Iterative-Free Program Analysis

Mizuhito Ogawa†∗ mizuhito@jaist.ac.jp
Zhenjiang Hu‡∗ hu@mist.i.u-tokyo.ac.jp
Isao Sasano† sasano@jaist.ac.jp

†Japan Advanced Institute of Science and Technology,
‡The University of Tokyo, and
∗Japan Science and Technology Corporation, PRESTO

Abstract

Program analysis is the heart of modern compilers. Most control flow analyses are reduced to the problem of finding a fixed point in a certain transition system, and such fixed point is commonly computed through an *iterative procedure* that repeats tracing until convergence.

This paper proposes a new method to analyze programs through *recursive graph traversals* instead of iterative procedures, based on the fact that most programs (without *spaghetti* GOTO) have well-structured control flow graphs, *graphs with bounded tree width*. Our main techniques are; an algebraic construction of a control flow graph, called *SP Term*, which enables control flow analysis to be defined in a natural recursive form, and the *Optimization Theorem*, which enables us to compute optimal solution by dynamic programming.

We illustrate our method with two examples; dead code detection and register allocation. Different from the traditional standard iterative solution, our dead code detection is described as a simple combination of bottom-up and top-down traversals on SP Term. Register allocation is more interesting, as it further requires optimality of the result. We show how the Optimization Theorem on SP Terms works to find an optimal register allocation as a certain dynamic programming.

Categories and Subject Descriptors

D.1.1 [**Applicative (Functional) Programming**]: Functional Programming; D.1.2 [**Automatic Programming**]: Program Generation; D.3.3 [**Language Constructs and Features**]: Programming with Graphs

General Terms

Algorithms, Language

Keywords

Program Analysis, Control Flow Graph, Register Allocation, Tree Width, SP Term, Dynamic Programming, Catamorphism.

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

1 Introduction

Program analysis is the heart of modern compilers. Most control flow analyses are reduced to the problem of finding a fixed point in a certain transition system. Ordinary method to compute a fixed point is an iterative procedure that repeats tracing until convergence.

Our starting observation is that most programs (without *spaghetti* GOTO) have quite well-structured control flow graphs. This fact is formally characterized in terms of *tree width* of a graph [25]. Thorup showed that control flow graphs of GOTO-free C programs have tree width at most 6 [33], and recent empirical study shows that control flow graphs of most Java programs have tree width at most 3 (though in general it can be arbitrary large) [16].

Once a graph has bounded tree width, we can construct a graph in an algebraic way [3, 4]. This suggests that finding a fixed point would be computed by recursive traversals on the algebraic structure, and the optimal solution would be obtained with a dynamic programming.

Unfortunately, the existing results are not sufficient for our purpose. For instance, the algebraic construction of graphs with bounded tree width treats only undirected graphs [3]. This problem can be easily coped with, but a more serious problem is that it has too many recursive constructors, $k(k+1)(k+2)/6$ for tree width k, which makes it hard to write recursive definitions over it.

This paper proposes a new algebraic construction *SP Term* of graphs with bounded tree width, and a new method to analyze programs through *recursive graph traversals* instead of iterative procedures, based on the fact that most programs (without *spaghetti* GOTO) have well-structured control flow graphs.

Our main theoretical result (Theorem 2) is that a (directed) graph G can be represented by an SP Term in SP_k if and only if G has tree width at most k (and has at least k nodes). Note that SP Term construction reduces the number of recursive constructors to 2 (regardless of the size of tree width k), at the cost of increase of k^2-k+1 constants. These constants express either diedges from the i-th special node (called terminal) to the j-th, or a graph with no edges, and they can be treated in a uniform way. This makes writing recursive definitions on SP terms feasible.

We illustrate our methodology with two examples: dead code detection and register allocation. Different from the traditional standard iterative solution, our dead code detection is described as a simple combination of bottom-up and top-down traversals on an SP Term. Register allocation is more interesting, as it further requires optimality of the result. We solve it as an instance of *maximum marking problems* [26, 27, 5]; mark the nodes of a control flow graph under a certain condition such that the sum of weight of marked nodes is maximum (or, minimum). We make use of *Optimization Theorem* from our previous work [26, 27], and show how it works to find an optimal register allocation as a certain dynamic programming on

SP Terms.

The rest of the paper is organized as follows. We start by an overview of our basic idea in Section 2, through an example of dead code detection on a simple flowchart program without GOTO. Its control flow graph has tree width at most 2, i.e., the class of *series-parallel graphs*.

Section 3 presents an optimal register allocation with the fixed number of registers for a flowchart program. The core of our technique is *Optimization Theorem* [26, 27], which automatically gives an efficient solution for maximum marking problems by certain generic dynamic programming. The advantage and the problem of our method are also briefly discussed.

Section 4 introduces the general definition of SP Term, and demonstrates how to extend dead code detection to a program that has a control flow graph with larger tree width. We show that once the reachability description is given, the description of dead code detection will be uniformly extended to larger tree width.

Section 5 discusses related work, and Section 6 concludes the paper. Throughout the paper, we consider only intra-procedural control flow analyses (0-CFA), and describe algorithms in Haskell-like notations.

2 Dead Code Detection without Iteration

In this section, we explain our idea through a simple case study, dead code detection of flowchart programs. This class of graphs corresponds to the control flow graphs of structured (in strict sense) programs, i.e., programs that consist of single-entry and single-exit blocks.

The syntax of flowchart programs is described below. At the end of the whole program, the end statement is assumed to be added.

Prog :=	$x := e$	assignment
|	*input* x	input statement
|	*output* x	output statement
|	*Prog*; *Prog*	sequence
|	**if** e **then** *Prog* **else** *Prog* **fi**	conditional statement
|	**while** e **do** *Prog* **od**	while loop

Our key to the dead code detection without an iterative procedure is the algebraic construction of control flow graphs, called *SP Term*.

After translation from a flowchart program to an SP Term, we show how to compute the sets of used and newly defined variables in each program fragment by a single bottom-up traversal over an SP Term, and explain how to compute the set of live variables at each terminal in each (sub) SP Term by a single top-down traversal over an SP Term.

2.1 SP Terms for Control Flow Graphs of Flowchart Programs

Algebraic Construction of Series-Parallel Graphs

Control flow graphs of flowchart programs are graphs with tree width at most 2, which are known as *series-parallel directed graphs* (digraphs) [32]. Such graphs can be specified in terms of *SP Term*. Note that the following definition is somewhat simplified compared to that in Section 4.1 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]

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

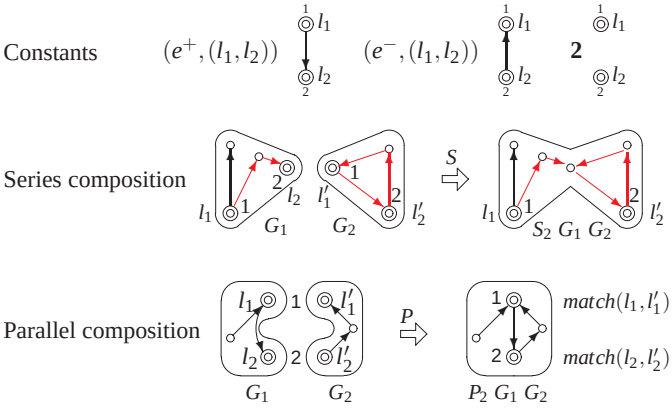

Figure 1. Interpretation of e^+, e^-, 2, S, and P.

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 of two nodes, called *terminals*. We can regard the first terminal as the single-entry, and the second terminal as the next node of the single-exit. Labels (l_1, l_2) are the identifiers of terminals. Let $match(l, l')$ be the function that returns

$$
\begin{cases}
l & \text{if } l = l' \text{ or } l' = * \\
l' & \text{if } l = * \\
\bot & \text{otherwise}
\end{cases}
$$

(i.e., accept the special label $*$ as a wild card during matching).

The constant $(e^+,\ (l_1,l_2))$ is interpreted as a diedge from the first terminal to the second terminal, $(e^-,\ (l_1,l_2))$ as a diedge from the second to the first terminal, and **2** as two isolated terminals. The series composition $S\ (t_1,(l_1,l_2))\ (t_2,(l_1',l_2'))$ fuses the second terminal in t_1 and the first terminal of t_2 if $match(l_2,l_1') \neq \bot$, and regard the first terminal in t_1 as the first and the second 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 terminals in t_1 and t_2 if $match(l_1,l_1'), match(l_2,l_2') \neq \bot$, and label $match(l_1,l_1')$ on the first terminal and $match(l_2,l_2')$ on the second terminal.

The interpretation of each function symbol and constant is described in Fig. 1; a terminal is presented as a double circle, and labels l_1, l_2 are associated to terminals.

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

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

[1] In Section 4.1, e^+, e^-, P_2 are denoted by $e_2(1,2)$, $e_2(2,1)$, and P_2 respectively. For readability, we set $S\ t_1\ t_2 = S_2\ (chT\ t_2)\ t_1$, where S_k is uniformly defined in Section 4.1 for $k \geq 2$.

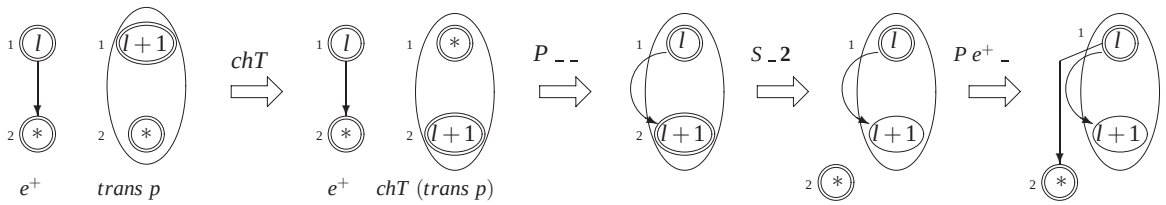

Figure 2. Translation of while statement to an SP Term.

Translation from Programs to SP Terms

We add labels to each statements in *Prog* to identify each node in a control flow graph. We denote the set of such labeled programs by *LProg*. The implementation *trans* of the transformation from a labeled program to an SP Term is given below.

$$
\begin{aligned}
&\mathit{trans} :: \mathit{LProg} \to \mathit{SP}_2\\
&\mathit{trans}\ (l : x := e) = (e^+, (l, *))\\
&\mathit{trans}\ (l : \mathit{input}\ x) = (e^+, (l, *))\\
&\mathit{trans}\ (l : \mathit{output}\ x) = (e^+, (l, *))\\
&\mathit{trans}\ (p_1; p_2) = (S\ (\mathit{trans}\ p_1)\ (\mathit{trans}\ p_2),\ (l, *))\\
&\qquad\qquad \text{where } l \text{ is the starting line of } p_1\\
&\mathit{trans}\ (l : \textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \textbf{fi})\\
&\quad = (P\ (S\ (e^+, (l, *))\ (\mathit{trans}\ p_1,\ (l, *)))\\
&\qquad\quad (S\ (e^+, (l, *))\ (\mathit{trans}\ p_2,\ (l, *))),\ (l, *))\\
&\mathit{trans}\ (l : \textbf{while}\ e\ \textbf{do}\ p\ \textbf{od})\\
&\quad = (P\ (e^+, (l, *))\\
&\qquad\quad (S\ (P\ (e^+, (l, *))\ (\mathit{chT}(\mathit{trans}\ p)\ (*, l+1)),\ (l, l+1))\\
&\qquad\qquad (\mathbf{2}, (*, *)),\ (l, *)),\\
&\qquad\quad (l, *))
\end{aligned}
$$

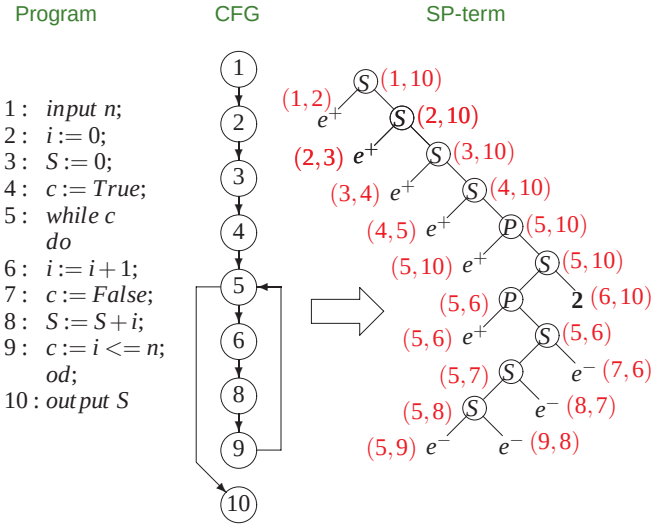

Figure 3. An example of control flow graph and its transformation to SP Term

For instance, the translation of **while**-statement proceeds as in Fig. 2. Intuition behind the wild character label "$*$" is; for each fragment of a program, the first label 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 labeled with "$*$". At the end, $*$ is replaced with the label for the end statement, i.e., the end of the program.

Leaf nodes in an SP Term are either e^+, e^-, and **2**. Each edge in a control flow graph uniquely corresponds to either e^+ or e^-, and each while loop uniquely corresponds to **2**. Thus, the number of leaves in an SP Term is equal to the sum of the number of edges and while loops, which is proportional to the size of a program.[2] This concludes that transformation from a program to an SP Term has (at most) linear growth in size.

Fig. 3 describes the control flow graph of the example program (in Section 1), which computes the sum of $1, 2, \cdots, n$ for an input n, and its transformation to an SP Term by *trans*. In Fig. 3, a tuple associated to each subtree is a tuple of terminals at the interpretation of the subtree.

Note that the description of a control flow graph by an SP Term is *not* unique. For instance, gives an alternative description of the same program in Fig. 3 (Transformation *trans* is already nondeterministic for $p_1; p_2$. Fig. 3 is obtained by the left most decomposition, and Fig. 4 is by the righter most decomposition)

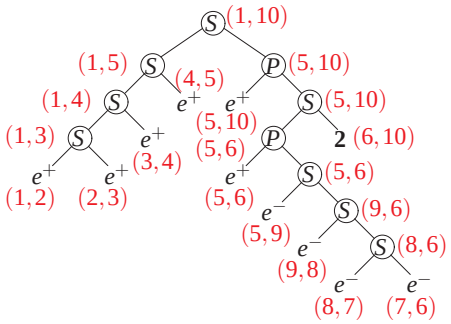

Figure 4. Another equivalent SP Term description

2.2 Dead Code Detection of Flowchart Programs

Our target is dead code detection, i.e., whether defined variables are used before redefined. We use the following functions to extract information from a node labeled l.

$\mathit{defv}\ l$	$= \{x\}$	if the node is an assignment $x := e$ or a input statement $\mathit{input}\ x$.
	$= \_$	if the node is just an expression.
$\mathit{usev}\ l$	$= FV(e)$	if the node is either an assignment $x := e$ or an expression e.
	$= \{x\}$	if the node is an output statement $\mathit{output}\ x$.

Detecting Used and Defined Variables in a Fragment

We first prepare the functions $\mathit{use}_1\ g$, $\mathit{use}_2\ g$, $\mathit{def}_{1\to 2}\ g$, and $\mathit{def}_{2\to 1}\ g$ that detect which variables are used and/or defined in a sub SP Term of g. $\mathit{use}_1\ g$ returns the set of variables that are used

[2] Assuming that tree width is at most k, $|E| \leq k|V|$ where V, E are the set of nodes and edges, respectively [23].

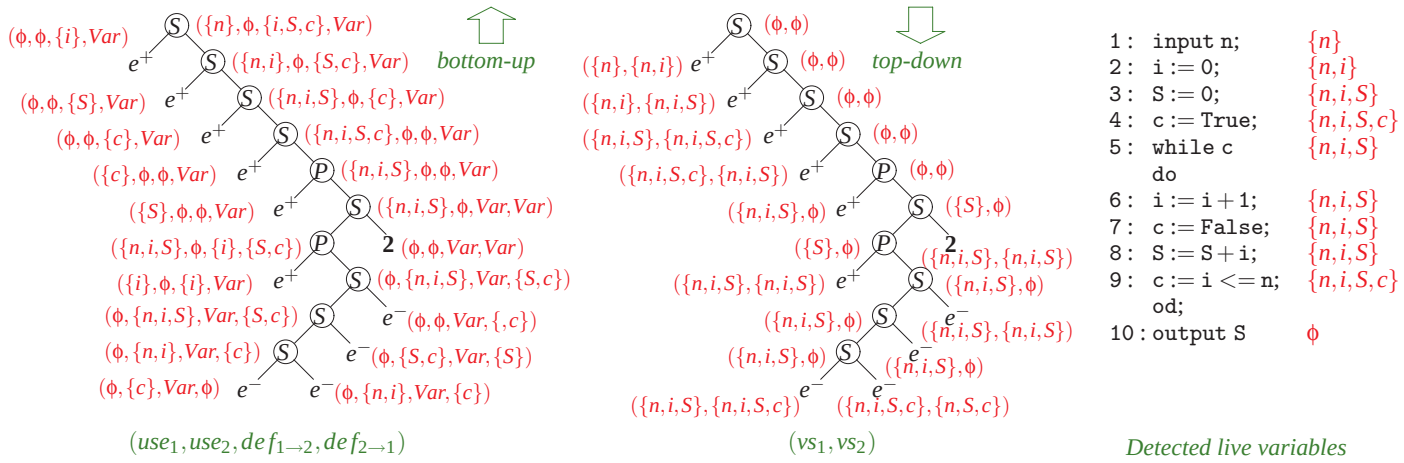

Figure 5. Examples of use_1, use_2, $def_{1\to2}$, $def_{1\to2}$, and $addLive$

before being redefined in some path in g starting from terminal 1.[3] $def_{1\to2}\ g$ returns the set of variables that are newly defined in all paths in g from terminal 1 to terminal 2 (if terminal 2 is reachable from terminal 1); and returns Var (the set of all variables)[4], otherwise. We omit the complementary definitions for use_2 and $def_{2\to1}\ g$.

$$
\begin{array}{l}
use_1\ (e^+,\ (l_1,l_2)) \ =\ usev\ l_2\\
use_1\ (e^-,\ \_) \ =\ \phi\\
use_1\ (\mathbf{2},\ \_) \ =\ \phi\\
use_1\ (S\ x\ y,\ \_) \ =\ use_1\ x \cup (use_1\ y \setminus def_{1\to2}\ x)\\
use_1\ (P\ x\ y,\ \_) \ =\ use_1\ x \cup (use_2\ y \setminus def_{1\to2}\ x) \cup\\
\qquad\qquad\qquad\quad use_1\ y \cup (use_2\ x \setminus def_{1\to2}\ y)
\end{array}
$$

$$
\begin{array}{l}
def_{1\to2}\ (e^+,\ (l_1,l_2)) \ =\ defv\ l_2\\
def_{1\to2}\ (e^-,\ \_) \ =\ Var\\
def_{1\to2}\ (\mathbf{2},\ \_) \ =\ Var\\
def_{1\to2}\ (S\ x\ y,\ \_) \ =\ def_{1\to2}\ x \cup def_{1\to2}\ y\\
def_{1\to2}\ (P\ x\ y,\ \_) \ =\ def_{1\to2}\ x \cap def_{1\to2}\ y
\end{array}
$$

Note that by tupling use_1, use_2, $def_{1\to2}$, and $def_{2\to1}$, we can compute the sets of live variables at terminal 1 and 2 in each sub SP Term of g by a single bottom-up traversal on g [18].

Live Variable Detection without Iteration

After the computation of use_1, use_2, $def_{1\to2}$, and $def_{2\to1}$, we assume that each sub SP Term in g has additional information of the results of these functions.

Next we give a function $addLive$ that associates the information of live variables to each terminal in each sub SP Term in g. The function $addLlive$ takes an SP Term g and two sets of variables vs_1 and vs_2 (both with the initial value of ϕ), where vs_1 denotes live variables outgoing from g at terminal 1 and vs_2 denotes live variables outgoing from g at terminal 2. It returns a pair of an SP Term and a tuple of the sets of variables that are alive at the terminal 1 and 2.

$$
\begin{array}{l}
addLive\ (e^+,\ (l_1,l_2))\ vs_1\ vs_2\\
\quad =\ (e^+,\ (l_1,\ l_2,\ vs_1 \cup usev\ l_2 \cup (vs_2 \setminus defv\ l_2),\ vs_2))\\
addLive\ (e^-,\ (l_1,l_2))\ vs_1\ vs_2\\
\quad =\ (e^-,\ (l_1,\ l_2,\ vs_1,\ vs_2 \cup usev\ l_1 \cup (vs_1 \setminus defv\ l_1)))\\
addLive\ (\mathbf{2},\ (l_1,l_2))\ vs_1\ vs_2\ =\ (\mathbf{2},\ (l_1,l_2,\ vs_1,vs_2))
\end{array}
$$

[3] $use_1\ g$ omits the used variables at terminal 1.

[4] Var satisfies $X \cap Var = X$, $X \cup Var = Var$, and $X \setminus Var = \phi$ for each set X of variables.

```
1 : input n;          ⇐ defv(l1) = {n} ⊆ {n}
2 : i := 0;           ⇐ defv(l2) = {i} ⊆ {n,i}
3 : S := 0;           ⇐ defv(l3) = {S} ⊆ {n,i,S}
4 : c := True;        ⇐ defv(l4) = {c} ⊆ {n,i,S,c}
5 : while c
    do
6 : i := i + 1;       ⇐ defv(l6) = {i} ⊆ {n,i,S}
7 : c := False;       ⇐ defv(l7) = {c} ⊈ {n,i,S}
8 : S := S + i;       ⇐ defv(l8) = {S} ⊆ {n,i,S}
9 : c := i <= n;      ⇐ defv(l9) = {c} ⊆ {n,i,S,c}
    od;
10 : output S
```

Figure 6. Dead code detection of a flow chart program

$$
\begin{array}{l}
addLive\ (S\ x\ y,\ (l_1,l_2))\ vs_1\ vs_2\\
\quad =\ (S\ (addLive\ x\ vs_1\ (use_1\ y \cup (vs_2 \setminus def_{1\to2} y)))\\
\qquad\quad (addLive\ y\ (use_2\ x \cup (vs_1 \setminus def_{2\to1} x))\ vs_2),\\
\qquad (l_1,\ l_2,\ vs_1,\ vs_2))\\
addLive\ (P\ x\ y,\ (l_1,l_2))\ vs_1\ vs_2\\
\quad =\ (P\ (addLive\ x\\
\qquad\quad (vs_1 \cup use_1\ y \cup ((vs_2 \cup use_2\ x) \setminus def_{1\to2}\ y))\\
\qquad\quad (vs_2 \cup use_2\ y \cup ((vs_1 \cup use_1\ x) \setminus def_{2\to1}\ y)))\\
\qquad (addLive\ y\\
\qquad\quad (vs_1 \cup use_1\ x \cup ((vs_2 \cup use_2\ y) \setminus def_{1\to2}\ x))\\
\qquad\quad (vs_2 \cup use_2\ x \cup ((vs_1 \cup use_1\ y) \setminus def_{2\to1}\ x))),\\
\qquad (l_1,\ l_2,\ vs_1,\ vs_2))
\end{array}
$$

With the assumption that use_1, use_2, $def_{1\to2}$, and $def_{2\to1}$ are computed and their results are stored, $addLive$ is done in a single top-down traversal on g.

Fig. 5 shows computation of $(use_1,use_2,def_{1\to2},def_{1\to2})$ and $addLive$ on a control flow graph in Fig. 3. At the terminals in a leaf in an SP Term, the detected set of live variables at each node is obtained.

Dead Code Detection of Flowchart Programs

Now that the set of live variables at each node in a control flow graph has been computed, dead code detection is straightforward. A variable is *dead* if it is not live. Dead code is an assignment that assigns a value to a dead variable. Thus, in the example in Fig. 5, the assignment Z:=X+2 at line 8 is a dead code, since Z is dead as in shown in Fig. 6.

3 Register Allocation for Flowchart Program

In this section, we show how to find an optimal register allocation as an instance of a *maximum marking problem*. Our strategy is, first write down the finite mutumorphic specification *checking* whether marking represents correct register allocation, and the weight w that counts the number of required LOAD/STORE instructions. Second, transform *checking* to the form with $foldSP$ by tupling transformation [18]. Then, if w is *homomorphic*, Optimization Theorem (Theorem 1 [27, 26]) automatically gives how to detect an optimal register allocation with certain generic dynamic programming (i.e., a single traversal on an SP Term), assuming the live variables are pre-computed. Note that we restrict ourselves to control flow graphs with bounded tree width, and do not intend $P = NP$, where the conventional optimal register allocation based on graph coloring [10] is *NP*-complete.

For simplicity, we consider a flowchart programs (without GOTO) as in Section 2. We assume that functions $defv\ l$, $usev\ l$, and live variables at terminal labeled l are pre-computed (as in Section 2).

3.1 Register Allocation

In a real computer, an instruction is executed with values on limited number of registers. If needed inputs are not on registers, then they must be *loaded* from memory; and if there are no room for them, some values on registers must be *stored*. These LOAD/STORE instructions are usually expensive, and register allocation is an optimization that under fixed number of registers, find an optimal register usage, i.e., a program execution with the minimum use of LOAD (from memory to register) and STORE (from register to memory).

Basic operations on registers are either LOAD, STORE, *move*, or execution of an instruction. When the number of registers is 4, for instance, we have

LOAD x r_2	y _ z _	→	y x z _
STORE x r_2	y x z _	→	y _ z _
$r_4 := r_1$ (move)	y x z _	→	_ x z y
$r_4 := r_1 + r_2$	y x z _	→	y x z u

where $u = y + x$

For simplicity, we only concentrate on the number of LOAD/STORE instructions, and do not care on the number of *move*.

Fig. 7 shows the optimal register allocation for a simple program (appeared in Section 1), which computes the sum of 1 to n with 3 registers.[5] Here, each tuple of three variables represents a register allocation just before each instruction is executed. The special symbol _ means that the register is either empty or permitted to overwritten. Note that between line 7 and 8, STORE c needs not to be inserted; instead we just overwrite S on c. This is correct, because c is dead at line 7.

3.2 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 elements in x such that the marked data satisfies a certain property p and has the maximum (or, equivalently, minimum) 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.

[5] Strictly speaking, LOAD/STORE instruction must be inserted at the machine code level, but for simplicity we just insert LOAD/STORE instructions into a flowchart program.

	instruction	register	live variables
1 :	*input n*;	$(\_,\_,\_)$	$\{n\}$
2 :	$i := 1$;	$(n,\_,\_)$	$\{n,i\}$
3 :	$S := 0$;	$(n,i,\_)$	$\{n,i,S\}$
	$STORE\ S$	(n,i,S)	
4 :	$c := True$;	$(n,i,\_)$	$\{n,i,S,c\}$
5 :	*while c*	(n,i,c)	$\{n,i,S\}$
	do		
6 :	$i := i+1$;	(n,i,c)	$\{n,i,S\}$
7 :	$c := False$;	(n,i,c)	$\{n,i,S\}$
	$LOAD\ S$	(n,i,c)	
8 :	$S := S+i$;	(n,i,S)	$\{n,i,S\}$
	$STORE\ S$	(n,i,S)	
9 :	$c := i <= n$;	$(n,i,\_)$	$\{n,i,S,c\}$
	od;		
	$LOAD\ S$	(n,i,c)	
10 :	*output S*	(n,i,S)	$\{\}$

Figure 7. An example of optimal register allocation

MMP includes many interesting problems, such as knapsack problems, and optimized range problems in data mining [28]. Of course, it is not expected that every MMP problem can be solved efficiently. In fact, MMP includes NP-hard problems, such as the knapsack problem. However, for instance, the knapsack problem restricted to integer weight can be computed in linear time.

Let us consider more formally. The specification of MMP is described as follows, where constraints are expressed by a boolean-valued function p and a weight function w.

$$mmp\ w\ p = selectmax\ w \circ filter\ p \circ gen$$

The function *gen* generates all possible marking on elements:

$$gen \quad : \quad D \to \{D^*\}$$

D^* is the data structure derived from D where each node is attached with a mark. The function $filter\ p$ takes a set of marked data and selects ones that satisfy the property p. The function $selectmax\ w$ takes a set of marked data and select one that has the maximum value with respect to the weight function w. Then, we can derive a linear time algorithm mechanically if the property p is defined by finite mutumorphisms, and the weight function w is homomorphic [26].

Mutumorphism is a set of mutually recursive functions, among which no nested function calls occur and each argument of recursive call is a sub-structure of the input [15]. Note that by tupling transformation, mutumorphism is transformed to a single catamorphism [18]. Although mutumorphism is defined on more general data structures, from now on, we will consider SP Terms only.

DEFINITION 2 (FINITE MUTUMORPHIC PROPERTY [27]). *A property p is* finite mutumorphic *if it is defined by*

$$\begin{array}{lcl} p & : & SP^* \to Bool \\ p\ (e^+,\ a) & = & \phi_{e^+}\ a \\ p\ (e^-,\ a) & = & \phi_{e^-}\ a \\ p\ (\mathbf{2},\ a) & = & \phi_{\mathbf{2}}\ a \\ p\ (S\ x_1\ x_2,\ a) & = & \phi_S\ (h\ x_1)\ (h\ x_2)\ a \\ p\ (P\ x_1\ x_2,\ a) & = & \phi_P\ (h\ x_1)\ (h\ x_2)\ a \end{array}$$

where $h\ x = (p\ x,\ f_1\ x,\ f_2\ x, \ldots,\ f_m\ x)$, which may use auxiliary

functions $f_1, \ldots, f_m$ each of which has finite range of C_i.

$$\begin{array}{lcl} f_i & : & SP^* \to C_i \\ f_i\ (e^+,\ a) & = & \phi_{ie^+}\ a \\ f_i\ (e^-,\ a) & = & \phi_{ie^-}\ a \\ f_i\ (\mathbf{2},\ a) & = & \phi_{i\mathbf{2}}\ a \\ f_i\ (S\ x_1\ x_2,\ a) & = & \phi_{iS}\ (h\ x_1)\ (h\ x_2)\ a \\ f_i\ (P\ x_1\ x_2,\ a) & = & \phi_{iP}\ (h\ x_1)\ (h\ x_2)\ a \end{array}$$

If p is finite mutumorphic, tupling transformation [18] will yield a catamorphism for h. Therefore a finite mutumorphic property p can be described in the form of

$$p = fst \circ foldSP\ p_{e^+}\ p_{e^-}\ p_{\mathbf{2}}\ p_S\ p_P$$

where fst is the function that takes the first element in a tuple and the fold (catamorphism) operation $foldSP$ on SP terms is defined below.

$$\begin{array}{l} foldSP\ \varphi_{e^+}\ \varphi_{e^-}\ \varphi_{\mathbf{2}}\ \varphi_S\ \varphi_P = \varphi \\ \quad \text{where} \quad \varphi\ (e^+,\ a) = \varphi_{e^+}\ a \\ \qquad\qquad \varphi\ (e^-,\ a) = \varphi_{e^-}\ a \\ \qquad\qquad \varphi\ (\mathbf{2},\ a) = \varphi_{\mathbf{2}}\ a \\ \qquad\qquad \varphi\ (S\ x_1\ x_2,\ a) = \varphi_S\ (\varphi\ x_1)\ (\varphi\ x_2)\ a \\ \qquad\qquad \varphi\ (P\ x_1\ x_2,\ a) = \varphi_P\ (\varphi\ x_1)\ (\varphi\ x_2)\ a \end{array}$$

To be concrete, recall the dead code detection in Section 2.2, where we have reached the point that each node is added with a set of live variables. Assume that some nodes in the graph are marked (which can be checked by *isM*.) Now we may define the property *md* by $foldSP$ that all marked nodes in the graph are dead.

$$\begin{array}{l} md = foldSP\ \varphi_1\ \varphi_1\ \varphi_1\ \varphi_2\ \varphi_2 \\ \quad \text{where} \\ \qquad \varphi_1\ (l_1, l_2, vs_1, vs_2) = valid\ (l_1, vs_1) \wedge valid\ (l_2, vs_2) \\ \qquad \varphi_2\ p_1\ p_2\ a = p_1 \wedge p_2 \wedge \varphi_1\ a \end{array}$$

Here *valid* is to determine whether a marked terminal node is dead, i.e., $valid\ (l, vs) =$ if *isM l* then *defv* $l \subseteq vs$ else *True*.

DEFINITION 3 (HOMOMORPHIC WEIGHT FUNCTION [26]). *A weight function w is* homomorphic *if w is defined as a fold*

$$\begin{array}{l} w : SP^* \to Weight \\ w = foldSP\ \psi_{e^+}\ \psi_{e^-}\ \psi_{\mathbf{2}}\ \psi_S\ \psi_P \end{array}$$

where ψ_S and ψ_P is described as a summation in a form like

$$\begin{array}{l} \psi_S\ r_1\ r_2\ a = r_1 + r_2 + v_S\ a \\ \psi_P\ r_1\ r_2\ a = r_1 + r_2 + v_P\ a \end{array}$$

for some functions v_S and v_P.

Continuing with the dead code detection problem, we may define a weight function *nd* to count the number of the marked dead nodes[6]

$$\begin{array}{l} nd = foldSP\ \psi_1\ \psi_1\ \psi_1\ \psi_2\ \psi_2 \\ \quad \text{where} \\ \qquad \psi_1\ (l_1, l_2, vs_1, vs_2) = c\ l_1 + c\ l_2 \\ \qquad \psi_2\ p_1\ p_2\ a = p_1 + p_2 \end{array}$$

Here $c\ l$ returns 1 if the node l is marked, and 0 otherwise.

THEOREM 1 (OPTIMIZATION THEOREM [26, 27]). *If the property p is finite mutumorphic and the weight function w is homomorphic, MMP specified by*

$$\begin{array}{lcl} spec & : & SP \to SP^* \\ spec & = & mmp\ w\ p \end{array}$$

has an $O(|C'| \cdot n)$ algorithm described as

$$opt\ \psi_{e^+}\ \psi_{e^-}\ \psi_{\mathbf{2}}\ \psi_S\ \psi_P\ fst\ p_{e^+}\ p_{e^-}\ p_{\mathbf{2}}\ p_S\ p_P$$

where $C' = C_1 \times \cdots \times C_m$ and n is the size of an input.

The core of Optimization Theorem is a generic dynamic programming. The idea is; during data traversal, compute intermediate maxima for all possible states that may contribute to the finial maximum. Finite mutumorphisms $f_1, \ldots, f_m$ describe state transition, and finiteness of their ranges guarantee that such states are finite. For the definition of *opt* and detail, refer to [27, 26].

It follows from this theorem that we can detect all dead codes with the property *md* and the weight function *nd* by the following program.

$$opt\ \psi_1\ \psi_1\ \psi_1\ \psi_2\ \psi_2\ id\ \varphi_1\ \varphi_1\ \varphi_1\ \varphi_2\ \varphi_2$$

We will see a more interesting application of the theorem in the next Section.

3.3 Optimal Register Allocation as MMP

As an application of Optimization Theorem, we demonstrate the register allocation problem.

Check Whether Each Terminal Has a Correct Mark

Let *Var* be the set of variables that appears in a program, and let _ be the special symbol that means a register is either empty or ready to overwrite. The set *Reg* of register allocations (we consider the size of registers is three) is defined as:

$$\begin{array}{rl} Reg\ = & \{(v_1, \cdots, v_n) \mid v_i \in Var \cup \{\_\}, \\ & \qquad v_i \neq v_j\ \vee\ v_i = v_j = \_ \quad \text{if } i \neq j\}. \end{array}$$

An element in *Reg* is labeled to each terminal in an SP Term as a mark, which represents the register allocation state just before the instruction at the terminal being executed. Below, we will describe

- *checking*, which *checks* whether each terminal has a correct mark, and
- *w*, which counts the number of required LOAD/STORE instructions under a certain marking of the program.

A mark in *Reg* is a tuple, and we use the following operations (as analogy to set operations). Let $r = (v_1, \cdots, v_n)$, $r' = (v'_1, \cdots, v'_n) \in Reg$.

$$\begin{array}{l} r \setminus\setminus r' = (v''_1, \cdots, v''_n) \quad \text{where } v''_i = \text{if } v_i = v'_i \text{ then } \_ \text{ else } v_i \\ RV\ r\ \ = \{v_1, \cdots, v_n\} \setminus \{\_\} \end{array}$$

The function *checking g* takes a marked SP-term associated with the line numbers in a program (denoted by l_1, l_2), the sets of live variables (denoted by vs_1 and vs_2), and the marking that represents (pre-execution) register status (denoted by m_1 and m_2) at terminal 1 and 2, and returns a Boolean value.

$$\begin{array}{l} checking\ (e^+,\ ((l_1, l_2, vs_1, vs_2),\ (m_1, m_2))) \\ \quad = ch\ ((l_1, l_2,\ vs_1, vs_2),\ (m_1, m_2)) \\ checking\ (e^-,\ ((l_1, l_2, vs_1, vs_2),\ (m_1, m_2))) \\ \quad = ch\ ((l_1, l_2,\ vs_1, vs_2),\ (m_1, m_2)) \\ checking\ (\mathbf{2},\ ((l_1, l_2,\ vs_1, vs_2),\ (m_1, m_2))) \\ \quad = ch\ ((l_1, l_2,\ vs_1, vs_2),\ (m_1, m_2)) \end{array}$$

[6] This is not exactly true. In fact, all marked dead nodes except for the two terminal nodes of the whole graph are counted twice.

$$
\begin{array}{l}
\mathit{checking}\ (S\ x\ y,\ ((l_1,l_2,\ vs_1,vs_2),\ (m_1,m_2))) \\
\quad =\ \mathit{let}\ (m_1',m_2') = \mathit{getMarks}\ x \\
\qquad\quad (m_1'',m_2'') = \mathit{getMarks}\ y \\
\qquad \mathit{in}\ \mathit{checking}\ x\ \wedge\ \mathit{checking}\ y \\
\qquad\quad \wedge\ m_1 = m_1'\ \wedge\ m_2 = m_2''\ \wedge\ m_2' = m_1'' \\
\mathit{checking}\ (P\ x\ y,\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2))) \\
\quad =\ \mathit{let}\ (m_1',m_2') = \mathit{getMarks}\ x \\
\qquad\quad (m_1'',m_2'') = \mathit{getMarks}\ y \\
\qquad \mathit{in}\ \mathit{checking}\ x\ \wedge\ \mathit{checking}\ y \\
\qquad\quad \wedge\ m_1 = m_1'\ \wedge\ m_1 = m_1''\ \wedge\ m_2 = m_2'\ \wedge\ m_2 = m_2'' \\
\mathit{ch}\ ((l_1,l_2,\ vs_1,vs_2),\ (m_1,m_2)) \\
\quad =\ \mathit{usev}\ l_1 \subseteq \mathit{RV}\ m_1 \subseteq (vs_1 \cup \mathit{usev}\ l_1)\ \wedge \\
\qquad \mathit{usev}\ l_2 \subseteq \mathit{RV}\ m_2 \subseteq (vs_2 \cup \mathit{usev}\ l_2)\ \wedge \\
\qquad |\mathit{defv}\ l_1| + |\mathit{RV}\ m_1 \cap vs_1 \setminus \mathit{defv}\ l_1| \leq n\ \wedge \\
\qquad |\mathit{defv}\ l_2| + |\mathit{RV}\ m_2 \cap vs_2 \setminus \mathit{defv}\ l_2| \leq n \\
\mathit{getMarks}\ (t,a)\ =\ \mathit{snd}\ a
\end{array}
$$

The judgment $\mathit{usev}\ l_1 \subseteq \mathit{RV}\ m_1$ corresponds to the *pre-condition* of the instruction l_1 at terminal 1 in g, i.e., each variable used in the instruction must be in some register in m_1, and $|\mathit{defv}\ l_1| + |(m_1 \cap vs_1) \setminus \mathit{defv}\ l_1| \leq n$ corresponds to the *post-condition*, i.e., m_1 has a room to write defined variables ($\mathit{defv}\ l_1$); otherwise, some live variables in m_1 except for those defined at l_1 will be overwritten before stored. Notice the obvious optimizing conditions

$$
\begin{array}{l}
\mathit{RV}\ m_1 \subseteq vs_1 \cup \mathit{usev}\ l_1 \\
\mathit{RV}\ m_2 \subseteq vs_2 \cup \mathit{usev}\ l_2
\end{array}
$$

in $\mathit{ch}\ ((l_1,l_2,vs_1,vs_2),(m_1,m_2))$, which mean that live variables in registers are as many as possible.

The *checking* property is defined as finite mutumorphisms with the function *getMarks*. By tupling transformation, we get the following form:

$$
\mathit{checking} = \mathit{fst} \circ \mathit{foldSP}\ p_{e^+}\ p_{e^-}\ p_{\mathbf{2}}\ p_S\ p_P
$$

where

$$
\begin{array}{l}
p_{e^+}\ (x,\ a) = (\mathit{ch}\ a,\ \mathit{snd}\ a) \\
p_{e^-}\ (x,\ a) = (\mathit{ch}\ a,\ \mathit{snd}\ a) \\
p_{\mathbf{2}}\ (x,\ a) = (\mathit{ch}\ a,\ \mathit{snd}\ a) \\
p_S\ x\ y\ a \\
\quad =\ \mathit{let}\ (m_1',m_2') = \mathit{snd}\ x\ \ (m_1'',m_2'') = \mathit{snd}\ y\ \ (m_1,m_2) = \mathit{snd}\ a \\
\qquad \mathit{in}\ (\mathit{fst}\ x \wedge \mathit{fst}\ y\ \wedge\ m_1 = m_1'\ \wedge\ m_2 = m_2''\ \wedge\ m_2' = m_1'', \\
\qquad\quad (m_1,m_2)) \\
p_P\ x\ y\ a \\
\quad =\ \mathit{let}\ (m_1',m_2') = \mathit{snd}\ x\ \ (m_1'',m_2'') = \mathit{snd}\ y\ \ (m_1,m_2) = \mathit{snd}\ a \\
\qquad \mathit{in}\ (\mathit{fst}\ x\ \wedge\ \mathit{fst}\ y\ \wedge\ (m_1 = m_1'\ \wedge\ m_1 = m_1'') \\
\qquad\qquad\qquad\qquad \wedge\ (m_2 = m_2'\ \wedge\ m_2 = m_2''), \\
\qquad\quad (m_1,m_2))
\end{array}
$$

Here we include obvious optimizing conditions

$$
\begin{array}{l}
\mathit{RV}\ m_1 \subseteq vs_1\ \cup\ \mathit{usev}\ l_1 \\
\mathit{RV}\ m_2 \subseteq vs_2\ \cup\ \mathit{usev}\ l_2
\end{array}
$$

to $\mathit{ch}\ (l_1,l_2,vs_1,vs_2,m_1,m_2)$, which mean that live variables in registers are as many as possible.

Weight Counts the Number of Required LOAD/STORE

The weight function w is defined as follows.

$$
\begin{array}{l}
w\ (e^+,\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2)))\ =\ \mathit{count}\ l_1\ vs_1\ m_1\ m_2 \\
w\ (e^-,\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2)))\ =\ \mathit{count}\ l_2\ vs_2\ m_2\ m_1 \\
w\ (\mathbf{2},\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2)))\ =\ 0 \\
w\ (S\ x\ y,\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2)))\ =\ w\ x + w\ y \\
w\ (P\ x\ y,\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2)))\ =\ w\ x + w\ y
\end{array}
$$

$$
\begin{array}{l}
\mathit{count}\ l\ vs\ m\ m' \\
\quad =\ \mathit{let}\ V = (\mathit{RV}\ m \cap vs) \cup \mathit{defv}\ l \\
\qquad \mathit{in}\ \mathit{if}\ V\ \not\subseteq\ \mathit{RV}\ m' \\
\qquad\quad \mathit{then}\ \mathit{max}|(V \setminus \mathit{RV}\ m') \cap vs| + |\mathit{RV}\ m' \setminus V| \\
\qquad\quad \mathit{else}\ \mathit{if}\ |V| < n\ \mathit{then}\ |\mathit{RV}\ m' \setminus V| \\
\qquad\qquad\qquad\qquad \mathit{else}\ \mathit{if}\ \mathit{RV}\ (m'\ \backslash\backslash\ m) = \phi\ \mathit{then}\ 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ 2
\end{array}
$$

The function w counts the number of required LOAD/STORE at each edge (uniquely represented by e^+ or e^-), and just sums up for recursive constructors S and P.

The intuition for *count* is: V is the set of variables that are placed on the registers after an instruction is executed. If V is not included in the next register status m', then their difference must be stored and loaded. Between *STORE* and *LOAD* operations, we can reorder the positions of variables by *move* operations. Assume V is included in the next register status m'. If $|V| < n$, this means there exists a register with _, and we can reorder variables in V. Otherwise, $V = \mathit{RV}\ m'$, and if $\mathit{RV}\ (m'\ \backslash\backslash\ m) \neq \phi$ we need to make room by a pair of *STORE* and *LOAD* operations for reordering.

The above definition of the weight function w is homomorphic, and w is defined by foldSP as follows.

$$
\begin{array}{l}
\psi_{e^+}\ a \\
\quad =\ \mathit{let}\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2)) = a\ \ \mathit{in}\ \ \mathit{count}\ l_1\ vs_1\ m_1\ m_2 \\
\psi_{e^-}\ a \\
\quad =\ \mathit{let}\ ((l_1,l_2,vs_1,vs_2),\ (m_1,m_2)) = a\ \ \mathit{in}\ \ \mathit{count}\ l_2\ vs_2\ m_2\ m_1 \\
\psi_{\mathbf{2}}\ a\ =\ 0 \\
\psi_S\ x\ y\ a\ =\ x + y \\
\psi_P\ x\ y\ a\ =\ x + y
\end{array}
$$

Applying Optimization Theorem, and Discussion

With all the above, Theorem 1 automatically derives the solution for optimal register allocation.

At last, two points are worth remarking:

1. In real compilers, there are often practical requirements of hardware, such as, some instruction must use some specific registers, some register must be used together with some specific registers, or the result of some instruction must be written in a different register. These requirements are hard for the conventional graph coloring method [10], but our method is easy to handle them by modifying the function *checking*.
2. We obtain an optimal register allocation without iteration. The core of the technique is dynamic programming on SP Terms. The cost to pay is huge marking space, which grows exponentially to the number of registers. However, since *checking* can be judged locally (like forall in Haskell), most of marking is avoided by default. We expect demand-driven computation.helps the situation.

4 Analyzing Control Flow Graphs with Larger Tree Width

In this section, we discuss how our method can be extended to wider class of programs. The example is again dead code detection; but for a program that has a control flow graph with tree width larger than 2. For simplicity, we mostly consider control flow graphs with tree width at most 3. Our construction is uniform and extension for larger tree width is straight forward, if we assume the description on reachability among terminals.

Due to lack of space, we do not explain *tree decomposition*, which gives the original definition of tree width [25]. Instead, we treat a (di)graph with tree width at most k as a (di)graph denoted by an SP Term in SP_k.

4.1 SP Terms

In Section 2.1, 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 give definition of general SP Term (for graphs with larger tree width) and show that translation will be done in linear time.

DEFINITION 4. *An SP Term is a pair of a ground term t and a tuple $(l(1),\cdots,l(k))$, defined as the following.*

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

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

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 G, G' with terminals $(l(1),\cdots,l(k)))$ and

$$\begin{cases} V(G)=\{l(1),\cdots,l(k)\}, & E(G)=\{(l(i),l(j)\}, \\ V(G')=\{l(1),\cdots,l(k)\}, & E(G')=\phi. \end{cases}$$

i.e., k-nodes $l(1),\cdots,l(k)$ with one diedge from $l(i)$ to $l(j)$, and i.e., k isolated nodes $l(1),\cdots,l(k)$, respectively (See Fig. 8).

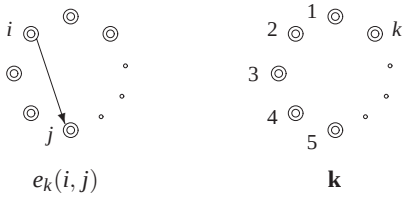

Figure 8. Interpretation of constants $e_k(i,j)$ and k

Series composition $(S_k\ t_1\ \cdots\ t_k,\ (l(1),\cdots,l(k)))$ is interpreted in 3 steps. See Fig. 9 (In Fig. 9 and 10, a double circle expresses a terminal). Let $t_i=(t_i',\ l_i(1),\cdots,l_i(k))$.

1. Shift the numbering of terminals in t_i, i.e., the j-th terminal to the $j+1$-th terminal for each j with $i \leq j \leq k$.
2. Fuse each terminal of the same numbering and put a label

   $$match\ (l_1(i-1),\cdots,l_{i-1}(i-1),l_{i+1}(i),\cdots,l_k(i))$$

   to the i-th terminal if

   $$match(l_1(i-1),\cdots,l_{i-1}(i-1),l_{i+1}(i),\cdots,l_k(i)) \neq \perp .$$

3. Remove the last terminal (labeled \$ in Fig. 9).

where *match* $(l(1),\cdots,l(k))$ is an abbreviation of

$$match\ (l(1),match\ (l(2),\cdots,match\ (l(k-1),l(k))\cdots).$$

Parallel composition $(P_k\ t_1\ t_2,\ (l(1),\cdots,l(k)))$ is interpreted similar to P in Section 2; fuse each terminal of the same numbering in $t_1=(t_1',\ l_1(1),\cdots,l_1(k))$ and $t_2=(t_2',\ l_2(1),\cdots,l_2(k))$, and put a label *match* $(l_1(i),l_2(i))$ to the i-th terminal if *match* $(l_1(i),l_2(i)) \neq \perp$. See Fig. 10.

Intuition behind is; like that the parallel composition p_k constructs any subgraph in a complete graph K_k, 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}).

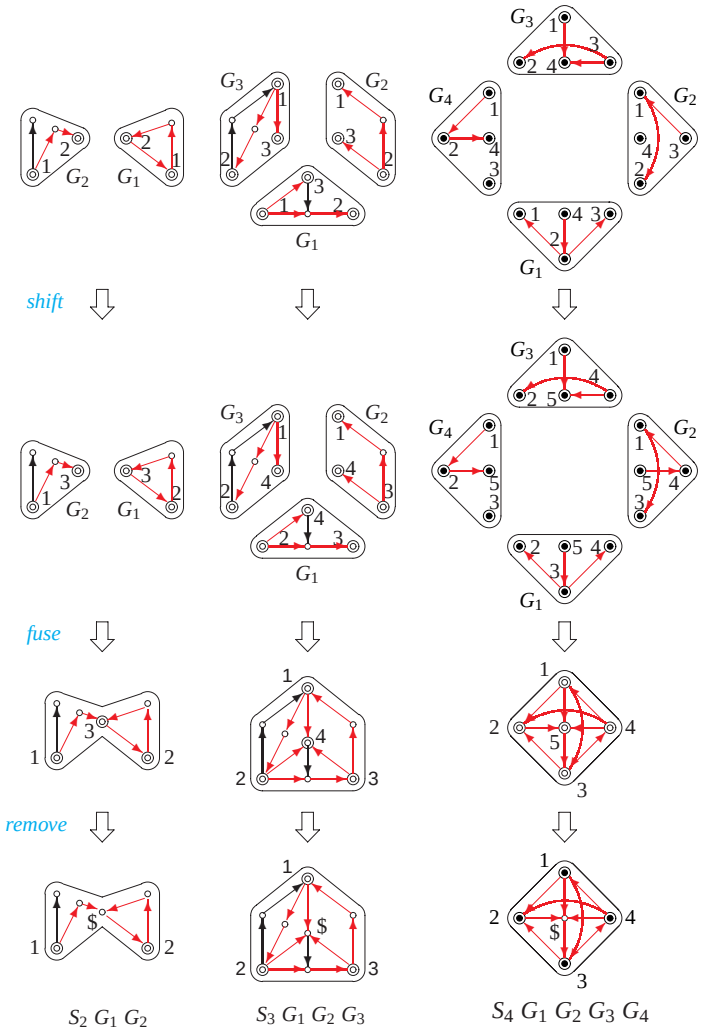

Figure 9. Interpretation of series composition S_2, S_3, S_4

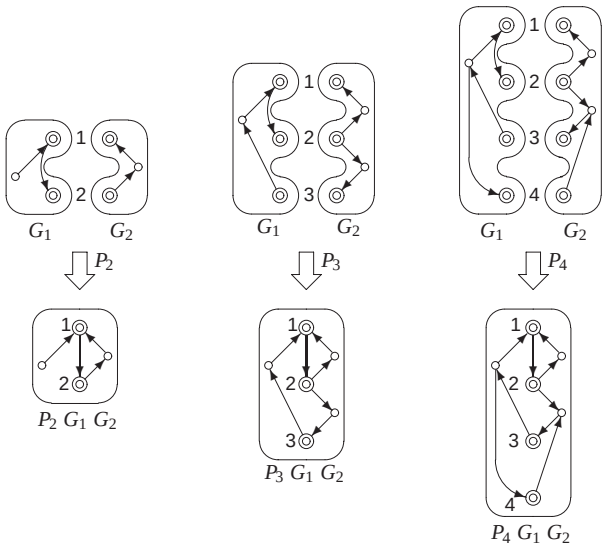

Figure 10. Interpretation of parallel composition P_2, P_3, P_4

EXAMPLE 1. In Fig. 9, 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

$$\begin{array}{l} G_1 = S_3(P_3(e_3(2,3),e_3(3,1)),P_3(e_3(1,2),e_3(1,3)),\mathbf{3}) \\ G_2 = S_3(P_3(e_3(1,2),e_3(1,3)),e_3(3,1),\mathbf{3}) \\ G_3 = S_3(\mathbf{3},e_3(1,2), \\ \qquad S_3(P_3(e_3(1,2),e_3(1,3)),P_3(e_3(1,2),e_3(3,1)),\mathbf{3})) \end{array}$$

REMARK 1. *SP_2 (series parallel graphs) allows several choices for definition of the series composition. For instance, definition of S in Section 2 is different from S_2 here; they can be related as*

$$S\ x\ y = chT(S_2\ x\ (chT\ y)).$$

S_2 is the part of uniform way to define the series composition for each SP_k; however, for readability, we used the simplified version S in Section 2.

REMARK 2. *The definition of series composition S_k and parallel composition P_k are given by Arnborg, et.al. in a different aspect of an algebraic construction of graphs with bounded tree width [3]. Note that if once an SP Term t is given, tree decomposition [25] of a corresponding graph is straightforward: a backborn tree T as $V(T) = \{s \mid s \subseteq t\}$ and a covering X_s for $s \in V(T)$ as the set of terminals in s.*

THEOREM 2. *Let G be a digraph 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 interpretation is a pair of k-terminal digraph $\tilde{G}$ and a tuple of k-terminals with $G = \tilde{G}$ by neglecting terminals.*

In general, deciding the tree width of a graph is NP-complete; however, for fixed k, whether a graph has tree width at most k is decidable in linear time [7, 24]. Fortunately, we already know the upper bound of the tree width of control flow graphs of some specific programming languages.

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. However, a direct translation (such as *trans* in Section 2.1) from a program will be much more efficient, because a control flow graph loses the parsing information of an original program. For a simple imperative language with GOTO, such a translation is shown in Appendix A.

4.2 Alternative Definition of Dead Code Detection on SP_2

Before the extension, we give the alternative definitions of the functions $use_1\ g$, $use_2\ g$, $addLive\ g\ vs_1\ vs_2$ for SP_2 in Section 2). Recall that the original definition is as follows (taking account into the modification of S instead of S_2):

$$\begin{array}{l} use_1\ (e_2(1,2),\ (l_1,l_2)) = usev\ l_2 \\ use_1\ (e_2(2,1),\ \_) = \phi \\ use_1\ (\mathbf{2},\ \_) = \phi \\ use_1\ (S_2\ x\ y,\ \_) = use_1\ y \cup (use_2\ x \setminus def_{1\to 2}\ y) \\ use_1\ (P_2\ x\ y,\ \_) = use_1\ x \cup (use_2\ y \setminus def_{1\to 2}\ x) \cup \\ \qquad\qquad use_1\ y \cup (use_2\ x \setminus def_{1\to 2}\ y) \end{array}$$

$$\begin{array}{l} def_{1\to 2}\ (e_2(1,2),\ (l_1,l_2)) = defv\ l_2 \\ def_{1\to 2}\ (e_2(2,1),\ \_) = Var \\ def_{1\to 2}\ (\mathbf{2},\ \_) = Var \\ def_{1\to 2}\ (S_2\ x\ y,\ \_) = def_{1\to 2}\ y \cup def_{2\to 1}\ x \\ def_{1\to 2}\ (P_2\ x\ y,\ \_) = def_{1\to 2}\ x \cap def_{1\to 2}\ y \end{array}$$

$$\begin{array}{l} addLive\ (e_2(1,2),\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = (e_2(1,2),\ (l_1,\ l_2,\ vs_1 \cup usev\ l_2 \cup (vs_2 \setminus defv\ l_2),\ vs_2)) \\ addLive\ (e_2(2,1),\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = (e_2(2,1),\ (l_1,\ l_2,\ vs_1,\ vs_2 \cup usev\ l_1 \cup (vs_1 \setminus defv\ l_1))) \\ addLive\ (\mathbf{2},\ (l_1,l_2))\ vs_1\ vs_2 = (\mathbf{2},\ (l_1,l_2,\ vs_1,vs_2)) \\ addLive\ (S_2\ x\ y,\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = (S_2\ (addLive\ y\ vs_1\ (use_2\ x \cup (vs_2 \setminus def_{2\to 1} x))) \\ \qquad\quad (addLive\ x\ vs_2\ (use_2\ y \cup (vs_1 \setminus def_{2\to 1} y))), \\ \qquad (l_1,\ l_2,\ vs_1,\ vs_2)) \\ addLive\ (P_2\ x\ y,\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = (P_2\ (addLive\ x \\ \qquad\quad (vs_1 \cup use_1\ y \cup ((vs_2 \cup use_2\ x) \setminus def_{1\to 2}\ y)) \\ \qquad\quad (vs_2 \cup use_2\ y \cup ((vs_1 \cup use_1\ x) \setminus def_{2\to 1}\ y))) \\ \qquad\quad (addLive\ y \\ \qquad\quad (vs_1 \cup use_1\ x \cup ((vs_2 \cup use_2\ y) \setminus def_{1\to 2}\ x)) \\ \qquad\quad (vs_2 \cup use_2\ x \cup ((vs_1 \cup use_1\ y) \setminus def_{2\to 1}\ x))), \\ \qquad (l_1,\ l_2,\ vs_1,\ vs_2)) \end{array}$$

The alternative definition below contains some redundant computation. This is because generality of the definition, if one considers to extend to general k. Note that distributivity of $\cap$ wrt $\cup$ and inclusion like $use_2\ g \setminus def_{1\to 2}\ g \subseteq use_1\ g$ can absorb the differences.

$$\begin{array}{l} use_1\ (e_2(1,2),\ (l_1,l_2)) = usev\ l_2 \\ use_1\ (e_2(2,1),\ (l_1,l_2)) = \phi \\ use_1\ (\mathbf{2},\ (l_1,l_2)) = \phi \\ use_1\ g@(S_2\ x\ y,\ (l_1,l_2)) \\ \quad = use_1\ x \cup (use_1\ y \setminus def_{1\to 2}\ g) \cup ((use_2\ x \cup use_2\ y) \setminus def_{1\to 2}\ x) \\ use_1\ g@(P_2\ x\ y,\ (l_1,l_2)) \\ \quad = (use_1\ x \cup use_1\ y) \cup ((use_2\ x \cup use_2\ y) \setminus def_{1\to 2}\ g) \end{array}$$

$$\begin{array}{l} addLive\ (e_2(1,2),\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = (e_2(1,2),\ (l_1,\ l_2,\ vs_1 \cup usev\ l_2 \cup (vs_2 \setminus defv\ l_2),\ vs_2)) \\ addLive\ (e_2(2,1),\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = (e_2(2,1),\ (l_1,\ l_2,\ vs_1,\ vs_2 \cup usev\ l_1 \cup (vs_1 \setminus defv\ l_1))) \\ addLive\ (\mathbf{2},\ (l_1,l_2))\ vs_1\ vs_2 = (\mathbf{2},\ (l_1,\ l_2,\ vs_1,vs_2)) \\ addLive\ g@(S_2\ x\ y,\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = (S_2\ (addLive\ y\ vs_1\ use_2\ x \cup ((vs_2 \cup use_1\ x) \setminus def_{2\to 1} x)) \\ \qquad\quad (addLive\ x\ vs_2\ use_2\ y \cup ((vs_1 \cup use_1\ y) \setminus def_{2\to 1} y)), \\ \qquad (l_1,\ l_2,\ vs_1,vs_2)) \\ addLive\ g@(P_2\ x\ y,\ (l_1,l_2))\ vs_1\ vs_2 \\ \quad = let\ vs_1' = vs_1 \cup ((vs_2 \cup use_2\ x \cup use_2\ y) \setminus def_{1\to 2}\ g) \\ \qquad\quad vs_2' = vs_2 \cup ((vs_1 \cup use_1\ x \cup use_1\ y) \setminus def_{2\to 1}\ g) \\ \qquad in\ (P_2\ (addLive\ x\ (use_1\ y\ \cup vs_1')\ (use_2\ y\ \cup vs_2')) \\ \qquad\qquad\quad (addLive\ y\ (use_1\ x\ \cup vs_1')\ (use_2\ x\ \cup vs_2')), \\ \qquad\quad (l_1,\ l_2,\ vs_1,\ vs_2)) \end{array}$$

4.3 Dead Code Detection for Larger Tree Width

Used/Defined Variables in a Fragment in SP_3

$$use_1\ (e_3(i,j),\ (l_1,l_2,l_3)) = \begin{cases} usev\ l_j & \text{if } i=1 \\ \phi & \text{otherwise} \end{cases}$$

$$\begin{array}{l} use_1\ (\mathbf{3},\ (l_1,l_2,l_3)) = \phi \\ use_1\ g@(S_3\ x\ y\ z,\ (l_1,l_2,l_3)) \\ \quad = use_1\ y \cup use_1\ z \cup \\ \qquad ((use_1\ x \cup use_2\ z) \setminus def_{1\to 2}\ g) \cup \\ \qquad ((use_2\ x \cup use_2\ y) \setminus def_{1\to 3}\ g) \cup \\ \qquad ((use_3\ x \cup use_3\ y \cup use_3\ z) \setminus def_{1\to \$}\ g) \\ use_1\ g@(P_3\ x\ y,\ (l_1,l_2,l_3)) \\ \quad = (use_1\ x \cup use_1\ y) \cup \\ \qquad ((use_2\ x \cup use_2\ y) \setminus def_{1\to 2}\ g) \cup \\ \qquad ((use_3\ x \cup use_3\ y) \setminus def_{1\to 3}\ g) \end{array}$$

The basic idea is the same as that in $use_1\ g$ for SP_2 except

for the modification $((use_3\ x \cup use_3\ y \cup use_3\ z) \setminus def_{1\to\$}\ g)$ in $use_1\ g@(S\ x\ y\ z,\ (l_1,l_2,l_3))$. \$ is the newly introduce symbol that represents the removed terminal in $S\ x\ y\ z$, i.e., terminal 3 in x,y,z. For SP_2, $def_{1\to\$}\ (S\ x\ y)$ coincides with $def_{1\to 2}\ x$, because paths from terminal 1 to terminal 2 in x are only paths from terminal 1 to \$ in g without loops. Thus, for SP_2, the need for \$ is hidden.

$$
\begin{array}{l}
def_{1\to 2}\ (e_3(i,j),\ (l_1,l_2,l_3)) \\
\quad = \begin{cases} def v\ l_2 & \text{if } i=1,\ j=2 \\ Var & \text{otherwise} \end{cases} \\
def_{1\to 2}\ (\mathbf{3},\ (l_1,l_2,l_3)) = Var \\
def_{1\to 2}\ (S_3\ x\ y\ z,\ (l_1,l_2,l_3)) \\
\quad = def_{1\to 2}\ z \cap \\
\qquad (def_{1\to 2}\ y \cup def_{2\to 1}\ x) \cap (def_{1\to 3}\ z \cup def_{3\to 1}\ x) \cap \\
\qquad (def_{1\to 3}\ y \cup def_{3\to 2}\ z) \cap (def_{1\to 3}\ y \cup def_{3\to 1}\ x) \cap \\
\qquad (def_{1\to 3}\ z \cup def_{3\to 2}\ y \cup def_{2\to 1}\ x) \cap \\
\qquad (def_{1\to 2}\ y \cup def_{2\to 3}\ x \cup def_{3\to 2}\ z) \\
def_{1\to 2}\ (P_3\ x\ y,\ (l_1,l_2,l_3)) \\
\quad = def_{1\to 2}\ x \cap def_{1\to 2}\ y \cap \\
\qquad (def_{1\to 3}\ y \cup def_{3\to 2}\ x) \cap (def_{1\to 3}\ x \cup def_{3\to 2}\ y) \\
def_{1\to\$}\ (S\ x\ y\ z,\ (l_1,l_2,l_3)) \\
\quad = def_{1\to 3}\ y \cap def_{1\to 3}\ z \cap \\
\qquad (def_{1\to 2}\ y \cup def_{2\to 3}\ x) \cap (def_{1\to 3}\ z \cup def_{3\to 3}\ x) \cap \\
\qquad (def_{1\to 2}\ y \cup def_{2\to 1}\ x \cup def_{2\to 3}\ z) \cap \\
\qquad (def_{1\to 2}\ z \cup def_{1\to 2}\ x \cup def_{2\to 3}\ y)
\end{array}
$$

The definition of $def_{1\to 2}$ in SP_3 is quite complex especially for $S\ x\ y\ z$. The intuition can be obtained by replacing $\cup,\cap$ with $\wedge,\vee$, respectively. Then, by setting values for base cases as

$$
\begin{array}{l}
reach_{1\to 2}\ (e_3(i,j),\ (l_1,l_2,l_3)) = \begin{cases} True & \text{if } (i,j)=(1,2) \\ False & \text{otherwise} \end{cases} \\
reach_{1\to 2}\ (\mathbf{3},\ (l_1,l_2,l_3)) = False
\end{array}
$$

the same definition gives us the judgment of reachability from terminal 1 to terminal 2 in g.

Live Variables Detection for SP_3

Now, we give definition of *addLive* to detect live variables for SP_3.

$$
\begin{array}{l}
addLive\ g@(e_3(1,2),\ (l_1,l_2,l_3))\ vs_1\ vs_2\ vs_3 \\
\quad = (e_3(1,2), \\
\qquad (l_1,\ l_2,\ l_3,\ vs_1 \cup usev\ l_2 \cup (vs_2 \setminus def v\ l_2),\ vs_2,\ vs_3)) \\
addLive\ (\mathbf{3},\ (l_1,l_2,l_3))\ vs_1\ vs_2\ vs_3 \\
\quad = (\mathbf{3},\ (l_1,\ l_2,\ l_3,\ vs_1,\ vs_2,\ vs_3)) \\
addLive\ g@(S_3\ x\ y\ z,\ (l_1,l_2,l_3))\ vs_1\ vs_2\ vs_3 \\
\quad = let\ vs_1' = (vs_1 \cup use_1\ y \cup use_1\ z) \cup \\
\qquad\qquad ((vs_2 \cup use_1\ x \cup use_2\ z) \setminus def_{1\to 2}\ g) \cup \\
\qquad\qquad ((vs_3 \cup use_2\ x \cup use_2\ y) \setminus def_{1\to 3}\ g) \cup \\
\qquad\qquad ((use_3\ x \cup use_3\ y \cup use_3\ z) \setminus def_{1\to\$}\ g) \\
\qquad vs_2' = ((vs_1 \cup use_1\ y \cup use_1\ z) \setminus def_{2\to 1}\ g) \cup \\
\qquad\qquad (vs_2 \cup use_1\ x \cup use_2\ z) \cup \\
\qquad\qquad ((vs_3 \cup use_2\ x \cup use_2\ y) \setminus def_{2\to 3}\ g) \cup \\
\qquad\qquad ((use_3\ x \cup use_3\ y \cup use_3\ z) \setminus def_{2\to\$}\ g) \\
\qquad vs_3' = ((vs_1 \cup use_1\ y \cup use_1\ z) \setminus def_{3\to 1}\ g) \cup \\
\qquad\qquad ((vs_2 \cup use_1\ x \cup use_2\ z) \setminus def_{3\to 2}\ g) \cup \\
\qquad\qquad (vs_3 \cup use_2\ x \cup use_2\ y) \cup \\
\qquad\qquad ((use_3\ x \cup use_3\ y \cup use_3\ z) \setminus def_{3\to\$}\ g) \\
\qquad vs' = ((vs_1 \cup use_1\ y \cup use_1\ z) \setminus def_{\$\to 1}\ g) \cup \\
\qquad\qquad ((vs_2 \cup use_1\ x \cup use_2\ z) \setminus def_{\$\to 2}\ g) \cup \\
\qquad\qquad ((vs_3 \cup use_2\ x \cup use_2\ y) \setminus def_{\$\to 3}\ g) \cup \\
\qquad\qquad ((use_3\ x \cup use_3\ y \cup use_3\ z) \\
\quad in\ (S\ (addLive\ x\ vs_2'\ vs_3'\ vs')\ (addLive\ y\ vs_1'\ vs_3'\ vs') \\
\qquad (addLive\ z\ vs_1'\ vs_2'\ vs'), \\
\qquad (l_1,\ l_2,\ l_3,\ vs_1,\ vs_2,\ vs_3))
\end{array}
$$

$$
\begin{array}{l}
addLive\ g@(P_3\ x\ y,\ (l_1,l_2,l_3))\ vs_1\ vs_2\ vs_3 \\
\quad = let\ vs_1' = (vs_1 \cup use_1\ x \cup use_1\ y) \cup \\
\qquad\qquad ((vs_2 \cup use_2\ x \cup use_2\ y) \setminus def_{1\to 2}\ g) \cup \\
\qquad\qquad ((vs_3 \cup use_3\ x \cup use_3\ y) \setminus def_{1\to 3}\ g) \\
\qquad vs_2' = ((vs_1 \cup use_1\ x \cup use_1\ y) \setminus def_{2\to 1}\ g) \cup \\
\qquad\qquad (vs_2 \cup use_2\ x \cup use_2\ y) \cup \\
\qquad\qquad ((vs_3 \cup use_3\ x \cup use_3\ y) \setminus def_{2\to 3}\ g) \\
\qquad vs_3' = ((vs_1 \cup use_1\ x \cup use_1\ y) \setminus def_{3\to 1}\ g) \cup \\
\qquad\qquad ((vs_2 \cup use_2\ x \cup use_2\ y) \setminus def_{3\to 2}\ g) \cup \\
\qquad\qquad (vs_3 \cup use_3\ x \cup use_3\ y) \\
\quad in\ (P\ (addLive\ x\ vs_1'\ vs_2'\ vs_3')\ (addLive\ y\ vs_1'\ vs_2'\ vs_3'), \\
\qquad (l_1,\ l_2,\ l_3,\ vs_1,\ vs_2,\ vs_3))
\end{array}
$$

Discussion

Here, we present our study only for SP_3, i.e., tree width at most 3. It is worth mentioning the analogy between $def_{i\to j}$ and $reach_{i\to j}$, and the difficulty to extend to graphs with larger tree width is focused on the reachability description among terminals. Current our description is *not* parametric wrt tree width, i.e., we must describe, say, dead code detection for each SP_k. However, we have a basic feel that there would be some generic *skeleton*-like structure regarding reachability. That is, with the description of reachability for SP_k and the description of an analysis for SP_2, we can generate the description of an analysis for general SP_k.

Of course, with the increase of tree width, the number of functions rapidly grows. But, recall that most JAVA programs (and possibly other imperative programs) have a control flow graph with tree width at most 3 [16]. Thus, even for relatively small tree width, our method would cover quite large portion of real programs.

5 Related Work

Many researches have been devoted to the *declarative approaches* to program analyses. Steffen and Schmidt [31, 30] showed that temporal logic is well suited to *describe* data dependencies and other properties exploited in classical compiler optimization. Lacey, et.al. [21] formalized program optimization as rewriting systems with temporal logic side conditions (described in CTL-FV) and shows that CTL-FV plays a crucial role in the *proofs* of correctness of classical optimizations. Instead of temporal logic, de Moor, et.al. [12] proposed another functional approach to control flow analyses. Their specification language is the regular path condition, but the efficiency of derived programs is not discusses.

There are several functional approaches for computation on graphs. For instance, Fegaras and Sheard [14] treat graphs with embedded functions, i.e., graphs are treated as functions that generates all paths in a graph. Erwig introduces the *active pattern matching*, which is a conditional pattern matching mechanism [13]. Their approaches are interesting in description, but the existence of strong side conditions limits the chance to optimize. Instead, we restrict ourselves to graphs with bounded tree width, in which many NP-hard graph problems are solved in linear-time [11, 9].

The concept of a graph with bounded tree width [25] independently appeared from early 80's; partial k-tree in terms of cliques, some algebraic construction of k-terminal graphs [4, 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. For graphs with bounded tree width, there have been lots of work on automatic generation of linear-time algorithms from specification in monadic second order formulae, which are frequently NP-complete for general graphs [11, 9].

Our starting observation is that most programs (without spaghetti GOTO) have control flow graphs with bounded tree width, and

many control flow analyses can be specified in temporal logic (such as CTL-FV). By combining them, it seems easy to obtain (almost) linear-time algorithm for control flow analyses. This is true in theory, but not in practice; each existence of quantifiers in a formula causes the exponential explosion of the constant factor. Our approach is, directly write functional specification on the simple data structure, SP Term. This approach drastically reduces the constant factor [27]. Further, an SP Term is more approachable especially from programming point of view, and it does not refuse to capture better algorithmic ideas.

For an algebraic construction of graphs, one of the early work for flowchart scheme is found in [29]. Bauderon and Courcelle [4] are also pioneers, 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 by Arnborg, et.al. [3] requires the recursive constructors l^i_j, r_j, s_j, p_j with $1 \leq i \leq j \leq k$ for graphs with tree width at most k. Thus, the number of their recursive constructors becomes $k(k+1)(k+2)/6$, and this makes us difficult to write recursive definitions. We proposed another construction, SP Term, which has only 2 recursive constructors S_k, P_k regardless of the size of k. The number of constants $e_k(i, j)$ has square growth, but they are interpreted as diedges from the i-th to the j-th terminal. For these constants, writing functional specification (base cases) is easy; even in uniform way.

Thorup [33] showed that a structured imperative program have a control flow graph with relatively small tree width. He also investigated on finding *near* optimal register allocation by the conventional graph coloring on an intersection graph. It is well known that register allocation is equivalently reduced to the graph coloring problem [10], which is known to be NP-complete. For precise solution, it seems pessimistic; Kannan and Proesbsting showed that the number of minimal coloring (thus deciding the minimum number of registers that can be allocated without spilling) is NP-complete even for SP_2 (series parallel graphs) [19].

However, if we further assume that the number of registers is fixed, we can obtain an efficient solution. Bodlaender, et.al. showed a linear-time algorithm to decide whether a program can be executed without *spilling* for a fixed number of registers [8]. This is elegant in theory; however, their estimation includes the blow up of tree width of an intersection graph of a family of subgraphs. Thus, their constant factor explodes.

Our method based on Optimization Theorem could also have a huge constant factor, which can grow to the power of the number of live variables. However, there is possibility to tame it. For instance, we could expect the number of live variables at each program point are not so large, and most of markings would be avoided immediately. These observation suggest that, in practice, there seems a room to improve the constant factor drastically by demand-driven computation and other program transformation techniques, which are available for functional programs. For instance, Ohori proposed another register allocation by proof transformation on typed assembly language, which reduces the number of candidates of optimal register allocations [22]. The combination with such methods would be worth exploring.

We should mention another classical efficient solution under certain restriction of control flow graphs: *reducible flow graph* [2, 1]. If a program has a reducible control flow graph, one can construct $n\ (log\ n)$ algorithms for program analyses, such as common subexpression detection. Knuth also showed that most FORTRAN programs have reducible control flow graphs by an empirical study [20].

SP Term is independent to the concept of a reducible flow graph; for instance, Hecht and Ullman showed a graph is reducible if and only if the left-hand-side figure in Fig. 11 is contained [17]. However, that graph is easy to treat from tree width point of view; it is described in SP_2 as

$$S_2\ e_2(1,2)\ (P_2\ (S_2\ (P_2\ e_2(1,2)\ e_2(2,1))\ e_2(1,2))\ e_2(2,1))$$

(with terminal 1 at the top and terminal 2 at the rightmost node). In contrast, a complete directed acyclic graph (DAG) with m-nodes (as in the right-hand-side figure in Fig. 11) is described in SP_m, proportional to the size m. However, any DAG is reducible.

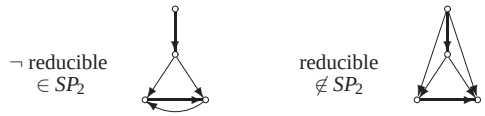

Figure 11. Difference between reducible flow graph and SP Term

6 Conclusion and Future Work

In this paper, we proposed an *iterative-free* approach to program analysis, based on the fact that control flow graphs of most practical programs are well structured. Our main contributions can be summarized as follows.

- We defined a simple but powerful algebraic construction of digraphs called SP Terms, on which program analyses can be naturally described as catamorphisms (or mutumorphism). As catamorphism enjoys many nice algebraic rules such as fusion and tupling for algorithmic optimization [6], this catamorphic formalization of program analyses makes it possible to systematically derive efficient analysis algorithms, which has not been really recognized so far.
- We identified that many program analyses can be considered as the maximum marking problems. By making use of the optimization theorem for them, we are able to obtain efficient analysis algorithms.
- As demonstrated by two examples, our method is quite powerful. In fact, many program analysis examples in the compiler textbook can be cast into this framework.

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

- As pointed in Section 3, the table for dynamic programming technique can easily explode. Although our method drastically improves constant factor compared to starting from formulae [27], still this is quite true. However, we only need the computation that can reach to the result satisfying given constraints, and, from our experience, computation in most part of the table does not contribute to obtain such results. Therefore, we hope demand-driven computation will improve the situation, and would like to confirm it by experiments.
- Currently, our description of analyses is *not* parametric wrt tree width k. However, as Section 4 suggests, the reachability description would work as a generic skeleton-like structure. We have a strong feel about it, but it must be more concrete.
- 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. For instance, in Fig. 5, different occurrences in the SP Term of a node in the control flow graph have the same set of live variables. This is guaranteed by user's semantic consideration. From its own theoretical interest and possible better support, we hope to give

the complete axiomatization of SP Terms under this interpretation.

Acknowledgments

The authors thank Oege de Moor and Jeremy Gibbons for stimulating discussions during their visit at the University of Tokyo, and thank Aki Takano for his helpful suggestions. We also thank anonymous referees and Fritz Henglein for their valuable comments and suggestions. Last but not least, we thank Masato Takeichi for his continuous support.

7 References

[1] A. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] F.E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[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. Maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, 2001.

[6] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1996.

[7] H.L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal Computing*, 25(6):1305–1317, 1996.

[8] H.L. Bodlaender, J. Gustedt, and J.A. Telle. Linear-time register allocation for a fixed number of registers. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms, SODA 1998*, pages 574–583. ACM Press, 1998.

[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] G.J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. ACM Symposium on Compiler Construction*, pages 98–105. ACM Press, 1982.

[11] B. Courcelle. 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.

[12] O. de Moor, D. Lacey, and E. van Wyk. Universal regular path queries. to appear in High Order Symbolic Computation, 2002.

[13] M. Erwig. Functional programming with graphs. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 52–65. ACM Press, 1997. SIGPLAN Notices 32(8).

[14] 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*, pages 284–294. ACM Press, 1996.

[15] M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.

[16] 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.

[17] M.S. Hecht and J.D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[18] 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.

[19] S. Kannan and T. Proebsting. Register allocation in structured programs. *Journal of Algorithms*, 29:223–237, 1998.

[20] D.E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1(2):105–134, 1971.

[21] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 283–294. ACM Press, 2002.

[22] A. Ohori. Register allocation by program transformation. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP03*, pages 399–413, 2003. Lecture Notes in Computer Science, Vol. 2618, Springer-Verlag.

[23] L. Perković and B. Reed. An improved algorithm for finding tree decompositions of small width. In Widmayer et al., editor, *WG'99*, pages 148–154, 1999. Lecture Notes in Computer Science, Vol. 1665, Springer-Verlag.

[24] 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.

[25] N. Robertson and P.D. Seymour. Graph minors II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[26] I. Sasano, Z. Hu, and M. Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In *Proc. 2nd International Workshop in Semantics, Applications, and Implementation of Program Generation*, volume 2196, pages 72–91. Springer-Verlag, 2001. Lecture Notes in Computer Science.

[27] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear-time algorithms for solving maximum-weightsum problems. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, pages 137–149. ACM Press, 2000.

[28] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Derivation of linear algorithm for mining optimized gain association rules. *Computer Software*, 19(4):39–44, 2002.

[29] H. Schmeck. Algebraic characterization of reducible flowcharts. *Journal of Computer System Science*, 27(2):165–199, 1983.

[30] D.A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 38–48. ACM Press, 1998.

[31] B. Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 346–364. Springer-Verlag,

1991.

[32] K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the Association for Computing Machinery*, 29:623–641, 1982.

[33] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142:159–181, 1998.

A Computing SP Terms directly from Imperative Programs

To show the direct translation from an imperative program with GOTO to an SP Term, we define a simple imperative language. The definition, given in Figure 12, is similar to that in [21], except for the additional "while" construct. For simplicity, this language has no exceptions or procedures.

P	::=	I; P	program
I	::=	$l : C$	instruction
C	::=	$x := e$	assignment
	|	*input* x	input statement
	|	*output* x	output statement
	|	**if** e **then** P **else** P **fi**	conditional statement
	|	**while** e **do** P **od**	while loop
	|	*goto* l	goto statement
	|	*break*	break statement
l	:	label	

Figure 12. A Simple Imperative Language

Let us consider a program in this language with at most n-GOTO. The translation *transG* to an SP Term is given below. The basic idea is; construct an SP Term by ignoring GOTO and memorize their source nodes as additional terminals. Then, scan the SP Term again, and add an edge by the parallel composition at some subterm in which the destination node eventually becomes a terminal. (Note that each node 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. 12. As in Section 2.1, 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 nodes 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 2.1, and

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

where functions $addG$, $nlift$, and $trans$ are defined below.

$$
\begin{array}{l}
nlift\ ::\ SP_2 \to SP_{n+2} \\
nlift\ (e^+,\ (l_1,l_2)) \ =\ (e_{n+2}(1,n+2),\ (l_1,so_1,\cdots,so_n,l_2)) \\
nlift\ (e^-,\ (l_1,l_2)) \ =\ (e_{n+2}(n+2,1),\ (l_1,so_1,\cdots,so_n,l_2)) \\
nlift\ (\mathbf{2},\ (l_1,l_2)) \ =\ (\mathbf{2},\ (l_1,so_1,\cdots,so_n,l_2)) \\
nlift\ (S_2\ x\ y,\ (l_1,l_2)) \\
\ =\ (S_{n+2}\ (permT\ (nlift\ x)) \\
\qquad\qquad (\mathbf{n+2},\ (l_1,so_2,so_3,\cdots,so_n,l_2,\$)) \\
\qquad\qquad (\mathbf{n+2},\ (l_1,so_1,so_3,\cdots,so_n,l_2,\$)) \\
\qquad\qquad \cdots \\
\qquad\qquad (\mathbf{n+2},\ (l_1,so_1,so_2,\cdots,so_{n-1},l_2,\$)) \\
\qquad\qquad (nlift\ y), \\
\qquad (l_1,so_1,\cdots,so_n,l_2)) \\
nlift\ (P_2\ x\ y,\ (l_1,l_2) \\
\ =\ (P_{n+2}\ (nlift\ x)\ (nlift\ y),\ (l_1,so_1,\cdots,so_n,l_2))
\end{array}
$$

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

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

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

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

The function *nlift* insert labels of goto statements 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 nodes of each goto-statement.

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}.