Linear Schemas for Program Dependence

ASTReNet Workshop 13
BCS

Sebastian Danicic

March 21, 2007

EPSRC
Engineering and Physical Sciences Research Council

EPSRC

Engineering and Physical Sciences Research Council

Goldsmiths UNIVERSITY OF LONDON

EPSRC
Engineering and Physical Sciences
Research Council

Goldsmiths UNIVERSITY OF LONDON

KING'S College LONDON Founded 1829

EPSRC
Engineering and Physical Sciences Research Council

EPSRC
Engineering and Physical Sciences Research Council

Goldsmiths University of London
King's College London Founded 1829
Brunel University West London
University of Essex

EPSRC
Engineering and Physical Sciences Research Council

Goldsmiths UNIVERSITY OF LONDON
KING'S College LONDON Founded 1829
Brunel UNIVERSITY WEST LONDON
University of Essex
uk PLC

Linear Schemas for Program Dependence EP/E002919/1

EPSRC

Engineering and Physical Sciences Research Council

Goldsmiths UNIVERSITY OF LONDON

KING'S College LONDON Founded 1829

Brunel UNIVERSITY WEST LONDON

University of Essex

uk PLC

AT&T

This Talk

1. People in the Schemas Project

This Talk

1. People in the Schemas Project
2. Program Dependence - Examples

This Talk

1. People in the Schemas Project
2. Program Dependence - Examples
3. Using Schemas gives more accurate notios of Dependence

This Talk

1. People in the Schemas Project
2. Program Dependence - Examples
3. Using Schemas gives more accurate notios of Dependence
4. Some Schema Theory

This Talk

1. People in the Schemas Project
2. Program Dependence - Examples
3. Using Schemas gives more accurate notios of Dependence
4. Some Schema Theory
5. Open Problems in Schema Theory

People

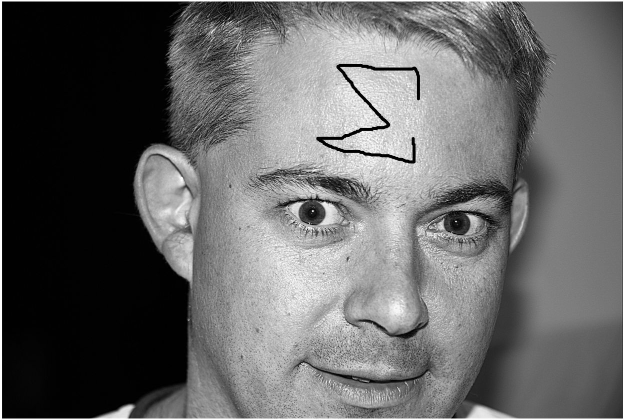
Mark Harman(C.I. –Kings)

Rob Hierons (C.I. –Brunel)

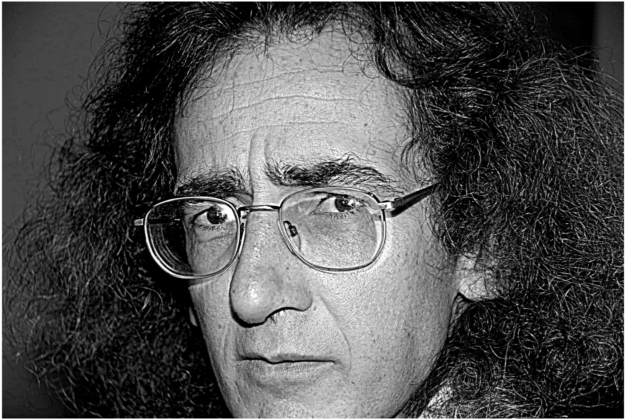
Chris Fox (Essex)

Lahcen Ouarbya (Goldsmiths)

Mike Laurence (R.A. –Goldsmiths)

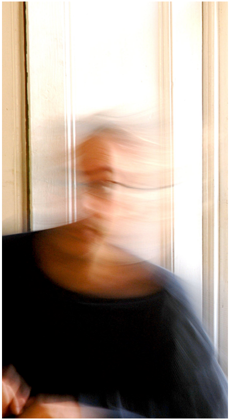


Dr. John Howroyd (Industrial Collaborator–@UK PLC)



Elaine Weyuker (Industrial Collaborator–AT&T Labs Research)

Sebastian Danicic (P.I.–Goldsmiths)



What is Program Dependence?

. . . it is static analysis of a program to find out which components affect which other components.

What is Program Dependence?

. . . it is static analysis of a program to find out which components affect which other components.

It enables us to answer questions like:

What is Program Dependence?

... it is static analysis of a program to find out which components affect which other components.

It enables us to answer questions like:

- Which bits of big program P affect the final value of variable x?

What is Program Dependence?

. . . it is static analysis of a program to find out which components affect which other components.

It enables us to answer questions like:

- Which bits of big program P affect the final value of variable x?
- Which bits of big program P affect the updating of this file?

What is Program Dependence?

. . . it is static analysis of a program to find out which components affect which other components.

It enables us to answer questions like:

- Which bits of big program P affect the final value of variable x?
- Which bits of big program P affect the updating of this file?
- What will be the impact of changing this bit of code here?

What is Program Dependence?

... it is static analysis of a program to find out which components affect which other components.

It enables us to answer questions like:

- Which bits of big program P affect the final value of variable x?
- Which bits of big program P affect the updating of this file?
- What will be the impact of changing this bit of code here?
- Which variables affect the value of this condition here?

What is Program Dependence?

. . . it is static analysis of a program to find out which components affect which other components.

It enables us to answer questions like:

- Which bits of big program P affect the final value of variable x?
- Which bits of big program P affect the updating of this file?
- What will be the impact of changing this bit of code here?
- Which variables affect the value of this condition here?
- Which bits of big program P affect the firing of this missile?

Applications of Dependence Analysis

- Program Comprehension

Applications of Dependence Analysis

- Program Comprehension
- Program Debugging

Applications of Dependence Analysis

- Program Comprehension
- Program Debugging
- Program Re-factoring

Applications of Dependence Analysis

- Program Comprehension
- Program Debugging
- Program Re-factoring
- Program Testing

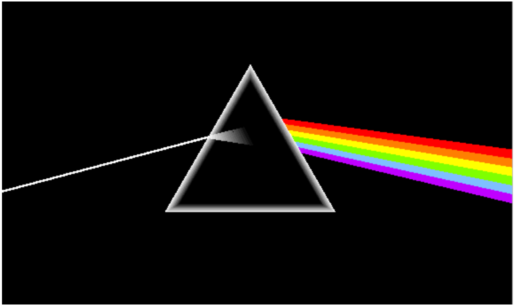
Applications of Dependence Analysis

- Program Comprehension
- Program Debugging
- Program Re-factoring
- Program Testing
- Program Security

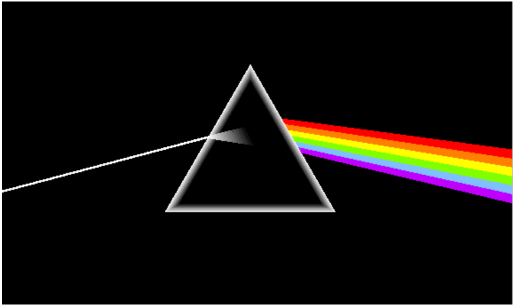
Applications of Dependence Analysis

- Program Comprehension
- Program Debugging
- Program Re-factoring
- Program Testing
- Program Security
- Program Slicing

Application of Dependence – Program Slicing

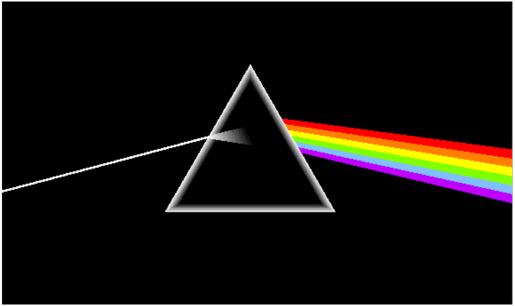


Application of Dependence – Program Slicing



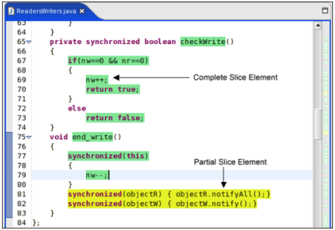
Program Slicing gives us different views of the same program

Application of Dependence – Program Slicing



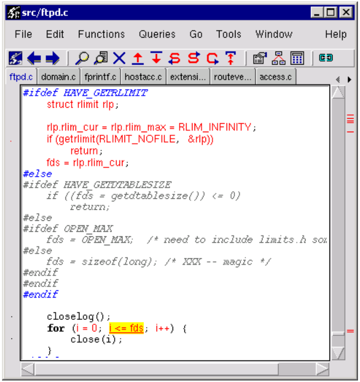
Program Slicing gives us different views of the same program
...depending what we are interested in.

Commercial Slicing Tools: Kaveri/Indus



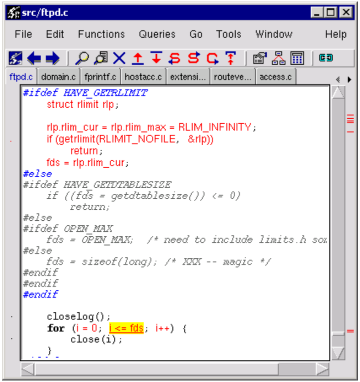
Kaveri is an eclipse plug-in front-end for the Indus Java slicer. It utilizes the Indus program slicer to calculate slices of Java programs and then displays the results visually in the editor. The purpose of this project is to create an effective tool for simplifying program understanding, program analysis, program debugging and testing.

Commercial Slicing Tools: Codesurfer



"The backward slice from a program point p includes all points that may influence whether control reaches p, and all points that may influence the values of the variables used at p when control gets there."

Commercial Slicing Tools: Codesurfer



"The backward slice from a program point p includes all points that may influence whether control reaches p, and all points that may influence the values of the variables used at p when control gets there."

What does may mean?

How Program Dependence is Calculated

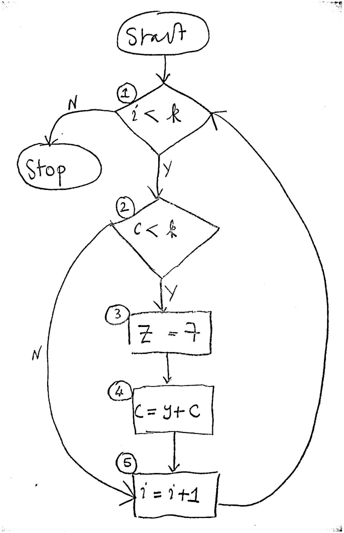
```
1   while (i<k)
    do
    begin
2      if (c<k)

       {
3         z=7;
4         c=y+c;
       }
5      i=i+1;
    end
```

How Program Dependence is Calculated

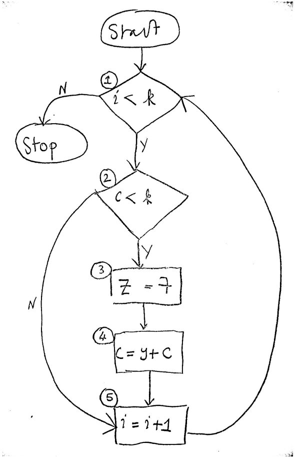
```
1    while (i<k)
     do
     begin
2       if (c<k)

        {
3          z=7;
4          c=y+c;
        }
5       i=i+1;
     end
```



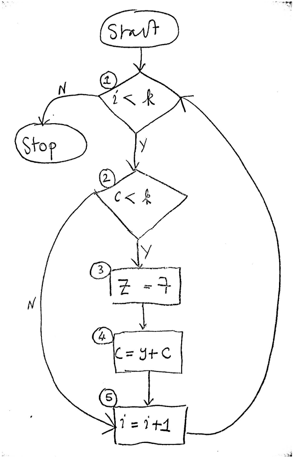
First convert the program into a Control Flow Graph

Data and Control Dependence



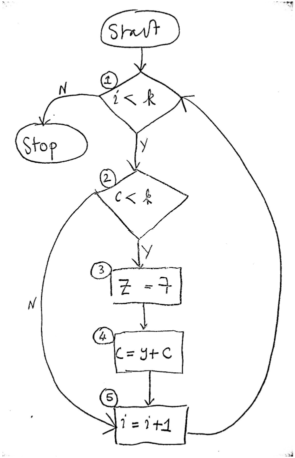
And then "chase back" the dependencies"

Data and Control Dependence



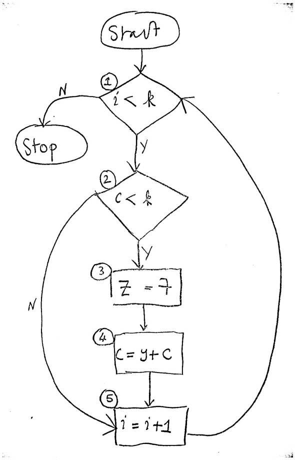
And then “chase back” the dependencies”

Final value of z is data dependent on (3)



And then "chase back" the dependencies"

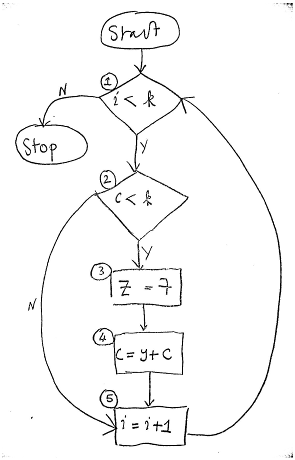
Final value of z is data dependent on (3)
(3) is control dependent on (2)



And then "chase back" the dependencies"

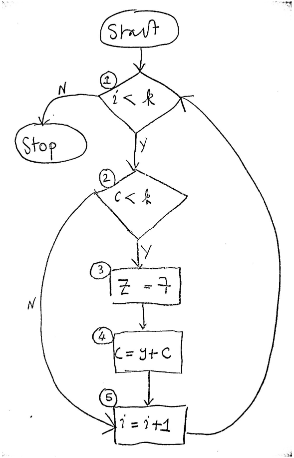
Final value of z is data dependent on (3)
(3) is control dependent on (2)
(2) is data dependent (loop carried) on (4)

Data and Control Dependence



And then "chase back" the dependencies"

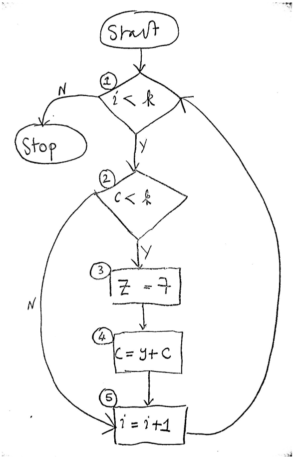
Final value of z is data dependent on (3)
(3) is control dependent on (2)
(2) is data dependent (loop carried) on (4)
(2) is control dependent on (1)



And then "chase back" the dependencies"

Final value of z is data dependent on (3)
(3) is control dependent on (2)
(2) is data dependent (loop carried) on (4)
(2) is control dependent on (1)
(1) is data dependent on (5)

Data and Control Dependence



And then "chase back" the dependencies"

Final value of z is data dependent on (3)
(3) is control dependent on (2)
(2) is data dependent (loop carried) on (4)
(2) is control dependent on (1)
(1) is data dependent on (5)

Slicing Algorithms compute the transitive closure of the union of the data and control dependence relations.

Slicing Example

Of course, we can compute the dependencies without the CFG.

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

Slicing Example

Which lines of this program affect the final value of z?

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

Conventional Program Slicers like Codesurfer will say “all of them!”

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

but why?

Slicing Example

```
while (i<k)
 {
    if (c<5)    <-----
    {
        z=7;    <-----
        c=y+c;
    }
    i=i+1;
 }
```

z=7 is *control–dependent* on (c<5)

Slicing Example

```
while (i<k)
 {
    if (c<5)   <-----
    {
        z=7;
        c=y+c; <-----
    }
    i=i+1;
 }
```

Because it's in a loop $c<5$ is data-dependent upon $c=y+c$;

Slicing Example

```
while (i<k)  <-----
 {
    if (c<5)   <-----
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

The if is control–dependent on the guard of the while

Slicing Example

```
while (i<k)  <-----
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1; <-----
 }
```

The guard of the while is data-dependent on $i=i+1$

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

So slicing on z gives the whole program

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

So slicing on z gives the whole program In fact, slicing algorithms compute the transitive closure of the dependence relation.

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

So slicing on z gives the whole program In fact, slicing algorithms compute the transitive closure of the dependence relation. But is this right?

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
        c=y+c;
    }
    i=i+1;
 }
```

So slicing on z gives the whole program In fact, slicing algorithms compute the transitive closure of the dependence relation. But is this right? Can you see a line that doesn't really affect the final value of z?

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
       c=y+c; <-----
    }
    i=i+1;
 }
```

But is z really dependent on this line?

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
       c=y+c; <-----
    }
    i=i+1;
 }
```

Clearly not because if we do execute c=y+c the value of z can't change any further, so it is irrelevant if we go through the true part of the if after that.

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;
       c=y+c; <-----
    }
    i=i+1;
 }
```

So transitive closure of dependence doesn't seem to be the most accurate way of computing dependencies.

Slicing Example

```
while (i<k)
 {
    if (c<5)
    {
        z=7;

    }
    i=i+1;
 }
```

This line should be removed from the slice.

But...

"Who cares!" I hear you say.

I do!

```
...millions of lines of code affecting y...
while (i<k)
{
    if (c<5)
    {
        z=7;
      c=y+c; <-----
    }
    i=i+1;
 }
```

What if there were millions of lines of code above this fragment that affected y? These would all, by transitivity, be unnecessarily included in the slice.

The Crux of the Problem

- Dependence is not transitive.

The Crux of the Problem

- Dependence is not transitive.
- The assumption that it is leads to many inaccuracies in dependency computation.

The Crux of the Problem

- Dependence is not transitive.
- The assumption that it is leads to many inaccuracies in dependency computation.
- So a statement that a slicing algorithm thinks may affect a variable often does not.

The Crux of the Problem

- Dependence is not transitive.
- The assumption that it is leads to many inaccuracies in dependency computation.
- So a statement that a slicing algorithm thinks may affect a variable often does not.
- This leads to slices that are too big.

The Crux of the Problem

- Dependence is not transitive.
- The assumption that it is leads to many inaccuracies in dependency computation.
- So a statement that a slicing algorithm thinks may affect a variable often does not.
- This leads to slices that are too big.
- Small is beautiful. – Big slices aren't very useful.

The Crux of the Problem

- Dependence is not transitive.
- The assumption that it is leads to many inaccuracies in dependency computation.
- So a statement that a slicing algorithm thinks may affect a variable often does not.
- This leads to slices that are too big.
- Small is beautiful. – Big slices aren't very useful.
- We want to find ways of producing more accurate dependence information and hence smaller slices.

Conventional Slicing May Remove Non–termination

```
while (true)
{

}
z=2;
```

What do we get if we slice on the final value of z?

Conventional Slicing May Remove Non–termination

```
while (true)
{

}
z=2;
```

The loop is removed since z=2 is not data or control dependent on it.

Conventional Slicing May Remove Non–termination

```
while (true)
{

}
z=2;
```

So transitive closure of dependence can introduce termination.

Conventional Slicing May Remove Non–termination

```
while (true)
{

}
z=2;
```

So, formally a program p and its slice s need only agree in initial states where p terminates.

Conventional Slicing May Remove Non–termination

```
while (true)
{

}
z=2;
```

So, formally a program *p* and its slice *s* need only agree in initial states where *p* terminates.
So, there's an even smaller slice of this program.

Conventional Slicing May Remove Non–termination

```
while (true)
{

}
z=2;
```

So, formally a program p and its slice s need only agree in initial states where p terminates.
So, there's an even smaller slice of this program.
The empty program – all statements can be removed.

A more subtle example

What is the slice on j at the end of the program? (Remembering that a program and its slice only need agree when the original terminates.)

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

A more subtle example

What is the slice on j at the end of the program? (Remembering that a program and its slice only need agree when the original terminates.)

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

Again, transitive closure of dependence gives the whole program. But can anyone see a line that can be removed?

A more subtle example

What is the slice on j at the end of the program? (Remembering that a program and its slice only need agree when the original terminates.)

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);    <-----
        z=h(z);
    }
 }
```

What about this line?

A more subtle example

What is the slice on j at the end of the program? (Remembering that a program and its slice only need agree when the original terminates.)

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);    <-----
        z=h(z);
    }
 }
```

It either causes the program to non–terminate or increases the number of iterations of the loop before termination.

A more subtle example

What is the slice on *j* at the end of the program? (Remembering that a program and its slice only need agree when the original terminates.)

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);    <-----
        z=h(z);
    }
 }
```

In initial states where the program terminates this line doesn't affect the final value of z.

A more subtle example

What is the slice on j at the end of the program? (Remembering that a program and its slice only need agree when the original terminates.)

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);    <-----
        z=h(z);
    }
 }
```

So, by definition, k=g(k) can be removed from the slice.

A more subtle example

Linear Schemas

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

The program above is in fact a Schema

A more subtle example

Linear Schemas

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

it is in fact a *linear* Schema because each function and predicate symbol occurs at most once.

A more subtle example

Linear Schemas

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=f(k);
        z=h(z);
    }
 }
```

Now it's not linear!

Linear Schemas

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

A schema is a program where all expressions have been replaced by symbolic expressions.

Linear Schemas

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

A schema represents a whole class of programs of similar structure.

Linear Schemas

```
while p(z)
 {
    if q(k) k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

For example the symbolic expression f(k), above, represents any expression involving just the variable k and no other variables. e.g. $k + 1$ or $2k * 5$

A Schema and a programs in its Equivalence Class

Schema

```
while p(z)
{
    if q(k)k=f(k);
    else
    {
        k=g(k);
        z=h(z);
    }
 }
```

A Schema and a programs in its Equivalence Class

Program

```
while(z<2)
{
    if (k<0)k=k+1;
    else
    {
        k=k-1;
        z=z+1;
    }
 }
```

A Schema and a programs in its Equivalence Class

Program

```
while(z<2)
{
    if (k<0)k=k+1;
    else
    {
        k=k-1;
        z=z+1;
    }
 }
```

Notice, in this case the slice speeds up termination

A Schema and a programs in its Equivalence Class

Another Program

```
while(z<2)
{
    if (k mod 2 == 0)k=k+2;
    else
    {
        k=k+1;
        z=z+1;
    }
 }
```

A Schema and a programs in its Equivalence Class

Another Program

```
while(z<2)
{
    if (k mod 2 == 0)k=k+2;
    else
    {
        k=k+1;
        z=z+1;
    }
 }
```

Notice, in this case the slice removes non-termination when k starts off odd.

Importantly...

```
while p(z)
{
    if q(k)k=f(k);
    else
    {
        k=g(k); <--------
        z=h(z);
    }
 }
```

using schema theory, we can prove that the final value of z is not dependent on this line for all programs in its equivalence class.

Using Schemas theory to Compute Dependence

- Schema theory allows us to compute dependence more accurately.

Program Dependence

- It is well known that *true program dependence* is not computable.

Program Dependence

- It is well known that *true program dependence* is not computable.
- In other words, we cannot decide in general whether line i of a program depends upon line j.

Program Dependence

- It is well known that *true program dependence* is not computable.
- In other words, we cannot decide in general whether line i of a program depends upon line j.
- This is equivalent to the halting problem.

Main Question: Are Dataflow Minimal Slices Computable?

but current dependence algorithms work at the ‘linear schema’ level of abstraction.

Main Question: Are Dataflow Minimal Slices Computable?

but current dependence algorithms work at the ‘linear schema’ level of abstraction.

- Can we compute minimal slices at the ‘linear schema’ level of abstraction?

Main Question: Are Dataflow Minimal Slices Computable?

but current dependence algorithms work at the 'linear schema' level of abstraction.

- Can we compute minimal slices at the 'linear schema' level of abstraction?
- In other words, can we compute true dependence at this level of abstraction?

Main Question: Are Dataflow Minimal Slices Computable?

but current dependence algorithms work at the ‘linear schema’ level of abstraction.

- Can we compute minimal slices at the ‘linear schema’ level of abstraction?
- In other words, can we compute true dependence at this level of abstraction?
- We are now asking whether line *i* of a program depends upon line *j*. for at least one program in its equivalence class.

Main Question: Are Dataflow Minimal Slices Computable?

but current dependence algorithms work at the 'linear schema' level of abstraction.

- Can we compute minimal slices at the 'linear schema' level of abstraction?
- In other words, can we compute true dependence at this level of abstraction?
- We are now asking whether line *i* of a program depends upon line *j*. for at least one program in its equivalence class.

These are problems in Schema Theory.

History of Schema Theory

- Schema theory was introduced in the 1950s by a Russian Mathematician, Ianov. It was seen as a way of proving the correctness of compiler optimisations.

History of Schema Theory

- Schema theory was introduced in the 1950s by a Russian Mathematician, Ianov. It was seen as a way of proving the correctness of compiler optimisations.
- Schemas are an abstract way of representing classes of programs with identical structure.

Seminal Work on Program Schemas

Some well-known computer scientists have worked on Schemas:

Seminal Work on Program Schemas

Some well-known computer scientists have worked on Schemas:

- Ianov (1960) "The Logical Schemes of Algorithms"
- M.S. Paterson (1968) "Program Schemata"
- D.C. Cooper(1969) "Program Scheme Equivalences and Logic"
- R.Milner(1970) "Equivalences on Program Schemes"
- Ershov (1971) "Theory of Program Schemata"
- Constable and Gries(1972) "On Classes of Program Schemata"
- Garland and Luckham(1973) "Program Schemes, Recursion Schemes and Formal Language"
- A.K.Chandra (1973) "On the Properties and Applications of Program Schemas"

The Semantics of Schemas

The Semantics of Schemas

$\text{States} = [\text{Variables} \to \text{Terms}] \cup \{\bot\}$

The Semantics of Schemas

States $=$ [Variables $\rightarrow$ Terms] $\cup\{\perp\}$
(Herbrand) Interpretations $=$ [Terms $\rightarrow$ {True,False}]

The Semantics of Schemas

States $=$ [Variables $\rightarrow$ Terms] $\cup\{\perp\}$
(Herbrand) Interpretations $=$ [Terms $\rightarrow$ {True,False}]
$\mathcal{M}$: Schemas $\rightarrow$ Interpretations $\rightarrow$ States $\rightarrow$ States.

The Semantics of Schemas

States $=$ [Variables $\rightarrow$ Terms] $\cup\{\perp\}$
(Herbrand) Interpretations $=$ [Terms $\rightarrow$ {True,False}]
$\mathcal{M}$: Schemas $\rightarrow$ Interpretations $\rightarrow$ States $\rightarrow$ States.

```
while p(z)
    {
       if q(k) k=f(k);
       else
        {
            k=g(k);
            z=h(z);
        }
    }
```

The Semantics of Schemas

States = [Variables → Terms] $\cup\{\perp\}$
(Herbrand) Interpretations = [Terms → {True,False}]
$\mathcal{M}$: Schemas → Interpretations → States → States.

```
while p(z)
    {
       if q(k) k=f(k);
       else
        {
            k=g(k);
            z=h(z);
        }
    }
```

John's Howroyd's Haskell Schema Interpreter readSch "boat.sch"

Decidability of Equivalence of Schemas

- Two Schemas are equivalent if the are semantically equivalent under all (Herbrand) interpretations.

Decidability of Equivalence of Schemas

- Two Schemas are equivalent if the are semantically equivalent under all (Herbrand) interpretations.
- The Decidability of Equivalence of Schemas implies the computability of minimal slices.

Decidability of Equivalence of Schemas

- Two Schemas are equivalent if the are semantically equivalent under all (Herbrand) interpretations.
- The Decidability of Equivalence of Schemas implies the computability of minimal slices.
- Why?

Decidability of Equivalence of Schemas

- Two Schemas are equivalent if the are semantically equivalent under all (Herbrand) interpretations.
- The Decidability of Equivalence of Schemas implies the computability of minimal slices.
- Why?
-
  1. First add *killing assignments* to all the uninteresting variables at the end of the program.
  2. Then try all combinations of deleting statements (not the killing assignments) and check for equivalence of the resulting schema with the original.

Linear Schemas

- We were surprised that no work had been done on Linear Schemas.

Linear Schemas

- We were surprised that no work had been done on Linear Schemas.
- Serendipitously, it turned out that the linearity condition helped in proving decidability of equivalence of schemas.

Decidability of Equivalence of Schemas

- Paterson (1967): Equivalence of General Schemas is Undecidable.

Decidability of Equivalence of Schemas

- Paterson (1967): Equivalence of General Schemas is Undecidable.
- For Linear Schemas decidability of equivalence is an open problem.

Decidability of Equivalence of Schemas

- Paterson (1967): Equivalence of General Schemas is Undecidable.
- For Linear Schemas decidability of equivalence is an open problem.
- S.Danicic et al: For certain classes of Linear Schemas equivalence is decidable.

Interesting Classes of Schema

A *Free Schema* is one in which for all paths through the schema, there is an interpretation which follows that path.

Free Schemas

Is this free?:

```
while p(j)
    {
       if q(k) k=f(k);
       else
        {
            k=g(k);
            j=h(j);
        }
    }
```

Free Schemas

Is this free?:

```
while p(j)
    {
       if q(k) k=f(k);
       else
        {
            k=g(k);
            j=h(j);
        }
    }
```

No - in a free schema, predicate terms never repeat.

Is this free?:

```
while p(j)
    {
        if q(k)
        {
            k=f(k);
            j=m(j)
        }
        else
        {
            k=g(k);
            j=h(j);
        }
    }
```

Free Schemas

Is this free?:

```
while p(j)
    {
        if q(k)
        {
            k=f(k);
            j=m(j)
        }
        else
        {
            k=g(k);
            j=h(j);
        }
    }
```

Yes

Conservative Schemas

For every assignment $x = E$, the symbolic expression E mentions x.

```
while p(j)
    {
         if q(k)
         {
            k=f(k);
            j=m(j)
         }
         else
         {
            k=g(k);
            j=h(j);
         }
    }
```

Liberal Schemas

At every assignment the same term is never computed.

```
while p(j)
    {
         if q(k)
         {
            k=f(k);
            j=m(j)
         }
         else
         {
            k=g(k);
            j=h(j);
         }
    }
```

convervative implies liberal

Recent Work

- We proved that equivalence of conservative, free, linear schemas is decidable.
  Michael R. Laurence, Sebastian Danicic, Mark Harman, Rob Hierons, and John Howroyd.
  Equivalence of conservative, free, linear program schemas is decidable.
  Theoretical Computer Science, 290:831–862, January 2003.

Recent Work

- We proved that equivalence of conservative, free, linear schemas is decidable.
  Michael R. Laurence, Sebastian Danicic, Mark Harman, Rob Hierons, and John Howroyd.
  Equivalence of conservative, free, linear program schemas is decidable.
  Theoretical Computer Science, 290:831–862, January 2003.

- ... and strengthened this by proving that equivalence of liberal, free linear schemas is decidable in polynomial time.
  Sebastian Danicic, Mark Harman, Robert Mark Hierons, John Howroyd, and Mike Laurence.
  Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time.
  Theoretical Computer Science, 2006.

  Equivalence of Predicate Linear, free, liberal,schemas is decidable
  The Journal of Logic and Algebraic Programming, 2007.

Recent Work

- We proved that equivalence of conservative, free, linear schemas is decidable.
  Michael R. Laurence, Sebastian Danicic, Mark Harman, Rob Hierons, and John Howroyd.
  Equivalence of conservative, free, linear program schemas is decidable.
  Theoretical Computer Science, 290:831–862, January 2003.

- ... and strengthened this by proving that equivalence of liberal, free linear schemas is decidable in polynomial time.
  Sebastian Danicic, Mark Harman, Robert Mark Hierons, John Howroyd, and Mike Laurence.
  Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time.
  Theoretical Computer Science, 2006.

  Equivalence of Predicate Linear, free, liberal,schemas is decidable
  The Journal of Logic and Algebraic Programming, 2007.

- ... and also found conditions when standard slicing algorithms give minimal slices.
  Sebastian Danicic, Chris Fox, Mark Harman, Robert Mark Hierons, John Howroyd, and Mike Laurence.
  Slicing algorithms are minimal for programs which can be expressed as linear, free, liberal schemas.
  The Computer Journal, 48(6):737–748, 2005.

Recent Work

- We proved that equivalence of conservative, free, linear schemas is decidable.
  Michael R. Laurence, Sebastian Danicic, Mark Harman, Rob Hierons, and John Howroyd.
  Equivalence of conservative, free, linear program schemas is decidable.
  Theoretical Computer Science, 290:831–862, January 2003.
- ... and strengthened this by proving that equivalence of liberal, free linear schemas is decidable in polynomial time.
  Sebastian Danicic, Mark Harman, Robert Mark Hierons, John Howroyd, and Mike Laurence.
  Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time.
  Theoretical Computer Science, 2006.

  Equivalence of Predicate Linear, free, liberal,schemas is decidable
  The Journal of Logic and Algebraic Programming, 2007.
- ... and also found conditions when standard slicing algorithms give minimal slices.
  Sebastian Danicic, Chris Fox, Mark Harman, Robert Mark Hierons, John Howroyd, and Mike Laurence.
  Slicing algorithms are minimal for programs which can be expressed as linear, free, liberal schemas.
  The Computer Journal, 48(6):737–748, 2005.
- This represented significant progress in the field of schema theory after a hiatus of about twenty years.

But we still don't know whether dataflow minimal slices are computable!

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable?

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable?

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable? Yes

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable? Yes
- Is equivalence of free linear schemas decidable?

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable? Yes
- Is equivalence of free linear schemas decidable? Don't know

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable? Yes
- Is equivalence of free linear schemas decidable? Don't know
- Is equivalence of linear schemas decidable?

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable? Yes
- Is equivalence of free linear schemas decidable? Don't know
- Is equivalence of linear schemas decidable? Don't know

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable? Yes
- Is equivalence of free linear schemas decidable? Don't know
- Is equivalence of linear schemas decidable? Don't know
- Is freeness of linear schemas decidable?

Open Problems in Decidability to Investigate

- Is equivalence of free conservative linear schemas decidable? Yes
- Is equivalence of free liberal linear schemas decidable? Yes
- Is equivalence of free linear schemas decidable? Don't know
- Is equivalence of linear schemas decidable? Don't know
- Is freeness of linear schemas decidable? Don't know

Aims of the Schemas Project

- There is strong evidence that the imposition of this extra but natural condition of linearity will lead to further decidability results in the theory of schemas.

Aims of the Schemas Project

- There is strong evidence that the imposition of this extra but natural condition of linearity will lead to further decidability results in the theory of schemas.
- We hope that our new results will lead to a re-appraisal of the substantial body of work in program schema theory and to further research on its applications in a modern framework.

Aims of the Schemas Project

- There is strong evidence that the imposition of this extra but natural condition of linearity will lead to further decidability results in the theory of schemas.
- We hope that our new results will lead to a re-appraisal of the substantial body of work in program schema theory and to further research on its applications in a modern framework. More accurate algorithms for computing dependence.

Thanks for listening!
Any Questions?

Schema Events

- Third Schema Meeting 30 March

Schema Events

- Third Schema Meeting 30 March
- http://sebastian.doc.gold.ac.uk/PTA/schemas/