Dynamic Inference of Abstract Types

Philip J. Guo, Jeff H. Perkins, Stephen McCamant, Michael D. Ernst
Computer Science and A.I. Lab
Massachusetts Institute of Technology

Declared types

```
// Order cost = base cost + tax + shipping
int totalCost(int miles, int price, int tax)
{
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

Few **declared types** (e.g., `int`, `float`) often used for conceptually-distinct values

Abstract types

```
// Order cost = base cost + tax + shipping
Money totalCost(Distance miles, Money price, Money tax)
{
  Time year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    Money shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

- Values of the same **abstract type** are conceptually similar and can be used in the same contexts
- Inferring abstract types:
  - Value interactions unify their abstract types
  - Variables have the same abstract type if their values do

Uses of abstract types

- **For program understanding**
  - Indicates how variables relate
  - Case study demonstrates effectiveness
- **For program development**
  - Compare inferred types to expectations
  - Bug finding, refactoring
- **For automated analysis tools**
  - Tools operate on variables of the same type
  - Abstract types finer than declared types, so analysis results and performance improve

Inference of abstract types

- **The problem:** Automatically infer abstract types from a program
- **Previous work:** Static analysis [O'Callahan97]
  - Examples of imprecision:
    - Flow-insensitive - Each variable has only 1 abstract type throughout execution
    - Confounds values stored inside of arrays
- **Our contribution:** The first known dynamic approach

Dynamic inference of abstract types

- **Technique**
  - Observe interactions to infer types for values
  - Merge value types to obtain variable types
- **Implementations**
  - x86/Linux binaries (C/C++), Java bytecodes
- **Evaluation**
  - Accuracy
  - Program understanding
  - Invariant detection

Dynamic inference of abstract types

- **Technique**
  - Observe interactions to infer types for **values**
  - Merge value types to obtain **variable** types
- **Implementations**
  - x86/Linux binaries (C/C++), Java bytecodes
- **Evaluation**
  - Accuracy
  - Program understanding
  - Invariant detection

Infer abstract types for values

- Maintain disjoint *interaction sets* of values
  - Each set represents an abstract type
- **Value creation:**
  - Each new value is placed into a singleton *interaction set*
  - New value created from a literal in the code (e.g., 42), data read from file, or user input
- **Value interaction:**
  - When values interact during execution, merge their *interaction sets*

Value interaction

- An interaction is a binary operation
- Interactions convey programmer intent
- Interactions merge value abstract types
  - arithmetic (+, -, *, /, ...) `profit = revenue - cost`
  - comparison (==, <, >, ...) `isWin = myScore > yourScore`
  - logical (&&, ||, ...) `if (p && *p) { ... }`

Value type inference example

```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

Value type inference example

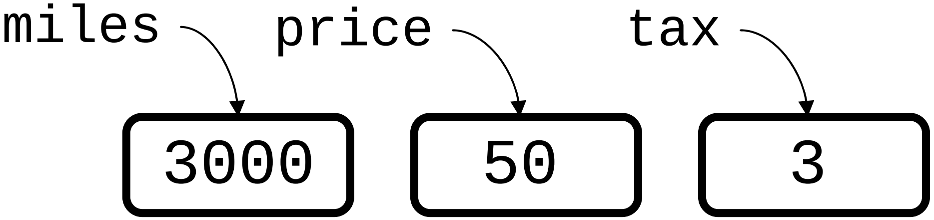
```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

```
totalCost(3000, 50, 3);
```

Value type inference example

```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

`totalCost(3000, 50, 3);`



miles

price

tax

3000

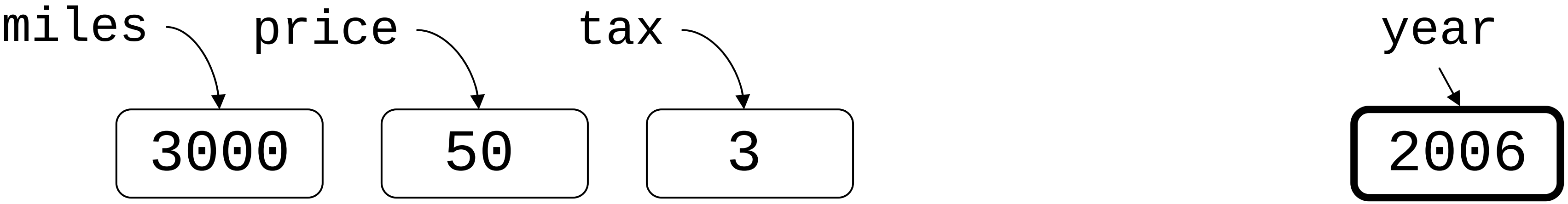
50

3

Value type inference example

```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

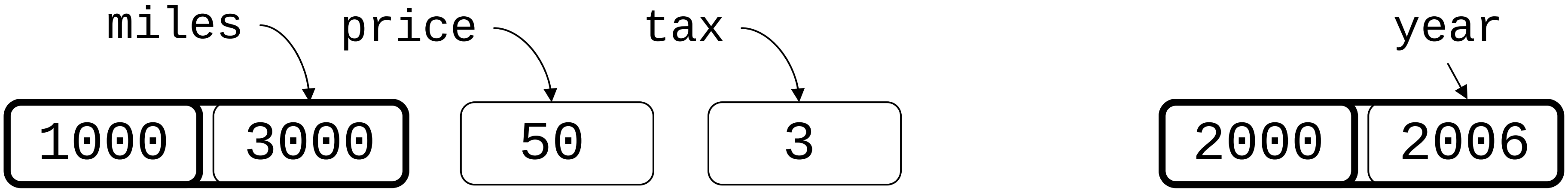
`totalCost(3000, 50, 3);`



Value type inference example

```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

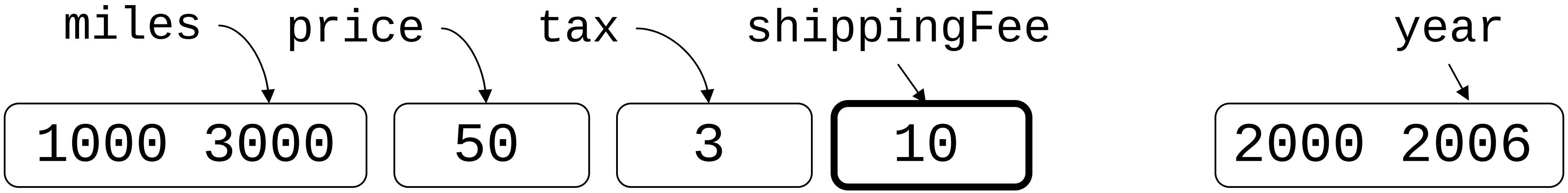
`totalCost(3000, 50, 3);`



Value type inference example

```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

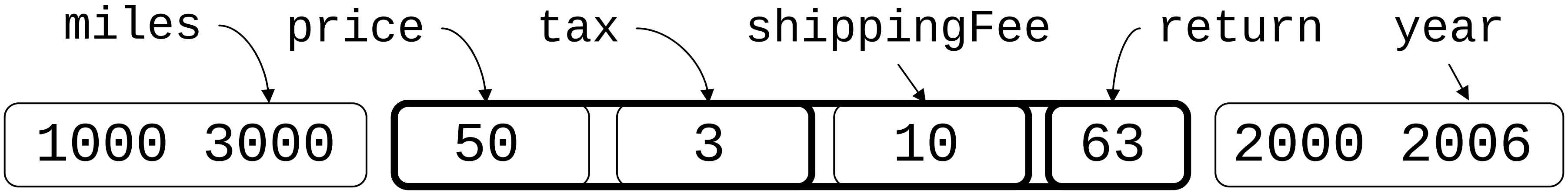
`totalCost(3000, 50, 3);`



Value type inference example

```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

totalCost(3000, 50, 3);



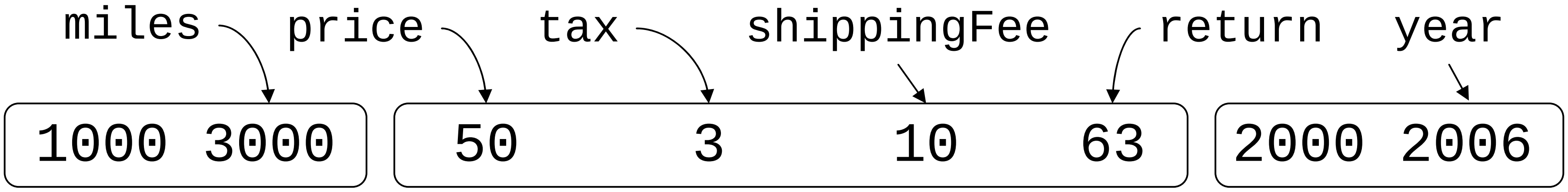
miles price tax shippingFee return year

1000 3000 | 50 | 3 | 10 | 63 | 2000 2006

Infer abstract types for variables

```
int totalCost(int miles, int price, int tax) {
  int year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    int shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

```
totalCost(3000, 50, 3);
```

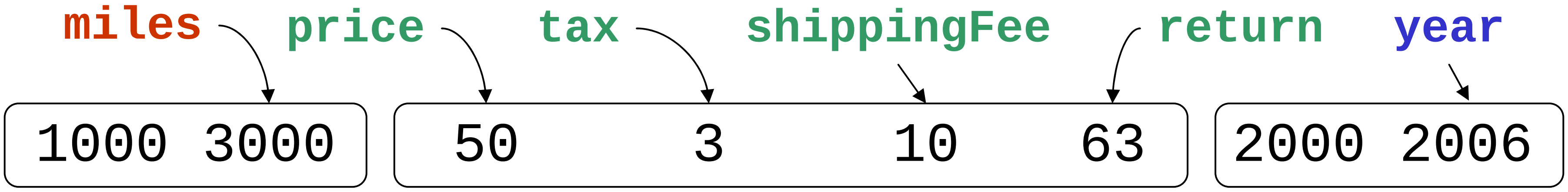


- Variables have the same abstract type if their values do
- Occurs at function entrance and exit points

Variable type inference example

```
Money totalCost(Distance miles, Money price, Money tax) {
  Time year = 2006;
  if ((miles > 1000) && (year > 2000)) {
    Money shippingFee = 10;
    return price + tax + shippingFee;
  } else {
    return price + tax;
  }
}
```

```
totalCost(3000, 50, 3);
```



Analysis output for totalCost():

{miles}, {price, tax, shippingFee, return}, {year}

Why track values?

- Naturally achieves context- and flow- sensitivity

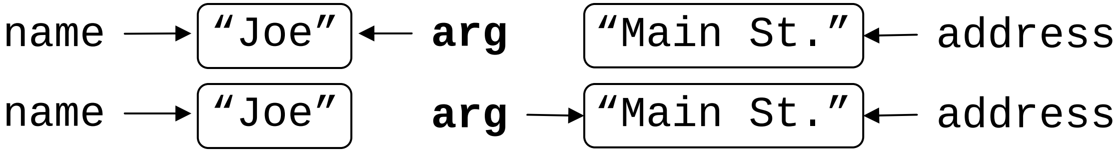
```
int strlen(char* arg);
char *name, *address;     {name}, {address}
```

Tracking variables

```
strlen(name);             {name, arg}, {address}
strlen(address);          {name, arg, address}
...
// Use name, address      {name, address}
```

Tracking values

```
strlen(name);
strlen(address);
...
// Use name, address      {name}, {address}
```



name → "Joe" ← **arg**   "Main St." ← address

name → "Joe"   **arg** → "Main St." ← address

Dynamic inference of abstract types

- **Technique**
  - Observe interactions to infer types for values
  - Merge value types to obtain variable types
- **Implementations**
  - x86/Linux binaries (C/C++), Java bytecodes
- **Evaluation**
  - Accuracy
  - Program understanding
  - Invariant detection

Implementations

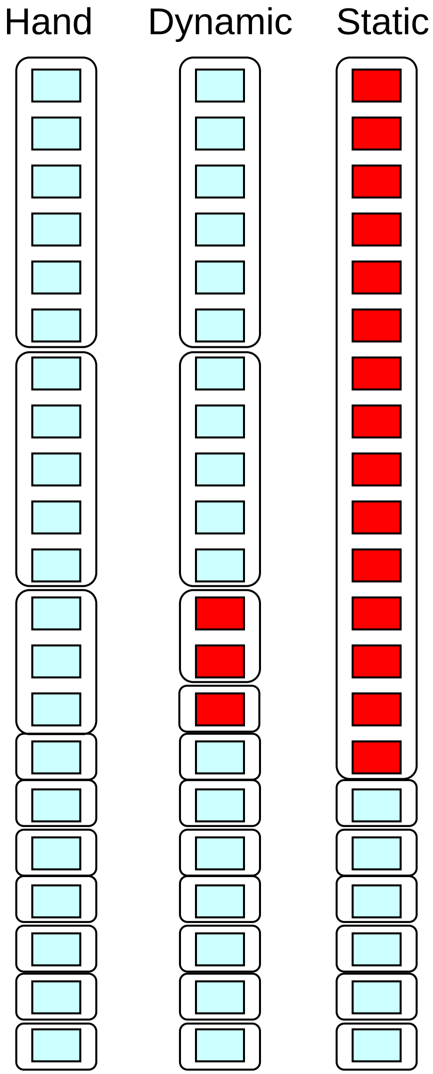
- Maintain a 32-bit tag along with each value
- Instrumentation code creates tags, copies tags, and unifies *interaction sets* of tags
- For x86/Linux binaries (currently C and C++)
  - Dynamic binary instrumentation using Valgrind
  - Tag for each register and for every byte of memory
- For Java 1.5 programs
  - Bytecode instrumentation using BCEL
  - Tag for every primitive variable on stack and for every primitive field within objects

Dynamic inference of abstract types

- **Technique**
  - Observe interactions to infer types for values
  - Merge value types to obtain variable types
- **Implementations**
  - x86/Linux binaries (C/C++), Java bytecodes
- **Evaluation**
  - Accuracy
  - Program understanding
  - Invariant detection

Evaluation of accuracy

- anagram generator program (740 LOC)
- 21 global variables
- Hand examination of code and comments revealed 10 abstract types
  - e.g., “word”, “word count”
- Our dynamic analysis found 11 types
  - Failed to unify two variables of type “word length” because their values never interact
- Static analysis (Lackwit) found 7 types
  - Failed to distinguish elements of `argv[]` array
  - `wordplay –d<depth> <word> -f <dictionary>`
  - Confounded “recursion depth” and “word” types



Program understanding – `RNAfold`

- RNA folding program written in C (1804 LOC)
- Programmer refactored 55 `int` variables of abstract type *energy* to type `double`
  - Took 16 hours of work to find the *energy* variables amongst hundreds of `int`s; tedious & error-prone
  - 2 iterations before he was confident of correctness
- Ran our analysis on a 100 base pair RNA sequence
  - Found 60 `int` variables in one abstract type
  - 5 non-*energy* variables were used inconsistently in complex initialization code
  - He quickly recognized and filtered out these variables
- Programmer estimated that our tool would have saved 90-95% of his effort

Program understanding – SVM-Light

- Support vector machine written in C (5834 LOC)
- Programmer wanted to understand and port it
- Our analysis increased his confidence in his understanding of the algorithm
- Perfect correspondence for “error bounds” vars.
- A variable `buffer` was in the same type as many other variables
  - He initially suspected tool imprecision
  - He learned that `buffer` was used pervasively

Invariant detection with abstract types

- Daikon uses machine learning to infer relations between variables (e.g., `tax < price`)
  - Only compares variables of the same type
- Abstract types improve results
  - Relations between variables of different abstract types are likely to be spurious (e.g., `miles > tax`)
  - Produces fewer and more relevant invariants
- Abstract types improve run time and memory use
  - No need to find relations between variables of different abstract types

Invariant detection with abstract types

	Time	Memory	# invariants
Representation types (default)	1.0	1.0	1.0
Declared types	0.85	0.84	0.70
Abstract types	0.65	0.64	0.13

- Averages for 8 programs (C and Java)
- We examined many eliminated invariants; all spurious
- Greater improvements on larger programs
  - Largest C program was a 17 KLOC module within `perl` (105 KLOC)
  - Largest Java program was a 13 KLOC module within `javac` (40 KLOC)
- Static analysis did not scale

Contributions

- Dynamic approach to inference of abstract types
  - Operates on values; maps to variables
  - Conceptually simple, precise, and effective in practice
- Implementations for C/C++ and Java
- Evaluation
  - Accurate
  - Assists programmers in understanding code
  - Improves results and performance of an automated tool