A Complete and Inconsistent Programming Language

Mike Burrell
mburrel@uwo.ca

UWORCS 2007
• Classical logics
  • must be true or false
  • cannot be true and false
• Intuitionistic logics
  • cannot be true and false
  • could be neither
• Inconsistent logics
  • must be true or false
  • can be both true and false!!
Intuitionistic Logics

• Gödel’s Incompleteness Theorem:
  • Sometimes a proposition cannot be proven true or false.
  • Intuitionistic logic says it’s neither true nor false!
Inconsistent Logics

- What if we got rid of consistency?
- We could get completeness.
- Tradeoff: some things can be true and false.
- :O
x := Dictionary new.
x add: "hello" withKey: 1.
x add: "goodbye" withKey: 2.
y := x allKeys.

Java
Dictionary x = new Dictionary();
x.add("hello", 1);
x.add("goodbye", 2);
List y = x.allKeys();
What’s an object?

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>yesno</th>
</tr>
</thead>
<tbody>
<tr>
<td>strings</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>integers</td>
<td>meh</td>
<td>yes</td>
</tr>
<tr>
<td>classes</td>
<td>mostly</td>
<td>yes</td>
</tr>
<tr>
<td>conditionals</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>loops</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>code</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
List := Object subclass.
List addInstanceMessage: "append:" withMethod:
    [ o | if cond: (self hasVarSet: "head")
        then: [| self.tail add: o. ]
        else: [| self.head = o. self.tail = List new. ].
].
x := List new.
x add: 3.
x add: "hello".
return x.
Inconsistency

- Message calls are the foundation of iteration and recursion.
- The language is complete if we take away message calls!
Inconsistency

Now for every message call we consider 2 possibilities:

• Call the method
• Don’t call the method
• If the method eventually halts, we ignore the second state of things
Closure addInstanceMessage: "haltsWith:"
withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.
].

z := 5.
if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. |]
  else: [| z := 7. |].
return z.

z = {}
Closure addInstanceMessage: "haltsWith:"

withMethod: [:x |
    halts := false.
    self doWith: x.
    halts := true.
    return halts.
].

z := 5.
if cond: ([ n : n * n ] haltsWith: z)
    then: [|z| z := 6.]
    else: [|z| z := 7.].
return z.

z = {5}
Closure addInstanceMessage: "haltsWith:"
withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.
].

z := 5.
if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. |]
  else: [| z := 7. |].
return z.

z = {5}
Closure addInstanceMessage: "haltsWith:"
withMethod: [
    x |
    halts := false.
    self doWith: x.
    halts := true.
    return halts.
].
z := 5.
if cond: ([ n : n * n ] haltsWith: z)
then: [| z := 6. ]
else: [| z := 7. ].
return z.

z = {5}
Closure addInstanceMessage: "haltsWith:"
withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.
].
z := 5.
if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. |
  else: [| z := 7. |].
return z.

z = {5}
Closure addInstanceMessage: "haltsWith:"
withMethod: [ x |
    halts := false.
    self doWith: x.
    halts := true.
    return halts.
].
z := 5.
if cond: ([ n : n * n ] haltsWith: z)
    then: [| z := 6. |]
    else: [| z := 7. |].
return z.

halts = {false}
z = {5}
Closure addInstanceMessage: "haltsWith:"

withMethod: [ x |
    halts := false.
    self doWith: x.
    halts := true.
    return halts.
].
z := 5.
if cond: ([ n : n * n ] haltsWith: z)
    then: [| z := 6. |]
    else: [| z := 7. |].
return z.
Closure addInstanceMessage: "haltsWith:"

withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.
].

z := 5.
if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. |]
  else: [| z := 7. |].
return z.
Closure addInstanceMessage: “haltsWith:”

withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.
].

z := 5.

if cond: ([ n : n * n ] haltsWith: z)
  then: [ [ z := 6. ] ]
  else: [ [ z := 7. ] ].

return z.
Closure addInstanceMessage: "haltsWith:"
withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.]

z := 5.
if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. ]
  else: [| z := 7. ].
return z.

halts = \{false, true\}
z = \{5, 6, 7\}
Closure addInstanceMessage: "haltsWith:"

withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.
].

z := 5.

if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. |]
  else: [| z := 7. |].

return z.
Closure addInstanceMessage: “haltsWith:”
  withMethod: [ x |
    halts := false.
    self doWith: x.
    halts := true.
    return halts.
  ].

z := 5.
if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. ]
  else: [| z := 7. ].
return z.
Closure addInstanceMessage: "haltsWith:"

withMethod: [ x |
    halts := false.
    self doWith: x.
    halts := true.
    return halts.
].

z := 5.

if cond: ([ n : n * n ] haltsWith: z)
    then: [| z := 6. |]
    else: [| z := 7. |].

return z.
Closure addInstanceMessage: "haltsWith:"

withMethod: [ x |
  halts := false.
  self doWith: x.
  halts := true.
  return halts.
].
z := 5.
if cond: ([ n : n * n ] haltsWith: z)
  then: [| z := 6. |]
  else: [| z := 7. |].
return z.
Skipped call merging

- Idea: once a method finishes, backtrack and update execution
  - As if skipping the message call never happened

- Downside: extremely¹ slow!

- BUT! If a language is recursive, the added inconsistency will disappear!

¹ gross understatement
haltsWith:
...

if cond: (z haltsWith: x)
then: [: 3 ]
else: [: 4 ]

f := [: if cond: (z haltsWith: x)
then: [: 3 ]
else: [: 4 ] ].

if cond: (y haltsWith: w)
then: [: (f do) * 2 ]
else: [: (f do) * 3 ]
z := ...; x := ...;
y := ...; w := ...;

f := [: if cond: (z haltsWith: x)
then: [: 3 ]
else: [: 4 ] ].

if cond: (y haltsWith: w)
then: [: (f do) * 2 ]
else: [: (f do) * 3 ]
z := ...; x := ...;
y := ...; w := ...;
f := [: if cond: (z haltsWith: x)
then: [: 3 ] else: [: 4 ] :].
if cond: (y haltsWith: w)
then: [: (f do) * 2 ]
else: [: (f do) * 3 ]
z := ...; x := ...;
y := ...; w := ...;
f := [: if cond: (z haltsWith: x)
then: [: 3 ]
else: [: 4 ] ].

if cond: (y haltsWith: w)
then: [: (f do) * 2 ]
else: [: (f do) * 3 ]

6, 8, 9, 12
RE and co-RE problems

- General method: the program “terminates” right away by skipping messages
- As computation goes on (considering that the messages were sent), the result is updated
- Eventually the computation may terminate, and we find the answer
Uses

• Dealing with (possibly) undecidable problems

• Dealing with intermediate results:
  • UI updates
  • soft realtime
Future directions

• Force the correct answer to be returned

• Performance optimizations
  • Static analysis?

• Incorporate more side-effects (I/O?)

• Framework for existing language?
http://www.csd.uwo.ca/~mburrel/yesno/

Questions?