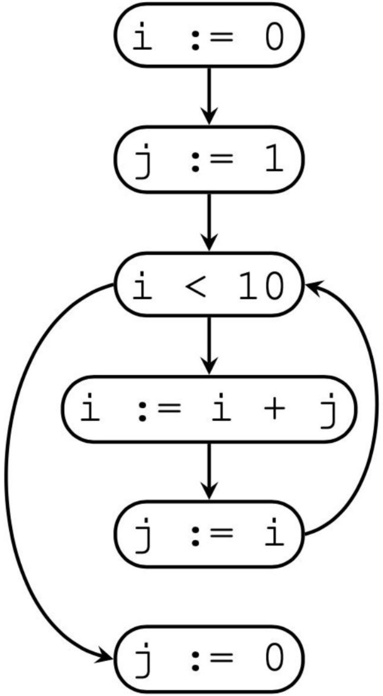
CSC410: Dataflow Analysis

September 29, 2023

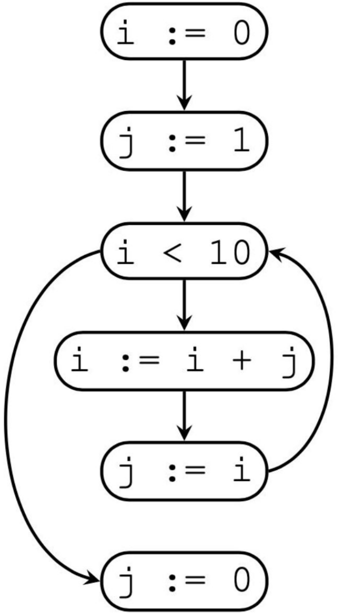
Recall: Live Variable Analysis

```
i = 0;
j = 1;
while (i < 10) {
  i = i + 1;
  j = i
}
j = 0;
```



Dataflow analysis operates on the control-flow graph (CFG).

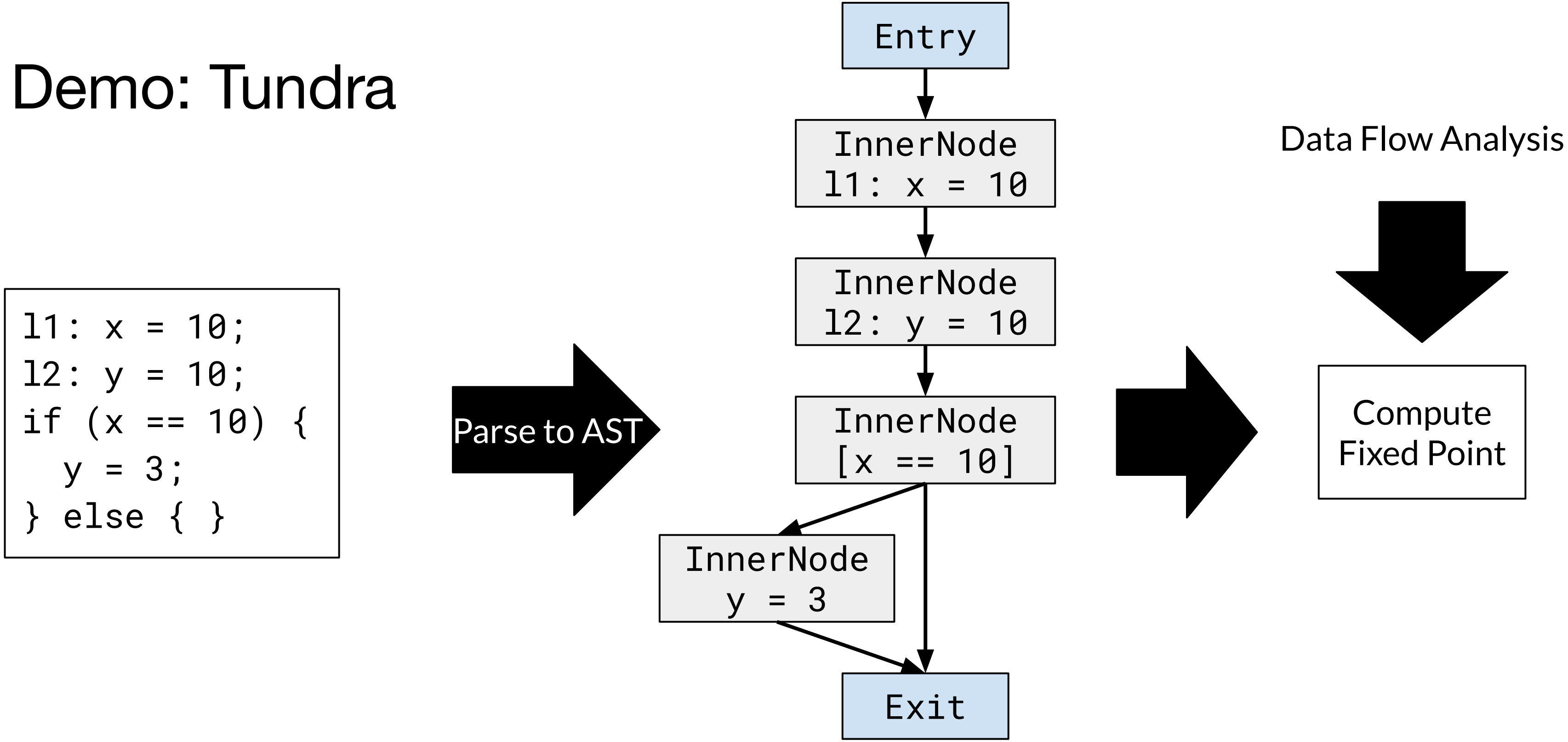
Recall: Live Variable Analysis



Label	LV_{exit}	LV_{entry}
`i := 0`	{i}	∅
`j := 1`	{i,j}	{i}
`i < 10`	{i,j}	{i,j}
`i := i + j`	{i}	{i,j}
`j := i`	{i,j}	{i}
`j := 0`	∅	∅

CFG nodes have values that are updated through *transfer functions*.

Demo: Tundra



Tundra is a toy language and dataflow analysis framework (more detail later).

Demo: Live Variable Analysis in Tundra

Domain

Powerset of program variables

Initial Values

Empty Set

Transfer Function

$$
\begin{aligned}
LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\
LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')
\end{aligned}
$$

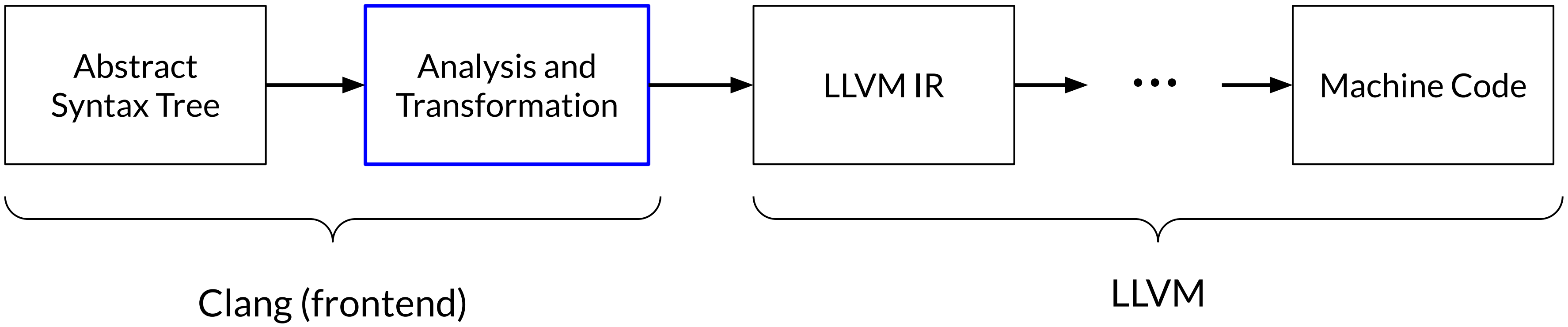
Demo: Live Variable Analysis in Tundra

Real World Applications: Dataflow Analysis

Clang: a C language family frontend for LLVM

The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project. Both a GCC-compatible compiler driver (`clang`) and an MSVC-compatible compiler driver (`clang-cl.exe`) are provided. You can get and build the source today.

Real World Applications: Clang compilation pipeline



Lesser-Known Clang Tools…

Refactoring

Clang's refactoring engine

This document describes the design of Clang's refactoring engine and provides a couple of examples that show how various primitives in the refactoring API can be used to implement different refactoring actions. The **LibTooling** library provides several other APIs that are used when developing a refactoring action.

Tidy

clang-tidy is a clang-based C++ "linter" tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis. **clang-tidy** is modular and provides a convenient interface for writing new checks.

Clang provides tools for analysis on the AST. LLVM actually compiles.

Clang Dataflow Example: Automatic Refactoring

```
struct Customer {
  int account_id;
  std::string name;
}

void GetCustomer(Customer *c) {
  c->account_id = ...;
  if (...) {
    c->name = ...;
  } else {
    c->name = ...;
  }
}
```

Before C++11, there were no “move” semantics.

Functions used “output parameters” to pass around pointers.

Clang Dataflow Example: Automatic Refactoring

```
struct Customer {
  int account_id;
  std::string name;
}

void GetCustomer(Customer *c) {
  c->account_id = ...;
  if (...) {
    c->name = ...;
  } else {
    c->name = ...;
  }
}
```

```
Customer GetCustomer() {
  Customer c;
  c.account_id = ...;
  if (...) {
    c.name = ...;
  } else {
    c.name = ...;
  }
  return c;
}
```

Idiomatic C++ code should use return values in this case.

Clang Dataflow Example: Identify Candidates with DFA

```
struct Customer {
  int account_id;
  std::string name;
}

void GetCustomer(Customer *c) {
  c->account_id = ...;
  if (...) {
    c->name = ...;
  } else {
    c->name = ...;
  }
}
```

Candidates for refactoring fulfill the following properties:

1. pointee is completely overwritten by the function
2. pointee is not read before it is overwritten

Question: is c an output parameter?

Clang Dataflow Example Two

```
struct Customer {
  int account_id;
  std::string name;
}

void GetCustomer(Customer *c) {
  c->account_id = ...;
  if (...) {
    c->name = ...;
  }
}
```

Candidates for refactoring fulfill the following properties:

1. pointee is completely overwritten by the function
2. pointee is not read before it is overwritten

Question: is c an output parameter?

Clang Dataflow Example Two

```
struct Customer {
  int account_id;
  std::string name;
}
```

```
void GetCustomer(Customer *c) {
  c->account_id = ...;
  if (...) {
    c->name = ...;
  }
}
```

Candidates for refactoring fulfill the following properties:

1. **pointee is completely overwritten by the function**
2. pointee is not read before it is overwritten

Result of DFA: there exists a path with no stores to `c->name`.
Example two is not a refactoring candidate.

Other Clang-Based Refactor Examples Using DFA

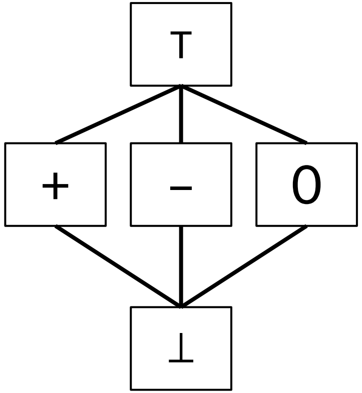
- Refactor raw pointers to unique_ptr
- Find dead stores
- Finding uninitialized variables
- Sign analysis

Some Actual Code…

```
template <typename Derived, typename LatticeT>
class DataflowAnalysis : public TypeErasedDataflowAnalysis {
public:
  /// Bounded join-semilattice that is used in the analysis.
  using Lattice = LatticeT;

  explicit DataflowAnalysis(ASTConte
```

```
class SignPropagationAnalysis
    : public DataflowAnalysis<SignPropagationAnalysis, NoopLattice> {
public:
  SignPropagationAnalysis(ASTContext &Context)
      : DataflowAnalysis<SignPropagationAnalysis, NoopLattice>( &: Context),
        TransferMatchSwitch(buildTransferMatchSwitch()) {}

  static NoopLattice initialElement() { return {}; }

  void transfer(const CFGElement &Elt, NoopLattice &L, Environment &Env) {
    LatticeTransferState State( &: L,  &: Env);
    TransferMatchSwitch(Elt,  &: getASTContext(),  &: State);
  }
```

Lattices: Not So Complicated




```
bool SignPropagationAnalysis::merge(QualType Type, const Value &Val1,
                                    const Environment &Env1, const Value &Val2,
                                    const Environment &Env2, Value &MergedVal,
                                    Environment &MergedEnv) {
  if (!Type->isIntegerType())
    return false;
  SignProperties Ps1 = getSignProperties( Val: Val1,  Env: Env1);
  SignProperties Ps2 = getSignProperties( Val: Val2,  Env: Env2);
  if (!Ps1.Neg || !Ps2.Neg)
    return false;
  BoolValue &MergedNeg =
      mergeBoolValues( &: *Ps1.Neg, Env1,  &: *Ps2.Neg, Env2,  &: MergedEnv);
  BoolValue &MergedZero =
      mergeBoolValues( &: *Ps1.Zero, Env1,  &: *Ps2.Zero, Env2,  &: MergedEnv);
  BoolValue &MergedPos =
      mergeBoolValues( &: *Ps1.Pos, Env1,  &: *Ps2.Pos, Env2,  &: MergedEnv);
  setSignProperties( &: MergedVal,
                     Ps: SignProperties{ .Neg: &MergedNeg,  .Zero: &MergedZero,  .Pos: &MergedPos});
  return true;
}
```

Other Levels of Dataflow Analysis: LLVM IR Examples

```
for (int i = 0; i < n; ++i)
  a[i] = b[i] * c[i];
```

Vectorize

```
for (int i = 0; i < n/4*4; i += 4) {
  b_v = vec_load(&b[i]);
  c_v = vec_load(&c[i]);
  store_vec(&a[i], b_v*c_v)
}
```

This transformation is unsafe! Why?

Other Levels of Dataflow Analysis: LLVM IR Examples

```
for (int i = 0; i < n; ++i)
  a[i] = b[i] * c[i];
```

Vectorize

```
for (int i = 0; i < n/4*4; i += 4) {
  b_v = vec_load(&b[i]);
  c_v = vec_load(&c[i]);
  store_vec(&a[i], b_v*c_v)
}
```

b and c must point to different memory regions than a.

Other Levels of Dataflow Analysis: Alias Analysis

```
for (int i = 0; i < n; ++i)
  a[i] = b[i] * c[i];
```

Alias Analysis →

Alias Sets:

{a, b} a **may alias** b
{a, c} a **may alias** c

```
if !alias(a, b) && !alias(a, c)
  vectorizedLoop();
else
  originalLoop();
```

Generate runtime checks to guard the loops.