Dependencies between Analyses and Transformations in the Middle-End of a Compiler

François Irigoin & Fabien Coelho & Béatrice Creusillet

MINES ParisTech - Centre de Recherche en Informatique

3 April 2011
Analyze to compile? Compile to analyze?

Whom is this talk for?
- Compiler designers?
- Analyzer developers?
- Who is here today anyway?
What is abstract interpretation?

Winter 81-82, Metz, ante Feautrier 😊:
- A model for the environment (possibly)
What is abstract interpretation?

Winter 81-82, Metz, ante Feautrier 😊:

- A model for the environment (possibly)
- A model for the values and the state: *The abstract domain*
What is abstract interpretation?

Winter 81-82, Metz, ante Feautrier 😊:
- A model for the environment (possibly)
- A model for the values and the state: *The abstract domain*
- A model for the commands (possibly)
What is abstract interpretation?

Winter 81-82, Metz, ante Feautrier 😊:

- A model for the environment (possibly)
- A model for the values and the state: The abstract domain
- A model for the commands (possibly)
- Automatic and correct derivation of the model(s) from the program
What is abstract interpretation?

Winter 81-82, Metz, ante Feautrier 😊:

- A model for the environment (possibly)
- A model for the values and the state: *The abstract domain*
- A model for the commands (possibly)
- Automatic and correct derivation of the model(s) from the program
- Automatic solving of the model equations
What is abstract interpretation?

Winter 81-82, Metz, ante Feautrier 😊:

- A model for the environment (possibly)
- A model for the values and the state: *The abstract domain*
- A model for the commands (possibly)
- Automatic and correct derivation of the model(s) from the program
- Automatic solving of the model equations
- Decidability: over-approximations
What’s missing from a compiler viewpoint?

- Changing code after each transformation pass
What’s missing from a compiler viewpoint?

- Changing code after each transformation pass
- Multiple analyses
What’s missing from a compiler viewpoint?

- Changing code after each transformation pass
- Multiple analyses
- Dependant analyses
What’s missing from a compiler viewpoint?

- Changing code after each transformation pass
- Multiple analyses
- Dependant analyses
- Under-approximations or exactness for non-monotonous equations
What’s missing from a compiler viewpoint?

- Changing code after each transformation pass
- Multiple analyses
- Dependant analyses
- Under-approximations or exactness for non-monotonous equations
- ...
Running example

```c
int main()
{
    float a[100][100];
    int n = read_input(100, a);
    int p = init_parameter(n, a);
    compute(p, n, a);
    write_output(n, a);
}

void compute(int p, int n, float a[n][n])
{
    float t[n];
    int k = n;
    int i, j;
    for(i = 0; i <= n-1; i += 1) {
        k--;
        t[k] = a[i][i];
        for(j = 0; j <= n-1; j += 1)
            if (p>0)
                a[i][j] /= t[k];
    }
}
```
void compute(int p, int n, float a[n][n])
{
    float t[n];
    int k = n;
    int i, j;
    for(i = 0; i <= n-1; i += 1) {
        k--;
        t[k] = a[i][i];
        for(j = 0; j <= n-1; j += 1)
            if (p>0)
                a[i][j] /= t[k];
    }
}

void compute(int p, int n, float a[n][n])
{
    int i, j;
    //PIPS generated variable
    float __scalar__0;
    #pragma omp parallel for private(j,__scalar__0)
    for(i = 0; i <= 99; i += 1) {
        __scalar__0 = a[i][i];
        #pragma omp parallel for
        for(j = 0; j <= 99; j += 1)
            a[i][j] /= __scalar__0;
    }
}
Control simplification

```c
void compute(int p, int n, float a[n][n])
{
    float t[n];
    int k = n;
    int i, j;
    for(i = 0; i <= n-1; i += 1) {
        k--;
        t[k] = a[i][i];
        for(j = 0; j <= n-1; j += 1)
            if (p>0)
                a[i][j] /= t[k];
    }
}
```

- Condition $c$, $p>0$, is always true under precondition $P$
  \[
  \{ \sigma | P(\sigma) \land E(\bar{c}, \sigma) \} = \emptyset
  \]

- Similar use of $P$ for zero and one-trip loops, for infinite loops
Preconditions: where do they come from?

```c
// P() {n==100, 1<=p}
void compute(int p, int n, float a[n][n])
{
    float t[n];
    int k = n;
    int i, j;
    for(i = 0; i <= n-1; i += 1) {
        k--;
        t[k] = a[i][i];
        for(j = 0; j <= n-1; j += 1)
            // P(i,j,k) {i+k==99, n==100, 0<=i, i<=99, 0<=j, j<=99, 1<=p}
            if (p>0)
                a[i][j] /= t[k];
    }
}

// P(n) {n==100}
int main()
{
    float a[100][100];
    int n = read_input(100, a);
    // P(n) {n==100}
    int p = init_parameter(n, a);
    // P(n,p) {n==100, 1<=p}
    compute(p, n, a);
    write_output(n, a);
}

// T(init_parameter) {1<=init_parameter}
int init_parameter(int n, float a[n][n])
{
    int p;
    // T(p) {p<=0, p#init<=p, p#init<=0}
    while (p<=0)
        // T(p) {p==p#init+1}
        p++;
    // T(init_parameter) {init_parameter==p}
    return p;
}
```
Induction variable substitution

Variable $k$ can be substituted in statement $S$ with precondition $P$ within a loop of index $i$ if $P$ defines a mapping from $\sigma(i)$ to $\sigma(k)$:

$$v \rightarrow \{v' | \exists \sigma \in P \; \sigma(i) = v \land \sigma(k) = v' \}$$
An expression $e$ can be substituted under precondition $P$ if:

$$|\{v \in Val | \exists \sigma \in P \; v = \mathcal{E}(e, \sigma)\}| = 1$$
Iterative transformer and precondition equations

- For a sequence of statements $S_i$ at step $n$:
  \[
  T^n_{S_i} = T(S_i, P^{n-1}_{S_i}) \land P^{n-1}_{S_i}
  
  P^n_{S_i} = T^n_{S_{i-1}} \circ P^n_{S_{i-1}}
  
  P^0_{S_i} = Id
  \]

- Useful for some (rare) nested loops...
- No convergence guarantee
- Not used for running example 😊
After Allen&Kennedy Parallelization

```c
void compute(int p, int n, float a[n][n])
{
    float t[100];
    int k = 100;
    int i, j;
    for(i = 0; i <= 99; i += 1) {
        k = -i+99;
        t[-i+99] = a[i][i];
        for(j = 0; j <= 99; j += 1)
            a[i][j] /= t[-i+99];
    }
}
```

```c
void compute(int p, int n, float a[n][n])
{
    float t[100];
    int k = 100;
    int i, j;
    for(i = 0; i <= 99; i += 1)
        k = -i+99;
    #pragma omp parallel for
    for(i = 0; i <= 99; i += 1)
        t[-i+99] = a[i][i];
    #pragma omp parallel for
    for(i = 0; i <= 99; i += 1)
        #pragma omp parallel for
        for(j = 0; j <= 99; j += 1)
            a[i][j] /= t[-i+99];
}
```

Dependence system for two references in statements $S_1$ and $S_2$:

$\sigma_1(i) \prec \sigma_2(i) \land T_{S_1,S_2}(\sigma_1, \sigma_2) \land P_{S_1}(\sigma_1) \land P_{S_2}(\sigma_2) \land \ldots$
Coarse Grain Parallelization

- Assumes convex array regions $R$ and $W$:
  
  $$R, W \in Id \times \Sigma \rightarrow P(\Phi)$$

- Direct parallelization of a loop using convex array regions and Bernstein’s condition on body $B$ iterations:
  
  $$\forall \nu \in Id$$
  
  $$\{\phi | \forall \sigma, \sigma' \in P_B \phi \in R_{B,\nu}(\sigma) \land \phi \in W_{B,\nu}(\sigma') \land \sigma(i) < \sigma'(i)\} = \emptyset$$

- Beyond Bernstein’s conditions using $IN$ and $OUT$ array regions:
  
  $$\forall \nu \in Id$$
  
  $$\{\phi | \forall \sigma, \sigma' \in P_B \phi \in OUT_{B,\nu}(\sigma) \land \phi \in IN_{B,\nu}(\sigma') \land \sigma(i) < \sigma'(i)\} = \emptyset$$
Array privatization (different example)

```c
void compute(int p, int n, float a[n][n])
{
    int t[n];
    int k = n, i, s;
    for(i = 0; i <= n-1; i += 1) {
        int j;
        for(j = 0; j <= n-1; j += 1)
            t[j] = a[i][j]+j;
        for(j = 0; j <= n-1; j += 1)
            if (p>0)
                a[i][j] = t[j];
    }
}

void compute(int p, int n, float a[n][n])
{
    int t[100];
    int k = 100, i, s;
    #pragma omp parallel for private(t[100])
    for(i = 0; i <= 99; i += 1) {
        int j;
        #pragma omp parallel for
        for(j = 0; j <= 99; j += 1)
            t[j] = a[i][j]+j;
        #pragma omp parallel for
        for(j = 0; j <= 99; j += 1)
            a[i][j] = t[j];
    }
}
```

- An array \(k\) is privatizable in a loop \(l\) with body \(B\) if \(IN_{B,k} = OUT_{B,k} = \emptyset\)
- \(IN_{B,k}\) is the set of elements of \(k\) whose input values are used in \(B\)
  \[IN(S_1; S_2) = IN(S_1) \cup IN(S_2) \circ T(S_1) − W(S_1)\]
- \(OUT_{B,k}\) is the set of elements of \(k\) whose output values are used by the continuation of \(B\).
After scalarization (back to the running example)

```
void compute(int p, int n, float a[n][n])
{
    float t[100];
    int k = 100;
    int i, j;
    for(i = 0; i <= 99; i += 1) {
        k = -i+99;
        t[-i+99] = a[i][i];
        for(j = 0; j <= 99; j += 1)
            a[i][j] /= t[-i+99];
    }
}
```

```
void compute(int p, int n, float a[n][n])
{
    float t[100];
    int k = 100;
    int i, j;
    //PIPS generated variable
    float __scalar__0;
    #pragma omp parallel for private(k)
    for(i = 0; i <= 99; i += 1)
        k = -i+99;
    #pragma omp parallel for private(j, __scalar__0)
    for(i = 0; i <= 99; i += 1) {
        __scalar__0 = a[i][i];
        #pragma omp parallel for
        for(j = 0; j <= 99; j += 1)
            a[i][j] /= __scalar__0;
    }
}
```

- Let $B$ and $i$ be a loop body and index, and $W_{B,k}$ the $k$ region function
- Let $R : Val \rightarrow \mathcal{P}(\Phi)$ s.t. $R(v) = \{ \phi | \exists \sigma \sigma(i) = v \land \phi \in W_{B,k}(\sigma) \}$
- If $R$ is a mapping, $k$ can be replaced by a scalar.
void compute(int p, int n, float a[n][n])
{
    float t[100];
    int k = 100;
    int i, j;
    //PIPS generated variable
    float __scalar__0;
    #pragma omp parallel for private(k)
    for(i = 0; i <= 99; i += 1)
        k = -i+99;
    #pragma omp parallel for private(j,__scalar__0)
    for(i = 0; i <= 99; i += 1) {
        __scalar__0 = a[i][i];
        #pragma omp parallel for
        for(j = 0; j <= 99; j += 1)
            a[i][j] /= __scalar__0;
    }
}

A graph-based algorithm for a change!

Françöis Irigoin & Fabien Coelho & Béatrice Creusillet
ACCA, Chamonix, 3 April 2011
The whole picture, almost...
Sample of passes applied

SCALARIZATION updating CODE(compute)
SCALARIZATION made for compute.
Request: build phase/rule PRIVATIZE_MODULE for module compute.
  PROPER_EFFECTS building PROPER_EFFECTS(compute)
  CUMULATED_EFFECTS building CUMULATED_EFFECTS(compute)
  ATOMIC_CHAINS building CHAINS(compute)
  PRIVATIZE_MODULE updating CODE(compute)
PRIVATIZE_MODULE made for compute.
Selecting rule: PRINT_PARALLELIZEDOMP_CODE
Module compute selected
Request: build resource PARALLELPRINTED_FILE for module compute.
  PROPER_EFFECTS building PROPER_EFFECTS(compute)
  ATOMIC_CHAINS building CHAINS(compute)
  CUMULATED_EFFECTS building CUMULATED_EFFECTS(compute)
  SUMMARY_EFFECTS building SUMMARY_EFFECTS(compute)
  INITIAL_PRECONDITION building INITIAL_PRECONDITION(compute)
  PROPER_EFFECTS building PROPER_EFFECTS(main)
  CUMULATED_EFFECTS building CUMULATED_EFFECTS(main)
  SUMMARY_EFFECTS building SUMMARY_EFFECTS(main)
  INITIAL_PRECONDITION building INITIAL_PRECONDITION(main)
  PROGRAM_PRECONDITION building PROGRAM_PRECONDITION()
  TRANSFORMERS_INTER_FULL building TRANSFORMERS(compute)
  TRANSFORMERS_INTER_FULL building TRANSFORMERS(write_output)
  TRANSFORMERS_INTER_FULL building TRANSFORMERS(init_parameter)
  TRANSFORMERS_INTER_FULL building TRANSFORMERS(read_input)
  TRANSFORMERS_INTER_FULL building TRANSFORMERS(main)
  INTERPROCEDURAL_SUMMARY_PRECONDITION building SUMMARY_PRECONDITION(main)
  PRECONDITIONS_INTER_FULL building PRECONDITIONS(main)
  INTERPROCEDURAL_SUMMARY_PRECONDITION building SUMMARY_PRECONDITION(compute)
Approximate number of passes called

After database initialization: 0
After partial evaluation: 135
After parallelization: 158
After scalar privatization: 181
After array privatization: 255
After array scalarization: 328
After dead code elimination: 399
Difficulties hidden in a few analyses: 
\( T, P, W, R, IN, OUT \)

Program transformations as simple\(^1\) consequences of analyses? 
*mapping, function, empty set,*...

Yes, sometimes 
*Control simplification, contrant propagation, partial evaluation, induction variable substitution, privatization, scalarization, coarse grain loop parallelization,*...

But not always: graph algorithms are useful too 
*Dead code elimination,*...

Transformations used to simplify analyses 
*Compile to Analyze?*

---

\(^1\) Complexity?
Conclusion: genericity to simplify compilers?

- Analysis A depends on Analysis B, cycles are possible
- So abstract domain A depends on Abstract domain B
- A generic abstract domain interface such as APRON is not enough
- Simple domains are not useful with transformers
- Over-approximations are not sufficient: under-approximations or exact results are necessary for non-monotonous equations
- Do program transformations preserve abstract information? *Too much recomputing in PIPS 😊*
Questions?