From Data to Effects Dependence Graphs: Source-to-Source Transformations for C

Dixièmes rencontres de la communauté française de compilation

Nelson Lossing[1] Pierre Guillou[1] Mehdi Amini[2] François Irigoin[1]

[1]firstname.lastname@mines-paristech.fr

[2]mehdi@amini.fr

MINES ParisTech

MINES ParisTech, PSL Research University

Banyuls-sur-Mer, France, 23 Septembre 2015

Source-to-Source Compilers

- Fortran code
- C code

```
int main() {
  int i=10, j=1;
  int k = 2*(2*i+j);

  return k;
}
```

- Static analyses
- Instrumentation/ Dynamic analyses
- Transformations
- Source code generation
- Code modelling
- Prettyprint

- Fortran code
- C code

```
//PRECONDITIONS
int main() {
//  P() {}
    int i = 10, j = 1;

//  P(i,j) {i==10, j==1}
    int k = 2*(2*i+j);

//  P(i,j,k) {i==10,
j==1,  k==42}
    return k;
}
```

Loop Distribution on C99 Code

```
void example(unsigned int n)
{
    int a[n], b[n];
    for(int i=0; i<n; i++) {
        a[i] = i;
        typedef int mytype;
        mytype x;
        x = i;
        b[i] = x;
    }
    return;
}
```

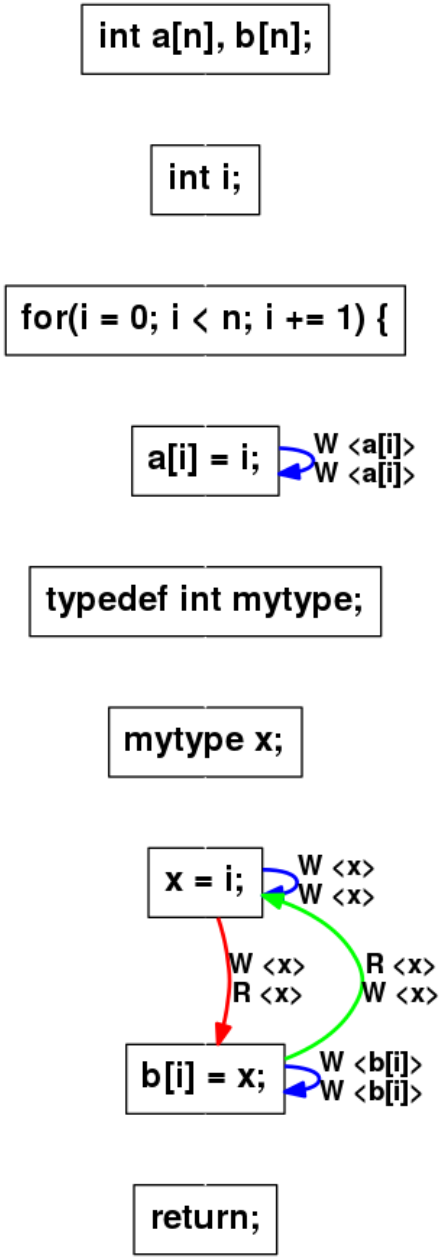
Loop Distribution on C99 Code

```
void example(unsigned int n)
{
    int a[n], b[n];
    for(int i=0; i<n; i++) {
        a[i] = i;
        typedef int mytype;
        mytype x;
        x = i;
        b[i] = x;
    }
    return;
}
```

```
void example(unsigned int n)
{
    int a[n], b[n];
    {
        int i;
        for(i = 0; i < n; i += 1) {
            a[i] = i;
        }
        for(i = 0; i < n; i += 1) {
            typedef int mytype;
        }
        for(i = 0; i < n; i += 1) {
            mytype x;
        }
        for(i = 0; i < n; i += 1) {
            x = i;
            b[i] = x;
        }
    }
    return;
}
```

Data Dependence Graph

```
void example(unsigned int n)
{
    int a[n], b[n];
    for(int i=0; i<n; i++) {
        a[i] = i;
        typedef int mytype;
        mytype x;
        x = i;
        b[i] = x;
    }
    return;
}
```



Outline

1. Limitations of the Data Dependence Graph
2. Effects Dependence Graph
3. Impact on Existing Code Transformations

Data Dependence Graph

- constraints on memory accesses for preventing incorrect reordering of operations/statements/loop iterations
- 3 types of constraints
  - flow dependence: read after write
  - anti-dependence: write after read
  - output dependence: write after write
- Limitations with C99

  declarations anywhere references after declaration
  user-defined types anywhere variable declaration after type declaration
  dependent types type write after variable write

Workarounds

Flatten Declarations

- Move every declarations at the function scope

Frame Pointer

- Use a low-level representation for the memory allocations

Flatten Declarations

Principle

- Move declarations at the function scope
- Perform α-*renaming* when necessary

Advantage

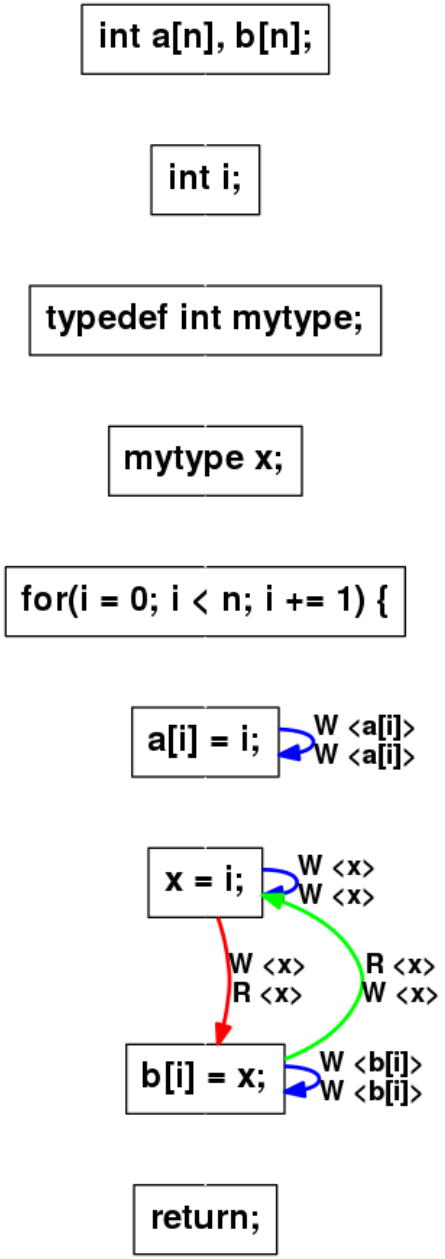
- Implementation is easy

Drawbacks

- Source code altered and less readable
- Possible stack overflow
- Not compatible with dependent types

Code Flattening

```
void example(unsigned int n)
{
    int a[n], b[n];
    int i;
    typedef int mytype;
    mytype x;
    for(i = 0; i < n; i += 1) {
        a[i] = i;
        x = i;
        b[i] = x;
    }
    return;
}
```



Code Flattening

```
void example(unsigned int n)
{
    int a[n], b[n];
    int i;
    typedef int mytype;
    mytype x;
    for(i = 0; i < n; i += 1) {
        a[i] = i;
        x = i;
        b[i] = x;
    }
    return;
}
```

```
void example(unsigned int n)
{
    int a[n], b[n];
    int i;
    typedef int mytype;
    mytype x;
    for(i = 0; i < n; i += 1)
        a[i] = i;
    for(i = 0; i < n; i += 1) {
        x = i;
        b[i] = x;
    }
    return;
}
```

Code Flattening & Dependent Type

```
void example(unsigned int n)
{
    int m;
    m = n+1;
    {
        int a[m], b[m];
        for(int i=0; i<m; i++) {
            a[i] = i;
            typedef int mytype;
            mytype x;
            x = i;
            b[i] = x;
        }
    }
    return;
}
```

```
void example(unsigned int n)
{
    int m;
    int a[m], b[m];
    int i;
    typedef int mytype;
    mytype x;
    m = n+1;
    for(i = 0; i < m; i += 1) {
        a[i] = i;
        x = i;
        b[i] = x;
    }
    return;
}
```

Explicit Memory Access Mechanism

Principle

- Type management:
  - Add a hidden variable ($\$type$) to represent the size in bytes of the type.
- Variable management:
  - Add a hidden variable (fp) that points to a memory location.
  - For each declaration, compute the address with fp.
  - Whenever a variable is referenced, pass by its address to analyze it.

Advantage

- Similar to compiler assembly code

Drawbacks

- New hidden variables added in IR $\rightarrow$ possible problem of coherency
- Overconstrained $\rightarrow$ declarations are serialized
- Hard to regenerate high-level source code

Explicit Access Mechanism, Implementation Idea

Initial Code:

```
void example(unsigned int n)
{

 int a[n], b[n];



 {
  int i;

  for(i=0;i<n;i+=1){
   a[i] = i;
   typedef int mytype;
   mytype x;

   x = i;
   b[i] = x;
  }
 }
 return;
}
```

Possible IR:

```
void example(unsigned int n)
{
 void* fp=...;
 a = fp;
 fp -= n*$int;
 b = fp;
 fp -= n*$int;
 {
  &i = fp;
  fp -= $int;
  for(*(&i)=0;*(&i)<n;*(&i)+=1) {
   a[*(&i)] = *(&i);
   $mytype = $int;
   &x = fp;
   fp -= $mytype;
   *(&x) = *(&i);
   b[*(&i)] = *(&x);
  } fp += $mytype;
 } fp += $int;
 return;
}
```

Background

$$\text{Identifier} \xrightarrow{\rho} \text{Location} \xrightarrow{\sigma} \text{Value}$$

- *Identifier*, *Location*, *Value* `int a = 0;`
- Environment ρ : *Identifier* $\rightarrow$ *Location*
- Memory State σ : *Location* $\rightarrow$ *Value*
- Statement S : $Env \times Mem \rightarrow Env \times Mem$
- Memory Effect E : $Statement \rightarrow (Env \times Mem \rightarrow \{Location\})$
  - Read Effect
  - Write Effect
  - used for building the Data Dependence Graph

Our Solution: New Kinds of Effects

Environment and Type Effects

- Environment
  - Read for each access of a variable
  - Write for each declaration of variable
- Type
  - Read for each use of a defined type
  - Write for each `typedef`, `struct`, `union` and `enum`

Our Solution: New Kinds of Effects

Environment and Type Effects

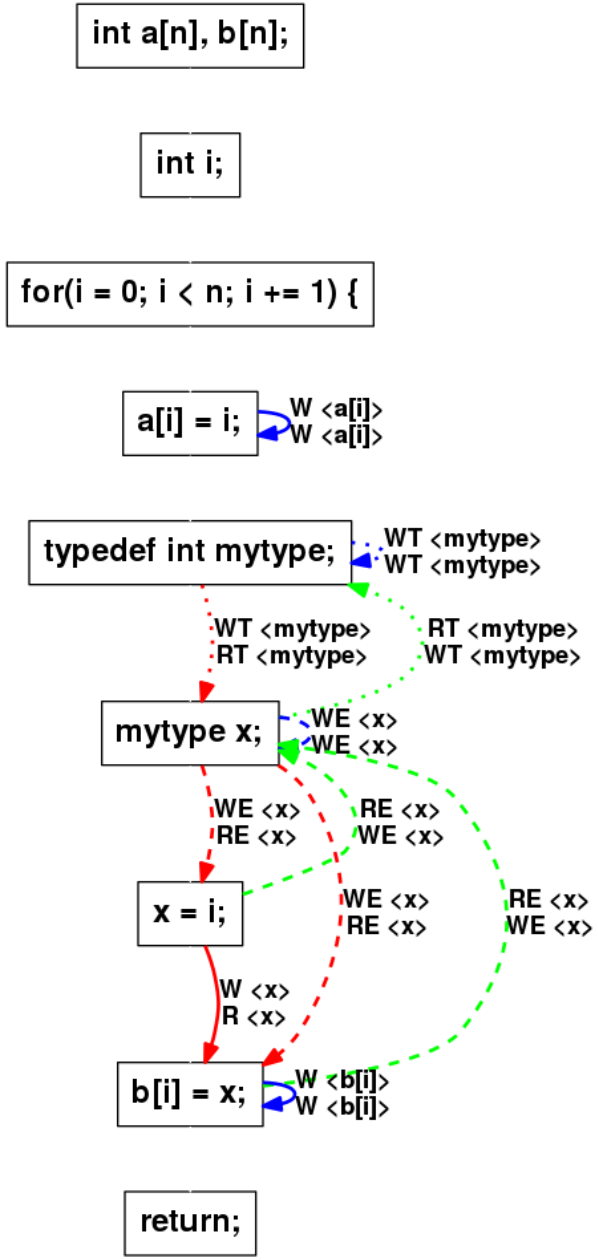
- Environment
  - Read for each access of a variable
  - Write for each declaration of variable
- Type
  - Read for each use of a defined type
  - Write for each `typedef`, `struct`, `union` and `enum`

Effects Dependence Graph (FXDG)

DDG + Environment & Type Effects

- No source code alteration
- More constraints to schedule statements properly
- Some code transformations need to be adapted

Loop Distribution With Extended Effects

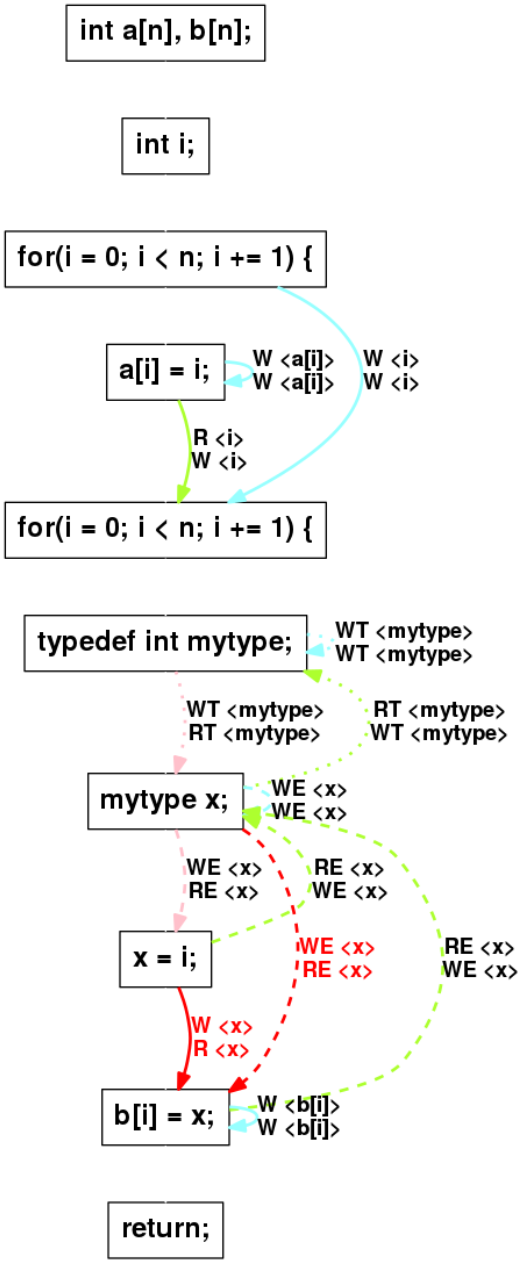



```
void example(unsigned int n)
{
    int a[n], b[n];
    {
        int i;
        for(i = 0; i < n; i += 1) {
            a[i] = i;
        }
        for(i = 0; i < n; i += 1) {
            typedef int mytype;
            mytype x;
            x = i;
            b[i] = x;
        }
    }
    return;
}
```

Impact of FXDG

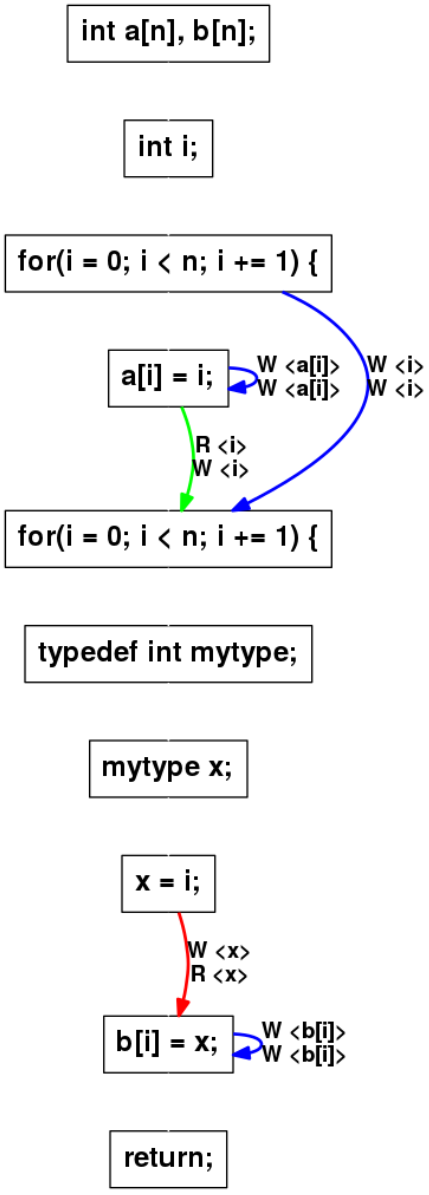
- Transformations benefitting from the FXDG
  - Allen & Kennedy
  - Loop Distribution
  - Dead Code Elimination
- Transformations hindered by the new effects
  - Forward Substitution
  - Scalarization
  - Isolate Statement
- Transformations needing further work
  - Flatten Code
  - Loop Unrolling
  - Loop-Invariant Code Motion
- Transformations not impacted
  - Strip Mining
  - Coarse Grain Parallelization

Forward Substitution with Extended Effects




```
void example(unsigned int n)
{
    int a[n], b[n];
    {
        int i;
        for(i = 0; i < n; i += 1) {
            a[i] = i;
        }
        for(i = 0; i < n; i += 1) {
            typedef int mytype;
            mytype x;
            x = i;
            b[i] = x;
        }
    }
    return;
}
```

Forward Substitution, Filtering the New Effects




```
void example(unsigned int n)
{
    int a[n], b[n];
    {
        int i;
        for(i = 0; i < n; i += 1) {
            a[i] = i;
        }
        for(i = 0; i < n; i += 1) {
            typedef int mytype;
            mytype x;
            x = i;
            b[i] = i;
        }
    }
    return;
}
```

Related Work

Other Source-to-Source Compilers

- **OSCAR** Fortran Code only
- **Cetus** C89 code only
- **Pluto** not compatible with declarations anywhere
- **Rose** C99 support through the EDG front-end

Low-level Source-to-Source Compilers

- **Polly** LLVM IR $\rightarrow$ LLVM IR

Conclusion

Standard data dependency is not enough

- no constraints on variable/type declarations
- C is too flexible

Effects Dependence Graph

- new Environment and Type Effects
- DDG extension

Impact on code transformations

- direct benefits: Loop Distribution, ...
- need to filter: Forward Substitution, ...
- affected in more complex ways.

$\Longrightarrow$ **different transformations need different Dependence Graphs**

From Data to Effects Dependence Graphs: Source-to-Source Transformations for C

Dixièmes rencontres de la communauté française de compilation

Nelson Lossing[1] Pierre Guillou[1] Mehdi Amini[2] François Irigoin[1]

[1]firstname.lastname@mines-paristech.fr

[2]mehdi@amini.fr

MINES ParisTech

MINES ParisTech, PSL Research University

Banyuls-sur-Mer, France, 23 Septembre 2015