Source-to-Source Compilers

- Fortran code
- C code

```c
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

```c
//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;
}
```
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

```c
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;
}
```
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 $\alpha$-renaming when necessary

**Advantage**
- Implementation is easy

**Drawbacks**
- Source code altered and less readable
- Possible stack overflow
- Not compatible with dependent types
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;
        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;
        x = i;
        for (i = 0; i < n; i += 1) {
            b[i] = x;
        }
    }
    return;
}
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:

```c
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:

```c
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);
            typedef int mytype;
            mytype x;
            &x = fp;
            fp -= $mytype;
            *(&x) = *(&i);
            b[*(&i)] = *(&x);
        } fp += $mytype;
    }
    fp += $int;
    return;
}
```
Background – Effects

\[ \text{Identifier} \xrightarrow{\rho} \text{Location} \xrightarrow{\sigma} \text{Value} \]

- **Identifier, Location, Value**
- **Environment, Env \( \rho: \text{Identifier} \rightarrow \text{Location} \)**
- **Memory State, MemState \( \sigma: \text{Location} \rightarrow \text{Value} \)**
- **Statement \( S: \text{Env} \times \text{MemState} \rightarrow \text{Env} \times \text{MemState} \)**
- **Memory Effect \( E: \)**
  - **Statement \( \xrightarrow{\text{Env} \times \text{MemState} \rightarrow \mathcal{P}(\text{Location})} \)**
  - Read Effect \( E_R \)
  - Write Effect \( E_W \)

```plaintext
int x = 0;
```
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
Motivation

Limitations of the Data Dependence Graph

Effects Dependence Graph

Impact on Existing Code Transformations

Conclusion

Loop Distribution With Extended Effects

```c
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

```c
void example(unsigned int n)
{
    int a[n], b[n];
    
    int i;
    for(i=0; i<n; i++) {
        a[i] = i;
    }
    for(i=0; i<n; i++) {
        typedef int mytype;
        mytype x;
        x = i;
        b[i] = x;
    }
    return;
}
```
Forward Substitution, Filtering the New Effects

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

⇒ different transformations need different Dependence Graphs
From Data to Effects Dependence Graphs: Source-to-Source Transformations for C

SCAM 2016

Nelson Lossing    Pierre Guillou    François Irigoin
firstname.lastname@mines-paristech.fr

MINES ParisTech
MINES ParisTech, PSL Research University

October 3, 2016 - Raleigh, NC, U.S.A
Environment and Type Effect Syntax in PIPS

+ action_kind = store:unit + environment:unit + type_declaration:unit ;
- action = read:unit + write:unit ;
+ action = read:action_kind + write:action_kind ;
syntax = reference + [...] ;
expression = syntax ;
entity = name:string x [...] ;
reference = variable:entity x indices:expression* ;
cell = reference + [...] ;
effect = cell x action x [...] ;
effects = effects:effect* ;
# Frame pointer: DDG

```c
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;
}
```
VLA Example

Initial Code

```c
void foo(int n) {
  int a[n];
  /* ... */
}
```

ASM Code

```asm
; int a[n];
  mov  -0x24(%rbp),%eax
  movslq %eax,%rdx
  sub   $0x1,%rdx
  mov   %rdx,-0x18(%rbp)
  movslq %eax,%rdx
  mov   %rdx,%r10
  mov   $0x0,%r11d
  movslq %eax,%rdx
  mov   %rdx,%r8
  mov   $0x0,%r9d
  cltq
  shl   $0x2,%rax
  lea   0x03(%rax),%rdx
  mov   $0x10,%eax
  sub   $0x1,%rax
  add   %rdx,%rax
  mov   $0x10,%esi
  mov   $0x0,%edx
  div   %rsi
  imul  $0x10,%rax,%rax
  sub   %rax,%rsp
  mov   %rsp,%rax
  add   $0x3,%rax
  shr   $0x2,%rax
  shl   $0x2,%rax
  mov   %rax,-0x10(%rbp)
```
VLA Example

LLVM Representation

; ModuleID = 'vla.c'

; Function Attrs: nounwind uwtable
define void @foo(i32 %n) #0 {
    %1 = alloca i32 , align 4
    %2 = alloca i8*
    store i32 %n , i32 * %1 , align 4
    %3 = load i32 * %1 , align 4
    %4 = zext i32 %3 to i64
    %5 = call i8* @llvm.stacksave()
    store i8* %5 , i8** %2
    %6 = alloca i32, i64 %4, align 16
    %7 = load i8** %2
    /* ... */
    call void @llvm.stackrestore(i8* %7)
    ret void
}

; Function Attrs: nounwind
declare i8* @llvm.stacksave() #1

; Function Attrs: nounwind
declare void @llvm.stackrestore(i8*) #1

LLVM to C Code

/* ... */
#if __GNUC__ < 4 /* Old GCC's, or compilers not GCC*/
#define __builtin_stack_save() 0
/* not implemented*/
#define __builtin_stack_restore(X) /* noop*/
#endif

void foo(unsigned int llvm_cbe_n) {
    unsigned int llvm_cbe_tmp__1;
    unsigned char *llvm_cbe_tmp__2;
    unsigned int llvm_cbe_tmp__3;
    unsigned char *llvm_cbe_tmp__4;
    unsigned int *llvm_cbe_tmp__5;
    unsigned char *llvm_cbe_tmp__6;

    (%(unsigned long long)(%(unsigned int)llvm_cbe_tmp__3)));
    llvm_cbe_tmp__6 = *(llvm_cbe_tmp__2);
    /* ... */
    _%%_builtin_stack_restore(llvm_cbe_tmp__6);
    return;
}