Cost Analysis with Recurrence Relations (using CiaoPP) and its Applications

Manuel Hermenegildo\textsuperscript{1,2} P. López-García\textsuperscript{1,3} M. Klemen\textsuperscript{1} U. Liqat\textsuperscript{1}

\textsuperscript{1}IMDEA Software Institute
\textsuperscript{2}T. U. of Madrid (UPM)
\textsuperscript{3}Spanish Research Council (CSIC)

Dagstuhl 17291 – Jul 17-21, 2017
Resource analysis

*Statically and automatically* infer upper/lower bounds on the usage that a program makes of resources.

- Much work on cost analysis (well represented here!) since Wegbreit75’s seminal work: Rosendahl’89, Debray/Lin/Hermenegildo’90, CASLOG, CiaoPP, COSTA, APProVe/KoAT, RAML, CoFloCo, LOOPUS, GUBS, Hammond, Barthe et al. (relational cost), Reps, ...(too long to list!).
  - And of course much related work in the WCET community.

- We are interested in a large class of resources (including *user-definable!*):
  - Execution steps, data sizes, time, memory, *energy* ...
  - Bits sent/received over a socket, SMSs, database accesses, procedure calls, files left open, money spent, ...

- Such resources can have different characteristics, e.g.:
  - platform-dependent vs. platform-independent,
  - cumulative vs. non-cumulative,
  - actual bounds or asymptotic (applications often need actual), ...
Resource analysis

*Statically and automatically* infer upper/lower bounds on the usage that a program makes of resources.

- Much work on cost analysis (well represented here!) since Wegbreit75’s seminal work: Rosendahl’89, Debray/Lin/Hermenegildo’90, CASLOG, CiaoPP, COSTA, APProVe/KoAT, RAML, CoFloCo, LOOPUS, GUBS, Hammond, Barthe et al. (relational cost), Reps, ...(too long to list!).
  - And of course much related work in the WCET community.

- We are interested in a large class of resources (including *user-definable!*):
  - Execution steps, data sizes, time, memory, **energy** ...
  - Bits sent/received over a socket, SMSs, database accesses, procedure calls, files left open, money spent, . . .

- Such resources can have different characteristics, e.g.:
  - platform-dependent vs. platform-independent,
  - cumulative vs. non-cumulative,
  - actual bounds or asymptotic (applications often need actual), . . .
Resource analysis

**Statically and automatically** infer upper/lower bounds on the usage that a program makes of resources.

- Much work on cost analysis (well represented here!) since Wegbreit75’s seminal work: Rosendahl’89, Debray/Lin/Hermenegildo’90, CASLOG, CiaoPP, COSTA, APProve/KoAT, RAML, CoFloCo, LOOPUS, GUBS, Hammond, Barthe et al. (relational cost), Reps, ...(too long to list!).
  - And of course much related work in the WCET community.

- We are interested in a large class of resources (including *user-definable*!):
  - Execution steps, data sizes, time, memory, **energy** ...
  - Bits sent/received over a socket, SMSs, database accesses, procedure calls, files left open, money spent, ... 

- Such resources can have different characteristics, e.g.:
  - platform-dependent vs. platform-independent,
  - cumulative vs. non-cumulative,
  - actual bounds or asymptotic (applications often need actual), ...
Resource analysis: applications

Granularity control in parallel programs

Probably the first practical application (GraCos/CiaoPP - [PLDI’90, JSC’96]):

- Sequentialize parallel tasks if not enough work.

\[
parallel\_cost \leq sequential\_cost \\
\max(cost_1, ..., cost_n) + \text{overheads} \leq \sum(cost_1, ..., cost_n)
\]

- Compare computing vs. communications costs (uses data size inference!).

\[
\text{if } (10 \times \text{length}(x) + 1 > 100) \\
y = \text{process\_collection}(x) \mid \mid m = r(z); \\
y = \text{process\_collection}(x); m = r(z); ...
\]

\[
\text{if } (\text{length}(x) > 10) ... \\
\text{if } (\text{length\_gt}(10) ...) \\
\rightarrow \text{not worth sending out for any data size.}
\]

- Scheduling, load balancing, ...

- Note that \textit{lower bounds} generally more useful here!
Resource analysis: applications

Granularity Control System Output Example (note data-size management!)

```
g_qsort([], []).
g_qsort([First|L1], L2) :-
    partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg),
    Size_Ls > 20 ->
        ( Size_Lg > 20 ->
            g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2)  % Run in parallel
            ; g_qsort(Ls, Ls2), s_qsort(Lg, Lg2) % Run sequentially
        ; ( Size_Lg > 20 ->
            s_qsort(Ls, Ls2), g_qsort(Lg, Lg2)
            ; s_qsort(Ls, Ls2), s_qsort(Lg, Lg2)) ), % Stop granularity control
    append(Ls2, [First|Lg2], L2).

partition3o4o(F, [], [], [], 0, 0).
partition3o4o(F, [X|Y], [X|Y1], Y2, SL, SG) :-
    X =< F, partition3o4o(F, Y, Y1, Y2, SL1, SG), SL is SL1 + 1.
partition3o4o(F, [X|Y], Y1, [X|Y2], SL, SG) :-
    X > F, partition3o4o(F, Y, Y1, Y2, SL, SG1), SG is SG1 + 1.
```
Resource analysis: applications

8 processors, no granularity control
Resource analysis: applications
8 processors, with granularity control (same scale)
Resource analysis: applications

Granularity control in parallel programs

Other program optimizations: partial evaluation, etc.

- Comparing the cost before and after an optimization step.
Resource analysis: applications

Granularity control in parallel programs

Other program optimizations: partial evaluation, etc.

Static performance verification/debugging/certification
- Static performance debugging, static profiling,
- Resource guarantees, resource proof carrying code, service quality certification (SLA), ...
Example in Ciao:

```prolog
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- entry nrev/2 : {list, ground} * var.

:- pred nrev(A,B) : list(A) => list(B) + (not_fails, is_det, steps_o(exp(length(A),2))).

nrev([] ) := [].
nrev([H|L] ) := conc( nrev(L),[H] ).

:- pred conc(A,_,_) + (terminates, is_det, steps_o(length(A))).

conc([], L ) := L.
conc([H|L], K ) := [ H | conc(L,K) ].
```
Example in Ciao (program after analysis and assertion checking):

```
:- checked comp nrev(A,B) : list(A)
    + ( not_fails, is_det, steps_o(exp(length(A),2)) ) .

:- true pred nrev(A,B) : ( list(A), var(B) )
    => ( list(A), list(B), size_lb(A,length(A)), size_ub(A,length(A)),
         size_lb(B,length(A)), size_ub(B,length(A)) )
    + ( steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1),
        steps_ub(0.5*exp(length(A),2)+1.5*length(A)+1),
        is_det, mut_exclusive, not_fails, covered ).

nrev([],[]).

nrev([H|L],_1) :- nrev(L,_2), conc(_2,[H],_1).

:- checked comp conc(A,_A,_B)
    + ( terminates, is_det, steps_o(length(A)) ) .

:- true pred conc(A,L,_A)
    : ( list(A), rt3(L), var(_A) )
    => ( list(A), non_empty_list(L), non_empty_list(_A),
         size_lb(A,length(A)), size_lb(L,length(L)),
         size_lb(_A,length(L)+length(A)), size_ub(A,length(A)),
         size_ub(L,length(L)), size_ub(_A,length(L)+length(A)) )
    + ( steps_lb(length(A)+1), steps_ub(length(A)+1),
        is_det, mut_exclusive, not_fails, covered ).

conc([],L,L).

conc([H|L],K,[H|_1]) :- conc(L,K,_1).

:- regtype rt3/1.    rt3([A]) :- term(A).
```
XC Program (FIR Filter) w/Energy Specification [HIP3ES’15]

```
#pragma check fir(xn, coeffs, state, N) :
    (1 <= N) ==> (energy <= 416079189)

#pragma true fir(xn, coeffs, state, N) :
    (3347178*N + 13967829 <= energy &&
     energy <= 3347178*N + 14417829)

#pragma checked fir(xn, coeffs, state, N) :
    (1 <= N && N <= 120) ==> (energy <= 416079189)

#pragma false fir(xn, coeffs, state, N) :
    (121 <= N) ==> (energy <= 416079189)
```

```c
int fir(int xn, int coeffs[], int state[], int ELEMENTS) {
    unsigned int ynl; int ynh;
    ynl = (1<<23); ynh = 0;
    for(int j=ELEMENTS-1; j!=0; j--) {
        state[j] = state[j-1];
        {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
    }
    state[0] = xn;
    {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
    if (sext(ynh,24) == ynh) {
        ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
    else if (ynh < 0) { ynh = 0x80000000; }
    else { ynh = 0x7fffffff; }
    return ynh;
}
```
XC Program (FIR Filter) w/Energy Specification [HIP3ES’15]

```c
#pragma check fir(xn, coeffs, state, N) :
   (1 <= N) ==> (energy <= 416079189)

#pragma true fir(xn, coeffs, state, N) :
   (3347178*N + 13967829 <= energy &&
    energy <= 3347178*N + 14417829)

#pragma checked fir(xn, coeffs, state, N) :
   (1 <= N && N <= 120) ==> (energy <= 416079189)

#pragma false fir(xn, coeffs, state, N) :
   (121 <= N) ==> (energy <= 416079189)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
    unsigned int ynl; int ynh;
    ynl = (1<<23); ynh = 0;
    for(int j=ELEMENTS-1; j!=0; j--) {
        state[j] = state[j-1];
        {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
    }
    state[0] = xn;
    {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
    if (sext(ynh,24) == ynh) {
        ynh = (ynh << 8) | (((unsigned) ynl) >> 24);
    } else if (ynh < 0) { ynh = 0x80000000; }
    else { ynh = 0x7fffffff; }
    return ynh;
}
```

Hermenegildo, Lopez-Garcia, Klemen, Liqat
Cost Analysis with Recurrence Relations in CiaoPP
Dagstuhl 17291 Jul 17-21'17 10 / 117
Resource analysis: applications

Granularity control in parallel programs

Other program optimizations: partial evaluation, etc.

Static performance verification/debugging/certification

Security

- Proof of absence of side channels (time, memory, energy, ...).
Resource analysis: applications

Granularity control in parallel programs

Other program optimizations: partial evaluation, etc.

Static performance verification/debugging/certification

Security

Database query optimization

Etc.
Some key observations about resource analysis

In the context of these applications

1. Undecidable!
   - Not always possible to infer exactly
     → infer bounds that are safe and also as accurate as possible.
   - I.e., best and worst cases (mean and variance would also be desirable).
   - Abstract interpretation useful for reasoning about safety.

2. Dependent on input data metrics
   - infer the bounds as functions of input data sizes.
   - Examples; list length, array dimensions, numerical values, ....
     \[
     cost_{process\_collection}(x) = 10 \times length(x) + 1
     \]
     ```
     \{ ...; if (10*length(x)+1 > 100) 
     \hspace{1cm} y = process\_collection(x) || m = r(z); ... \}
     ```
   - (Alternative: bound loops)

3. Typically bound functions need to be inferred on the data sizes themselves.
   ```
   f(x,y){... x=create\_collection(m); y=process\_collection(x); ...}
   ```
Some key observations about resource analysis

In the context of these applications

1. Undecidable!
   - Not always possible to infer exactly
     → infer **bounds** that are safe and also as accurate as possible.
   - I.e., best and worst cases (mean and variance would also be desirable).
   - Abstract interpretation useful for reasoning about safety.

2. Dependent on input data metrics
   → infer the bounds as functions of input data sizes.
   - Examples; list length, array dimensions, numerical values, ....

\[
\text{cost}_{\text{process\_collection}}(x) = 10 \times \text{length}(x) + 1
\]

```plaintext
{ ...; if (10*\text{length}(x)+1 > 100)
    \text{y} = \text{process\_collection}(\text{x}) \mid \mid \text{m} = \text{r(z)}; ... }
```

- (Alternative: bound loops)

3. Typically bound functions need to be inferred on the data sizes themselves.

\[
\text{f}(x,y)\{... \text{x}=\text{create\_collection}(\text{m}); \text{y}=\text{process\_collection}(\text{x}); ...\}
\]
Some key observations about resource analysis

In the context of these applications

1 Undecidable!
   ▶ Not always possible to infer exactly
     \( \rightarrow \) infer bounds that are safe and also as accurate as possible.
   ▶ I.e., best and worst cases (mean and variance would also be desirable).
   ▶ Abstract interpretation useful for reasoning about safety.

2 Dependent on input data metrics
   \( \rightarrow \) infer the bounds as functions of input data sizes.
   ▶ Examples; list length, array dimensions, numerical values, ....
     \( \text{cost}_{\text{process\_collection}}(x) = 10 \times \text{length}(x) + 1 \)

     \{ \ldots; \text{if } (10 \times \text{length}(x)+1 > 100) \}
     \hspace{1cm} y = \text{process\_collection}(x) \text{ || } m = r(z); \ldots \}

   ▶ (Alternative: bound loops)

3 Typically bound functions need to be inferred on the data sizes themselves.

\( f(x, y) \{ \ldots \text{x=create\_collection}(m); y=\text{process\_collection}(x); \ldots \} \)
Using an Intermediate Representation

- Some current systems support several languages by translation to a kernel representation. E.g.: rules (in CiaoPP Horn clauses \[^{[\text{LOPSTR'07}]}\]).

→ Allows making analysis largely language-independent.

► Of course, languages have unique features (e.g.: higher order, backtracking, lazy evaluation, concurrency, ...) which need to be modeled/supported.

► But IR centers attention around *language features* rather than languages—concrete languages are a collage of such features.

- Horn clause-based block representation (cf. *Abstract Compilation* \[^{[\text{ICLP'88}]}\]):

  ► Source: Program P in \(L_P\) + (possibly abstract) Semantics of \(L_P\)

  ► Target: A (C) Horn Clause program capturing \([P]\) (or, possibly, \([P]^{\alpha}\))

The HC IR:

► Reduces everything to recursions, which simplifies analysis.

► Makes the semantics of loops, recursions, etc. precise: all variables, increments/decrements, scoping, etc. is explicit.

► Advantage over just syntactic blocks: *captures both procedural and declarative/denotational semantics* (model of the Horn clauses implies properties of the program).
Using an Intermediate Representation

- Some current systems support several languages by translation to a kernel representation. E.g.: rules (in CiaoPP Horn clauses \[^{LOPSTR'07}\]).
  → Allows making analysis largely language-independent.
  - Of course, languages have unique features (e.g.: higher order, backtracking, lazy evaluation, concurrency, ...) which need to be modeled / supported.
  - But IR centers attention around *language features* rather than languages –concrete languages are a collage of such features.

- Horn clause-based block representation (cf. *Abstract Compilation* \[^{ICLP'88}\]):
  - **Source:** Program P in \(L_{P} +\) (possibly abstract) Semantics of \(L_{P}\)
  - **Target:** A (C) Horn Clause program capturing \([P]\) (or, possibly, \([P]^{\alpha}\))

The HC IR:
  - Reduces everything to recursions, which simplifies analysis.
  - Makes the semantics of loops, recursions, etc. precise: all variables, increments/decrements, scoping, etc. is explicit.
  - Advantage over just syntactic blocks: *captures both procedural and declarative/denotational semantics* (model of the Horn clauses implies properties of the program).
Intermediate Repr.: (Constraint) Horn Clauses (CiaoPP)

Transformation:
- **Source:** Program $P$ in $L_P$ + (possibly abstract) Semantics of $L_P$
- **Target:** A (C) Horn Clause program capturing $\llbracket P \rrbracket$ (or, possibly, $\llbracket P \rrbracket^\alpha$)

- Block-based CFG. Each block represented as a *Horn clause.*
- Used for all analyses: aliasing, CHA/shape/types, data sizes, resources, etc.
- Allows supporting multiple languages.

- **Transformation:**
  - Source: Program $P$ in $L_P$ + (possibly abstract) Semantics of $L_P$
  - Target: A (C) Horn Clause program capturing $\llbracket P \rrbracket$ (or, possibly, $\llbracket P \rrbracket^\alpha$)

- Block-based CFG. Each block represented as a *Horn clause.*
- Used for all analyses: aliasing, CHA/shape/types, data sizes, resources, etc.
- Allows supporting multiple languages.
Transformation example - source

Adding numbers from 1 to n

```java
public static int r(int n) {
    int ans = 0;
    while (n > 0) {
        ans += n;
        n--; // n-=1;
    }
    return ans;
}
```
Transformation example - source

Adding numbers from 1 to n

source

```java
public static int r(int n) {
    int ans=0;
    while(n>0) {
        ans+=n;
        n-=1;
    }
    return ans;
}
```

Horn clause IR

```prolog
:- pred r/2 : num * var.

r(N,Ret) :-
    Ans=0,
    r_1(N, Ans, Ret).

r_1(N, Ans, Ret) :-
    N>0,
    add(Ans, N, Ans1),
    N1 is N - 1,
    r_1(N1, Ans1, Ret).

r_1(N, Ans, Ret):- 
    N=<0,
    Ret=Ans.
```
Xcore ISA Example: Control Flow Graph (CFG)

<fact>:
0x01: entsp (u6) 0x2
0x02: stw (ru6) r0, sp[0x1]
0x03: ldw (ru6) r1, sp[0x1]
0x04: ldc (ru6) r0, 0x0
0x05: lss (3r) r0, r0, r1
0x06: bf (ru6) r0, 0x1 <0x08>
0x07: bu (u6) 0x2 <0x10>
0x08: mkmsk (rus) r0, 0x1
0x09: retsp (u6) 0x2
0x10: ldw (ru6) r0, sp[0x1]
0x11: sub (2rus) r0, r0, 0x1
0x12: bl (u10) -0xc <fact>
0x13: ldw (ru6) r1, sp[0x1]
0x14: mul (13r) r0, r1, r0
0x15: retsp (u6) 0x2
Xcore ISA Example: Block Representation

<fact>
0x01: entsp (u6) 0x2
0x02: stw (ru6) r0, sp[0x1]
0x03: ldw (ru6) r1, sp[0x1]
0x04: ldc (ru6) r0, 0x0
0x05: lss (3r) r0, r0, r1
0x06: bf (ru6) r0, 0x1 <0x08>
0x07: bu (u6) 0x2 <0x10>
0x10: ldw (ru6) r0, sp[0x1]
0x11: sub (2rus) r0, r0, 0x1
0x12: bl (u10) -0xc <fact>
0x13: ldw (ru6) r1, sp[0x1]
0x14: mul (l3r) r0, r1, r0
0x15: retsp (u6) 0x2
0x08: mkmsk (rus) r0, 0x1
0x09: retsp (u6) 0x2
:- entry fact/2.

fact(R0,R0_3):-
    entsp(_,)
    stw(R0,Sp0x1),
    ldw(R1,Sp0x1),
    ldc(R0_1,0x0),
    lss(R0_2,R0_1,R1),
    bf(R0_2, _),
    bf01(R0_2,Sp0x1,R0_3,R1_1).

bf01(1,Sp0x1,R0_4,R1):-
    bu(_),
    ldw(R0_1,Sp0x1),
    sub(R0_2,R0_1,0x1),
    bl(_),
    fact(R0_2,R0_3),
    ldw(R1,Sp0x1),
    mul(R0_4,R1,R0_3),
    retsp(_).

bf01(0,Sp0x1,R0,R1):-
    mkmsk(R0,0x1),
    retsp(_).
Example: Java Bytecode

```java
@Resources({Resource.STEPS})
public class Fact {

    public int factorial(int n) {
        if (n == 0)
            return 1;
        else
            return n * factorial(n - 1);
    }
}
```

```
Fact.factorial(Ret,This,N)
Builtin.eq(void, N,0)
Builtin.asg(Ret,1)

Fact.factorial(Ret,This,N)
Builtin.ne_int(void,N,0)
Builtin.sub(I1,N,1)
Fact.factorial(I2, This, I1)
Builtin.mul(I3,N,I2)
Fact.factorial(Ret,This,N)
Builtin.asg(Ret,1)
```
Example: Java Bytecode

```
:- entry 'Fact.factorial'/3:var*atm*num.
:- resource 'STEPS'.
'Fact.factorial'(Ret, This, N):-
  eq_int(void,N,int,0,int),
  asg_int(Ret,int,1,int).
'Fact.factorial'(Ret, This, N):-
  ne_int(void,N,int,0,int),
  sub(I1,int,N,int,1,int),
  Fact.factorial(I2,This,I1),
  mul(I3, int,N,int,I2,int),
  asg_int(Ret,int,I3,int).
```

- Annotations (since Java 1.5) are preserved in the bytecode so they can be carried over to our IR.
Example: Java Bytecode

```
:- entry 'Fact.factorial'/3:var*atm*num.
:- resource 'STEPS'.
'Fact.factorial'(Ret, This, N):-
  eq_int(void,N,0,int),
  asg_int(Ret,int,1,int).

'Fact.factorial'(Ret, This, N):-
  ne_int(void,N,0,int),
  sub(I1,int,N,1,int),
  Fact.factorial(I2,This,I1),
  mul(I3,int,N,int,I2,int),
  asg_int(Ret,int,I3,int).
```

- Annotations (since Java 1.5) are preserved in the bytecode so they can be carried over to our IR.
Generating the Intermediate Representation

- **Typical tasks:**
  - Generation of block-based CFG.
  - SSA transformation (e.g., splitting of input/output param).
  - Conversion of loops into recursions among blocks.
  - Branching, cases, dynamic dispatch → blocks w/same signature.

- **Some specifics for Java:**
  - Control flow graph is constructed from bytecode.
  - Elimination of stack variables.
  - Conversion to three-address statements.
  - Explicit representation of this and ret as extra block parameters.

- **Some specifics for XC:**
  - Control flow graph is constructed from ISA or LLVM IR representation.
  - Inferring block parameters.
  - Resolving branching to predicates with multiple clauses.

- **Can be done via:**
  - **partial evaluation of an interpreter** (implementing the semantics of the low-level code) w.r.t. the concrete low-level program or
  - **directly** (cf. Futamura projections).
Resource Analysis (CiaoPP)

Java Source → javac → Java Bytecode

Java parser

soot + Ciao transform.

Ciao Source

ISA / LLVM / ...

xcc / clang / Ciao

Java Bytecode

Transformations

IR − CFG (Horn clauses)

Analysis

(AAbstract Interpretation)

Domains

Sharing

Shapes/sizes

Resources

Fixpoint algorithm

Sets of Pre/Post pairs (prog. point info)

(incl. size and resource usage functions)

Assertions (incl. resource models)

[PLDI’90, PASCO’94, JSC’96, SAS’94, ILPS’97, ICLP’07, LOPSTR’07, PPDP’08, NASA FM’08]

[Bytecode’09, ICLP’10, ICLP’13, LOPSTR’13, TPLP’14] [FOPARA’15, HIP3ES’16] [TPLP’16, FLOPS’16]
CiaoPP Resource Analysis

- The objective of the resource analysis is to obtain for each predicate/block call - resource usage function pairs:
  - Arithmetic functions providing lower/upper bounds on the resource usage of the predicate/block given the sizes of its input data for a particular entry condition.

Example

```ciao
#pragma true nrev(x) : list(x) =>
   ( 0 <= resource(energy) && resource(energy) <= 1+length(x)**2 )
```

- `x` points to a list → energy consumed ≤ 1 + `length(x)²`

1. Programmer defines the resource consumption for basic elements (e.g., instructions, bytecodes, libraries, …) – the “cost model.”
2. System infers resource usage bound functions for rest of program. (Can be polynomial, exponential, logarithmic, …)
CiaoPP Resource Analysis

The objective of the resource analysis is to obtain for each predicate/block call - resource usage function pairs:
- Arithmetic functions providing lower/upper bounds on the resource usage of the predicate/block given the sizes of its input data for a particular entry condition.

Example

```cpp
#pragma true nrev(x) : list(x) ==> 
    ( 0 <= resource(energy) && resource(energy) <= 1+length(x)*2 )
```

- x points to a list → energy consumed ≤ 1 + length(x)^2

1. Programmer defines the resource consumption for basic elements (e.g., instructions, bytecodes, libraries, ...) – the “cost model.”
2. System infers resource usage bound functions for rest of program. (Can be polynomial, exponential, logarithmic, ...)

1. Programmer defines the resource consumption for basic elements (e.g., instructions, bytecodes, libraries, ...) – the “cost model.”
2. System infers resource usage bound functions for rest of program. (Can be polynomial, exponential, logarithmic, ...)

Programmer defines the resource consumption for basic elements (e.g., instructions, bytecodes, libraries, ...) – the “cost model.”
System infers resource usage bound functions for rest of program. (Can be polynomial, exponential, logarithmic, ...)

User-definable aspects of the analysis [ICLP'07, Bytecode'09]

- A cost model defines an upper/lower bound cost for primitive operations (e.g., methods, bytecode instructions).
  - Provided by the user, also via the assertion language.
    
    ```java
    @Cost("cents","2*size(data)"
    public native void Stream.send(java.lang.String data);
    ```
  - Some predefined in system libraries.

For platform-dependent resources such as execution time or energy consumption model needs to consider low level factors.

- Assertions:
  - Also used to provide other inputs to the resource analysis such as argument sizes, size metrics, etc. if needed.
  - Also allow improving the accuracy and scalability of the system.
  - Output of resource analysis also expressed via assertions.
  - Used additionally to state resource-related specifications which allows finding bugs, verifying, certifying, etc.
A cost model defines an upper/lower bound cost for primitive operations (e.g., methods, bytecode instructions).

- Provided by the user, also via the assertion language.

```java
@Cost("cents","2*size(data)")
public native void Stream.send(java.lang.String data);
```

- Some predefined in system libraries.

For platform-dependent resources such as execution time or energy consumption model needs to consider low level factors.

Assertions:

- Also used to provide other inputs to the resource analysis such as argument sizes, size metrics, etc. if needed.
- Also allow improving the accuracy and scalability of the system.
- Output of resource analysis also expressed via assertions.
- Used additionally to state resource-related specifications which allows finding bugs, verifying, certifying, etc.
Resource Usage-Related Assertions

- `resource ⟨resource_name⟩.`
- `head_cost(⟨approx⟩,⟨resource_name⟩,⟨arith_expression⟩)).`
- `literal_cost(⟨approx⟩,⟨resource_name⟩,⟨arith_expression⟩)).`
- `trust ... + cost(⟨approx⟩,⟨resource_name⟩,⟨arith_expr⟩)).`

Example

- `resource steps.`
- `head_cost(ub,steps, 1).`
- `literal_cost(ub,steps, 0).`
- `trust pred append(X,Y,Z) : ( list(X), list(Y), var(Z) ) + cost(ub,steps,length(X)+1).`
Examples of resource definitions and analysis

Adding numbers from 1 to n (numeric loop)

```java
public static int r(int n)
{
    int ans=0;
    while(n>0)
    {
        ans+=n;
        n-=1;
    }
    return ans;
}
```

```plaintext
:- resource nadds.

:- true pred add(A,B,C)
  : ( num(A), num(B), var(C) )
  => ( num(A), num(B), num(C),
       rsize(A, num(AL, AU)),
       rsize(B, num(BL, BU)),
       rsize(C, num(AL+BL, AU+BU))
       + ( cardinality(1,1),
           costb(nadds, 1, 1) ).
```

Horn clause IR

```prolog
:- pred r/2 : num * var.

r(N, Ret) :-
    Ans=0,
    r_1(N, Ans, Ret).

r_1(N, Ans, Ret) :-
    N>0,
    add(Ans, N, Ans1),
    N1 is N - 1,
    r_1(N1, Ans1, Ret).

r_1(N, Ans, Ret) :-
    N=<0,
    Ret=Ans.
```

Output (in IR)

```prolog
:- resource bits_sent.

:- pragma ...(resource decls)...
```

source
Examples of resource definitions and analysis

Multiplying numbers from 1 to n (numeric loop)

```c
int r(int n)
{
    int ans=1;
    while(n>0)
    {
        ans*=n;
        n--;
    }
    return ans;
}
```

Horn clause IR:

```prolog
:- entry r/2 : num*var.

r(N,Ret):-
   Ans=1,
   while(N, Ans, Ret).

while(N, Ans, Ret):-
   N>0,
   mul(Ans1, Ans, N),
   N1 is N - 1,
   while(N1, Ans1, Ret).
while(N, Ans, Ret):-
   N=<0,
   Ret=Ans.
```

Output (in IR):

```prolog
:- true pred r(N,Ret)
   : ( num(N), var(Ret) )
   => ( num(N), num(Ret),
        rsize(N,num(A,B)),
        rsize(Ret,num(fact(A),fact(B))) )
   + ( not_fails, is_det,
       cardinality(1,1),
       costb(mults,1,1) ).
```
### Some Examples of Resources and Resource Functions Inferred [ICLP’07]

<table>
<thead>
<tr>
<th>Program</th>
<th>Resource</th>
<th>Usage Function</th>
<th>Metrics</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>“bits received”</td>
<td>( \lambda x \cdot 8 \cdot x )</td>
<td>length</td>
<td>186</td>
</tr>
<tr>
<td>color_map</td>
<td>“unifications”</td>
<td>39066</td>
<td>size</td>
<td>176</td>
</tr>
<tr>
<td>copy_files</td>
<td>“files left open”</td>
<td>( \lambda x \cdot x )</td>
<td>length</td>
<td>180</td>
</tr>
<tr>
<td>eight_queen</td>
<td>“queens movements”</td>
<td>19173961</td>
<td>length</td>
<td>304</td>
</tr>
<tr>
<td>eval_polynom</td>
<td>“FPU usage”</td>
<td>( \lambda x \cdot 2.5x )</td>
<td>length</td>
<td>44</td>
</tr>
<tr>
<td>fib</td>
<td>“arith. operations”</td>
<td>( \lambda x \cdot 2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3 )</td>
<td>value</td>
<td>116</td>
</tr>
<tr>
<td>grammar</td>
<td>“phrases”</td>
<td>24</td>
<td>length/size</td>
<td>227</td>
</tr>
<tr>
<td>hanoi</td>
<td>“disk movements”</td>
<td>( \lambda x \cdot 2^x - 1 )</td>
<td>value</td>
<td>100</td>
</tr>
<tr>
<td>insert_stores</td>
<td>“accesses Stores”</td>
<td>( \lambda n, m \cdot n + k )</td>
<td>length</td>
<td>292</td>
</tr>
<tr>
<td>perm</td>
<td>“bytecode instructions”</td>
<td>( \lambda x \cdot (\sum_{i=1}^{x} 18 \cdot x!) + (\sum_{i=1}^{x} 14 \cdot \frac{x!}{i!}) + 4 \cdot x! )</td>
<td>length</td>
<td>98</td>
</tr>
<tr>
<td>power_set</td>
<td>“output elements”</td>
<td>( \lambda x \cdot \frac{1}{2} \cdot 2^{x+1} )</td>
<td>length</td>
<td>119</td>
</tr>
<tr>
<td>qsort</td>
<td>“lists parallelized”</td>
<td>( \lambda x \cdot 4 \cdot 2^x - 2x - 4 )</td>
<td>length</td>
<td>144</td>
</tr>
<tr>
<td>send_files</td>
<td>“bytes read”</td>
<td>( \lambda x, y \cdot x \cdot y )</td>
<td>length/size</td>
<td>179</td>
</tr>
<tr>
<td>subst_exp</td>
<td>“replacements”</td>
<td>( \lambda x, y \cdot 2xy + 2y )</td>
<td>size/length</td>
<td>153</td>
</tr>
<tr>
<td>zebra</td>
<td>“steps”</td>
<td>30232844295713061</td>
<td>size</td>
<td>292</td>
</tr>
</tbody>
</table>

- Different complexity functions, resources, size metrics, types of loops/recursion, etc.
public class CellPhone {

    void sendSms(SmsPacket smsPk, Encoder enc, Stream stm) {
        if (smsPk != null) {
            stm.send(enc.format(smsPk.sms));
            sendSms(smsPk.next, enc, stm);
        }
    }

    class SmsPacket{
        String sms;
        SmsPacket next;
    }

    abstract class Stream{
        @Cost({"cents", "2*size(data)"})
        native void send(String data);
    }

    interface Encoder{
        String format(String data);
    }

    class TrimEncoder implements Encoder{
        @Cost({"cents", "0"})
        @Size("size(ret)<=size(s)")
        public String format(String s){
            return s.trim();
        }
    }

    class UnicodeEncoder implements Encoder{
        @Cost({"cents", "0"})
        @Size("size(ret)<=6*size(s)")
        public String format(String s){
            return java.net.URLEncoder.encode(s);
        }
    }

    Goal is to infer a safe estimate of the cost of sending \( n \) SMSs, assuming that the carrier charges 2 cents per character.
Annotations (since Java 1.5) are preserved in the bytecode so they can be carried over to our IR.
### Some results (Java) [Bytecode'09]

<table>
<thead>
<tr>
<th>Program</th>
<th>Resource(s)</th>
<th>t</th>
<th>Resource Usage Func. (*) / Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>BST</td>
<td>Heap usage</td>
<td>367</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n \equiv$ tree depth</td>
</tr>
<tr>
<td>CellPhone</td>
<td>SMS monetary cost</td>
<td>386</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n \equiv$ packets length</td>
</tr>
<tr>
<td>Client</td>
<td>Bytes received and</td>
<td>527</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>Bandwidth required</td>
<td></td>
<td>$n \equiv$ stream length</td>
</tr>
<tr>
<td>Dhrystone</td>
<td>Energy consumption</td>
<td>759</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n \equiv$ int value</td>
</tr>
<tr>
<td>Divbytwo</td>
<td>Stack usage</td>
<td>219</td>
<td>$O(\log_2(n))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n \equiv$ int value</td>
</tr>
<tr>
<td>Files</td>
<td>Files left open and</td>
<td>649</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>Data stored</td>
<td></td>
<td>$n \equiv$ number of files</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$m \equiv$ stream length</td>
</tr>
<tr>
<td>Join</td>
<td>DB accesses</td>
<td>460</td>
<td>$O(n \times m)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n, m \equiv$ table records</td>
</tr>
<tr>
<td>Screen</td>
<td>Screen width</td>
<td>536</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n \equiv$ stream length</td>
</tr>
</tbody>
</table>

- Again, different complexity functions, resources, size metrics, types of loops/recursion, etc.

(*) We represent just order for brevity (but full upper and lower bounds inferred).
Demo!

We will be putting the examples used in the demo in the CiaoPP playground.

(http://play.ciao-lang.org)
Analysis: CiaoPP Parametric AI Framework

- Analysis *parametric* w.r.t. abstractions, resources, ... (and languages).
- Efficient fixpoint algorithm for (C)HC IR.

\[\text{JLP'92, POPL'94, TOPLAS'99, SAS'96, TOPLAS'00, FTfJP'07}\]
\[\text{NAACL'89, ICLP'91, ICLP'97, SAS'02, FLOPS'04, LOPSTR'04, PADL'06, PASTE'07}\]
\[\text{VMCAI'08, LCPC'08, PASTE'08, CC'08, ISMM'09, NGC'10, LCPC'08}\]
Efficient, Parametric Fixpoint Algorithm

- **Generic framework** for implementing HC-based analyses:
  given $P$ (as a set of HCs) and abstract domain(s),
  computes $\text{lfp}(S^\alpha_P) = \llbracket P \rrbracket^\alpha$, s.t. $\llbracket P \rrbracket^\alpha$ safely approximates $\llbracket P \rrbracket$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC’s.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{\text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}}\}$.
    - Exit states for calls to $\text{block}$ satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
  - **A dependency arc table**: $\{A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B}\}$.
    - Answers for call $A : \lambda_{\text{in}A}$ depend on the answers for $B : \lambda_{\text{in}B}$:
      (if exit for $B : \lambda_{\text{in}B}$ changes, exit for $A : \lambda_{\text{in}A}$ possibly also changes).
    - $\text{Dep}(B : \lambda_{\text{in}B}) = \text{the set of entries depending on } B : \lambda_{\text{in}B}$.

- Characteristics:
  - **Precision**: context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency**: memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity**: abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

[NACLP’89, JLP’92, POPL’94, SAS’96, TOPLAS’00, FTfJP’07]
Efficient, Parametric Fixpoint Algorithm

- **Generic framework** for implementing HC-based analyses: given $P$ (as a set of HCs) and abstract domain(s), computes $\text{lfp}(S^\alpha_P) = [P]_\alpha$, s.t. $[P]_\alpha$ safely approximates $[P]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC’s.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{\text{block_\lambda : } \lambda_{\text{in}} \mapsto \lambda_{\text{out}}\}$.
    - Exit states for calls to block satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
  - **A dependency arc table**: $\{A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B}\}$.
    - Answers for call $A : \lambda_{\text{in}A}$ depend on the answers for $B : \lambda_{\text{in}B}$:
      (if exit for $B : \lambda_{\text{in}B}$ changes, exit for $A : \lambda_{\text{in}A}$ possibly also changes).
    - $\text{Dep}(B : \lambda_{\text{in}B}) = \text{the set of entries depending on } B : \lambda_{\text{in}B}$.

- Characteristics:
  - **Precision**: context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency**: memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity**: abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and incremental.
  - Handles library calls, externals, ...

[NACLP’89, JLP’92, POPL’94, SAS’96, TOPLAS’00, FTfJP’07]
Efficient, Parametric Fixpoint Algorithm

- **Generic framework** for implementing HC-based analyses: given $P$ (as a set of HCs) and abstract domain(s), computes $\text{lfp}(S_P^\alpha) = [P]_\alpha$, s.t. $[P]_\alpha$ safely approximates $[P]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC’s.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries \{block : $\lambda_{in} \mapsto \lambda_{out}$\}.
    - Exit states for calls to block satisfying precond $\lambda_{in}$ meet postcond $\lambda_{out}$.
  - **A dependency arc table**: \{A : $\lambda_{inA} \Rightarrow B : \lambda_{inB}$\}.
    - Answers for call A : $\lambda_{inA}$ depend on the answers for B : $\lambda_{inB}$:
      (if exit for B : $\lambda_{inB}$ changes, exit for A : $\lambda_{inA}$ possibly also changes).
    - $\text{Dep}(B : \lambda_{inB}) = \text{the set of entries depending on } B : \lambda_{inB}$.

- Characteristics:
  - **Precision**: context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency**: memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity**: abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

[NAACL’89, JLP’92, POPL’94, SAS’96, TOPLAS’00, FTtJP’07]
Efficient, Parametric Fixpoint Algorithm

- **Generic framework** for implementing HC-based analyses:
  given $P$ (as a set of HCs) and abstract domain(s),
  computes $\text{lfp}(S^\alpha_P) = [P]^\alpha$, s.t. $[P]^\alpha$ safely approximates $[P]$.

  → Essentially efficient, incremental, abstract OLDT resolution of HC’s.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{\text{block} : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to $\text{block}$ satisfying precond $\lambda_{in}$ meet postcond $\lambda_{out}$.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$:
      (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $\text{Dep}(B : \lambda_{inB}) = \text{the set of entries depending on } B : \lambda_{inB}$.

- Characteristics:
  - **Precision**: context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency**: memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity**: abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

[NACLP'89, JLP'92, POPL'94, SAS'96, TOPLAS'00, FTfJP'07]
Overview of the Analysis ("classical" view)

1. Convert to intermediate representation: iterations to recursions, etc.
2. Perform all the required supporting analyses (examples):
   - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.
3. Set up recurrence equations representing the size of each (relevant) output argument as a function of the input data sizes.
   - Size metrics are derived from inferred shape (type) information.
   - Data dependency graphs determine relative sizes of variable contents.
4. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

[PLDI'90, SAS'94, PASCO'94]
Overview of the Analysis ("classical" view)

1. Convert to intermediate representation: iterations to recursions, etc.

2. Perform all the required supporting analyses (examples):
   - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.

3. Set up recurrence equations representing the size of each (relevant) output argument as a function of the input data sizes.
   - Size metrics are derived from inferred shape (type) information.
   - Data dependency graphs determine relative sizes of variable contents.

4. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

[PLDI’90, SAS’94, PASCO’94]
Overview of the Analysis ("classical" view)

1. Convert to intermediate representation: iterations to recursions, etc.
2. Perform all the required supporting analyses (examples):
   - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.
3. Set up recurrence equations representing the size of each (relevant) output argument as a function of the input data sizes.
   - Size metrics are derived from inferred shape (type) information.
   - Data dependency graphs determine relative sizes of variable contents.
4. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

[PLDI'90, SAS'94, PASCO'94]
Overview of the Analysis ("classical" view)

1. Convert to intermediate representation: iterations to recursions, etc.

2. Perform all the required supporting analyses (examples):
   - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.

3. Set up recurrence equations representing the size of each (relevant) output argument as a function of the input data sizes.
   - Size metrics are derived from inferred shape (type) information.
   - Data dependency graphs determine relative sizes of variable contents.

4. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

5. E.g.: #pragma true conc(x,y) : list * list
   
   ==> ( length(x)+length(y) <= size(ret) && size(ret) <= length(x)+length(y) )

[PLDI'90, SAS'94, PASCO'94]
Overview of the Analysis ("classical" view)

1. Convert to intermediate representation: iterations to recursions, etc.
2. Perform all the required supporting analyses (examples):
   - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.
3. Set up recurrence equations representing the size of each (relevant) output argument as a function of the input data sizes.
   - Size metrics are derived from inferred shape (type) information.
   - Data dependency graphs determine relative sizes of variable contents.
4. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.
5. Use the size information to set up recurrence equations representing the computational cost of each block and compute bounds to their solutions to obtain resource usage functions.

[PLDI'90, SAS'94, PASCO'94]
Overview of the Approach

Data Dependency Analysis

Program + Assertions

Abstract Interpreter (Types, Modes, and Non-Failure)

Assertion Processing

Size Analysis

Resource Analysis

Cost Functions

Data Depend. Graph

Metrics/Modes

Size/Metrics assertions

Non-Failure

Resource assertions

Size equations

Solved size relations

Diff. Eqs. Solver

Resource equations
Size Metrics

Various size metrics can be used to determine the size of an input:

- the actual value of a number,
- the length of a list,
- the size (number of constant and function symbols) in a structure (term),
- etc.

These are automatically inferred based on type (shape) analysis and other information (program control flow and operations).

The function $size_m(t)$ defines the size of a structure $t$ under the metric $m$:

- $size_{\text{length}}([4, 2, 7]) = 3$
- $size_{\text{length}}([]) = 0$
- $size_{\text{term\_depth}}(f(a, g(b))) = 2$

The function $diff_m(t_1, t_2)$ gives the size difference between two terms $t_1$ and $t_2$ under the metric $m$:

- $diff_{\text{length}}([2, 3|L], [4|L]) = 1$
- $diff_{\text{length}}(L, [H|L]) = -1$
- $diff_{\text{term\_depth}}(f(a, g(X)), X) = 2$
Size Metrics

- Various size metrics can be used to determine the size of an input:
  - the actual value of a number,
  - the length of a list,
  - the size (number of constant and function symbols) in a structure (term),
  - etc.

- These are automatically inferred based on type (shape) analysis and other information (program control flow and operations).

- The function $size_m(t)$ defines the size of a structure $t$ under the metric $m$:
  \[
  size_{length}([4, 2, 7]) = 3 \\
  size_{length}([]) = 0 \\
  size_{term\_depth}(f(a, g(b))) = 2
  \]

- The function $diff_m(t_1, t_2)$ gives the size difference between two terms $t_1$ and $t_2$ under the metric $m$:
  \[
  diff_{length}([2, 3|L], [4|L]) = 1 \\
  diff_{length}(L, [H|L]) = -1 \\
  diff_{term\_depth}(f(a, g(X)), X) = 2
  \]
Size Analysis (size relations): Example

```prolog
:- entry nrev/2 : list(num) * var.

nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).

app([],L,L).
app([H|L],L1,[H|R]) :- app(L,L1,R).
```

- The automatically inferred size metric is \textit{length} (list length) for all arguments.
- All arguments inferred to be input except last ones (output). No aliasing.
- Let \langle b.i \rangle denote (a bound on) the size of the term(s) appearing in the \textit{i}^{th} argument position in the head of a block \textit{b}.
- Let \langle p.j.i \rangle denote (a bound on) the size of the term(s) appearing in the \textit{j}^{th} argument position in the \textit{i}^{th} body call of a block \textit{b}.
  \[\rightarrow p\] denotes the procedure called (for readability).
- Example: \langle \text{app.1}.1 \rangle = \text{length}(L) \text{ and } \langle \text{app.1} \rangle = \text{length}(\text{[H|L]})..
- First, we consider procedure \texttt{app} \texttt{(A,B,C)} (third arg is output).
- We want to obtain \textbf{intra-procedure} argument size relations:
  \(Sz_3^{app}(x,y)\) represents the size of the third argument of \texttt{app} as a function of its input data sizes \((x = \text{length}(A) \text{ and } y = \text{length}(B))\).
The automatically inferred size metric is \textit{length} (list length) for all arguments.

All arguments inferred to be input except last ones (output). No aliasing.

Let \(\langle b.i \rangle\) denote (a bound on) the size of the term(s) appearing in the \(i^{th}\) argument position in the head of a block \(b\).

Let \(\langle p.j.i \rangle\) denote (a bound on) the size of the term(s) appearing in the \(i^{th}\) argument position in the \(j^{th}\) body call of a block \(b\).

\(\rightarrow p\) denotes the procedure called (for readability).

Example: \(\langle \text{app.1.1} \rangle = \text{length}(\textbf{L})\) and \(\langle \text{app.1} \rangle = \text{length}([H|L])\).

First, we consider procedure \texttt{app}(A, B, C) (third arg is output).

We want to obtain intra-procedure argument size relations:

\(Sz_3^{\text{app}}(x, y)\) represents the size of the third argument of \texttt{app} as a function of its input data sizes \((x = \text{length}(A)\) and \(y = \text{length}(B))\).
Size Analysis (size relations): Example

```prolog
:- entry nrev/2 : list(num) * var.

nrev([],[]).

nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).

app([],L,L).

app([H|L],L1,[H|R]) :- app(L,L1,R).
```

- The automatically inferred size metric is \textit{length} (list length) for all arguments.
- All arguments inferred to be input except last ones (output). No aliasing.
- Let $\langle b.i \rangle$ denote (a bound on) the size of the term(s) appearing in the $i^{th}$ argument position in the head of a block $b$.
- Let $\langle p.j.i \rangle$ denote (a bound on) the size of the term(s) appearing in the $i^{th}$ argument position in the $j^{th}$ body call of a block $b$.
  - $\rightarrow p$ denotes the procedure called (for readability).
- Example: $\langle \textit{app.1.1} \rangle = \text{length}(L)$ and $\langle \textit{app.1} \rangle = \text{length}([H|L])$.
- First, we consider procedure \textit{app}(A, B, C) (third arg is output).
- We want to obtain \textit{intra-procedure} argument size relations:
  $Sz_3^{\text{app}}(x, y)$ represents the size of the third argument of \textit{app} as a function of its input data sizes ($x = \text{length}(A)$ and $y = \text{length}(B)$).
Size Analysis (size relations): Example

```
app([],L,L).
app([H|L],L1,[H|R]) :- app(L,L1,R).
```

- Argument size relations for the recursive clause:
  
  \[ \langle app.1.1 \rangle = \langle app.1 \rangle + \text{diff}(L, [H|L]) \quad (\text{inter-procedure}) \]
Size Analysis (size relations): Example

\begin{verbatim}
app([], L, L).
app([H|L], L1, [H|R]) :- app(L, L1, R).
\end{verbatim}

- Argument size relations for the recursive clause:

\[
\text{length}(L) = \text{length}([H|L]) + \text{diff}(L, [H|L])
\]
Size Analysis (size relations): Example

```prolog
app([], L, L).
app([H|L], L1, [H|R]) :- app(L, L1, R).
```

- Argument size relations for the recursive clause:
  \[ \text{length}(L) = \text{length}([H|L]) - 1 \]
Arguments size relations for the recursive clause:

\[ \langle app.1.1 \rangle = \langle app.1 \rangle - 1 \equiv length(L) = length([H|L]) - 1 \]
Size Analysis (size relations): Example

\[\text{app([]}, L, L)\].
\[\text{app([H|L]}, L1, [H|R]) :- \text{app}(L, L1, R).\]

- Argument size relations for the recursive clause:
  \[\langle app.1.1 \rangle = \langle app.1 \rangle - 1\]
Size Analysis (size relations): Example

app([],L,L).
app([H|L],L1,[H|R]) :- app(L,L1,R).

- Argument size relations for the recursive clause:
  \[ \langle app.1.1 \rangle = \langle app.1 \rangle - 1 \]
  \[ \langle app.1.2 \rangle = \langle app.2 \rangle + \text{diff}(L1,L1) \quad (\text{inter-procedure}) \]
Size Analysis (size relations): Example

\[ \text{app}([], L, L). \]
\[ \text{app}([H|L], L1, [H|R]) :- \text{app}(L, L1, R). \]

- Argument size relations for the recursive clause:
  \[ \langle \text{app.1.1} \rangle = \langle \text{app.1} \rangle - 1 \]
  \[ \langle \text{app.1.2} \rangle = \langle \text{app.2} \rangle \equiv \text{length}(L1) = \text{length}(L1) + 0 \]
Size Analysis (size relations): Example

\begin{verbatim}
app([],L,L).
app([H|L],L1,[H|R]) :- app(L,L1,R).
\end{verbatim}

- Argument size relations for the recursive clause:
  \[ \langle app.1.1 \rangle = \langle app.1 \rangle - 1 \]
  \[ \langle app.1.2 \rangle = \langle app.2 \rangle \equiv length(L1) = length(L1) \]
Size Analysis (size relations): Example

\[
\begin{align*}
\text{app}([], L, L). \\
\text{app}([H|L], L1, [H|R]) & :- \text{app}(L, L1, R).
\end{align*}
\]

- Argument size relations for the recursive clause:
  \[
  \begin{align*}
  \langle \text{app.1.1} \rangle &= \langle \text{app.1} \rangle - 1 \\
  \langle \text{app.1.2} \rangle &= \langle \text{app.2} \rangle \equiv \text{length}(L1) = \text{length}(L1) \\
  \langle \text{app.1.3} \rangle &= Sz^\text{app}_3(\langle \text{app.1.1} \rangle, \langle \text{app.1.2} \rangle) \quad (\text{intra-procedure})
  \end{align*}
  \]
**Size Analysis (size relations): Example**

```
app([], L, L).
app([H|L], L1, [H|R]) :- app(L, L1, R).
```

- **Argument size relations for the recursive clause:**
  
  \[
  \langle app.1.1 \rangle = \langle app.1 \rangle - 1 \\
  \langle app.1.2 \rangle = \langle app.2 \rangle \equiv \text{length}(L1) = \text{length}(L1) \\
  \langle app.1.3 \rangle = Sz^app_3(\langle app.1 \rangle - 1, \langle app.1.2 \rangle) \text{ (normalizing)}
  \]
Size Analysis (size relations): Example

\begin{verbatim}
app([], L, L).
app([H|L], L1, [H|R]) :- app(L, L1, R).
\end{verbatim}

- Argument size relations for the recursive clause:
  \begin{align*}
  \langle app.1.1 \rangle &= \langle app.1 \rangle - 1 \\
  \langle app.1.2 \rangle &= \langle app.2 \rangle \equiv length(L1) = length(L1) \\
  \langle app.1.3 \rangle &= Sz^{app}_3 (\langle app.1 \rangle - 1, \langle app.2 \rangle) \quad (normalizing)
  \end{align*}
Size Analysis (size relations): Example

\begin{align*}
\text{app}([], L, L).
\text{app}([H | L], L_1, [H | R]) & : \text{app}(L, L_1, R).
\end{align*}

- Argument size relations for the recursive clause:
  \begin{align*}
  \langle app.1.1 \rangle &= \langle app.1 \rangle - 1 \\
  \langle app.1.2 \rangle &= \langle app.2 \rangle \equiv length(L_1) = length(L_1) \\
  \langle app.1.3 \rangle &= Sz^\text{app}_3(\langle app.1 \rangle - 1, \langle app.2 \rangle) \\
  \langle app.3 \rangle &= \langle app.1.3 \rangle + \text{diff}([H \mid R], R) \quad \text{(inter-procedure)}
  \end{align*}
Size Analysis (size relations): Example

\[ \text{app}([], L, L). \]
\[ \text{app}([\text{H}|L], L1, [\text{H}|R]) :- \text{app}(L, L1, R). \]

- **Argument size relations for the recursive clause:**
  \[ \langle \text{app.1}.1 \rangle = \langle \text{app.1} \rangle - 1 \]
  \[ \langle \text{app.1}.2 \rangle = \langle \text{app.2} \rangle \equiv \text{length}(L1) = \text{length}(L1) \]
  \[ \langle \text{app.1}.3 \rangle = Sz_{3}^{\text{app}}(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) \]
  \[ \langle \text{app.3} \rangle = \langle \text{app.1.3} \rangle + 1 \]
Size Analysis (size relations): Example

app([],L,L).
app([H|L],L1,[H|R]) :- app(L,L1,R).

- Argument size relations for the recursive clause:

\[
\langle app.1.1 \rangle = \langle app.1 \rangle - 1
\]
\[
\langle app.1.2 \rangle = \langle app.2 \rangle \equiv length(L1) = length(L1)
\]
\[
\langle app.1.3 \rangle = Sz^app_3(\langle app.1 \rangle - 1, \langle app.2 \rangle)
\]
\[
\langle app.3 \rangle = Sz^app_3(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1 \quad (normalizing)
\]
Size Analysis (size relations): Example

\begin{verbatim}
app([],L,L).
app([H|L],L1,[H|R]) :- app(L,L1,R).
\end{verbatim}

- Argument size relations for the recursive clause:
  \[ \langle app.1.1 \rangle = \langle app.1 \rangle - 1 \]
  \[ \langle app.1.2 \rangle = \langle app.2 \rangle \equiv length(L1) = length(L1) \]
  \[ \langle app.1.3 \rangle = Sz_{app}^3(\langle app.1 \rangle - 1, \langle app.2 \rangle) \]
  \[ \langle app.3 \rangle = Sz_{app}^3(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1 \]
  \[ Sz_{app}^3(\langle app.1 \rangle, \langle app.2 \rangle) = \langle app.3 \rangle \quad (intra-procedure) \]
Size Analysis (size relations): Example

\texttt{app([],L,L).} \\
\texttt{app([H|L],L1,[H|R]) :- app(L,L1,R).}

- Argument size relations for the recursive clause:
  \begin{align*}
  \langle \text{app.1.1} \rangle &= \langle \text{app.1} \rangle - 1 \\
  \langle \text{app.1.2} \rangle &= \langle \text{app.2} \rangle \equiv \text{length}(L1) = \text{length}(L1) \\
  \langle \text{app.1.3} \rangle &= Sz_{\text{app}}^3(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) \\
  \langle \text{app.3} \rangle &= Sz_{\text{app}}^3(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) + 1 \\
  Sz_{\text{app}}^3(\langle \text{app.1} \rangle, \langle \text{app.2} \rangle) &= Sz_{\text{app}}^3(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) + 1 \quad \text{(normalizing)}
  \end{align*}
Size Analysis (size relations): Example

\[
\text{app}([], L, L).
\]
\[
\text{app}([H|L], L_1, [H|R]) :- \text{app}(L, L_1, R).
\]

- Argument size relations for the recursive clause:
  \[
  \langle \text{app.1.1} \rangle = \langle \text{app.1} \rangle - 1
  \]
  \[
  \langle \text{app.1.2} \rangle = \langle \text{app.2} \rangle \iff \text{length}(L_1) = \text{length}(L_1)
  \]
  \[
  \langle \text{app.1.3} \rangle = Sz^app_3(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle)
  \]
  \[
  \langle \text{app.3} \rangle = Sz^app_3(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) + 1
  \]
  \[
  Sz^app_3(\langle \text{app.1} \rangle, \langle \text{app.2} \rangle) = Sz^app_3(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) + 1
  \]
Size Analysis (size relations): Example

app([], L, L).
app([H|L], L1, [H|R]) :- app(L, L1, R).

- Argument size relations for the recursive clause:
  \[
  \langle app.1.1 \rangle = \langle app.1 \rangle - 1
  \]
  \[
  \langle app.1.2 \rangle = \langle app.2 \rangle \equiv length(L1) = length(L1)
  \]
  \[
  \langle app.1.3 \rangle = Sz_{app}^3(\langle app.1 \rangle - 1, \langle app.2 \rangle)
  \]
  \[
  \langle app.3 \rangle = Sz_{app}^3(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1
  \]
  \[
  Sz_{app}^3(\langle app.1 \rangle, \langle app.2 \rangle) = Sz_{app}^3(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1
  \]

- From the first block of \texttt{app}, we obtain the equation:
  \[
  Sz_{app}^3(0, \langle app.2 \rangle) = \langle app.2 \rangle
  \]
Size Analysis (size relations): Example

\[
\text{app}([], L, L).
\]
\[
\text{app}([H|L], L1, [H|R]) :- \text{app}(L, L1, R).
\]

- Argument size relations for the recursive clause:
  \[
  \langle \text{app.1.1} \rangle = \langle \text{app.1} \rangle - 1
  \]
  \[
  \langle \text{app.1.2} \rangle = \langle \text{app.2} \rangle \equiv \text{length}(L1) = \text{length}(L1)
  \]
  \[
  \langle \text{app.1.3} \rangle = Sz_3^{\text{app}}(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle)
  \]
  \[
  \langle \text{app.3} \rangle = Sz_3^{\text{app}}(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) + 1
  \]
  \[
  Sz_3^{\text{app}}(\langle \text{app.1} \rangle, \langle \text{app.2} \rangle) = Sz_3^{\text{app}}(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) + 1
  \]

- From the first block of \text{app}, we obtain the equation:
  \[
  Sz_3^{\text{app}}(0, \langle \text{app.2} \rangle) = \langle \text{app.2} \rangle
  \]

- The equations:
  \[
  Sz_3^{\text{app}}(0, \langle \text{app.2} \rangle) = \langle \text{app.2} \rangle
  \]
  \[
  Sz_3^{\text{app}}(\langle \text{app.1} \rangle, \langle \text{app.2} \rangle) = Sz_3^{\text{app}}(\langle \text{app.1} \rangle - 1, \langle \text{app.2} \rangle) + 1
  \]
Size Analysis (size relations): Example

\[
\text{app}([], L, L). \\
\text{app}([H|L], L1, [H|R]) :- \text{app}(L, L1, R).
\]

- Argument size relations for the recursive clause:
  \[
  \langle \text{app}.1.1 \rangle = \langle \text{app}.1 \rangle - 1 \\
  \langle \text{app}.1.2 \rangle = \langle \text{app}.2 \rangle \equiv \text{length}(L1) = \text{length}(L1) \\
  \langle \text{app}.1.3 \rangle = \text{Sz}_{app}^3(\langle \text{app}.1 \rangle - 1, \langle \text{app}.2 \rangle) \\
  \langle \text{app}.3 \rangle = \text{Sz}_{app}^3(\langle \text{app}.1 \rangle - 1, \langle \text{app}.2 \rangle) + 1 \\
  \text{Sz}_{app}^3(\langle \text{app}.1 \rangle, \langle \text{app}.2 \rangle) = \text{Sz}_{app}^3(\langle \text{app}.1 \rangle - 1, \langle \text{app}.2 \rangle) + 1
\]

- From the first block of \text{app}, we obtain the equation:
  \[\text{Sz}_{app}^3(0, \langle \text{app}.2 \rangle) = \langle \text{app}.2 \rangle\]

- The equations \((n = \langle \text{app}.1 \rangle, m = \langle \text{app}.2 \rangle)\):
  \[\text{Sz}_{app}^3(n, m) = m \quad \text{if } n = 0\]
  \[\text{Sz}_{app}^3(n, m) = \text{Sz}_{app}^3(n - 1, m) + 1 \quad \text{if } n > 0\]
Argument size relations for the recursive clause:
\[
\langle app.1.1 \rangle = \langle app.1 \rangle - 1
\]
\[
\langle app.1.2 \rangle = \langle app.2 \rangle \equiv \text{length}(L1) = \text{length}(L1)
\]
\[
\langle app.1.3 \rangle = \text{Sz}_{3}^{app}(\langle app.1 \rangle - 1, \langle app.2 \rangle)
\]
\[
\langle app.3 \rangle = \text{Sz}_{3}^{app}(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1
\]
\[
\text{Sz}_{3}^{app}(\langle app.1 \rangle, \langle app.2 \rangle) = \text{Sz}_{3}^{app}(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1
\]

From the first block of \textit{app}, we obtain the equation:
\[
\text{Sz}_{3}^{app}(0, \langle app.2 \rangle) = \langle app.2 \rangle
\]

The equations \((n = \langle app.1 \rangle, m = \langle app.2 \rangle)\):
\[
\text{Sz}_{3}^{app}(n, m) = m \quad \text{if } n = 0
\]
\[
\text{Sz}_{3}^{app}(n, m) = \text{Sz}_{3}^{app}(n - 1, m) + 1 \quad \text{if } n > 0
\]
are solved, obtaining the closed-form function:
\[
\text{Sz}_{3}^{app}(n, m) = n + m \quad \text{if } n \leq 0
\]
Argument size relations for the recursive clause:
\[
\langle app.1.1 \rangle = \langle app.1 \rangle - 1 \\
\langle app.1.2 \rangle = \langle app.2 \rangle \equiv \text{length}(L_1) = \text{length}(L_1) \\
\langle app.1.3 \rangle = Sz_{3}^{app}(\langle app.1 \rangle - 1, \langle app.2 \rangle) \\
\langle app.3 \rangle = Sz_{3}^{app}(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1 \\
Sz_{3}^{app}(\langle app.1 \rangle, \langle app.2 \rangle) = Sz_{3}^{app}(\langle app.1 \rangle - 1, \langle app.2 \rangle) + 1
\]

From the first block of \texttt{app}, we obtain the equation:
\[
Sz_{3}^{app}(0, \langle app.2 \rangle) = \langle app.2 \rangle
\]

The equations \((n = \langle app.1 \rangle, m = \langle app.2 \rangle)\):
\[
Sz_{3}^{app}(n, m) = m \quad \text{if } n = 0 \\
Sz_{3}^{app}(n, m) = Sz_{3}^{app}(n - 1, m) + 1 \quad \text{if } n > 0
\]
are solved, obtaining the closed-form function:
\[
Sz_{3}^{app}(n, m) = n + m \quad \text{if } n \leq 0
\]
which is used for the analysis of procedure \texttt{nrev}.
Size Analysis (size relations): Example

\[
\text{nrev([],[]).} \\
\text{nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).}
\]

- We now switch to procedure \text{nrev(A, B)}, where \(A\) and \(B\) are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle + \text{diff}(L, [H|L]) \quad \text{(inter-procedure)}
  \]
We now switch to procedure \( \text{nrev}(A, B) \), where \( A \) and \( B \) are input and output arguments respectively.

**Argument size relations for the recursive clause:**

\[
\text{length}(L) = \text{length}([H|L]) + \text{diff}(L, [H|L])
\]
We now switch to procedure \texttt{nrev(A, B)}, where \( A \) and \( B \) are input and output arguments respectively.

Argument size relations for the recursive clause:
\[
\text{length}(L) = \text{length}([H|L]) - 1
\]
Size Analysis (size relations): Example

\[
\begin{align*}
n\text{rev}([],[]). \\
n\text{rev}([H|L],R) & :\text{~} n\text{rev}(L,R_1), \text{app}(R_1,[H],R).
\end{align*}
\]

- We now switch to procedure \( \text{nrev}(A, B) \), where \( A \) and \( B \) are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle n\text{rev.1.1} \rangle = \langle n\text{rev.1} \rangle - 1 \equiv length(L) = length([H|L]) - 1
  \]
Size Analysis (size relations): Example

nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).

- We now switch to procedure \texttt{nrev(A, B)}, where \texttt{A} and \texttt{B} are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[ \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \]
We now switch to procedure \texttt{nrev}(A, B), where A and B are input and output arguments respectively.

Argument size relations for the recursive clause:

\begin{align*}
\langle \texttt{nrev.1.1} \rangle &= \langle \texttt{nrev.1} \rangle - 1 \\
\langle \texttt{nrev.1.2} \rangle &= Sz_2^{\texttt{nrev}}(\langle \texttt{nrev.1.1} \rangle) \quad \text{(intra-procedure)}
\end{align*}
Size Analysis (size relations): Example

\[
\text{nrev}([],[]).
\]
\[
\text{nrev}([\text{H}|\text{L}], R) :- \text{nrev}(\text{L}, \text{R}1), \text{app}(\text{R}1, [\text{H}], R).
\]

- We now switch to procedure \text{nrev}(A, B), where \(A\) and \(B\) are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle \text{nrev.1.1} \rangle = \langle \text{nrev.1} \rangle - 1
  \]
  \[
  \langle \text{nrev.1.2} \rangle = Sz_{2}^{\text{nrev}}(\langle \text{nrev.1.1} \rangle) \equiv length(\text{R}1) = Sz_{2}^{\text{nrev}}(length(\text{L}))
  \]
Size Analysis (size relations): Example

```
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).
```

- We now switch to procedure `nrev(A, B)`, where `A` and `B` are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[ \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \]
  \[ \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \quad (normalizing) \]
We now switch to procedure `nrev(A, B)`, where `A` and `B` are input and output arguments respectively.

Argument size relations for the recursive clause:

- \[ \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \]
- \[ \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \]
- \[ \langle app.2.1 \rangle = \langle nrev.1.2 \rangle \quad (inter-procedure) \]
Size Analysis (size relations): Example

\[
\text{nrev}([], []).
\text{nrev}([H|L], R) :- \text{nrev}(L, R1), \text{app}(R1, [H], R).
\]

- We now switch to procedure \text{nrev}(A, B), where A and B are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1
  \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1)
  \langle app.2.1 \rangle = \langle nrev.1.2 \rangle \equiv \text{length}(R1) = \text{length}(R1)
  \]
We now switch to procedure \texttt{nrev(A, B)}, where A and B are input and output arguments respectively.

Argument size relations for the recursive clause:
\[
\langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \\
\langle nrev.1.2 \rangle = Sz_{nrev}^2(\langle nrev.1 \rangle - 1) \\
\langle app.2.1 \rangle = Sz_{nrev}^2(\langle nrev.1 \rangle - 1) \quad (normalizing)
\]
Size Analysis (size relations): Example

\[
\text{nrev([],[]).} \\
\text{nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).}
\]

- We now switch to procedure \text{nrev(A, B)}, where \text{A} and \text{B} are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \\
  \langle nrev.1.2 \rangle = Sz^{nrev}_2 (\langle nrev.1 \rangle - 1) \\
  \langle app.2.1 \rangle = Sz^{nrev}_2 (\langle nrev.1 \rangle - 1) \\
  \langle app.2.2 \rangle = length([H]) \quad \text{(explicit size)}
  \]
Size Analysis (size relations): Example

\texttt{nrev([],[]).}
\texttt{nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).}

- We now switch to procedure \texttt{nrev(A, B)}, where \texttt{A} and \texttt{B} are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \\
  \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \\
  \langle app.2.1 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \\
  \langle app.2.2 \rangle = 1
  \]
Size Analysis (size relations): Example

\[ \text{nrev([],[]).} \]
\[ \text{nrev([H|L], R) :- nrev(L, R1), app(R1, [H], R).} \]

- We now switch to procedure \text{nrev(A, B)}, where \text{A} and \text{B} are input and output arguments respectively.

- Argument size relations for the recursive clause:
  \[ \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \]
  \[ \langle nrev.1.2 \rangle = Sz_2^{nrev} (\langle nrev.1 \rangle - 1) \]
  \[ \langle app.2.1 \rangle = Sz_2^{nrev} (\langle nrev.1 \rangle - 1) \]
  \[ \langle app.2.2 \rangle = 1 \]
  \[ \langle app.2.3 \rangle = Sz_3^{app} (\langle app.2.1 \rangle, \langle app.2.2 \rangle) \quad \text{(intra-procedure)} \]
**Size Analysis (size relations): Example**

\[
\text{nrev}([],[]).
\text{nrev}([H|L], R) :- \text{nrev}(L, R1), \text{app}(R1, [H], R).
\]

- We now switch to procedure \text{nrev}(A, \ B), where \ A \ and \ B \ are \ input \ and \ output \ arguments \ respectively.

- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \\
  \langle nrev.1.2 \rangle = Sz^\text{nrev}_2 (\langle nrev.1 \rangle - 1) \\
  \langle app.2.1 \rangle = Sz^\text{nrev}_2 (\langle nrev.1 \rangle - 1) \\
  \langle app.2.2 \rangle = 1 \\
  \langle app.2.3 \rangle = \langle app.2.1 \rangle + \langle app.2.2 \rangle \quad \text{using} \quad Sz^\text{app}_3 (x, y) = x + y
  \]

\]
We now switch to procedure \texttt{nrev(A, B)}, where \texttt{A} and \texttt{B} are input and output arguments respectively.

Argument size relations for the recursive clause:
\[
\langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1
\]
\[
\langle nrev.1.2 \rangle = Sz_{nrev}^2 (\langle nrev.1 \rangle - 1)
\]
\[
\langle app.2.1 \rangle = Sz_{nrev}^2 (\langle nrev.1 \rangle - 1)
\]
\[
\langle app.2.2 \rangle = 1
\]
\[
\langle app.2.3 \rangle = Sz_{nrev}^2 (\langle nrev.1 \rangle - 1) + 1 \quad (\text{normalizing})
\]
Size Analysis (size relations): Example

\texttt{nrev([],[]).}
\texttt{nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).}

- We now switch to procedure \texttt{nrev(A, B)}, where \texttt{A} and \texttt{B} are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[ \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \]
  \[ \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \]
  \[ \langle app.2.1 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \]
  \[ \langle app.2.2 \rangle = 1 \]
  \[ \langle app.2.3 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1 \]
  \[ \langle nrev.2 \rangle = \langle app.2.3 \rangle + \text{diff}(R, R) \quad \text{(inter-procedure)} \]
Size Analysis (size relations): Example

\[ nrev([], []). \]
\[ nrev([H|L], R) :- nrev(L, R1), app(R1, [H], R). \]

- We now switch to procedure \texttt{nrev(A, B)}, where \texttt{A} and \texttt{B} are input and output arguments respectively.

- Argument size relations for the recursive clause:
  \[ \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \]
  \[ \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \]
  \[ \langle app.2.1 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \]
  \[ \langle app.2.2 \rangle = 1 \]
  \[ \langle app.2.3 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1 \]
  \[ \langle nrev.2 \rangle = \langle app.2.3 \rangle + 0 \]
We now switch to procedure \texttt{nrev}(A, B), where \texttt{A} and \texttt{B} are input and output arguments respectively.

Argument size relations for the recursive clause:
\begin{align*}
\langle nrev.1.1 \rangle &= \langle nrev.1 \rangle - 1 \\
\langle nrev.1.2 \rangle &= Sz_{nrev}^2(\langle nrev.1 \rangle - 1) \\
\langle app.2.1 \rangle &= Sz_{nrev}^2(\langle nrev.1 \rangle - 1) \\
\langle app.2.2 \rangle &= 1 \\
\langle app.2.3 \rangle &= Sz_{nrev}^2(\langle nrev.1 \rangle - 1) + 1 \\
\langle nrev.2 \rangle &= Sz_{nrev}^2(\langle nrev.1 \rangle - 1) + 1 \quad \text{(normalizing)}
\end{align*}
Size Analysis (size relations): Example

\[
\text{nrev}([],[]).
\]
\[
\text{nrev}([\text{H}|\text{L}], \text{R}) :- \text{nrev}(\text{L}, \text{R}1), \text{app}(\text{R}1, [\text{H}], \text{R}).
\]

- We now switch to procedure \text{nrev}(A, B), where A and B are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1
  \]
  \[
  \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1)
  \]
  \[
  \langle app.2.1 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1)
  \]
  \[
  \langle app.2.2 \rangle = 1
  \]
  \[
  \langle app.2.3 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1
  \]
  \[
  \langle nrev.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1
  \]
  \[
  Sz_2^{nrev}(\langle nrev.1 \rangle) = \langle nrev.2 \rangle \quad (intra-procedure)
  \]
**Size Analysis (size relations): Example**

```
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).
```

- We now switch to procedure `nrev(A, B)`, where `A` and `B` are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle \text{nrev.1.1} \rangle = \langle \text{nrev.1} \rangle - 1 \\
  \langle \text{nrev.1.2} \rangle = Sz_{\text{nrev}}^2 (\langle \text{nrev.1} \rangle - 1) \\
  \langle \text{app.2.1} \rangle = Sz_{\text{nrev}}^2 (\langle \text{nrev.1} \rangle - 1) \\
  \langle \text{app.2.2} \rangle = 1 \\
  \langle \text{app.2.3} \rangle = Sz_{\text{nrev}}^2 (\langle \text{nrev.1} \rangle - 1) + 1 \\
  \langle \text{nrev.2} \rangle = Sz_{\text{nrev}}^2 (\langle \text{nrev.1} \rangle - 1) + 1 \\
  Sz_{\text{nrev}}^2 (\langle \text{nrev.1} \rangle) = Sz_{\text{nrev}}^2 (\langle \text{nrev.1} \rangle - 1) + 1 \quad (normalizing)
  \]
We now switch to procedure \texttt{nrev(A, B)}, where \texttt{A} and \texttt{B} are input and output arguments respectively.

Argument size relations for the recursive clause:
\[
\langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \\
\langle nrev.1.2 \rangle = Sz_2^{\text{nrev}}(\langle nrev.1 \rangle - 1) \\
\langle app.2.1 \rangle = Sz_2^{\text{nrev}}(\langle nrev.1 \rangle - 1) \\
\langle app.2.2 \rangle = 1 \\
\langle app.2.3 \rangle = Sz_2^{\text{nrev}}(\langle nrev.1 \rangle - 1) + 1 \\
\langle nrev.2 \rangle = Sz_2^{\text{nrev}}(\langle nrev.1 \rangle - 1) + 1 \\
Sz_2^{\text{nrev}}(\langle nrev.1 \rangle) = Sz_2^{\text{nrev}}(\langle nrev.1 \rangle - 1) + 1
\]
Size Analysis (size relations): Example

- \texttt{nrev([],[])}.
- \texttt{nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R)}.

- We now switch to procedure \texttt{nrev(A, B)}, where \(A\) and \(B\) are input and output arguments respectively.

- Argument size relations for the recursive clause:
  \[
  \langle \text{nrev.1.1} \rangle = \langle \text{nrev.1} \rangle - 1 \\
  \langle \text{nrev.1.2} \rangle = Sz_2^{\text{nrev}}(\langle \text{nrev.1} \rangle - 1) \\
  \langle \text{app.2.1} \rangle = Sz_2^{\text{nrev}}(\langle \text{nrev.1} \rangle - 1) \\
  \langle \text{app.2.2} \rangle = 1 \\
  \langle \text{app.2.3} \rangle = Sz_2^{\text{nrev}}(\langle \text{nrev.1} \rangle - 1) + 1 \\
  \langle \text{nrev.2} \rangle = Sz_2^{\text{nrev}}(\langle \text{nrev.1} \rangle - 1) + 1 \\
  Sz_2^{\text{nrev}}(\langle \text{nrev.1} \rangle) = Sz_2^{\text{nrev}}(\langle \text{nrev.1} \rangle - 1) + 1 \\
  
  - From the first block of \texttt{nrev}, we obtain the equation: \(Sz_2^{\text{nrev}}(0) = 0\)
Size Analysis (size relations): Example

\begin{verbatim}
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).
\end{verbatim}

- We now switch to procedure \texttt{nrev(A, B)}, where \(A\) and \(B\) are input and output arguments respectively.
- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \\
  \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \\
  \langle app.2.1 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \\
  \langle app.2.2 \rangle = 1 \\
  \langle app.2.3 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1 \\
  \langle nrev.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1 \\
  Sz_2^{nrev}(\langle nrev.1 \rangle) = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1
  \]
- From the first block of \texttt{nrev}, we obtain the equation:
  \(Sz_2^{nrev}(0) = 0\)
- The equations:
  \(Sz_2^{nrev}(0) = 0\)
  \(Sz_2^{nrev}(\langle nrev.1 \rangle) = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1\)
Size Analysis (size relations): Example

\begin{verbatim}
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).
\end{verbatim}

- We now switch to procedure \texttt{nrev(A, B)}, where \texttt{A} and \texttt{B} are input and output arguments respectively.

- Argument size relations for the recursive clause:
  \[
  \langle nrev.1.1 \rangle = \langle nrev.1 \rangle - 1 \\
  \langle nrev.1.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \\
  \langle app.2.1 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) \\
  \langle app.2.2 \rangle = 1 \\
  \langle app.2.3 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1 \\
  \langle nrev.2 \rangle = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1 \\
  Sz_2^{nrev}(\langle nrev.1 \rangle) = Sz_2^{nrev}(\langle nrev.1 \rangle - 1) + 1
  \]

- From the first block of \texttt{nrev}, we obtain the equation:
  \[
  Sz_2^{nrev}(0) = 0
  \]

- The equations (\( n = \langle nrev.1 \rangle \)):
  \[
  Sz_2^{nrev}(0) = 0 \\
  Sz_2^{nrev}(n) = Sz_2^{nrev}(n - 1) + 1
  \]
### Size Analysis (size relations): Example

```
nrev([],[]).
nrev([H|L], R) :- nrev(L, R1), app(R1, [H], R).
```

- We now switch to procedure `nrev(A, B)`, where `A` and `B` are input and output arguments respectively.
- Argument size relations for the recursive clause:
  - $\langle \text{nrev.1.1} \rangle = \langle \text{nrev.1} \rangle - 1$
  - $\langle \text{nrev.1.2} \rangle = Sz_2^{nrev}(\langle \text{nrev.1} \rangle - 1)$
  - $\langle \text{app.2.1} \rangle = Sz_2^{nrev}(\langle \text{nrev.1} \rangle - 1)$
  - $\langle \text{app.2.2} \rangle = 1$
  - $\langle \text{app.2.3} \rangle = Sz_2^{nrev}(\langle \text{nrev.1} \rangle - 1) + 1$
  - $\langle \text{nrev.2} \rangle = Sz_2^{nrev}(\langle \text{nrev.1} \rangle - 1) + 1$
  - $Sz_2^{nrev}(\langle \text{nrev.1} \rangle) = Sz_2^{nrev}(\langle \text{nrev.1} \rangle - 1) + 1$
- From the first block of `nrev`, we obtain the equation:
  - $Sz_2^{nrev}(0) = 0$
- The equations ($n = \langle \text{nrev.1} \rangle$):
  - $Sz_2^{nrev}(0) = 0$
  - $Sz_2^{nrev}(n) = Sz_2^{nrev}(n - 1) + 1$
are solved, obtaining the closed-form function:
  - $Sz_2^{nrev}(n) = n$
Size Analysis (size relations): Example

- The size of the output argument of \texttt{nrev(A, B)} is given by the following equations (where \( n = \text{length}(A) \)):
  \[
  Sz_{nrev}^2(0) = 0 \\
  Sz_{nrev}^2(n) = Sz_{nrev}^2(n - 1) + 1
  \]

- Solution: \( Sz_{nrev}^2(n) = n \).
  The length (size) of the output argument of \texttt{nrev} is equal to the length of its input.
Cost Analysis (cost relations): Example

\[
\text{nrev}([],[]).
\text{nrev}([H|L], R) :- \text{nrev}(L, R_1), \text{app}(R_1, [H], R).
\]

\[
\text{app}([], L, L).
\text{app}([H|L], L_1, [H|R]) :- \text{app}(L, L_1, R).
\]

- **Cost relations**
  \[ n = \text{length}(X) \] (length of list \(X\))

- **Cost of \text{nrev}:**
  \[
  C_{\text{nrev}}(0) = 1 \\
  C_{\text{nrev}}(n) = 1 + C_{\text{nrev}}(n-1) + C_{\text{app}}(n-1, 1) \quad \text{if} \ n > 0
  \]

- **Cost of \text{app}:**
  \[
  C_{\text{app}}(0, m) = 1 \\
  C_{\text{app}}(n, m) = 1 + C_{\text{app}}(n-1, m) \quad \text{if} \ n > 0
  \]

- **Approach described in** \([\text{PLDI}'90, \text{ILPS}'97]\) (for lower bounds, nondet relations, balanced costs), \([\text{ICLP}'07, \text{Bytecode}'09]\) (for user-defined resources) – see slides at end for algorithm with user-defined resources.
Cost Analysis (cost relations): Example

\[ \text{nrev}([],[]). \]
\[ \text{nrev}([H|L],R) :\text{-- nrev}(L,R1), \text{app}(R1,[H],R). \]

\[ \text{app}([],L,L). \]
\[ \text{app}([H|L],L1,[H|R]) :\text{-- app}(L,L1,R). \]

- **Cost relations**
  \[ n = \text{length}(X) \text{ (length of list } X) \]

- **Cost of \text{nrev}:**
  \[ C_{n\text{rev}}(0) = 1 \]
  \[ C_{n\text{rev}}(n) = 1 + C_{n\text{rev}}(n-1) + C_{\text{app}}(n-1,1) \text{ if } n > 0 \]

- **Cost of \text{app} → closed form:**
  \[ C_{\text{app}}(n,m) = n + 1 \text{ for } n \geq 0. \]
  \[ C_{\text{app}}(0,m) = 1 \]
  \[ C_{\text{app}}(n,m) = 1 + C_{\text{app}}(n-1,m) \text{ if } n > 0 \]

- Approach described in [PLDI’90], [ILPS’97] (for lower bounds, nondet relations, balanced costs), [ICLP’07, Bytecode’09] (for user-defined resources) –see slides at end for algorithm with user-defined resources.
Cost Analysis (cost relations): Example

\begin{verbatim}
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).

app([],L,L).
app([H|L],L1,[H|R]) :- app(L,L1,R).
\end{verbatim}

- **Cost relations**

\[
\text{n} = \text{length}(X) \text{ (length of list } X)\]

- **Cost of } nrev:}\
\[
C_{nrev}(0) = 1 \\
C_{nrev}(n) = 1 + C_{nrev}(n-1) + n \quad \text{if } n > 0
\]

- **Cost of } app \rightarrow \text{ closed form:} } C_{app}(n,m) = n + 1 \text{ for } n \geq 0.\]
\[
C_{app}(0,m) = 1 \\
C_{app}(n,m) = 1 + C_{app}(n-1,m) \quad \text{if } n > 0
\]

- **Approach described in [PLDI'90], [ILPS'97] (for lower bounds, nondet relations, balanced costs), [ICLP'07, Bytecode'09] (for user-defined resources) —see slides at end for algorithm with user-defined resources.
Cost Analysis (cost relations): Example

\[
\text{nrev([],[]).}
\]
\[
\text{nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).}
\]
\[
\text{app([],L,L).}
\]
\[
\text{app([H|L],L1,[H|R]) :- app(L,L1,R).}
\]

- **Cost relations**
  \[ n = \text{length}(X) \text{ (length of list } X) \]

- **Cost of \textit{nrev} \rightarrow closed form:**
  \[ C_{\text{nrev}}(n) = \frac{1}{2} n^2 + \frac{3}{2} n + 1, \text{ for } n \geq 0. \]
  \[ C_{\text{nrev}}(0) = 1 \]
  \[ C_{\text{nrev}}(n) = 1 + C_{\text{nrev}}(n - 1) + n \text{ if } n > 0 \]

- **Cost of \textit{app} \rightarrow closed form:**
  \[ C_{\text{app}}(n, m) = n + 1 \text{ for } n \geq 0. \]
  \[ C_{\text{app}}(0, m) = 1 \]
  \[ C_{\text{app}}(n, m) = 1 + C_{\text{app}}(n - 1, m) \text{ if } n > 0 \]

- **Approach described in** [PLDI’90], [ILPS’97] (for lower bounds, nondet relations, balanced costs), [ICLP’07, Bytecode’09] (for user-defined resources) – see slides at end for algorithm with user-defined resources.
Cost Analysis (cost relations): Example

\[
\text{nrev([],[]).}
\]
\[
\text{nrev([H|L],R) :- nrev(L,R1), app(R1,[H],R).}
\]
\[
\text{app([],L,L).}
\]
\[
\text{app([H|L],L1,[H|R]) :- app(L,L1,R).}
\]

- **Cost relations**
  \[ n = \text{length}(X) \] (length of list \( X \))

- **Cost of \text{nrev} \to\text{closed form}**: \[ C_{\text{nrev}}(n) = \frac{1}{2} n^2 + \frac{3}{2} n + 1, \text{ for } n \geq 0. \]
  \[ C_{\text{nrev}}(0) = 1 \]
  \[ C_{\text{nrev}}(n) = 1 + C_{\text{nrev}}(n - 1) + n \text{ if } n > 0 \]

- **Cost of \text{app} \to\text{closed form}**: \[ C_{\text{app}}(n,m) = n + 1 \text{ for } n \geq 0. \]
  \[ C_{\text{app}}(0,m) = 1 \]
  \[ C_{\text{app}}(n,m) = 1 + C_{\text{app}}(n - 1, m) \text{ if } n > 0 \]

- **Approach described in [PLDI'90], [ILPS'97]** (for lower bounds, nondet relations, balanced costs), [ICLP'07, Bytecode'09] (for user-defined resources) –see slides at end for algorithm with user-defined resources.
Cost analysis w/recurrences: many other interesting further developments –past and present!

- Improved/specific recurrence solvers.
- Balancing techniques for, e.g., divide and conquer (related to amortized).
- Accumulated cost / static profiling.
- Analyses for time, memory, energy...
  (→ see our talk on energy analysis tomorrow!).
- Relational cost.
- Cost analysis for concurrent programs.
- ...

We cover some in the following (see also the appendices).
A Challenge: Recurrence Relation Solving

- A key factor for the power/accuracy of this approach to resource analysis is *recurrence relation solving*.

- We perform extensive normalization of expressions to facilitate operations:
  - simplification, classification, comparison, ...

- Many solvers available (CiaoPP’s built-in solver, Mathematica, Maxima, PURRS, PUBS, etc.), however:
  - Each one can solve different classes of recurrences.
  - Some relevant recurrences cannot be solved by any of them.

- We need either **exact** solutions or **bounds**.

- Recent improvements/extensions within CiaoPP include the design and implementation of:
  - A modular framework for solving recurrence relations.
  - A specialized solver (combined with ranking function generation) for a common class of recurrence relations (with increasing argument sizes).

(see appendix for more details)
Architecture of the Modular Solver Framework

- **Solver_Strategies**: common interface to strategies.
- **Strat_i**: strategy to solve (or over-approximate) recurrences, using back-end solvers and the classifier.
- **Rec_Classifier**: associates a label to each input recurrence relation, to identify the class of recurrence.
- **Solver_Utils**: common interface to back-end solvers.
- **BS_i**: implements the interface defined by **Solver_Utils**, connecting back-end solvers (Mathematica, CiaoPP’s built-in, etc.)
- Also combination with ranking function generation (and partial evaluation).
Setting up and Solving Non-deterministic Recurrences
[ILPS'97]

```prolog
define(entry, qsort/2 : list(num) → var).

qsort([], []).
qsort([H|L], R) :- part(L, H, L1, L2), qsort(L1, R1), qsort(L2, R2), app(R1, [H|R2], R).

part([], C, [], []).
part([E|T], C, [E|L], R) :- E < C, part(T, C, L, R).
part([E|T], C, L, [E|R]) :- E ≥ C, part(T, C, L, R).

app([], L, L).
app([H|A], B, [H|C]) :- app(A, B, C).
```

- For (frequent) divide-and-conquer programs in which the sizes of the output arguments of the “divide” part are dependent.
- Classical approach: independent size analysis for output arguments
  → very conservative upper/lower bounds.
For (frequent) divide-and-conquer programs in which the sizes of the output arguments of the “divide” part are dependent.

Classical approach: independent size analysis for output arguments

→ upper bounds: $S_{3}^{\text{part}}(x) = x$ and $S_{4}^{\text{part}}(x) = x$. 
Setting up and Solving Non-deterministic Recurrences
[ILPS'97]

```
:- entry qsort/2 : list(num) * var.

qsort([],[]).
qsort([H|L],R) :- part(L,H,L1,L2), qsort(L1,R1), qsort(L2,R2), app(R1,[H|R2],R).

part([],C,[[],[]]).
part([E|T],C,[E|L],R) :- E < C, part(T,C,L,R).
part([E|T],C,L,[E|R]) :- E >= C, part(T,C,L,R).

app([],L,L).
app([H|A],B,[H|C]) :- app(A,B,C).
```

- For (frequent) divide-and-conquer programs in which the sizes of the output arguments of the “divide” part are dependent.
- Classical approach: independent size analysis for output arguments
  \[ Sz^3_{part}(x) = 0 \] and \[ Sz^4_{part}(x) = 0. \]
Setting up and Solving Non-deterministic Recurrences

[ILPS'97]

```prolog
:- entry qsort/2 : list(num) * var.

qsort([],[]).
qsort([H|L],R) :- part(L,H,L1,L2), qsort(L1,R1), qsort(L2,R2), app(R1,[H|R2],R).

part([],C,[]).  % Example: part([],E,L,L2).
part([E|T],C,L,R) :- E < C, part(T,C,L,R).
part([E|T],C,L,[E|R]) :- E >= C, part(T,C,L,R).

app([],L,L).
app([H|A],B,[H|C]) :- app(A,B,C).
```

We use a relational size analysis for output arguments.

E.g., for \( \text{part}(L,H,L1,L2) \rightarrow \text{length}(L) = \text{length}(L1) + \text{length}(L2) \)

- Let \( \text{length}(L) = n', \ n' \geq 0 \), and \( \text{length}(L1) = k, \ 0 \leq k \leq n' \).
- then \( \text{length}(L2) = n' - k \).
Setting up and Solving Non-deterministic Recurrences
[ILPS'97]

```
:- entry qsort/2 : list(num) * var.

qsort([],[]).
qsort([H|L],R) :- part(L,H,L1,L2), qsort(L1,R1), qsort(L2,R2), app(R1,[H|R2],R).

part([],C,[],[]).
part([E|T],C,[E|L],R) :- E < C, part(T,C,L,R).
part([E|T],C,L,[E|R]) :- E >= C, part(T,C,L,R).

app([],L,L).
app([H|A],B,[H|C]) :- app(A,B,C).
```

- Non-deterministic cost relations
  \( (n' = n - 1, \text{length}(L) = n', \text{length}(L1) = k): \)

  \[
  C_{\text{qsort}}(n) = 1 \text{ if } n = 0 \\
  C_{\text{qsort}}(n) = 1 + C_{\text{part}}(n - 1) + C_{\text{qsort}}(k) + C_{\text{qsort}}(n - 1 - k) + \\
  C_{\text{app}}(k) \text{ if } n > 0 \text{ and } 0 \leq k \leq n - 1
  \]
Replacing values (already computed cost functions
\( C_{\text{app}}(x) = x + 1 \) and \( C_{\text{part}}(y) = y + 1 \):

\[
C_{\text{qsort}}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n + k + 2 + C_{\text{qsort}}(k) + C_{\text{qsort}}(n - 1 - k) & \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1
\end{cases}
\]
Non-deterministic cost relation:

\[
\begin{align*}
C_{qsort}(n) & = 1 \text{ if } n = 0 \\
C_{qsort}(n) & = n + k + 2 + C_{qsort}(k) + \\
& \quad C_{qsort}(n - 1 - k) \text{ if } n > 0 \text{ and } 0 \leq k \leq n - 1
\end{align*}
\]
Setting up and Solving Non-deterministic Recurrences
[ILPS'97]

Non-deterministic cost relation:

\[ C_{\text{qsort}}(n) = 1 \text{ if } n = 0 \]
\[ C_{\text{qsort}}(n) = n + k + 2 + C_{\text{qsort}}(k) + C_{\text{qsort}}(n - 1 - k) \text{ if } n > 0 \text{ and } 0 \leq k \leq n - 1 \]

Approach:

- Using *computation/evaluation trees* for such expression with nodes for each call \( C_{\text{qsort}}(n) \).
- Reasoning on sets of whole computation/evaluation trees and balancing/bounding the number of nodes and the node costs.
- Any computation tree (depth-first traversal) is characterized by a succession of values for \( k \).
- Any computation tree corresponding to such expression has:
  - \( n + 1 \) terminal nodes (\( C_{\text{qsort}}(0) \)) and
  - \( n \) non-terminal nodes (\( C_{\text{qsort}}(n), n > 0 \)).
Setting up and Solving Non-deterministic Recurrences

[ILPS'97]

- Non-deterministic cost relation:

  \[ C_{qsort}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\ 
  n + k + 2 + C_{qsort}(k) + \\ 
  C_{qsort}(n - 1 - k) & \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1 
\end{cases} \]

- To infer an upper bound on \( C_{qsort}(n) \) we replace \( n + k + 2 \) by an upper bound on it, \( n + k + 2 \leq 2n + 1 \) (by taking \( k = n - 1 \)), and set up:

  \[ C_{qsort}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\ 
  2n + 1 + 1 + C_{qsort}(n - 1) & \text{if } n > 0 
\end{cases} \]

  Our solver obtains a closed-form upper bound: \( C_{qsort}(n) = n^2 + 3n + 1 \).
Setting up and Solving Non-deterministic Recurrences
[ILPS'97]

- Non-deterministic cost relation:
  
  \[ C_{qsort}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  n + k + 2 + C_{qsort}(k) + C_{qsort}(n-1-k) & \text{if } n > 0 \text{ and } 0 \leq k \leq n-1 
  \end{cases} \]

- To infer a lower bound on \( C_{qsort}(n) \) we replace \( n + k + 2 \) by a lower bound on it \( 3 \leq n + k + 2 \) (with \( n = 1 \) and \( k = 0 \)), and set up:
  
  \[ C_{qsort}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  3 + 1 + C_{qsort}(n-1) & \text{if } n > 0 
  \end{cases} \]

  Our solver obtains a closed-form lower bound: \( C_{qsort}(n) = 4 \ n + 1 \).
Resource Analysis as an Abstract Interpretation

[TPLP'14, ICLP'13]

- In classical CiaoPP resource analysis the last steps (setting up and solving recurrences) were not implemented as an abstract domain.

- We have recently integrated resource analysis as an abstract domain "plug-in" of the generic analysis fixpoint –we get for free:
  - Multivariance: e.g., separate different call patterns for same block:
    \[
    \text{sort}(\text{lst}(\text{int}), \text{var}) \ldots \text{sort}(\text{lst}(\text{flt}), \text{var}) \ldots \text{sort}(\text{var}, \text{lst}(\text{int}))
    \]
  - Easier combination with other domains.
  - Easier integration w/static debugging/verification and rt-checking.
  - Many other engineering advantages.

- New domain for size analysis (sized types) that infers bounds on the size of data structures and substructures.

<table>
<thead>
<tr>
<th>listnum</th>
<th>sized(listnum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>listnum -&gt; []</td>
<td>\text{listnum}^{(\alpha,\beta)} \left(\text{num}^{(\gamma,\delta)}\right)</td>
</tr>
<tr>
<td>listnum -&gt; [num</td>
<td>listnum]</td>
</tr>
</tbody>
</table>

- Competitive results with state-of-the-art systems (e.g., RAML).

- Used in the XC energy analysis.
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta$</td>
</tr>
<tr>
<td>appAll</td>
<td>$a_1 a_2 a_3$</td>
<td>$a_1$</td>
<td>$b_1 b_2 b_3$</td>
</tr>
<tr>
<td>coupled</td>
<td>$\mu$</td>
<td>0</td>
<td>$\nu$</td>
</tr>
<tr>
<td>dyade</td>
<td>$\alpha_1 \alpha_2$</td>
<td>$\alpha_1 \alpha_2$</td>
<td>$\beta_1 \beta_2$</td>
</tr>
<tr>
<td>erathos</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>fib</td>
<td>$\phi^\mu$</td>
<td>$\phi^\mu$</td>
<td>$\phi^\nu$</td>
</tr>
<tr>
<td>hanoi</td>
<td>1</td>
<td>0</td>
<td>$2^\nu$</td>
</tr>
<tr>
<td>isort</td>
<td>$\alpha^2$</td>
<td>$\alpha^2$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>isortl</td>
<td>$a_1^2$</td>
<td>$a_1^2$</td>
<td>$b_1^2 b_2$</td>
</tr>
<tr>
<td>lisfact</td>
<td>$\alpha \gamma$</td>
<td>$\alpha$</td>
<td>$\beta \delta$</td>
</tr>
<tr>
<td>listnum</td>
<td>$\mu$</td>
<td>$\mu$</td>
<td>$\nu$</td>
</tr>
<tr>
<td>minsort</td>
<td>$\alpha^2$</td>
<td>$\alpha$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>nub</td>
<td>$a_1$</td>
<td>$a_1$</td>
<td>$b_1^2 b_2$</td>
</tr>
<tr>
<td>part</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta$</td>
</tr>
<tr>
<td>zip3</td>
<td>$\min(\alpha_i)$</td>
<td>0</td>
<td>$\min(\beta_i)$</td>
</tr>
</tbody>
</table>
Inferring Accumulated Cost [TPLP’16, FLOPS’16]

- Helping developers make (resource-related) design decisions:
  - Which parts of the program are the most resource-consuming?
  - Which predicates should be optimized first?

- The standard/classical notion of cost only partially meets these objectives:
  - Predicates w/highest (standard) costs may not need to be optimized first.
  - E.g., perhaps predicates with lower costs but which are called more often.
  - The input sizes to such calls are also relevant.

- Need info resulting from a static profiling of the program to:
  - identify the parts of a program responsible for highest fractions of the cost → accumulated cost.
  - I.e., how the total resource usage of the execution of a program is distributed over selected parts of it (cost centers → predicates).

Static profiling → static inference of the kinds of information that are usually obtained at run-time by profilers.

Main contribution

Novel, general, and flexible framework for setting up cost equations/relations. → can be instantiated for performing a wide range of static resource usage analyses, including both accumulated cost and standard cost.
Accumulated-cost: Intuition

\[
p(0).
p(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y).
\]

\[
q(0).
q(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y).
\]

\[
r(0).
r(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y).
\]

We want to know how the standard/total cost of \( p \) is distributed between the predicates of the program.

\[
\begin{array}{c}
p(2) \quad \text{1} \\
p(1) \quad \text{1} \\
r(1) \quad \text{1} \\
r(0) \quad \text{1} \\
r(0) \quad \text{1} \\
qu(1) \quad \text{1} \\
qu(0) \quad \text{1} \\
r(0) \quad \text{1} \\
r(0) \quad \text{1} \\
p(0) \quad \text{1} \\
q(0) \quad \text{1} \\
q(0) \quad \text{1}
\end{array}
\]
Accumulated-cost: Intuition

We declare that predicates \( p, q, \) and \( r \) are cost centers.

Cost centers are user-defined program points (predicates, in our case) to which execution costs are assigned during the execution of a program.
Accumulated-cost: Intuition

\[
p(0) .
\]
\[
p(X) :- X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y).
\]
\[
q(0) .
\]
\[
q(X) :- X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y).
\]
\[
r(0) .
\]
\[
r(X) :- X > 0, \ Y \text{ is } X - 1, \ r(Y).
\]

Set of cost centers:
\[
\diamondsuit = \{ p, q, r \}
\]

The cost of a call \( p(2) \) accumulated in cost center \( r \), denoted \( C_{p}^{r}(2) \)

Is the sum of the resolution steps that are descendant (in the call stack) of \( p(2) \), and whose closest ancestor in the call stack that is a cost center, is \( r \)
Accumulated-cost: Intuition

\[ p(0). \]
\[ p(X) :- X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y), \ p(Y). \]
\[ q(0). \]
\[ q(X) :- X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y). \]
\[ r(0). \]
\[ r(X) :- X > 0, \ Y \ is \ X - 1, \ r(Y). \]

Set of cost centers:
\[ \diamond = \{ p, q, r \} \]

The cost of a call \( p(2) \) accumulated in cost center \( r \rightarrow C^r_{p(2)} = 4 \)

Is the sum of the resolution steps that are descendant (in the call stack) of \( p(2) \), and whose closest ancestor in the call stack that is a cost center, is \( r \)
Accumulated-cost: Intuition

\[
\begin{align*}
p(0). \\
p(X) & : - X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y). \\
q(0). \\
q(X) & : - X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y). \\
r(0). \\
r(X) & : - X > 0, \ Y \text{ is } X - 1, \ r(Y).
\end{align*}
\]

Set of cost centers:
\[\diamondsuit = \{p, q, r\}\]

The cost of a call \(p(2)\) accumulated in cost center \(q\), denoted \(C^q_{p}(2)\) is the sum of the resolution steps that are descendant (in the call stack) of \(p(2)\), and whose closest ancestor in the call stack that is a cost center, is \(q\).

\[
\begin{array}{c}
p(2) \\
| \\
r(1) \\
| \\
r(0) \\
| \\
r(0) \\
| \\
q(0) \ 1 \\
| \\
r(0) \\
| \\
q(0) \ 1 \\
| \\
r(0) \\
| \\
q(0) \ 1 \\
| \\
p(0)
\end{array}
\]
Accumulated-cost: Intuition

\[
p(0).
p(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y), p(Y).
\]
\[
q(0).
q(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y).
\]
\[
r(0).
r(X) :- X > 0, Y \text{ is } X - 1, r(Y).
\]

Set of cost centers:
\[
\emptyset = \{p, q, r\}
\]

The cost of a call \(p(2)\) accumulated in cost center \(q\) \(\rightarrow c_p^q(2) = 3\)

Is the sum of the resolution steps that are descendant (in the call stack) of \(p(2)\), and whose closest ancestor in the call stack that is a cost center, is \(q\)
Accumulated-cost: Intuition

\[ p(0). \]
\[ p(X) :- X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y), \ p(Y). \]
\[ q(0). \]
\[ q(X) :- X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y). \]
\[ r(0). \]
\[ r(X) :- X > 0, \ Y \ is \ X - 1, \ r(Y). \]

Set of cost centers:  
\[ \set = \{ p, q, r \} \]

The cost of a call \( p(2) \) accumulated in cost center \( p \), denoted \( C_p^{p}(2) \)

Is the sum of the resolution steps that are descendant (in the call stack) of \( p(2) \), and whose closest ancestor in the call stack that is a cost center, is \( p \)
Accumulated-cost: Intuition

\[ p(0). \]
\[ p(X) :- \ X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y). \]
\[ q(0). \]
\[ q(X) :- \ X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y). \]
\[ r(0). \]
\[ r(X) :- \ X > 0, \ Y \text{ is } X - 1, \ r(Y). \]

Set of cost centers:
\[ \diamondsuit = \{ p, q, r \} \]

The cost of a call \( p(2) \) accumulated in cost center \( p \rightarrow \mathcal{C}_p^p(2) = 3 \)

Is the sum of the resolution steps that are descendant (in the call stack) of \( p(2) \), and whose closest ancestor in the call stack that is a cost center, is \( p \)
Accumulated-cost: Intuition

\[
p(0).
p(X) :- X > 0, \text{Y is } X - 1, r(Y), q(Y), p(Y).
q(0).
q(X) :- X > 0, \text{Y is } X - 1, r(Y), q(Y).
r(0).
r(X) :- X > 0, \text{Y is } X - 1, r(Y).
\]

Set of cost centers:
\[
\diamond = \{p, q, r\}
\]

\[
C_p(2) = C_{p_p}(2) + C_{p}(2) + C_{r}(2)
\]

\[
10 = 3 + 3 + 4
\]

\[
p(2) 1
\]

\[
r(2) 1
\]

\[
q(2) 1
\]

\[
p(1) 1
\]

\[
r(1) 1
\]

\[
r(0) 1
\]

\[
r(0) 1
\]

\[
q(0) 1
\]

\[
r(0) 1
\]

\[
r(0) 1
\]

\[
q(0) 1
\]

\[
r(0) 1
\]

\[
q(0) 1
\]

\[
p(0) 1
\]
Accumulated-cost: Definition

Definition: Accumulated Cost

The cost of a (single) call $p(n)$ accumulated in cost center $q$, denoted $C_p^q(n)$:

- Is the sum of the costs of all the computations that are descendants (in the call stack) of the call $p(n)$, and are under the scope of any call to $q$.
- We say that a computation is under the scope of a call to cost center $q$, if the closest ancestor of such computation in the call stack that is a cost center, is $q$.
- Expresses how much of the standard cost of the call to $p$ is attributed to $q$.
Application to Energy Analysis: Come to the Talk Tomorrow!
The Team

- Working specifically in CiaoPP resource analysis:
  - Pedro López-García
  - Manuel Hermenegildo
  - Maximiliano Klemen
  - Umer Liqat

- CiaoPP overall:
  - José-Francisco Morales
  - Nataliia Stulova
  - Isabel García-Contreras

- Previous main contributors to CiaoPP resource analysis:
  - Saumya Debray
  - Alejandro Serrano
  - Nai-wei Lin
  - Mario Méndez-Lojo
  - Jorge Navas
  - Edison Mera

Work currently at: IMDEA Software Institute, T.U. Madrid (UPM).
And previously at: U. T. Austin, MCC, U. of Arizona, U. of New Mexico.
Playground at: [http://play.ciao-lang.org](http://play.ciao-lang.org)
Timeline of our Work
<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>Method for static inference of upper-bound functions on execution cost and data structure sizes (<a href="#">PLDI’90</a>) (building on Wegbreit):</td>
</tr>
<tr>
<td></td>
<td>- Techniques for setting up, solving/approximating recurrence relations.</td>
</tr>
<tr>
<td></td>
<td>- For Horn-clause programs → used widely as IR for other languages.</td>
</tr>
<tr>
<td></td>
<td>- Motivation: task granularity control in automatic parallelization.</td>
</tr>
<tr>
<td></td>
<td>- Experimental results (resulting in improved parallel speedups).</td>
</tr>
<tr>
<td></td>
<td>- Implementation (leading to CASLOG) but I/O arguments, types, measures, etc. had to be provided by the user.</td>
</tr>
<tr>
<td>1993-</td>
<td><strong>First fully automatic system</strong>, including all auxiliary analyses: GraCos (Granularity Control System), implemented within CiaoPP (<a href="#">SAS’94, PASCO’94</a>).</td>
</tr>
<tr>
<td>1994</td>
<td>- Reducing data size computation overhead. (<a href="#">ICLP’95</a>)</td>
</tr>
<tr>
<td></td>
<td>- Further improvements. (<a href="#">JSC’96</a>)</td>
</tr>
<tr>
<td></td>
<td>- Precision improved w/determinacy, partial eval. . . . (<a href="#">LOPSTR’04, NGC’10</a>)</td>
</tr>
<tr>
<td>1997</td>
<td><strong>Lower bounds cost analysis; divide-and-conquer</strong>. (<a href="#">ILPS’97</a>)</td>
</tr>
<tr>
<td></td>
<td>- Lower bounds required developing non-failure (no-exceptions) analysis, guard coverage, ... (<a href="#">ICLP’97, FLOPS’04</a>)</td>
</tr>
<tr>
<td></td>
<td>- Also in [ILPS’97]: proposed non-deterministic recurrence relations, special for divide-and-conquer programs: looking at sets of computation trees and balancing/bounding node cost (e.g., quadratic bound for qsort).</td>
</tr>
</tbody>
</table>
### 1990
Method for static inference of upper-bound functions on execution cost and data structure sizes \([\text{PLDI}’90]\) (building on Wegbreit):

- Techniques for setting up, solving/approximating recurrence relations.
- For Horn-clause programs → used widely as IR for other languages.
- Motivation: task granularity control in automatic parallelization.
- Experimental results (resulting in improved parallel speedups).
- Implementation (leading to CASLOG) but I/O arguments, types, measures, etc. had to be provided by the user.

### 1993-1994
**First fully automatic system**, including all auxiliary analyses: **GraCos** (Granularity Control System), implemented within **CiaoPP** \([\text{SAS’94, PASCO’94}]\).

- Reducing data size computation overhead. \([\text{ICLP’95}]\)
- Further improvements. \([\text{JSC’96}]\)
- Precision improved w/determinacy, partial eval. . . . \([\text{LOPSTR’04, NGC’10}]\)

### 1997
**Lower bounds** cost analysis; **divide-and-conquer**. \([\text{ILPS’97}]\)

- Lower bounds required developing non-failure (no-exceptions) analysis, guard coverage, . . . \([\text{ICLP’97, FLOPS’04}]\)
- Also in \([\text{ILPS’97}]\): proposed *non-deterministic recurrence relations*, special for divide-and-conquer programs: looking at sets of computation trees and balancing/bounding node cost (e.g., quadratic bound for qsort).
1997- Verification: assert. lang, comp./run-time [AADEBUG'97, LOPSTR'99, ILPS-WS'97, LNCS'00, SAS'03]
    simple function comparisons (orders).

2003

2004  Abstraction carrying code for resources. [PPDP'05, LPAR'04]

2006  Probabilistic Cost Analysis. [CLEI'06]

2007  User-definable resources. [ICLP'07]

2007  Multi-language support (Java bytecode, C#, FP, CLP)
    via Horn clause-based IR. [LOPSTR'07]
    - Combined with user-definable resources:
      no need to develop specific analyzers for specific languages!
    - Instrumental analyses: sharing/nullity/class [VMCAI'08, PASTE'08] dependence [LCPC'08] shape
      [CC'08, SAS'02].

2008  Application to execution time (using bytecode-level models, obtained by regression). [PPDP'08]

2008  Application to energy consumption of Java bytecode. [NASA FM'08]

2009  User-definable resources for Java bytecode. [Bytecode'09]
<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Abstraction carrying code for resources.</td>
<td>PPDP'05, LPAR'04</td>
</tr>
<tr>
<td>2006</td>
<td>Probabilistic Cost Analysis.</td>
<td>CLEI'06</td>
</tr>
<tr>
<td>2007</td>
<td><strong>User-definable resources.</strong></td>
<td>ICLP'07</td>
</tr>
<tr>
<td>2007</td>
<td><strong>Multi-language support</strong> (Java bytecode, C#, FP, CLP) via <strong>Horn clause-based IR.</strong></td>
<td>LOPSTR'07</td>
</tr>
<tr>
<td></td>
<td>- Combined with user-definable resources: no need to develop specific analyzers for specific languages!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Instrumental analyses: sharing/nullity/class dependence shape</td>
<td>VMCAI'08, PASTE'08, LCPC'08, CC'08, SAS'02</td>
</tr>
<tr>
<td>2008</td>
<td>Application to <strong>execution time</strong> (using bytecode-level models, obtained by regression).</td>
<td>PPDP'08</td>
</tr>
<tr>
<td>2008</td>
<td>Application to <strong>energy consumption</strong> of Java bytecode.</td>
<td>NASA FM'08</td>
</tr>
<tr>
<td>2009</td>
<td>User-definable resources for Java bytecode.</td>
<td>Bytecode'09</td>
</tr>
<tr>
<td>Year</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
</tbody>
</table>
| 1997-2003 | **Verification:** assert. lang, comp./run-time [AADEBUG’97, LOPSTR’99, ILPS-WS’97, LNCS’00, SAS’03]
| 2003 | simple function comparisons (orders). |
| 2004 | Abstraction carrying code for resources. [PPDP’05, LPAR’04] |
| 2006 | Probabilistic Cost Analysis. [CLEI’06] |
| 2007 | **User-definable** resources. [ICLP’07] |
| 2007 | **Multi-language** support (Java bytecode, C#, FP, CLP) via **Horn clause-based IR**. [LOPSTR’07] |
| | • Combined with user-definable resources: no need to develop specific analyzers for specific languages! |
| | • Instrumental analyses: sharing/nullity/class [VMCAI’08, PASTE’08] dependence [LCPC’08] shape [CC’08, SAS’02]. |
| 2008 | Application to **execution time** (using bytecode-level models, obtained by regression). [PPDP’08] |
| 2008 | Application to **energy consumption** of Java bytecode. [NASA FM’08] |
| 2009 | **User-definable resources** for Java bytecode. [Bytecode’09] |
2010-2015 Resource verification: interval-based, improved function comparison. [ICLP’10, FOPARA’12, HIP3ES’15]

2012-2014 Cost analysis as multivariant abstract interpretation. [TPLP’14]

→ Multivariant, integrated with assertion checking, modular, incremental.

Domain: interval (piece-wise) functions. [ICLP’10, FOPARA’12]

2013 Using sized shapes (sized types). [ICLP’13]

2013-2016 Analysis and verification of Energy:

- At the ISA level [LOPSTR’13]
- Comparing LLVM and ISA levels [FOPARA’15]
- At the block level [HIP3ES’16]

2016 Static profiling / accumulated cost. [FLOPS’16, TPLP’16]
[ICLP’10, FOPARA’12, HIP3ES’15]

2012-2014 Cost analysis as multivariant abstract interpretation. [TPLP’14]

→ Multivariant, integrated with assertion checking, modular, incremental.

Domain: interval (piece-wise) functions. [ICLP’10, FOPARA’12]

2013 Using sized shapes (sized types). [ICLP’13]

2013-2016 Analysis and verification of Energy:

- At the ISA level [LOPSTR’13]
- Comparing LLVM and ISA levels [FOPARA’15]
- At the block level [HIP3ES’16]

2016 Static profiling / accumulated cost. [FLOPS’16, TPLP’16]
Thank you!
Selected Bibliography on CiaoPP
(and Additional Slides)
CiaoPP References – Analysis and Verification of Energy

Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks.

Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR.

Towards Energy Consumption Verification via Static Analysis.

Energy Consumption Analysis of Programs based on XMOS ISA-Level Models.

Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
CiaoPP References – Intermediate Repr. / Multi-Lingual Support

A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.

CiaoPP References – Analysis and Verification of Resources in General


A Transformational Approach to Parametric Accumulated-cost Static Profiling.

Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types.

Sized Type Analysis of Logic Programs (Technical Communication).
Interval-Based Resource Usage Verification: Formalization and Prototype
In Foundational and Practical Aspects of Resource Analysis. Second International Workshop
FOPARA 2011, Revised Selected Papers. Lecture Notes in Computer Science, 2012, 7177,
54–71, Springer.

In Technical Communications of the 26th ICLP. Leibniz Int’l. Proc. in Informatics (LIPIcs), Vol.
7, pages 104–113, Dagstuhl, Germany, July 2010.

User-Definable Resource Usage Bounds Analysis for Java Bytecode.
In Workshop on Bytecode Semantics, Verification, Analysis and Transformation
(BYTECODE’09), volume 253 of ENTCS, pages 6–86. Elsevier, March 2009.

Towards Execution Time Estimation in Abstract Machine-Based Languages.
In 10th Int’l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming

User-Definable Resource Bounds Analysis for Logic Programs.
In 23rd International Conference on Logic Programming (ICLP’07), LNCS Vol. 4670. Springer,
2007.

Probabilistic Cost Analysis of Logic Programs: A First Case Study.
In XXXII Latin-American Conference on Informatics (CLEI 2006), August 2006.

ICLP 2017 10-year Test of Time Award.
Lower Bound Cost Estimation for Logic Programs.

Estimating the Computational Cost of Logic Programs.

[ICLP’95] P. López-García and M. Hermenegildo.
Efficient Term Size Computation for Granularity Control.

A Methodology for Granularity Based Control of Parallelism in Logic Programs.

Towards Granularity Based Control of Parallelism in Logic Programs.

Task Granularity Analysis in Logic Programs.
CiaoPP References – Assertion Language

Assertion-based Debugging of Higher-Order (C)LP Programs.

[ICLP’09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.

An Assertion Language for Constraint Logic Programs.

An Assertion Language for Debugging of Constraint Logic Programs.
CiaoPP References – Overall Debugging and Verification Model


Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.
In Logic-based Program Synthesis and Transformation (LOPSTR’99), number 1817 in LNCS,

M. Hermenegildo, G. Puebla, and F. Bueno.
Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program
Validation and Debugging.

F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and
G. Puebla.
On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic
Programs.

CiaoPP References – Abstraction-Carrying Code

Reduced Certificates for Abstraction-Carrying Code.
In 22nd Intl. Conference on Logic Programming (ICLP 2006), number 4079 in LNCS, pages

M. Hermenegildo, E. Albert, P. López-García, and G. Puebla.
Abstraction Carrying Code and Resource-Awareness.

E. Albert, G. Puebla, and M. Hermenegildo.
Abstraction-Carrying Code.
CiaoPP References – Fixpoint-based Analyzer (Abstract Interpreter)

An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.

Incremental Analysis of Constraint Logic Programs.

Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.

Optimized Algorithms for the Incremental Analysis of Logic Programs.

Analyzing Logic Programs with Dynamic Scheduling.

Compile-time Derivation of Variable Dependency Using Abstract Interpretation.

On the Practicality of Global Flow Analysis of Logic Programs
CiaoPP References – Modular Analysis, Analysis of Concurrency

A Practical Type Analysis for Verification of Modular Prolog Programs.  

Context-Sensitive Multivariant Assertion Checking in Modular Programs.  

A Model for Inter-module Analysis and Optimizing Compilation.  

Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs.  

Analyzing Logic Programs with Dynamic Scheduling.  

Global Analysis of Standard Prolog Programs.  
CiaoPP References – Domains: Sharing/Aliasing

Identification of Logically Related Heap Regions.

Efficient Set Sharing using ZBDDs.
In *21st Int’l. WS on Languages and Compilers for Parallel Computing (LCPC’08)*, LNCS. Springer-Verlag, August 2008.

Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models.
In *21st Int’l. WS on Languages and Compilers for Parallel Computing (LCPC’08)*, LNCS. Springer-Verlag, August 2008.

Sharing Analysis of Arrays, Collections, and Recursive Structures.

Precise Set Sharing Analysis for Java-style Programs.

Efficient top-down set-sharing analysis using cliques.
CiaoPP References – Domains: Shape/Type Analysis

Sized Type Analysis of Logic Programs (Technical Communication).

Efficient context-sensitive shape analysis with graph-based heap models.

Heap Analysis in the Presence of Collection Libraries.

More Precise yet Efficient Type Inference for Logic Programs.
CiaoPP References – Domains: Non-failure, Determinacy

Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.

Determinacy Analysis for Logic Programs Using Mode and Type Information.

Multivariant Non-Failure Analysis via Standard Abstract Interpretation.

Non-Failure Analysis for Logic Programs.
The Cost Analysis Method for User-defined Resources [ICLP’07]
Resource Usage Equations for Procedures/Methods

- Consider a procedure/method $p$ defined by blocks $C_1, \ldots, C_m$.
- Then, the resource usage:
  - for a call to $p$,
  - with approximation $ap$ (upper/lower bounds),
  - expressed in units of resource $r$,
  - for input parameters of size $\bar{n}$ (a vector representing the sizes of input parameters) is:

$$RU_{pro}(p, ap, r, \bar{n}) = \bigcirc (ap)_{1 \leq i \leq m}\{RU_{block}(C_i, p, ap, r, \bar{n})\}$$

where $\bigcirc (ap)$ is a function that takes an approximation identifier $ap$ and returns a function which applies over all $RU_{block}(C_i, p, ap, r, \bar{n})$. 

General equation

$$RU_{pro}(p, ap, r, \bar{n}) = \bigcirc (ap)_{1 \leq i \leq m} \{RU_{block}(C_i, p, ap, r, \bar{n})\}$$

Particular case: Upper bound on execution time

$$RU_{pro}(p, ub, ex, \bar{n}) = \sum_{1 \leq i \leq m} \{RU_{block}(C_i, p, ub, ex, \bar{n})\}$$

- $ap = ub$ (upper bound)
- $r = ex$ (execution time)
- $\bigcirc (ub) = \sum$

For mutually exclusive blocks (using type based analysis)

$$RU_{pro}(p, ub, ex, \bar{n}) = \max_{1 \leq i \leq m} \{RU_{block}(C_i, p, ub, ex, \bar{n})\}$$
General equation

\[ RU_{\text{pro}}(p, ap, r, \bar{n}) = \bigcirc (ap)_{1 \leq i \leq m} \{ RU_{\text{block}}(C_i, p, ap, r, \bar{n}) \} \]

Particular case: Upper bound on execution time

\[ RU_{\text{pro}}(p, ub, ex, \bar{n}) = \sum_{1 \leq i \leq m} \{ RU_{\text{block}}(C_i, p, ub, ex, \bar{n}) \} \]

- \( ap = ub \) (upper bound)
- \( r = ex \) (execution time)
- \( \bigcirc (ub) = \sum \)

For mutually exclusive blocks (using type based analysis)

\[ RU_{\text{pro}}(p, ub, ex, \bar{n}) = \max_{1 \leq i \leq m} \{ RU_{\text{block}}(C_i, p, ub, ex, \bar{n}) \} \]
Resource Usage Equations for Blocks (no Backtracking)

Consider now a block $C \equiv H : S_1, \ldots, S_k$ of procedure $p$. Then:

$$ RU_{\text{block}}(C, ap, r, \bar{n}) = \text{closed\_form}(RU(p, ap, r, \bar{n})) $$

where:

$$ RU(p, ap, r, \bar{n}) = \delta(ap, r)(C, \bar{n}) + \lim_{\text{lim}(ap, C)} \sum_{i=1}^{\text{lim}(ap, C)} (\beta(ap, r)(S_i, \psi_i(\bar{n})) + RU_{\text{stm}}(S_i, ap, r, \psi_i(\bar{n}))) $$

- $\delta(ap, r)(C, \bar{n})$ is the resource usage of block head (e.g., parameter passing).
- $\delta(ap, r)$ is a function that takes an approximation identifier $ap$ and a resource identifier $r$ and returns a function
  - $\delta(ap, r) : block \times \bar{n} \rightarrow \text{arith\_expr}$
  - which takes a block and returns an arithmetic resource usage expression $< \text{arith\_expr} >$. 
Resource Usage Equations for Blocks (no Backtracking)

Consider now a block $C \equiv H : \cdots S_1, \ldots, S_k$ of procedure $p$. Then:

$$RU_{block}(C, ap, r, \bar{n}) = closed \_ form(RU(p, ap, r, \bar{n}))$$

where:

$$RU(p, ap, r, \bar{n}) = \delta(ap, r)(C, \bar{n}) + \\
\lim_{ap,C} \sum_{i=1}^{lim(ap,C)} (\beta(ap, r)(S_i, \psi_i(\bar{n})) + RU_{stm}(S_i, ap, r, \psi_i(\bar{n})))$$

- $\beta(ap, r)(S_i, \psi_i(\bar{n}))$ is the resource usage of preparing the call to statement $S_i$.
- $\psi_i(\bar{n})$ represents the sizes of the input arguments to $S_i$.
- $\beta(ap, r)$ is a function that takes an approximation identifier $ap$ and a resource identifier $r$ and returns a function

$$\beta(ap, r) : body \_ stm \times \bar{n} \rightarrow arith \_ expr$$

which takes a body statement and returns an arithmetic resource usage expression $< arith \_ expr >$. 
Resource Usage Equations for Blocks (no Backtracking)

Consider now a block $C \equiv H \leftarrow S_1, \ldots, S_k$ of procedure $p$. Then:

$$RU_{\text{block}}(C, ap, r, \bar{n}) = \text{closed\_form}(RU(p, ap, r, \bar{n})) \text{ where:}$$

$$RU(p, ap, r, \bar{n}) = \delta(ap, r)(C, \bar{n}) + \lim_{\text{lim}(ap, C)} \sum_{i=1}^{\infty} (\beta(ap, r)(S_i, \psi_i(\bar{n})) + RU_{\text{stm}}(S_i, ap, r, \psi_i(\bar{n})))$$

- $RU_{\text{stm}}(S_i, ap, r, \psi_i(\bar{n}))$ is:
  - If $S_i$ is a call to a recursive procedure $q$, $\rightarrow$ a symbolic expression $RU(q, ap, r, \psi_i(\bar{n}))$.
  - If $S_i$ is not recursive $\rightarrow$ the already computed resource usage function for $q$, say $\Phi$, applied to $\psi_i(\bar{n})$, i.e., $\Phi(\psi_i(\bar{n}))$.
  - In both cases, if there is a “trust” resource usage assertion for $S_i$,
    $\vdash$ trust pred $q$... + cost$(ap, r, \langle res\_usage\_func \rangle)$, then use it.
  - The most precise between the function in the trust assertion and the function inferred by the analysis is taken.
Resource Usage-Related Assertions

- :- resource r.
- :- head_cost(ap,r,δ(ap,r)).
- :- literal_cost(ap,r,β(ap,r)).
- :- trust ... + cost(ap,r,⟨arith_expr⟩).

Example

- :- resource steps.
- :- head_cost(ub,steps, 1).
- :- literal_cost(ub,steps, 0).
- :- trust pred append(X,Y,Z) : ( list(X), list(Y), var(Z) )
  + cost(ub,steps,length(X)+1).
General equation

\[ RU(p, ap, r, \bar{n}) = \delta(ap, r)(C, \bar{n}) + \]
\[ \lim_{lim(ap, C)} \sum_{i=1}^{\beta(ap, r)(S_i, \psi_i(\bar{n})) + RU_{stm}(S_i, ap, r, \psi_i(\bar{n}))} \]

Particular case: Upper bound on execution time

- \( ap = ub \) and \( r = ex \) :- resource ex.
- Each \( S_i \) is a call to bytecode \( B \) or a call to a block procedure.
- Assertions for each bytecode \( B \):
  
  :- trust pred \( B \) + cost(ub,ex, \( T(B) \))

- \( \delta(ub, ex) = \gamma \) :- head_cost(ub,ex,\( \gamma \)).
- \( \gamma \) gives the exec. time of determining that the previous blocks to \( C \) of procedure \( p \) will not yield a solution.
- \( \beta(ub, ex) \) is the “zero” function \( \rightarrow \) :- literal_cost(ub,ex,0).
- \( \lim(ap, C) = k \) (all statements are taken into account).
Consider now a block \( C \equiv H :- S_1, \ldots, S_k \). Then, the resource usage of block \( C \) of procedure \( p \) is

\[
RU_{\text{block}}(C, ap, r, \bar{n}) = \text{closed\_form}(RU(p, ap, r, \bar{n}))
\]

where:

\[
RU(p, ap, r, \bar{n}) = \delta(ap, r)(C, \bar{n}) + \lim_{ap \rightarrow C} \sum_{i=1}^{\text{lim}(ap,C)} \left( \prod_{j \prec i} \text{Sols}_{S_j}(\psi_j(\bar{n})) \right) \beta(ap, r)(S_i, \psi_i(\bar{n})) + RU_{\text{stm}}(S_i, ap, r, \psi_i(\bar{n}))
\]

- \( \text{Sols}_{S_j}(\psi_j(\bar{n})) \) is the number of solutions statement \( S_j \) can generate.
- \( \prod_{j \prec i} \text{Sols}_{S_j}(\psi_j(\bar{n})) \) is the number of times the statement \( S_i \) will be executed (it depends on the number of solutions of the previous statements, because of alternative paths).
Using Non-Deterministic Recurrences (more details)
In some cases, classical approaches to cost analysis based on setting up recurrence equations infer very conservative upper/lower bounds.

For example, when dealing with (the frequent) divide-and-conquer programs in which the sizes of the output arguments of the “divide” part are dependent.

The classical approach performs an independent size analysis for output arguments

→ very conservative upper/lower bounds.

Our approach for more accurate/useful estimations:

► Using a relational size analysis for output arguments.

→ size relations involve sizes of (several) output arguments (specially for divide procedures).

► Setting up *non-deterministic recurrence relations* (both for cost and size).

► Reasoning on sets of whole computation/evaluation trees and balancing/bounding the number of nodes and the node costs.
Running Example: Quicksort

:- entry qsort/2 : list(num) * var.

qsort([],[]).
qsort([H|L],R) :- part(L,H,L1,L2), qsort(L1,R1), qsort(L2,R2), app(R1,[H|R2],R).

part([],C,[],[]).
part([E|T],C,[E|L],R) :- E < C, part(T,C,L,R).
part([E|T],C,L,[E|R]) :- E >= C, part(T,C,L,R).

app([],L,L).
app([H|A],B,[H|C]) :- app(A,B,C).
Classical Size Analysis of \texttt{append/3} (upper/lower bounds)

\begin{verbatim}
append([],L,L).
append([H|A],B,[H|C]) :- append(A,B,C).
\end{verbatim}

- Let \( x \) and \( y \) be the size (length) of the input lists (1st and 2nd arguments).
- The size of the third argument of \texttt{append/3} (\( Sz^{app}_3(x, y) \)) is:

\[
Sz^{app}_3(0, y) = y \quad (1st \, clause, \, boundary \, condition)
\]
\[
Sz^{app}_3(x, y) = 1 + Sz^{app}_3(x - 1, y) \quad (2nd \, clause)
\]

- For the second equation we use the relation:

\[
sz([H|C]) = \text{diff}([H|C], C) + sz(C) = 1 + sz(C)
\]

- Solution: \( Sz^{app}_3(x, y) = x + y \).
Size Analysis of \texttt{partition/4} (upper/lower bounds)

\texttt{partition([ ]},C,[ ]),[ ]).
\texttt{partition([E | T],C,[E | L],R)} : \neg E < C, \texttt{partition(T,C,L,R)}.
\texttt{partition([E | T],C,L,[E | R])} : \neg E \geq C, \texttt{partition(T,C,L,R)}.

- Let $x$ and $y$ be the sizes of the input arguments (1st and 2nd arguments).
- Let $Sz_3^{part}(x, y)$ be the size of the 3rd (output) argument as a function of the sizes of the two input arguments.

\textbf{Recurrence equations:}

\begin{align*}
Sz_3^{part}(0, y) &= 0 \quad (1st \ clause, \ boundary \ condition) \\
Sz_3^{part}(x, y) &= Sz_3^{part}(x - 1, y) + 1 \quad (2nd \ clause) \\
Sz_3^{part}(x, y) &= Sz_3^{part}(x - 1, y) \quad (3rd \ clause)
\end{align*}

- Upper-bound closed-form:
  $Sz_3^{part}(x, y) = x$ $y$ is ignored $\rightarrow$ $Sz_3^{part}(x) = x$

- Lower-bound closed-form:
  $Sz_3^{part}(x, y) = 0$ $y$ is ignored $\rightarrow$ $Sz_3^{part}(x) = 0
Classical Size Analysis of \texttt{partition/4 (contd.)}

\begin{verbatim}
\texttt{partition([],C,[],[])}.
\texttt{partition([E | T],C,[E | L],R) :- E < C, partition(T,C,L,R).}
\texttt{partition([E | T],C,L,[E | R]) :- E >= C, partition(T,C,L,R).}
\end{verbatim}

- Similar process for the size of the 4th argument of \texttt{partition/4}.
- Recurrence equations:

\begin{align*}
\text{Sz}_{\text{part}}^4(0, y) &= 0 \quad (1st \text{ clause, boundary condition}) \\
\text{Sz}_{\text{part}}^4(x, y) &= \text{Sz}_{\text{part}}^4(x - 1, y) \quad (2nd \text{ clause}) \\
\text{Sz}_{\text{part}}^4(x, y) &= \text{Sz}_{\text{part}}^4(x - 1, y) + 1 \quad (3rd \text{ clause})
\end{align*}

- Upper-bound closed-form:
  \begin{align*}
  \text{Sz}_{\text{part}}^4(x, y) &= x \quad y \text{ is ignored} \rightarrow \text{Sz}_{\text{part}}^4(x) = x
  
\end{align*}

- Lower-bound closed-form:
  \begin{align*}
  \text{Sz}_{\text{part}}^4(x, y) &= 0 \quad y \text{ is ignored} \rightarrow \text{Sz}_{\text{part}}^4(x) = 0
\end{align*}
Classical Size Analysis of \( q\text{sort}/2 \) (upper bounds)

\[
\text{qsor}[\text{t}][2](\text{[]}[,\text{[]}]) . \\
\text{qsor}[\text{t}][2](\text{[H|L]},\text{R}) : - \text{partition}(\text{L},\text{H},\text{L1},\text{L2}), \\
\text{qsor}[\text{t}][2](\text{L1},\text{R1}), \text{qsor}[\text{t}][2](\text{L2},\text{R2}), \\
\text{append}(\text{R1},[\text{H|R2}],\text{R}) .
\]

\[
\begin{align*}
 Sz_{qsort}^2(0) & = 0 \\
 Sz_{qsort}^2(x) & = Sz_{qsort}^2(Sz_{part}^2(x - 1)), 1 + Sz_{qsort}^2(Sz_{part}^2(x - 1))) \\
 & = Sz_{qsort}^2(x - 1), 1 + Sz_{qsort}^2(x - 1)) \\
 & = Sz_{qsort}^2(x - 1) + 1 + Sz_{qsort}^2(x - 1) \\
 & = 2 \cdot Sz_{qsort}^2(x - 1) + 1
\end{align*}
\]

- Solution: \( Sz_{qsort}^2(x) = 2^x - 1 \)
- Clearly an overestimation, due to the overestimation of the upper bounds on the output arguments of \( \text{partition}/4 \).
Classical Cost Analysis (upper bounds)

\begin{align*}
\text{partition}([ ], C, [ ], [ ]). \\
\text{partition}([E \mid T], C, [E \mid L], R) :& \quad E < C, \text{ partition}(T, C, L, R). \\
\text{partition}([E \mid T], C, L, [E \mid R]) :& \quad E \geq C, \text{ partition}(T, C, L, R).
\end{align*}

\begin{align*}
C_{\text{part}}(0, y) &= 1 \\
C_{\text{part}}(x, y) &= 1 + C_{\text{part}}(x - 1, y) \\
C_{\text{part}}(x, y) &= 1 + C_{\text{part}}(x - 1, y)
\end{align*}

• Solution: \( C_{\text{part}}(x, y) = x + 1 \quad \text{\( y \) is ignored} \rightarrow C_{\text{part}}(x) = x + 1 \)
Classical Cost Analysis of `append/3` (upper bounds)

\[
\begin{align*}
append([], L, L). \\
append([H|A], B, [H|C]) & : \leftarrow append(A, B, C).
\end{align*}
\]

\[
\begin{align*}
C_{app}(0, y) & = 1 \\
C_{app}(x, y) & = 1 + C_{app}(x - 1, y)
\end{align*}
\]

Solution: \( C_{app}(x, y) = x + 1 \) \quad y \text{ is ignored} \rightarrow \( C_{app}(x) = x + 1 \)
Classical Cost Equations for \texttt{qsort/2} (upper bounds)

\begin{align*}
\text{qsort}([], []). \\
\text{qsort}([H|L], R) & : \text{partition}(L,H,L1,L2), \\
& \quad \text{qsort}(L1, R1), \\
& \quad \text{qsort}(L2, R2), \\
& \quad \text{append}(R1, [H|R2], R).
\end{align*}

\begin{align*}
C_{\text{qsort}}(0) &= 1 \\
C_{\text{qsort}}(x) &= 1 + C_{\text{part}}(x − 1) + C_{\text{qsort}}(Sz^3_{\text{part}}(x − 1)) + C_{\text{qsort}}(Sz^4_{\text{part}}(x − 1)) + \\
& \quad C_{\text{app}}(Sz^2_{\text{qsort}}(Sz^3_{\text{part}}(x − 1))) \\
& = 1 + x + C_{\text{qsort}}(x − 1) + C_{\text{qsort}}(x − 1) + C_{\text{app}}(Sz^2_{\text{qsort}}(x − 1)) \\
& = 1 + x + 2 \cdot C_{\text{qsort}}(x − 1) + C_{\text{app}}(2^{x−1} − 1) \\
& = 1 + x + 2^{x−1} + 2 \cdot C_{\text{qsort}}(x − 1)
\end{align*}

- Solution: \( C_{\text{qsort}}(x) = (x + 3) \cdot 2^x − x − 2 \)
- Clearly an overestimation.
Setting up Non-deterministic Recurrences

\[
\text{qsort}([],[]).
\]
\[
\text{qsort}([H|L],R) :- \text{partition}(L,H,L1,L2),
\text{qsort}(L1,R1),
\text{qsort}(L2,R2),
\text{append}(R1,[H|R2],R).
\]

- We use a relational size analysis for output arguments:
  \[
  \text{part}(L,H,L1,L2) \rightarrow \text{length}(L) = \text{length}(L1) + \text{length}(L2)
  \]
  \[\begin{align*}
  & \text{Let } \text{length}(L) = n', \ n' \geq 0, \text{ and } \text{length}(L1) = k, \ 0 \leq k \leq n'. \\
  & \text{then } \text{length}(L2) = n' - k.
  \end{align*}\]

- Non-deterministic cost relations \((n' = n - 1)\):

\[
\begin{align*}
\text{C}_{\text{qsort}}(n) &= 1 \text{ if } n = 0 \\
\text{C}_{\text{qsort}}(n) &= 1 + \text{C}_{\text{part}}(n-1) + \text{C}_{\text{qsort}}(k) + \text{C}_{\text{qsort}}(n-1-k) + \\
& \quad \text{C}_{\text{app}}(k) \text{ if } n > 0 \text{ and } 0 \leq k \leq n - 1
\end{align*}
\]
Solving Non-deterministic Recurrences

- Non-deterministic cost relations:

\[
C_{\text{qsort}}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
1 + C_{\text{part}}(n - 1) + C_{\text{qsort}}(k) + C_{\text{qsort}}(n - 1 - k) + C_{\text{app}}(k) & \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1 
\end{cases}
\]
Replacing values (already computed cost functions $C_{\text{app}}(x) = x + 1$ and $C_{\text{part}}(y) = y + 1$):

$$C_{\text{qsort}}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n + k + 2 + C_{\text{qsort}}(k) + C_{\text{qsort}}(n - 1 - k) & \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1 \end{cases}$$
Solving Non-deterministic Recurrences

- Replacing values (already computed cost functions $C_{app}(x) = x + 1$ and $C_{part}(y) = y + 1$):

  \[
  C_{qsort}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  n + k + 2 + C_{qsort}(k) + C_{qsort}(n - 1 - k) & \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1
  \end{cases}
  \]

- To infer an upper bound on $C_{qsort}(n)$ we replace $n + k + 2$ by an upper bound on it, $n + k + 2 \leq 2n + 1$ (by taking $k = n - 1$), and set up:

  \[
  C_{qsort}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  2n + 1 + 1 + C_{qsort}(n - 1) & \text{if } n > 0
  \end{cases}
  \]

  Our solver obtains a closed-form upper bound: $C_{qsort}(n) = n^2 + 3n + 1$. 
Solving Non-deterministic Recurrences

- Replacing values (already computed cost functions $C_{app}(x) = x + 1$ and $C_{part}(y) = y + 1$):

\[
\begin{align*}
C_{qsort}(n) &= 1 \text{ if } n = 0 \\
C_{qsort}(n) &= n + k + 2 + C_{qsort}(k) + C_{qsort}(n - 1 - k) \text{ if } n > 0 \text{ and } 0 \leq k \leq n - 1
\end{align*}
\]

- To infer a lower bound on $C_{qsort}(n)$ we replace $n + k + 2$ by a lower bound on it $3 \leq n + k + 2$ (with $n = 1$ and $k = 0$), and set up:

\[
\begin{align*}
C_{qsort}(n) &= 1 \text{ if } n = 0 \\
C_{qsort}(n) &= 3 + 1 + C_{qsort}(n - 1) \text{ if } n > 0
\end{align*}
\]

Our solver obtains a closed-form lower bound: $C_{qsort}(n) = 4 \ n + 1$. 
Solving Non-deterministic Recurrences

Consider the non-deterministic recurrence:

\[ \phi(0) = C \]
\[ \phi(n) = \phi(n - 1 - k) + \phi(k) \quad \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1 \]

where \( k \) is an arbitrary value and \( C \) is a constant.

A computation tree for such an expression is a tree in which

- each non-terminal node is labeled with \( \phi(n) \), \( n > 0 \), and has two children \( \phi(n - 1 - k) \) and \( \phi(k) \) (left- and right-hand-side respectively), and
- terminal nodes are labeled with \( \phi(0) \).

Assume that we construct a tree for \( \phi(n) \) following a depth-first traversal.

In each non-terminal node, we (arbitrarily) chose a value for \( k \) such that \( 0 \leq k \leq n - 1 \).

We say that the computation succession of the tree (in depth-first traversal) is the succession of values that have been chosen for \( k \) in chronological order.

This way, each computation tree can be characterized by its computation succession (of \( k \)'s).
Given the non-deterministic recurrence:
\[
\phi(0) = C \\
\phi(n) = \phi(n - 1 - k) + \phi(k) \quad \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1
\]
where \( k \) is an arbitrary value and \( C \) is a constant.

Any computation tree corresponding to such expression has \( n + 1 \) terminal nodes and \( n \) non-terminal nodes.

- Proof by induction on \( n \).

Closed-form solution: \( \phi(n) = (n + 1) \times C \).

- Proof: any computation tree has \( n + 1 \) terminal nodes labeled with \( \phi(0) \) and the evaluation of each of these terminal nodes is \( C \).
Given the non-deterministic recurrence:
\[
\phi(0) = C, \\
\phi(n) = \phi(n - 1 - k) + \phi(k) + g(k) \quad \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1
\]
where \( k \) is an arbitrary value, \( g(k) \) a function, and \( C \) a constant.

For any computation tree corresponding to it:
\[
\phi(n) = (n + 1) \times C + \sum_{i=1}^{n} g(k_i)
\]
where \( \{k_i\}_{i=1}^{n} \) is the computation succession of the tree.

Proof:

- Any computation tree has \( n + 1 \) terminal nodes and \( n \) non-terminal nodes.
- The evaluation of each terminal node yields the value \( C \) and each time a non-terminal node \( i \) is evaluated, \( g(k_i) \) is added.

If \( g(k) \) is an increasing monotonic function then:
The succession \( \{k_i\}_{i=1}^{n} \), where \( k_i = 0 \) (respectively \( k_i = n - 1 \)) for all \( 1 \leq i \leq n \) gives the minimum (respectively maximum) value for \( \phi(n) \) of all computation trees.
We can replace $g(k)$ by any lower/upper bound on it to compute a lower/upper bound on $\phi(n)$.

We can also take any lower/upper bound on each $g(k_i)$.

For example, if $g(k)$ is an increasing monotonic function then $g(k_i) \leq g(n-1)$ and $g(k_i) \geq g(0)$ for $1 \leq i \leq n$, thus:

- $\phi(n) \leq (n + 1) \times C + n \times g(n - 1)$ and
- $\phi(n) \geq (n + 1) \times C + n \times g(0)$. 
Solving Non-deterministic Recurrences (contd.)

- Given the non-deterministic recurrence:
  \[
  \phi(0) = C, \\
  \phi(n) = \phi(n - 1 - k) + \phi(k) + g(n, k) \quad \text{if } n > 0 \text{ and } 0 \leq k \leq n - 1
  \]
  where \( k \) is an arbitrary value, \( g(n, k) \) a function, and \( C \) a constant.

- If \( L \) is a lower (resp. upper) bound on \( g(n, k) \), then the solution of:
  \[
  f(0) = C, \\
  f(n) = f(n - 1) + C + L \quad \text{for } n > 0,
  \]
  is a lower (resp. upper) bound on \( \phi(n) \) for all \( n \geq 0 \) and for any computation tree corresponding to \( \phi(n) \).

- In particular, if \( g(n, k) \) is an increasing monotonic function, then
  \[
  L \equiv g(1, 0) \quad \text{(resp. } L \equiv g(n, n - 1) \text{)} \quad \text{is a lower (resp. upper) bound on } g(n, k).
  \]
Combination with Ranking Functions: Solving Recurrences with Increasing Arguments (more details)
Solving Recurrences with Increasing Argument Sizes

Given the following program (in Horn Clause form):

```prolog
sum(I, N, A, A):-
    I > N.
sum(I, N, AI, AO):-
    I =< N,
    I1 is I + 1,
    AT is AI + I,
    sum(I1, N, AT, AO).
```

We want to estimate the cost of a call `sum(I, N, 0, A)`

The system derives the recurrence relation:

\[
C_{sum}(I, N) = 1 \quad \text{if } I > N
\]
\[
C_{sum}(I, N) = 1 \times C_{sum}(I + 1, N) + 1 \quad \text{if } I \leq N
\]

Which matches the following pattern:

\[
f(\bar{u}) = C \quad \text{if } \varphi(\bar{u})
\]
\[
f(\bar{u}) = K \times f(h(\bar{u})) + g(\bar{u}) \quad \text{if } \varphi'(\bar{u})
\]
Solving Recurrences with Increasing Argument Sizes

Given the following program (in Horn Clause form):

```prolog
sum(I, N, A, A):-
  I > N.
sum(I, N, AI, AO):-
  I =< N,
  I1 is I + 1,
  AT is AI + I,
  sum(I1, N, AT, AO).
```

We want to estimate the cost of a call `sum(I, N, 0, A)`
The system derives the recurrence relation:

\[
C_{\text{sum}}(I, N) = 1 \quad \text{if } I > N \\
C_{\text{sum}}(I, N) = 1 \times C_{\text{sum}}(I + 1, N) + 1 \quad \text{if } I \leq N
\]

Which matches the following pattern:

\[
f(u) = C \quad \text{if } \varphi(u) \\
f(u) = K \times f(h(u)) + g(u) \quad \text{if } \varphi'(u)
\]
Solving Recurrences with Increasing Argument Sizes

Given the following program (in Horn Clause form):

\[
\text{sum}(I, N, A, A) :- \\
\quad I > N.
\]

\[
\text{sum}(I, N, AI, AO) :- \\
\quad I \leq N, \\
\quad I1 \text{ is } I + 1, \\
\quad AT \text{ is } AI + I, \\
\quad \text{sum}(I1, N, AT, AO).
\]

We want to estimate the cost of a call \(\text{sum}(I, N, 0, A)\)

The system derives the recurrence relation:

\[
\begin{align*}
C_{\text{sum}}(I, N) &= 1 & \text{if } I > N \\
C_{\text{sum}}(I, N) &= 1 \times C_{\text{sum}}(I + 1, N) + 1 & \text{if } I \leq N
\end{align*}
\]

Which matches the following pattern:

\[
\begin{align*}
\phi(u) &= C & \text{if } \varphi(u) \\
\phi(u) &= K \times \phi(h(u)) + g(u) & \text{if } \varphi'(u)
\end{align*}
\]
Recurrences with Increasing Argument Sizes (contd.)

Recurrence relation (pattern)

\[
\begin{align*}
  f(\bar{u}) &= C & \text{if } \varphi(\bar{u}) \\
  f(\bar{u}) &= K \times f(h(\bar{u})) + g(\bar{u}) & \text{if } \varphi'(\bar{u})
\end{align*}
\]

- \(K\) and \(C\) are constants, \(h\) a linear arithmetic function over \(\bar{u}\), and
- \(g\) is a non-recursive arithmetic function.
- \(\varphi(\bar{u})\) and \(\varphi'(\bar{u})\) are constraints.

Equivalent to:

\[
f(\bar{u}) = K^i \times C + K^{i-1} \times g(h^{i-1}(\bar{u})) + \cdots + K \times g(h(\bar{u})) + g(\bar{u})
\]

Where \(i\) represents the number of applications of the recursive case.
Recurrences with Increasing Argument Sizes (contd.)

### Recurrence relation (pattern)

\[
\begin{align*}
    f(\overline{u}) &= C & \text{if } \varphi(\overline{u}) \\
    f(\overline{u}) &= K \times f(h(\overline{u})) + g(\overline{u}) & \text{if } \varphi'(\overline{u})
\end{align*}
\]

- $K$ and $C$ are constants, $h$ a linear arithmetic function over $\overline{u}$, and
- $g$ is a non-recursive arithmetic function.
- $\varphi(\overline{u})$ and $\varphi'(\overline{u})$ are constraints.

### Equivalent to:

\[
f(\overline{u}) = K^i \times C + K^{i-1} \times g(h^{i-1}(\overline{u})) + \cdots + K \times g(h(\overline{u})) + g(\overline{u})
\]

- Where $i$ represents the *number of applications* of the recursive case.
Recurrences with Increasing Argument Sizes (contd.)

Equivalent to:

\[
f(\overline{u}) = K^i \times C + K^{i-1} \times g(h^{i-1}(\overline{u})) + \cdots + K \times g(h(\overline{u})) + g(\overline{u})
\]

- Where \( i \) represents the number of applications of the recursive case.

In order to bound such expression:

- We find a ranking function \( r(\overline{u}) \) that is an upper bound of \( i \) (using, e.g., [Podelski 2004]), and
- We maximize \( g(\overline{u}) \) w.r.t. the constraint \( \varphi'(\overline{u}) \), obtaining \( M \).

\[
f(\overline{u}) \leq \begin{cases} 
K^{r(\overline{u})} \times C + \sum_{j=0}^{r(\overline{u})-1} K^j \times M & \text{if } \varphi'(\overline{u}) \\
C & \text{otherwise}
\end{cases}
\]
Recurrences with Increasing Argument Sizes (contd.)

Equivalent to:

\[ f(\overline{u}) = K^i \times C + K^{i-1} \times g(h^{i-1}(\overline{u})) + \cdots + K \times g(h(\overline{u})) + g(\overline{u}) \]

- Where \( i \) represents the number of applications of the recursive case.

In order to bound such expression:

- We find a ranking function \( r(\overline{u}) \) that is an upper bound of \( i \) (using, e.g., [Podelski 2004]), and
- We maximize \( g(\overline{u}) \) w.r.t. the constraint \( \varphi'(\overline{u}) \), obtaining \( M \).

\[ f(\overline{u}) \leq \begin{cases} 
    K^{r(\overline{u})} \times C + \sum_{j=0}^{r(\overline{u})-1} K^j \times M & \text{if } \varphi'(\overline{u}) \\
    C & \text{otherwise}
\end{cases} \]
Recurrences with Increasing Argument Sizes (contd.)

Equivalent to:

\[ f(\bar{u}) = K^i \times C + K^{i-1} \times g(h^{i-1}(\bar{u})) + \cdots + K \times g(h(\bar{u})) + g(\bar{u}) \]

- Where \( i \) represents the \textit{number of applications} of the recursive case.

In order to bound such expression:

- We find a \textit{ranking function} \( r(\bar{u}) \) that is an \textit{upper bound} of \( i \) (using, e.g., [Podelski 2004]), and
- We \textit{maximize} \( g(\bar{u}) \) w.r.t. the constraint \( \varphi'(\bar{u}) \), obtaining \( M \).

\[ f(\bar{u}) \leq \begin{cases} K^{r(\bar{u})} \times C + \sum_{j=0}^{r(\bar{u})-1} K^j \times M & \text{if } \varphi'(\bar{u}) \\ C & \text{otherwise} \end{cases} \]
Recurrences with Increasing Argument Sizes (contd.)

For our example:

\[
\begin{align*}
C_{sum}(I, N) &= 1 & \text{if } I > N \\
C_{sum}(I, N) &= 1 \times C_{sum}(I + 1, N) + 1 & \text{if } I \leq N
\end{align*}
\]

We have:

- \( r(I, N) = N - I \)
- \( K = C = g(I, N) = 1 \)
- \( h(I, N) = (I + 1, N) \)
- \( M = 1 \) (Maximize \( g(I, N) = 1 \) given \( I \leq N \))

Obtaining:

\[
\begin{align*}
f(I, N) \leq & \begin{cases} 
N - I + 1 & \text{if } I \leq N \\
1 & \text{otherwise}
\end{cases}
\end{align*}
\]
Recurrences with Increasing Argument Sizes (contd.)

For our example:

\[ C_{\text{sum}}(I, N) = 1 \quad \text{if } I > N \]
\[ C_{\text{sum}}(I, N) = 1 \times C_{\text{sum}}(I + 1, N) + 1 \quad \text{if } I \leq N \]

We have:

- \( r(I, N) = N - I \)
- \( K = C = g(I, N) = 1 \)
- \( h(I, N) = (I + 1, N) \)
- \( M = 1 \) (Maximize \( g(I, N) = 1 \) given \( I \leq N \))

Obtaining:

\[ f(I, N) \leq \begin{cases} N - I + 1 & \text{if } I \leq N \\ 1 & \text{otherwise} \end{cases} \]
Recurrences with Increasing Argument Sizes (contd.)

For our example:

\[ C_{\text{sum}}(I, N) = 1 \quad \text{if} \quad I > N \]
\[ C_{\text{sum}}(I, N) = 1 \times C_{\text{sum}}(I + 1, N) + 1 \quad \text{if} \quad I \leq N \]

We have:

- \( r(I, N) = N - I \)
- \( K = C = g(I, N) = 1 \)
- \( h(I, N) = (I + 1, N) \)
- \( M = 1 \) (Maximize \( g(I,N) = 1 \) given \( I \leq N \))

Obtaining:

\[
f(I, N) \leq \begin{cases} 
N - I + 1 & \text{if} \ I \leq N \\
1 & \text{otherwise}
\end{cases}
\]
The Ciao Assertions Model
**Definition**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is prt. correct w.r.t. $\mathcal{I}<em>\alpha$ if $\alpha([P]) \leq \mathcal{I}</em>\alpha$</td>
<td>$[P]<em>{\alpha^+} \leq \mathcal{I}</em>\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $\mathcal{I}<em>\alpha$ if $\mathcal{I}</em>\alpha \leq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \leq [P]</em>{\alpha^+}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $\mathcal{I}<em>\alpha$ if $\alpha([P]) \not\leq \mathcal{I}</em>\alpha$</td>
<td>$[P]<em>{\alpha^+} \not\leq \mathcal{I}</em>\alpha$, or $[P]<em>{\alpha^+} \cap \mathcal{I}</em>\alpha = \emptyset \land [P]_{\alpha^+} \neq \emptyset$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $\mathcal{I}<em>\alpha$ if $\mathcal{I}</em>\alpha \not\leq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \not\leq [P]</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>

[ADEBUG’97, LNCS’99, LOPSTR’99, PBH00a, SAS’03, PPDP’05, LPAR’06, PEPM’08, ICLP’09, PPDP’14, TPLP’15, PPDP’16]
Integrated Static/Dynamic Debugging and Verification

- Based throughout on the notion of safe approximation (abstraction).
- Run-time checks generated for parts of asserts. not verified statically.
- Diagnosis (for both static and dynamic errors).
- Comparison not always trivial: e.g., resource debugging/certification
  - Need to compare functions.
  - “Segmented” answers.

[AADEBUG'97, LNCS'99, LOPSTR'99, PBH00a, SAS'03, PPDP'05, LPAR'06, PEPM’08, ICLP’09, PPDP'14, TPLP’15, PPDP'16]
Resource Verification and Function Comparison
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
Let $\Psi_1(n)$ and $\Psi_2(n)$ be (resource usage bound) functions.

**Objective:** determine intervals for $n$ in which:

$\Psi_1(n) > \Psi_2(n)$ or $\Psi_1(n) = \Psi_2(n)$ or $\Psi_1(n) < \Psi_2(n)$.

**Approach:**

- Define $f(n) = \Psi_1(n) - \Psi_2(n)$.
- **Find roots of $f(n)$** — assume $m$ roots, $n_1, \ldots, n_m$.
  (These roots are the intersection points of $\Psi_1(n)$ and $\Psi_2(n)$.)
- Consider the intervals:
  $S_1 = [0, n_1), S_2 = (n_1, n_2), \ldots, S_m = (n_{m-1}, n_m), S_{m+1} = (n_m, \infty)$.
- For each interval $S_i$, $1 \leq i \leq m$, we select a value $v_i$ in the interval.
- If $f(v_i) > 0$ (respectively $f(v_i) < 0$),
  then $\Psi_1(n) > \Psi_2(n)$ (respectively $\Psi_1(n) < \Psi_2(n)$) for all $n \in S_i$. 
Finding Roots

- **Polynomial functions → GNU Scientific Library:**
  - uses analytical methods for polynomials up to order four, and
  - numerical methods for higher order polynomials.

- **Exponential → approximated by polynomials** using Taylor series:
  
  \[
  e^x \approx \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \quad \text{for all } x
  \]

  \[
  a^x = e^x \ln a \approx 1 + x \ln a + \frac{(x \ln a)^2}{2!} + \frac{(x \ln a)^3}{3!} + \ldots
  \]

  - For our purposes, in practice expanding up to order 8 is typically enough.
Safety of the Root Approximation [ICLP’10, FOPARA’12]

Root approximation must be safe:
Intervals computed must ensure correct verification outcome.

Example

- If $\forall x \in S$, $AU(x) < SL(x) \rightarrow$ incorrect
- We define $F(x) = SL(x) - AU(x)$.
- If $\forall x \in S$, $F(x) > 0 \rightarrow$ incorrect.
Accumulated Cost (more details)
Inferring Accumulated Cost

[TPLP’16, FLOPS’16]

● Helping developers make (resource-related) design decisions:
  ▶ Which parts of the program are the most resource-consuming?
  ▶ Which predicates should be optimized first?

● The standard/classical notion of cost only partially meets these objectives:
  ▶ Predicates w/highest (standard) costs may not need to be optimized first.
  ▶ E.g., perhaps predicates with lower costs but which are called more often.
  ▶ The input sizes to such calls are also relevant.

● Need info resulting from a static profiling of the program to:
  ▶ identify the parts of a program responsible for highest fractions of the cost → accumulated cost.
  ▶ I.e., how the total resource usage of the execution of a program is distributed over selected parts of it (cost centers → predicates).

Static profiling → static inference of the kinds of information that are usually obtained at run-time by profilers.

Main contribution

Novel, general, and flexible framework for setting up cost equations/relations.
→ can be instantiated for performing a wide range of static resource usage analyses, including both accumulated cost and standard cost.
Accumulated-cost: Intuition

\begin{align*}
p(0). \\
p(X) & :- X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y). \\
q(0). \\
q(X) & :- X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y). \\
r(0). \\
r(X) & :- X > 0, \ Y \text{ is } X - 1, \ r(Y).
\end{align*}

We want to know how the standard/total cost of \( p \) is distributed between the predicates of the program.
Accumulated-cost: Intuition

We declare that predicates $p$, $q$, and $r$ are cost centers.

Cost centers are user-defined program points (predicates, in our case) to which execution costs are assigned during the execution of a program.
Accumulated-cost: Intuition

\[
p(0).
p(X) :- X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y).
q(0).
q(X) :- X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y).
r(0).
r(X) :- X > 0, \ Y \text{ is } X - 1, \ r(Y).
\]

Set of cost centers:
\[\diamondsuit = \{p, q, r\}\]

The cost of a call \(p(2)\) accumulated in cost center \(r\), denoted \(C^r_p(2)\)

Is the sum of the resolution steps that are descendant (in the call stack) of \(p(2)\), and whose closest ancestor in the call stack that is a cost center, is \(r\)
Accumulated-cost: Intuition

\[
p(0).
p(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y), p(Y).
\]

\[
q(0).
q(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y).
\]

\[
r(0).
r(X) :- X > 0, Y \text{ is } X - 1, r(Y).
\]

Set of cost centers:
\[\diamondsuit = \{p, q, r\}\]

The cost of a call \(p(2)\) accumulated in cost center \(r \rightarrow C_p^r(2) = 4\) is the sum of the resolution steps that are descendant (in the call stack) of \(p(2)\), and whose closest ancestor in the call stack that is a cost center, is \(r\).
Accumulated-cost: Intuition

The cost of a call $p(2)$ accumulated in cost center $q$, denoted $C_q^{p}(2)$, is the sum of the resolution steps that are descendant (in the call stack) of $p(2)$, and whose closest ancestor in the call stack that is a cost center, is $q$. 

\[
\begin{align*}
p(0) & , \\
p(X) & : = X > 0, \ Y \ \text{is} \ X - 1, \ r(Y), \ q(Y), \ p(Y) . \\
q(0) & , \\
q(X) & : = X > 0, \ Y \ \text{is} \ X - 1, \ r(Y), \ q(Y) . \\
r(0) & , \\
r(X) & : = X > 0, \ Y \ \text{is} \ X - 1, \ r(Y) . \\
\end{align*}
\]
Accumulated-cost: Intuition

\[
p(0).
p(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y), p(Y).
\]

\[
q(0).
q(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y).
\]

\[
r(0).
r(X) :- X > 0, Y \text{ is } X - 1, r(Y).
\]

Set of cost centers:
\[
\diamondsuit = \{p, q, r\}
\]

The cost of a call \(p(2)\) accumulated in cost center \(q\) \(\rightarrow\) \(C^q_p(2) = 3\)

Is the sum of the resolution steps that are descendant (in the call stack) of \(p(2)\), and whose closest ancestor in the call stack that is a cost center, is \(q\)
Accumulated-cost: Intuition

\[
p(0).
p(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y).
\]
\[
q(0).
q(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y).
\]
\[
r(0).
r(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y).
\]

Set of cost centers:
\[
\diamondsuit = \{p, q, r\}
\]

The cost of a call \( p(2) \) accumulated in cost center \( p \), denoted \( C_p^p(2) \)

Is the sum of the resolution steps that are descendant (in the call stack) of \( p(2) \), and whose closest ancestor in the call stack that is a cost center, is \( p \).
Accumulated-cost: Intuition

\textbf{Set of cost centers:} \\
\[ \diamond = \{ p, q, r \} \]

The cost of a call \( p(2) \) accumulated in cost center \( p \rightarrow C_p^p(2) = 3 \)

Is the \textbf{sum of the resolution steps} that are descendant (in the call stack) of \( p(2) \), and whose closest ancestor in the call stack that is a cost center, is \( p \)
Accumulated-cost: Intuition

\[ p(0). \]
\[ p(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y), p(Y). \]
\[ q(0). \]
\[ q(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y). \]
\[ r(0). \]
\[ r(X) :- X > 0, Y \text{ is } X - 1, r(Y). \]

Set of cost centers:
\[ \diamond = \{p, q, r\} \]

\[
C_p(2) = C_p^p(2) + C_p^q(2) + C_p^r(2) \\
10 = 3 + 3 + 4
\]
Accumulated-cost: Intuition

\[
p(0).
p(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y), p(Y).
\]

\[
q(0).
q(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y).
\]

\[
r(0).
r(X) :- X > 0, Y \text{ is } X - 1, r(Y).
\]

We declare that predicates \( p, q \), are cost centers, and \( r \) is not.

\[
\diamond = \{ p, q \}
\]
Accumulated-cost: Intuition

Set of cost centers:

\[ \diamond = \{ p, q \} \]

\[
\begin{align*}
C_p(2) &= C_p(2) + C_q(2) \\
10 &= 6 + 4
\end{align*}
\]
Accumulated-cost: Definition

Definition: Accumulated Cost

The cost of a (single) call \( p(n) \) accumulated in cost center \( q \), denoted \( C^q_p(n) \):

- Is the **sum of the costs** of all the computations that are descendants (in the call stack) of the call \( p(n) \), and are under the scope of any call to \( q \).
- We say that a computation is under the scope of a call to cost center \( q \), if the closest ancestor of such computation in the call stack that is a cost center, is \( q \).
- Expresses how much of the standard cost of the call to \( p \) is attributed to \( q \).
Cost Relations for Accumulated-costs in Cost Center $r$

Set of cost centers:  
\[ \diamond = \{p, q, r\} \]

Cost relations

The cost of $p$ accumulated in $r$:

\[
C^r_p(0) = 0 \\
C^r_p(n) = 0 + C^r_r(n-1) + C^r_q(n-1) + C^r_p(n-1) \quad \text{if } n > 0
\]

The cost of $q$ accumulated in $r$:

\[
C^r_q(0) = 0 \\
C^r_q(n) = 0 + C^r_r(n-1) \quad \text{if } n > 0
\]

The cost of $r$ accumulated in $r$:

\[
C^r_r(0) = 1 \\
C^r_r(n) = 1 + C^r_r(n-1) \quad \text{if } n > 0
\]
Cost Relations for Accumulated-costs in Cost Center \( r \)

Set of cost centers:
\[ \diamondsuit = \{ p, q, r \} \]

Cost relations

The cost of \( p \) accumulated in \( r \):
\[
\begin{align*}
C^r_p(0) & = 0 \\
C^r_p(n) & = 0 + C^r_r(n-1) + C^r_q(n-1) + C^r_p(n-1) \quad \text{if } n > 0
\end{align*}
\]

E.g. \( (n = 2) \):
\[
C^r_p(2) = 0 + C^r_r(1) + C^r_q(1) + C^r_p(1) = 0 + 2 + 1 + 1 = 4
\]
Cost Relations for Accumulated-costs in Cost Center $r$

Set of cost centers:

\[ \diamondsuit = \{p, q, r\} \]

\[
p(0).
p(X) :- X > 0, Y is X - 1, r(Y), q(Y), p(Y).
\]

\[
q(0).
q(X) :- X > 0, Y is X - 1, r(Y), q(Y).
\]

\[
r(0).
r(X) :- X > 0, Y is X - 1, r(Y).
\]

Cost relations

The cost of $p$ accumulated in $r$:

\[
C^{r}_{p}(0) = 0
\]

\[
C^{r}_{p}(n) = 0 + C^{r}_{r}(n - 1) + C^{r}_{q}(n - 1) + C^{r}_{p}(n - 1) \quad \text{if } n > 0
\]

The cost of $q$ accumulated in $r$:

\[
C^{r}_{q}(0) = 0
\]

\[
C^{r}_{q}(n) = 0 + C^{r}_{r}(n - 1) + C^{r}_{q}(n - 1) \quad \text{if } n > 0
\]

The cost of $r$ accumulated in $r$:

\[
C^{r}_{r}(0) = 1
\]

\[
C^{r}_{r}(n) = 1 + C^{r}_{r}(n - 1) \quad \text{if } n > 0
\]

\[ n = \text{size}(X) = X \text{ (actual value of } X) \]
Cost Relations for Accumulated-costs in Cost Center $r$

Set of cost centers:

$$\diamondsuit = \{p, q, r\}$$

Cost relations

The cost of $p$ accumulated in $r$:

$$C_r^p(0) = 0$$

$$C_r^p(n) = 0 + C_r^r(n - 1) + C_r^q(n - 1) + C_r^p(n - 1)$$  \hspace{1cm} \text{if } n > 0$$

The cost of $q$ accumulated in $r$:

$$C_r^q(0) = 0$$

$$C_r^q(n) = 0 + C_r^r(n - 1) + C_r^q(n - 1)$$  \hspace{1cm} \text{if } n > 0$$

The cost of $r$ accumulated in $r$  \rightarrow closed form:  

$$C_r^r(n) = n + 1$$  \hspace{1cm} \text{for } n \geq 0.$$
Cost Relations for Accumulated-costs in Cost Center \( r \)

\[
p(0).
p(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y), \ p(Y).
\]

\[
q(0).
q(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y), \ q(Y).
\]

\[
r(0).
r(X) \leftarrow X > 0, \ Y \text{ is } X - 1, \ r(Y).
\]

Set of cost centers:

\[ \diamondsuit = \{ p, q, r \} \]

Cost relations

The cost of \( p \) accumulated in \( r \):

\[
C_r^p(0) = 0
\]

\[
C_r^p(n) = 0 + C_r^r(n - 1) + C_q^r(n - 1) + C_p^r(n - 1) \quad \text{if } n > 0
\]

The cost of \( q \) accumulated in \( r \):

\[
C_q^r(0) = 0
\]

\[
C_q^r(n) = 0 + n + C_q^q(n - 1) \quad \text{if } n > 0
\]

The cost of \( r \) accumulated in \( r \rightarrow \) closed form: \( C_r^r(n) = n + 1 \), for \( n \geq 0 \).

\[
C_r^r(0) = 1
\]

\[
C_r^r(n) = 1 + C_r^r(n - 1) \quad \text{if } n > 0
\]
Cost Relations for Accumulated-costs in Cost Center $r$

Set of cost centers:

$$\Diamond = \{p, q, r\}$$

Cost relations

The cost of $p$ accumulated in $r$:

$$C^r_p(0) = 0$$
$$C^r_p(n) = 0 + C^r_r(n - 1) + C^r_q(n - 1) + C^r_p(n - 1)$$  if $n > 0$

The cost of $q$ accumulated in $r$:

$$C^r_q(0) = 0$$
$$C^r_q(n) = 0 + n + C^r_q(n - 1)$$  if $n > 0$

The cost of $r$ accumulated in $r$:

$$C^r_r(0) = 1$$
$$C^r_r(n) = 1 + C^r_r(n - 1)$$  if $n > 0$

$n = size(X) = X$ (actual value of $X$)
Cost Relations for Accumulated-costs in Cost Center $r$

\begin{align*}
p(0) & . \\
p(X) & : - X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y), \ p(Y) . \\
q(0) & . \\
q(X) & : - X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y) . \\
r(0) & . \\
r(X) & : - X > 0, \ Y \ is \ X - 1, \ r(Y) .
\end{align*}

Set of cost centers:
\[ \diamondsuit = \{ p, q, r \} \]

Cost relations

The cost of $p$ accumulated in $r$:
\[ C_{rp}(0) = 0 \]
\[ C_{rp}(n) = 0 + n + \frac{1}{2}(n - 1)^2 + \frac{1}{2}(n - 1) + C_{rp}(n - 1) \quad \text{if } n > 0 \]

The cost of $q$ accumulated in $r$:
\[ C_{rq}(0) = 0 \]
\[ C_{rq}(n) = 0 + n + C_{rq}(n - 1) \quad \text{if } n > 0 \]

The cost of $r$ accumulated in $r$:
\[ C_{rr}(0) = 1 \]
\[ C_{rr}(n) = 1 + C_{rr}(n - 1) \quad \text{if } n > 0 \]

$n = \text{size}(X) = X$ (actual value of $X$)
Cost Relations for Accumulated-costs in Cost Center $r$

Set of cost centers:
\[ \diamond = \{ p, q, r \} \]

Cost relations

The cost of $p$ accumulated in $r$:
\[
C_r^p(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n, \quad \text{for } n \geq 0.
\]

The cost of $q$ accumulated in $r$:
\[
C_r^q(n) = \frac{1}{2}n^2 + \frac{1}{2}n, \quad \text{for } n \geq 0.
\]

The cost of $r$ accumulated in $r$:
\[
C_r^r(n) = n + 1, \quad \text{for } n \geq 0.
\]
Cost Relations for Accumulated-costs in Cost Center $q$

Set of cost centers: $\diamondsuit = \{p, q, r\}$

Cost relations

The cost of $p$ accumulated in $q$:

\[
\begin{align*}
C^q_p(0) &= 0 \\
C^q_p(n) &= 0 + C^q_r(n-1) + C^q_q(n-1) + C^q_p(n-1) & \text{if } n > 0
\end{align*}
\]

The cost of $q$ accumulated in $q$:

\[
\begin{align*}
C^q_q(0) &= 1 \\
C^q_q(n) &= 1 + C^q_r(n-1) + C^q_q(n-1) & \text{if } n > 0
\end{align*}
\]

The cost of $r$ accumulated in $q$:

\[
\begin{align*}
C^q_r(0) &= 0 \\
C^q_r(n) &= 0 + C^q_r(n-1) & \text{if } n > 0
\end{align*}
\]
Cost Relations for Accumulated-costs in Cost Center $q$

Set of cost centers:
$\diamond = \{p, q, r\}$

Cost relations

The cost of $p$ accumulated in $q$:
\[
\begin{align*}
C^q_p(0) &= 0 \\
C^q_p(n) &= 0 + C^q_r(n - 1) + C^q_q(n - 1) + C^q_p(n - 1) \quad \text{if } n > 0
\end{align*}
\]

The cost of $q$ accumulated in $q$:
\[
\begin{align*}
C^q_q(0) &= 1 \\
C^q_q(n) &= 1 + C^q_r(n - 1) + C^q_q(n - 1) \quad \text{if } n > 0
\end{align*}
\]

The cost of $r$ accumulated in $q$ → $C^q_r(n) = 0$, for $n \geq 0$.

\[\forall r, q \in \diamond, \text{ if } r \not\rightarrow^* q \text{ then } C^q_r(\overline{x}) = 0 \quad \text{(Lemma 3)}\]
Cost Relations for Accumulated-costs in Cost Center \( q \)

Set of cost centers:
\[ \Diamond = \{ p, q, r \} \]

Cost relations

The cost of \( p \) accumulated in \( q \):
\[
C^q_p(0) = 0
\]
\[
C^q_p(n) = 0 + C^q_r(n-1) + C^q_q(n-1) + C^q_p(n-1) \quad \text{if } n > 0
\]

The cost of \( q \) accumulated in \( q \) → \( C^q_q(n) = n + 1 \), for \( n \geq 0 \).
\[
C^q_q(0) = 1
\]
\[
C^q_q(n) = 1 + C^q_r(n-1) + C^q_q(n-1) \quad \text{if } n > 0
\]

The cost of \( r \) accumulated in \( q \) → \( C^q_r(n) = 0 \), for \( n \geq 0 \).

\[ \forall r, q \in \Diamond, \text{ if } r \not\to^*_\alpha q \text{ then } C^q_r(\bar{x}) = 0 \quad \text{(Lemma 3)} \]
Cost Relations for Accumulated-costs in Cost Center $q$

Set of cost centers:
\[ \diamondsuit = \{ p, q, r \} \]

Cost relations

The cost of $p$ accumulated in $q$ → $C^q_p(n) = \frac{1}{2} n^2 + \frac{1}{2} n$.

The cost of $q$ accumulated in $q$ → $C^q_q(n) = n + 1$, for $n \geq 0$.

The cost of $r$ accumulated in $q$ → $C^q_r(n) = 0$, for $n \geq 0$.

\[ \forall r, q \in \diamondsuit, \text{ if } r \not\rightarrow^*_\alpha q \text{ then } C^q_r(\bar{x}) = 0 \] (Lemma 3)
Cost Relations for Accumulated-costs in Cost Center \( p \)

\[
\begin{align*}
p(0). \\
p(X) & : - X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y), \ p(Y). \\
q(0). \\
q(X) & : - X > 0, \ Y \ is \ X - 1, \ r(Y), \ q(Y). \\
r(0). \\
r(X) & : - X > 0, \ Y \ is \ X - 1, \ r(Y). 
\end{align*}
\]

Set of cost centers:

\[\lozenge = \{ p, q, r \}\]

Cost relations

The cost of \( p \) accumulated in \( p \):

\[
\begin{align*}
C_p^p(0) & = 1 \\
C_p^p(n) & = 1 + C_r^p(n - 1) + C_q^p(n - 1) + C_p^p(n - 1) \quad \text{if } n > 0 \\
C_q^p(n) & = 0 \quad \text{(by Lemma 3, since } q \not\leadsto^* p). \\
C_r^p(n) & = 0 \quad \text{(by Lemma 3, since } r \not\leadsto^* p). 
\end{align*}
\]

\[n = \text{size}(X) = X \quad \text{(actual value of } X)\]
Cost Relations for Accumulated-costs in Cost Center $p$

Set of cost centers:
\[ \diamondsuit = \{ p, q, r \} \]

Cost relations

The cost of $p$ accumulated in $p$:
\[
\begin{align*}
C_P^p(0) &= 1 \\
C_P^p(n) &= 1 + C_P^p(n-1) \quad \text{if } n > 0 \\
C_P^q(n) &= 0 \quad \text{(by Lemma 3, since } q \not\rightarrow^*_\alpha p) \\
C_P^r(n) &= 0 \quad \text{(by Lemma 3, since } r \not\rightarrow^*_\alpha p). 
\end{align*}
\]

\[ n = \text{size}(X) = X \quad \text{(actual value of } X) \]
Cost Relations for Accumulated-costs in Cost Center $p$

Cost relations

$n = \text{size}(X) = X$ (actual value of $X$)

The cost of $p$ accumulated in $p \rightarrow C_p(n) = n + 1$, for $n \geq 0$.

\[
C_p(0) = 1 \\
C_p(n) = 1 + C_p(n - 1) \text{ if } n > 0
\]

$C_q(n) = 0$ (by Lemma 3, since $q \not\alpha^* p$).

$C_r(n) = 0$ (by Lemma 3, since $r \not\alpha^* p$).
Need for Tracking the “Environment:” Example

Set of cost centers:
\[ \diamond = \{ p, q \} \]

The cost of \( p \) accumulated in \( q \):
\[
\begin{align*}
\mathcal{C}_q^p(0) & = 0 \\
\mathcal{C}_q^p(n) & = 0 + \mathcal{C}_{r,0}^q(n-1) + \mathcal{C}_{q}^q(n-1) + \mathcal{C}_p^p(n-1) \quad \text{if } n > 0
\end{align*}
\]

The cost of \( q \) accumulated in \( q \):
\[
\begin{align*}
\mathcal{C}_q^q(0) & = 1 \\
\mathcal{C}_q^q(n) & = 1 + \mathcal{C}_{r,1}^q(n-1) + \mathcal{C}_{q}^q(n-1) \quad \text{if } n > 0
\end{align*}
\]
Need for Tracking the “Environment:” Example

\[ p(0). \]
\[ p(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y), p(Y). \]
\[ q(0). \]
\[ q(X) :- X > 0, Y \text{ is } X - 1, r(Y), q(Y). \]
\[ r(0). \]
\[ r(X) :- X > 0, Y \text{ is } X - 1, r(Y). \]

Set of cost centers:
\[ \diamondsuit = \{ p, q \} \]

The cost of \( p \) accumulated in \( q \):
\[ C^q_p(0) = 0 \]
\[ C^q_p(n) = 0 + C^q_r,0(n - 1) + C^q_q(n - 1) + C^q_p(n - 1) \text{ if } n > 0 \]

The cost of \( q \) accumulated in \( q \):
\[ C^q_q(0) = 1 \]
\[ C^q_q(n) = 1 + C^q_r,1(n - 1) + C^q_q(n - 1) \text{ if } n > 0 \]

We have two versions for the cost of \( r \) accumulated in \( q \):

<table>
<thead>
<tr>
<th>Under the scope of ( q )</th>
<th>NOT under the scope of ( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ C^q_r,1(0) = 1 ]</td>
<td>[ C^q_r,0(0) = 0 ]</td>
</tr>
<tr>
<td>[ C^q_r,1(n) = 1 + C^q_r,1(n - 1) \text{ if } n &gt; 0 ]</td>
<td>[ C^q_r,0(n) = 0 + C^q_r,0(n - 1) \text{ if } n &gt; 0 ]</td>
</tr>
</tbody>
</table>
Our Extended Cost Relations for Accumulated-cost

The standard cost of a clause

\[ C \equiv p(\vec{x}) \oplus q_1(\vec{x}_1), \ldots, q_n(\vec{x}_n) \]

for a (single) call to \( p \):

\[ C_p(\vec{x}) = \varphi(p(\vec{x})) + \sum_{i=1}^{\lim(C,\vec{x})} \text{sols}_i \times C_{q_i}(\vec{x}_i) \]

E.g., for resolutions steps \( \rightarrow \varphi(p(\vec{x})) = 1 \).

- \( \lim(C,\vec{x}) \) def = index of the last body literal that is called in the execution of \( C \).
- \( \text{sols}_i \) def = product of the number of solutions produced by the ancestor literals of \( q_i(\vec{x}_i) \) in the clause body:

\[
\text{sols}_i = \prod_{j=1}^{i-1} s_{\text{pred}}(q_j(\vec{x}_j))
\]

- \( s_{\text{pred}}(q_j(\vec{x}_j)) \) def = number of solutions produced by \( q_j(\vec{x}_j) \)

The cost of a body literal \( q_i(\vec{x}_i) \) is obtained from the costs of all clauses applicable to it that are executed, by using an aggregation operator \( \odot \).
Our Extended Cost Relations for Accumulated-cost

The accumulated cost of a clause

\[ C \equiv p(\bar{x}) : - q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n) \]

for a (single) call to \( p \):

\[ C^c_{p, e}(\bar{x}) = B_\varphi(p, c, e) \times \varphi(p(\bar{x})) + \sum_{i=1}^{\text{lim}(C, \bar{x})} \text{sols}_i \times C^c_{q_i, e'}(\bar{x}_i) \times B(p, c, e, q_i) \]

E.g., for resolutions steps \( \rightarrow \varphi(p(\bar{x})) = 1 \).

- \( \text{lim}(C, \bar{x}) \text{ def } \) index of the last body literal that is called in the execution of \( C \).
- \( \text{sols}_i \text{ def } \) product of the number of solutions produced by the ancestor literals of \( q_i(\bar{x}_i) \) in the clause body:

\[ \text{sols}_i = \prod_{j=1}^{i-1} \text{s}_{\text{pred}}(q_j(\bar{x}_j)) \]

\[ \text{s}_{\text{pred}}(q_j(\bar{x}_j)) \text{ def } \) number of solutions produced by \( q_j(\bar{x}_j) \)

The cost of a body literal \( q_i(\bar{x}_i) \) is obtained from the costs of all clauses applicable to it that are executed, by using an aggregation operator \( \ominus \).
Our Extended Cost Relations for Accumulated-cost

The accumulated cost of a clause

\[ C \equiv p(\bar{x}) : - q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n) \]

for a (single) call to \( p \):

\[
C^c_{p,e} (\bar{x}) = B_\varphi(p, c, e) \times \varphi(p(\bar{x})) + \sum_{i=1}^{\lim(C, \bar{x})} \text{sols}_i \times C^c_{q_i, e'} (\bar{x}_i) \times B(p, c, e, q_i)
\]

- The environment \( e \) is a Boolean value (1 \( \equiv \) true and 0 \( \equiv \) false):

\[
e = \begin{cases} 
1 & \text{if the call to } p \text{ is under the scope of cost center } c \\
0 & \text{otherwise}
\end{cases}
\]

- Boolean functions:

\( B_\varphi(p, c, e) \) is 1 iff “the computation” is under the scope of \( c \).

\[
B_\varphi(p, c, e) \overset{\text{def}}{=} (p = c \lor (p \not\in \diamond \land e))
\]

\( B(p, c, e, q) \) is 1 iff the body literal is under the scope of \( c \), or it may call \( c \).

\[
B(p, c, e, q) \overset{\text{def}}{=} B_\varphi(p, c, e) \lor (q \leadsto^*_\alpha c)
\]

- \( e' = \mathcal{E}(p, c, e, q_i(\bar{x}_i)) \), and \( \mathcal{E} \) is the environment change function:

\[
\mathcal{E}(p, c, e, \_ ) \overset{\text{def}}{=} (p = c \lor (p \not\in \diamond \land e))
\]
Our Extended Cost Relations for Accumulated-cost

The accumulated cost of a clause

\[ C \equiv p(\bar{x}) : - q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n) \]

for a (single) call to \( p \):

\[ C^c_{p,e}(\bar{x}) = B_\varphi(p, c, e) \times \varphi(p(\bar{x})) + \sum_{i=1}^{\text{lim}(C, \bar{x})} \text{sols}_i \times C^c_{q_i,e'}(\bar{x}_i) \times B(p, c, e, q_i) \]

If a trust assertion gives the cost of \( p \) as a function \( \Psi(p)(\bar{x}) \), then:

\[ C_p(\bar{x}) = \Psi(p)(\bar{x}) \]
Our Extended Cost Relations for Accumulated-cost

The accumulated cost of a clause

\[ C \equiv p(\bar{x}) : - q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n) \]

for a (single) call to \( p \):

\[
C^c_{p,e}(\bar{x}) = B_\varphi(p, c, e) \times \varphi(p(\bar{x})) + \sum_{i=1}^{\text{lim}(C, \bar{x})} \text{sols}_i \times C^c_{q_i, e'}(\bar{x}_i) \times B(p, c, e, q_i)
\]

If a trust assertion gives the cost of \( p \) as a function \( \Psi(p)(\bar{x}) \), then:

\[
C^c_{p,e}(\bar{x}) = \Psi(p)(\bar{x}) \times B_\varphi(p, c, e)
\]
The cost a clause for a (single) call to $p$:

$$C_{c_p,e}(\bar{x}) = B_\varphi(p,c,e) \times \varphi(p(\bar{x})) + \sum_{i=1}^{\lim(C;\bar{x})} \text{sols}_i \times B(p,c,e,q_i) \times C_{c_{q_i},e'}(\bar{x}_i)$$

- A broad notion of environment $e$. E.g., for energy consumption:
  - state of the hardware or the whole system,
  - the last instruction executed (for modeling the switching cost), temperature, voltage, cache state, and pipeline state.
- Suitable definitions of the Boolean functions $B_\varphi(p,c,e)$ and $B(p,c,e,q)$ to control which terms of the cost relations should be considered.
- $C_{c_p,e}(\bar{x}) \overset{\text{def}}{=} \text{part of } C_p(\bar{x})$, performed in an environment $e$, that is attributed to cost center $c$ of the program.
Implementation

- Implementation within CiaoPP, directly as an abstract domain.
- The information abstracted at each program point includes the state + non-functional props.
- Cost relations are built incrementally, in the abstract domain.
- Features inherited for free:
  - Multivariance: separate equations built for each procedure version.
  - Equations are not built for unreachable parts of the program.
  - Easy combination with other abstract domains (reduced product based), in particular, the new sized types and a novel cardinality analysis.
  - Assertion verification.
  - Etc.
## Accumulated Cost: Experimental Results

<table>
<thead>
<tr>
<th>Cost-Centers &amp; Input Sizes</th>
<th>Accumulated Cost UB</th>
<th>Static vs. Dyn</th>
<th>Standard Cost UB</th>
<th>#Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>variance(n)</code>*</td>
<td>1</td>
<td>0%</td>
<td>$2n^2$</td>
<td>1</td>
</tr>
<tr>
<td><code>sq_diff(m_1, m_2)</code></td>
<td>$n - 1$</td>
<td>0%</td>
<td>$2m_1m_2 - 2m_2$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td><code>mean(u)</code></td>
<td>$2n^2 - n$</td>
<td>0%</td>
<td>$2u + 1$</td>
<td>$n$</td>
</tr>
<tr>
<td><code>is_prime(n)</code>*</td>
<td>1</td>
<td>0%</td>
<td>$(n - 1)! + n + 3$</td>
<td>$n$</td>
</tr>
<tr>
<td><code>fact(m)</code></td>
<td>$n$</td>
<td>0%</td>
<td>$m$</td>
<td>$n$</td>
</tr>
<tr>
<td><code>mult(u)</code></td>
<td>$(n - 1)! + 2$</td>
<td>0%</td>
<td>$u + 1$</td>
<td>$n$</td>
</tr>
<tr>
<td><code>app1(n_1, n_2, n_3)</code>*</td>
<td>$n_1$</td>
<td>0%</td>
<td>$O(n_1n_2n_3)^\dagger$</td>
<td>1</td>
</tr>
<tr>
<td><code>app2(m_1, m_2)</code></td>
<td>$n_1n_2$</td>
<td>0%</td>
<td>$m_1m_2$</td>
<td>$n_1$</td>
</tr>
<tr>
<td><code>app3(u)</code></td>
<td>$2n_1n_2n_3$</td>
<td>0%</td>
<td>$u$</td>
<td>$n_1n_2 + n_1$</td>
</tr>
<tr>
<td><code>dyade(n_1, n_2)</code>*</td>
<td>$n_1$</td>
<td>0%</td>
<td>$n_1(n_2 + 1)$</td>
<td>1</td>
</tr>
<tr>
<td><code>mult(m)</code></td>
<td>$n_1n_2$</td>
<td>0%</td>
<td>$m$</td>
<td>$n_1$</td>
</tr>
<tr>
<td><code>minsort(n)</code>*</td>
<td>$n + 1$</td>
<td>0%</td>
<td>$\frac{(n+1)^2}{2} + \frac{n+1}{2}$</td>
<td>1</td>
</tr>
<tr>
<td><code>findmin(m)</code></td>
<td>$\frac{(n+1)^2}{2} + \frac{n-1}{2}$</td>
<td>7%</td>
<td>$m$</td>
<td>$n + 1$</td>
</tr>
<tr>
<td><code>hanoi(n)</code>*</td>
<td>$2^n - 1$</td>
<td>0%</td>
<td>$2^{n+1} - 2$</td>
<td>1</td>
</tr>
<tr>
<td><code>move(m)</code></td>
<td>$2^n - 1$</td>
<td>0%</td>
<td>1</td>
<td>$2^n - 1$</td>
</tr>
<tr>
<td><code>coupled(n)</code>*</td>
<td>1</td>
<td>0%</td>
<td>$n + 2$</td>
<td>1</td>
</tr>
<tr>
<td><code>p(m)</code></td>
<td>$\frac{n^2}{2} + \frac{(-1)^n}{4} + \frac{3}{4}$</td>
<td>1.2%</td>
<td>$m + 1$</td>
<td>$n^2 - \frac{(-1)^n}{4} + \frac{1}{4}$</td>
</tr>
<tr>
<td><code>q(u)</code></td>
<td>$\frac{n^2}{2} - \frac{(-1)^n}{4} + \frac{1}{4}$</td>
<td>0%</td>
<td>$u + 1$</td>
<td>$n^2 + \frac{(-1)^n}{4} - \frac{1}{4}$</td>
</tr>
<tr>
<td><code>search(n)</code>*</td>
<td>1</td>
<td>0%</td>
<td>$2n + 2$</td>
<td>2</td>
</tr>
<tr>
<td><code>member(m)</code></td>
<td>$2n + 1$</td>
<td>0%</td>
<td>$2m + 1$</td>
<td>$2n + 1$</td>
</tr>
<tr>
<td><code>sublist(n_1, n_2)</code>*</td>
<td>$n_2 + 3$</td>
<td>5%</td>
<td>$n_1n_2 + 3n_2 + 2$</td>
<td>2</td>
</tr>
<tr>
<td><code>append(m)</code></td>
<td>$n_1n_2 + 2n_2 - 1$</td>
<td>40%</td>
<td>$2m - 1$</td>
<td>$n_1n_2 + 2n_2 - 1$</td>
</tr>
</tbody>
</table>
### Experimental Results: Times (milliseconds)

<table>
<thead>
<tr>
<th>Cost-Center</th>
<th>Accumulated Cost UB</th>
<th>Standard Cost UB</th>
<th>Acc / Std</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost Relations</td>
<td>Transformation (FLOPS’16)</td>
<td></td>
</tr>
<tr>
<td>variance*</td>
<td>3283 (-45%)</td>
<td>6038</td>
<td>3066</td>
</tr>
<tr>
<td>sq_diff</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isprime*</td>
<td>1245 (-42%)</td>
<td>2172</td>
<td>1231</td>
</tr>
<tr>
<td>fact</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>app1*</td>
<td>4150 (-34%)</td>
<td>6328</td>
<td>3757</td>
</tr>
<tr>
<td>app2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>app3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>minsort*</td>
<td>3400 (-29%)</td>
<td>4845</td>
<td>3300</td>
</tr>
<tr>
<td>findmin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dyade*</td>
<td>3097 (-24%)</td>
<td>4117</td>
<td>2853</td>
</tr>
<tr>
<td>mult</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hanoi*</td>
<td>1605 (-19%)</td>
<td>1996</td>
<td>1376</td>
</tr>
<tr>
<td>move</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coupled*</td>
<td>2407 (-14%)</td>
<td>3112</td>
<td>1877</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>search*</td>
<td>1079</td>
<td>N/A</td>
<td>1071</td>
</tr>
<tr>
<td>member</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sublist*</td>
<td>3674</td>
<td>N/A</td>
<td>3610</td>
</tr>
<tr>
<td>append</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>2652 (-33%)</td>
<td>4125</td>
<td>2542</td>
</tr>
</tbody>
</table>