Motivation

Both human- and computer-generated programs sometimes contain *data-flow anomalies*.

These anomalies result in the program being worse, in some sense, than it was intended to be.

Data-flow analysis is useful in locating, and sometimes correcting, these code anomalies.

Optimisation vs. debugging

Data-flow anomalies may manifest themselves in different ways: some may actually "break" the program (make it crash or exhibit undefined behaviour), others may just make the program "worse" (make it larger or slower than necessary).

Any compiler needs to be able to report when a program is broken (i.e. "compiler warnings"), so the identification of data-flow anomalies has applications in both optimisation and bug elimination.

Dead code

Dead code is a simple example of a data-flow anomaly, and LVA allows us to identify it.

Recall that code is *dead* when its result goes unused; if the variable `x` is not live on exit from an instruction which assigns some value to `x`, then the whole instruction is dead.

Dead code

```
⋮
a = x + 11;
b = y + 13;
c = a * b;      ← c DEAD
⋮
print z;
```

⋮
{ x, y, z }
{ a, y, z }
{ a, b, z }
{ z }
⋮
{ z }
{ }

Dead code

For this kind of anomaly, an automatic remedy is not only feasible but also straightforward: dead code with no live side effects is useless and may be removed.

Dead code

```
⋮                    ⋮
                 { x, y, z }
a = x + 11;
                 { a, y, z }
b = y + 13;
                 { a, b, z }
c = a * b;
                 { z }
⋮                    ⋮
                 { z }
print z;
                 { }
```

Dead code

```
⋮                         ⋮
                          { x, y, z }
a = x + 11;
                          { y, z }
b = y + 13;
                          { a, b, z }
c = a * b;
                          { z }
⋮                         ⋮
                          { z }
print z;
                          { }
```

Successive iterations may yield further improvements.

Dead code

The program resulting from this transformation will remain correct and will be both *smaller* and *faster* than before (cf. just smaller in *unreachable* code elimination), and no programmer intervention is required.

Uninitialised variables

In some languages, for example C and our 3-address intermediate code, it is syntactically legitimate for a program to read from a variable before it has definitely been initialised with a value.

If this situation occurs during execution, the effect of the read is usually *undefined* and depends upon unpredictable details of implementation and environment.

Uninitialised variables

This kind of behaviour is often undesirable, so we would like a compiler to be able to detect and warn of the situation.

Happily, the liveness information collected by LVA allows a compiler to see easily when a read from an undefined variable is possible.

Uninitialised variables

In a “healthy” program, variable liveness produced by later instructions is consumed by earlier ones; if an instruction demands the value of a variable (hence making it live), it is expected that an earlier instruction will define that variable (hence making it dead again).

Uninitialised variables

```
x = 11;
y = 13;
z = 17;
 ⋮
print x;
print y;
```

{ }

{ x }

{ x, y }

{ x, y }

⋮

{ x, y }

{ y }

{ }

Uninitialised variables

If any variables are still live at the beginning of a program, they represent uses which are potentially unmatched by corresponding definitions, and hence indicate a program with potentially undefined (and therefore incorrect) behaviour.

Uninitialised variables

```
x = 11;
y = 13;
 ⋮
print x;
print y;
print z;
```

{ z }   z LIVE

{ x, z }

{ x, y, z }

⋮

{ x, y, z }

{ y, z }

{ z }

{ }

Uninitialised variables

In this situation, the compiler can issue a warning: “variable `z` may be used before it is initialised”.

However, because LVA computes a safe (syntactic) overapproximation of variable liveness, some of these compiler warnings may be (semantically) spurious.

Uninitialised variables

```
if (p) {
  x = 42;
}
 ⋮
if (p) {
  print x;
}
```

{ x } x LIVE
{ }
{ x }
{ x }
⋮
{ x }
{ x }
{ }
{ }

Uninitialised variables

Here the analysis is being *too* safe, and the warning is unnecessary, but this imprecision is the nature of our computable approximation to semantic liveness.

So the compiler must either risk giving unnecessary warnings about correct code ("false positives") or failing to give warnings about incorrect code ("false negatives"). Which is worse?

Opinions differ.

Uninitialised variables

Although dead code may easily be remedied by the compiler, it's not generally possible to automatically fix the problem of uninitialised variables.

As just demonstrated, even the decision as to whether a warning indicates a genuine problem must often be made by the programmer, who must also fix any such problems by hand.

Uninitialised variables

Note that higher-level languages have the concept of (possibly nested) *scope*, and our expectations for variable initialisation in "healthy" programs can be extended to these.

In general we expect the set of live variables at the beginning of any scope to *not contain* any of the variables local to that scope.

Uninitialised variables

```
int x = 5;
int y = 7;
if (p) {
   int z;
   ⋮
   print z;
}
print x+y;
```

{ x, y, z } z LIVE

Write-write anomalies

While LVA is useful in these cases, some similar data-flow anomalies can only be spotted with a different analysis.

Write-write anomalies are an example of this. They occur when a variable may be written twice with no intervening read; the first write may then be considered unnecessary in some sense.

```
x = 11;
x = 13;
print x;
```

Write-write anomalies

A simple data-flow analysis can be used to track which variables may have been written but not yet read at each node.

In a sense, this involves doing LVA in reverse (i.e. forwards!): at each node we should remove all variables which are referenced, then add all variables which are defined.

Write-write anomalies

$$in\text{-}wnr(n) = \bigcup_{p \in pred(n)} out\text{-}wnr(p)$$

$$out\text{-}wnr(n) = \Big(in\text{-}wnr(n) \setminus ref(n)\Big) \cup def(n)$$

$$wnr(n) = \bigcup_{p \in pred(n)} \Big((wnr(p) \setminus ref(p)) \cup def(p) \Big)$$

Write-write anomalies

y is also dead here.

y is rewritten here without ever having been read.

```
        { }
x = 11;
        { x }
y = 13;
        { x, y }
z = 17;
        { x, y, z }
 ⋮       ⋮
        { x, y, z }
print x;
        { y, z }
y = 19;
        { y, z }
 ⋮       ⋮
```

Write-write anomalies

But, although the second write to a variable *may* turn an earlier write into dead code, the presence of a write-write anomaly doesn't *necessarily* mean that a variable is dead — hence the need for a different analysis.

Write-write anomalies

```
x = 11;
if (p) {
    x = 13;
}
print x;
```

`x` is live throughout this code, but if `p` is true during execution, `x` will be written twice before it is read. In most cases, the programmer can remedy this.

Write-write anomalies

```
if (p) {
  x = 13;
} else {
  x = 11;
}
print x;
```

This code does the same job, but avoids writing to `x` twice in succession on any control-flow path.

Write-write anomalies

```
if (p) {
  x = 13;
}
if (!p) {
  x = 11;
}
print x;
```

Again, the analysis may be too approximate to notice that a particular write-write anomaly may never occur during any execution, so warnings may be inaccurate.

Write-write anomalies

As with uninitialised variable anomalies, the programmer must be relied upon to investigate the compiler's warnings and fix any genuine problems which they indicate.

Clash graphs

The ability to detect data-flow anomalies is a nice compiler feature, but LVA's main utility is in deriving a data structure known as a *clash graph* (aka *interference graph*).

Clash graphs

When generating intermediate code it is convenient to simply invent as many variables as necessary to hold the results of computations; the extreme of this is "normal form", in which a new temporary variable is used on each occasion that one is required, with none being reused.

Clash graphs

```
x = (a*b) + c;
y = (a*b) + d;
```

lex, parse, translate

```
MUL t1,a,b
ADD x,t1,c
MUL t2,a,b
ADD y,t2,d
```

Clash graphs

This makes generating 3-address code as straightforward as possible, and assumes an imaginary target machine with an unlimited supply of "virtual registers", one to hold each variable (and temporary) in the program.

Such a naïve strategy is obviously wasteful, however, and won't generate good code for a real target machine.

Clash graphs

Before we can work on improving the situation, we must collect information about which variables actually *need* to be allocated to different registers on the target machine, as opposed to having been *incidentally* placed in different registers by our translation to normal form.

LVA is useful here because it can tell us which variables are *simultaneously live*, and hence *must* be kept in separate virtual registers for later retrieval.

Clash graphs

```
x = 11;
y = 13;
z = (x+y) * 2;
a = 17;
b = 19;
z = z + (a*b);
```

Clash graphs

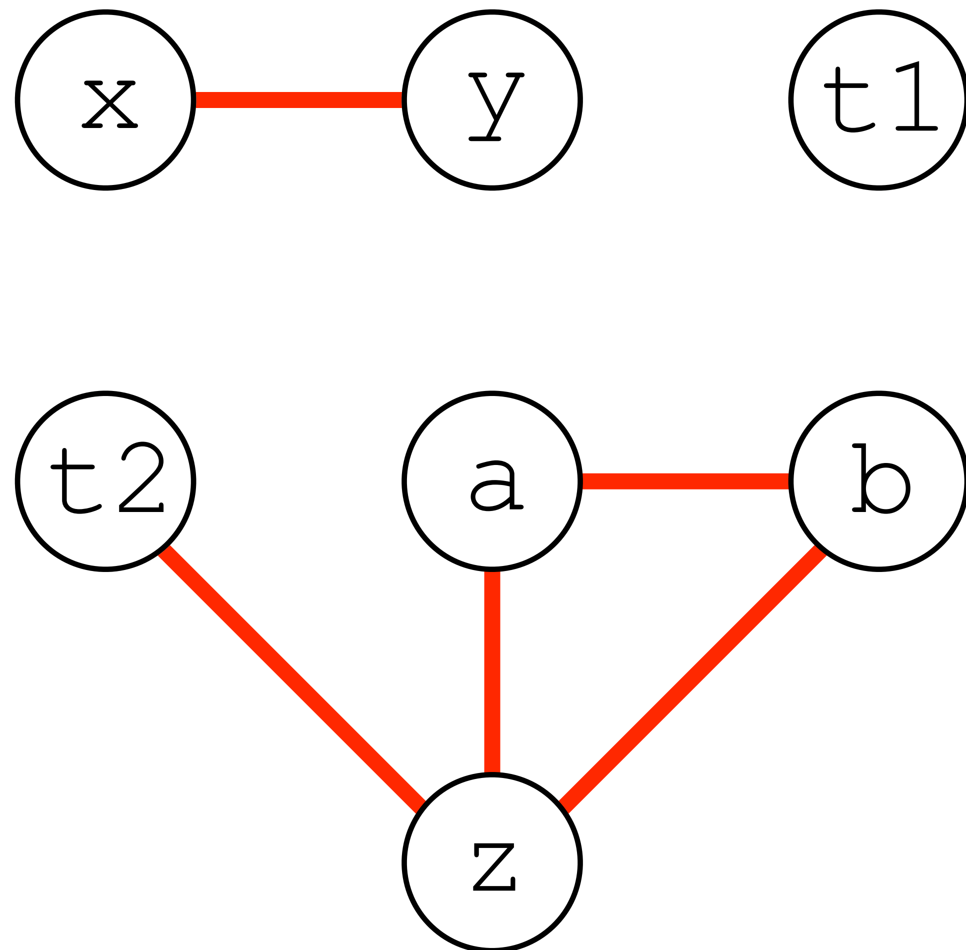
```
MOV x,#11        { }
MOV y,#13        { x }
ADD t1,x,y       { x, y }
MUL z,t1,#2      { t1 }
MOV a,#17        { z }
MOV b,#19        { a, z }
MUL t2,a,b       { a, b, z }
ADD z,z,t2       { t2, z }
```

Clash graphs

In a program's clash graph there is one node for each virtual register and an edge between nodes when their two registers are ever simultaneously live.

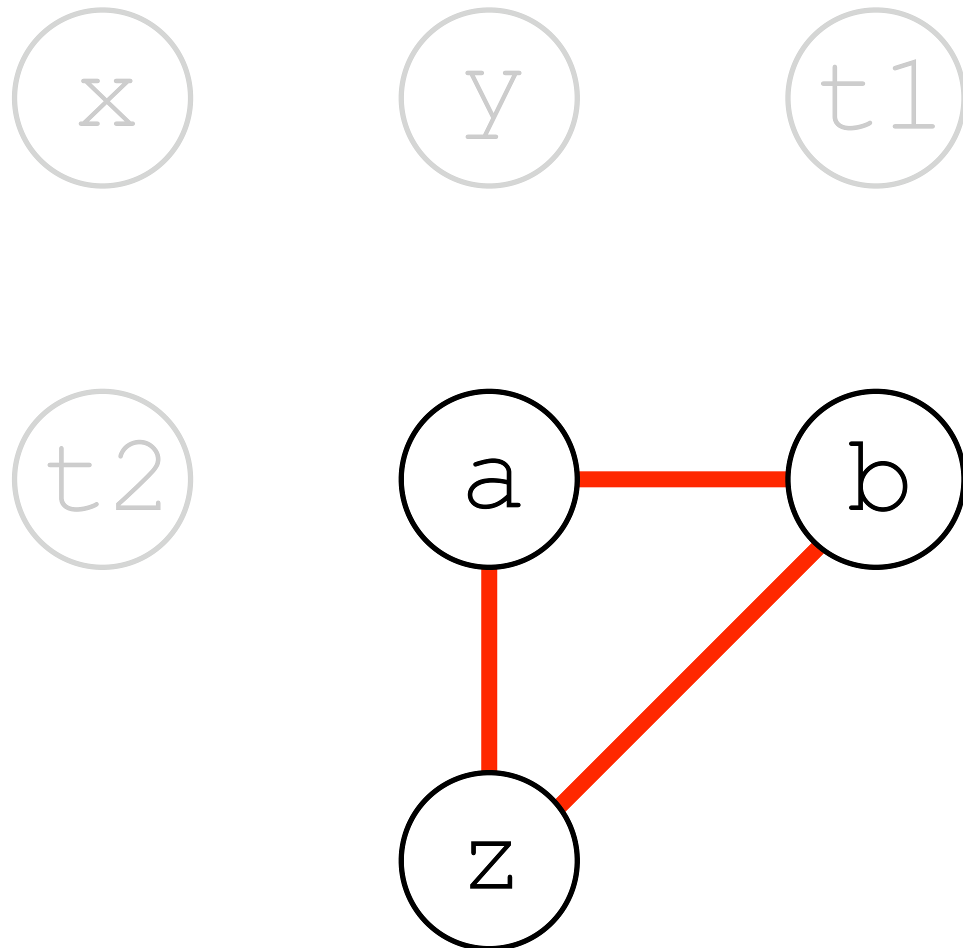
```
{ }
{ x }
{ x, y }
{ t1 }
{ z }
{ a, z }
{ a, b, z }
{ t2, z }
```

Clash graphs



This graph shows us, for example, that `a`, `b` and `z` must all be kept in separate registers, but that we may reuse those registers for the other variables.

Clash graphs




```
MOV a,#11
MOV b,#13
ADD a ,a,b
MUL z,a ,#2
MOV a,#17
MOV b,#19
MUL a ,a,b
ADD z,z,a
```

Summary

- Data-flow analysis is helpful in locating (and sometimes correcting) data-flow anomalies
- LVA allows us to identify dead code and possible uses of uninitialised variables
- Write-write anomalies can be identified with a similar analysis
- Imprecision may lead to overzealous warnings
- LVA allows us to construct a clash graph