The Whiley Programming Language

David J. Pearce
Victoria University of Wellington
New Zealand

http://whiley.org
Java:

```java
void buildLabelMap(List<Bytecode> bytecodes) {
    HashMap<String,Integer> labels = new HashMap<String,Integer>();
    int idx = 0
    for(Bytecode b : bytecodes) {
        if(b instanceof Bytecode.Label) {
            Bytecode.Label lab = (Bytecode.Label) b;
            labels.put(lab.name, idx);
        }
        idx = idx + 1;
    }
}
```

Python:

```python
def buildLabelMap(bytecodes):
    labels = {}
    idx = 0
    for b in bytecodes:
        if type(b) == "Label":
            labels[b.name] = idx
        idx = idx + 1
```
The Whiley Language

void buildLabelMap([Bytecode] bytecodes):
    labels = {->}
    idx = 0
    for b in bytecodes:
        if b is Label:
            labels[b.name] = idx
            idx = idx + 1

• Design Goals:
  - Look and feel of a dynamically typed language
  - But, still provide static type checking
  - Simple programming model
  - Amenable to program verification
Data Types

• **Unbound Integers and Rationals**
  - *int* currently implemented as BigInteger
  - *real* currently implemented as BigRational
  - How to do this efficiently on JVM?

• Lists, Sets and Maps:

```java
int sum([int] list):
    r = 0
    for x in list:
        r = r + x
    return r
```

• Records:

```java
define Point as {int x, int y}
```
int sum([int] list) requires no \{x in $ | x < 0\},
ensures $ >= 0:

    r = 0
    for x in list:
        r = r + x
    return r

define nat as int where $ >= 0
define natlist as [nat]

nat sum(natlist list):
    ...
    return r

• **Pre/Post Conditions and Invariants:**
  - **Goal** is to check at compile time
  - **Currently**, checked at run time
Flow-Sensitive Typing

Value evaluate(Expr e, {string->Value} env) throws Error:
    if e is int:
        return e
    else if e is Var:
        return env[e.id]
    else if e is BinOp:
        lhs = evaluate(e.lhs, env)
        rhs = evaluate(e.rhs, env)
        ...

• Flow-Sensitive Typing:
  - Variables declared by assignment
  - Variables automatically retyped by type tests
  - Variables can have different types at different points
Union Types

- Union types have many uses
  - Such as neatly handling null references
  - Or, combining different kinds (i.e. unions of structs)

```python
null|int indexOf(string str, char c):
    ...

[string] split(string str, char c):
    idx = indexOf(str,c)
    if idx is int:
        below = str[0..idx]
        above = str[idx..]
        return [below,above]
    else:
        return [str]
```
Structural Subtyping

- Defined types are **not nominal**
  - i.e. `LinkedList` is just a name that “expands”

```java
define LinkedList as null | {int dat, LinkedList nxt}

int sum(LinkedList l):
    if l == null:
        return 0
    else:
        return l.dat + sum(l.nxt)

void main(System sys, [string] args):
    l={dat: 1, nxt: null}
    l={dat: 2, nxt: l}
    sys.out.println(sum(l))
```
Define LinkedList as null | {int dat, LinkedList nxt}

int sum(LinkedList l):
    if l == null:
        return 0
    else:
        return l.dat + sum(l.nxt)

void main(System sys, [string] args):
    l={dat: 1, nxt: null}  // l is {int dat, null nxt}
    l={dat: 2, nxt: l}    // l is {int dat, {int dat, null nxt} nxt}
    sys.out.println(sum(l))

• Defined types are not nominal
  - i.e. LinkedList is just a name that “expands”
Value Semantics

Board applyMove(Move move, Board board) throws Move.Invalid:
    nboard = applyMoveDispatch(move, board)
    if !validMove(move, board, nboard):
        throw Move.Invalid(board, move)
    else:
        return nboard

- Whiley does not have references!
  - Everything is pass-by-value
  - Data propagates only via return
  - Much more functional in nature
  - Requires different way of thinking
Board applySimple(Board b, Pos old, Pos pos, Piece piece):
    b[old.col][old.row] = null
    b[pos.col][pos.row] = piece
    return board

Board applyMove(Board b, Move m):
    if move is Simple:
        ...
        return update(b, m.from, m.to, m.piece)

• Value semantics:
  - Copy board for call to applySimple()
  - Copy again for assignments in applySimple()
  - But, this is very inefficient!!!

  - Reference counting can really help here...
Digression

- Item 24, Effective Java
  - Make Defensive Copies when Needed

“It is essential to make a defensive copy of each mutable parameter to the constructor”

--- Josh Bloch
# Effectiveness of Ref Counting

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th># Clones</th>
<th>% Clones</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gunzip</td>
<td>815</td>
<td>873 / 140561</td>
<td>0.62%</td>
</tr>
<tr>
<td>JASM</td>
<td>2333</td>
<td>12878 / 29968</td>
<td>43.0%</td>
</tr>
<tr>
<td>Chess</td>
<td>784</td>
<td>6438 / 416116</td>
<td>1.6%</td>
</tr>
<tr>
<td>Calc</td>
<td>225</td>
<td>0 / 81527</td>
<td>0.0%</td>
</tr>
<tr>
<td>SCC</td>
<td>169</td>
<td>12602 / 258968</td>
<td>4.86%</td>
</tr>
</tbody>
</table>
Performance
Java Interoperation

**Sum.whiley:**

```java
public native int sum([int] list):
...
```

**Sum$native.java:**

```java
public BigInteger sum(wyjc.runtime.List list) {
    ...
}
...
```
Testing

- Some 600 end-end tests
- Over 15,000 unit tests of type system!
Eclipse Plugin

- **Update Site:** http://whiley.org/eclipse
http://whiley.org