자동화된 프로그램 시험을 위한 입력 자료구조의 모양 식별

(Identifying a Shape of Input Data Structure for Automated Program Testing)

정 인 상 †

(Insang Chung)

요 약 프로그램 시험 비용은 테스트 데이타를 생성하는 과정을 자동화함으로써 상당히 줄일 수 있다. 테스트 데이타 생성은 보통 선택된 프로그램 경로를 실행하는 입력 값들을 식별하는 데 주안점을 둔다. 지금까지 많은 연구가 있어왔지만 여전히 해결해야할 문제가 있다. 그러한 문제들 중에 모양 문제가 있다. 모양 문제는 주어진 프로그램 경로를 수행하기위해 요구되는 입력 자료구조를 밝혀내는 문제이다. 이 논문에서 이 모양 문제에 대한 새로운 방법을 제시한다. 이 방법은 주어진 경로를 포인터 역참조가 없는 정적 단일 할당문 (Static Single Assignment, SSA) 형태로 변환한다. 이는 주어진 경로 상에 존재하는 각 프로그램 문장을 등식이나 부등식과 같은 제약식으로 간주할 수 있게 해준다. 이러한 제약식에 대한 해는 각 입력 변수에 대한 "points-to relation" 형태로 나타난다. 간단한 예들을 통하여 제안한 방법에 대해 설명한다.

키워드 : 프로그램 테스팅, 자동 테스트 데이타 생성, 모양 분석

Abstract We can significantly reduce the cost of program testing by automating the process of test data generation. Test data generation usually concerns identifying input values on which a selected path is executed. Although lots of research has been done so far, there still remains a lot of issues to be addressed. One of the issues is the shape problem. The shape problem refers to the problem of figuring out a shape of the input data structure required to cause the traversal of a given path. In this paper, we introduce a new method for the shape problem. The method converts the selected path into static single assignment (SSA) form without pointer dereferences. This allows us to consider each statement in the selected path as a constraint involving equality or inequality. We solve the constraints to get a solution which will be represented in terms of the points-to relations for each input variable. Simple, but illustrative examples are given to explain the proposed method.

Key words : Program Testing, Automated Program Test Data Generation. Shape Analysis

1. Introduction

Software testing is an essential step for improving software quality, but suffers from large amount of time and computing resource. The cost of software testing can be reduced significantly by automating the process of test data generation. Test data generation problem can be stated as a search problem which finds input values to exercise a selected path. Through the years, several methods have been proposed to attack the problem [1-11]. However, it is true that there still remain a lot of issues that have to be addressed to put automatic test data generation in practice.

One of the issues that we concern in this paper is the "shape problem": the problem of finding a shape of the input data structure to cause the traversal of a selected path. The shape problem arises when testing is conducted for programs with pointers. Most work to date has focused on search algorithms that come up with solutions to traverse

· This research was financially supported by Hansung University in the year of 2004.

† 종신회원 : 한성대학교 컴퓨터 공학부 교수
insang@hansung.ac.kr

논문접수 : 2003년 8월 29일
심사완료 : 2004년 7월 28일

the selected path in the absence of pointers.

Up to our knowledge, the work performed by Korel is unique in dealing with the shape problem [7]. This method requires actual execution of a program so that the program flow can be monitored to determine whether the intended path was taken. Because the exact locations that pointer variables point to known during program execution, it is possible to identify a shape of the input data structure required to traverse a given path. However, many iterations may be required if the program execution flows in the wrong direction before a suitable input (pointer) value is found.

This paper introduces a new method to help identify automatically test data on which a selected path in the program is executed in the presence of pointers. The proposed method statically analyzes the selected path. If the analysis is completed successfully, then we yield:

- a shape of the input data structure and
- a set of constraints describing how to assign values for input variables which are not of pointer type in order to cause the traversal of the selected path.

A shape of the input data structure is described in terms of points-to relations (i.e. what pointer variables are pointing to) [12] of each input pointer variable. The key point of the proposed method is to introduce a new points-to relation whenever necessary. For example, suppose the situation where we encounter a statement "x=*y" for an input pointer variable y. If y does not point to any storage location up to that point, it would be necessary to create a storage location pointed to by y in order to execute that statement without any violation. This can be viewed as a search process of figuring out a shape of the input data structure.

The proposed method also generates a constraint system from the selected path. To the end, each statement is converted into Static Single Assignment(SSA) form [13] involving no pointer dereferences. One important feature of SSA form is that each variable has at most one definition (meaning that it is assigned at most once). This feature allows us to deal with each program variable as a logical variable [3].

The main contribution of this paper is a static approach to automatic program testing for programs in the presence of pointers. The approach does not require any means for controlling execution of the target program unlike dynamic approaches which require the actual execution of the program [2,4,7,10]. In general, dynamic approaches formulate the test data generation problem as a function minimization problem by treating each branch predicate on the given path as a function that becomes minimal when the desired outcome is produced. This implies that they should be equipped with certain mechanism which can monitor the program's execution and force execution toward the desired direction. According to the types of the programming languages used, different control mechanisms need to be developed.

Another aspect of our approach is separation of test data generation for non-pointer types and the shape analysis problem. Such separation of concern enables us to take advantages of current test data generation techniques for non-pointer variables which have been well studied relatively. Since, in particular, the proposed method produces a set of constraints for non-pointer types in terms of equalities or inequalities between variables, conventional constraint solving techniques can be employed, leading to the reduction of the needed development effort.

The rest of the paper is organized as follows. In Section 2, we will explain in detail SSA form, basic terminologies, and definitions which will be used in the subsequent sections. In Section 3, we define transfer functions associated with various types of statements dealing with statically allocated memory objects and the dereference operator '*'. We also illustrate our method through an example. In Section 4, we extend our method to heap directed pointers which point to objects dynamically allocated in the heap. Finally, conclusion and future work will be given.

2. Preliminaries

One straightforward way to test data generation is to extract a number of constraints (equalities or inequalities) from a path under consideration and

solve the constraint system. This can be done by transforming the path into SSA form.

A key property of SSA form is that each variable has a unique static definition point [13]. In order to ensure this property, variable renaming is usually done as follows:

- every assignment to a variable v generates a new SSA variable v_i where i is a unique number,
- just after the assignment to v, v_i becomes the current name (the last version or the current instance) of v, and
- every subsequent use of v is replaced by its current name v_i.

We assume that the subscript number of each SSA variable starts with 0. For example, the sequence of code "x=10;x=x+3;" is converted into SSA form as follows: "x_1=10; x_2=x_1+3". In this example, we have two SSA variables x_1 and x_2 which can be treated as logical variables rather than program variables. Consequently, we can regard the statements as the constraints. That is to say, the first statement can be treated as the equality to assert that x_1 is equal to 10. Analogously, we can treat the second statement as the equality to assert that the value of x_2 is equal to the result of adding 3 to the value of x_1.

However, the presence of pointers complicates conversion of the selected path into SSA form because aliases can occur (i.e., two or more names exist for the same memory location) and a variable can be defined indirectly via a pointer dereference. This makes it necessary to exploit points-to information at each program point in the selected path during conversion to SSA form [14].

At each program point, we collect the points-to information and then replace each pointer dereference with its points-to result. For example, the sequence of assignments given by "x=&a;*x=10;y=a" can be converted to the SSA form without the pointer dereference "x_1=&a_0;a_1=10;y_1=a_1" by using the points-to information that x points to a after executing the first assignment.

However, since the variable a does not appear textually on the left-hand sides of the assignments, the naive conversion to SSA form would lead to the incorrect inference that the reference of a of the last assignment is a use of a_0. In fact, it is a use of a_1 because it is defined at the second assignment indirectly. Thus, it is necessary to have information about what pointer variables are pointing to in order to convert the selected path into SSA form correctly.

We represent points-to relations for each program point with σ mapping variables to memory locations:

$$\sigma \in \textbf{State} = \textbf{Var} \rightarrow \textbf{Loc}$$

Var is the (finite) set of variables occurring in the SSA form of the program path of interest. **Loc** is a set of locations (addresses) partially ordered as depicted in Fig. 1.

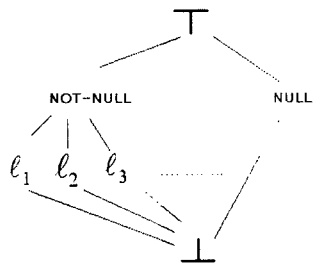

Fig. 1 The structure of locations

Then, σ(x) will now either be

- ⊤ meaning that x may possibly point to any location (x can be NULL),
- ⊥ meaning that x is not a pointer variable or its points-to relation is undefined,
- **NOT-NULL** meaning that x points to certain memory location, but its exact address is not yet known,
- l_i meaning that x points to a memory location whose address is l_i, or
- **NULL** meaning that x is not currently pointing to any location at all.

States are assumed to be ordered as follows:

$$\sigma_i \sqsubseteq \sigma_j \text{ if for all x, } \sigma_i(x) \sqsubseteq \sigma_j(x)$$

We also introduce $\perp_\sigma$ such that for all $\sigma \in$State, $\perp_\sigma \sqsubseteq \sigma$. We will use $\perp_\sigma$ to denote that a selected path is infeasible (i.e, we can not find input values on which the selected path is executed).

When we refer to a location, it is often convenient to use a symbolic name instead of its address. In this paper, we assume that the targets

of pointers always possess a (symbolic) name. Under this assumption, the fact that variable x points to a location named y can be represented by $\sigma(x)=y$ without any confusion.

However, this assumption does not hold for the variables that are not in the scope of a function but might be accessible through indirect reference. For example, consider function "fun" with the formal parameter x of type int**:

fun(int** x) { ... }

The problem lies in the fact that the function "fun" can refer to memory locations through '*x' or '**x' which are not in the scope of "fun". In order to capture points-to information accurately, we need to name such locations. To the end, we make use of the concept of *Invisible Variables* [12]. Invisible variables are names used for the variables that are not in the scope of a function but accessible through indirect reference. For example, the invisible variable x with type int** are 1_x with type int* and 2_x with type int, respectively.

We also define the total function $last_\sigma$ which gives a last version of a variable with respect to σ. For example, suppose that σ is the state after executing the sequence of the assignments "x=10; y=x+1; x=y". Then, $last_\sigma(x)$ will give x_2. $last_\sigma$ can also accept the SSA variable as input instead of the original variable. Thus, $last_\sigma(x)$, $last_\sigma(x_1)$, and $last_\sigma(x_n)$ $(n\geq 2)$ will get the same result x_2. On the other hand, let σ be the state immediately after executing the first assignment. Then, $last_\sigma(x)$ will give x_1 because we assume that the SSA number starts with 0 and x is defined at the first statement.

The pointer variables are partitioned into disjoint collections. A collection is a set of pointer variables which *should* point to the same memory location. On the contrary, two pointer variables belonging to distinct collections can not designate the same location. We assume that each pointer variable initially belongs to a distinct collection. We also assume that pointer variables point to different memory locations unless it can be shown that they must point to the same memory location.

We shall provide a function $[-]_\sigma$ which takes a variable as input and offers a collection which the last version of the variable with respect to σ belongs to. It is immediate that the following properties hold:

P1 $\sigma(last_\sigma(x))=\sigma(y)$ if $y\in[x]_\sigma$

P2 $last_\sigma(y)\in[x]_\sigma$ if and only if $last_\sigma(x)\in[y]_\sigma$

P3 $[x]_\sigma=\{last_\sigma(x)\}$ if $\sigma(last_\sigma(x))=NULL$

The property P1 states that all pointer variables belonging to the same collection should have the same value. For example, if x points to a certain memory location and y belongs to the same collection as x belongs to, then y also should point to the same memory location as x is pointing to. This leads to an alias pair (*x, *y) which represents that *x and *y refer to the same memory location.

The implication of the property P2 is straightforward. For all pointer variables belonging to the same collection, the collections provided by the function $[-]_\sigma$ should be identical. Finally, P3 states that if the pointer variable x points to no memory location at all, then the collection which x belongs to is a just singleton collection consisting of only the pointer variable x itself.

3. Shape Analysis for Program Testing

This section defines the transfer functions for boolean expressions and assignments used in the shape analysis. We then illustrate how the shape analysis algorithm works to give the solution which embodies as much information as possible about the shapes of the input data structure that can cause the traversal of the given path. In the rest of this section, we assume that a variable name means the last (SSA) version of the variable when there is no ambiguity.

3.1 Transfer functions for boolean expressions

Fig. 2. shows the transfer functions for boolean expressions involving pointers. For a given boolean expression and a given state, the main idea here is to derive the largest solution (state) from the given state which will evaluate the target boolean expression to true.

The first transfer function checks whether the pointer variable x equals **NULL**. If x is equal to NULL on the state σ, then the transfer function will leave σ unchanged because σ evaluates the

$\| x==\mathbf{NULL} \|\sigma$ $= \sigma \odot \{k \rightarrow \mathbf{NULL} | k \in [x]_\sigma\}$ if $\sigma(last_\sigma(x)) \sqcap \mathbf{NULL} \neq \bot$
$= \bot_\sigma$ otherwise

$\| x<>\mathbf{NULL} \|\sigma$ $= \sigma \odot \{(k, new\text{-}name(x, \sigma)) | k \in [x]_\sigma\}$ if $\sigma(last_\sigma(x)) = \top$
σ else if $last_\sigma(x)) = \mathbf{NOT\text{-}NULL}$ or $\sigma(last_\sigma(x)) = l$
$= \bot_\sigma$ otherwise

$\| x==l \|\sigma$ $= \sigma \odot \{(k, l) | k \in [x]_\sigma\}$ if $\sigma(last_\sigma(x)) \sqcap l \neq \bot$
$= \bot_\sigma$ otherwise

$\| x==y \|\sigma$ $= \sigma \odot \{(k, \sigma(last_\sigma(x)) \sqcap \sigma(last_\sigma(y))) | k \in [x]_\sigma\}$ if
$\sigma(last_\sigma(x)) \sqcap \sigma(last_\sigma(y)) \neq \bot$
$= \bot_\sigma$ otherwise

$\| x<>y \|\sigma$ $= \bot_\sigma$ if $last_\sigma(x) \in [y]_\sigma$ or $last_\sigma(y) \in [x]_\sigma$
$= \sigma$ otherwise

Fig. 2 The operational semantics of transfer functions for boolean expressions. The operator $\odot$ is the function overriding operator. The function $f \odot g$ is defined on the union of the domains f and g. On the domain of g it agrees with g, and elsewhere on its domain it agrees with f. l denotes the address of a certain location.

boolean expression to true.

An interesting case arises when the value of the pointer variable x is $\top$, i.e., $\sigma(last_\sigma(x)) = \top$. It implies that x can possibly point to any memory location including the situation where x designates no memory location at all. The present case requires the state that evaluates x to **NULL**. This necessitates to narrow the value of x (i.e., $\top$) on the current state σ into **NULL**. Consequently, a largest state which will evaluate the boolean expression to true can be derived from the current state σ as follows: $\sigma \odot \{last_\sigma(x) \rightarrow \mathbf{NULL}\}$.

We also have to consider the collection $[x]_\sigma$ because the property (P1) mentioned in the previous section stipulates that all pointer variables in the collection should have the same value. This is why we assign **NULL** to all the pointer variables belonging to $[x]_\sigma$.

We are now in a position to illustrate the transfer function associated with the boolean expression of the form "x<>**NULL**". What is interesting is when $\sigma(last_\sigma(x)) = \top$. This implies the points-to relation such that the pointer variable, say $\sigma(last_\sigma(x))$, should point to a certain memory location, but its exact address may be unknown at the moment. Then, we face a problem of how to represent such a points-to relation.

In order to cope with the problem, we materialize a concrete location from $\top$. The address of the materialized location is given NOT-NULL rather than a specific address. We also need to name the location. The naming is done by using the function "new-name(v, σ)" defined as follows:

$$new\text{-}name(v, \sigma) = \begin{cases} k+1\_last_\sigma(x) & \text{if } k\_last_\sigma(x) \in [v]\sigma \\ 1\_last_\sigma(v) & \text{otherwise} \end{cases}$$

The function "new-name(v, σ)" is based on the concept of invisible variables and associates a name with the location pointed to by $last_\sigma(v)$. It firstly checks whether a invisible variable is included in the collection (i.e., $[v]_\sigma$) the pointer variable, $last_\sigma(v)$, belongs to. If there exists a invisible variable of the form $k\_last_\sigma(p)$, then the anonymous location will be named $k+1\_last_\sigma(p)$. There may be a situation where more than one invisible variable exists in $[v]_\sigma$. In this case, it does not matter which one is used to name the anonymous location. If no invisible variables exist in $[v]_\sigma$, then we create an invisible variable $1\_last_\sigma(v)$ to name the anonymous location.

Once a name is associated with the newly created location, we introduce a new points-to relation by making the pointer variable y point to

new-name(y, σ). Note that the address of the materialized location is regarded as NOT-NULL to reflect that it can represent any location. This is very important when there exists another pointer variable, say y, which points to a concrete location named m and at some point in the given program path, m is shown to refer to the materialized location. That is, (*x, *y) forms an alias pair. Then, the exact address of the materialized location is reduced to the address of m. If we assign a specific address to the materialized location, it would not possible to detect such an alias pair because inconsistency occurs, i.e., $\sigma(x) \sqcap \sigma(y) = \perp_\sigma$.

The next transfer function is associated with the boolean expression of the form "x==*l*" where *l* designates a memory address. As a matter of fact, it makes no difference from the case for the boolean expression "x==**NULL**" except that x points to a certain location whose address is given *l*.

The transfer function associated with the boolean expression of the form "x==y" concerns that x and y can possibly point to the same memory location. Note that the the transfer function does not ignore the case where either x or y has the NULL value.

The boolean expression of the form "x<>y" checks whether or not two pointer variables belong to the same collection. If the current (SSA) instance of the pointer variable x (or y) belongs to the same collection as the current instance of the pointer variable y (or x) is belonging to, then we can conclude that they (should) refer to the same memory location and thus the boolean expression will evaluate to false. Otherwise, the current state will remain unchanged.

3.2 Transfer functions for assignments

The forms of assignments that we consider in this paper include "x=y", "x=*y", "*x=y", "x=&y", and "x=NULL". Complex statements can be treated in terms of the basic assignments. For example, the statement "*x=*y" are broken into the basic assignments as follows: "temp=*y;*x=temp".

The effect of the assignments that have in common is to generate new SSA variables since they define variables directly or indirectly. If the variable x is defined at an assignment, then the transfer function associated with the assignment

```
FUNCTION GP(x,σ:State) returns σ
1:    generate a new SSA variable, Nx for lastσ(x);
2:    set σ(Nx) to ⊤;
3:    set WorkList to {lastσ(x)};
4:    for each k in WorkList do
5:          delete k from WorkList;
6:          for each p pointing to k w.r.t  do σ
7:                generate a new SSA variable, Np, for p;
8:                set σ(Np) to Nk
9:                add p to WorkList;
10:         endfor
11:   endfor
```

Fig. 3 Function for generating new SSA variables. We assume that Nt denotes a newly created SSA variable for the variable lastσ(t) For example, if lastσ(t) is ti for i≥0, then N_t denotes t_{i+1}.

make use of the function "GP(x,σ)" which handles the generation of a new SSA variable for the variable x with respect to the state σ and records the newly created SSA variable as the last version of x (line 1). Note that the newly created SSA variable N_x for x is initialized to ⊤ (line 2).

The function GP(x,σ) also generates new SSA variables for all pointer variables that point to the last version of the variable x, i.e., $last_\sigma(x)$, on the state σ (line 4 through line 11). For example, consider an assignment which defines the variable x under the situation that a pointer variable p is pointing to x. Even though p does not appear textually on the left-hand side of the assignment, this case can be viewed as an indirect definition of p. Thus we need to create a new SSA variable for the pointer variable p. This process is repeated until all pointers that can reach the storage location named x are taken into account.

Fig. 4. defines the transfer functions for the assignments. Note that the transfer functions are formulated in terms of those of the boolean expressions. Since the first three transfer functions can be illustrated similarly, we consider only the transfer function associated with the assignment of the form "x=y".

The first effect of the assignment is to create new versions of the variables defined at the assignment directly or indirectly. This can be done by using the function GP as stated earlier. The result is a new state σ′ where the transfer function

$\| x=\textbf{NULL} \|\sigma$	=	$\| x=\textbf{NULL} \| GP(x,\sigma)$
$\| x=\&a \|\sigma$	=	$\| x==l_k \| GP(x,\sigma)$
$\| x=y \|\sigma$	=	$\| x==y \| GP(x,\sigma)$
$\| x=*y \|\sigma$	=	$\| x==new\text{-}name(y,\sigma) \| GP(x,\sigma_y)$ if $\sigma(last_\sigma(y))=\top$
	=	$\| x==m_y \| GP(x,\sigma)$ else if $\sigma(last_\sigma(y))=$**NOT-NULL** or l_{my}
	=	$\perp_\sigma$ otherwise
$\| *x=y \|\sigma$	=	$\| new\text{-}name(x,\sigma)==y \| GP(x,\sigma_x)$ if $\sigma(last_\sigma(x))=\top$
	=	$\| m_x==y \| GP(m_x,\sigma)$ if $\sigma(last_\sigma(x))=$**NOT-NULL** or l_{mx}
	=	$\perp_\sigma$ otherwise

Fig. 4 The transfer fumctions for assignments. In the transfer functions, lk}$ denotes the address of k, m_p denotes the location pointed to by $last_\sigma(p)$, σ_p, is the state computed by $\sigma_p=\sigma\odot\{(k,new\text{-}name(p,\sigma)\mid k\in[p]_\sigma\}$.

associated with the boolean expression "x==y" is applied and makes the pointer variable x point to the same location as y is pointing to. Recall that (the current instance of) the pointer variable x is initialized to $\top$ before evaluating the equality test because it is newly created. As a consequence, x will get the value of y by taking a meet of the values of the two pointer variables x and y.

The transfer functions associated with the last two assignments play an important role in determining a shape of the input data structure required to traverse the path of interest. The primary effect of these types of assignments is to introduce new points-to relationships whenever necessary, then making concrete a shape of input data structure. We illustrate only the transfer function associated with the assignment of the form "x=*y" because the other can be easily understood.

The transfer function associated with the assignment of the form "x=*y" attempts to transform the assignment into the form without the pointer dereference operator. This can be done by using the points-to information for y. The first clause of the transfer function of the assignment takes care of the case where $\sigma(last_\sigma(y))=\top$. In this case, we materialize a location from $\top$ whose name is given by new-name(y,σ). Once a name is associated with the materialized location, we introduce a new points-to relation by making the pointer variable y point to new-name(y,σ). Of course, this change should be made for all pointer variables belonging to the same collection as y is belonging to. The next step is simply to evaluate the transfer function associated with the equality "x==new-name(y,σ)".

The second clause of the transfer function takes case of the case where y points to a concrete location. In this case, we simply replace the right-hand side of the assignment with the location y is pointing to. For example, if y points to a certain location, say v, then the right-hand side of the assignment will be replaced by v and then the transfer function associated with the boolean expression "x==v" will be evaluated.

The last clause concerns the case where y has the NULL value. Obviously, dereferencing y at the assignment causes a violation. Thus, the result will be $\perp_\sigma$, indicating that the path under consideration cannot be executed.

3.3 The shape analysis algorithm

Fig. 5 shows the shape analysis algorithm for identifying the information about the shapes of the

input data structure that can cause the traversal of the given path $<s_1,...,s_n>$. The view that we take in the algorithm is that a program path is a constraint system describing how input data structure (or input values) should be formed in order to traverse the path. The idea is to extract a number of constraints from the given path by transforming it into SSA form without pointer dereferences. For the sub-path $<s_1,..., s_i>(i \geq n)$, a solution to the constraint system will be a state σ_i after evaluating the sub-path. The state σ_i has information about the shapes of the input data structure required to traverse the sub-path in terms of points-to relations for each pointer variable. Since the constraint system does not necessarily have a unique solution, the largest solution is desired.

First of all, we need to construct an initial state σ_0. For every variable x, its initial (SSA) version of the variable, x_0, needs to be generated. Concerning the points-to relation, every SSA variable generated from input variables, i.e., formal parameters or global variables, is assumed to point to anything. That is to say, $\sigma(x_0)=\top$ if x is an input pointer variable. This is reasonable because memory locations pointed to by input variables should not be initially restricted. On the other hand, if it is a local variable or it is not of pointer type, its points-to relation is initially set to undefined.

Basically, the algorithm composes the transfer functions associated with the program points in the given path $p=<s_1,...,s_n>$. That is to say, the solution will be a state σ given by:

$$\sigma=\sigma_n=f_p(\sigma_0)=f_{sn} \circ ... \circ f_{s1} \circ id(\sigma_0)$$

where "id" is the identity function for the empty path and f_{si} is the transfer function associated with statement s_i. A transfer function associated with each statement specifies how the statement acts on the input state and changes the input state to a new state. Starting with an initial state σ_0, we can yield the final state σ which will be the largest solution if the constraint system generated from the given path is consistent. On the other hand, if the system is inconsistent (i.e., denoted by $\perp_\sigma$), then we conclude that the path is infeasible. In the case, we can not find an input data structure that will cause the path to be executed.

Lines 9 through 12 take care of the points-to

FUNCTION get-shape(p:path) returns σ

1: for every variable x, generate its initial SSA version x_0 of the variable x;
2: construct σ_0 such that

$$\delta_0(x_0) = \begin{cases} \top & \text{if x is an input pointer variable of pointer type} \\ \perp & \text{otherwise} \end{cases}$$

3: set i to 1;
4: for each s_i in p do
5: if (s_i is of the form x<>y) then $\sigma_i=\sigma_{i-1}$;
6: else $\sigma_i=||s_i||\sigma_{i-1}$;
7: if $\sigma_i<>\perp_\sigma$ then
8: transform s_i into SSA form $\underline{s_i}$ without pointer dereferences
9: if ($\underline{s_i}$ is of the form x==NULL) then for all k in $[x]_{\sigma i-1}$, $[k]_{\sigma i}=\{k\}$
10: if ($\underline{s_i}$ is of the form x==l and $\exists y:\sigma_i(last_{\sigma i}(y))=l$) then $[x]_{\sigma i}=[y]_{\sigma i}=[x]_{\sigma i-1} \cup [y]_{\sigma i-1}$;
11: else if ($\underline{s_i}$ is of the form x==y) then $[x]_{\sigma i}=[y]_{\sigma i}=[x]_{\sigma i-1} \cup [y]_{\sigma i-1}$;
12: else $[x]_{\sigma i}=[x]_{\sigma i-1}$ for every variable x;
13: else report that the path is inconsistent and exit;
14: increment i;
15: endfor
16: for each $\underline{st}$ of the form x<>y do
17: if ($last_\sigma(x) \in [y]_\sigma$ or $last_\sigma(y) \in [x]_\sigma$) then report that the path is inconsistent and exit;
18: endfor

Fig. 5 Function for computing shape information for the selected path $p=<s_1,...,s_n>$

information. It suffices to consider the forms as follows: "x==NULL","x==*l*", and "x==y". Note that even though the assignments can affect the points-to information, they are eventually transformed into the above forms after processing line 8.

Line 9 concerns the form "x==NULL". In this case, all the pointer variables in the collection $[x]_{\sigma_{i-1}}$ point to no memory location at all. Thus, each pointer pointer variable k in $[x]_{\sigma_{i-1}}$ is separated in such way that $[k]_{\sigma_i}$ gives {k}. The next form that affects the points-to information is "x==y". The condition at line 10 checks whether there already exists a pointer variable which point to the memory location whose address is *l*. If such a variable, say y, exists, then it is necessary to merge $[x]_{\sigma_{i-1}}$ and $[y]_{\sigma_{i-1}}$ to reflect the fact that after evaluating "x==*l*", the pointer variables belonging to $[x]_{\sigma_i}$ (or $[y]_{\sigma_i}$) should point to the memory location whose address is *l*. Line 11 concerns the form "x==y". An interesting case arises when x and y belong to disjoint collections, but they are not in conflicts (i.e., $\sigma_i(x) \sqcap \sigma_i(y) \neq \perp$). Then, we need to merge the collections to indicate that they should point to the same memory location from now on. The other forms does not affect the points-to information.

Note that evaluation of the boolean expression of the form "x<>y" is deferred until points-to information for the path is collected. The reason for the lazy evaluation is because pointer variables are assumed to point to distinct locations unless it can be shown that they point to the same location. Without having information about what pointer variables are pointing to, we could not accurately determine whether two pointer variables are pointing to.

The time complexity of the shape analysis algorithm is determined as follows. It is not difficult to see that σ_i is computed for each s_i, that is, |p| times where |p| is the number of the statements plus the expressions in the given path p. The time complexity of the function GP in Fig. 3 is proportional to the square of the number of the variables in the path 'p', $|v|^2$ because the iteration is traversed |v| in the worst case and in each iteration, all possible points-to relations have to be considered, which equals |v|. This entails that the worst case time complexity of the shape analysis algorithm is $O(|p||X||v|^2)$.

The space complexity is proportional to the points-to information which is computed at each program point. Since the points-to information is proportional to the number of variables, the space complexity of the shape analysis algorithm is O(|p||X||v|).

3.4 An example

Fig. 6 shows an example program to illustrate the proposed method. Suppose that we want to identify a shape of input data structure required to traverse the path <1,2,3,4,6,8,9,10,11>. For the sake of clarity, we will represent each state by a pictorial representation called a *shape graph*. In a shape graph, square nodes model concrete memory locations. Edges model pointer values. Suppose that $\sigma(x)$ gets y. Then, there exists a directed edge from the square node named x to the square node named y. But, we do not explicitly show $\perp$ in shape graphs. From now on, by σ_0 we mean an initial state and by σ_i we mean the state immediately after processing statement I.

```
void Example(int **x, int **y, int v) {
int *p, *q, *r, z;

1:      p = *x;
2:      q = *y;
3:      if (p==q) {
4,5:        if (p == NULL) *q = v;
6,7:        else if (q == NULL) *p = v;
            else {
8:              r = &z;
9:              *r = 10
10,11:          if (z == v) *q=v;
              }
        }
        else {
12,13:    *p = v; *q = v;
        }
}
```

Fig. 6 An example program

First of all, we start with an initial state σ_0 such that $\sigma_0(x_0)=\sigma_0(y_0)=\top, \sigma_0(p_0)=\sigma_0(q_0)=\sigma_0(r_0)=\sigma_0(v_0)=\sigma_0(z_0)=\perp$. Recall that initial versions of input variables of pointer type are initialized to $\top$ while local variables or variables not of pointer type are initialized

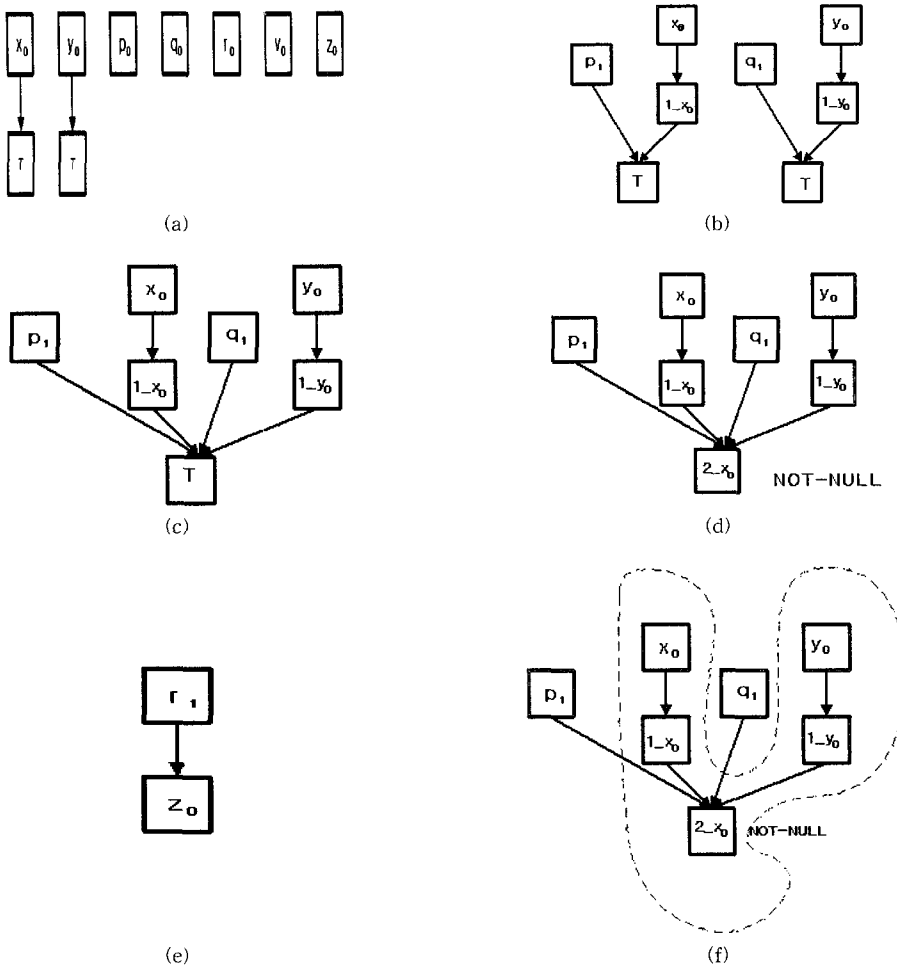

Fig. 7 The shape graphs that arise when the shape analysis algorithm is applied to the given path of the example program in Fig. 6; (a) depicts the initial state σ0,(b) depicts the shape graph after evaluating the sub-path <1,2>, (c) depicts the shape graph after evaluating the sub-path <1,2,3>, and (d) depicts the shape graph after evaluating the sub-path <1,2,3,4>. As a matter of fact, evaluation of the given path <1,2,3,4,6,8,9,10,11> does not affect the shape graph given in (d) any more. (e) shows the points-to relation arisen after evaluating the assignment 8. The part enclosed in the dotted line in (f) shows the shape of the input data structure that can cause the traversal of the target path <1,2,3,4,6,8,9,10,11>.

to ⊥. We also assume that the pointer variables are initially partitioned into singleton collections consisting of only one pointer variable. Fig. 7(a). depicts the shape graph corresponding to the initial state σ_0.

After evaluating statements 1 and 2, we get the following points-to information:

- x_0 points to 1_x_0 and
- y_0 points to 1_y_0

That is to say, the effect of assignments 1 and 2 is to introduce new points-to relations by materializing the locations named 1_x_0 and 1_y_0 from ⊤ pointed to by x_0 and y_0, respectively. We also observe that since they define p and q, respectively, their last versions are changed to p_1 and q_1. As a result we yield the shape graph in Fig. 7(b)

Let us consider the boolean expression "p==q". Its effect is 1) to leave the current state unchanged

if the last versions of p and q, say p_1 and q_1, point to the same location, 2) to make p_1 and q_1 belong to the same collection if they can possibly point to the same location, or to indicate that the boolean condition cannot evaluate to true if p_1 and q_1 are in conflicts.

The present case is applicable to 2). Consequently, the top nodes pointed by p_1 and q_1 are merged, so that it indicates that p_1 and q_1 should point to the same location as shown in Fig. 7(c).

Now, we are in a position to evaluate the boolean expression "p==NULL". Since we want the boolean expression to evaluate to false, we can consider the form "p<>NULL" instead. It allows us to exclude the case where both p_1 and q_1 will have the NULL value. Thus, the top node pointed by both p_1 and q_1 (of course, also pointed by 1_x_0 and 1_y_0) is changed to the node labelled with **NOT-NULL** which we need to name. The candidates include 1_p_1, 2_x_0, 2_y_0, and 1_q_1. It does not matter which one is used. Fig. 7(d). shows the situation where 2_x_0 is selected as the name of the NOT-NULL node.

Similarly, we can evaluate the boolean expression "q<>NULL". Fig. 7(d) shows the current instance of q, q_1, should point to the node named 2_x_0. It means that q cannot be NULL when evaluating the boolean expression. Thus, nothing will be affected.

Here, it seems interesting to imagine the situation where it is required that the boolean expression "q==NULL" should evaluate to true. Suppose that we want to exercise the path <1,2,3,4,6,7>. In order to cause the traversal of the program point 7, the current instance of q, q_1, should be NULL. This will cause a contradiction since the current state as shown in Fig. 7(d) requires that q_1 should not be NULL. As a result we would get $\perp_\sigma$ indicating that an infeasible path is detected.

Whenever we encounter a statement such as statement 9 where either of its left-hand side or its right-hand side involves the pointer dereference, we convert the statement into SSA form without pointer dereferences using the points-to information. Fig. 7(e) shows the points-to information that r_1 points to z_0 introduced immediately after evaluation of the assignment 8. Consequently, the variable z_0 will be defined at the assignment 9. In addition, the function GP generates new versions of r and z: r_2 and z_1. As a result, the assignment is converted into SSA form without pointer dereference: "z_1=10".

What happens if neither left-hand side nor right-hand side of an assignment is of pointer type? In this case, traditional conversion to SSA form will be performed by replacing the variables with their last (SSA) versions. For example, conversion of the boolean expression "z==v" at statement 10 into SSA form "z_1== v_0".

The last statement we need to consider is "*q=v". Its evaluation is carried out in the same manner as that of the assignment 9 by using the points-to information of q. Since q_1 points to 2_x_0, the assignment is converted into the SSA form without the pointer dereference: "2_x_0= v_0".

We are interested in the portion of the final shape graph that is associated with initial input (pointer) variables because it gives a shape of the input data structure required to traverse the selected path. By initial input variables we mean the versions of input variables before any modification; they have 0 as their subscript number. The partial shape graph enclosed by dotted lines in Fig. 7(f). shows the input data structure required to traverse the selected path.

Once we have found the input data structure required to traverse the selected path, we need to find values for input variables of non-pointer types. We can find such values by solving the constraints generated from the selected path. Having the constraints, we can apply various methods [3-6,8-9] to come up with a solution.

In the example, we have the constraints as follows:

z_1==10

z_1==v_0

2_x_0==v_0

It is not difficult to see that the solution will be: z_1:10, v_0:10, p_2:10.

The variable v_0 is only of concern to us since it gives the input value of v which suould be supplied from outside. Consequently, we need to form the input data structure as the part enclosed by the

dotted line shown Fig. 7(f) and provide 10 for the value of the formal parameter v.

4. Heap-Based Data Structure

So far, we have focused on pointers pointing to statically-allocated memory objects (typically stacks). In this section, our approach will be extended to cope with *heap-directed* pointers, which point to objects dynamically allocated in the heap. Heap-directed pointers often involve structures. A structure has multiple fields, each of which is accessed using field identifiers. For the sake of simplicity, we assume that each field can either be an integer, a pointer to another location or it can be NULL.

4.1 Transfer functions for structures and heap-directed pointers

Introducing structures and heap-directed pointers necessitate to deal with the statements of the form:

- x.f=expression
- k=x.f
- x=y
- x.f rel k where rel is one of {>,<,<>,==,<=,>=}.[1)]
- p→f=k
- k=q→f
- p=malloc(-)
- free(p)
- p→f rel k

In order to define the transfer functions associated with these statements, we firstly present how to assign SSA numbers to structures. Since we need to treat every field in a structure as a separate variable, we associate SSA numbers with all the fields as well as with the structure itself. For the SSA numbering, we adopt the rule presented in [14]; assignment to a field(i.e., x.f=...) increments the SSA numbers associated with the field and a structure copy assignment (i.e., x=y) changes the SSA number associated with the structure as a whole. Fig. 8 shows a program fragment illustrating how to assign SSA numbers to structures. Note that only the structure copy assignment generates a new version of a variable of structure type.

1) If k is of pointer type, rel will be allowed to be one of {<>, ==}.

```
struct foo {          struct foo {
    int f;                int f;
    int g;                int g;
} x, y;               } x, y;
x.f=10;               x0.f1=10;
x.g=20;               x0.g1=20;
y.f=x.f;              y0.f1=x0.f1;
x.f=30;               x0.f2=30;
y=x;                  y1=x0;
x.g=y.f;              x0.g2=y1.f1;
```

(a) (b)

Fig. 8 A program fragment showing SSA numbers for structures

The transfer functions associated with the first four forms make no difference from those associated with ordinary assignments. The reason is that the variable of the form x.f can be regarded as a separate variable. Thus, we will not go into details. From now on, we will assume that the function GP is changed to cope with structures accordingly.

Fig. 9 shows the transfer functions for the statements involving the pointer dereference operator '→' and the functions for allocating and releasing memory locations: malloc() and free(). The functions involving the pointer dereference operator are transformed into the forms without pointer dereferences using the points-to information as in the case of dealing with the pointer dereference operator '*'.

First of all, it is checked what each pointer variable is pointing to. If a pointer variable points to a node labeled with $\top$, then a memory location is materialized from the top node. In this case, we need to name the newly created (anonymous) location. As stated earlier, this can be done by using the function "new-name".

For example, suppose that x is a pointer variable which *can* point to any structure in the state σ, i.e., $\sigma(last_\sigma(x))=\top$. We also assume that the collection $[x]_\sigma$ is the singleton set $\{last_\sigma(x)\}$. Then, the invisible variable $1\_last_\sigma(x)$ can be used to stand for the structure materialized from the top node and the pointer dereference "x→f" is replaced with the invisiable variable $1\_last_\sigma(x).f$. This can be

$\| y=x\rightarrow f \|\sigma$ = $\| y==new\text{-}name(x,\sigma).f \| GP(y,\sigma_x)$ if $\sigma(last_\sigma(x))=\top$

= $\| y==m_x.f \| GP(y,\sigma)$ else if $\sigma(last_\sigma(x))$=**NOT-NULL** or l_{mx}

= $\perp_\sigma$ otherwise

$\| x\rightarrow f=y \|\sigma$ = $\| new\text{-}name(x,\sigma).f==y \| GP(x,\sigma_x)$ if $\sigma(last_\sigma(x))=\top$

= $\| m_x.f==y \| GP(m_x,\sigma)$ else if $\sigma(last_\sigma(x))$=**NOT-NULL** or l_{mx}

= $\perp_\sigma$ otherwise

$\| p=malloc(-) \|\sigma$ = $\| p=\&heap\_loc \|\sigma$, where heap_loc refers to a location allocated from heap

$\| free(p) \|\sigma$ = $\| p=NULL \|\sigma$

Fig. 9 The transfer functions for the statements involving the pointer dereference operator '→', malloc() and free() functions.

viewd as a process of concretizing a shape of the data structure; the top node $\top$ can be regarded as the 'primordial soup' [15]. Since the other cases of the transfer functions shown in Fig. 11 are similar to the transfer functions associated with the statements "y=*x" and "*x=y", we will not give more explanation.

Fig. 9 also includes the transfer function for the statement "p=malloc(-)" which creates a new location from heap which will be pointed to by p. Since the statement creates an anonymous object, we need to name it. To the end, we use the place (statement) in the program, prefixed by the word "heap_", where an anonymous heap object is created.

Let us consider the following sequence of code:

1: p=malloc(-);

2: q=p;

3: p=malloc(-);

Then, the sequence of code can be converted int SSA form as follows: p_1=&heap_1; $q_1=p_1$; p_2= &heap_3;

The analysis of the sequence of code comes up with the desired result that p_1 and q_1 point to the same heap object named 'heap_1', but p_2 points to the heap object named 'heap_3'.

The memory release function "free(p)" returns the memory location pointed to by p to the heap. Without loss of semantic information, this is equal to say that p does not point to anything after "free(p)". Thus, we can use the transfer function associated with the assignment "p=NULL" for "free(p)".

Finally, we need to consider the boolean expression of the form

p→f rel k

We will not show the transfer functions associated with the boolean expression of the above form because they can be defined in the same manner as those associated with the assignments of the form "p→f=k" (or k=p→f). That is to say, the points-to information about what the pointer variable p is pointing to will be used to transform the boolean expression into a boolean expression involving no the pointer dereference operator →.

```
struct Node {
        int data;
        struct Node *left;
        struct Node *right;
};
typedef struct Node *NodePointer;

void Find(NodePointer L, int y, NodePointer q) {
        NodePointer p;
1:  p = L;
2:  q = NULL;
3:  while (p != NULL) {
4:      if (y == p->data) {
5:          q = p;
6:          p = NULL;
        }
        else {
7,8:        if (y < p->data) p = p->left;
9:          else p= p->right;
        }
        }
}
```

Fig. 10 An example program

4.2 Example

Fig. 10 shows an example program that has been taken from [7]. Suppose that we want to identify a shape of the input data structure that will traverse the path <1,2,3,4,7,8,3,4,7,9,3,4,5,6,3>.

Fig. 11(a) shows the partial shape graph showing the initial state with which the analysis starts. For the sake of simplicity, we will depict only what pointer variables are pointing to.

Fig. 11(b) shows the state after evaluating the sub-path <1,2>. It is observed that the pointer variable p points to whatever L is pointing to after evaluating statement 1 and q has the NULL value after evaluating statement 2 (this is not explicitly shown in Fig. 11(b)). Note that the subscripts of p and q have been incremented by one because the assignments define the variables, respectively.

The next one we need to evaluate is the boolean expression "p<>NULL". The result should be a maximal state that evaluates the expression to true. Such a state can be obtained by materializing a concrete location from the top node pointed to by p_1 and L_0. This is equal to say that a structure is created and it is pointed to by p_1 and also by L_0. Consequently, we will have a state depicted in Fig. 11(c). Whenever a location is created, we ensure

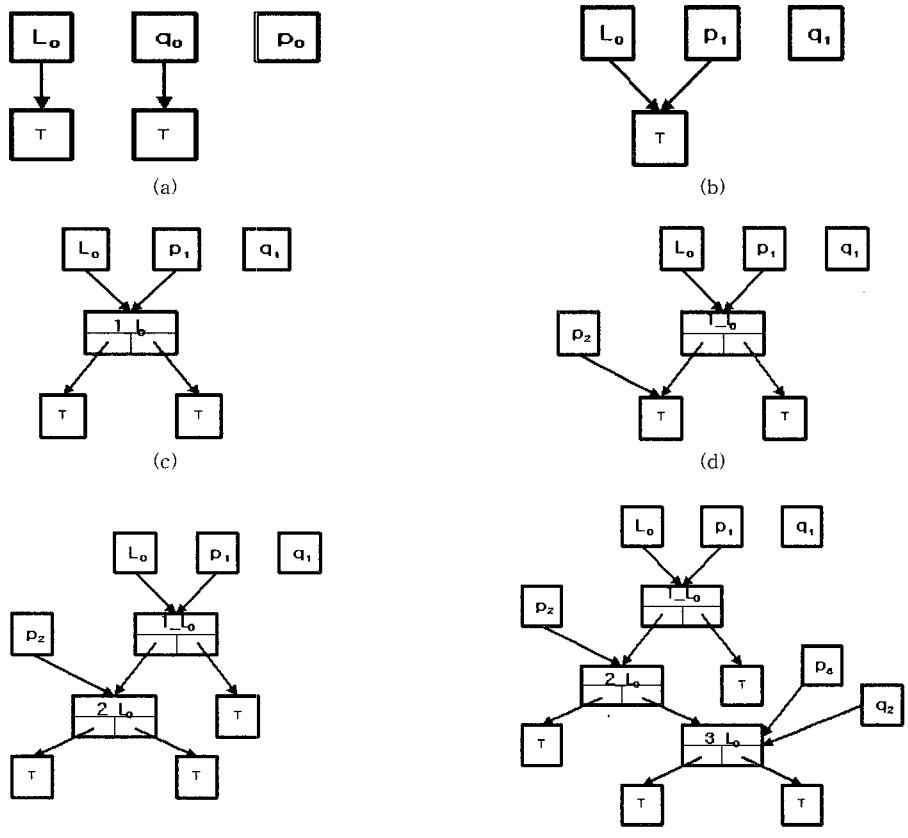

Fig. 11 The shape graphs that arise when the shape analysis algorithm is applied to the given path of the example program in Fig. 10; (a) depicts the initial state σ0, (b) depicts the shape graph after evaluating the sub-path <1,2>, (c) depicts the shape graph after evaluating the sub-path <1,2,3>, (d) depicts the shape graph after evaluating the sub-path <1,2,3,4,7,8>, (e) depicts the shape graph after evaluating the sub-path <1,2,3,4,7,8,3,4,7>, and (f) depicts the shape graph after evaluating the target path <1,2,3,4,7,8,3,4,7,9,3,4,5,6,3>.

that all pointer fields can possibly point to any place by initializing them with ⊤. Note that the newly created location has been labeled with 1_L_0 using the notion of invisible variable.

Evaluation of the boolean expression "y<>p→data" does not affect the points-to information, but generates a constraint without the pointer dereference as follows: "y_0<>1_L_0.data". Similarly, boolean expression "y<p→data" will be transformed into the constraint "y_0<1_L_0.data". Thus, the constraints will be reduced to "y_0<1_L_0.data".

The next statement we have to consider is "p=p→left". First of all, we have to check what the current instance of the pointer variable p (i.e., p_1) is pointing to. Fig. 11(c) shows that p_1 points to a location named 1_L_0. As a result, we can rewrite the assignment into the SSA form involving no pointer dereferences as follows: "p_2=1_L_0.left". Evaluation of the SSA form will lead to the state depicted in Fig. 11(d).

The next statement we have to consider is the boolean expression "p<>NULL". Evaluation of the boolean expression ensures that the location pointed to by the current instance of p, i.e., p_2 should not be NULL. Consequently, the top node pointed to by p_2 needs to be materialized into the node named 2_L_0. The result will be the shape graph depicted in Fig. 11(e).

We continue to evaluate the boolean expressions "y<>p→data" and "y>p→data". We can observe that the evaluation does not affect the points-to information, but generates an additional constraint "y_0>2_L_0.data" which should be satisfied by the data field of the structure pointed to by p_2.

Now we are in a position to evaluate the assignment "p=p→right" and then "p<>NULL". The total effect is to create a (NOT-NULL) node which will be pointed by p_3 and 2_L_0.right. All we have to do at the moment is to evaluate the statements in the sub-path <4,5,6>. The boolean expression "y==p→data" will associate an invisible variable, say 3_L_0 with the NON-NULL node pointed to by p_3 and 2_L_0.right and be transformed into the equality as follows: y_0==1_L_3.data. Evaluation of the assignments "q=p" and "p=NULL" can be done without any difficulties. The result is shown in Fig. 11(f).

The input data structure shown in Fig. 11(f) needs to be materialized because it actually embodies all possible shapes that will cause the traversal of the selected path. The pointer fields of the nodes have the top value, meaning that it does not matter whatever values they take. In order to keep an input data structure as simple as possible, we can assign NULL to the pointer fields. In addition, the values of each data field of the structures can be generated by solving the constraints:

y_0<1_L_0.data

y_0>2_L_0.data

y_0==1_L_3.data

Assuming that 10, 20, 8, and 10 have been chosen for the input variable y and the data fields, respectively, the following input data structure will be created:

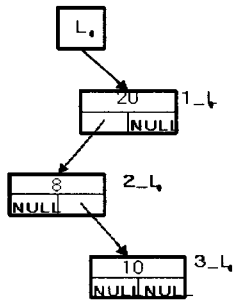

Fig. 12 A shape of the input data structure

5. Conclusion

Most work in automated test data generation has focused on finding input values for non-pointer types. However, handling pointers is crucial to test data generation for programs written in procedural languages such as C. In this paper, we have presented a static approach to determine a shape of input data structures required to cause the traversal of a selected path in the presence of pointers.

We can use an approach to program analysis called shape analysis that has been studied for applications including compiler optimization and parallelization [15]. Shape analysis allows us to determine the space of shapes that input data structures might have for all possible execution paths. However, we do not directly apply shape

analysis techniques to test data generation problem because we are interested in a specific path rather than in all possible execution paths. Even though we limit shape analysis to the selected path, overly conservative information might be produced, meaning that the analysis result may contain the shapes of an input data structure that can not traverse the selected path. This is in part because unbounded data structures need to be summarized in some finite way. Note that we only need to find a shape of an input data structure that can traverse the selected path. Since a program path is assumed to be finite, we do not also have to consider the problem that the summarization of unbounded data structures gives rise to.

For our purpose, we have defined the transfer functions associated with various forms of boolean expressions and assignments. Basically, the transfer functions collect points-to information for each program point and introduce new points-to information whenever necessary. In addition, each statement is converted into SSA form without pointer dereferences, so that each statement can be treated as constraints involving (in)equalities. Thus, we can make use of current constraint solving techniques to solve the constraint system to get a solution for non-pointer variables.

References

[1] Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Program," *IEEE Trans. on Software Eng.* vol. 2. no. 3. pp. 215-222, 1976.

[2] Gallagher, M. J. and Narasimhan, V. L. "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Trans. on Software Eng,* vol. 23. no. 8. pp. 473-484, 1997.

[3] Gotlieb, A., Botella, B., Rueher, M., "Automatic Test Data Generation using Constraint Solving Techniques," In *Proc. ACM ISSTA,* pp. 53-62, 1998.

[4] Gupta, N., Mathur, A., Soffa, M. L., "Automated test data generation using an iterative relaxation method," In *Proc. ACM SIGSOFT Foundations of Software Engineering,* pp. 231-244, 1998.

[5] Gupta, N., Mathur, A., Soffa, M. L., "UNA based iterative test data generation and its evaluation," In *Proc. 14th IEEE Int. Conf. on Automated Software Eng.* pp. 224-232, 1999.

[6] Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," *IEEE Trans on Software Eng.,* vol. 4. no. 4. pp. 266-2781, 1977.

[7] Korel, B. "Automated Software Test Data Generation," *IEEE Trans. onSoftware Eng,* vol. 16. no. 8. pp. 870-879, 1990.

[8] Lapierre, S., Merlo, E., Antoniol, G., Fiutem, R., Tonella, P., "Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees," In *Proc. Int. Conf. on Software Maintenance.* pp. 189-198, 1999.

[9] Meudec, C., "ATGen:Automatic Test Data generation using Constraint Logic Programming and Symbolic Execution," In *Proc. IEEE/ACM Int. Workshop on Automated Program Analysis Testing and Verfication.* pp. 22-31, 2000.

[10] Roger, F., Korel, B, "The Chaining Approach for Software Test Data Generation," *ACM Trans. on Soft. Eng. Methodology,* vol. 5. no.1. pp.63-86, 1996.

[11] Offutt, J., Pan, J., "The Dynamic Domain Reduction Approach to Test Data Generation," *Software-Practice and Experience,* vol 29. no. 2. pp. 167-193, 1997.

[12] Emami, M., Ghiya, R., Hendren, L., "Context Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," In *SIGPLAN Conf. Prog. Lang. Design and Impl.*, 1994.

[13] Cytron, R. Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans on Programming Languages and Systems.* vol. 13. no. 4. pp. 451-490, 1991.

[14] Lapkowski, C., Hendren, L. J., "Extended SSA Numbering: Introducing SSA properties to Languages with Multi-level Pointers," ACAPS Tecnical Memo 102. School of Computer Science. McGill Univ. Canada. 1996.

[15] Sagiv, M., Reps, T., Wilhelm, R., "Solving Shape-Analysis Problems in Languages with Destructive Updating," *ACM Trans. Prog. Lang. Syst.* vol. 20. no. 1. pp. 1-50, 1998.

정 인 상

1983년~1987년 서울대학교 컴퓨터공학과(학사). 1987년~1989년 한국과학기술원 전산학과(석사). 1989년~1993년 한국과학기술원 전산학과(박사). 1993년 8월~1994년 2월 한국전자통신연구원 박사후 연수 연구원. 1994년~1998년 한림대학교 부교수. 1994년 3월~1994년 12월 한국전자통신연구원 초빙연구원. 1995년 7월~1995년 8월 영국 Durham 대학 Centre for Software Maintenance 방문연구원. 1997년 1998년 미국 purdue 대학 방문교수. 1999년~현재 한성대학교 컴퓨터 공학부 부교수