Polyhedral Extraction Tool
(http://freecode.com/projects/libpet)

Sven Verdoolaege  Tobias Grosser

LIACS, Leiden
INRIA/ENS, Paris
sverdool@liacs.nl
tobias.grosser@inria.fr

January 23, 2012
Polyhedral Program Analysis and Transformation

```
for (i = 0; i <= N; ++i)
    a[i] = ...
for (i = 0; i <= N; ++i)
    b[i] = f(a[N-i])
```

```
for (i = 0; i <= N; ++i) {
    a[i] = ...
    b[N-i] = f(a[i])
}
```
Polyhedral Program Analysis and Transformation

\begin{align*}
\text{for } (i = 0; i \leq N; ++i) & \quad a[i] = ... \\
\text{for } (i = 0; i \leq N; ++i) & \quad b[i] = f(a[N-i])
\end{align*}
Basic Requirements

- Open source
- C99
  - iterator declarations
    ```
    for (int i = 0; i < N; ++i)
    ```
  - variable length arrays
    ⇒ parametric analysis
    ⇒ especially when arrays need to be linearized (e.g., CUDA)
- AST-level
  ⇒ source-to-source
Polyhedral Parsers

Cosy

LLVM/Polly

WRaP-IT gcc/graphite

IBM/XL

R-Stream

Atomium

insieme

clan

CHiLL

LooPo

pers

ROSE/PolyOpt

ROSE/PolyherdalModel

FADAlib

ROSE/Bee
Polyhedral Parsers

- Cosy
- LLVM/Polly
- gcc/graphite
- WRaP-IT
- CHiLL
- LooPo
- pers
- clan
- ROSE/PolyOpt
- ROSE/PolyherdalModel
- insieme
- FADAlib
- ROSE/Bee
- IBM/XL
- R-Stream
- Atomium
- Open source
Polyhedral Parsers

- C99
- Cosy
- LLVM/Polly
- gcc/graphite
- WRaP-IT
- insieme
- IBM/XL
- R-Stream
- Atomium
- ROSE/Bee
- ROSE/PolyherdalModel
- ROSE/PolyOpt
- CHiLL
- LooPo
- pers
- clan

Open source
Additional Requirements

- avoid arbitrary restrictions
- support features of both clan and pers

Before, we used

- clan
  - scops delimited by pragmas
  - used by PPCG: source-to-source compilers
target (currently): CUDA

- pers (SUIF)
  - scops autodetected
  - used by equivalence checker
    - CLooG outputs
    - data dependent constructs
    - array slices
  - used for derivation of polyhedral process networks
    - infinite time loop
Avoid Arbitrary Restrictions

Conditions and Index Expressions

Piecewise quasi-affine partial functions (\(\approx\) quasts) used to represent

- conditions \(\Rightarrow\) yes, no, undefined
- index expressions

(during construction)

May involve

- +, - (both unary and binary)
- * (at least one argument is piecewise constant)
- /, % (second argument is constant)
  
  \[ a / b \text{ is constructed as } a \geq 0 \ ? \ \text{floord}(a,b) : \text{ceild}(a,b) \]
- ?: 
- &&, ||,!
- <, <=, >, >=, ==, !=
Avoid Arbitrary Restrictions

Loops

```latex
\textbf{for} \ (i = \text{init}(n); \ \text{condition}(n,i); \ i \ += \ v)\\
```

- unique induction variable (may be declared)
- increment: \(i -= -v, i = i + v, ++i \) or \(--i\)
- any static piecewise quasi-affine condition  
  \(\implies\) needs to be satisfied for all iterations

Let

\[
D = \{i | \exists \alpha : \alpha \geq 0 \land i = \text{init}(n) + \alpha v\}\\
C = \{i | \text{condition}(n,i)\}
\]

Iteration domain (for \(v > 0\)):

\[
D \setminus (\{i' \to i | i' \leq i\}(D \setminus C)).
\]
Avoid Arbitrary Restrictions

Loops

\textbf{for } (i = \text{init}(n); \ condition(n,i); \ i += v) \n
\begin{itemize}
  \item unique induction variable (may be declared)
  \item increment: \ i -= -v, \ i = i + v, \ ++i \text{ or } --i
  \item \textbf{any} static piecewise quasi-affine condition \n      \Rightarrow \text{ needs to be satisfied for all iterations}
\end{itemize}

Let \n
\[ D = \{ i \mid \exists \alpha : \alpha \geq 0 \land i = \text{init}(n) + \alpha v \} \]

\[ C = \{ i \mid \text{condition}(n,i) \} \]

Iteration domain (for \( v > 0 \)): \n
\[ D \setminus (\{ i' \rightarrow i \mid i' \leq i \}(D \setminus C)) \].

Infinite loops

\begin{itemize}
  \item \textbf{for } (;;)
  \item \textbf{while } (1)
\end{itemize}
Context and Array Slices

Context describes assumptions on the parameters

Excludes

- values outside of parameter representation
- values that lead to negative array sizes
- values that necessarily lead to overflows

```c
int A[M][N];
f(A[4]);
⇒ access relation: [N, M] -> { S_0[] -> A[4, o1] }
```
Context and Array Slices

Context describes assumptions on the parameters

Excludes

- values outside of parameter representation
- values that lead to negative array sizes
- values that necessarily lead to overflows

Access to array row

```c
int A[M][N];
f(A[4]);

⇒ access relation: [N, M] -> { S_0[] -> A[4, o1] }
```
Parsing CLooG output

```c
for (c1=ceil(n,3);c1<=floor(2*n,3);c1++) {
    for (c2=0;c2<=n-1;c2++) {
        for (j=max(1,3*c1-n);j<=min(n,3*c1-n+4);j++) {
            p = max(ceil(3*c1-j,3),ceil(n-2,3));
            if (p <= min(floor(n,3),floor(3*c1-j+2,3))) {
                S2(c2+1,j,0,p,c1-p);
            }
        }
    }
}
```

- forward substitution
- special treatment of `floor` and `ceil`
- special treatment of `min` and `max`
Parsing CLooG output

\[
\begin{align*}
&\text{for } (c1=\text{ceild}(n,3); c1<=\text{floord}(2*n,3); c1++) \\
&\quad \text{for } (c2=0; c2<=n-1; c2++) \\
&\quad \quad \text{for } (j=\text{max}(1,3*c1-n); j<=\text{min}(n,3*c1-n+4); j++) \\
&\quad \quad \quad \text{p} = \text{max} (\text{ceild}(3*c1-j,3), \text{ceild}(n-2,3)); \\
&\quad \quad \quad \text{if } (\text{p} <= \text{min} (\text{floord}(n,3), \text{floord}(3*c1-j+2,3))) \\
&\quad \quad \quad \quad \text{S2}(c2+1,j,0,p,c1-p);
\end{align*}
\]

- forward substitution
- special treatment of \text{floord} and \text{ceild}
- special treatment of \text{min} and \text{max}
Parsing CLooG output

```c
for (c1=ceild(n,3);c1<=floord(2*n,3);c1++) {
    for (c2=0;c2<=n-1; c2++) {
        for (j=max(1,3*c1-n); j<=min(n,3*c1-n+4); j++) {
            p = max(ceild(3*c1-j,3),ceild(n-2,3));
            if (p <= min(floord(n,3),floord(3*c1-j+2,3))) {
                S2(c2+1,j,0,p,c1-p);
            }
        }
    }
}
```

- forward substitution
- special treatment of `floord` and `ceild`
- special treatment of `min` and `max`
Parsing CLooG output

for (c1=ceild(n,3);c1<=floord(2*n,3);c1++) {
    for (c2=0;c2<=n-1;c2++) {
        for (j=max(1,3*c1-n);j<=min(n,3*c1-n+4);j++) {
            p = max(ceild(3*c1-j,3),ceild(n-2,3));
            if (p <= min(floord(n,3),floord(3*c1-j+2,3))) {
                S2(c2+1,j,0,p,c1-p);
            }
        }
    }
}

- forward substitution
- special treatment of floord and ceild
- special treatment of min and max
Data Dependent Accesses and Conditions

Data dependent access

A[i + 1 + in2[i]]

- values of nested accesses are encoded in domain of access relation
- domain of outer access relation is itself a (wrapped) map
  - domain of wrapped map is the iteration domain
  - range of wrapped map are the values of the nested accesses

\{ [S_4[i] -> [i1]] -> A[1 + i + i1] \}

- list of nested access relation is maintained separately

\{ S_4[i] -> in2[i] \}
Data Dependent Accesses and Conditions

Data dependent access

\[ A[i + 1 + \text{in2}[i]] \]

- values of nested accesses are encoded in domain of access relation
- domain of outer access relation is itself a (wrapped) map
  - domain of wrapped map is the iteration domain
  - range of wrapped map are the values of the nested accesses

\{ [S_4[i] \rightarrow [i1]] \rightarrow A[1 + i + i1] \}

- list of nested access relation is maintained separately

\{ S_4[i] \rightarrow \text{in2}[i] \}

Data dependent conditions are handled similarly
⇒ statement domain is wrapped map
Equivalence Checking Example

for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2]=f(B[i-1+in2[i-2]]);
}

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at \([M - 8, M - 6]\) (when value of in2 in [-1,1])

Assumptions

- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions
Equivalence Checking Example

for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2]=f(B[i-1+in2[i-2]]);
}

Are the two programs on the left equivalent?
⇒ Same output when given same input
Yes, except at \([M - 8, M - 6]\) (when value of \(in2\) in \([-1,1]\))

Assumptions
- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:
- Parameters
- Recurrences
- Row accesses
- Data-dependent reads
Equivalence Checking Example

```c
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

Are the two programs on the left equivalent?

⇒ Same output when given same input
Yes, except at \([M - 8, M - 6]\) (when value of \(in2\) in \([-1,1]\))

Assumptions
- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:
- Parameters
- Recurrences
- Row accesses
- Data-dependent reads
Equivalence Checking Example

```c
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2]=f(B[i-1+in2[i-2]]);
}
```

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at $[M - 8, M - 6]$ (when value of in2 in $[-1,1]$)

Assumptions
- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:
- Parameters
- Recurrences
- Row accesses
- Data-dependent reads
Equivalence Checking Example

for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2]=f(B[i-1+in2[i-2]]);
}

Are the two programs on the left equivalent?
⇒ Same output when given same input
Yes, except at [M – 8, M – 6] (when value of in2 in [-1,1])

Assumptions
- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:
- Parameters
- Recurrences
- Row accesses
- Data-dependent reads
Equivalence Checking Example

```c
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2]=f(B[i-1+in2[i-2]]);
}
```

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at \([M - 8, M - 6]\)
(when value of \(\text{in2}\) in \([-1,1]\))

Assumptions
- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:
- Parameters
- Recurrences
- Row accesses
- Data-dependent reads
Support for unsigned integers

In C, unsigned integers undergo wrapping

- unsigned expressions are reduced modulo UINT_MAX + 1
  ⇒ clang tells us which expressions are unsigned + size
- use virtual iterator for loops with unsigned iterator
  ⇒ loop condition is composed with wrapping
  ⇒ schedule domain intersected with iteration domain
  ⇒ wrapping applied to domain and schedule
Support for unsigned integers

In C, unsigned integers undergo wrapping

- unsigned expressions are reduced modulo UINT_MAX + 1
  ⇒ clang tells us which expressions are unsigned + size
- use virtual iterator for loops with unsigned iterator
  ⇒ loop condition is composed with wrapping
  ⇒ schedule domain intersected with iteration domain
  ⇒ wrapping applied to domain and schedule

```c
for (unsigned char k=252; (k%9) <= 5; ++k)
   S:;
```

```c
domain: '{ S[k] : exists (e0 = [(507 - k)/256]:
        k >= 0 and k <= 255 and 256e0 >= 252 - k
        and 256e0 <= 261 - k) }'
schedule: '{ S[k] -> [0, o1] :
        exists (e0 = [(-k + o1)/256]:
        256e0 = -k + o1 and o1 >= 252 and
        k <= 255 and k >= 0 and o1 <= 261) }'
```
Integration into iscc

iscc: interactive environment
isl: manipulates parametric affine sets and relations
barvinok: counts elements in parametric affine sets and relations
CLooG: generates code to scan elements in parametric affine sets
Integration into iscc

iscc: interactive environment
isl: manipulates parametric affine sets and relations
barvinok: counts elements in parametric affine sets and relations
CLooG: generates code to scan elements in parametric affine sets
pet: extracts polyhedral model
Maximal Number of Live Memory elements

```plaintext
for (i = 0; i < N; ++i)
S1: t[i] = f(a[i]);
for (i = 0; i < N; ++i)
S2: b[i] = g(t[N-i-1]);
```

D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };  
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;  
W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;  
S := { S1[i] -> [0,i]; S2[i] -> [1,i] } * D;  
Dep := (last W before R under S)[0];  
LR := (lexmax (Dep . S)) . S^-1;  
LLT := S << S; LGE := S >>= S;  
After_Write := domain_map(LR) . LLT;  
Before_Read := range_map(LR) . LGE;  
N_Live := card ((After_Write * Before_Read)^-1);  
ub N_Live;
```

Result:

```
([N] -> { max(N) : N >= 2; max(N) : N = 1 }, True)
```
Maximal Number of Live Memory elements

\[
\text{for } (i = 0; i < N; ++i) \\
S1: \quad t[i] = f(a[i]); \\
\text{for } (i = 0; i < N; ++i) \\
S2: \quad b[i] = g(t[N-i-1]);
\]

\[
\begin{align*}
D &= [N] \rightarrow \{ \text{S1[i]} : 0 \leq i < N; \text{S2[i]} : 0 \leq i < N \}; \\
R &= [N] \rightarrow \{ \text{t[N-i-1]} \} \ast D; \\
M &= \text{parse_file("live.c");} \\
W &= D \ast M[0]; W := M[1]; R := M[2]; S := M[3] \ast D; \\
S &= \{ \text{S1[i]} \rightarrow [0,i]; \text{S2[i]} \rightarrow [1,i] \} \ast D; \\
\text{Dep} &= \text{(last W before R under S)}[0]; \\
\text{LR} &= \text{(lexmax (Dep . S)) . S}^{-1}; \\
\text{LLT} &= S \ll S; \text{LGE} &= S \gg S; \\
\text{After}_\text{Write} &= \text{domain_map(LR) . LLT;} \\
\text{Before}_\text{Read} &= \text{range_map(LR) . LGE;} \\
N_{\text{Live}} &= \text{card ((After}_\text{Write} \ast \text{Before}_\text{Read})^{-1}); \\
\text{ub} &= N_{\text{Live}};
\end{align*}
\]

Result:

\[
([N] \rightarrow \{ \max(N) : N \geq 2; \max(N) : N = 1 \}, \text{True})
\]