, if the node labeled $l_1$ (or $l_2$) is marked, then the variable defined in the node should be live after the execution of the node, otherwise return *True* since there is no marked node in this graph. Similarly for $(e^-,\ (l_1,l_2))$, and simpler $(\mathbf{2},\ (l_1,l_2))$ (consisting of 2 isolated terminals $l_1$ and $l_2$). For the series graph $(S\ x\ y)$ which merges the second terminals of graphs $x$ and $y$, we update live variables outgoing from the first and the second terminals and check $x$ (and $y$) recursively. For instance, the call $\mathit{dead}\ x\ vs_1\ (\mathit{live}_2\ y\ vs_2\ [])$ checks $x$ recursively, where the live variables outgoing from the terminal 1 of $x$ are the same as those outgoing from the terminal 1 of $S\ x\ y$, and the live variables from the terminal 2 are the live variables starting from the terminal 2 of $y$. For the parallel graph $(P\ x\ y)$, the idea is similar except for the possible loop.

The function $\mathit{live}_1$ ($\mathit{live}_2$) takes a graph $g$ and the two sets of variables $vs_1$ and $vs_2$ that may be used outside $g$ from the terminal 1 and the terminal 2 respectively, and returns a set of variables that are alive at the terminal 1 (terminal 2). We omit the complementary definition for $\mathit{live}_2$.

$$\begin{aligned}
\mathit{live}_1\ (e^+,\ (l_1,l_2))\ vs_1\ vs_2 &= vs_1\ \cup\ \mathit{usev}\ l_1\ \cup\\
&\qquad (\mathit{usev}\ l_2\ \setminus \{\mathit{defv}\ l_1\})\ \cup\ (vs_2 \setminus \{\mathit{defv}\ l_1, \mathit{defv}\ l_2\})\\
\mathit{live}_1\ (e^-,\ (l_1,l_2))\ vs_1\ vs_2 &= vs_1\\
\mathit{live}_1\ (\mathbf{2},\ (l_1,l_2))\ vs_1\ vs_2 &= vs_1\\
\mathit{live}_1\ (S\ x\ y)\ vs_1\ vs_2 &= \mathit{live}_1\ x\ vs_1\ (\mathit{live}_2\ y\ vs_2\ [])\\
\mathit{live}_1\ (P\ x\ y)\ vs_1\ vs_2 &= \mathit{live}_1\ x\ vs_1\ vs_2\ \cup\ \mathit{live}_1\ y\ vs_1\ vs_2
\end{aligned}$$

### 3.3 Marking on SP Terms

Just like what we did for marking flowchart programs in Section 2.2, we decompose the definition of *checking* into a composition:

$$\mathit{checking}\ =\ \mathit{checking}'\ \circ\ \mathit{pre}$$

such that $checking'$ is a finite mutumorphism. We may think of $checking$ function as an attribute grammar over SP Terms, where we have two inherited attributes, namely $vs_1$ and $vs_2$, and three synthesized attributes, namely $dead$, $live_1$ and $live_2$. The basic idea of our decomposition is to put off the computation of the attribute grammar on the synthesized attribute $dead$ as much as possible, i.e., we first compute as many attributes not depending on $dead$ as possible (i.e., $vs_1$, $vs_2$, $live_1$, and $live_2$), and then compute other attributes (i.e., $dead$).

For our example, $pre$ turns out to be a function to precompute the values of $vs_1$ and $vs_2$, and attaches them to each edge in SP Term $g$ as follows:

$$\begin{array}{l} pre :: SP_2^* \to SP_2'^* \\ pre\ g = addLive\ g\ [\,]\ [\,] \end{array}$$

where

$$\begin{array}{ll} addLive\ g@(e^+,\ (l_1,l_2))\ vs_1\ vs_2 = (g, vs_1, vs_2) & \\ addLive\ g@(e^-,\ (l_1,l_2))\ vs_1\ vs_2 = (g, vs_1, vs_2) & \\ addLive\ g@(\mathbf{2},\ (l_1,l_2))\ vs_1\ vs_2 & = (g, vs_1, vs_2) \\ addLive\ (S\ x\ y)\ vs_1\ vs_2 & = S\ (addLive\ x\ vs_1\ (live_2\ y\ vs_2\ [\,]))\ (addLive\ y\ vs_2\ (live_2\ x\ vs_1\ [\,])) \\ addLive\ (P\ x\ y)\ vs_1\ vs_2 & = P\ (addLive\ x\ (vs_1\ \cup\ live_1\ y\ vs_1\ vs_2)\ (vs_2\ \cup\ live_2\ y\ vs_1\ vs_2)) \\ & \quad\ (addLive\ y\ (vs_1\ \cup\ live_1\ x\ vs_1\ vs_2)\ (vs_2\ \cup\ live_2\ x\ vs_1\ vs_2)) \end{array}$$

and $checking'$ is a finite mutumorphism defined by

$$\begin{array}{ll} checking'\ ((e^+,\ (l_1,l_2)), vs_1, vs_2) & = ch1\ l_1\ l_2\ vs_1\ vs_2 \\ checking'\ ((e^-,\ (l_1,l_2)), vs_1, vs_2) & = ch1\ l_2\ l_1\ vs_2\ vs_1 \\ checking'\ ((\mathbf{2},\ (l_1,l_2)), vs_1, vs_2) & = ch2\ l_1\ l_2\ vs_1\ vs_2 \\ checking'\ (S\ x\ y) & = checking'\ x\ \wedge\ checking'\ y \\ checking'\ (P\ x\ y)\ vs_1\ vs_2 & = checking'\ x\ \wedge\ checking'\ y \end{array}$$

where

$$\begin{array}{l} ch1\ l_1\ l_2\ vs_1\ vs_2 \\ \quad = (marked\ l_1\ \Rightarrow\ defv\ l_1\ \notin\ vs_1 \cup (usev\ l_2 \cup (vs_2 \setminus defv\ l_2)))\ \wedge\ (marked\ l_2\ \Rightarrow\ defv\ l_2\ \notin\ vs_2) \\ ch2\ l_1\ {}_2\ vs_1\ vs_2 \\ \quad = (marked\ l_1\ \Rightarrow\ defv\ l_1 \notin vs_1)\ \wedge\ (marked\ l_2\ \Rightarrow\ defv\ l_2 \notin vs_2) \end{array}$$

Applying the optimization theorem [31] immediately gives the following efficient solution.

$$\begin{array}{l} marking\ ::\ SP_2 \to SP_2^* \\ marking\ g = opt_{SP_2'}\ accept\ \phi_{e^+}\ \phi_{e^-}\ \phi_{\mathbf{2}}\ \phi_s\ \phi_p\ (pre\ g) \\ \quad \textbf{where} \\ \qquad accept = id \\ \qquad \phi_{e^+}\ ((e^+, (l_1,l_2)), vs_1, vs_2) = ch1\ l_1\ l_2\ vs_1\ vs_2 \\ \qquad \phi_{e^-}\ ((e^+, (l_1,l_2)), vs_1, vs_2) = ch1\ l_2\ l_1\ vs_2\ vs_1 \\ \qquad \phi_{\mathbf{2}}\ ((\mathbf{2}, (l_1,l_2)), vs_1, vs_2) = ch2\ l_1\ l_2\ vs_1\ vs_2 \\ \qquad \phi_s\ c_1\ c_2 = c1\ \wedge\ c_2 \\ \qquad \phi_p\ c_1\ c_2 = c1\ \wedge\ c_2 \end{array}$$

Note the function $opt_{SP_2'}$, like $opt_{Prog'}$ as in Fig. 8, is a specialized version (wrt $SP$ Term) of the generic function $opt$ as defined in [31].

## 4 Formal Study

### 4.1 Tree Width of Control Flow Graphs

The concept of a graph with bounded tree width [29] independently appeared from early 80's; partial $k$-tree in terms of cliques, some algebraic construction of $k$-terminal graphs [4, 12, 3], and in terms of separators, and they are all equivalent. The class of graphs with bounded tree width is quite restrictive; but the significant tread-off is: the class of graphs with bounded tree width frequently has a linear time algorithm for graph problems that are NP-complete for general graphs [12, 9].

For simplicity, we consider digraphs without multiple edges and loops (i.e., no edges from/to the same vertex). We first give definitions for undirected graphs without labels, but later we will treat directed graphs (digraphs) with labels on vertices. The set of vertices of $G$ is denoted by $V(G)$ and the set of edges of $G$ is denoted by $E(G)$.

**Definition 3.** A tree decomposition $\{X_t \mid t \in V(T)\}$ of a graph $G$ is a set of subsets of $V(G)$ indexed by elements in $V(T)$ for a tree $T$ such that

- $\cup_{t \in V(T)} X_t = V(G)$,
- for each edge of $G$, its end vertices are contained in some $X_t$, and
- for each $t, t', t'' \in V(T)$, $X_t \cap X_{t''} \subseteq X_{t'}$. if $t'$ is in a path of $T$ between $t$ and $t''$. ■

**Definition 4.** A graph $G$ has tree width at most $k$ if there exists a tree decomposition $\{X_t \mid t \in V(T)\}$ of $G$ such that $|X_t| \leq k+1$. For a graph $G$, the least such $k$ is the tree width of $G$ and is denoted by $twd(k)$.

From the algorithmic aspect, the concept of tree width (or tree decomposition) shows elegant prospect; frequently NP-complete graph algorithms are reduced to linear-time algorithms for graphs with bounded tree width [12, 9]. The key is an algebraic construction of graphs with bounded tree width followed by dynamic programming techniques.

In general, deciding the tree width of a graph is NP-complete [2]; however, for fixed $k$, whether a graph has tree width at most $k$ is decided in linear time [27]. Fortunately, we already know the upper bound of the tree width of control graphs of specific programming languages.

**Theorem 1.** [37] A control flow graph $G$ satisfies

- `goto`-free Algol and Pascal programs : $twd(G) \leq 3$
- Modula-2 programs: $twd(G) \leq 5$
- `goto`-free C programs: $twd(G) \leq 6$ ■

For Java programs, the control flow graphs may have unbounded tree width; however, the recent empirical test shows that most JAVA programs have control flow graphs with tree width at most 3 [20].

**Lemma 1.** Let $\{X_t \mid t \in T\}$ be a tree decomposition of a graph $G$. Then, for each $(u, v) \notin E(G)$ with $u, v \in V(G)$, $\{X_t \cup \{v\} \mid t \in T\}$ is a tree decomposition of a graph $G + (u, v)$ (i.e., $(V(G), E(G) \cup \{(u, v)\})$). ■

**Corollary 1.** A control flow graph $G$ of a program with at most $n$-`goto`s satisfies

- Algol and Pascal programs : $twd(G) \leq 3 + n$
- C programs: $twd(G) \leq 6 + n$ ■

This estimation is *not* optimal; for instance, if each `goto` sentence is distributed to each block, then the tree width increases at most 1. Further, if `goto` is *structured*, such as `break` or `exit` in `while`-loop, then sometimes the tree width does not increase; for instance, the flowchart scheme (in Section 2) with single `break` or `exit` in each `while`-loop has a control flow graph with tree width at most 2.

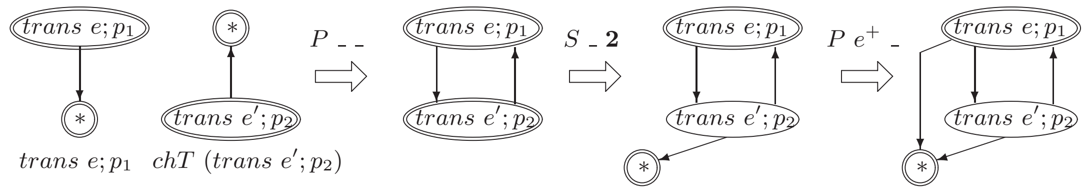


**Fig. 5.** Translation of `while` sentence with single `break` to an SP Term.

Transformation to an SP Term is obtained by replacing the definition of *trans* for `while`-sentence in Section 3.1 (see Fig. 5) with

$$
\begin{aligned}
&\mathit{trans}\ (\mathtt{l} : \textbf{while}\ e\ \textbf{do}\ p_1;\ (\mathtt{l}' : \textbf{if}\ e'\ \textbf{then break});\ p_2\ \textbf{od})\\
&\quad= P\ (e^+, (\mathtt{l}, *))\\
&\qquad (S\ (P\ (S\ (e^+, (l, l+1))\ (\mathit{chT}\ (\mathit{trans}\ p_1)))\ (S\ (\mathit{chT}\ (\mathit{trans}\ p_2))\ (e^+, (l', l'+1))))\\
&\qquad\quad (e^-, (*, \mathtt{l}'))).
\end{aligned}
$$

Thus, except for *spaghetti* `goto`'s, there is enough possibility to restrict the tree width in the feasible level.

## 4.2 Algebraic Construction of Digraphs with Bounded Tree Width

Basically, we obey notations and definitions to [3] except that we concentrate on digraphs, where it treats only undirected graphs; the extension to digraphs is straightforward. In this section, we treat only digraphs without labels.

**Definition 5.** A $k$-terminal digraph $(G, (l(1), \cdots, l(k)))$ is a digraph $G$ with a tuple $(l(1), \cdots, l(k))$ of $k$ vertices, called *terminals*.

**Definition 6.** Let $B_k$ be sorts for $k \geq 0$. Let $l_k^i, p_k, r_k, e_k(i,j), \mathbf{k}$ be signatures with sorts below

$$
\begin{cases}
l_k^i \colon B_{k-1} \to B_k, & p_k \quad\ : B_k \times B_k \to B_k, \quad r_k \colon B_k \to B_{k-1}\\
\mathbf{k} : B_k, & e_k(i,j) \colon B_k \quad (\text{for } k \geq 2)
\end{cases}
$$

where $1 \leq i \neq j \leq k$. ■

These function symbols $l_k^i, p_k, r_k, \mathbf{k}, e_k(i,j)$ are interpreted as operations on $k$-terminal graphs.

**Definition 7.** We define operations among $k$-terminal graphs as

- $l_k^i(s)$ is a lifting for $1 \leq i \leq k$, i.e., insert a new isolated terminal at the $i$-th position in $k-1$ terminals.
- $p_k(s,t)$ is a parallel composition for $k \geq 0$, i.e., fuse each $i$-th terminal in $s$ and $t$ for $1 \leq i \leq k$.
- $r_k(t)$ removes the last-terminal.
- $\mathbf{k}$ consists of $k$-isolated terminals.
- $e_k(i,j)$ consists of $k$ terminals with a diedge from the $i$-th terminal to the $j$-th. ■

Fig. 6 shows examples of the interpretation. A double circle expresses a terminal; each number associated to a terminal shows the numbering of terminals.

**Definition 8.** [3] Let $k \geq 2$. A series composition $s_k : \underbrace{B_k \times \cdots \times B_k}_{k} \to B_k$ is

$$s_k(t_1, \cdots, t_k) = r_{k+1}(p_{k+1}(l_{k+1}^1(t_1), p_{k+1}(l_{k+1}^2(t_2), \cdots, l_{k+1}^k(t_k)))).$$

■

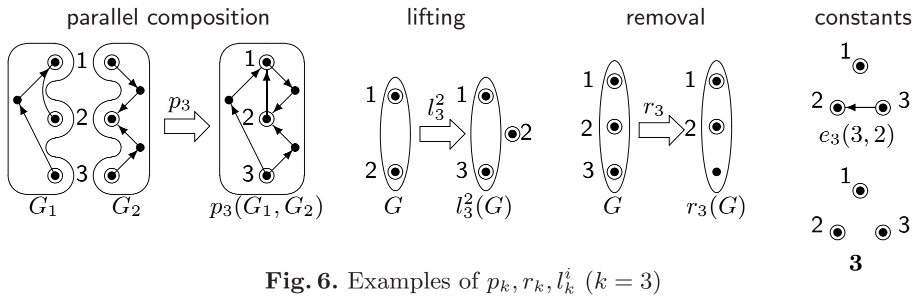


**Fig. 6.** Examples of $p_k, r_k, l_k^i$ $(k = 3)$

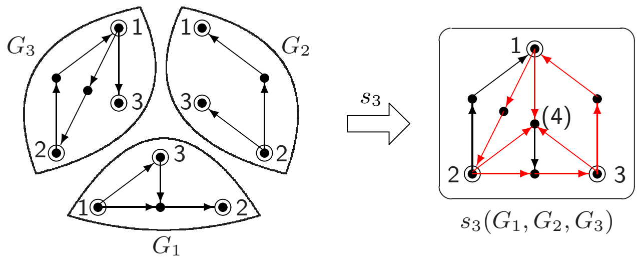


**Fig. 7.** Examples of the series composition $(k = 3)$

Intuition of $s_k$ is given in Fig. 7. That is, the parallel composition $p_k$ constructs any subgraph in $K_k$, and the series composition $s_k$ combines such components and produces a clique of the size $k + 1$ (i.e., an embedding of $K_{k+1}$), as emphasized by red lines in Fig. 7. A double circle expresses a terminal; each number associated to a terminal shows the numbering of terminals.

The next theorem shows that by a suitable choice of $k$ terminals in a digraph with tree width at most $k$ leads an algebraic construction by $p_k$ and $s_k$ in linear time (For the proof, see Appendix B).

**Theorem 2.** Let $G$ be a digraph with $twd(G) \leq k$ and $|V(G)| \geq k$ for $k \geq 2$. Then, a term $t$ constructed from $p_k, s_k, e_k(i,j), \mathbf{k}$ $(1 \leq i \neq j \leq k)$ is computed in linear time (wrt $|V(G)|$) such that $t$ is evaluated to a $k$-terminal digraph $\tilde{G}$ with $G = r_k^*(\tilde{G})$ where $r_k^* = r_1 \cdot \dots \cdot r_k$. ■

Remark that the converse is also true, i.e., for a term constructed from $p_k, s_k, e_k(i,j), \mathbf{k}$, its evaluation $G$ has tree width at most $k$.

**Lemma 2.** The evaluation $G$ of a term $t$ constructed from $p_k, s_k, e_k(i,j), \mathbf{k}$ has tree width at most $k$.

*Proof.* Let $\bar{t}$ be a term obtained from $t$ by replacing $s_k$ with its definition (see Definition 8). Let $T$ be a term $\bar{t}$ regarded as a tree, and let $X_{t'}$ be the set of terminals of the evaluation of $t'(\subseteq \bar{t})$. Then, $\{X_{t'} \mid t' \subseteq \bar{t}\}$ is a tree decomposition of $G$ with $|X_{t'}| \leq k + 1$. ■

*Example 1.* In Fig. 7, the digraph $s_3(G_1, G_2, G_3)$ has tree width 3, and $G_1, G_2.G_3$ have tree width 2. The SP Terms of $G_1, G_2, G_3$ are described as

$$\begin{aligned}
G_1 &= s_3(p_3(e_3(2,3), e_3(3,1)), p_3(e_3(1,2), e_3(1,3)), \mathbf{3}) \\
G_2 &= s_3(p_3(e_3(1,2), e_3(1,3)), e_3(3,1), \mathbf{3}) \\
G_3 &= s_3(\mathbf{3}, e_3(1,2), s_3(p_3(e_3(1,2), e_3(1,3)), p_3(e_3(1,2), e_3(3,1)), \mathbf{3}))
\end{aligned}$$

## 4.3 SP Terms

In Section 3.1, we show how an SP Term of a control flow graph of a flowchart program (i.e., a series-parallel graph) is computed in linear time. In this section, we generalize this result for general control flow graphs.

First, we define SP Terms $SP_k$ for digraphs with tree width at most $k$, and then we state that an SP Term is computed in linear time. We will also show the implementation of the translation function for a simple imperative language with the limited number of `goto`, and show an example.

**Definition 9.** An SP Term is a pair of a ground term $t$ and a tuple $(l(1), \cdots, l(k))$, defined as the following.

$$\begin{array}{rll} SP_k := & (e_k(i,j),\ (l(1),\cdots,l(k))) & (i \neq j) \\ | & (\mathbf{k},\ (l(1),\cdots,l(k))) & \\ | & S_k\ \underbrace{SP_k\ \ldots\ SP_k}_{k} & \\ | & P_k\ SP_k\ SP_k & \end{array}$$

Here, $l(1), \cdots, l(k)$ are called labels, $P_k$ is the parallel composition, and $S_k$ is the series composition defined as:

$$\begin{array}{l} S_k\ (t_1, (l_1(1), \cdots, l_1(k))\ \cdots\ (t_k, (l_k(1), \cdots, l_k(k)))) \\ \quad = \ (s_k(t_1, \cdots, t_k),\ (l_2(1), l_1(1), \cdots, l_1(k-1))) \\ \quad\ \ \text{if } \wedge_i\ \mathit{match}(l_1(i-1), \cdots, l_{i-1}(i-1), l_{i+1}(i), \cdots l_k(i)) \end{array}$$

$$\begin{array}{l} P_k\ (t_1, (l(1), \cdots, l(k))\ (t_2, (l'(1), \cdots, l'(k)))) \\ \quad = \ (p_k(t_1, t_2),\ (l(1), \cdots, l(k))) \quad \text{if } \wedge_i\ \mathit{match}(l(i), l'(i)) \end{array}$$

where $\mathit{match}(l(1), \cdots, l(k))$ is $\mathit{true}$ if $l(i) = l(j)$ for each $l(i), l(j) \neq *$. (i.e., accept the special label $*$ as a wild card). ■

SP Terms $(e_k(i,j), (l(1), \cdots, l(k)))$ and $(\mathbf{k}, (l(1), \cdots, l(k)))$ are interpreted as $k$-terminal digraphs; they are

$$(\{l(1), \cdots, l(k)\}, \{(l(i), l(j)\}, (l(1), \cdots, l(k)))$$

i.e., $k$-vertices $l(1), \cdots, l(k)$ with one diedge from $l(i)$ to $l(j)$, and

$$(\{l(1), \cdots, l(k)\}, \phi, (l(1), \cdots, l(k)))$$

i.e., $k$ isolated vertices $l(1), \cdots, l(k)$, respectively. From Theorem 2, the next corollary is immediate.

**Corollary 2.** Let $G$ be a control flow graph with $twd(G) \leq k$ and $|V(G)| \geq k$ for $k \geq 2$. Then, an SP Term is computed in linear time (wrt $|V(G)|$) such that its evaluation is a pair of $k$-terminal digraph $\tilde{G}$ and a tuple of $k$-terminals with $G = r_k^*(\tilde{G})$. ■

This shows the general method to compute an SP Term from a control flow graph via tree decomposition. This is done in linear-time, but not so efficient linear time. There would be possibility of an efficient direct translation from a program itself (such as *trans* in Section 3.1). For a simple imperative language with `GOTO`, such a translation is shown in Appendix C.

### 4.4 Analyzing SP Term through Checking

As demonstrated in Section 3, efficient algorithms for analyzing on SP Terms can be derived systematically by developing efficient catamorphic algorithms for checking marked SP Terms.

**Definition 10 (Catamorphism on SP Terms).** A function $\mathit{cata}\ ::\ SP_k \to R$ is called a catamorphism on SP terms if it is defined in the following form:

$$\begin{array}{ll} \mathit{cata}\ (e_k(i,j),\ (l(1),\cdots,l(k))) & = f_{ij}\ (e_k(i,j),\ (l(1),\cdots,l(k))) \quad (i \neq j) \\ \mathit{cata}\ (\mathbf{k},\ (l(1),\cdots,l(k))) & = g\ (\mathbf{k},\ (l(1),\cdots,l(k))) \\ \mathit{cata}\ (S_k\ x_1\ x_2\ \ldots\ x_k) & = h_s\ (\mathit{cata}\ x_1)\ (\mathit{cata}\ x_2)\ \ldots\ (\mathit{cata}\ x_k) \\ \mathit{cata}\ (P_k\ x_1\ x_2) & = h_p\ (\mathit{cata}\ x_1)\ (\mathit{cata}\ x_2) \end{array}$$

where $f_{ij}$, $g$, $h_s$ and $h_p$ are given functions. ■

Given a function $checking :: SP_k^* \to Bool$ for check whether the marked labels in a marked SP Term satisfy a certain property, the following theorem gives a sufficient condition for existence of an efficient algorithm $marking :: SP_k \to SP_k^*$ to mark as many labels as possible of a (non-marked) SP Term, such that the marked term holds the property.

**Theorem 3 (Checking-Marking Connection).** If the checking function

$$checking \; :: \; SP_k^* \to Bool$$

is an *efficient* algorithm described in the form:

$$checking \; = \; post \circ checking' \circ pre$$

such that $checking' :: SP_k'^* \to R$ is a catamorphism on SP term, in which (1) $R$ is a finite domain, and (2) $SP_k'^*$ has the same structure as $SP_k^*$ except that there may be some change on some node, then an efficient algorithm for marking

$$marking \; :: \; SP_k \to SP_k^*$$

can be automatically generated. ■

We have seen an example for detecting dead codes in $SP_2$ terms in Section 3, and we omit the proof of this theorem in this paper. It should be noted that our condition for the function *checking* is not restrictive. For instance, even if *pre* and *post* are identity function, the finite catamorphism itself has enough descriptive powerful to describe various kinds of properties. In fact, mutumorphisms [17, 21] (which cover primitive recursive functions) and attribute grammars [18] can be transformed into our catamorphic form.

## 5 Related Work

Many researches have been devoted to the so-called *declarative approaches* to program analyses. Steffen and Schmidt [34, 33] showed that temporal logic is well suited to *describe* data dependencies and other properties exploited in classical compiler optimization. Lacey and de Moor [25] added temporal logic side conditions to express complex restriction when transforming imperative programs, and it is shown in [26] that temporal logic plays a crucial role in the *proofs* of correctness of optimizing transformation. For practical application, an attempt has been made in [13] to extend an existing declarative language (Prolog) for code program analysis and program transformation. Our work can be thought of as an important step towards *reasoning and optimizing* program analyses, based on the the theory of program calculation [7]. Our checking-marking connection theorem demonstrates a useful calculation rule for deriving efficient program analysis algorithms from a naive functional specification.

In fact, our catamorphic approach was greatly motivated by the successful application of *program calculational approach* to optimization of functional programs. It was first proposed as the theory of lists [5], and was then extended to be a general theory of datatypes [7]. It has proved to be very useful not only in deriving variant efficient programs [23], but also in constructing optimization passes in compilers [36, 21, 22]. Our formulating program analysis in this framework enables utilization of existing calculation techniques.

There are several works on catamorphic approach to computation on graphs. For instance, [15] treats graphs with embedded functions, i.e., graphs are treated as functions that generates all paths in a graph. [14] introduces the *active pattern matching*, which is a conditional pattern matching mechanism. Their approaches are interesting in description, but the existence of strong side conditions limits the use of program calculation. Instead, we restrict the class of graphs to graphs with bounded tree width, in which many NP-complete graph algorithms are reduced to linear time by dynamic programming techniques [12, 9].

The key, which enables us to apply a catamorphic approach, is an algebraic construction of graphs with bounded tree width. One of the early work for flowchart scheme (i.e., series-parallel graphs, or in other words, graphs with tree width at most 2) is found in [32]. Bauderon and Courcelle are also pioneers [4, 12], and our SP Term is greatly in debt to the work by Arnborg, et.al. [3]. However, their constructions do not fit to our purpose; for instance, the construction in [3] requires the operations $l_j^i, r_j, s_j, p_j$ with $1 \leq i \leq j \leq k$ to construct a graph with tree width at most $k$. Thus, the number of their operations are more than $k(k+1)(k+2)/6$, and this makes us difficult to describe functional specification. Here, we propose another construction, SP Term, which has only 2 constructors $S_k, P_k$ except for constants. The number of constants $e_k(i, j)$ has square growth, but they are interpreted as an edge between $i$-th and $j$-th terminals. For these constants, writing functional specification (base cases) will be easy; even in homogeneous way.

Of course, restricting graphs with bounded tree width will not be meaningful without interesting applications; we set them on control flow analyses.

The start point is the fact that a structured imperative program have a control flow graph with relatively small tree width [37]. Thorup, et.al. investigate the applications of this fact, such as the register allocation (as the coloring problem) and the generalized dominator detection [1]. Their approach is problem specific, and we intend to derive linear time algorithms from general functional specifications, such that the generalized dominator detection is one of instances.

Currently, we treat only imperative languages *without* procedure call. For interprocedural control flow analyses, we need so called CFG (context-free grammar) reachability [28]. We foresee that the reduction of CFG reachability to the automata theoretic methods, such as in [11, 10], would extend to interprocedural analyses.

## 6 Conclusion and Future Work

We proposed a catamorphic approach to deriving control flow analyses from functional specifications. The contribution of the paper can be viewed from two aspects:

– **From theory:** There have been lots of work on automatic generation of linear-time algorithms based on tree decomposition [12, 9] (or equivalent concepts in relational database theory [38]). They are elegant in theory, but far from practice; the specification is frequently given by monadic second order formulae, and each existence of quantifiers causes the exponential explosion of the constant factor. Our approach applies the functional specification, which drastically reduces the constant factor with the aid of program calculation. The key is the concept of SP Term; it reduces the number of function symbols (except for constants) from the square of the upper bound of tree width to 2, and makes writing functional specification feasible.
– **From practice:** Recent developments of model checking systems, such as SMV and SPIN, enables us to efficiently implement control/data flow analyses as model checking. The efficiency of model checkers are mostly depending on their implementation techniques, such as BDD. Our viewpoint is that the most instances of applications of model-checkers would be rather well-structured transition systems, and there may be enough room to improve the efficiency also from theory [16]. For example, the control flow graph of an imperative program with the limited number of `goto` can be regarded as a graph with bounded tree width (see Section 4.1). By this fact, we can reduce an iterative procedure to an one-path linear time algorithm by dynamic programming techniques.

This research is just at the beginning, and there are lots of subjects to conquer. For instance:

– The set of SP Terms is an initial algebra; however, a $k$-terminal graph may have multiple representations by SP Terms. This means whether the user defined functional specification is consistent with the interpretation of SP Terms to $k$-terminal graphs is up to the user's responsibility. From its own theoretical interest and possible better support, we hope to give the complete axiomatization of SP Terms under this interpretation.

– Section 4.3 shows the general method to compute an SP Term from a control flow graph via tree decomposition. This is done in linear-time, but not so efficient with the large constant factor. There would be possibility of an efficient direct translation from a program itself (such as *trans* in Section 3.1). We hope that such direct translation would be obtained by applying program calculational techniques to the composition of the translation from a program to a tree decomposition of a control flow graph, and that from a tree composition to an SP Term.
– Currently, the relation between monadic second order specification and functional specification are not so clear. In [9], they translate formulae to functions, but very inefficient way. We hope to find some systematic way to translate formulae to efficient fintie mutumorphisms.

## Acknowledgments

The authors thank Oege de Moor and Jeremy Gibbons for stimulating discussions during their visit of Tokyo University, and thank Aki Takano and Yuji Yamaoka for their helpful suggestions on this work.

## References

1. S. Alstrup, P.W. Lauridsen, and M. Thorup. Generalized dominators for structured programs. *Algorithmica*, 27(3):244–253, 200.
2. S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a $k$-tree. *SIAM Journal Algebraic and Discrete Method*, 8(2):277–284, 1987.
3. S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *Journal of the Association for Computing Machinery*, 40(5):1134–1164, 1993.
4. M. Bauderon and B. Courcelle. Graph expressions and graph rewritings. *Mathematical System Theory*, 20:83–127, 1987.
5. R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.
6. R. Bird. Maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, 2001.
7. R.S. Bird and O. de Moor. *Algebras of Programming.* Prentice Hall, 1996.
8. H.L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal Computing*, 25(6):1305–1317, 1996.
9. R.B. Borie, R.G. Parker, and C.A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.
10. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR (CONCUR '97)*, pages 135–150X, 1997. Lecture Notes in Computer Science, Vol. 1243, Springer-Verlag.
11. D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Automata, Languages and Programming (23rd ICALP)*, pages 194–205, 1996. Lecture Notes in Computer Science, Vol. 1099, Springer-Verlag.
12. B. Courcell. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5, pages 194–242. Elsevier Science Publishers, 1990.
13. S. Drape, O. de Moor, and G. Sittampalam. Transforming the .net intermediate language using path logic programming. In *Proceedings of PPDP 2002*. ACM Press, 2002.
14. M. Erwig. Functional programming with graphs. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming, ICFP '97*, pages 52–65. ACM Press, 1997. SIGPLAN Notices 32(8).
15. L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions *(or, programs from outer space)*. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96*, pages 284–294. ACM Press, 1996.
16. J. Flum. Tree-decompositions and the model-checking problem. *Bulletin of the Europian Association for Theoretical Computer Science*, 73:78–98, 2001.

17. M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.
18. M. Fokkinga, J. Jeuring, L. Meertens, and E. Meijer. A translation from attribute grammers to catamorphisms. *Squiggolist*, pages 1–6, November 1990.
19. Jeremy Gibbons. An initial-algebra approach to directed acyclic graphs. In B. Moeller, editor, *Mathematics of Program Construction*, pages 122–138. Lecture Notes in Computer Science 947, Springer Verlag, 1995.
20. J. Gustedt, O.A. Mæhle, and A. Telle. The treewidth of Java programs. In *Proc. 4th Workshop on Algorithm Engineering and Experiments, ALENEX 2002*, pages 86–97, 2002. Lecture Notes in Computer Science, Vol. 2409, Springer-Verlag.
21. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data transversals. In *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 9–11. ACM Press, 1997. Amsterdam, The Nederlands.
22. Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.
23. J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.
24. S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* Available online: `http://www.haskell.org`, February 1999.
25. D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Proceedings of Compiler Construction 2001.* LNCS, Srpinger Verlag, 2001.
26. David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Symposium on Principles of Programming Languages (POPL 2002)*, pages 283–294. ACM Press, 2002.
27. L. Perković and B. Reed. An improved algorithm for finding tree decompositions of small width. *International Journal of Foundations of Computer Science*, 11(3):365–371, 2000.
28. T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:701–726, 1998.
29. N. Robertson and P.D. Seymour. Graph minors II. algorithmmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.
30. I. Sasano, Z. Hu, and M. Takeichi. Generation of efficient programs for maximum multi-marking problems. In *ACM SIGPLAN Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, pages 72–91, Firenze, Italy, September 2001. Springer Verlag, LNCS 2196.
31. I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear time algorithm for solving maximum weightsum problems. In *The 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 137–149, Montreal, Canada, September 2000. ACM Press.
32. H. Schmeck. Algebraic characterization of reducible flowcharts. *Journal of Computer System Science*, 27(2):165–199, 1983.
33. David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48. ACM Press, 1998.
34. Bernhard Steffen. Data flow analysis as model checking. In A.R. Meyer T. Ito, editor, *Theoretical Aspects of Computer Science (TACS'91), Sendai (Japan)*, volume 526 of *Lecture Notes in Computer Science (LNCS)*, pages 346–364, Heidelberg, Germany, September 1991. Springer-Verlag.
35. K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on seires-parallel graphs. *Journal of the Association for Computing Machinery*, 29:623–641, 1982.
36. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.
37. M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142:159–181, 1998.
38. M.Y. Vardi. Constraint satisfaction and database theory: a tutorial. In *Proc. 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'00*, pages 76–85. ACM Press, 2000.

## A Maximum Marking Problem

Maximum marking problem (MMP for short) can be specified as follows: Given a data structure $x$, the task is to find a way to mark some elements in $x$ such that the marked data structure of $x$ satisfies a certain property $p$ and has the maximum value with respect to certain weight function $w$. This means that no other marking of $x$ satisfying $p$ can produce a larger value with respect to $w$.

By MMP, we can express various kinds of problems which include knapsack problems, optimized range problems in data mining, the maximum segment sum problem, and so on. Though those problems are from different application area, they can be formulated in the uniform way: Given a data $x$, find a feasible selection of elements whose weight is maximum in all the feasible selection. For instance, by expressing feasibility using a predicate $p$ and giving weight using a weight function $w$, the specification of MMP on lists can be described as follows:

$$mmp\ p \ = \ \uparrow_w / \circ \mathit{filter}\ p \circ gen\ [\mathit{True}, \mathit{False}]$$

Here *gen* is used for generating all the way of selecting elements with marks *True* and *False*. Selection is expressed by marking selected elements and unmarking non-selected elements, where *True* corresponds to "mark" and *False* corresponds to "unmark". The operator $\uparrow_f$ is called the selection operator [5] and is defined by

$$\begin{aligned} a \uparrow_f b &= b, \quad \textbf{if } f\ a \leq f\ b \\ &= a, \quad \textbf{otherwise}. \end{aligned}$$

In this definition, the value of $a \uparrow_f b$ is $b$ when $f\ a = f\ b$. The operator / is called the reduce operator [5], which takes an associative binary operator and a list, defined as follows:

$$\oplus/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

From the above specification for MMP, we can derive a linear time algorithm mechanically if property $p$ and weight function $w$ satisfy certain conditions. Of course, it is not expected to derive a linear one for every MMP problem since MMP includes NP-hard problems, such as the knapsack problems. However, for instance, the knapsack problem restricted to integer weight satisfies our requirements, and we can derive its linear time algorithm.

**Theorem 4 (Optimization Theorem).**
*MMP specified by*

$$\begin{aligned} &spec : \ \ D\ \alpha \rightarrow D\ \alpha^* \\ &spec = mmp\ p_0 \end{aligned}$$

*can be solved in linear time if the property description $p_0 : D\ \alpha^* \rightarrow Bool$ can be defined as mutumorphisms with other property descriptions $p_i : D\ \alpha^* \rightarrow Bool$ for $i = 1, \ldots, n$.*

Definition of the function `OptP` is shown in Fig. 8. For detail, refer to [31].

## B Proof of Theorem 2

**Theorem 5.** [2] Deciding tree decomposition is NP-complete. ■

**Theorem 6.** [27] For a fixed $k$, deciding whether the tree width of a graph $G$ is less-than-equal $k$ is linear-time solvable. Furthermore, if $twd(G) \leq k$, its tree decomposition is also computed in linear-time. ■

Without loss of generality, we can assume $X_t \neq X_{t'}$ for each $t, t$ with $t \neq t'$ for a tree decomposition $\{X_t \mid t \in V(T)\}$.

**Definition 11.** A tree decomposition $\{X_t \mid t \in V(T)\}$ of a graph $G$ is smooth

```
optP accept phiA phiR phiT phiS phiI phiW p =
    let opts = candidates phiA phiR phiT phiS phiW phiI p
    in  third (getmax [(c,w,t) | (c,w,t) <- opts,
                                 accept c])

candidates phiA phiR phiT phiS phiI phiW (Assign v@((x, e), vs)) =
    eachmax [(phiA mv,
              if marked mv then 1 else 0,
              Assign mv)
            | mv <- [mark v, unmark v]]

candidates phiA phiR phiT phiS phiI phiW (Read v@(x, vs)) =
    eachmax [(phiR mv,
              if marked mv then 1 else 0,
              Read mv)
            | mv <- [mark v, unmark v]]

candidates phiA phiR phiT phiS phiI phiW (Write v@(x, vs)) =
    eachmax [(phiT mv,
              if marked mv then 1 else 0,
              Write mv)
            | mv <- [mark v, unmark v]]

candidates phiA phiR phiT phiS phiI phiW (Seq p1 p2) =
     let opts1 = candidates phiA phiR phiT phiS phiW phiI p1
         opts2 = candidates phiA phiR phiT phiS phiW phiI p2
     in  eachmax [(phiS c1 c2, w1+w2, Seq cand1 cand2)
                 | (c1,w1,cand1) <- opts1, (c2,w2,cand2) <- opts2]

candidates phiA phiR phiT phiS phiI phiW (If e p1 p2) =
     let opts1 = candidates phiA phiR phiT phiS phiW phiI p1
         opts2 = candidates phiA phiR phiT phiS phiW phiI p2
     in  eachmax [(phiI e c1 c2, w1+w2, If e cand1 cand2)
                 | (c1,w1,cand1) <- opts1, (c2,w2,cand2) <- opts2]

candidates phiA phiR phiT phiS phiI phiW (While e p) =
     let opts = candidates phiA phiR phiT phiS phiW phiI p
     in  eachmax [(phiW e c, w, While e cand)
                 | (c,w,cand) <- opts]

getmax :: (Eq c, Ord w) => [(c,w,a)] -> (c,w,a)
getmax [] = error "No solution."
getmax xs = foldr1 f xs
  where f (c1,w1,cand1) (c2,w2,cand2)
          = if w1>w2 then (c1,w1,cand1) else (c2,w2,cand2)

eachmax :: (Ord w, Eq c) => [(c,w,a)] -> [(c,w,a)]
eachmax xs = foldl f [] xs
  where f [] (c,w,cand) = [(c,w,cand)]
        f ((c',w',cand') : opts) (c,w,cand) =
            if c==c' then
               if w>=w' then (c,w,cand) : opts
               else (c',w',cand') : opts
            else (c',w',cand') : f opts (c,w,cand)
```

**Fig. 8.** Optimization function *optP*

– $|X_t| = k+1$ for each $t \in V(T)$, and
– $|X_t \cap X_{t'}| = k$ for each pair of adjacent vertices $t, t \in V(T)$.

■

If there is a tree decomposition $\{X_t \mid t \in V(T)\}$ of a graph $G$, it is transformed to a smooth tree decomposition of the same tree width in linear time (see details for Section 2 in [8]) as the next lemma shows.

**Lemma 3.** (Lemma 2.5 in [8]) If a tree decomposition $\{X_t \mid t \in V(T)\}$ of a graph $G$ is smooth, $|V(T)| = |V(G)| - k$. ■

**Theorem 2** Let $G$ be a digraph with $twd(G) \leq k$ and $|V(G)| \geq k$ for $k \geq 2$. Then, a term $t$ constructed from $p_k, s_k, e_k(i,j), \mathbf{k}$ $(1 \leq i \neq j \leq k)$ is computed in linear time (wrt $|V(G)|$) such that $t$ is evaluated to a $k$-terminal digraph $\tilde{G}$ with $G = r_k^*(\tilde{G})$ where $r_k^* = r_1 \cdot \cdots \cdot r_k$.

*Proof.* From Theorem 6, a tree decomposition $\{X_t \mid t \in V(t)\}$ of $G$ is obtained in linear time.[2] Without loss of generality, we can assume that the tree decomposition $\{X_t \mid t \in V(t)\}$ of $G$ is smooth and $|X_t| = k+1$ for each $t \in V(T)$. Let us fix $t \in V(T)$. We define $t' \preceq t''$ if $t'$ is in the path between $t$ and $t''$ (i.e., $t$ is regarded as the root of $T$).

We define the characteristic vertices $\{v_t \mid t \in V(t)\}$ of the tree decomposition $\{X_t \mid t \in V(t)\}$ as follows. For the root $t$ of $T$, let $v_t$ be an arbitrary element in $X_t$. Let $t'$ be a child vertex of some $t'' \in V(T)$. Then, since $|X_{t'} \cap X_{t''}| = k$ and $|X_{t'}| = k+1$, let $v_{t'}$ be the (unique) element in $X_{t'} \setminus X_{t''}$.

For $t \in V(T)$, let $Y_1(t), \cdots, Y_{k+1}(t) \subseteq X_t$ be subsets such that $|Y_i(t)| = k$. Assume $v_t \in Y_i(t)$ for $1 \leq i \leq k$ and $Y_{k+1}(t) = X_t \setminus \{v_t\}$ (i.e., if the parent vertex $t'$ of $t$ exists, $Y_{k+1} = X_t \cap X_{t'}$).

Let a graph decomposition $\{G_t \mid t \in V(T)\}$ be a family of graphs $G_t$ with $V(G_t) = X_t$, $\cup_{t \in V(t)} E(G_t) = E(G)$, and $E(G_t) \cap E(G_{t'}) = \phi$ for each $t \neq t'$.

Let $G_1(t), \cdots, G_{k+1}(t)$ be subgraphs of $G_t$ such that $V(G_i(t)) = Y_i(t)$ and $\cup_{1 \leq i \leq k} E(G_i(t)) = E(G_t)$. We further assume that $E(G_i(t)) \cap E(G_j(t)) = \phi$ for $i \neq j$.

We inductively define a term $term(t)$ for $t \in V(T)$ with

– $term(t)$ is constructed from $p_k$, $s_k$, $e_k(i,j)$, and $\mathbf{k}$.
– $term(t)$ is evaluated to a $k$-terminal graph $G'$ such that $V(G') = \cup_{t \preceq t'} V(G'_t)$, $E(G') = \cup_{t \preceq t'} E(G'_t)$, and the set of terminals of $G'$ is $X_t \setminus \{v_t\}$.
– for the root $t$ of $T$, $r_k^*(term(t))$ is evaluated to $G$.

Let $t$ be a leaf in $T$. Since $|V(G_i(t))| = k$, we regard $G_i(t)$ as a $k$-terminal graph with $V(G_i(t))$ as the set of terminals. If $E(G_i(t)) = \phi$, then $\mathbf{k}$ is evaluated to $G_i(t)$. If $E(G_i(t)) \neq \phi$, a term $u_i = p_k(p_k(e_k(l,m), \cdots)$ with $(l,m), \cdots \in E(G_i(t))$ is evaluated to $G_i(t)$. Then $p_k(s_k(u_1, \cdots, u_k), u_{k+1})$ is evaluated to $G_t$ with $X_t \setminus \{v_t\}$ as the set of terminals.

Let $t$ be not a leaf in $T$. Similarly, we define $u_i$. Let $D_i$ be the set of child vertices of $t$ with $Y_i(t) = X_t \cap X_{t'}$ for $t' \in D_i$. Let $u'_i = p_k(u_i, p_k(term(t'), \cdots))$ for $t', \cdots \in D_i$. Then $p_k(s_k(u'_1, \cdots, u'_k), u'_{k+1})$ is evaluated to $G_t$ with $X_t \setminus \{v_t\}$ as the set of terminals.

Since each step requires $O(|E(G_t)|) + O(k)$ steps during the transformation, it needs $O(E(G)) + O(k|V(T)|)$ steps. Since $|E(G)| \leq k|V(G)| - k(k+1)/2$ (Lemma 2.1 in [8]) and $|V(T)| = |V(G)| - k$ (Lemma 3), the number of total steps of the transformation is $O(k|V(G)|)$. Thus, since the number of steps for constructing a tree decomposition is $O(k^2|V(G)|)$, we obtain a term evaluated to $G$ in linear time. ■

[2] More precisely $O(k^2|V(G)|)$.

## C Computing SP Terms of Control Flow Graphs of Imperative Languages

We introduce a simple imperative language, which we use to demonstrate generation of efficient algorithms for program analysis in the next section. The definition of the language, as in Figure 9, is almost the same as that in [26], except for the additional "while" construct. To provide a simple framework for generating efficient program analysis, this language has no exceptions or procedures.

| | | | |
|---|---|---|---|
| $P$ | $::=$ | $I;\ P$ | Program |
| $I$ | $::=$ | $l:\ C$ | Instruction |
| $C$ | $\mid$ | $V\ :=\ E$ | Assignment |
| | $\mid$ | `read` $V$ | Read sentence |
| | $\mid$ | `write` $V$ | Write sentence |
| | $\mid$ | **if** $E$ **then** $P$ **else** $P$ **fi** | If command |
| | $\mid$ | **while** $E$ **do** $P$ **od** | While command |
| | $\mid$ | **goto** | Goto command |
| | $\mid$ | **break** | Break command |
| | $\mid$ | **exit** | Exit command |
| $E$ | $::=$ | $V$ | Variable expression |
| | $\mid$ | $O\ E\ \cdots\ E$ | Application expression |
| $O$ | : | `operator` | |
| $V$ | : | `variable` | |
| $l$ | : | `label` | |

**Fig. 9.** A Simple Imperative Language

Let us consider a program in this language with at most $n$-`goto`'s. We give the implementation of the translation *transG* to an SP Term below. The basic idea is; construct an SP Term by ignoring `goto` and memorize their source vertices as additional terminals. Then, scan an SP Term again, and add an edge by the parallel composition at some subterm in which the destination vertex eventually becomes a terminal. (Note that each vertex in a control flow graph becomes a terminal of some subterm of an SP Term.)

Let *prog* be a program written in the language in Fig. 9. As in Section 3.1, we first preprocess *prog* to *lprog* by labeling each line of *prog*. Let $((so_1, des_1), \cdots, (so_n, des_n))$ be the tuple of $n$-pairs of the source and destination vertices of each `goto` in *lprog* (We assume $so_i \neq des_i$ for each $i$).

Let $lprog'$ be a program obtained from *lprog* by replacing `goto` with a null command `skip`. Then, $lprog'$ is regarded as a flowchart program in Section 3.1, and an SP Term of *prog* is obtained as

$$addG\ (nlift\ (trans\ lprog')).$$

where functions are defined in Fig. 10.

The function *nlift* insert labels of `goto` sentences as new $n$-terminals between the first and the second (original) terminal in an SP Term; *permT* permutes except for the last terminal to adapt to the series composition. next, *addG* adds an edge between the source and destination vertices of each GOTO-sentence. Here, the first and the second terminals of an SP Term $x \in SP$ are denoted by $x[1]$ and $x[2]$, respectively.

Note that if each *block* has at most $m$-`goto` then instead of $n$ (the sum of the numbers of `goto`) we can similarly transform a control flow graph to an SP Term in $SP_{m+2}$.

$$
\begin{array}{l}
\mathit{nlift} \;::\; SP \rightarrow SP_{n+2} \\
\mathit{nlift}\ (S\ x\ y) \\
\quad = \; S_{n+2}\ (\mathit{permT}\ (\mathit{nlift}\ x)) \\
\qquad\qquad (\mathbf{n+2}, (y[1], so_2, so_3, \cdots, so_n, x[1], x[2])) \\
\qquad\qquad (\mathbf{n+2}, (y[1], so_1, so_3, \cdots, so_n, x[1], x[2])) \\
\qquad\qquad \cdots \\
\qquad\qquad (\mathbf{n+2}, (y[1], so_1, so_2, \cdots, so_{n-1}, x[1], x[2])) \\
\qquad\qquad (\mathit{nlift}\ y) \\
\mathit{nlift}\ (P\ x\ y) \; = \; P_{n+2}\ (\mathit{nlift}\ x)\ (\mathit{nlift}\ y) \\
\mathit{nlift}\ (e^+,\ (l_1, l_2)) \; = \; (e_{n+2}(1, n+2)\ (l_1, so_1, \cdots, so_n, l_2)) \\
\mathit{nlift}\ (e^-,\ (l_1, l_2)) \; = \; (e_{n+2}(n+2, 1)\ (l_1, so_1, \cdots, so_n, l_2)) \\
\mathit{nlift}\ (\mathbf{2},\ (l_1, l_2)) \; = \; (\mathbf{2},\ (l_1, so_1, \cdots, so_n, l_2))
\end{array}
$$

$$
\begin{array}{l}
\mathit{permT} \;::\; SP_{n+2} \rightarrow SP_{n+2} \\
\mathit{permT}\ (S_{n+2}\ x_1\ \cdots\ x_{n+2}) \\
\quad = \; S_{n+2}\ (\mathit{permT}\ x_{n+1})\ (\mathit{permT}\ x_1)\ \cdots\ (\mathit{permT}\ x_n)\ (\mathit{permT}\ x_{n+2}) \\
\mathit{permT}\ (P_{n+2}\ x\ y) \; = \; P_{n+2}\ (\mathit{permT}\ x)\ (\mathit{permT}\ y) \\
\mathit{permT}\ (e_{n+2}(i, j),\ (l_1, \cdots, l_{n+2})) \\
\quad = \; (e_{n+2}((\mathit{perm}\ i), (\mathit{perm}\ j)),\ (l_{n+1}, l_1, \cdots, l_n, l_{n+2})) \\
\mathit{permT}\ (\mathbf{n+2},\ (l_1, \cdots, l_{n+2})) \; = \; (\mathbf{n+2},\ (l_{n+1}, l_1, \cdots, l_n, l_{n+2}))
\end{array}
$$

$$
\begin{array}{l}
\mathit{perm} \;::\; \mathit{Nat} \rightarrow \mathit{Nat} \\
\mathit{perm}\ m \; = \; \mathit{if}\ m == (n+2)\ \mathit{then}\ m \\
\qquad\qquad \mathit{else}\ \mathit{if}\ m == (n+1)\ \mathit{then}\ 1\ \mathit{else}\ m+1
\end{array}
$$

$$
\begin{array}{l}
\mathit{addG} \;::\; SP_{n+2} \rightarrow SP_{n+2} \\
\mathit{addG}\ (S_{n+2}\ x_1\ \cdots\ x_{n+2}) \; = \; \mathit{addE}\ (S_{n+2}\ (\mathit{addG}\ x_1)\ \cdots\ (\mathit{addG}\ x_{n+2})) \\
\mathit{addG}\ (P_{n+2}\ x\ y) \; = \; \mathit{addE}\ (P_{n+2}\ (\mathit{addG}\ x)\ (\mathit{addG}\ y)) \\
\mathit{addG}\ (e_{n+2}(i, j), (l_1, \cdots, l_{n+2})) \; = \; \mathit{addE}\ (e_{n+2}(i, j), (l_1, \cdots, l_{n+2})) \\
\mathit{addG}\ (\mathbf{n+2}, (l_1, \cdots, l_{n+2})) \; = \; \mathit{addE}\ (\mathbf{n+2}, (l_1, \cdots, l_{n+2}))
\end{array}
$$

$$
\begin{array}{l}
\mathit{addE} \;::\; SP_{n+2} \rightarrow SP_{n+2} \\
\mathit{addE}\ x@(t, (l, so_1, \cdots, so_n, l') \\
\quad = \; \mathit{if}\ (l == des_j) || (l' == des_j) \\
\qquad\quad \mathit{then}\ P_{n+2}\ x\ (e_{n+2}(so_j, des_j), (l, so_1, \cdots, so_n, l'))\ \mathit{else}\ x
\end{array}
$$

**Fig. 10.** Transformation of a control flow graph with bounded tree width