Introducing Wybe — a language for everyone

Peter Schachte
joint work with Matthew Giuca

The University of Melbourne
Department of Computing and Information Systems

4 December 2013
Motivation

- Many students have difficulty learning to program
- Python is simple and easy to learn, but:
  - It’s not efficient enough for some uses
  - Lack of static type checking hampers its use in large projects
- Java is efficient and scales well, but:
  - It is rather complex
  - It has numerous pitfalls
- Haskell is efficient and fairly simple, but:
  - Students have trouble with some of its concepts
  - Many students don’t find it intuitive
- Need a language that can span from learning through to practice
Student issues with Java

- Aliasing and its dangers, defensive copying, immutability
- Deep vs. shallow copying and equality
- Differences between primitive, object, and array types
- Static variables/methods and static classes
- Combinations of privacy, inheritance, static members
- Packages and build systems

But they can write non-trivial programs — build-and-fix works
Student issues with Haskell and Mercury

- Recursion
- Lack of destructive update
- Types of partially applied functions
- Monads
- Nondeterminism
- Some of the error messages
- Lack of a good IDE, debugger, REPL
- The numeric type classes (a bit)
- How (or why?) to take advantage of algebraic types
Several of the problems students have with Java stem from *action at a distance* — change happens for no apparent reason.

The problem: destructive update of aliased structures.

This is also a practical problem for software engineers.

Must have a mental model of computer memory.

Must have (and maintain!) a *global* understanding of aliasing.

Because this is impossible, lots of deep copying is recommended.

Because this is inefficient, lots of code lives dangerously.
For code to be *maintainable*, callers and callees should be able to develop and maintain their code independently.

A *local* understanding of each unit of code must be sufficient.

This requires a formal *interface* between callers and callees.

But there are really two interfaces:

- The **apparent interface** is what appears in declaration or call syntax.
- The **effective interface** between callers and callees is the information that passes into and out of callees.
Interface integrity

A function exhibits **interface integrity** if its apparent and effective interfaces are identical.

- This rules out:
  - Destructive update of aliased structures, since this would allow information flow not reflected in the apparent interface
  - Global variables, which would allow information flow from assignment to reference not reflected in the apparent interfaces
  - I/O (information flow into/out of the environment) without indication in the apparent interfaces
  - Unchecked exceptions

- This does not rule out:
  - Variable reassignment
  - Looping constructs

- Do what you like *inside* a function — as long as it’s not observable *outside*
Wybe basics

Simplicity is prerequisite for reliability.
— Edsger Wybe Dijkstra

Wybe is designed to:
- Enforce interface integrity
- Be easy to learn
- Scale to large applications
- Allow efficient implementation
- Support both functional and imperative programming

Wybe is in the early design stages

The syntax is not settled yet; take the following as an early conception
Hello World

- Comments introduced by hash (#)
- Hello World in Wybe:

  ```
  #!/usr/bin/env wybe
  !println("Hello, World!")
  ```

- Like a scripting language, top-level statements are executed
- I’ll explain the ! later
Information flow

- Direction of information flow (mode) is explicit
- A bit like Ada’s in, out, and in out
- Unadorned variable name denotes variable value (call by value)
- Caret (^) in front of variable name indicates variable (re-)assignment (call by result)
- Exclamation point (!) indicates both (call by value-result)
- \(^x = x + 1\) or \(x + 1 = ^x\) increments \(x\)
- so does \(\text{incr}(!x)\)
Procedures

- Same adornments are used in formal parameters
- `def foo(w, x, ^y, !z): ...` defines procedure with two inputs, one output, and one in-out parameter
- Adornments in call must match definition (but see below...)
- Body of a procedure definition is a sequence of statements
- There are a few built-in statement types, discussed below
- Procedure calls are statements
- `= and incr` are library procedures
Expressions

- Procedure call arguments can be expressions
- An expression can be a procedure call with the final argument omitted
- The value of such an expression is the value that would be assigned to its omitted argument
  
  \[ \text{E.g., } \text{bar}(x,y) \text{ as an expression means call } \text{bar}(x,y,^\text{temp}) \text{ and use temp as the value of } \text{bar}(x,y) \]

- \text{foo}(\text{bar}(x,y),^z) \text{ means } \text{bar}(x,y,^\text{temp})
- \text{foo}(\text{bar}(x,y),^z) \text{ means } \text{foo}(\text{temp},^z)

- \text{def foo}(x) = \text{bar}(x,x) \text{ is syntactic sugar for }
  
  \text{def foo}(x,^\text{result}): \text{bar}(x,x,^\text{result})

- Can use \text{foo} with either syntax regardless of which definition was used

- A few built-in expressions like \text{let stmts in expr and expr where stmts}
Reversibility

- Procedures can be overloaded based on mode
- \texttt{cons(head,tail,^list)} constructs
  \texttt{cons(^head,^tail,list)} deconstructs
- Expressions can be outputs (patterns) as well as inputs
- Expression \texttt{cons(h,t)} constructs list
  Expression \texttt{cons(^h,^t)} deconstructs
- \texttt{tail(!x, y)} replaces tail of \texttt{x} with \texttt{y}
- \texttt{tail(!x)} = \texttt{y} is exactly the same \quad \texttt{tail(x,^temp)}
- \texttt{head(tail(!x), y)} transforms to \quad \texttt{head(!temp,y)}
  \quad \texttt{tail(!x, temp)}
- Modes of all parameters but the last must uniquely determine the mode of the last
Value semantics

- Wybe has value semantics: aliasing is not semantically significant
- head(!list, val) does not mean RPLACA
- Equivalent to \(^\text{list} = \text{cons}(\text{val}, \text{tail}(\text{list}))\)
- Gives the feeling of changing values without action at a distance
- Compile-time garbage collection: when unique (unaliased), compiler transforms this (back) into destructive modification
- Can this be made predictable enough for programmers to have a good performance model and to write efficient code?
- Can this make declarative programming with arrays etc. practical?
Resources

- A *resource* is data that can be used and/or defined without being explicitly passed as a parameter.
- Similar to State and IO monads.
- Specified in procedure declaration, but not in call.
- Calls to procedures that use resources must be preceded with `!` to signify that they use some resources.
- Procedures can use as many resources as they want to declare.
- Resource can be declared as a name for several other resources.
- Useful for data that is widely used/modified in a module.
- I/O, command line arguments are resources visible at top level.

```python
def hello(name) with io:
    !print("Hello, ")
    !command_line([~name])
    !print(name)
```
Some procedure calls, called *tests*, can succeed or fail
Some modes of a procedure can be tests while others are not
Definition specifies that call can fail with `?` at left of signature
A test can also produce output: use it in place of a Maybe
  e.g., def ?cons(~head,~tail,list): ...
A test with no outputs can be used as an expression: it is reified into a bool
A call supplying an input where an output is expected is automatically a test that compares the output with the supplied input (like Mercury’s implied modes)
If statement

- Tests can be used in if statements
  
  ```
  if test1: statements ...
  test2: statements ...
  ...
  end
  ```

- Tests are tried in order; body of first to succeed is executed
- If none succeeds, none is executed
- Boolean expression $e$ is de-reified into the test $e = \text{true}$
- `else` is a test that always succeeds
- Also an expression version of if, where statements are replaced with expressions
- Tests of an if expression must be exhaustive
Tests as statements

- A test can be used as a statement
- Call must be preceded with `?`
- Sequence of statements is a test if any of them are tests
- Procedure is a test if its body is a test
- All tests must be declared with `?` at left
- Statement sequence fails if any of the tests fail
- Like logic programming or the Maybe monad
- If test fails, its effects are rolled back
- I/O is not allowed in tests
Case statement and expression

- case expr of
  - case1: body1 ...
  - case2: body2 ...
  ...

  is equivalent to

  if case1(expr): body1 ...
  case2(expr): body2 ...
  ...

- Except that tests must be exhaustive: checked at compile-time
- Can declare sets of exhaustive tests
- E.g.:

  def ++(x,y) = case x of
    []: y
    [^h|^t]: [h|t++y]
Loops

- One modular looping construct: `do loop-statements ...`

  `loop-statements` are any normal statements plus any special looping statements, including:
  - `while test` and `until test`
    - like conditional `break`
  - `when test` and `unless test`
    - like conditional `continue`
  - `for generator`

- Include as many of these constructs as you like in the loop, wherever you like, e.g.:
  
  ```
  do !print(prompt)
  !readln(^answer)
  until answer in ["y","n"]
  !println("Please answer 'y' or 'n'.")
  ```
Generators

- Generators are procedures that return any number of times
- Like `nondet` predicates in Mercury; similar to the list monad
- Generators are declared with a `*` before the signature
- Generators use `generate` statements to specify multiple results
- Each `generate` encloses multiple statements producing results from initial state
- Results are produced in the specified order
- Generators can call tests; if test fails, skip to next `generate`

E.g.,

```python
def *in(^elt,list):
    generate ?^elt = head(list)
    generate *^elt in tail(list)
```
Generators

- A procedure that calls a generator outside of a `for` construct is also a generator.
- A sequence of calls to generators generates the cross-product of their results.
- A test is like a generator restricted to at most one result.
- Generators can be used for logic programming-like coding.
- `for` statements allow iteration over generators, accumulating results.
- `do` loop can have multiple `for` statements to support lock-step iteration.
Types

- Type system is not designed yet; much work to be done
- These are some goals:
  - Strongly typed
  - Type inference for local variables and formal parameters of private procedures
  - Parametric polymorphism
  - Declaration of algebraic types produces constructors, deconstructors, accessors, mutators
  - Also possible to define all these as normal procedures, so one can directly implement types by defining their primitive operations
  - *E.g.*, can generate constructors, deconstructors, accessors, mutators for C structs passed through foreign interface
Types

- Interface inheritance: unify types with type classes
  - Abstract type ≜ type class
  - Allow a type $A$ to “implement” another type $B$, by defining all $B$’s primitive operations for type $A$
  - Then an $A$ is-a $B$: pass an $A$ where a $B$ is expected
  - E.g., allow list processing functions to work on arrays by defining car, cdr, and cons for arrays

- Implementation inheritance: declarative delegation
  - For a specified set of procedures, declare a function $f : a \rightarrow b$
    - to convert an $a$ argument to a $b$
  - Allows passing an $a$ for any $b$ type parameter to these procedures
  - Controlled coercion, or easy overloading
  - Allows composition to substitute for (multiple) inheritance
  - But no overriding
Simplicity does not precede complexity, but follows it.

— Alan Perlis

Looking for ways to simplify Wybe while satisfying its goals

More to add, too:

- Higher order
- Declared non-strict parameters
- Lightweight parallelism (generators can support this)
- “Identities” and relations among them, to allow networks of objects to be navigated and mutated OO-style