효율적인 포인터 오류 검증

이욱세
한양대학교 컴퓨터공학과

2007. 1. 31

```
*** STOP: 0x00000019 (0x00000000,0xC00E0FF0,0xFFFFEFD4,0xC0000000)
BAD_POOL_HEADER

CPUID:GenuineIntel 5.2.c irql:1f  SYSVER 0xf0000565

Dll Base DateStmp - Name                Dll Base DateStmp - Name
80100000 3202c07e - ntoskrnl.exe        80010000 31ee6c52 - hal.dll
80001000 31ed06b4 - atapi.sys           80006000 31ec6c74 - SCSIPORT.SYS
802c6000 31ed06bf - aic78xx.sys         802cd000 31ed237c - Disk.sys
802d1000 31ec6c7a - CLASS2.SYS          8037c000 31eed0a7 - Ntfs.sys
fc698000 31ec6c7d - Floppy.SYS          fc6a8000 31ec6ca1 - Cdrom.SYS
fc90a000 31ec6df7 - Fs_Rec.SYS          fc9c9000 31ec6c99 - Null.SYS
fc864000 31ed868b - KSecDD.SYS          fc9ca000 31ec6c78 - Beep.SYS
fc6d8000 31ec6c90 - i8042prt.sys        fc86c000 31ec6c97 - mouclass.sys
fc874000 31ec6c94 - kbdclass.sys        fc6f0000 31f50722 - VIDEOPORT.SYS
feffa000 31ec6c62 - mga_mil.sys         fc890000 31ec6c6d - vga.sys
fc708000 31ec6ccb - Msfs.SYS            fc4b0000 31ec6cc7 - Npfs.SYS
fefbc000 31eed262 - NDIS.SYS            a0000000 31f954f7 - win32k.sys
fefa4000 31f91a51 - mga.dll             fec31000 31eedd07 - Fastfat.SYS
feb8c000 31ec6e6c - TDI.SYS             feaf0000 31ed0754 - nbf.sys
feacf000 31f130a7 - tcpip.sys           feab3000 31f50a65 - netbt.sys
fc550000 31601a30 - el59x.sys           fc560000 31f8f864 - afd.sys
fc718000 31ec6e7a - netbios.sys         fc858000 31ec6c9b - Parport.sys
fc870000 31ec6c9b - Parallel.SYS        fc954000 31ec6c9d - ParVdm.SYS
fc5b0000 31ec6cb1 - Serial.SYS          fea4c000 31f5003b - rdr.sys
fea3b000 31f7a1ba - mup.sys             fe9da000 32031abe - srv.sys

Address  dword dump   Build [1381]                            - Name
fec32d84 80143e00 80143e00 80144000 ffdff000 00070b02          - KSecDD.SYS
801471c8 80144000 80144000 ffdff000 c03000b0 00000001          - ntoskrnl.exe
801471dc 80122000 f0003fe0 f030eee0 e133c4b4 e133cd40          - ntoskrnl.exe
80147304 803023f0 0000023c 00000034 00000000 00000000          - ntoskrnl.exe

Restart and set the recovery options in the system control panel
or the /CRASHDEBUG system start option.
```

포인터 오류

- Java 덕에 포인터 오류가 줄었지만,
- C/C++로 여전히 많은 프로그램, 특히 OS 및 디바이스 드라이버가 작성되고 있고 오류를 생산하고 있음
  - 디바이스 드라이버: OS 개발자가 아닌 비전문가가 개발, 포인터를 과도하게 사용
- 프로그램 분석기에서 포인터 연산은 종종 무시됨
  - 포인터가 있으면 분석의 안전성이 보장 안됨

포인터 오류 검증 도구의 덕목

- 빠르고
  - 느린 검증 도구는 실제 현장에서 불편
- 정확
  - 거짓 보고(false alarm)가 많은 도구는 불편

- 현재까지의 결과 [이욱세 2006, 프로그래밍언어논문지 20(1)]

	셀 식별	접근오류	메모리 누수	성능
포인터분석	△	X	X	O
출생지기반분석	△	△	△	O
모양분석	O	O	O	X

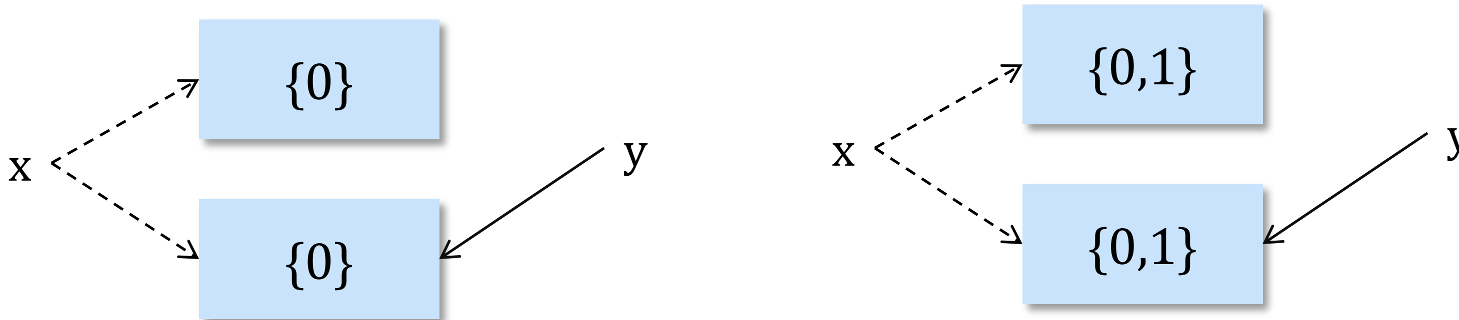
모양 분석 (Shape Analysis)

[Sagiv et al. 2002, TOPLAS 20(1)]

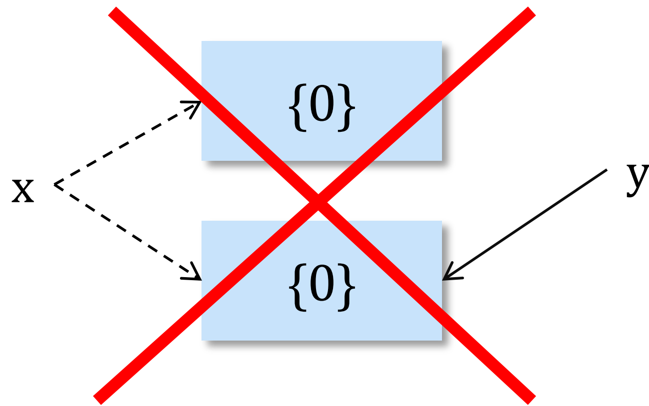
- 포인터 오류 검증의 새로운 가능성 제시
- 정확한 포인터 추적 가능
  - 완전한 변경(strong update)
  - 재귀적 자료구조를 적절히 요약
- 속도가 현재까지도 문제

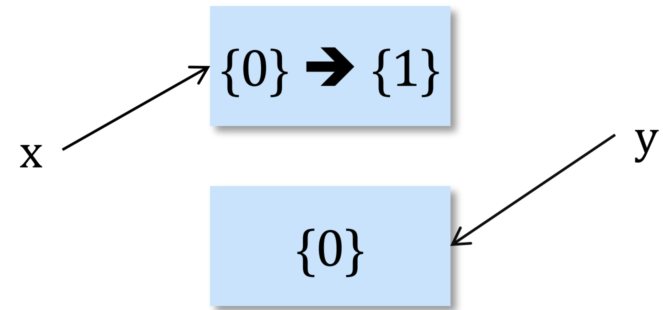
완전한 변경 (Strong Update)

- 포인터 저장문에 대해 제대로 분석하는지 여부
  - *x가 0일 때, *x = 1; 이후의 분석 결과는?
  - x 가 가리키는 셀이 확실할 때는 *x = { 1 } **(완전)**
  - x 가 가리키는 셀이 확실치 않을 때는 *x = { 0, 1 } **(불완전)** 게다가 다른 셀의 내용 변경까지 가능

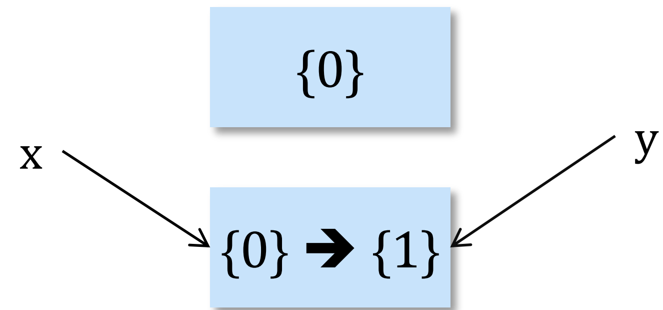


모양 분석에서 완전한 변경: 집합



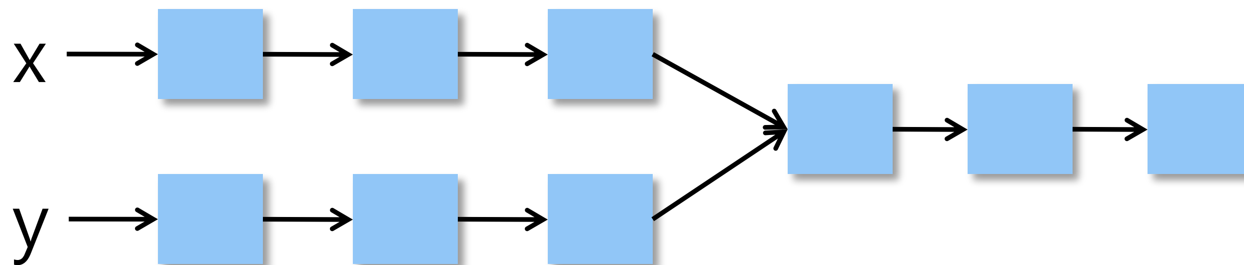


다양한 경우를 집합으로
관리하여 따로따로 분석



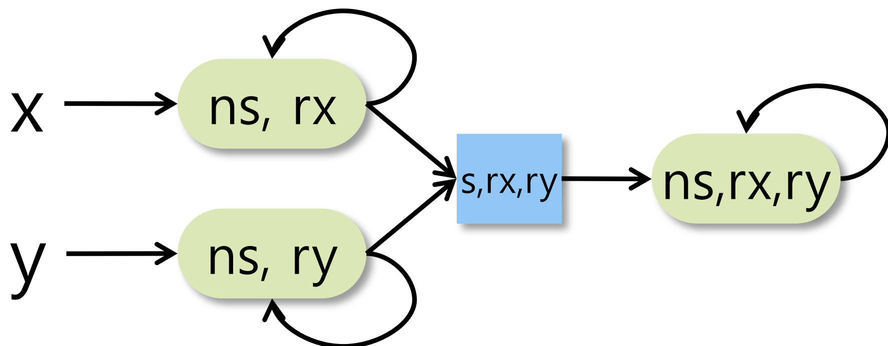
재귀적 자료 구조의 적절한 요약

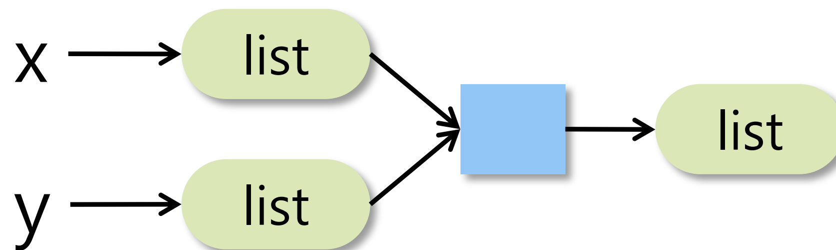
- 공유된 리스트



셀의 속성으로 요약
[Sagiv *et al.* 2002, TOPLAS]

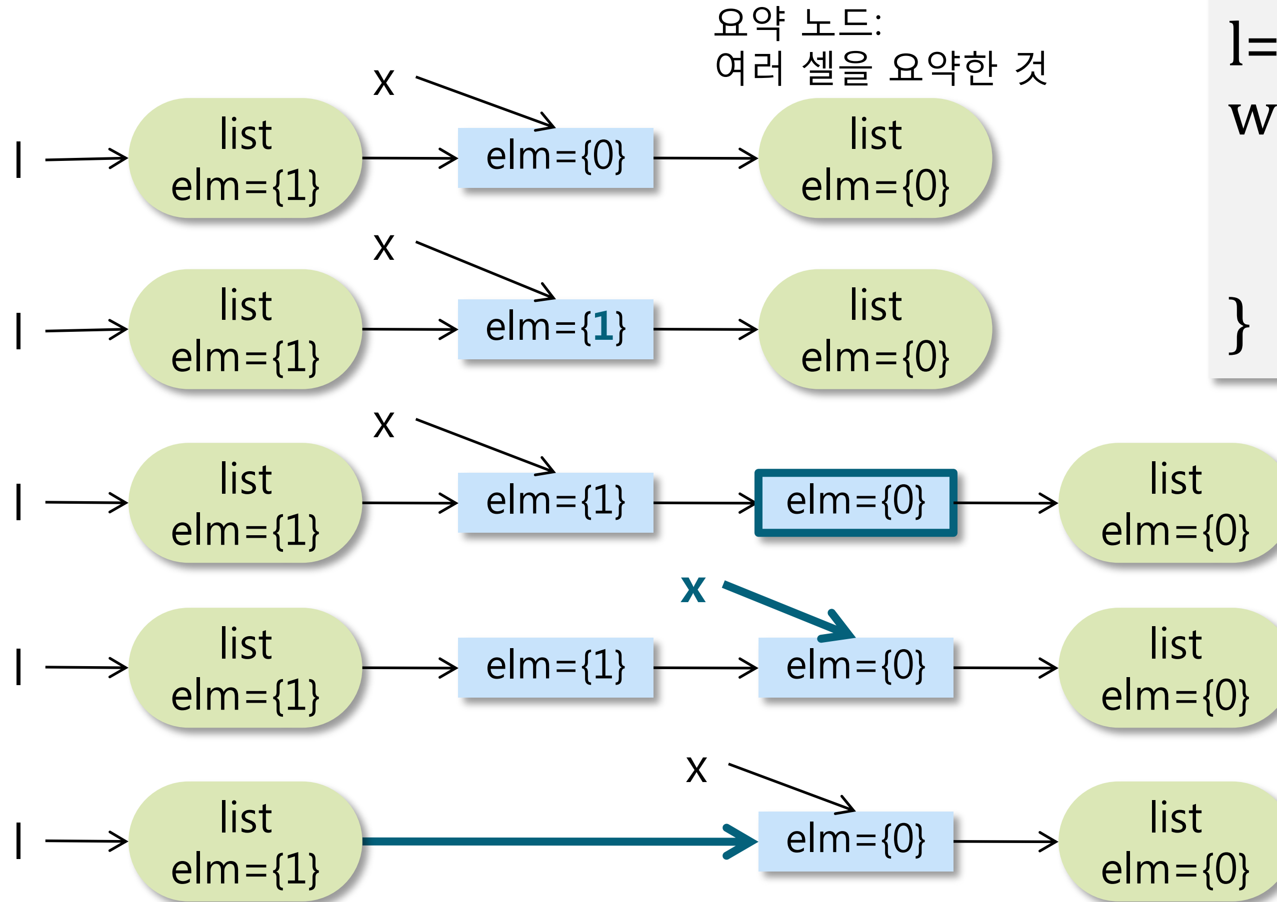
객체의 속성으로 요약
[Lee *et al.* 2005, ESOP]
[Berdine *et al.* 2007, CAV]





모양 분석에서 완전한 변경: 꺼내기, 넣기

```
l=x=list0;
while(x!=NULL) {
   x->elm = 1;
   x=x->next;
}
```



꺼내기!

넣기!

모양 분석 정리

- 포인터 오류 검증의 해답에 가까운 유력 후보!

- 메모리 집합을 표현하는 모양 그래프가 기본 단위
- 모양 그래프의 멱집합(powerset)을 요약 도메인(abstract domain)으로 하는 프로그램 분석기
- 특별한 연산
  - 꺼내기 (focusing): 포인터 연산 대상 셀에 대해 수행
  - 넣기 (abstraction 또는 summarization)
  - 합치기 (join)

오늘의 주제는

- **모양 분석(shape analysis)의 효율적인 수행**
- **CASE 1:** [Yang *et al.* 2008, TR, Queen Mary U. of London]
  - **복잡한 모양 분석 도메인: 이중 양방향 순환 리스트 (nested cyclic doubly-linked list)**
  - **문맥 민감 프로시저 간 분석 (context-sensitive interprocedural analysis)**
- CASE 2: [이욱세 외 6인, ETRI]
  - 단순 모양 분석 도메인: 트리 (tree)
  - 문맥 둔감 프로시저 간 분석 (context-insensitive interprocedural analysis)

프로시저 간 분석 (Interprocedural Analysis)

- 프로시저 호출을 통한 상호작용을 고려한 분석

```
int f(int x) { return x; }
int main() { printf("%d\n", f(1)+f(2)); }
```

- 문맥 둔감 (context insensitive)
  - 여러 호출을 하나로 묶어서 분석
  - 위 예제에서 f(1), f(2) 두 경우를 합쳐 x는 {1, 2} 이므로, 프로시저 f의 결과는 {1, 2}, 그러므로 f(1)+f(2)는 {2, 3, 4}.
- 문맥 민감 (context sensitive)
  - 여러 호출을 별도로 분석
  - 위 예제에서 f(1)의 경우 x는 1이므로 결과는 1, f(2)의 경우 x는 2이므로 결과는 2, 그러므로 f(1)+f(2)=3

테이블 기반 문맥 민감 프로시저 간 분석

- 각 프로시저 별로 입력, 결과 요약 상태를 표에 저장하고, 같은 입력에 대해서는 다시 분석하지 않는 방법

  [Reps *et al.* 2005, POPL]

f(1) → 1
f(2) → 2
f(2) → 2

입력	결과
1	1
2	2

- 모양 분석에서는 모양 그래프 개별 단위로 표에 저장

정말로 실제적인 분석이 될 수 있을까?

- 악재
  - 복잡한 모양 분석 도메인
  - 테이블 기반 문맥 민감 프로시저 분석
- 접근 방법
  - 경우 줄이기: 최적화, 요약 정도를 높임, 지역성 활용 (localization)
  - 메모리 줄이기: 유지하는 표의 크기 감소 및 중간 결과 버리기

BEFORE

Routine	LOC	Space (Mb)	Time (sec)	Result
t1394_BusResetRoutine	718	322.44	663	✓
t1394Diag_CancelIrp	693	1.97	0.56	⊘
t1394Diag_CancelIrpFix	697	263.45	724	✓
t1394_GetAddressData	693	2.21	0.61	⊘
t1394_GetAddressDataFix	698	342.59	1036	✓
t1394_SetAddressData	689	2.21	0.59	⊘
t1394_SetAddressDataFix	694	311.87	956	✓
t1394Diag_PnpRemoveDevice	1885	>2000.00	>9000	T/O

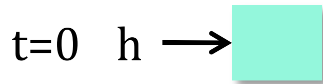
AFTER

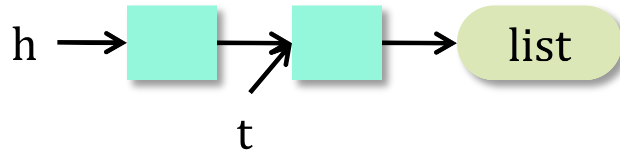
Program	LOC	Sec	Mb	Memory Leaks	Dereference Errors
pci-driver.c	2532	0.79	3.19	0	0
cdrom.c	6218	91.88	84.30	0	2
t1394Diag.c	10240	145.95	71.27	33	10
md.c	6635	1440.53	814.45	6	5
ll_rw_blk.c	5469	997.66	523.22	3	1
class.c	1983	6.68	8.36	0	0
scull.c	1010	0.21	1.47	0	0

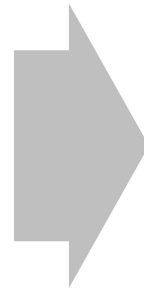
경우 줄이기 #1. 무효 변수 제거

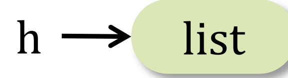
- 프로그램 일부분에서 잠깐 사용되고 더 이상 사용되지 않는 변수가 경우의 수를 증가시킴

t=? h=0







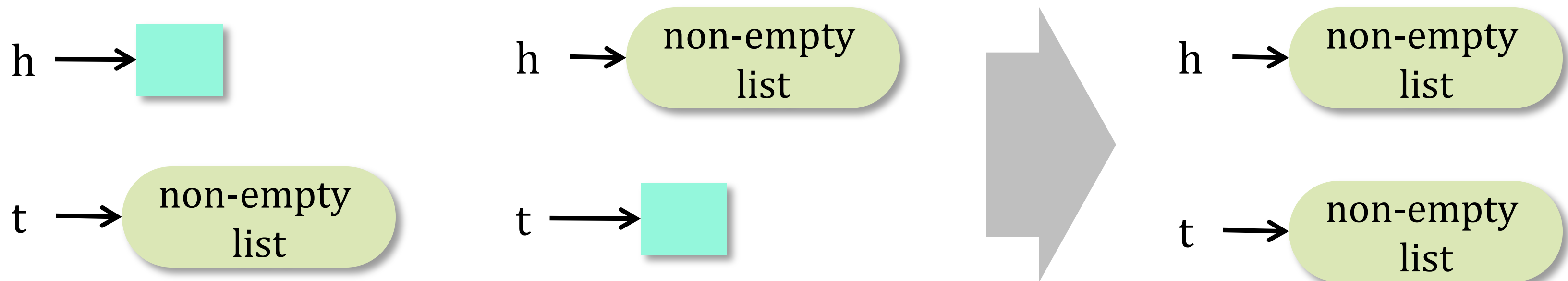



```
List* t, h=0;
/* t=0; */
while (nondet) {
    t = h;
    h = malloc();
    h->next=t;
    /* t=0; */
}
```

- 프로시저 내 생존 분석(liveness analysis)을 통해 더 이상 사용되지 않는 변수를 제거
  - 수작업한 코드의 분석 속도를 향상

경우 줄이기 #2. 과감한 경우 결합 (Join)

- 예전의 분석은 여러 경우로 나누어진 것을 다시 결합하는 연산(join)이 비효율적
  - 거의 합집합 수준: 다음 두 경우를 합치지 못함



- 확장 연산 (widening) 수준의 과감한 결합
  - 무한루프가 발생할 수 있는 프로시저의 출입구, 반복문에서만 사용

속도 향상 실험 결과: 결합

프로그램	결합 없이 경우의 수	결합 사용시 경우의 수
onelist.c	3	2
twolist.c	9	4
firewire.c	3,969	37

프로그램	줄	결합 없이		결합 사용	
		속도 (s)	메모리 (MB)	속도(s)	메모리 (MB)
t1394Diag.c (일부)	973	184.87	575.82	0.36	2.70
t1394Diag.c (일부)	1,825	> 90 min	-	1.21	5.16
pci-driver.c	2,532	> 90 min	-	0.55	4.42
cdrom.c	6,218	> 90 min	-	107.91	357.58

경우 줄이기 #3: 빈 리스트 처리

- 빈 리스트
  - 요약 노드가 빈 리스트(empty list)를 포함하는지 여부가 버그 찾는데 중요
  - 일반적으로 모양분석에서는 빈 리스트와 비지 않은 리스트를 구분하여 분석
  - 결과는 경우 폭발
- 예제: IEEE1394 드라이버
  - 한 구조체(structure)에는 다섯 개의 리스트 필드 존재
  - 각 필드 별로 빈 리스트, 비지 않은 리스트, 2 가지 경우가 나오므로 최소 $2^5 = 32$ 가지 경우가 발생
- 리스트의 요약 노드에 빈 리스트 포함 여부 표기
  - NE (non-empty), PE (possibly empty)
  - NE < PE 순서에 따라 합쳐 주면 (join) 경우의 수가 감소

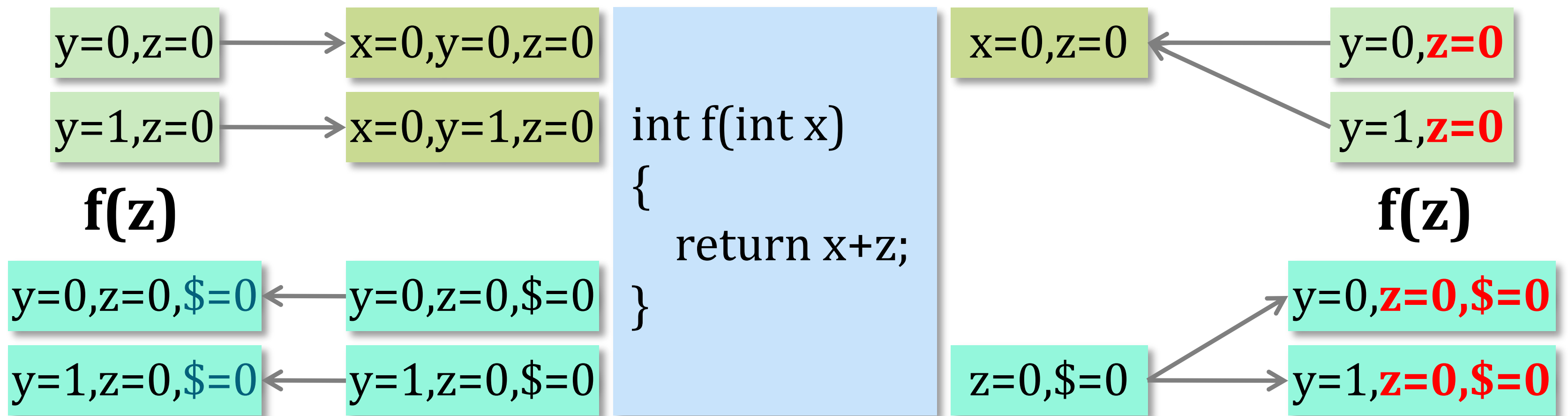
속도 향상 실험 결과: PE

프로그램	PE 없이 경우의 수	PE 사용시 경우의 수
onelist.c	2	1
twolist.c	4	1
firewire.c	37	1

프로그램	줄	NE		NE & PE	
		속도 (s)	메모리 (MB)	속도(s)	메모리 (MB)
t1394Diag.c	10,240	> 90 min	-	119.40	425.16
pci-driver.c	2,532	> 90 min	-	0.55	4.42
cdrom.c	6,218	> 90 min	-	107.91	357.58

경우 줄이기 #4: 프로시저 분석시 지역성 활용

- 프로시저 입력 상태에 프로시저와 상관 없는 것이 포함되면 경우의 수가 증가



- 프로시저와 상관 있는 것
  - 프로시저에서 사용하는 변수에서 도달 가능한 (reachable) 셀 들

속도 향상 실험 결과: 지역성

프로그램	줄	지역성 없이		지역성 사용	
		속도 (s)	메모리 (MB)	속도(s)	메모리 (MB)
t1394Diag.c (일부)	973	0.64	3.69	0.36	2.70
t1394Diag.c (일부)	1,825	2.70	9.58	1.21	5.16
pci-driver.c	2,532	1.75	9.09	0.55	4.42
cdrom.c	6,218	> 90 min	-	107.91	357.58

메모리 줄이기 #1. 듬성 듬성 분석 표

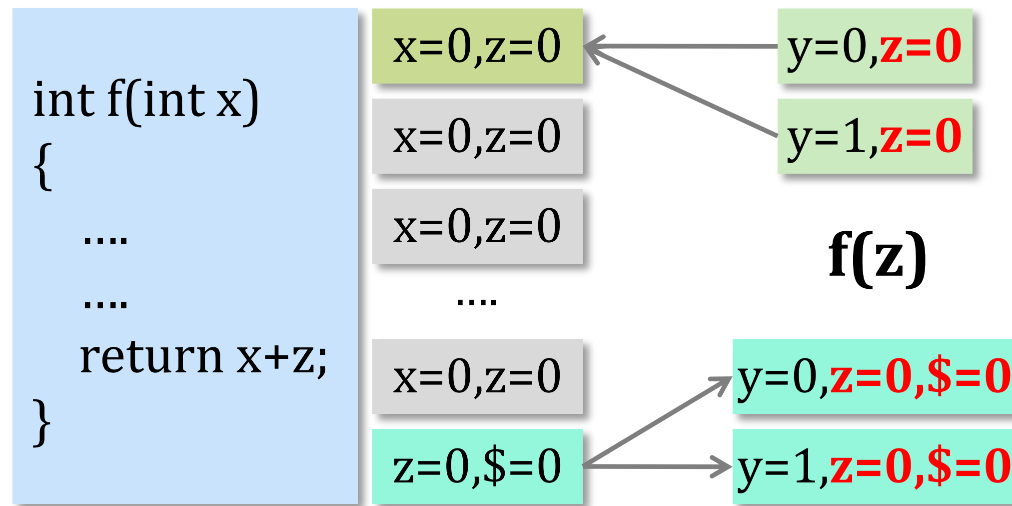
- 모든 프로그램 지점의 중간 결과를 저장할 필요는 없음
  - 고정점에 도달했다는 것을 확인하기 위해 필요
- 반복문, 결합지점 등 중요한 부분에서만 중간 결과 저장 및 고정점 확인
  - 기본 블록 (basic block) 개념

```
int f(int x)
{
   ....
   ....
   ....
   return x+z;
}
```

x=0,z=0
x=1,z=0
x=2,z=0
x=8,z=0
x=3,z=0
x=2,z=0
z=0,$=0

메모리 줄이기 #2. 중간 결과 버리기

- 프로시저 분석이 끝나고 나서 중간 결과들에 대한 표를 저장하고 있을 필요가 없음



- 중간 결과를 버림으로써 분석 도중 메모리 피크를 낮춤
  - 고정점 계산시 순서를 조정하여야만 버리는 시점을 알 수 있음

메모리 절약 실험 결과

프로그램	줄	유지		버리기	
		속도 (s)	메모리 (MB)	속도(s)	메모리 (MB)
pci-driver.c	2,532	0.55	4.42	0.75	3.19
cdrom.c	6,218	107.91	357.58	91.45	84.79
t1394Diag.c	10,240	119.40	425.16	137.78	73.24
md.c	6,635	> 90 min	-	1,819.53	1,010.81
ll_rw_blk.c	5,469	> 90 min	-	947.20	511.43

추가 개선 필요

- 요약 도메인 최적화
  - 현재는 분리논리식을 그대로 구현
  - 단순한 요약 도메인으로 변경했을 때 속도 향상 기대
- 전 분석을 통한 경우 줄이기
  - 필드에 대한 생존 분석
  - 전역 생존 분석
- 전 변환을 통한 프로시저 늘리기
  - 반복문의 경우 해당 코드를 프로시저로 떼어 냈을 때 속도 향상

풀리지 않은 문제: 정확도

- 보다 복잡한 자료 구조를 어느 정도까지 정확하게 처리하는 것이 좋을까?
  - 모든 자료 구조 X
  - 리스트 (순환, 양방향, 중첩), 트리, 그래프?
  - 결국은 검증기를 사용할 사용자에게 입력을 받아 처리할 수 있는 능력을 갖추는 것이 최선일 듯
- 문맥 민감도
  - 문맥 민감도를 낮추면서 정확도가 비슷한 수준으로 가능?

풀리지 않은 문제: 성능

- 10,000 줄 프로그램에 대해 100 초대로 검증한 예제가 하나 나왔을 뿐
  - 100 초가 만족스러운지? 6,635 줄이 1,819초도 있음
  - 평균적으로 기하급수 복잡도가 안 되도록 하는 것이 가능?
- 이론적인 복잡도의 한계 vs 엔지니어링
  - 정확도를 유지하면서 고성능 가능?
  - 분석 정확도 희생 필요?

풀리지 않은 문제: 기타

- 지역성 최대화
  - 도달 가능(reachable)이 아닌 실제 프로시저에 영향을 미치는 부분만 떼어 내는 것이 가능?
  - 입출력 관계 분석?
  - 이득 vs 손실?