Putting Numerical Abstract Domains to Work: A Study of Array-Bound Checking for C Programs

Arnaud J. Venet
Carnegie Mellon University
NASA Ames Research Center
arnaud.j.venet@nasa.gov

This work was partially supported by the NSF Grant #1136008 and the SSAT project under the Aviation Safety program of NASA'S Aeronautics Research Mission Directorate.

Abstract Interpretation

- A theory of sound semantic approximation introduced by Patrick & Radhia Cousot in the mid 70's
- First application to the computation of variable ranges (1976)
- Verification of the numerical algorithms in the A380 flight software (2005)
- Numerical abstract interpretation is an active field of research

Roadmap

- The domain of convex polyhedra
- Application to array-bound checking:
  - The buffer library of OpenSSH (700 LOC)
  - The flight software of Mars Exploration Rovers (550 KLOC)
- Improving scalability: the gauge domain

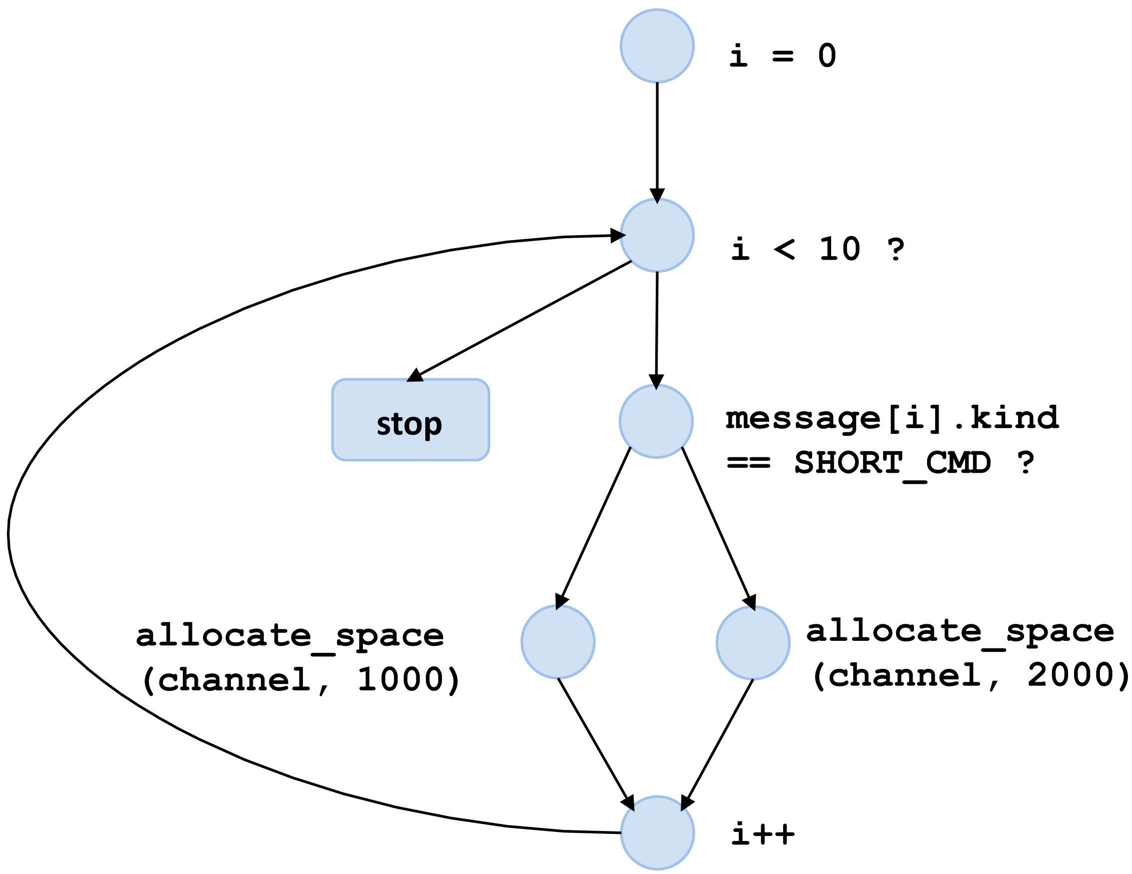
The domain of convex polyhedra

A simple example

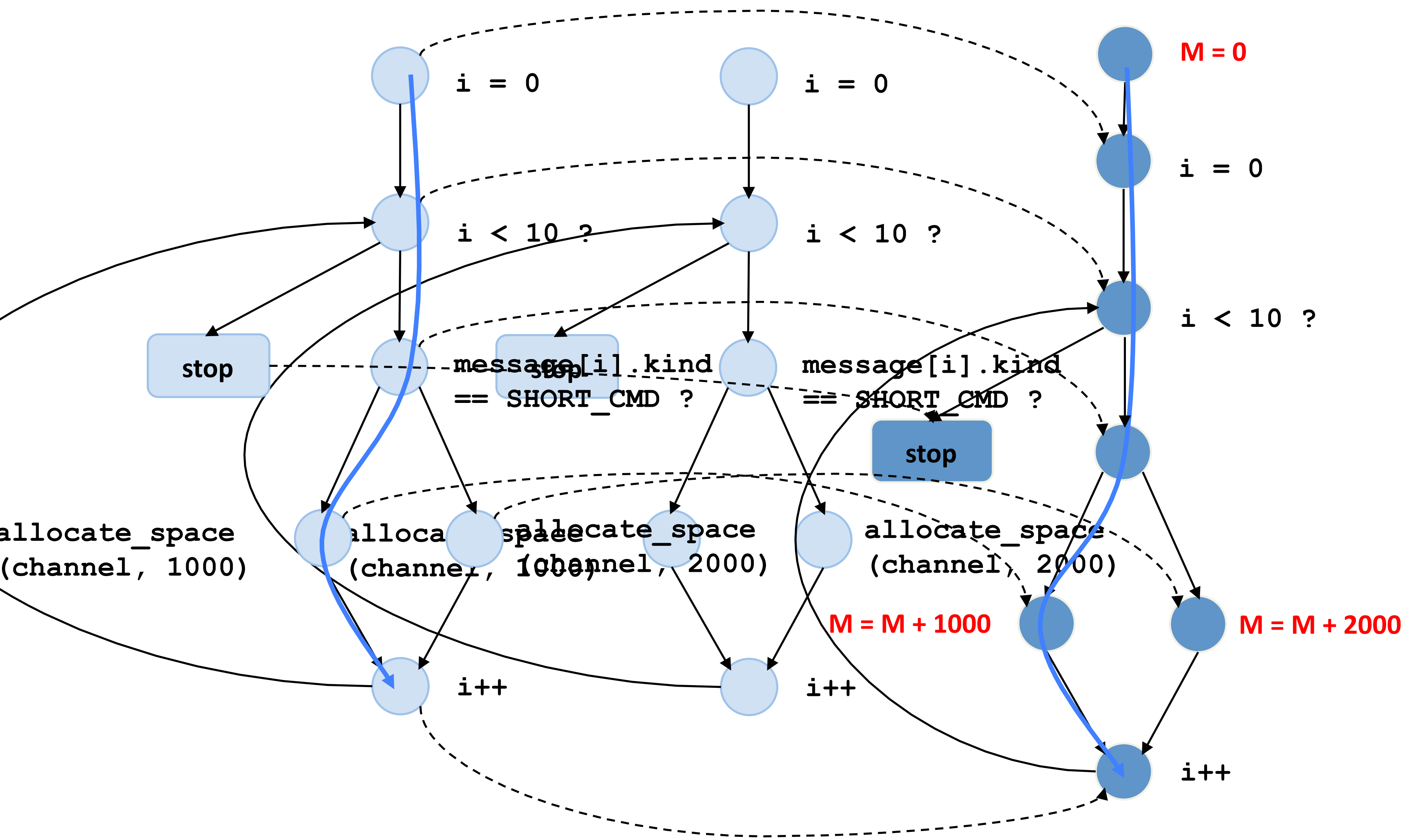
```
for(i = 0; i < 10; i++) {
  if(message[i].kind == SHORT_DATA)
    allocate_space (channel, 1000);
  else
    allocate_space (channel, 2000);
}
```

What are the memory requirements?

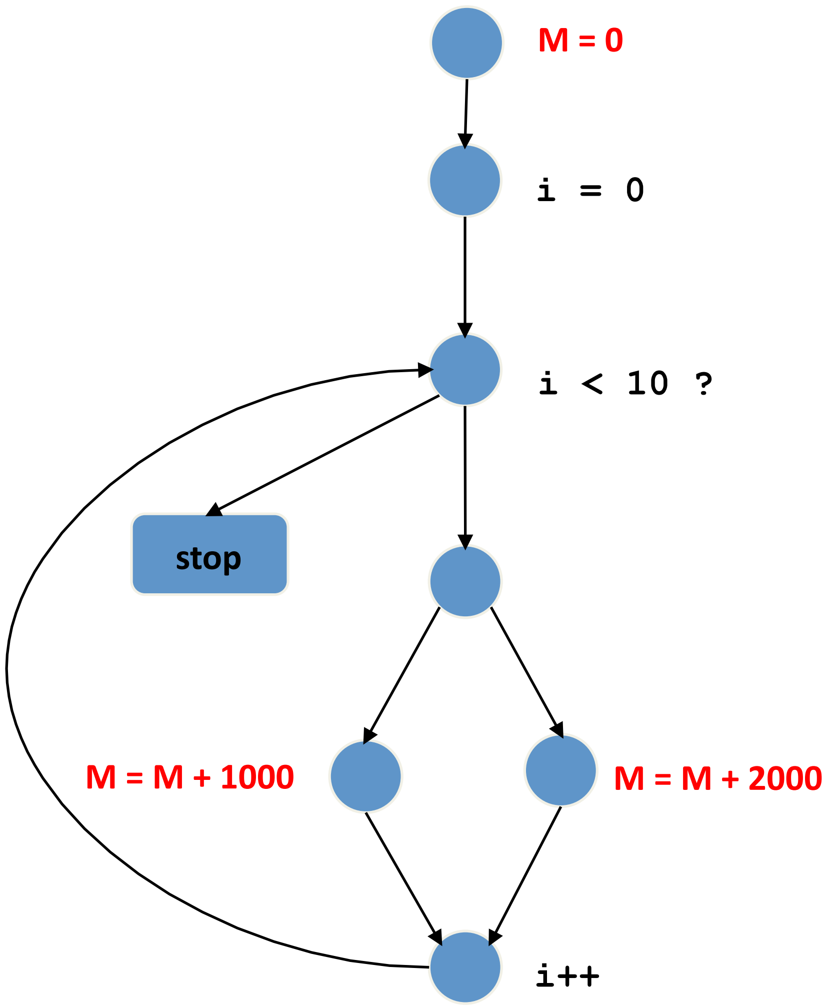
Control flow graph

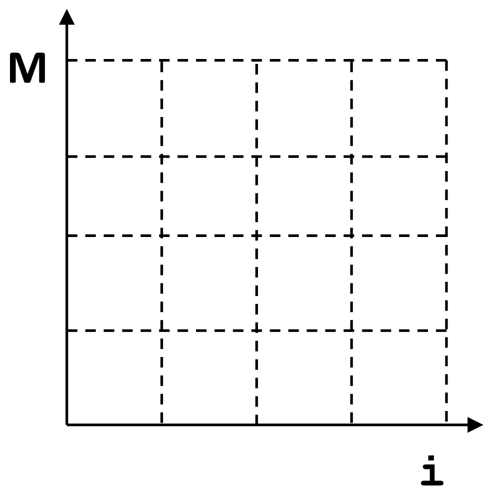


Abstract model of the code

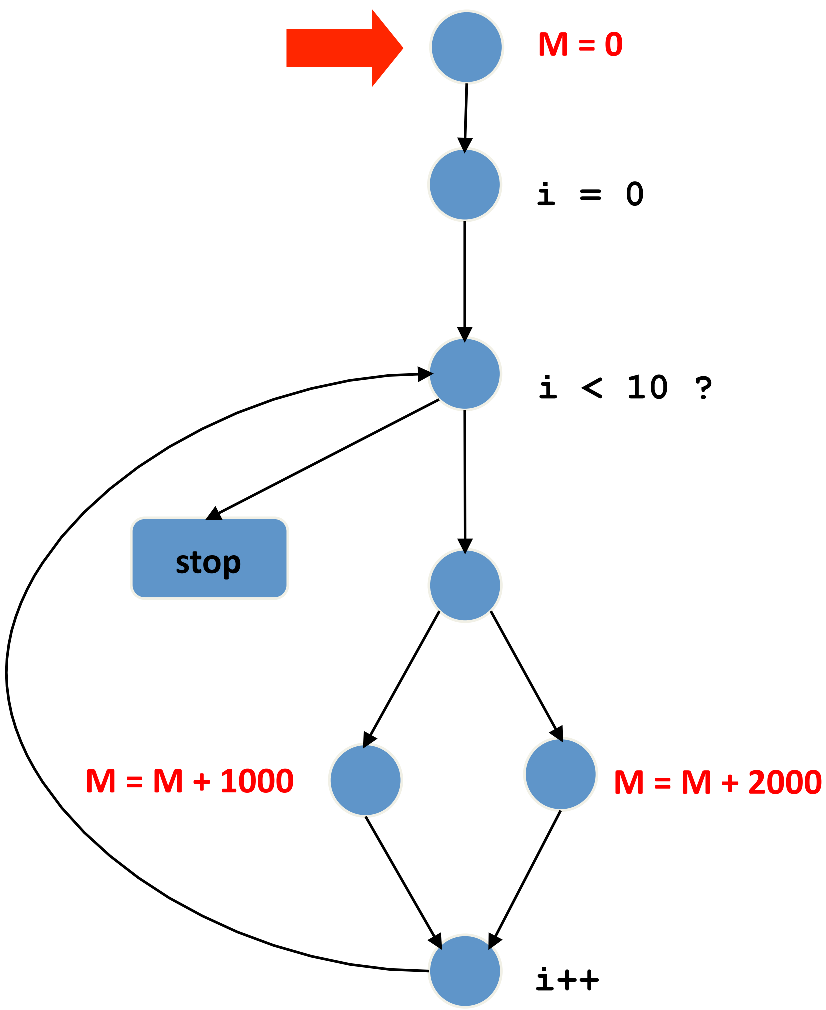


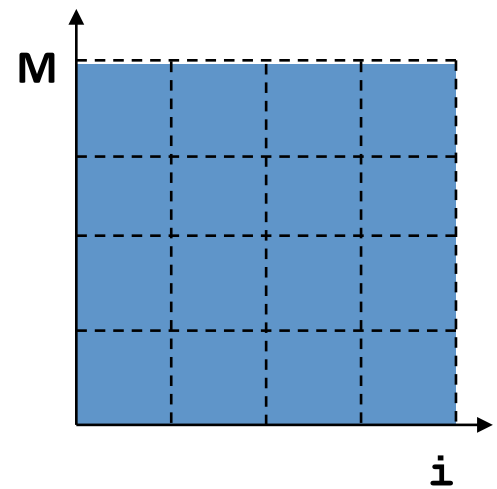
Analyzing the model



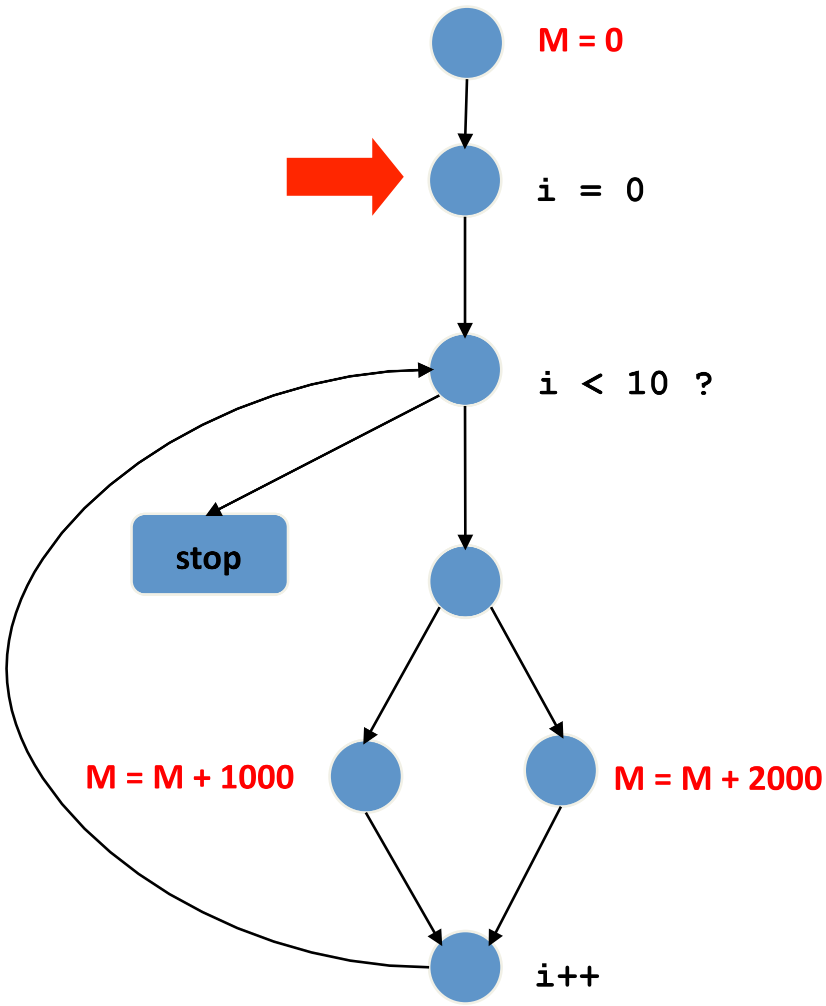


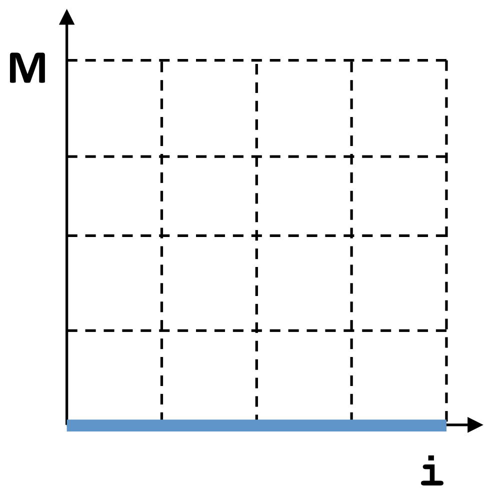
Initially



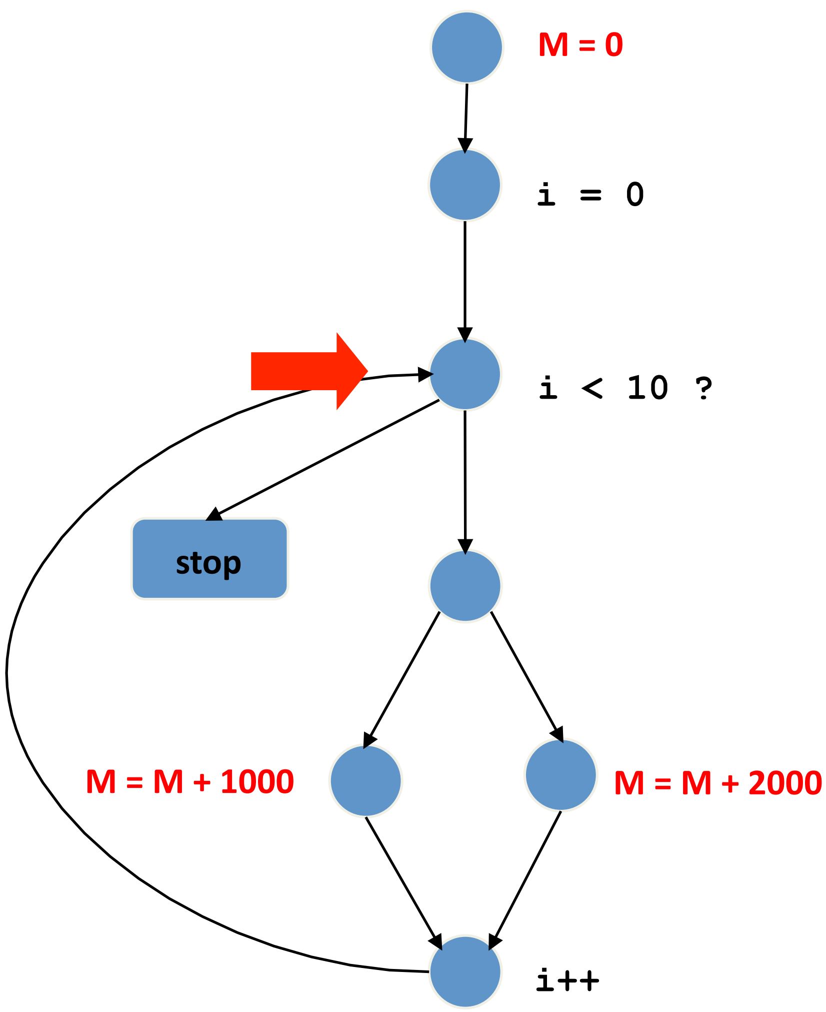


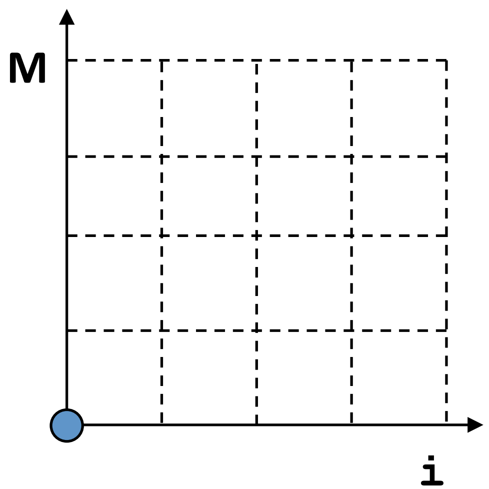
Loop initialization



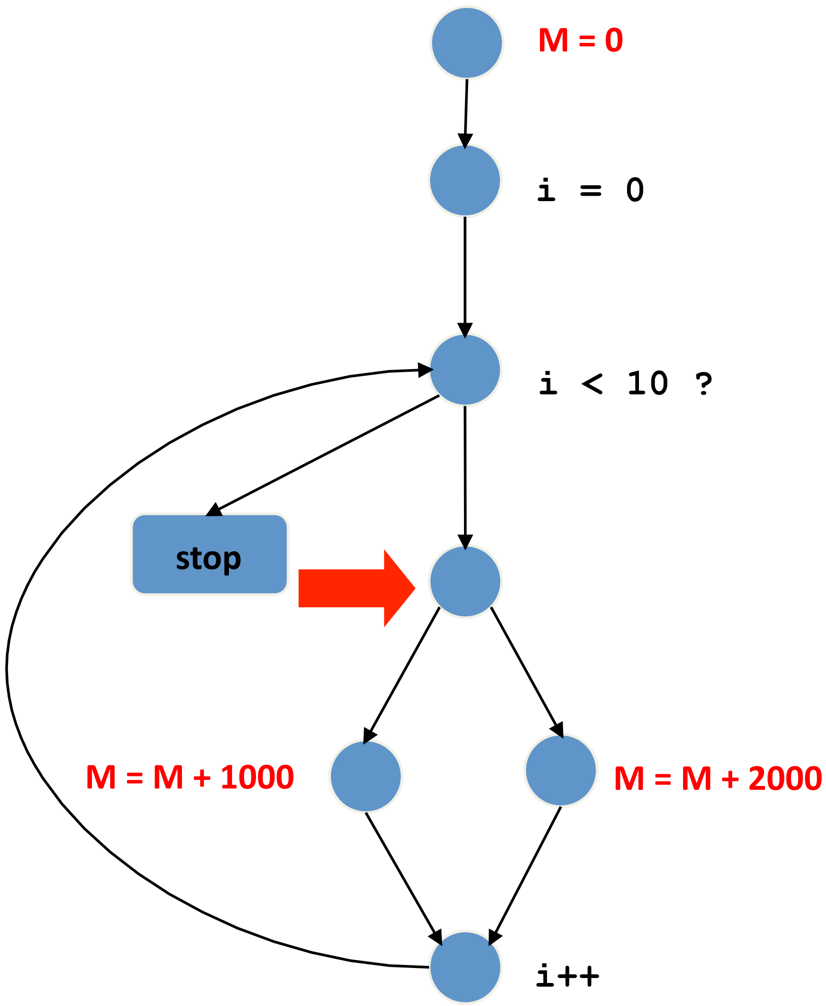


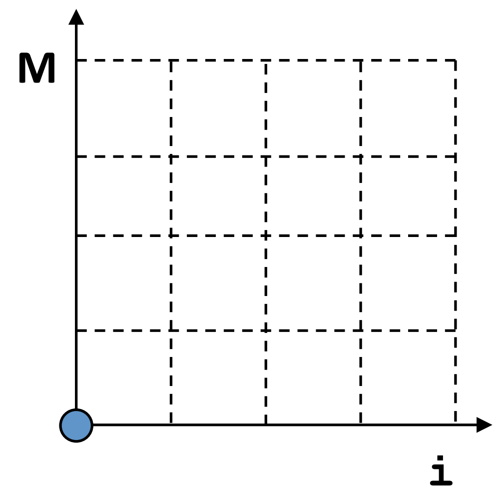
Loop entry



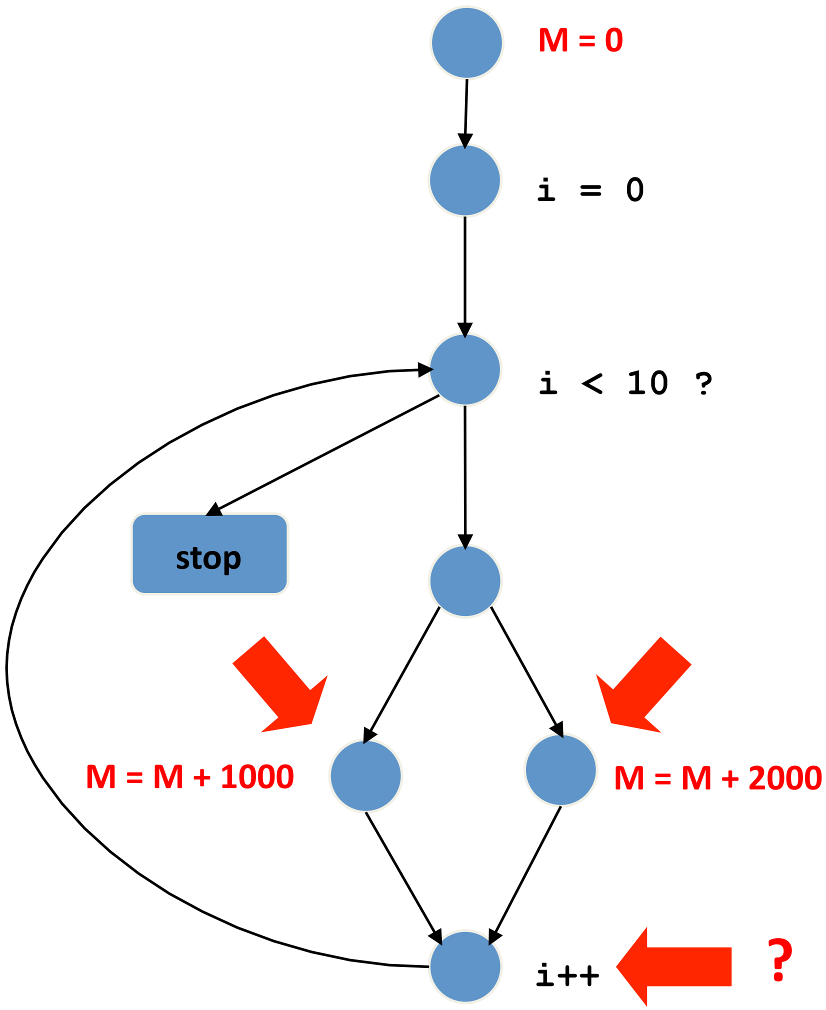


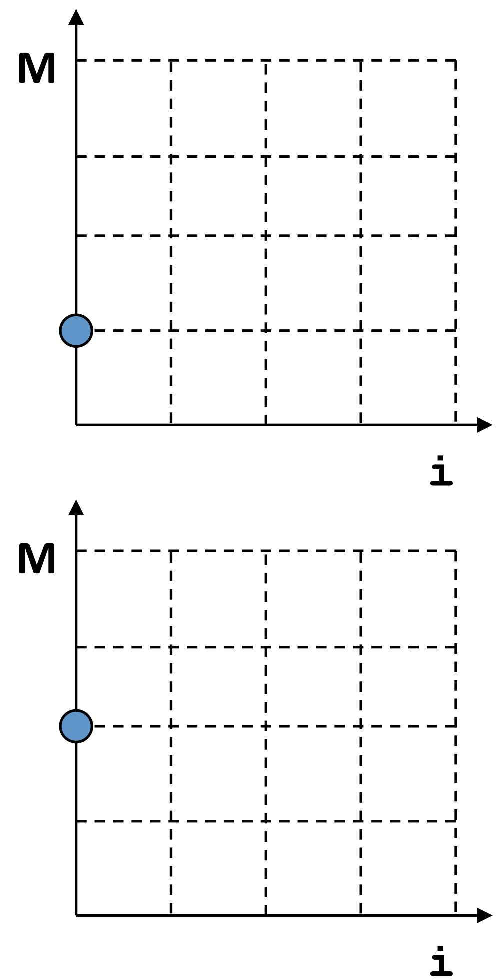
Analyzing a branching (1)



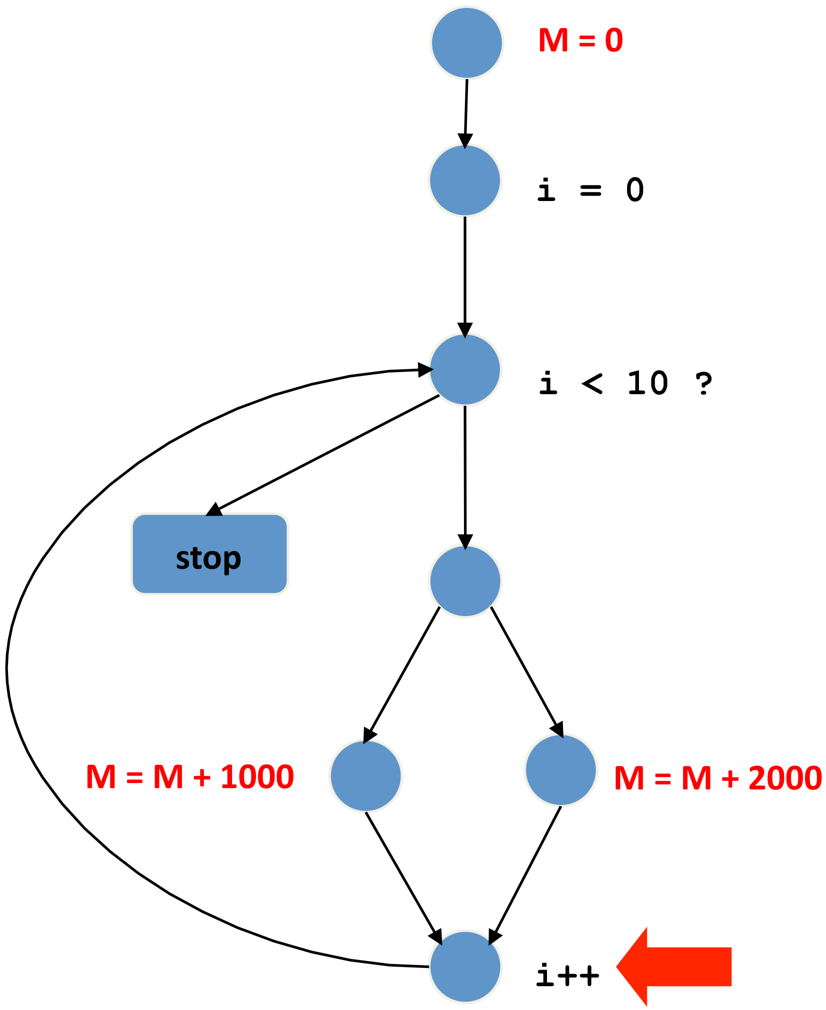


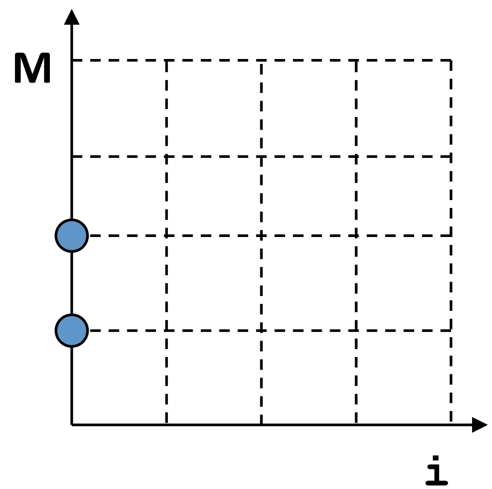
Analyzing a branching (2)





Accumulating all possible values

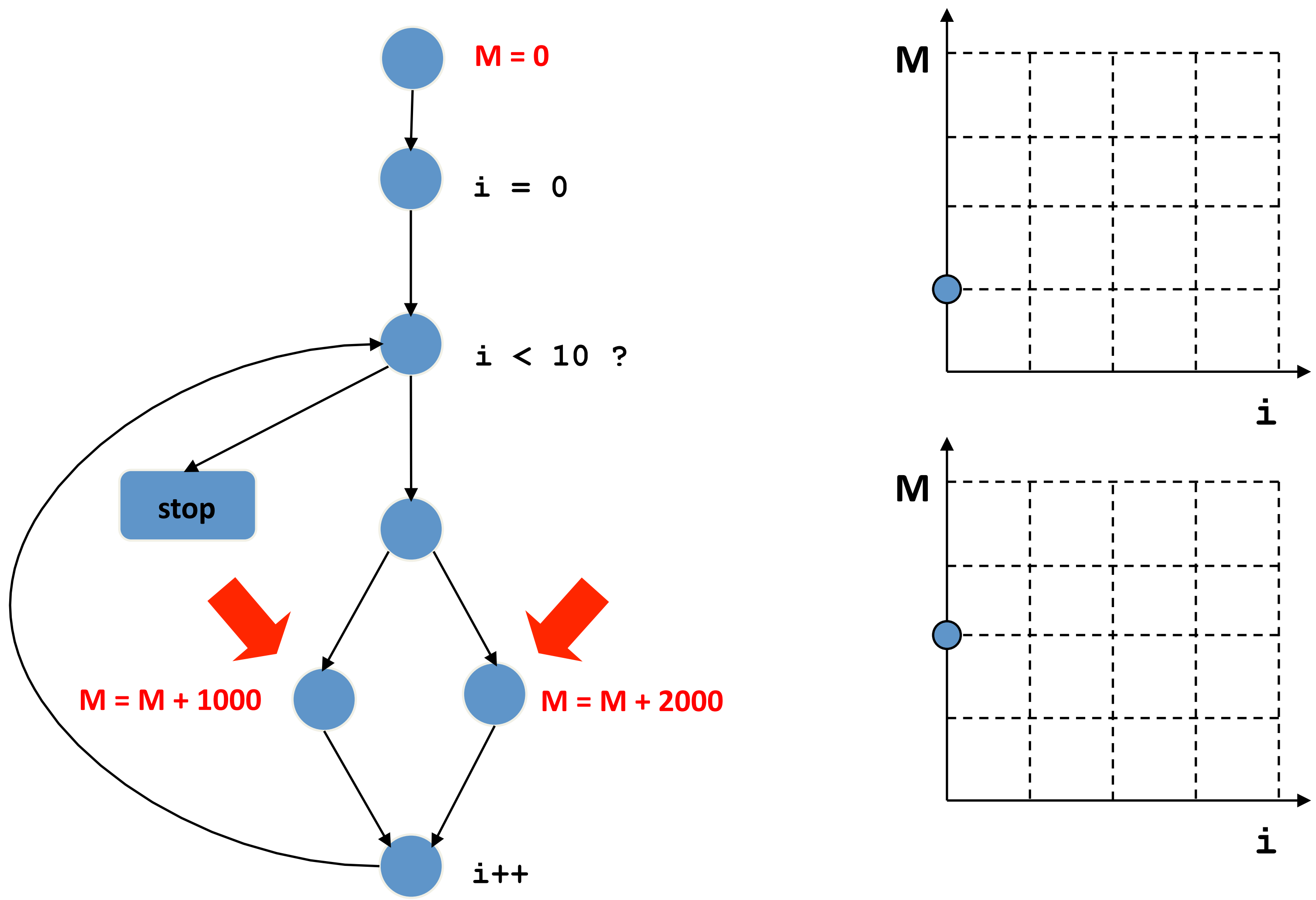




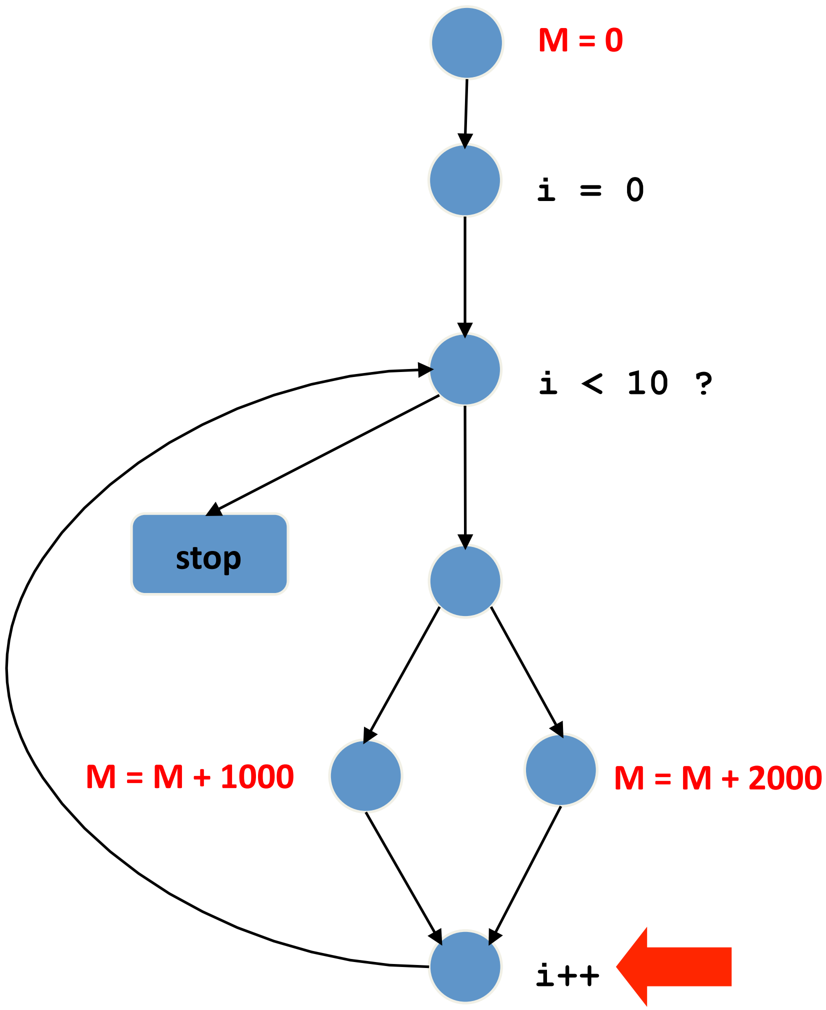
Abstraction of point clouds

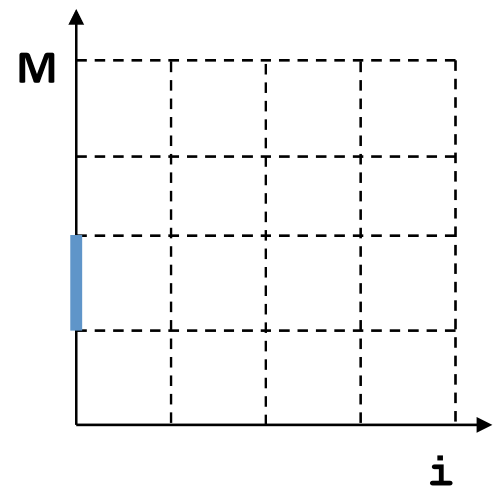
- We want the analysis to terminate in reasonable time
- We need a tractable representation of point clouds in arbitrary dimensions
- **Convex polyhedra (Cousot & Halbwachs, 1978)**
- Compute the convex hull of a point cloud

Analyzing a branching

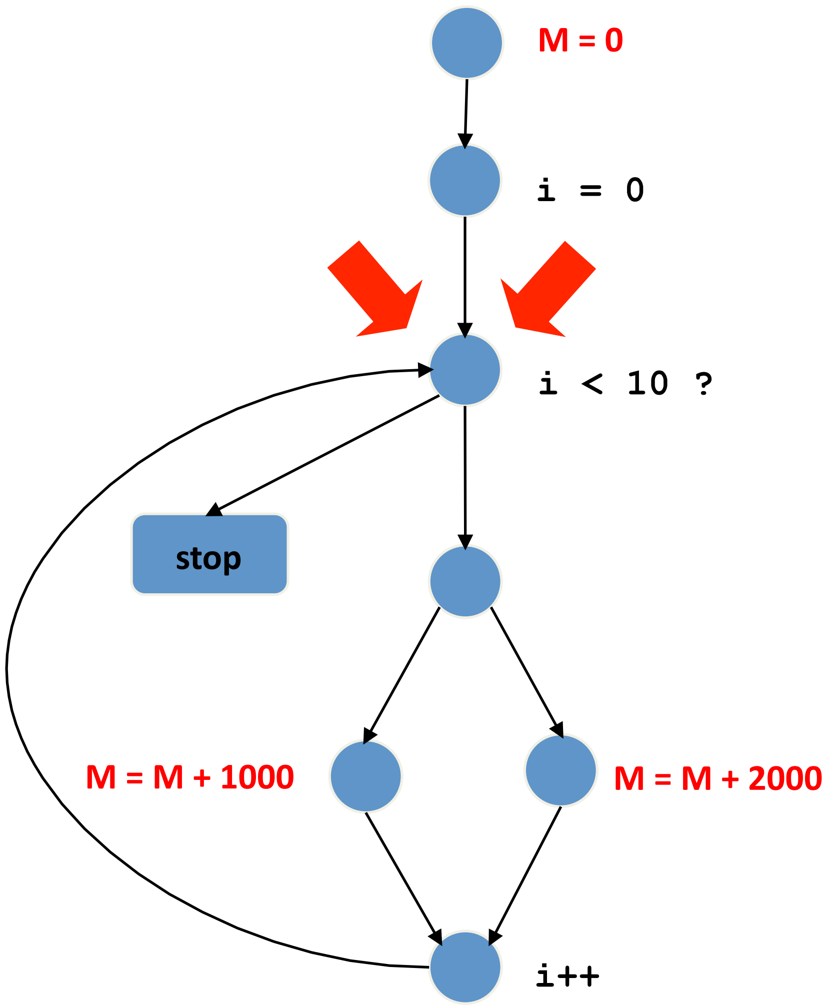


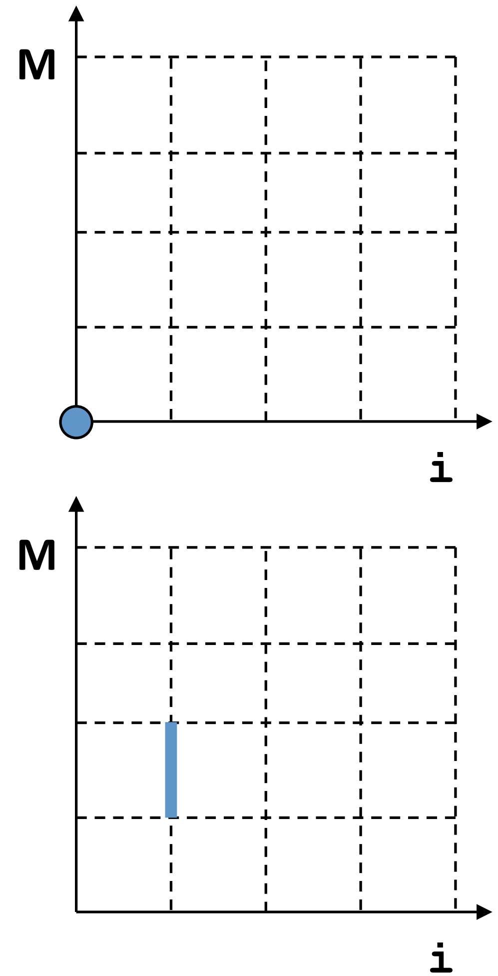
Convex hull



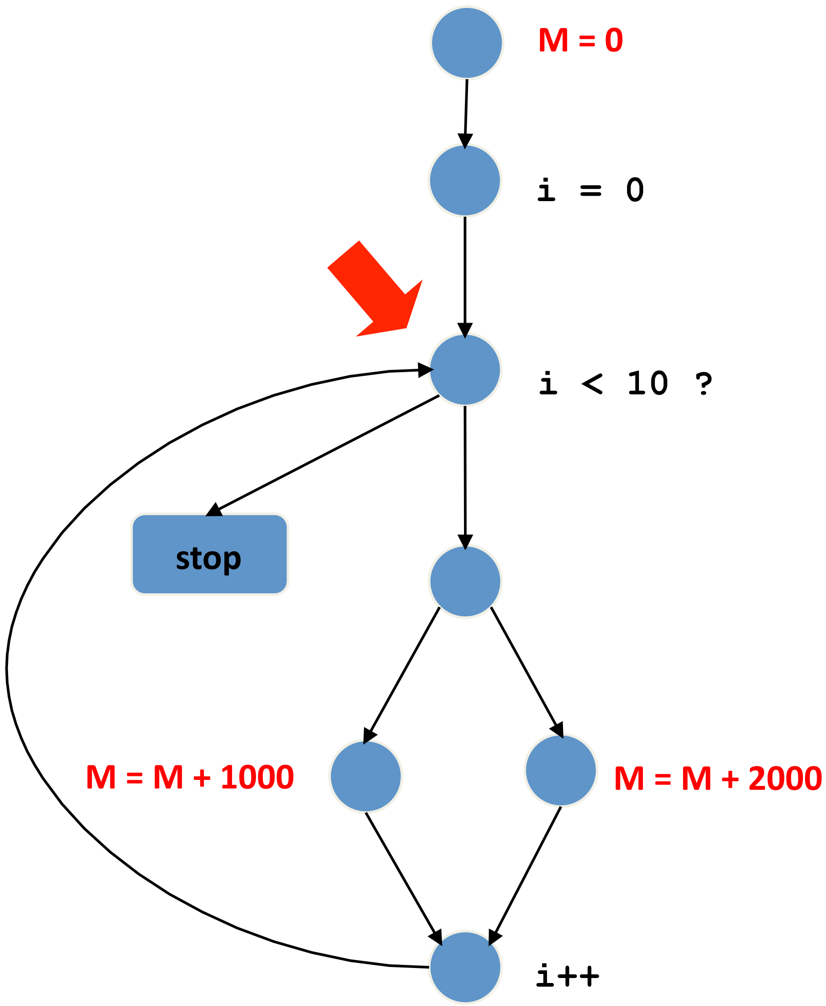


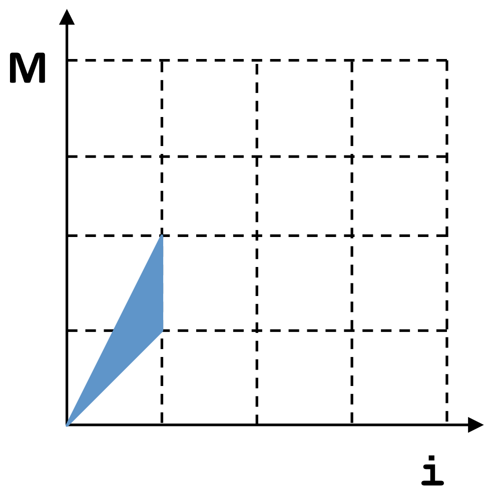
Iterating the loop analysis



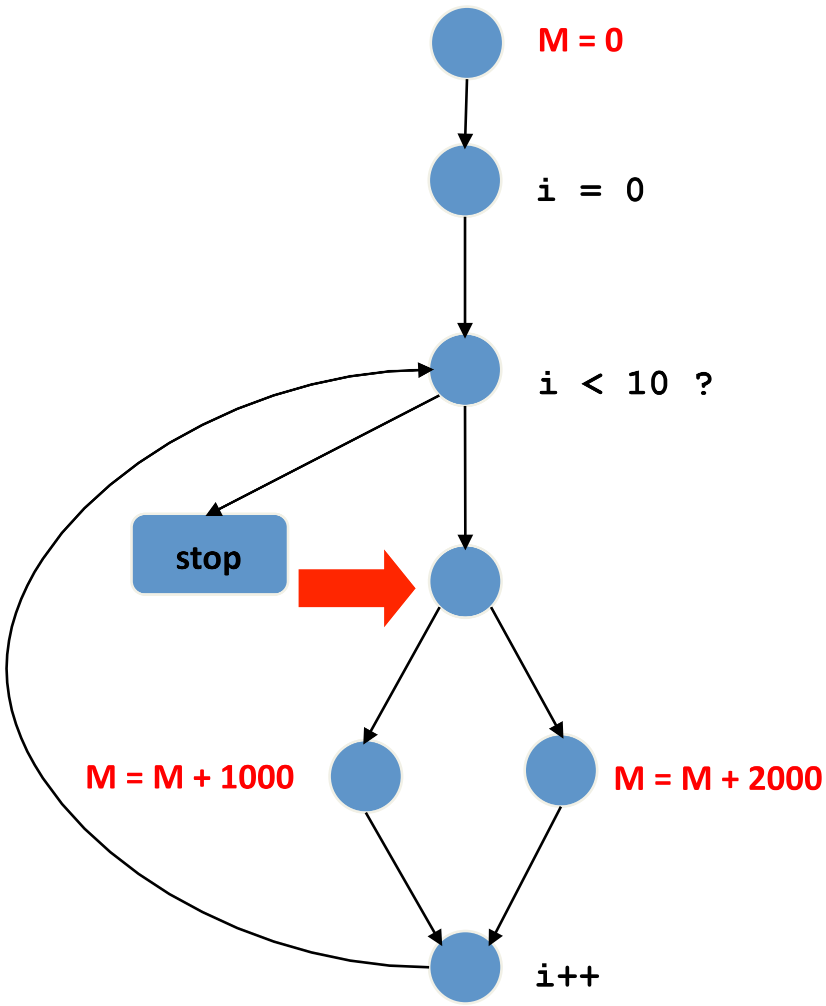


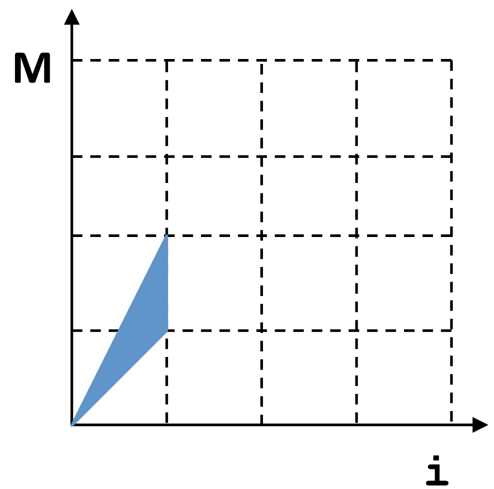
Building the loop invariant



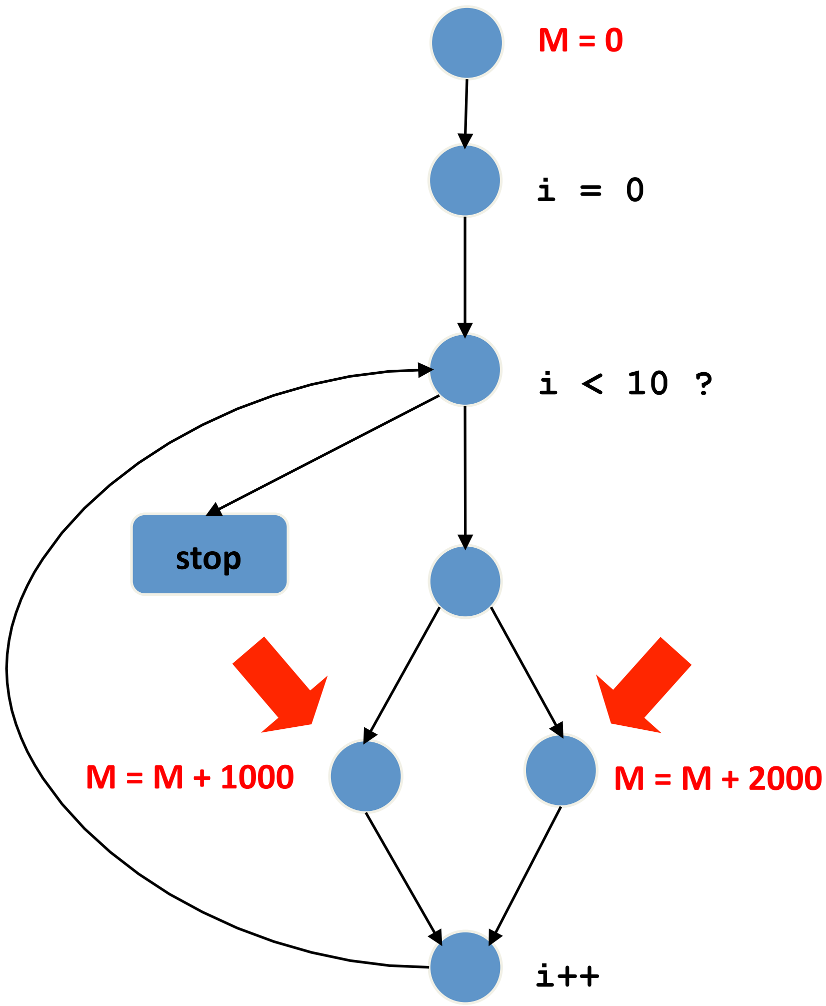


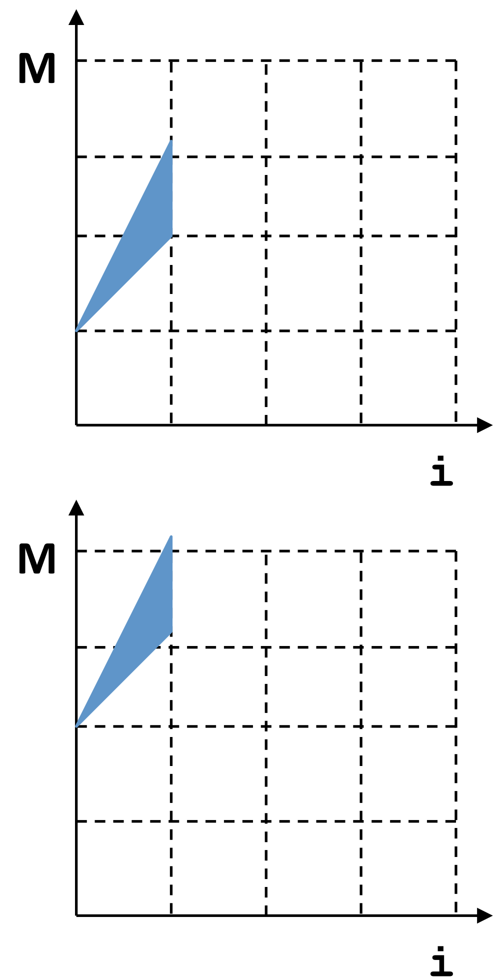
Analyzing a branching



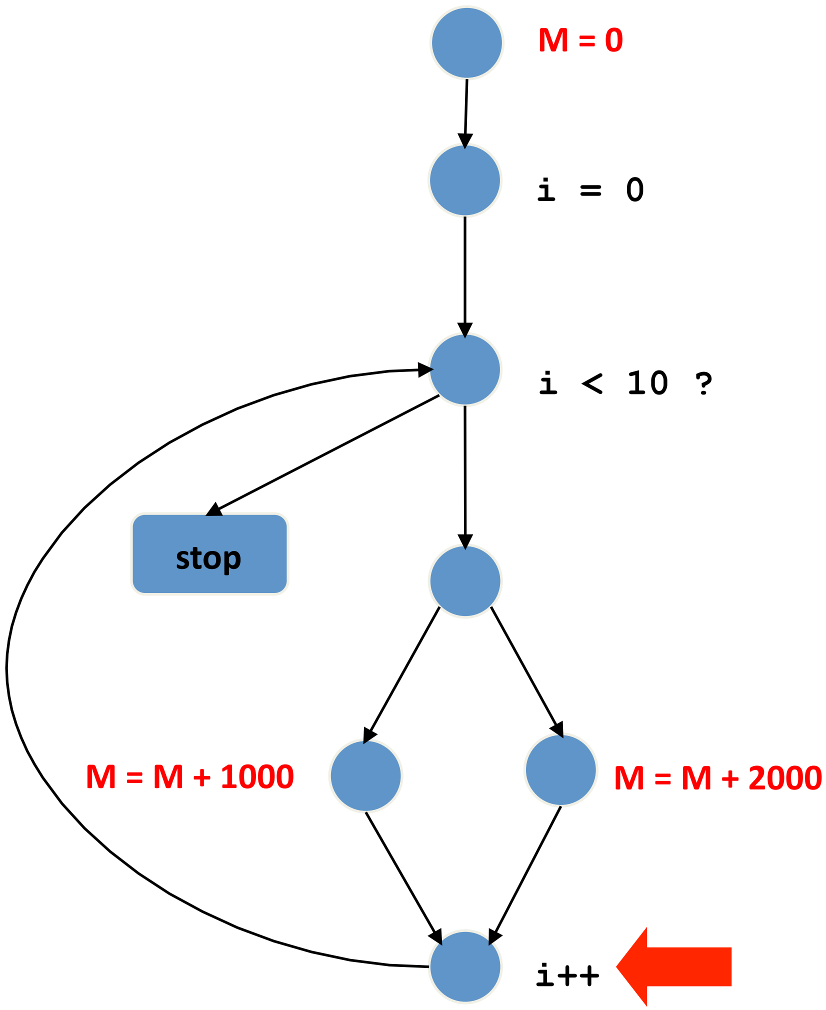


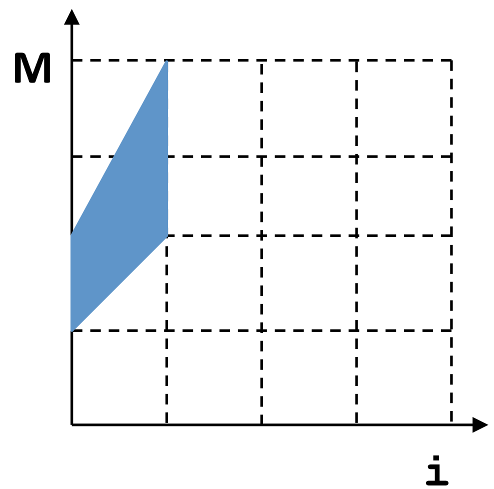
Analyzing a branching



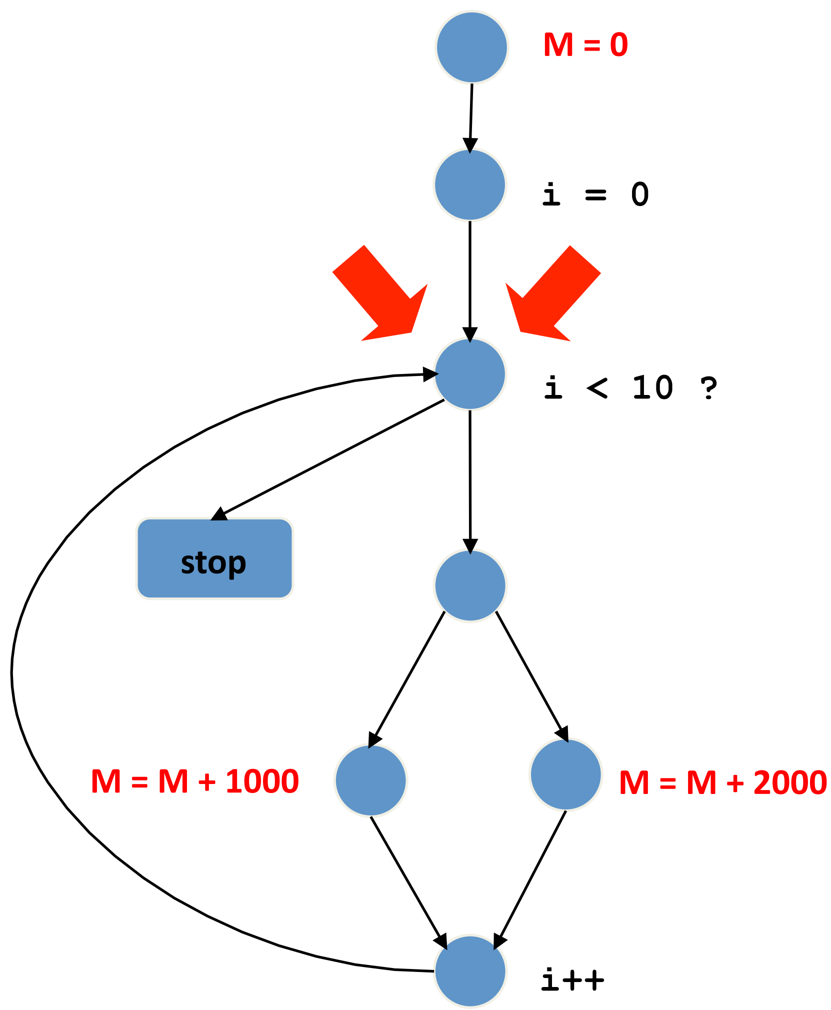


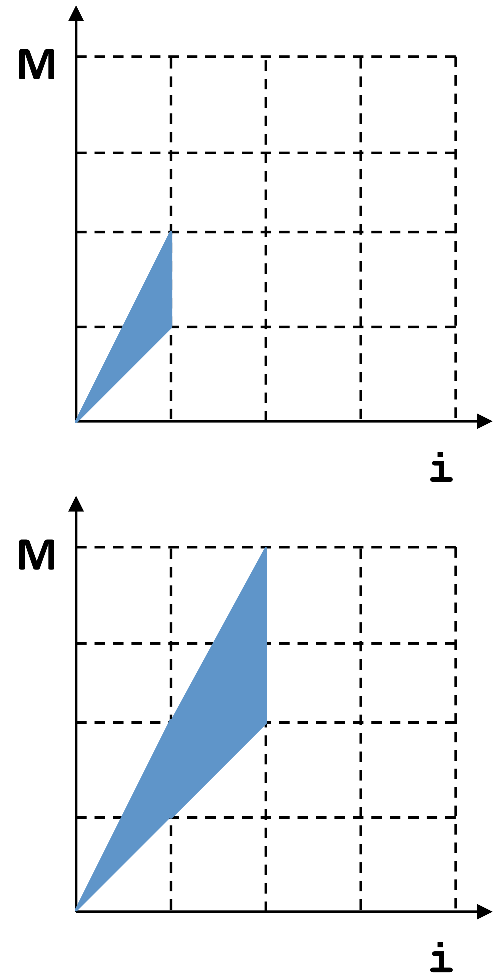
Convex hull



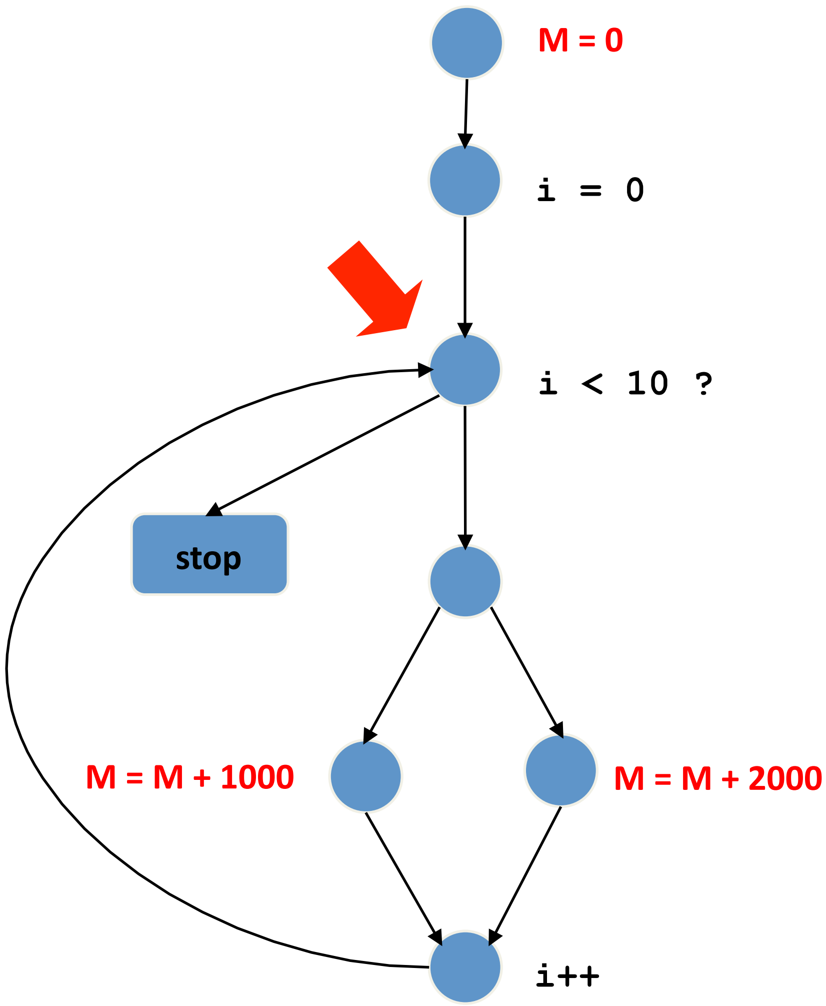


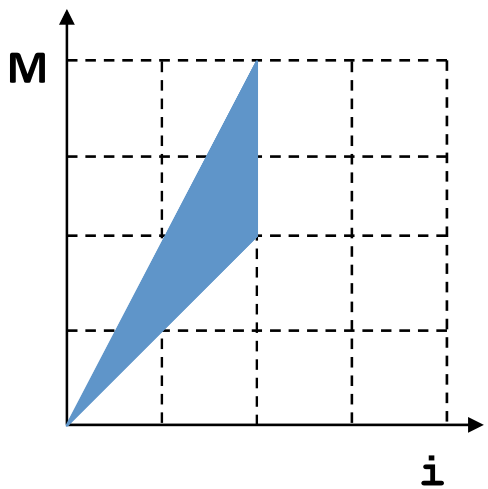
Building the loop invariant





Keep iterating...





Passing to the limit

- We want the analysis to terminate when analyzing loops
- After a few iteration steps, we use a ***widening*** operation at loop entry to enforce convergence

Widening ∇

- Let a_1, a_2, …a_n, … be a sequence of polyhedra, then the sequence
  - $w_1 = a_1$
  - $w_{n+1} = w_n \nabla a_{n+1}$

  is ultimately stationary
- The widening is a ***join*** operation:

$$a \subseteq a \nabla b \quad \& \quad b \subseteq a \nabla b$$

Widening for intervals

- $[a, b] \nabla [c, d] =$

$$[\textbf{if } c < a \textbf{ then } -\infty \textbf{ else } a, \textbf{ if } b < d \textbf{ then } +\infty \textbf{ else } b]$$

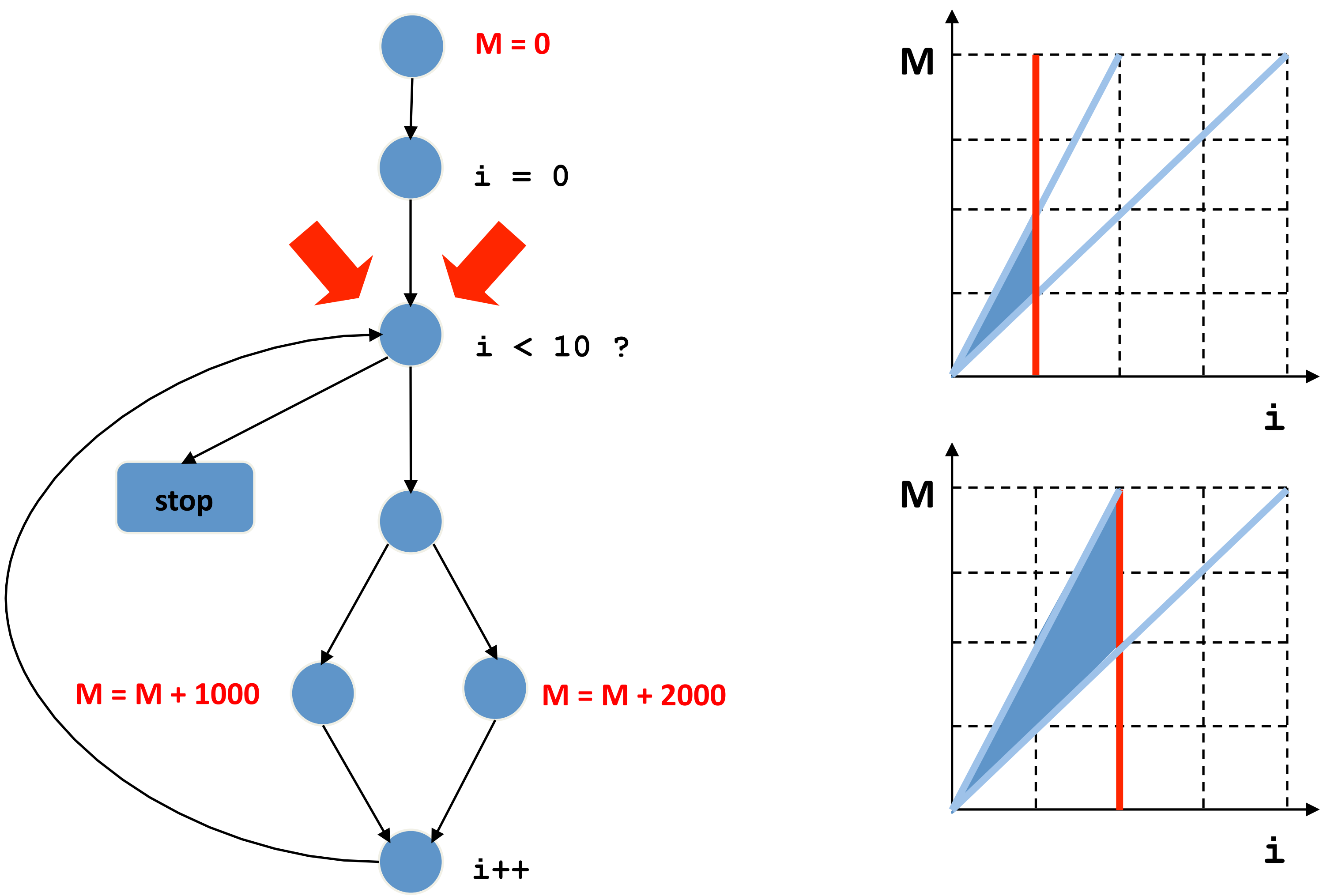
- Example:

$$[10, 20] \nabla [11, 30] = [10, +\infty]$$

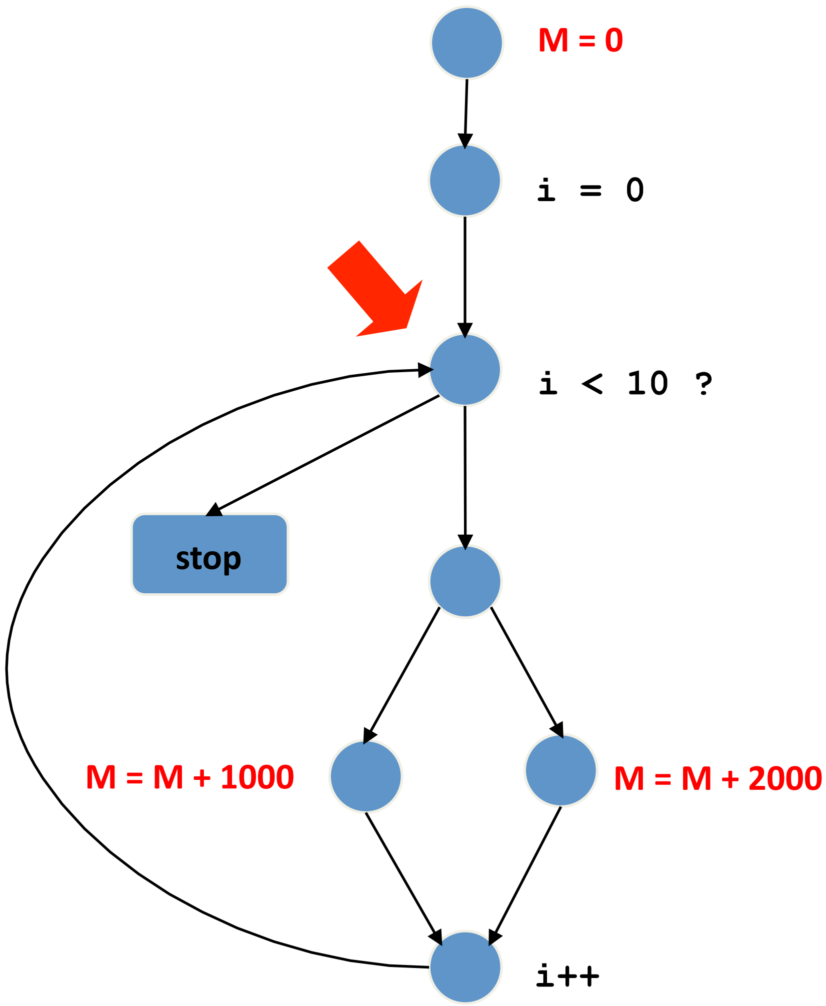
Widening for polyhedra

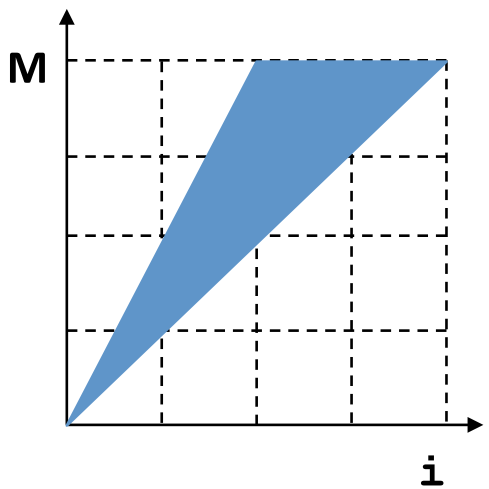
- We eliminate the faces of the computed convex envelope that are not stable
- Convergence is reached in at most N steps where N is the number of faces of the polyhedron at loop entry

Widening



After the widening





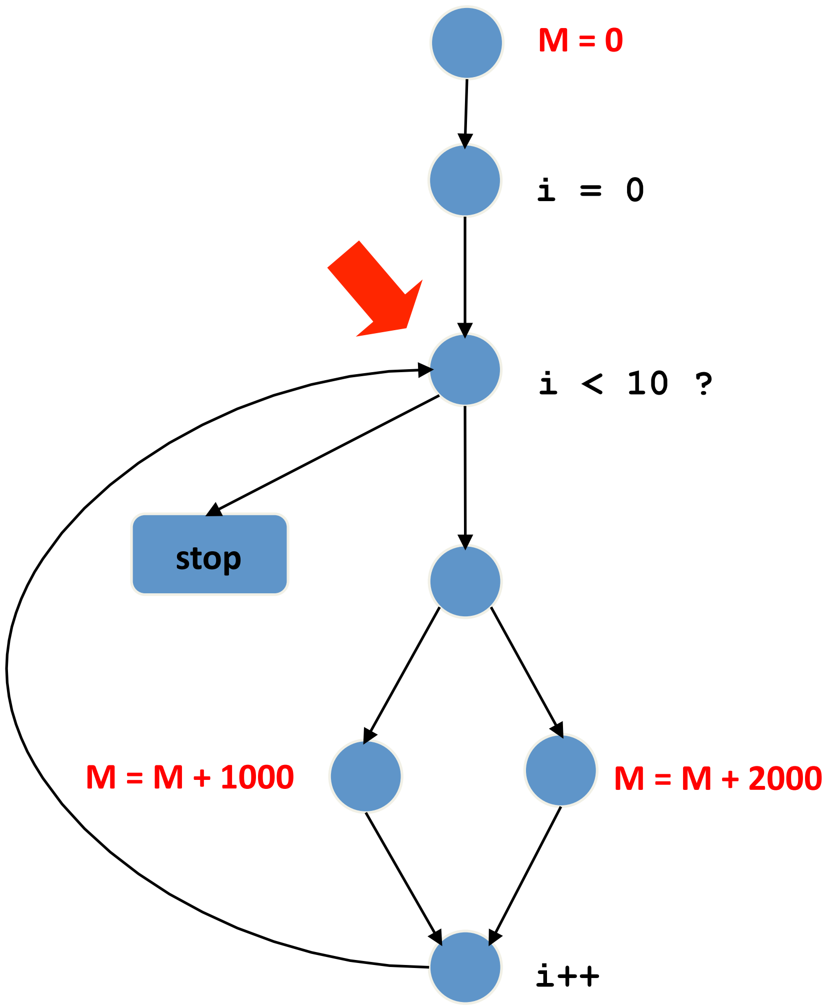
Detecting convergence

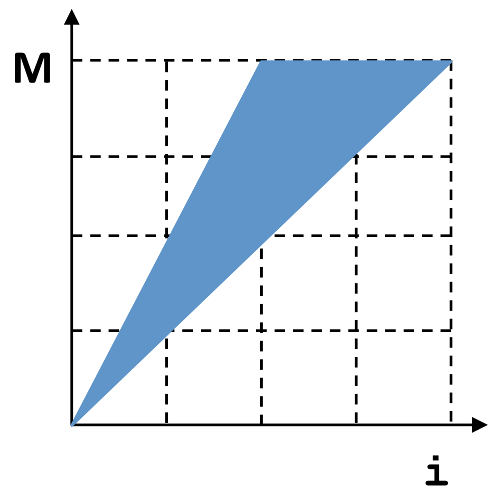
- Abstract iteration sequence
  - $F_1 = P$ (initial polyhedron)
  - $F_{n+1} = F_n$ if $\mathbf{S}(F_n) \subseteq F_n$

    $F_n \nabla \mathbf{S}(F_n)$ otherwise

  where **S** is the semantic transformer associated to the loop body
- **Theorem:** if there exists N such that $F_{N+1} \subseteq F_N$, then $F_n = F_N$ for $n > N$.

Convergence





The computation has converged

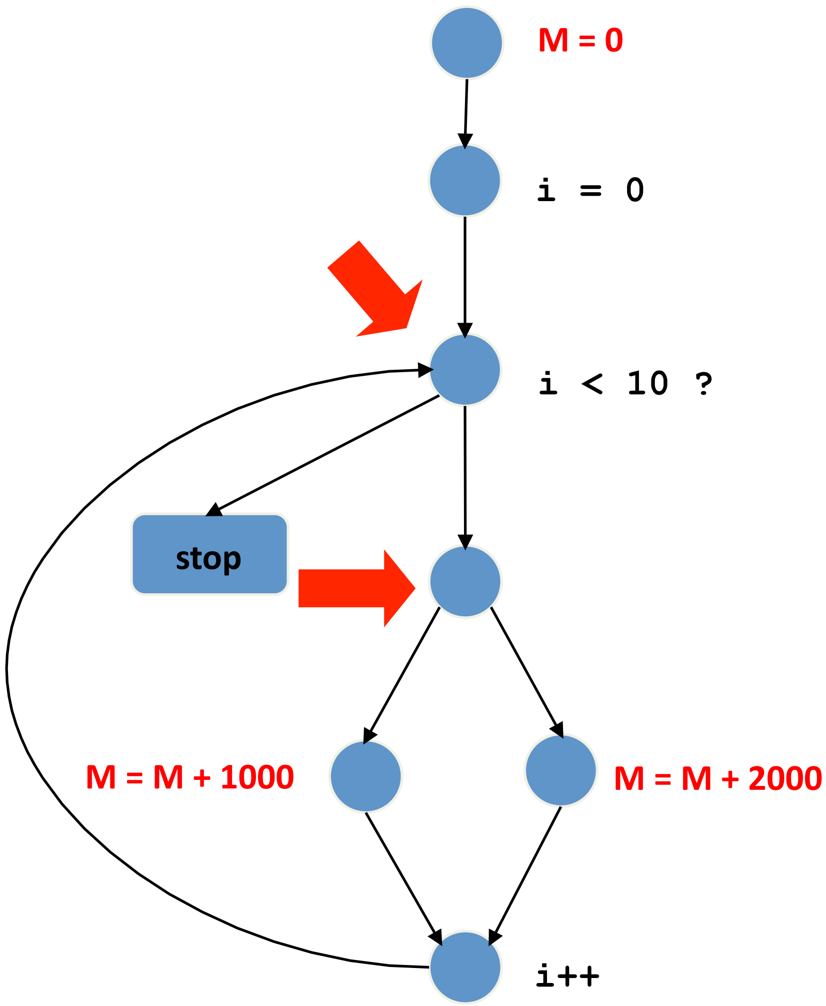
We are not done yet…

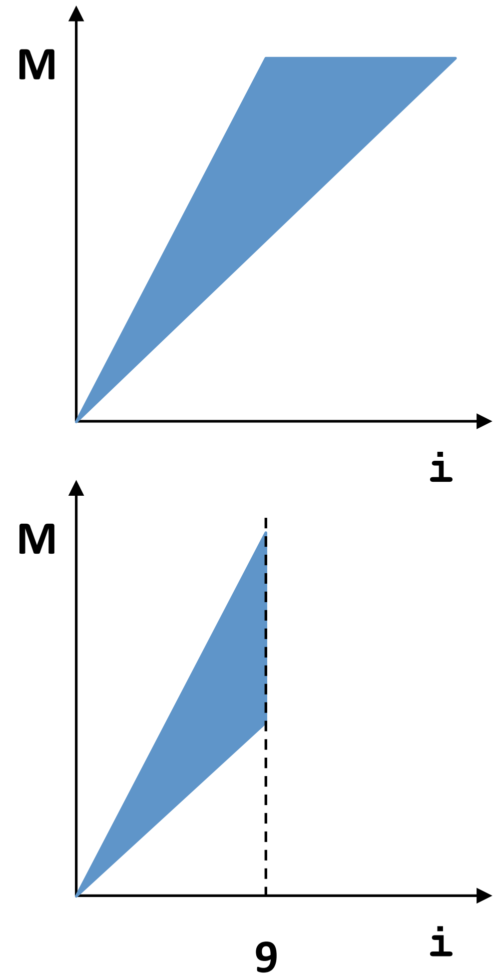
- The analyzer has just proven that

  $1000 * i \leq M \leq 2000 * i$

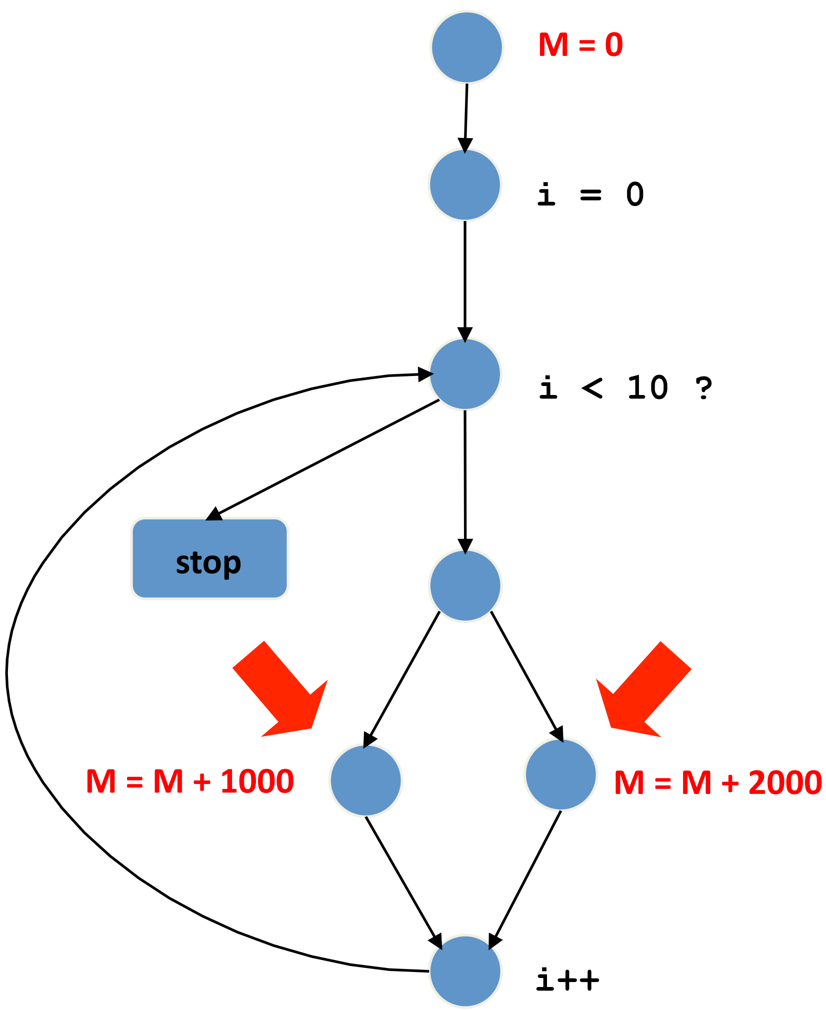
- But we have lost all information about the termination condition **$0 \leq i \leq 10$**
- Since we have obtained a superset of all possible values of the variables, if we run the computation again we still get a superset
- This new envelope may be smaller
- This refinement step is called ***narrowing***

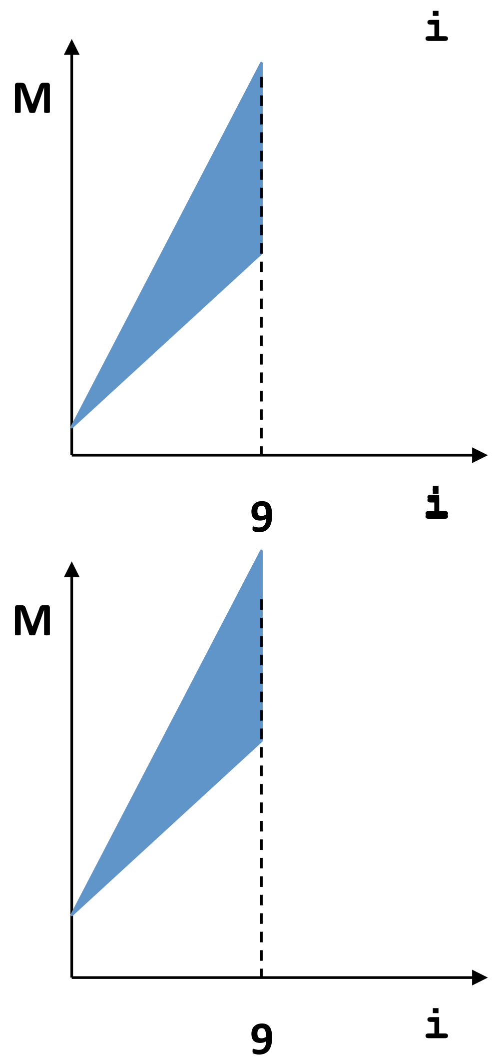
Refinement



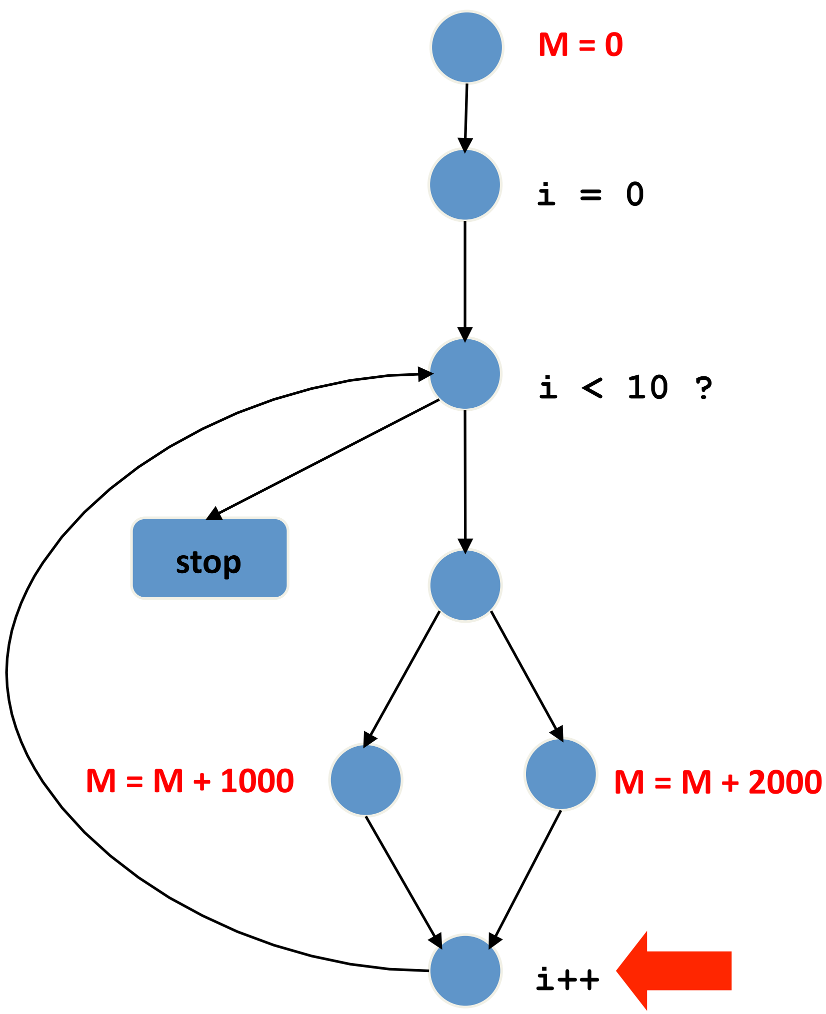


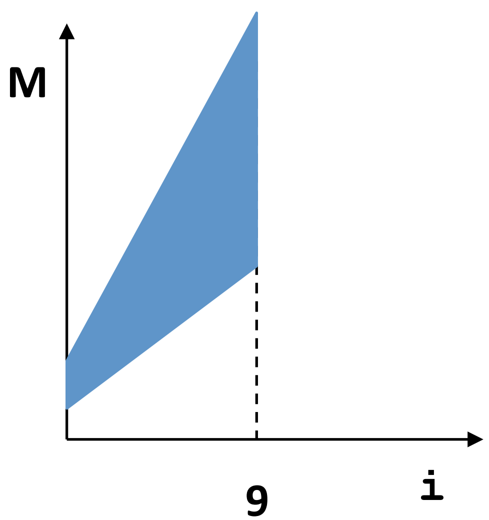
Analyzing a branching



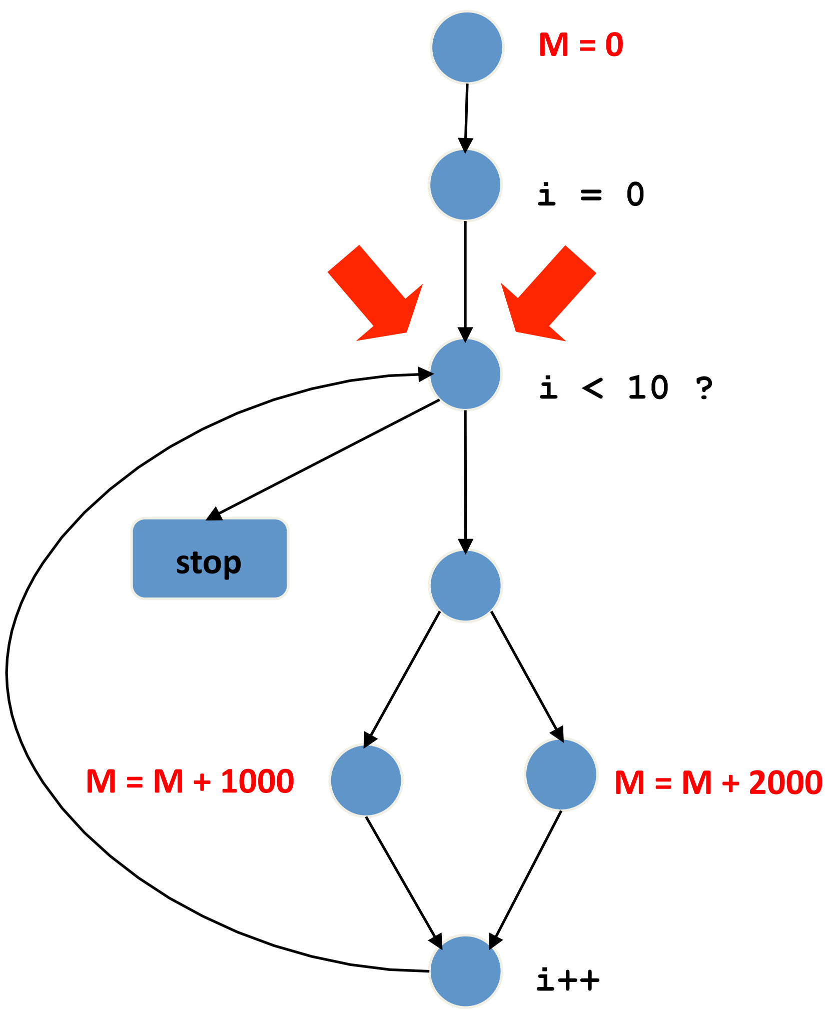


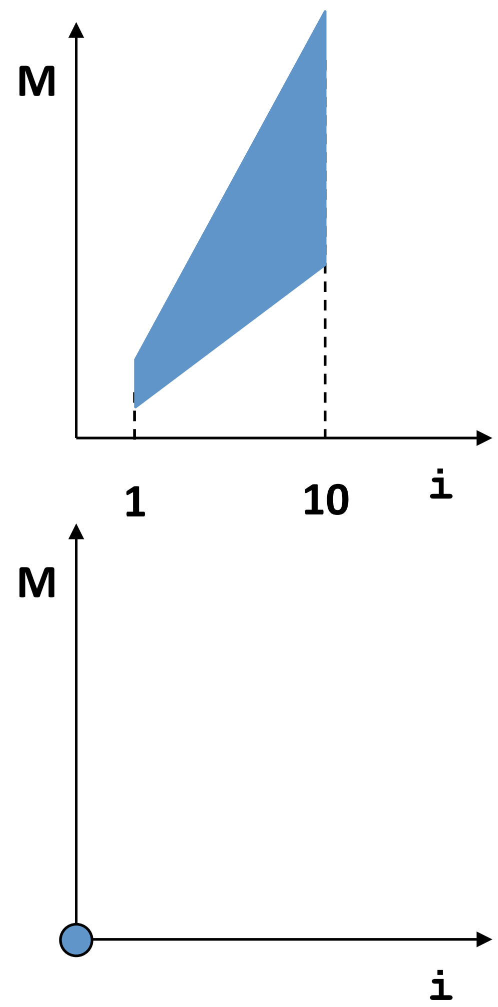
Convex hull



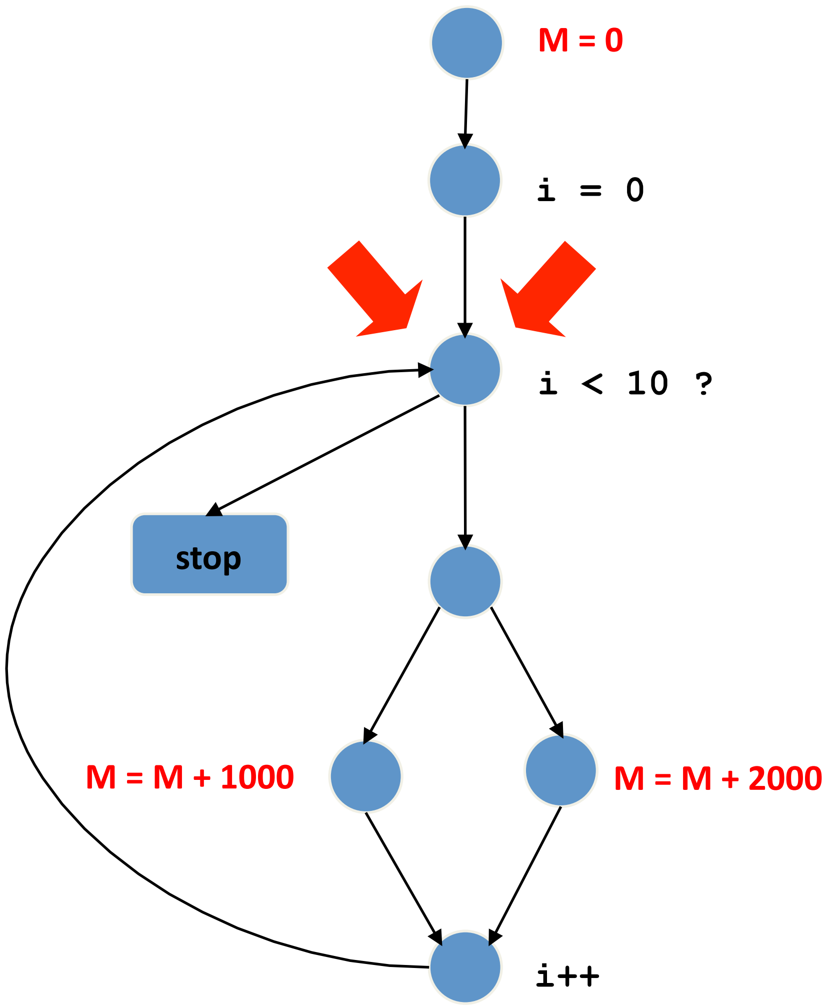


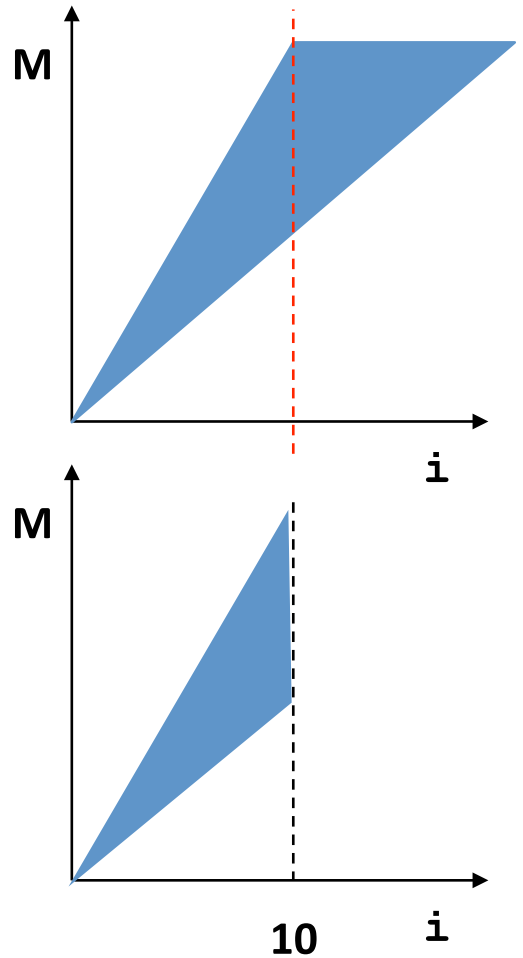
Back to loop entry



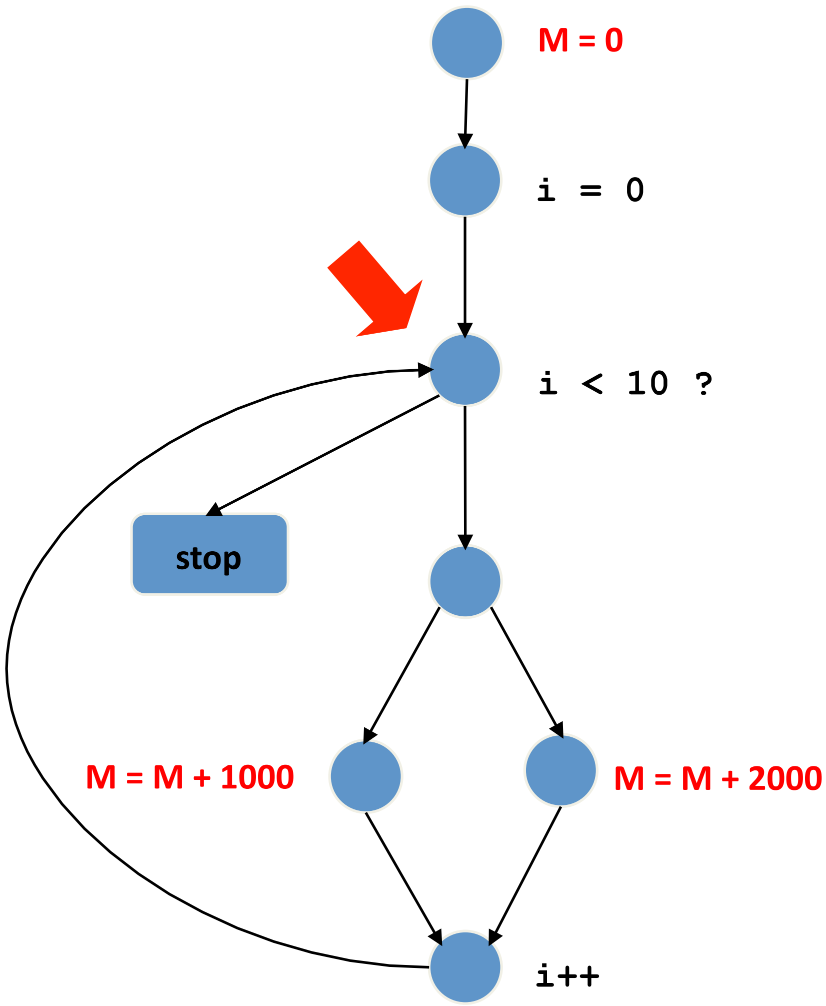


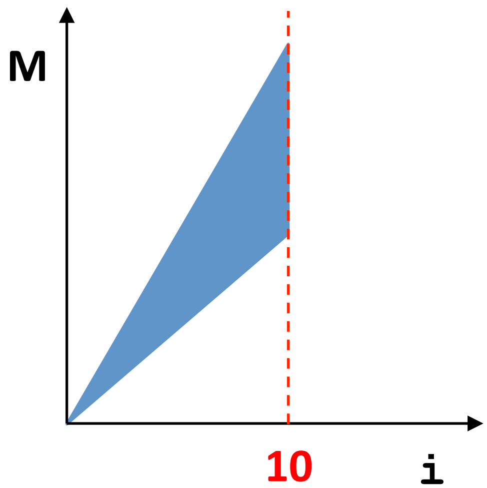
Narrowing



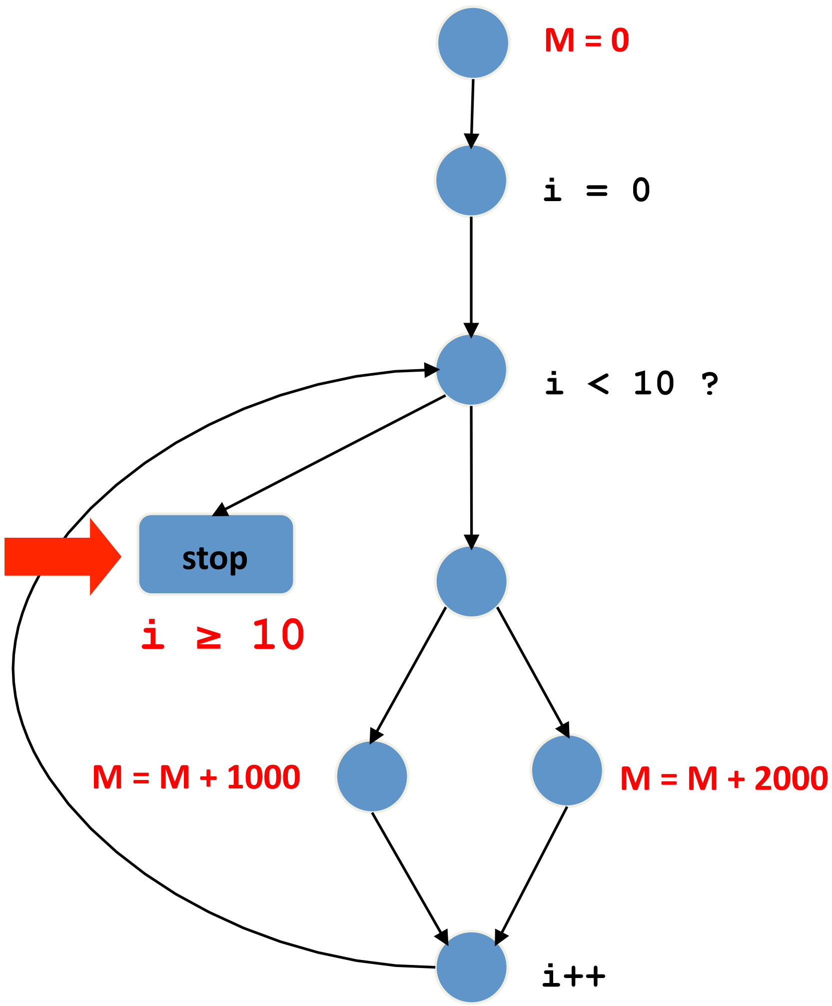


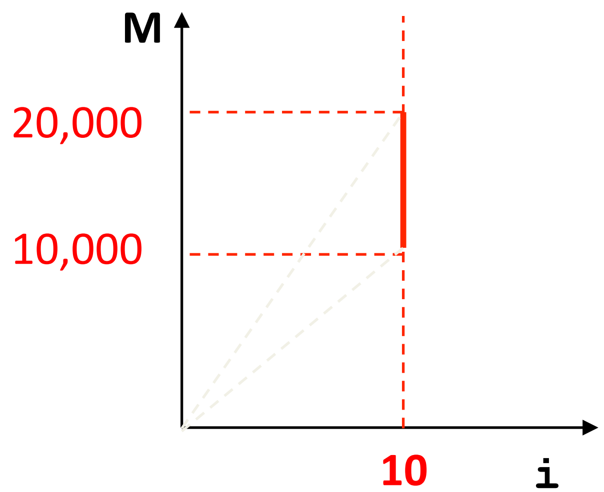
Refined loop invariant





Invariant at loop exit





Static array-bound checking

The problem

```
double a[10];
for (i = 0; i < 10; i++) {
    a[i] = 1.0; ✔
}
a[i] = 0.0; ✖
```

$i \in [0, 9]$ → `a[i]` = 1.0; ✔

$i = 10$ → `a[i]` = 0.0; ✖

- Do all array access operations occur within bounds?
- Requires the computation of numerical invariants

Why is it important?

- Most critical applications are written in C (flight software, SSH, BIND)
- No runtime checks
- The memory is silently corrupted
  - Source of nondeterminism
  - Vulnerability to malicious attacks
  - Standard test practices are of little help
- About 50% of all CERT reports originate from a buffer overflow

Arrays or pointers?

- In C, every memory access goes through a pointer:

```
a[i] = *(a + i)
```

- Tracking a pointer `p` requires
  - A symbolic address p_{addr} = `&A`, `malloc(…)`
  - A numerical offset p_{off} expressed in **bytes**
- It is not safe to rely on the type information in C
- `S.f.g` is translated into `<&S, off(f) + off(g)>`

Example

```
struct bytes {
  unsigned char b[4];
};
int i;
struct bytes *p = (struct bytes *)&i;
p->b[1] = 0x03;
...
```

- This comes from a real embedded application
- Byte-level granularity is required

Taxonomy (I)

- Ideal case: static allocation and bounded offsets

```
double a[10];
for (i = 0; i < 10; i++) {
  a[i] = 1.0;
}
a[i] = 0.0;
```

- Usually occurs at the function level
  - Local manipulations on stack allocated buffers
- In practice it is a small fraction of all array accesses

Taxonomy (II)

- Interprocedural pointers and bounded offsets

```
...
f(&big_struct.s);
...
```

```
void f(struct S *p) {
  int i;
  for (i = 0; i < 8; i++) {
    p->a[i] = ...;
  }
}
```

- Very common in embedded code
- MATLAB/Simulink autocode falls under this category

Taxonomy (III)

- Offsets and pointers are intertwined

```
...
f(&S[3], 8);
...
```

```
void f(double *p, int n) {
  int i;
  for (i = 0; i < n; i++) {
    p[i] = ...;
  }
}
```

- This is the worst case and is also very common
- Complex, critical codes:
  - Mars Exploration Rovers mission control software
  - Intelligent flight controllers
  - Security-sensitive applications (SSH, BIND)

What analysis to use?

- **Type I:**
  - Intervals at the function level
- **Type II:**
  - Separate pointer analysis: field sensitive, flow-insensitive, context-sensitive
  - Intervals at the function level
  - 99% accuracy on MATLAB/Simulink autocode
- **Type III:**
  - Relational numerical domain
  - Inline function calls and/or compute function summaries
  - Scalability is an issue

Roadmap

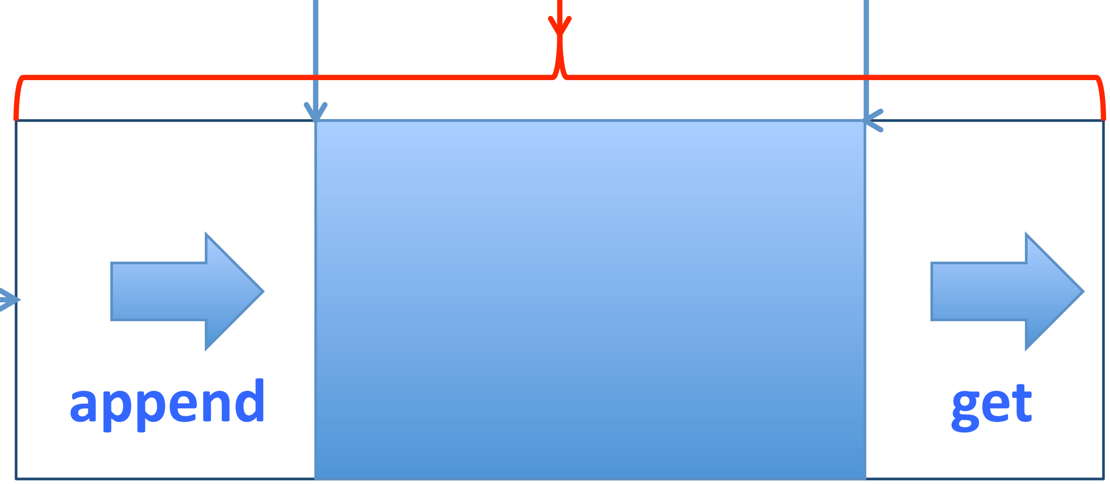
- There are many numerical domains available in the literature
- How to put the existing domains to work on real applications:
  - **The buffer library of OpenSSH (700 LOC)**
  - **The flight software of Mars Exploration Rovers (550 KLOC)**
- We may need different types of abstractions:
  - **The gauge domain**

OpenSSH

- **Description**
  - Open-source implementation of utilities based on the SSH protocol (ssh, scp, sftp, etc.)
  - Widely used, security sensitive
- **Implementation**
  - OpenSSH uses a single data structure to represent buffers
  - Cryptographic keys, deciphered messages, etc. are all stored in buffers
  - Good target for verification by static analysis

Buffer structure

```
typedef struct {
    u_char *buf;
    u_int alloc;
    u_int offset;
    u_int end;
} Buffer
```



Characteristics

- Standard FIFO queue
- 700 LOC
- Lots of Boolean logic added for fault tolerance
- The queue expands by increments if there is not enough space
  - The most complex algorithm in the library
  - "Weird" implementation using a backward goto

Expansion algorithm

```
void *
buffer_append_space(Buffer *buffer, u_int len)
{
        u_int newlen;
        void *p;

        if (len > BUFFER_MAX_CHUNK)
                fatal("buffer_append_space: len %u not supported", len);

        /* If the buffer is empty, start using it from the beginning. */
        if (buffer->offset == buffer->end) {
                buffer->offset = 0;
                buffer->end = 0;
        }
restart:
        /* If there is enough space to store all data, store it now. */
        if (buffer->end + len < buffer->alloc) {
                p = buffer->buf + buffer->end;
                buffer->end += len;
                return p;
        }
        /*
         * If the buffer is quite empty, but all data is at the end, move the
         * data to the beginning and retry.
         */
        if (buffer->offset > MIN(buffer->alloc, BUFFER_MAX_CHUNK)) {
                memmove(buffer->buf, buffer->buf + buffer->offset,
                        buffer->end - buffer->offset);
                buffer->end -= buffer->offset;
                buffer->offset = 0;
                goto restart;
        }
        /* Increase the size of the buffer and retry. */

        newlen = buffer->alloc + len + 32768;
        if (newlen > BUFFER_MAX_LEN)
                fatal("buffer_append_space: alloc %u not supported",
                    newlen);
        buffer->buf = xrealloc(buffer->buf, newlen);
        buffer->alloc = newlen;
        goto restart;
        /* NOTREACHED */
}
```

Add data of length `len`

`end + len < alloc` ➔ done

Try to pack data to the left and retry

Expand size by increment and retry

Appending data to the buffer

```
void
buffer_append(Buffer *buffer, const void *data, u_int len)
{
    void *p;
    p = buffer_append_space(buffer, len);
    memcpy(p, data, len);
}
```

Automatically prove that the operation stays within the bounds of the buffer

Design of the analysis

- The expressive power of convex polyhedra is required
- Inlining the library into the OpenSSH code is not conceivable
- Modular approach:
  - We build a simplified model of a client of the library on one buffer
  - The client nondeterministically calls functions of the library on the buffer with consistent arguments
  - We inline the library code into the client and analyze it

The client

```
volatile u_int random;
Buffer buffer;

buffer_init(&buffer);
for(random) {
  switch(random) {
    case 0: {
      u_int len = random;
      u_char *data = malloc(len);
      buffer_append(buffer, data, len);
      break;
  }
  …
}
```

First try

- Settings
  – Polyhedral domain: Bertrand Jeannet's New Polka
  – C front-end: CIL
  – Fixpoint iterator: Bourdoncle's algorithm
- Running the analysis:
  – Failure
  – The widening operation on polyhedra crashes because there are too many variables

Optimizations

- The front-end generates a lot of auxiliary variables, which weigh on the polyhedral domain
- Inlining also introduces lots of redundancy
- We run initial passes that perform:
  - Constant propagation
  - Copy propagation
  - Dead variable elimination
- The number of variables is greatly reduced
- New run: Crash!

A bit of head scratching

- The crash always occurs during the widening
- We make two observations:
  - The invariants contain a lot of linear **equalities**
  - Most of these equalities are common to both operands of the widening
- We decide to remove the common equalities from the invariants, apply the widening and add them back to the result

It works!

- The analysis runs in few seconds
- But all the nontrivial checks are flagged as warnings…
- It finally scales but now it’s not precise enough
- The problem comes from the logic inserted to make the library robust

Example

```
int
buffer_consume_ret(Buffer *buffer, u_int bytes)
{
    if (bytes > buffer->end - buffer->offset) {
        error("buffer_consume_ret: trying to get more bytes
than in buffer");
        return (-1);
    }
    buffer->offset += bytes;
    return (0);
}

void
buffer_consume(Buffer *buffer, u_int bytes)
{
    if (buffer_consume_ret(buffer, bytes) == -1)
        fatal("buffer_consume: buffer error");
}
```

Join of invariants

Loss of precision

Solution

- We could use trace partitioning techniques (Rival & Mauborgne)
  - Dramatically complicates the analysis
- We are only interested in execution traces that do not abort
  - We model the `fatal` function as bottom
  - We perform an iterated forward/backward analysis between the beginning and the end of each library operation
- **Full verification is achieved in 35 seconds!**

Observations

- If we turn off the initial optimizations the analyzer crashes
- How far can we push the scalability with the optimized widening?
- Not very far
  - We added one variable to the main loop of the client
  - The analyzer crashes
- The approach based on a general-purpose expressive domain seems very brittle

Mars Exploration Rovers

- Large flight software (550+ KLOC)
- Developed with an object-oriented approach
- Thousands of small generic functions
- Our approach:
  - Compute function summaries
  - No loops in summaries, just numerical invariants and symbolic pointer constraints
  - Use a weakly relational numerical domain to achieve scalability: difference-bound matrices (DBMs)

Example

```
void assign(double *p, double *q, int n) {
  int i;
  for (i = 0; i < n; i++) {
    p[i] = q[i];
  }
}
```

$$\mathbf{p}_{\mathrm{off}} \leq x \leq \mathbf{p}_{\mathrm{off}} + 8\mathbf{n}$$

- Not expressible in the domain of DBMs or even octagons

Templates for pointer arithmetic

- We introduce a symbolic expression based on the syntax of the pointer expression from the AST:

`p[i][j]` → $b + k_1 o_1 + k_2 o_2$

- Constraints on the parameters of the template are expressible as DBMs:

$$\begin{cases} b = \mathbf{p}_{\text{off}} \\ k_1 = 64 \\ o_1 = \mathbf{i} \\ k_2 = 8 \\ o_2 = \mathbf{j} \end{cases}$$

Scalability

- We can express general linear inequalities at the price of a larger number of variables
- First experiments are a disaster
  - It takes hours to analyze a single function
  - The DBMs were supposed to scale better (cubic in the worst case)
- The problem is that the upper complexity bound is always attained!

Explanation

- Range constraints in DBMs (or octagons) are expressed using a special variable Z that is semantically equal to 0
- $x = [a, b]$ is expressed as $x - Z \leq b$ and $Z - x \leq -a$
- Variables in a program are always initialized (hopefully)
- The graph of unitary relations over the program variables is then strongly connected
  - Worst case for the closure algorithm

Variable packing

- A solution is to only consider relations over small sets of variables like in ASTREE
- Problem:
  - A good packing can be determined statically in ASTREE because of the specificities of the code considered
  - In our case we have a fairly general C program
- Our approach:
  - Dynamic variable packing at analysis time
  - Variables appearing in a statement are put together

Technicalities

- Doing dynamic packing is not straightforward as partitions must be merged on the fly:
  - Complex domain structure (cofibered domain)
- Implicit relations must be taken into account:

```
for(…) {
   i++;     <------------  i = j
   j++;
}
```

- Variables modified within a loop are put in the same pack

Outcomes

- The whole MER flight software can be analyzed in less than 24 hours
- The precision is over 80%
- Downsides of the approach:
  – Scalability is achieved at the price of a careful and complex engineering
  – There isn't much margin left to improve on the precision

Scalability <u>and</u> precision?

The gauge domain

- The domain of polyhedra is expressive enough but doesn't scale
- Weakly relational domains scale better (somewhat) but are not expressive enough
- Design a specialized domain for a certain type of invariants: the gauge domain
  - Focuses on finding implicit loop invariants among variables

From Intervals to Gauges

- Intuitively, a gauge is an integer interval that linearly varies across the iteration space
- Interval:

$$a \leq x \leq b$$

- Gauge:

$$a_0 + a_1\lambda_1 + \ldots + a_n\lambda_n \leq x \leq b_0 + b_1\lambda_1 + \ldots + b_n\lambda_n$$

  – $\lambda_1, \ldots, \lambda_n \geq 0$
  – $a_i \leq b_i$
  – The parameters λ_i denote the iteration counters of all enclosing loops

Exposing Loop Counters

- We label each loop with a fresh counter λ
- We introduce operations on the λ's to model the semantics of loop iterations

```
i = 0;
while (i < 10) {
  j = 0;
  while (j < i) {
    …;
    j++;
  }
  i++;
}
```

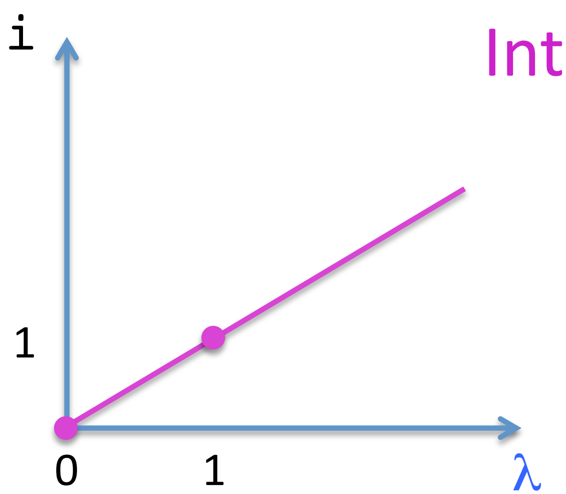
```
i = 0; new λ1
λ1: while (i < 10) {
  j = 0; new λ2
  λ2: while (j < i) {
    …;
    j++; inc λ2
  } forget λ2
  i++; inc λ1
} forget λ1
```

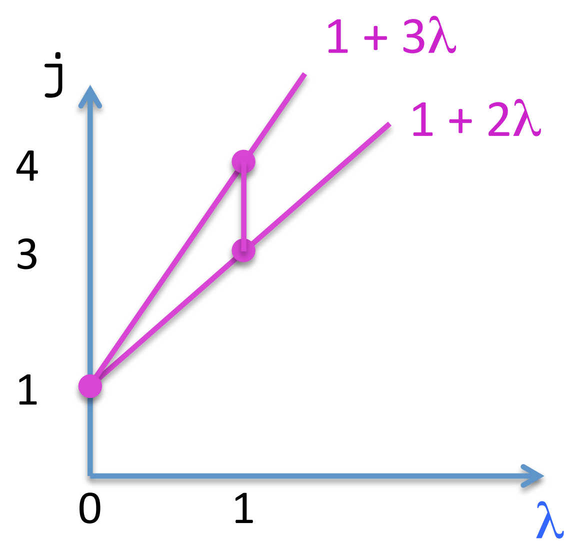
- This is an entirely automated process

How do we compute gauges?

```
    j = 1;
λ:  for (i = 0; i < 10; i++) {
      if (…) {
        j += 2;
      } else {
        j += 3;
      }
    }
```

Linear Interpolation





Computational Complexity

|Variables| = n

|Loop Depth| = k

- Joins and widenings: O(kn)
- Arithmetic operations: O(k)
- Loop operations (new, forget, inc): O(kn)
- If k is assumed bounded:
  - Linear complexity for domain operations
  - Constant complexity for semantic transformers
  - It is the same complexity as the domain of intervals

Experimental Results

- Buffer-overflow analysis performed on an intelligent flight control system developed at NASA
- 144 KLOC of C
- Complex adaptive avionics
- Analyses run on a laptop
  - Commercial tool: high-end server with 32 cores and 64GB memory

Analysis	Analysis Time	Precision
Intervals + Complete Inlining	41 min	79%
Commercial Tool	5 hours	91%
Octagons	> 27 hours	N/A
Gauges	10 min	91%

Unexpected benefits

- Some loops in the MATLAB/Simulink autocode have an unusual control structure:

```
p = &a[0];
i = 10;
while (i != 0) {
  *p++ = …;
  i--;
}
```

- This is bad for static analysis where only inequalities can be analyzed precisely T
  - The 1% not resolved by intervals

Gauges can help

- Relation between variables and loop counters

```
p = &a[0];
i = 10;
while (i != 0) {
   *p++ = …;
   i--;
}
```

`i` $= 10 - \lambda$

`p` $= 4\lambda$

- Since counters are monotonic and positive, we can automatically replace the test with `i > 0`
- **We obtain 100% precision**

Limitations of gauges

- The domain only provides information inside loops
  - The λ's are loop counters
- Outside of loops gauges are mere intervals
- Gauges have to be combined with other domains using the reduced product

D = Gauge × D1 × D2 × …

Perspectives

- There are many numerical domains available but few have been applied to real code
- We believe in combining simpler, specialized and efficient abstract domains over using a monolithic approach
- We are still a long way from being to able to automatically verify security-sensitive applications, even small ones