Translation Validation of Transformations of Embedded System Specifications using Equivalence Checking

Kunal Banerjee
Supervisors: Prof. C Mandal, Prof. D Sarkar

Dept of Computer Sc & Engg
IIT Kharagpur

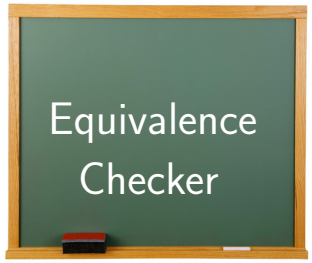
Motivation

gcc – Frequently Reported Bugs

There are many reasons why a reported bug doesn't get fixed. It might be difficult to fix, or fixing it might break compatibility. Often, reports get a low priority when there is a simple work-around. In particular, bugs caused by invalid code have a simple work-around: fix the code.
(source: `http://gcc.gnu.org/bugs/#known`)

Translation Validation

Specification



Implementation

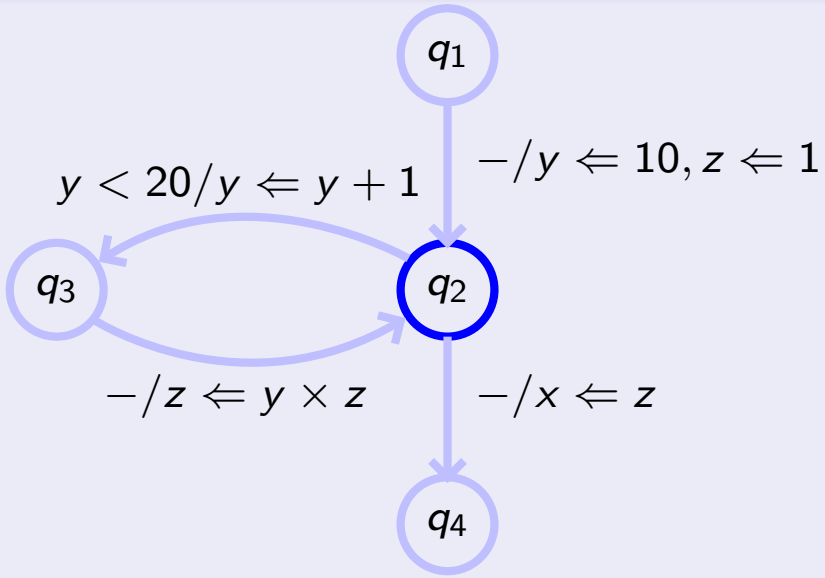
Compiler

Program as a combination of paths

A program can have *infinite* number of computations, a single computation can be *indefinitely long* — **cut loops**.

Representing a program using CDFG

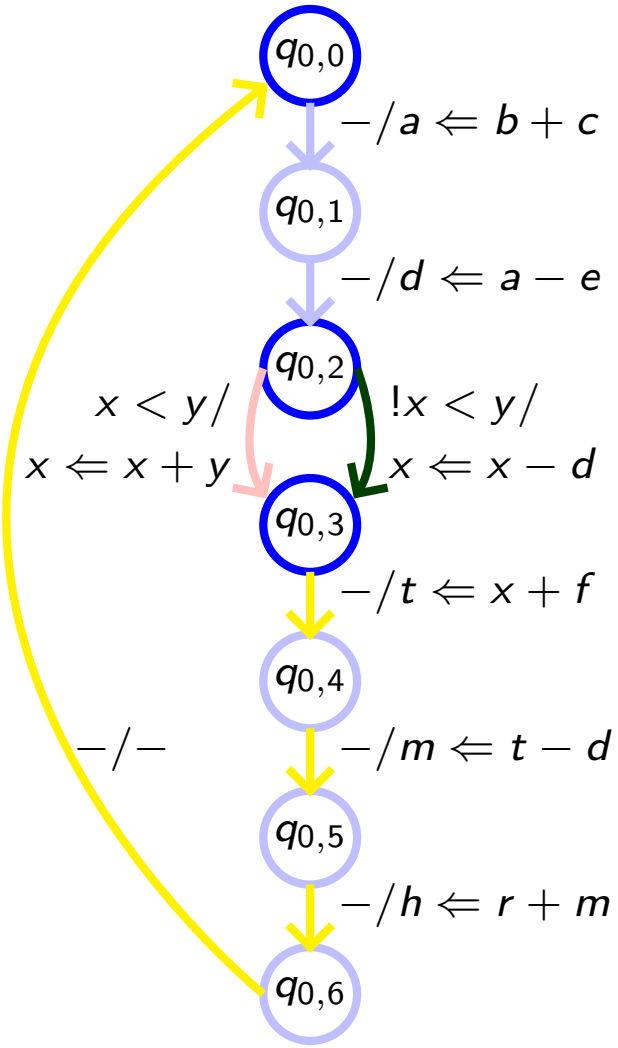
```
y := 10;
z := 1;
while ( y < 20 ) {
  y := y + 1;
  z := y × z;
}
x := z;
```



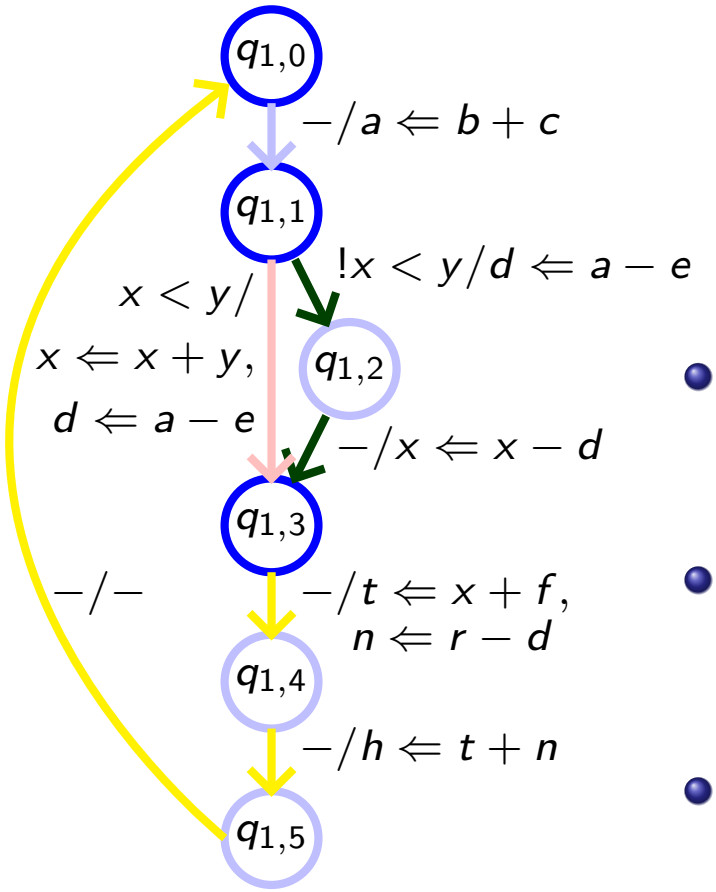
All computations of the program can be viewed as a concatenation of paths.

Example: $p_1.p_3$, $p_1.p_2.p_3$, $p_1.p_2.p_2.p_3$, $p_1.(p_2)^*.p_3$

Equivalence checking of FSMDs: A basic example



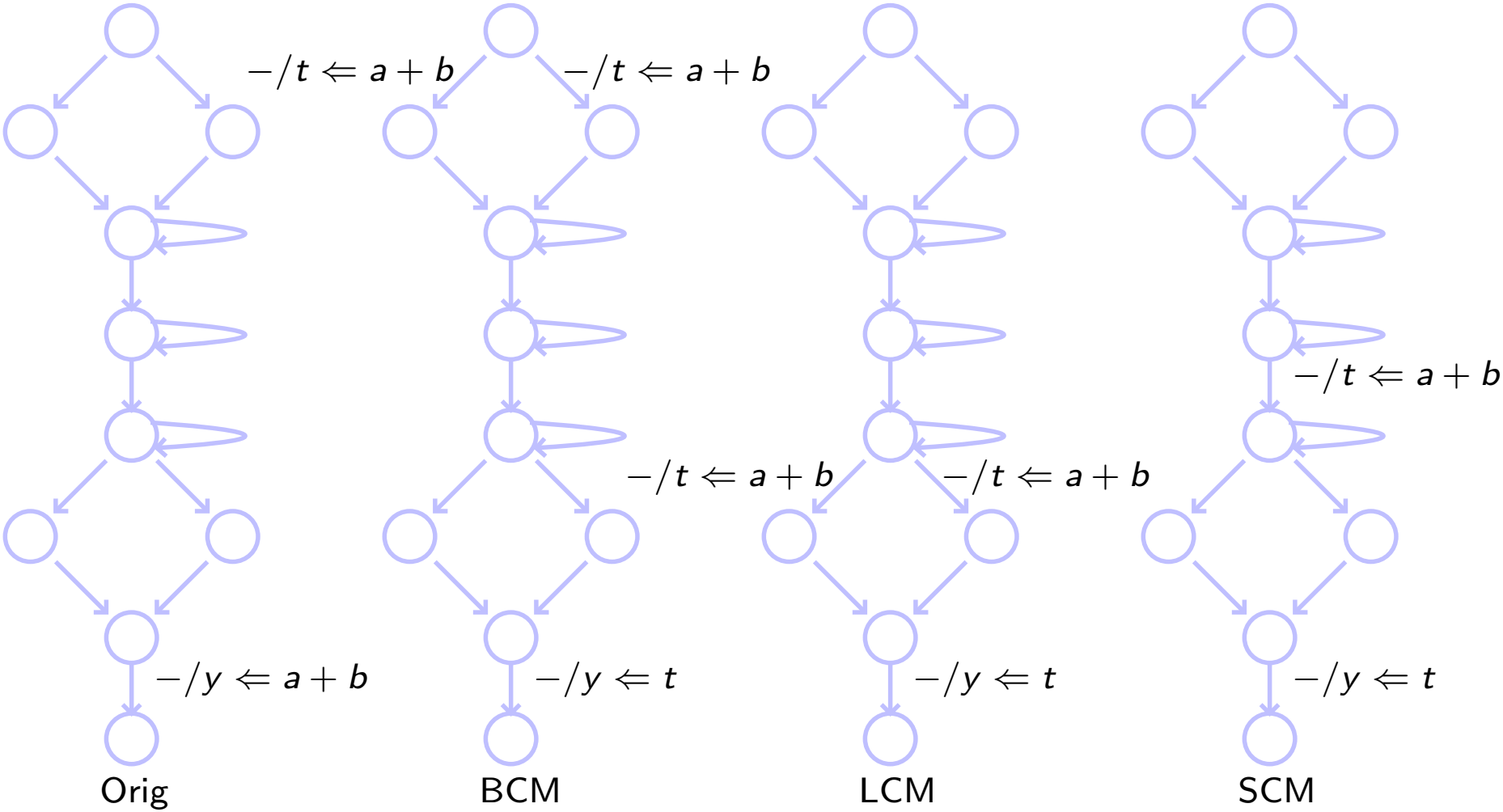
(a) M_0



(b) M_1

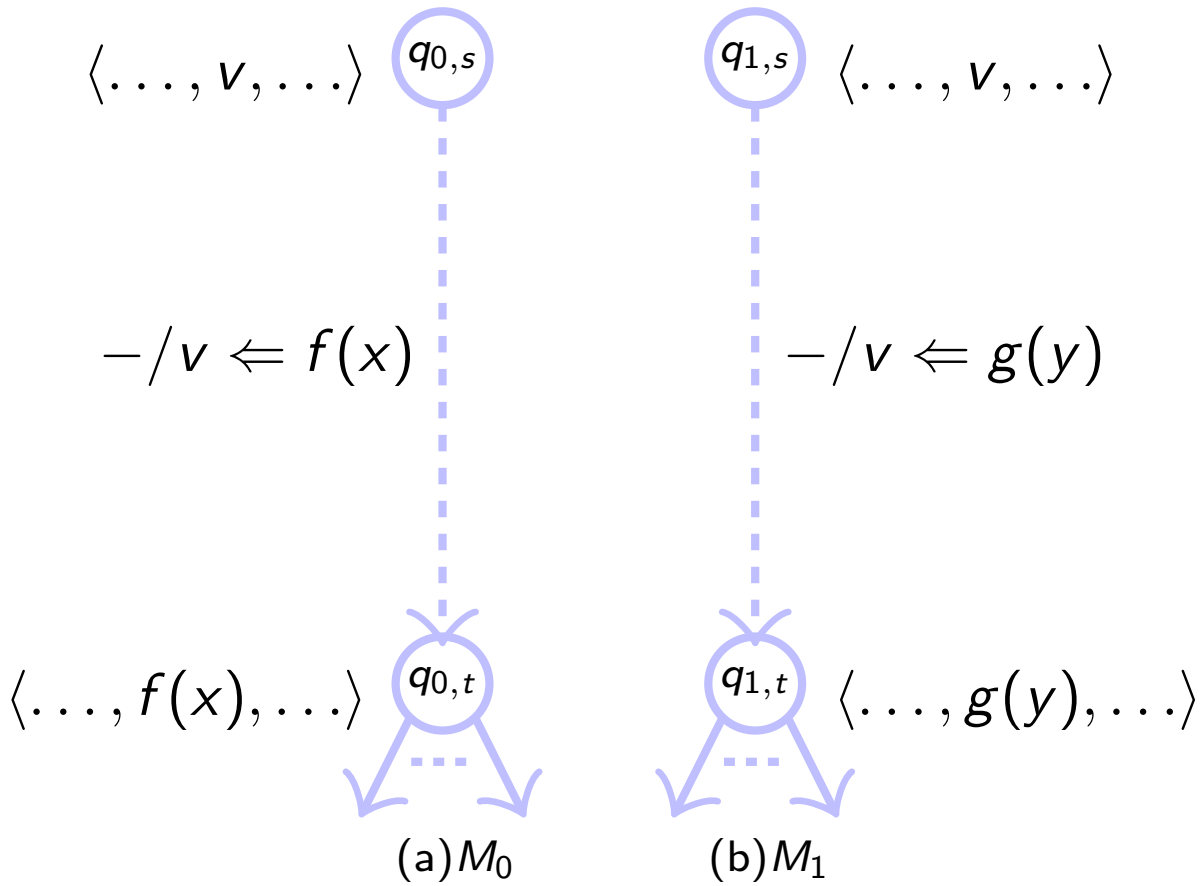
- Two Finite State Machines with Datapath (FSMDs) M_0 and M_1 are equivalent if for every path in P_0 there is an equivalent path in P_1 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{0,0} \overset{x<y}{\Longrightarrow} q_{0,3} \simeq q_{1,0} \overset{x<y}{\Longrightarrow} q_{1,3},\ q_{0,0} \overset{!x<y}{\Longrightarrow} q_{0,3} \simeq q_{1,0} \overset{!x<y}{\Longrightarrow} q_{1,3},\ q_{0,3} \Longrightarrow q_{0,0} \simeq q_{1,3} \Longrightarrow q_{1,0}\}$

A major challenge: Code motions across loops



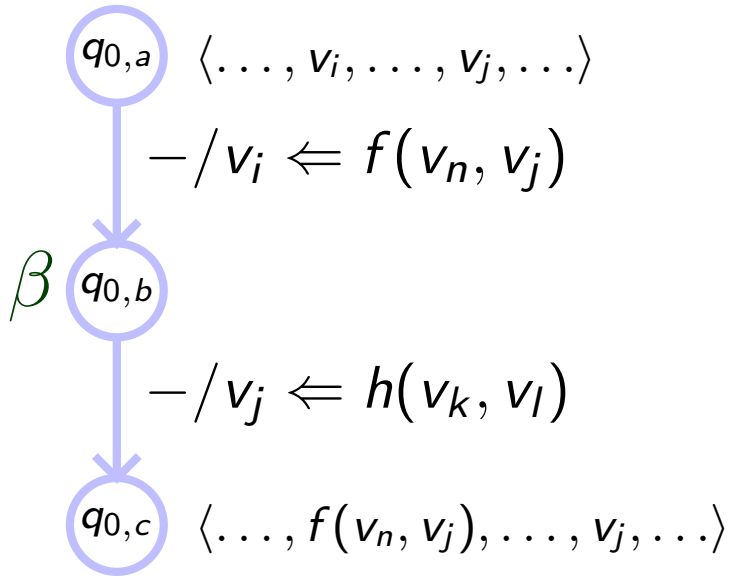
A path, by definition, cannot be extended beyond a loop.

The method of symbolic value propagation

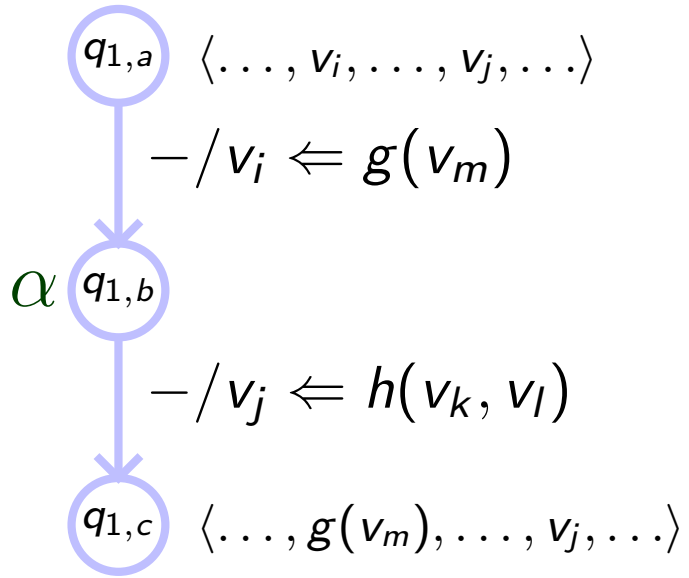


An example of value propagation

The method of symbolic value propagation



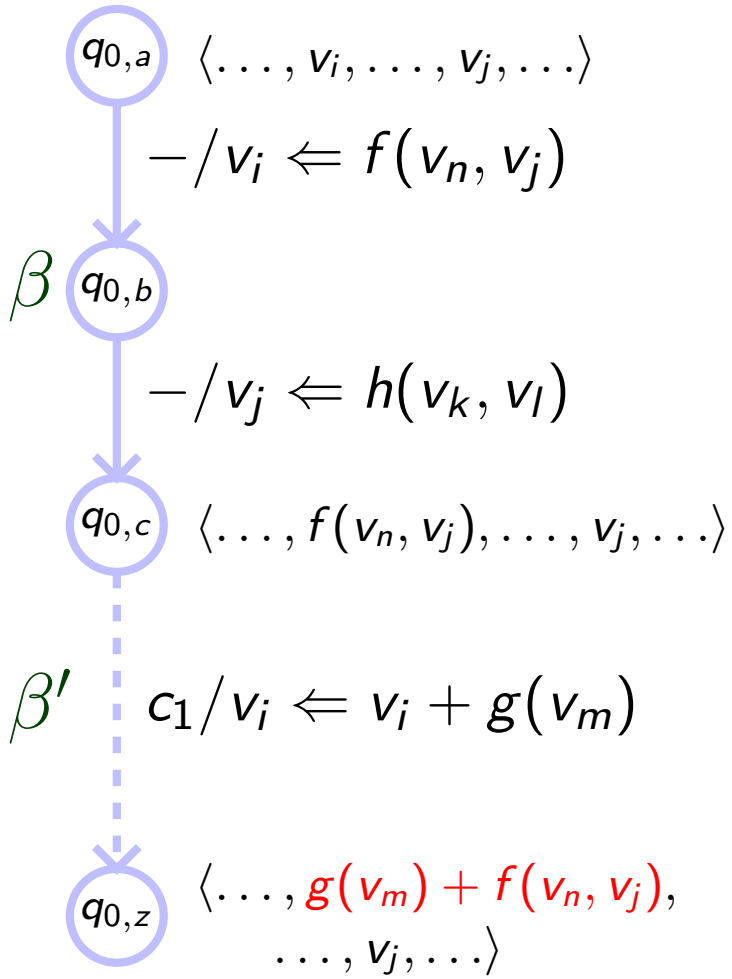
(a) M_0



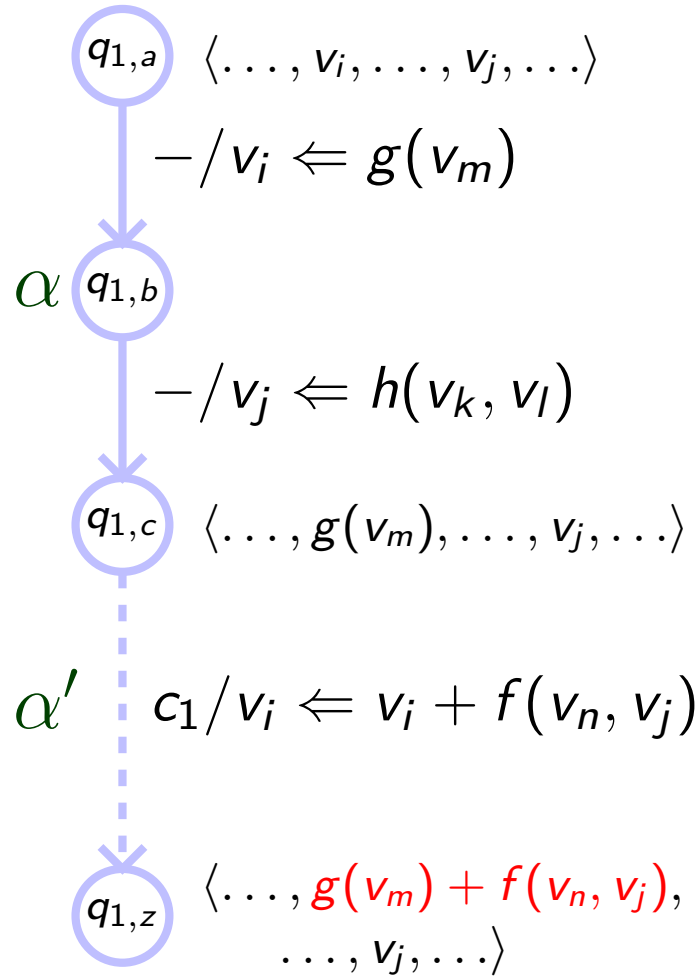
(b) M_1

An example of value propagation with dependency between propagated values

The method of symbolic value propagation



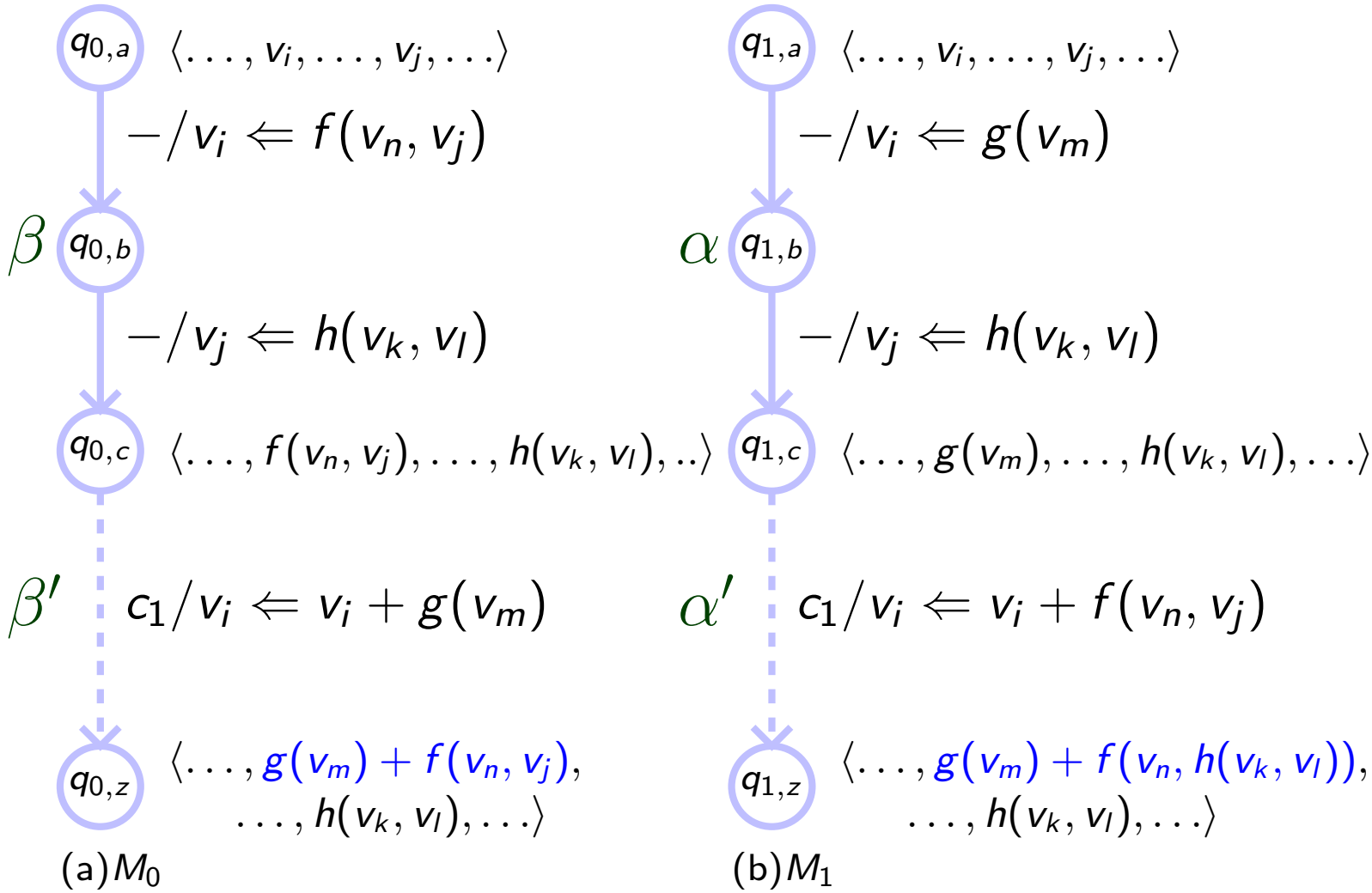
(a) M_0



(b) M_1

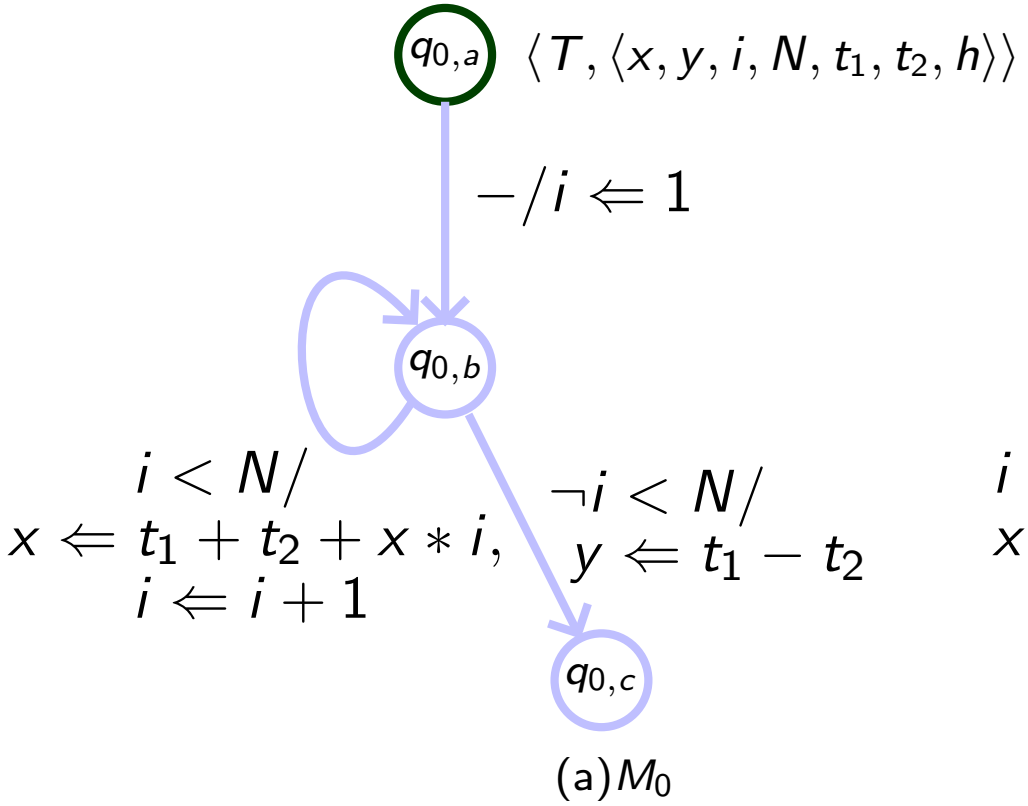
An *erroneous* decision taken

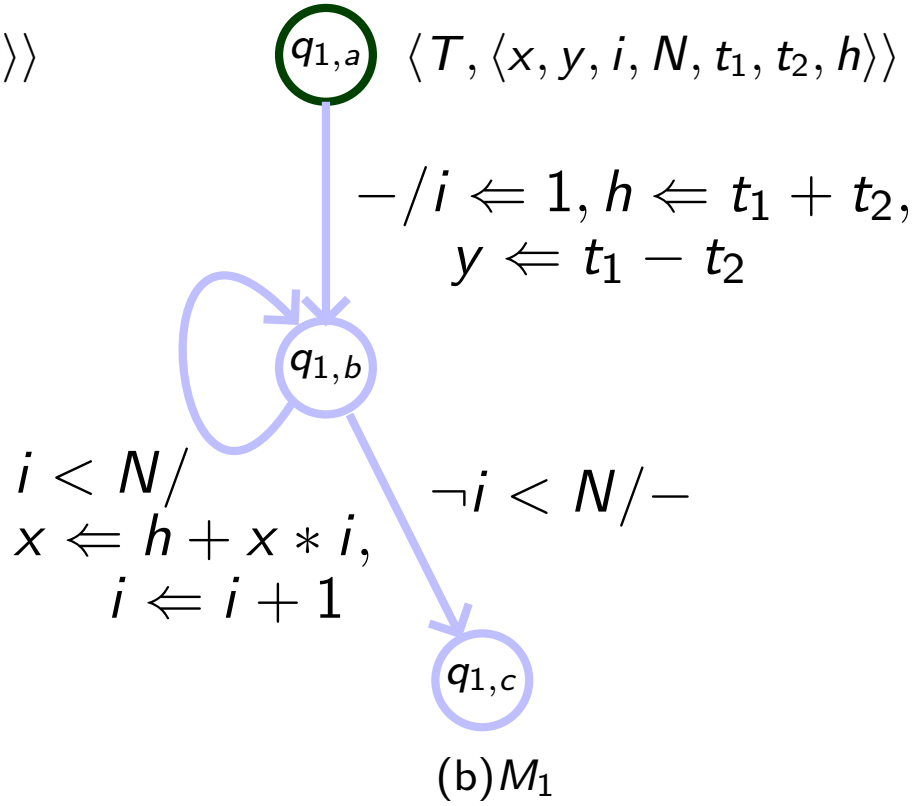
The method of symbolic value propagation



Correct decision taken

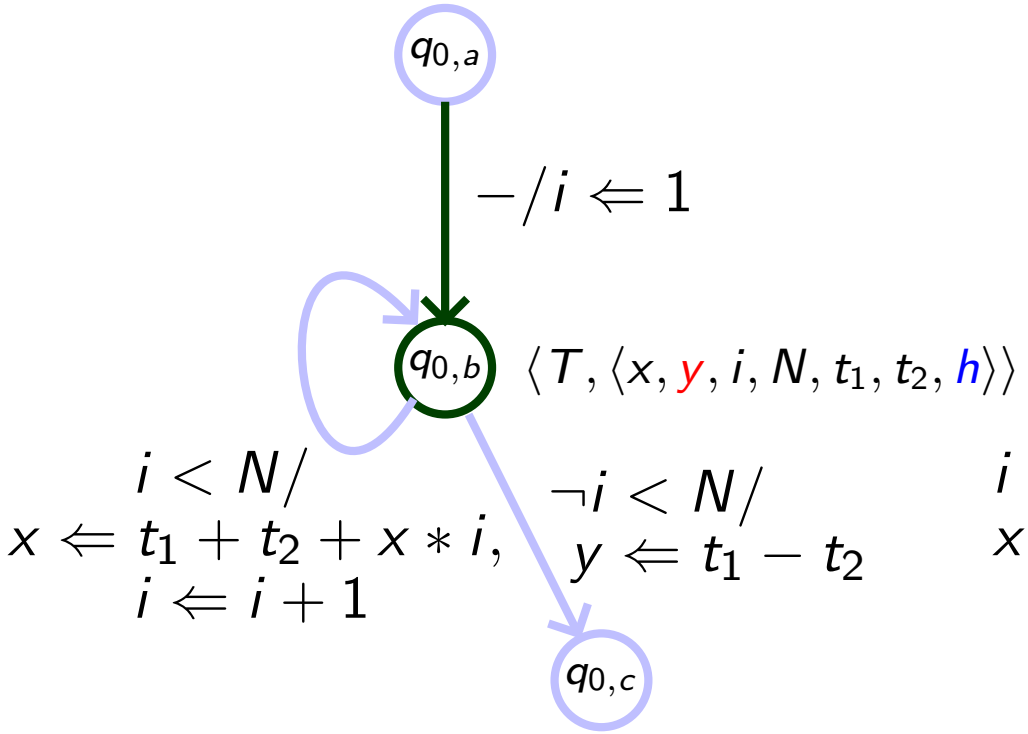
Equivalence checking using value propagation



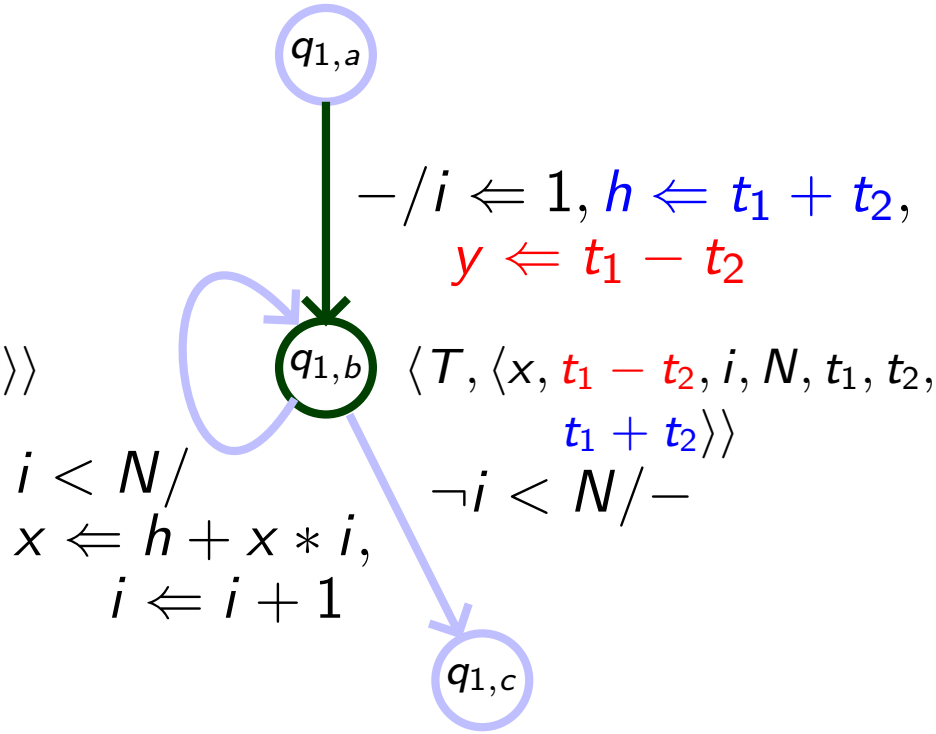


At the reset states

Equivalence checking using value propagation



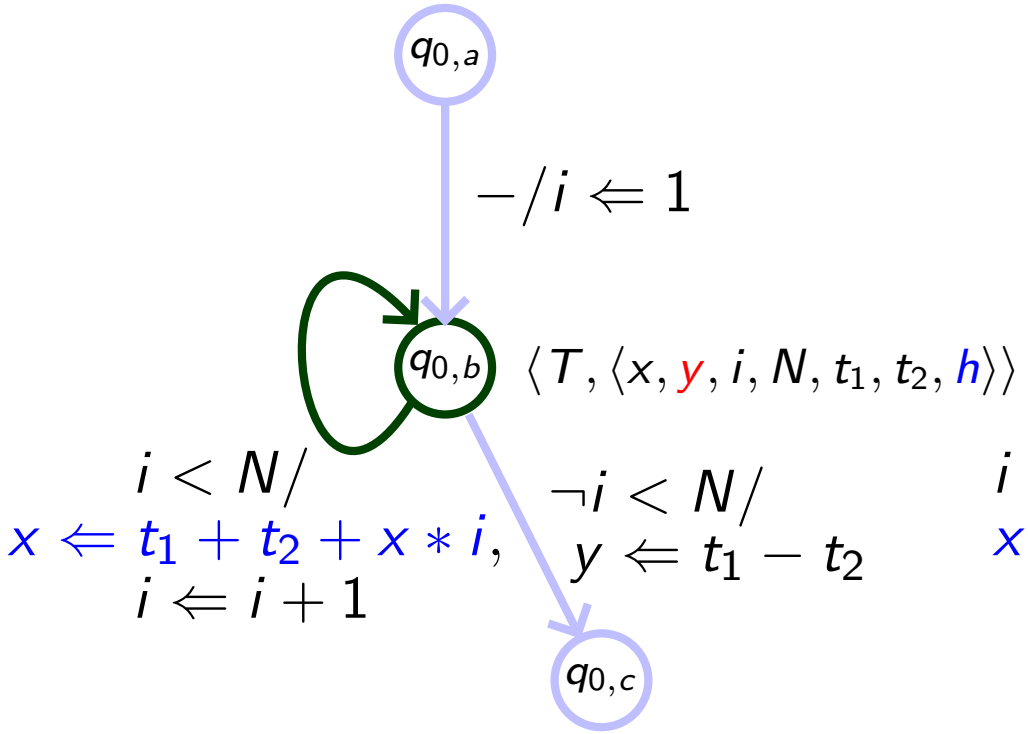
(a) M_0



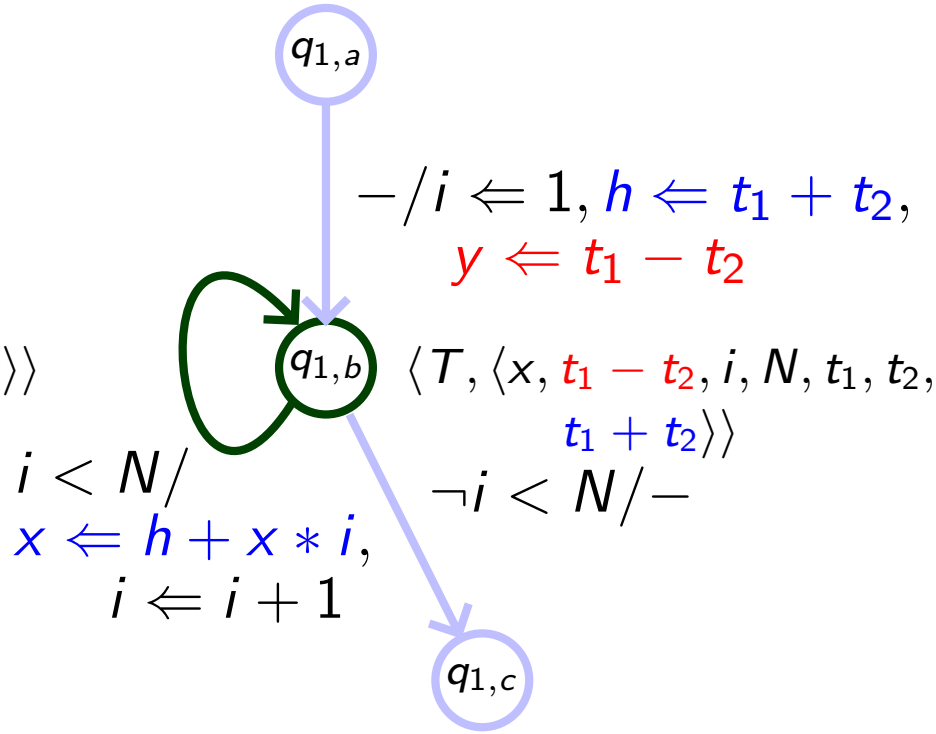
(b) M_1

At the beginning of the loops

Equivalence checking using value propagation



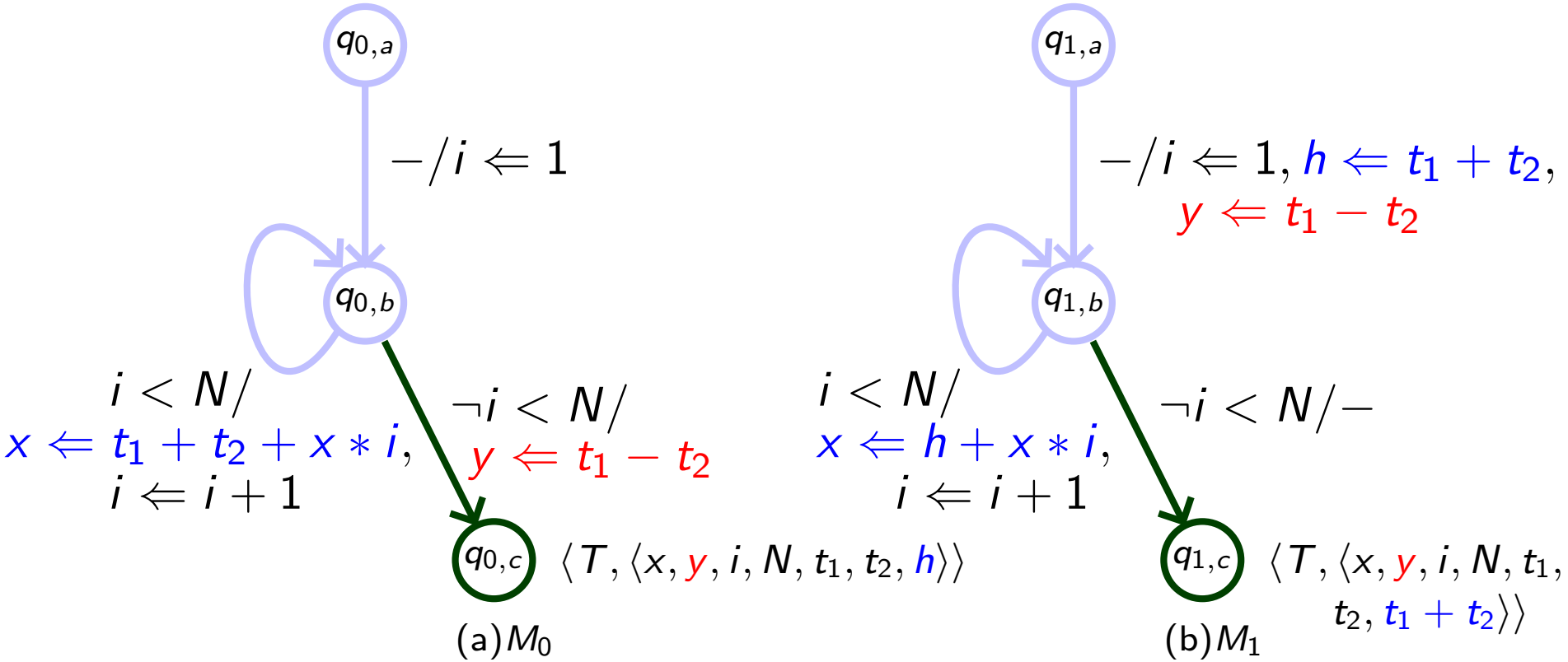
(a) M_0



(b) M_1

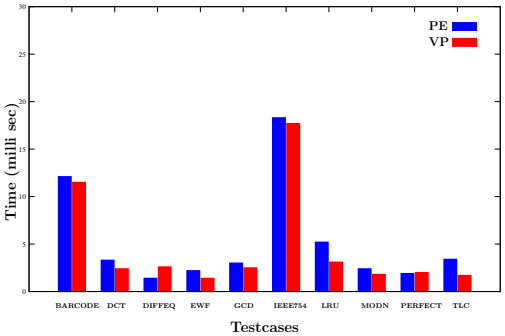
At the end of the loops

Equivalence checking using value propagation

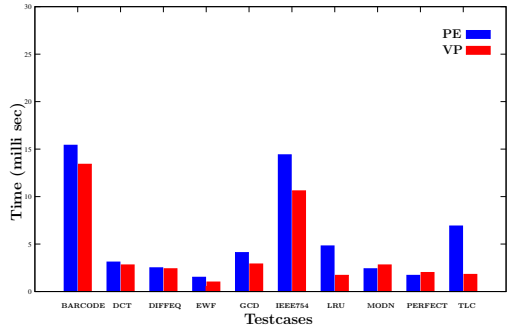


At the end states

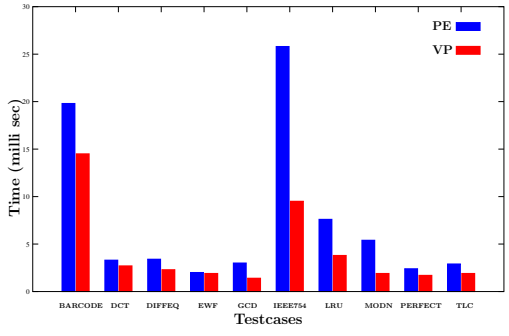
Experimental Results – 1



(a) BB-based



(b) Path-based



(c) SPARK

C. Mandal, and R. M. Zimmer, "A Genetic Algorithm for the Synthesis of Structured Data Paths," VLSI Design (2000)

R. Camposano, "Path-based Scheduling for Synthesis," TCAD (1991)

S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," VLSI Design (2003)

Experimental Results – 1 (contd.)

Benchmarks	Original FSMD		Transformed FSMD		#Variable		#across	Maximum	Time (ms)	
	#state	#path	#state	#path	com	uncom	loops	mismatch	PE	VP
BARCODE	33	54	25	56	17	0	0	3	20.1	16.2
DCT	16	1	8	1	41	6	0	6	6.3	3.6
DIFFEQ	15	3	9	3	19	3	0	4	5.0	2.6
EWF	34	1	26	1	40	1	0	1	4.2	3.6
LCM	8	11	4	8	7	2	1	4	–	2.5
IEEE754	55	59	44	50	32	3	4	3	–	17.7
LRU	33	39	32	38	19	0	2	2	–	4.0
MODN	8	9	8	9	10	2	0	3	5.6	2.5
PERFECT	6	7	4	6	8	2	2	2	–	0.9
QRS	53	35	24	35	25	15	3	19	–	15.9
TLC	13	20	7	16	13	1	0	2	9.1	4.1

★ K Banerjee et al., "A Value Propagation Based Equivalence Checking Method for Verification of Code Motion Techniques," ISED 2012.

★ K Banerjee et al., "Verification of Code Motion Techniques using Value Propagation," IEEE TCAD 2014.

Verifying Code Motions of Array-Handling Programs

The FSMD model does not provide formalism to capture arrays.
Steps taken to overcome this limitation:

- Proposed a new model, namely Finite State Machine with Datapath *having Arrays* (FSMDA), which allows representation of data computation involving arrays using McCarthy's access/change functions
- Improvised the normalization process to represent arithmetic expressions involving arrays in normalized forms
- Updated the previously mentioned equivalence checking method to accommodate extra rules for propagation of index and array variables

☞ Detected a bug in the implementation of copy propagation for array variables in the SPARK compiler.

★ K Banerjee et al., "Extending the FSMD Framework for Validating Code Motions of Array-Handling Programs," IEEE TCAD 2014.

Deriving Bisimulation Relations from Path Based Equivalence Checkers

	Bisimulation based verification		Path based equivalence checking
✓	Conventional	×	Unconventional
✓	Can handle loop shifting	×	Cannot handle loop shifting
×	Limited support for non-structure preserving transformations	✓	Adept in handling non-structure preserving transformations
×	Termination not guaranteed	✓	Termination guaranteed

★ K Banerjee et al., "Deriving Bisimulation Relations from Path Extension Based Equivalence Checkers," WEPL 2015.

A major challenge: Loop transformations for arrays

Loop and arithmetic transformations are used extensively to gain speed-ups (parallelization), save memory usage, reduce power, etc.

Loop Fusion

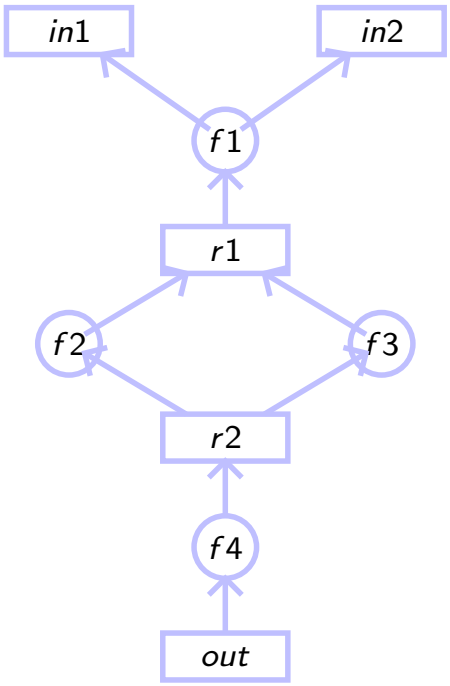
```
for (i=0; i<=7; i++) {
  for (j=0; j<=7; j++) {
    a[i+1][j+1] = F(in);
} }

for (i=0; i<=7; i++) {
  for (j=0; j<=7; j++) {
    b[i][j] = c[i][j];
} }
```

```
for (l1=0; l1<=3; l1++) {
  for (l2=0; l2<=3; l2++) {
    for (l3=0; l3<=1; l3++) {
      for (l4=0; l4<=1; l4++) {
        i = 2*l1 + l3;
        j = 2*l2 + l4;
        a[i+1][j+1] = F(in);
        b[i][j] = c[i][j];
} } } }
```

For array operations, **equivalence of data transformations and index spaces** have to be ensured.

Array Data Dependence Graphs (ADDGs)



ADDG

- Array data dependence graph (ADDG) model can capture array intensive programs [Shashidhar et al., DATE 2005]
- ADDGs have been used to verify static affine programs
- Equivalence checking of ADDGs can verify loop transformations as well as arithmetic transformations

Two equivalent array-handling programs

Loop fusion and arithmetic simplification

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```

```
for ( i = 1; i <= N; i++ ) {
  z[i] = 2 * a[i];
}
```

```
for ( i = 1; i <= 100; i++ ) { out[i-1] = in[i+1]; }
```

Jargons:

Iteration domain: Domain of the index variable. $\{i \mid 1 \leq i \leq 100\}$

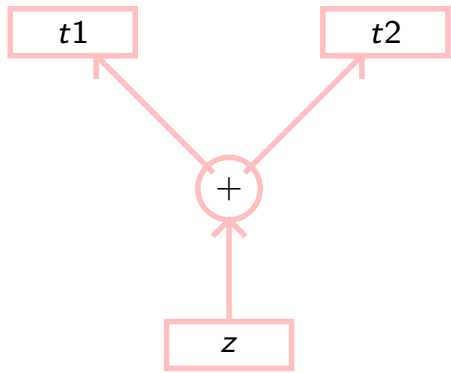
Definition domain: Domain of the (lhs) variable getting defined. $\{i \mid 0 \leq i \leq 99\}$

Operand domain: Domain of the operand variable. $\{i \mid 2 \leq i \leq 101\}$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

$${}_{I}M_{z} = \{k \rightarrow k+1 \mid 0 \leq k \leq N-1\} = {}_{I}M_{t1} = {}_{I}M_{t2}$$

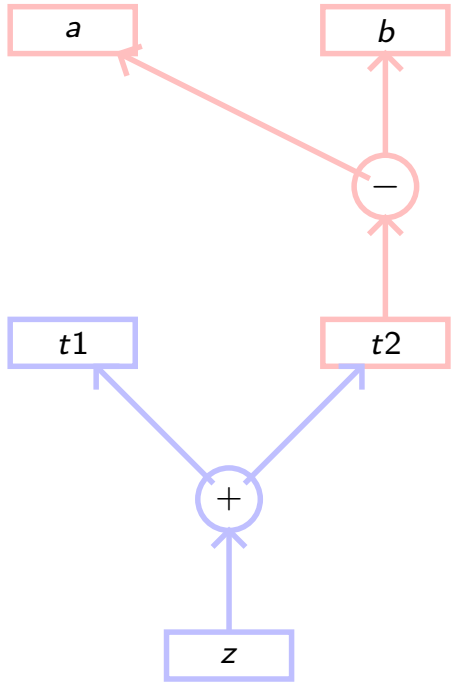
$${}_{z}M_{t1} = {}_{I}M_{z}{}^{-1} \diamond {}_{I}M_{t1} = \{k \rightarrow k \mid 1 \leq k \leq N\} = {}_{z}M_{t2}$$

$$r_{\alpha} : z = t1 + t2$$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

$_{t2}M_a = \{j \rightarrow j \mid 1 \leq j \leq N\} = {}_{t2}M_b$

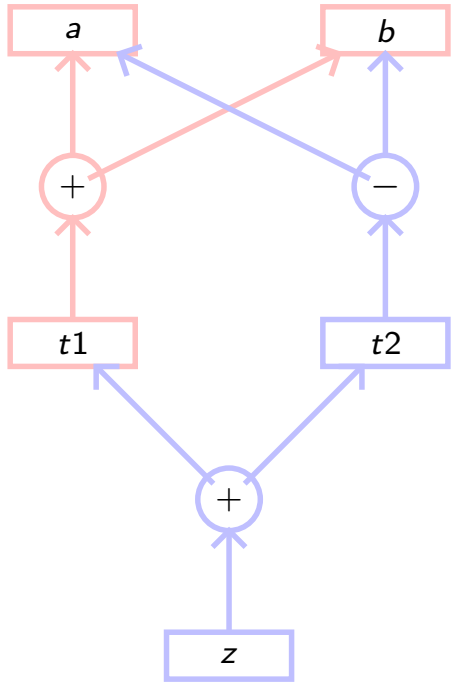
$_{z}M_{t1} = \{k \rightarrow k \mid 1 \leq k \leq N\}$ $\quad {}_{z}M_a = \{j \rightarrow j \mid 1 \leq j \leq N\} = {}_{z}M_b$

$r_\alpha : z = t1 + (a - b)$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i]
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j]
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

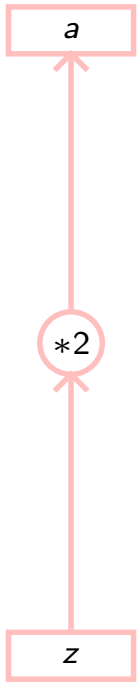
$_{t1}M_a = \{i \rightarrow i \mid 1 \leq i \leq N\} = {}_{t1}M_b$

$_{z}M_a = \{k \rightarrow k \mid 1 \leq k \leq N\} = {}_{z}M_b$

$r_\alpha : z = (a + b) + (a - b) = 2 * a$ – simplification possible since domains match

Construction of ADDG-2

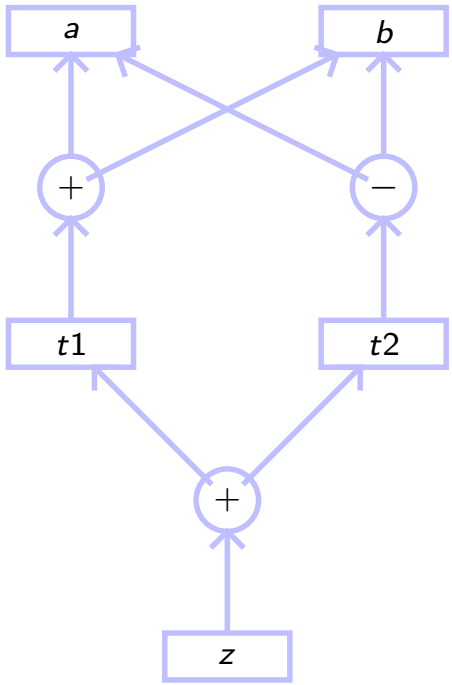
```
for ( i = 1; i <= N; i++ ) {
  z[i] = 2 * a[i];
}
```



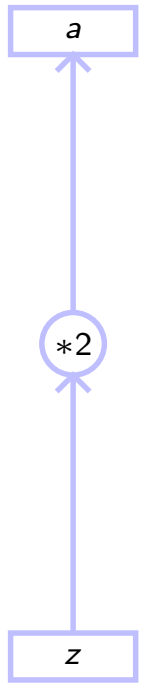
ADDG-2

$$_{I}M_{z} = \{i \rightarrow i \mid 1 \leq i \leq N\} = {}_{I}M_{a}$$
$$_{z}M_{a} = \{i \rightarrow i \mid 1 \leq i \leq N\}$$
$$r_{\beta} : z = 2 * a$$

Equivalence of ADDGs



ADDG-1



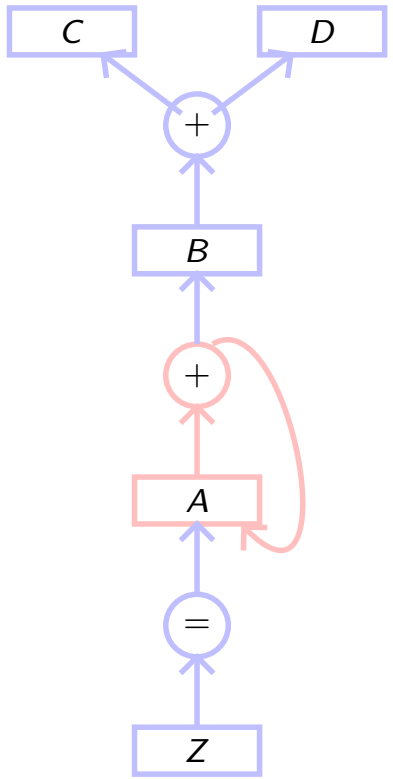
ADDG-2

Two ADDGs are said to be **equivalent** if their characteristic formulae – r_α and r_β, and corresponding mappings between the output arrays wrt input array(s) – ${}_zM_a^\alpha$ and ${}_zM_a^\beta$, match.
Hence, these two ADDGs are declared equivalent.

★ ISVLSI 2011, I-CARE 2013 (Best Paper Award), IEEE TCAD 2013.

Handling recurrences

```
for ( i = 1; i < N; i++ ) {
  B[i] = C[i] + D[i];
}
for ( i = 1; i < N; i++ ) {
  A[i] = A[i-1] + B[i];
}
for ( i = 1; i < N; i++ ) {
  Z[i] = A[i];
}
```

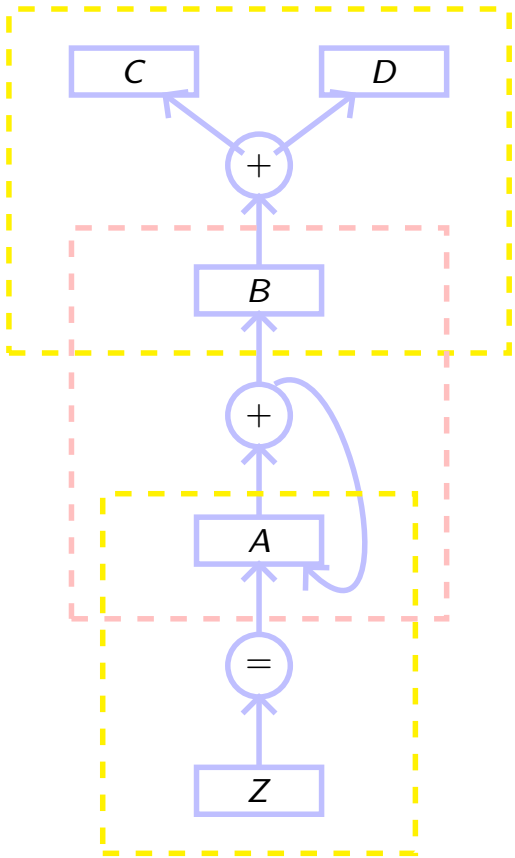


ADDG

Presence of recurrences leads to cycles in the ADDG and hence a closed form representation of r_α cannot be obtained.

Remedy – Separate DAGs from cycles

```
for ( i = 1; i < N; i++ ) {
  B[i] = C[i] + D[i];
}
for ( i = 1; i < N; i++ ) {
  A[i] = A[i-1] + B[i];
}
for ( i = 1; i < N; i++ ) {
  Z[i] = A[i];
}
```



ADDG

Try to establish equivalence of the *separated* ADDG portions.

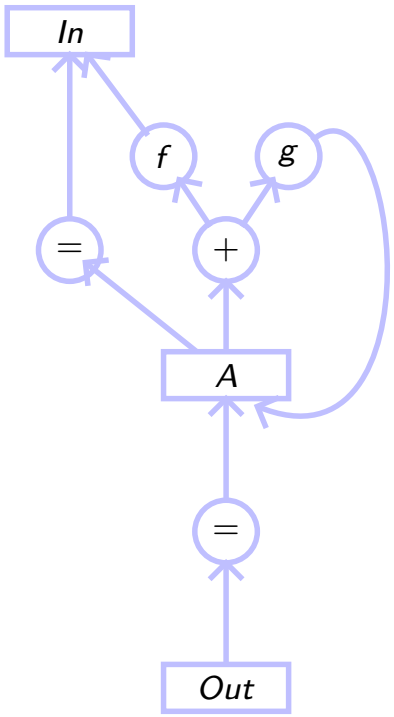
An illustrative example

A pair of programs involving recurrences

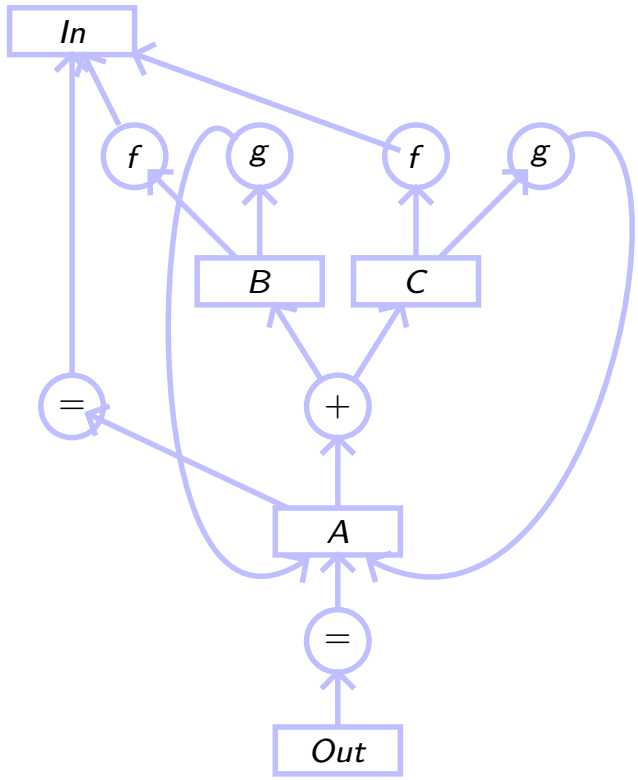
```
S1: A[0] = In[0];
for (i = 1; i < N; ++i) {
  S2: A[i] = f(In[i]) + g(A[i-1]);
}
S3: Out = A[N-1];
```

```
S1: A[0] = In[0];
for (i = 1; i < N; ++i) {
  if ( i%2 == 0 ) {
    S2: B[i] = f(In[i]);
    S3: C[i] = g(A[i-1]);
  } else {
    S4: B[i] = g(A[i-1]);
    S5: C[i] = f(In[i]);
  }
  S6: A[i] = B[i] + C[i];
}
S7: Out = A[N-1];
```

ADDGs with cycles

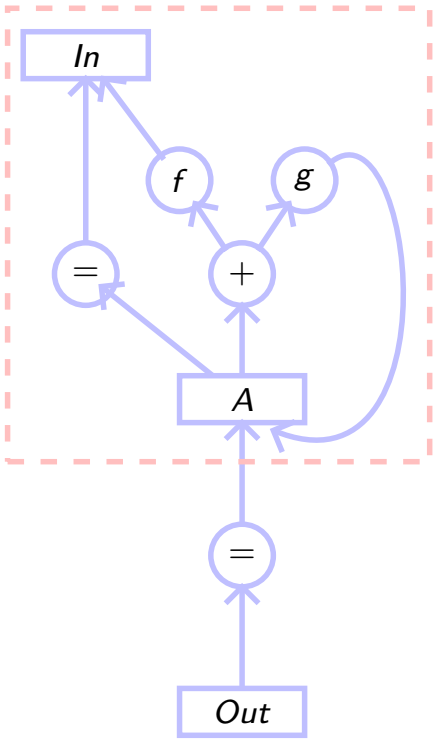


ADDG-1

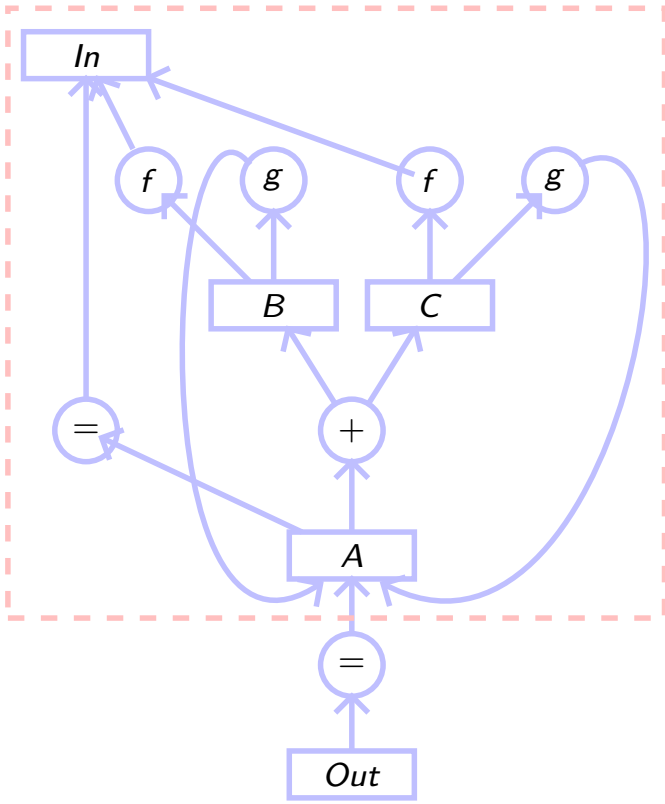


ADDG-2

ADDGs with cycles (contd.)

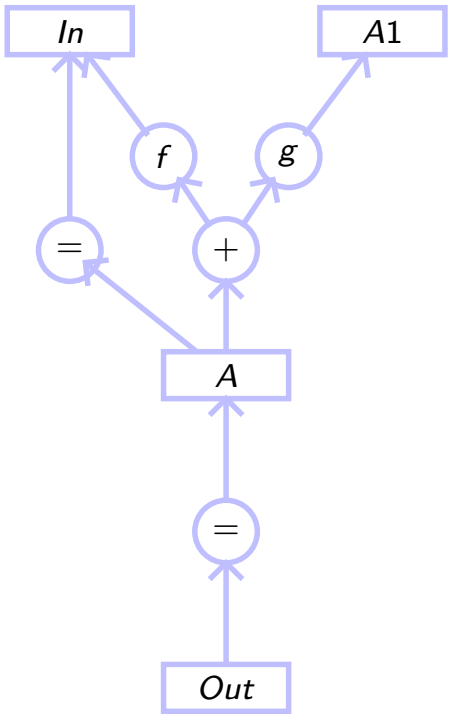


ADDG-1

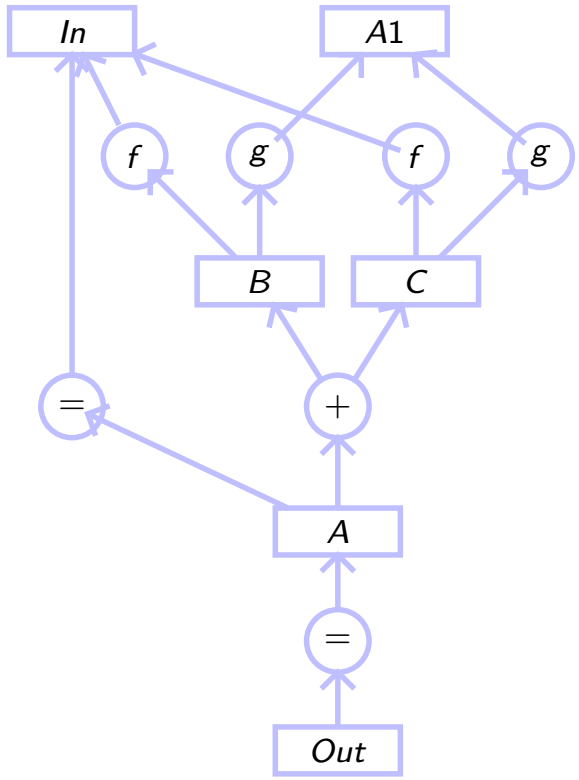


ADDG-2

ADDGs without cycles

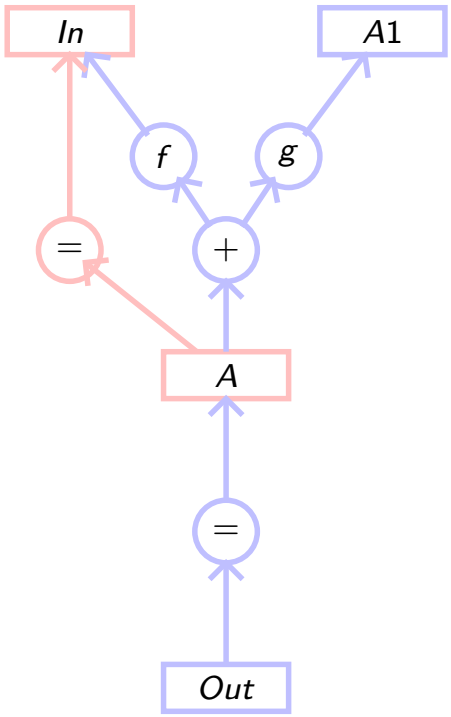


ADDG-1



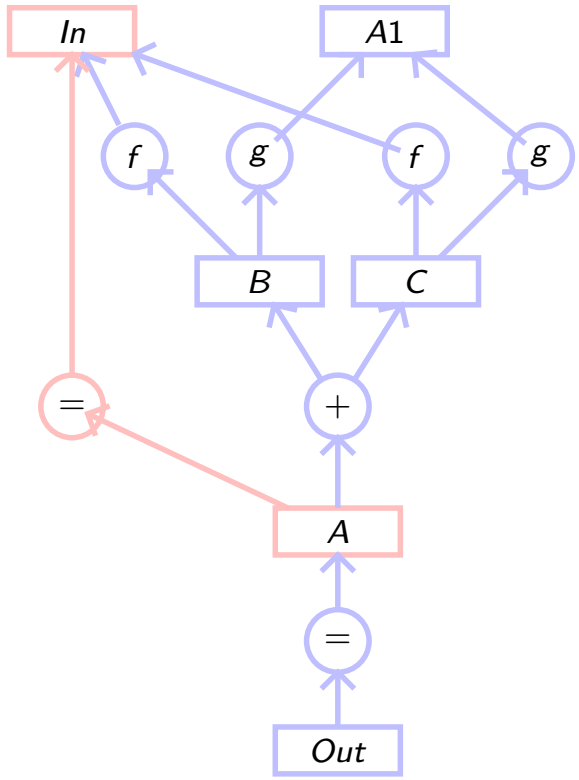
ADDG-2

ADDGs without cycles (contd.)



ADDG-1

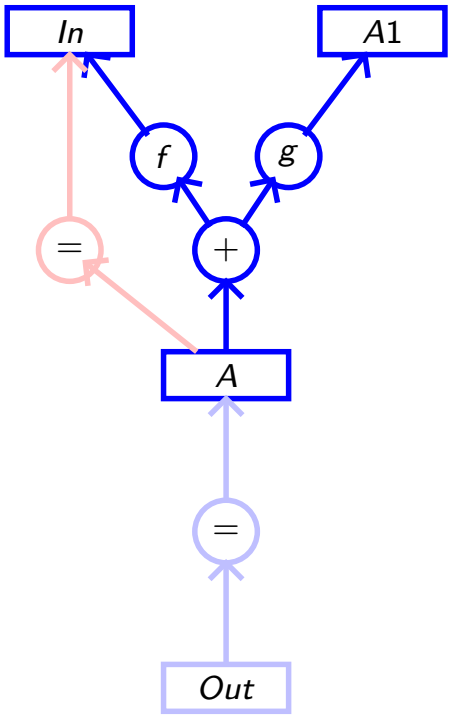
```
S1:  A[0] = In[0];
```



ADDG-2

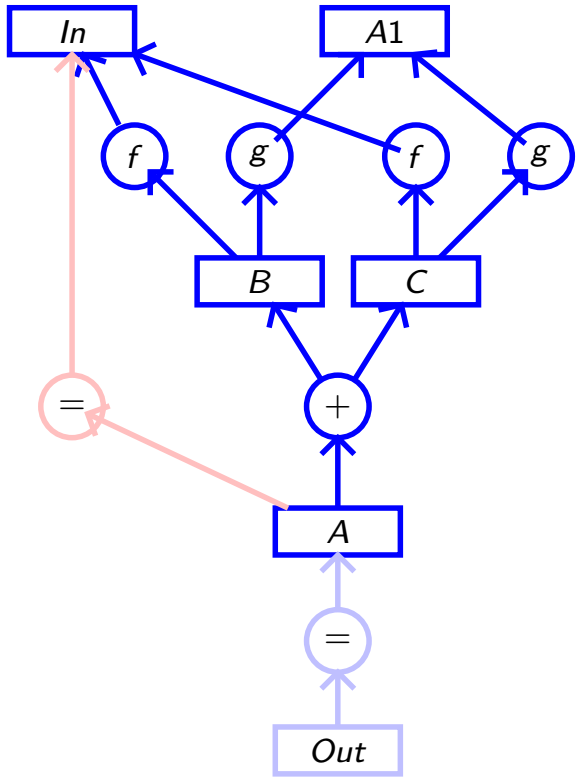
```
S1:  A[0] = In[0];
```

ADDGs without cycles (contd.)



ADDG-1

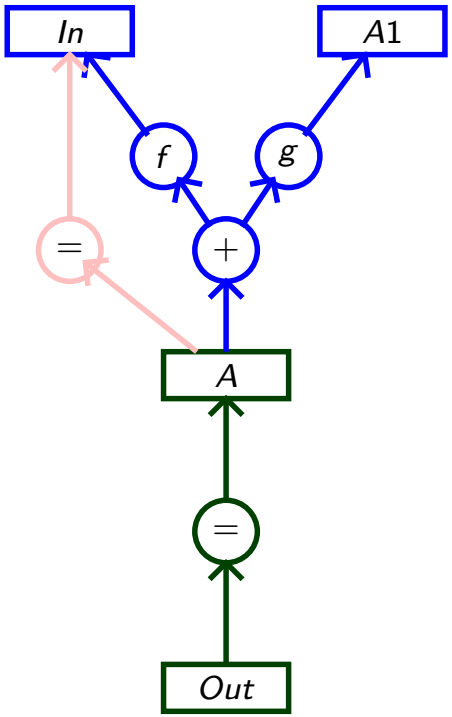
```
for (i = 1; i < N; ++i)
  A[i] = f(In[i]) + g(A[i-1]);
```



ADDG-2

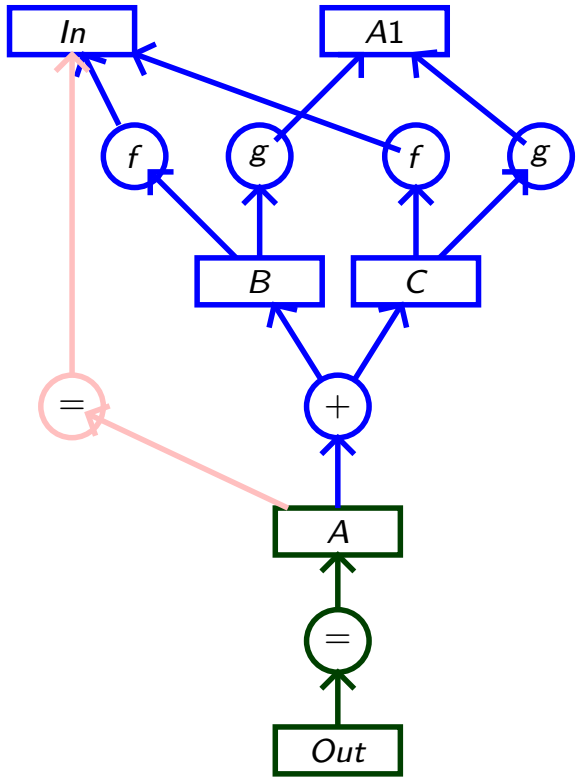
```
for (i = 1; i < N; ++i) {
  if ( i%2 == 0 ) { B[i] = f(In[i]); C[i] = g(A[i-1]); }
  else { B[i] = g(A[i-1]); C[i] = f(In[i]); }
  A[i] = B[i] + C[i]; }
```

ADDGs without cycles (contd.)



ADDG-1

```
S3:   Out = A[N-1];
```



ADDG-2

```
S7:   Out = A[N-1];
```

Experimental Results – 2

Sl No	Benchmark	C lines		loops		arrays		slices		Exec time (sec) [TOPLAS]	Exec time (sec) [TCAD]	Exec time (sec) [Our]
		src	trans	src	trans	src	trans	src	trans			
1	ACR1	14	20	1	3	6	6	1	1	0.18	0.76	0.55
2	LAP3	12	28	1	4	2	4	1	2	0.28	9.25	4.40
3	LIN1	13	13	3	3	4	4	2	2	0.12	0.62	0.52
4	LIN2	13	16	3	4	4	4	2	3	0.13	0.74	0.41
5	SOR	26	22	8	6	11	11	1	1	0.18	1.08	0.86
6	WAVE	17	17	1	2	2	2	4	4	0.31	6.83	3.82
7	ACR2	24	14	4	1	6	6	2	1	×	0.98	0.47
8	LAP1	12	21	1	3	2	4	1	1	×	2.79	1.06
9	LAP2	12	14	1	1	2	2	1	2	×	4.82	1.67
10	LOWP	13	28	2	8	2	4	1	2	×	9.17	3.90
11	SOB1	27	19	3	1	4	4	1	1	×	1.79	0.85
12	SOB2	27	27	3	3	4	4	1	1	×	1.85	1.08
13	EXM1	8	15	1	1	3	5	2	4	0.16	×	2.60
14	EXM2	8	13	1	1	3	5	2	3	0.12	×	2.12
15	SUM1	13	18	3	3	5	6	2	4	×	×	2.56
16	SUM2	13	20	3	4	5	7	2	4	×	×	2.68

★ K Banerjee, "An Equivalence Checking Mechanism for Handling Recurrences in Array-Intensive Programs," POPL-SRC 2015.

Thank you!

☞ http://cse.iitkgp.ac.in/~kunban/
✉ kunalb@cse.iitkgp.ernet.in