First-Class Nonstandard Interpretations by Opening Closures

Jeffrey Mark Siskind[1] Barak A. Pearlmutter[2]

[1]School of Electrical and Computer Engineering, Purdue University; qobi@purdue.edu
[2]Hamilton Institute, National University of Ireland Maynooth; barak@cs.nuim.ie

Symposium on Principles of Programming Languages
17 January 2007

What is a Nonstandard Interpretation (NSI)?

Reinterpret expression assigning new meanings to the free (constant, variable, function, and predicate) symbols.

(Terminology from model theory: nonstandard model of Peano Axioms, nonstandard integers, nonstandard interpretation of the reals, etc.)

What is a Nonstandard Interpretation (NSI)?

Reinterpret expression assigning new meanings to the free (constant, variable, function, and predicate) symbols.

(Terminology from model theory: nonstandard model of Peano Axioms, nonstandard integers, nonstandard interpretation of the reals, etc.)

What are NSIs used for?

- "lift" domain of language datatype: $\mathbb{R} \mapsto \mathbb{C}$, $\mathbb{R} \mapsto \mathbb{R}^3$, $\mathbb{R} \mapsto \mathbb{R}^{n\times n}$, ...
- systems programming: security sandbox, resource monitoring, tracing, logging, profiling, code instrumentation and metering, error checking, run-time code patching, virtualization, ...
- Web 2.0: redirecting I/O (e.g., AJAX)
- compiler techniques: flow analysis, partial evaluation, abstract interpretation, ...

$\vdots$

What is a Nonstandard Interpretation (NSI)?

Reinterpret expression assigning new meanings to the free (constant, variable, function, and predicate) symbols.

(Terminology from model theory: nonstandard model of Peano Axioms, nonstandard integers, nonstandard interpretation of the reals, etc.)

What are NSIs used for?

- "lift" domain of language datatype: $\mathbb{R} \mapsto \mathbb{C}$, $\mathbb{R} \mapsto \mathbb{R}^3$, $\mathbb{R} \mapsto \mathbb{R}^{n \times n}$, . . .
- systems programming: security sandbox, resource monitoring, tracing, logging, profiling, code instrumentation and metering, error checking, run-time code patching, virtualization, . . .
- Web 2.0: redirecting I/O (e.g., AJAX)
- compiler techniques: flow analysis, partial evaluation, abstract interpretation, . . .

  ⋮
- basically, everything good and wholesome!

How is NSI Typically Done?

- **Heavyweight**
  custom evaluator
  macros
  allows augmenting or reinterpreting core syntax
- **Lightweight**
  redefine variables bound to standard basis
  preserves core syntax
  reuses existing evaluator

Example: $\mathbb{C}$ for Language with Only $\mathbb{R}$

Redefine SCHEME numeric basis to operate on both native reals and complex numbers represented as SCHEME pairs `(`a `.` b`)`.

```
(define (complex+ +)
 (lambda (x y)
  (let ((x (if (pair? x) x (cons x 0)))
        (y (if (pair? y) y (cons y 0))))
   (cons (+ (car x) (car y))
         (+ (cdr x) (cdr y))))))

(define + (complex+ +))
```

Problems with this Approach

- Confining NSI to limited context
- Composing NSIs with specified order
- Multiple NSIs
- Reinterpreting constants: $x \mapsto$ `(`x ` . 0)`
- Lifted procedures must support both lifted and non-lifted values via dispatch

  ```
  (if (pair? x) x (cons x 0))
  ```

- Reinterpreting closed-over variables

  ```
  (define h
   (let ((plus +) (five 5))
    (lambda (x) (plus x five))))
  ```

What is a Nonstandard Interpretation?

Reinterpret expression assigning new meanings to the free (constant, variable, function, and predicate) symbols.

What is a closure?

An expression, along with an environment (a mapping of free variables to values).

What is a Nonstandard Interpretation?

Reinterpret expression assigning new meanings to the free (constant, variable, function, and predicate) symbols.

What is a closure?

An expression, along with an environment (a mapping of free variables to values).

Idea!

- Do NSI by altering closure environments

What is a Nonstandard Interpretation?

Reinterpret expression assigning new meanings to the free (constant, variable, function, and predicate) symbols.

What is a closure?

An expression, along with an environment (a mapping of free variables to values).

Idea!

- Do NSI by altering closure environments
- First-Class NSI (i.e., NSI objects that can be composed and applied)
- Treat constants as free (global) variables (i.e., constant conversion)
- Make API

An API for NSI

$$\texttt{map-closure}\, f\ \langle\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, e\rangle \triangleq \langle\{x_1 \mapsto (f\ x_1\ v_1), \ldots, x_n \mapsto (f\ x_n\ v_n)\}, e\rangle$$

- treat primitive procedures as having empty environments
- preserves hygiene: `(name` *var*`)` syntax, `name?`, `name=?`
- lazy: compute $(f\ x_i\ v_i)$ on first access to x_i

Useful Idiom

Recursively apply `per-slot` to each slot nested in `x` and `per-object` to each object nested in `x`.

```
(define (map-closure* per-slot per-object x)
 (let recurse ((x x))
  (per-object
   (cond
    ((procedure? x)
     (map-closure
      (lambda (n x) (recurse (per-slot n x)))
      x))
    ((pair? x)
     (cons (recurse (car x)) (recurse (cdr x))))
    (else x)))))
```

- with strict `map-closure`, analogous to stop-and-copy GC
- with lazy `map-closure`, analogous to incremental copy-on-read GC
- all applications we have found use similar recursive idioms

Using this Idiom to Implement `with-complex`

Invoke `thunk` where each real *x* that is reachable during the invocation is lifted to a pair `(`*x*` . 0)` and each copy of the addition procedure that is reachable during the invocation is lifted to complex addition.

```
(define (with-complex thunk)
 ((map-closure*
   ;; per-slot
   (lambda (n x) x)
   ;; per-object
   (lambda (x)
    (cond ((real? x) (cons x 0))
          ((eq? x +)
           (lambda (x y)
            (cons (+ (car x) (car y))
                  (+ (cdr x) (cdr y)))))
          (else x))))
   thunk)))
```

Using this Idiom to Implement a Sandbox

Check every procedure invocation during the invocation of `thunk` and `raise-an-exception` if that procedure invocation is not `allowed?`.

```
(define (sandbox allowed? raise-an-exception thunk)
 ((map-closure*
   ;; per-slot
   (lambda (n x) x)
   ;; per-object
   (lambda (x)
    (if (procedure? x)
        (lambda arguments
         (if (allowed? x arguments)
             (apply x arguments)
             (raise-an-exception)))
        x))
   thunk)))
```

Analogous mechanisms can perform tracing, logging, profiling, code instrumentation and metering, error checking, and virtualization.

Using this Idiom to Implement Variable Mutation

Invoke an altered continuation where the value of all instances of a slot `n` are replaced with `new`.

```
(define (set n new)
 ((call/cc
   (lambda (c)
    (map-closure*
     ;; per-slot
     (lambda (n1 old) (if (name=? n n1) new old))
     ;; per-object
     (lambda (x) x)
     c)))
  #f))

(define-syntax set!
 (syntax-rules () ((set! n new) (set (name n) new))))
```

- evaluator must rename arguments to procedures as they are invoked
- to handle circularity, `map-closure*` must recurse *after* calling `per-slot` and `map-closure` must be lazy
- illustrates power of `map-closure`; not intended as a practical implementation technique

Implementation

- via closure conversion (analogous to `call/cc` via CPS conversion)
- native implementation (using existing mechanisms for accessing variables and creating closures)
- prototypes of both available at `http://www.bcl.hamilton.ie/~qobi/map-closure/`

Issues

- integration with type systems
- not referentially transparent
- name-based (`per-slot`) vs. value-based (`per-object`)
  - value-based relies on ability to compare procedures for equality
  - cannot compare procedures for equality in ML or HASKELL
  - in SCHEME, can only compare procedures for equality with `eq?`, not `equal?`
  - but `map-closure` breaks `eq?` by copying (can hashcons)
- name-based uses lexical scoping to control reflection boundaries
  - how do you specify reflection boundaries in value-based?
  - how do you perform and control communication across reflection strata?
  - trace write in trace f in . . .
    does the outer trace write trace just the writes in f or also the writes in the tracing of f?
  - security holes: program can determine whether it is running in a sandbox

Take Home Message

There is a crying need for a construct that performs nonstandard interpretation which is:

- easy to use
- dynamic and first-class
- powerful and flexible
- efficient

The `map-closure` construct is an attempt to fill this gap.

Take Home Message

There is a crying need for a construct that performs nonstandard interpretation which is:

- easy to use
- dynamic and first-class
- powerful and flexible
- efficient

The `map-closure` construct is an attempt to fill this gap.

Q: Can `map-closure` be implemented efficiently?

Take Home Message

There is a crying need for a construct that performs nonstandard interpretation which is:

- easy to use
- dynamic and first-class
- powerful and flexible
- efficient

The `map-closure` construct is an attempt to fill this gap.

Q: Can `map-closure` be implemented efficiently?
A: Yes, with a sufficiently smart compiler. (We are building one.)

Take Home Message

There is a crying need for a construct that performs nonstandard interpretation which is:

- easy to use
- dynamic and first-class
- powerful and flexible
- efficient

The `map-closure` construct is an attempt to fill this gap.

Q: Can `map-closure` be implemented efficiently?
A: Yes, with a sufficiently smart compiler. (We are building one.)

Q: What if I hate `map-closure`?

Take Home Message

There is a crying need for a construct that performs nonstandard interpretation which is:

- easy to use
- dynamic and first-class
- powerful and flexible
- efficient

The `map-closure` construct is an attempt to fill this gap.

Q: Can `map-closure` be implemented efficiently?
A: Yes, with a sufficiently smart compiler. (We are building one.)

Q: What if I hate `map-closure`?
A: Propose your own first-class dynamic NSI construct.

Contingency Slides

Desiderata

Want to evaluate `(+ (* a x) b)` under an NSI.

- Confining NSI to limited context
- without need to transform whole program
- Composing NSIs with specified order
- Multiple NSIs
- Reinterpreting constants
- Reinterpreting closed-over variables

Global Nonstandard Interpretation by Mutating Top Level

- Confining NSI to limited context

```
(define (vector-nsi)
 (let ((v+ (vector+ + *)) (v* (vector* + *)))
  (set! + v+)
  (set! * v*)))

(define (under nsi code) (nsi) (code))

(under vector-nsi (lambda () (+ (* a x) b)))
```

- without need to transform whole program

```
(define (f x y) (+ x y))

(under vector-nsi (lambda () (f (* a x) b)))
```

- Composing NSIs with specified order

```
(define (matrix-nsi)
 (let ((m+ (matrix+ + *)) (m* (matrix* + *)))
  (set! + m+)
  (set! * m*)))

(define (compose nsi1 nsi2) (lambda () (nsi2) (nsi1)))

(under (compose vector-nsi matrix-nsi)
       (lambda () (+ (* a x) b)))

(under (compose matrix-nsi vector-nsi)
       (lambda () (+ (* a x) b)))
```

Global Nonstandard Interpretation by Mutating Top Level

- Multiple NSIs

```
(list (under vector-nsi (lambda () (+ (* a x) b)))
      (under matrix-nsi (lambda () (+ (* a x) b))))
```

- Reinterpreting constants

```
(define (vector-nsi)
 (let ((v+ (vector+ + * 0))
       (v* (vector* + * 0))
       (v0 (vector0 + * 0)))
  (set! + v+)
  (set! * v*)
  (set! 0 v0)))

(under vector-nsi (lambda () (+ (* a x) 0)))
```

- Reinterpreting closed-over variables

```
(define g (let ((p +) (c 0)) (lambda (a x) (p (* a x) c))))

(under vector-nsi (lambda () (g a x)))
```

Lexical Nonstandard Interpretation by Abstraction

- Confining NSI to limited context

```
(define (vector-nsi . env)
 (list (apply vector+ env) (apply vector* env)))

(define (under env code) (apply code env))

(under (vector-nsi + *) (lambda (+ *) (+ (* a x) b)))
```

- without need to transform whole program

```
(under (vector-nsi + *)
       (lambda (+ *)
        (define (f x y) (+ x y))
        (f (* a x) b)))
```

- Composing NSIs with specified order

```
(define (matrix-nsi . env)
 (list (apply matrix+ env) (apply matrix* env)))

(define (compose nsi1 nsi2)
 (lambda env (apply nsi1 (apply nsi2 env))))

(under ((compose vector-nsi matrix-nsi) + *)
       (lambda (+ *) (+ (* a x) b)))

(under ((compose matrix-nsi vector-nsi) + *)
       (lambda (+ *) (+ (* a x) b)))
```

Lexical Nonstandard Interpretation by Abstraction

- Multiple NSIs

```
(list (under (vector-nsi + *) (lambda (+ *) (+ (* a x) b)))
      (under (matrix-nsi + *) (lambda (+ *) (+ (* a x) b))))
```

- Reinterpreting constants

```
(define (vector-nsi . env)
 (list (apply vector+ env)
       (apply vector* env)
       (apply vector0 env)))

(under (vector-nsi + * 0) (lambda (+ * 0) (+ (* a x) 0)))
```

- Reinterpreting closed-over variables

```
(define g (let ((p +) (c 0)) (lambda (a x) (p (* a x) c))))

(under (vector-nsi + * 0) (lambda (+ * 0) (g a x)))
```

Nonstandard Interpretation with Dynamic Scoping

- Confining NSI to limited context

```
(define (vector-nsi code)
 (lambda ()
  (fluid-let ((+ (vector+ + *)) (* (vector* + *))) (code))))

(define (under nsi code) ((nsi code)))

(under vector-nsi (lambda () (+ (* a x) b)))
```

- without need to transform whole program

```
(define (f x y) (+ x y))

(under vector-nsi (lambda () (f (* a x) b)))
```

- Composing NSIs with specified order

```
(define (matrix-nsi code)
 (lambda ()
  (fluid-let ((+ (matrix+ + *)) (* (matrix* + *))) (code))))

(define (compose nsi1 nsi2) (lambda (code) (nsi2 (nsi1 code))))

(under (compose vector-nsi matrix-nsi)
       (lambda () (+ (* a x) b)))

(under (compose matrix-nsi vector-nsi)
       (lambda () (+ (* a x) b)))
```

Nonstandard Interpretation with Dynamic Scoping

- Multiple NSIs

```
(list (under vector-nsi (lambda () (+ (* a x) b)))
      (under matrix-nsi (lambda () (+ (* a x) b))))
```

- Reinterpreting constants

```
(define (vector-nsi code)
 (lambda ()
  (fluid-let ((+ (vector+ + * 0))
              (* (vector* + * 0))
              (0 (vector0 + * 0)))
   (code))))

(under vector-nsi (lambda () (+ (* a x) 0)))
```

- Reinterpreting closed-over variables

```
(define g (let ((p +) (c 0)) (lambda (a x) (p (* a x) c))))

(under vector-nsi (lambda () (g a x)))
```

Nonstandard Interpretation with map-closure

- Confining NSI to limited context

```
(define (vector-nsi code)
 (map-closure* (lambda (n x) x)
               (lambda (x)
                (cond ((equal? x 0) (vector+ + * 0))
                      ((eq? x +) (vector* + * 0))
                      ((eq? x *) (vector0 + * 0))
                      (else x)))
               code))

(define (under nsi code) ((nsi code)))

(under vector-nsi (lambda () (+ (* a x) b)))
```

- without need to transform whole program

```
(define (f x y) (+ x y))

(under vector-nsi (lambda () (f (* a x) b)))
```

Nonstandard Interpretation with `map-closure`

- Composing NSIs with specified order

```
(define (matrix-nsi code)
 (map-closure* (lambda (n x) x)
               (lambda (x)
                (cond ((equal? x 0) (matrix+ + * 0))
                      ((eq? x +) (matrix* + * 0))
                      ((eq? x *) (matrix0 + * 0))
                      (else x)))
              code))

(define (compose nsi1 nsi2) (lambda (code) (nsi2 (nsi1 code))))

(under (compose vector-nsi matrix-nsi)
       (lambda () (+ (* a x) b)))

(under (compose matrix-nsi vector-nsi)
       (lambda () (+ (* a x) b)))
```

- Multiple NSIs

```
(list (under vector-nsi (lambda () (+ (* a x) b)))
      (under matrix-nsi (lambda () (+ (* a x) b))))
```

- Reinterpreting constants

```
(under vector-nsi (lambda () (+ (* a x) 0)))
```

Nonstandard Interpretation with `map-closure`

- Reinterpreting closed-over variables

```
(define g (let ((p +) (c 0)) (lambda (a x) (p (* a x) c))))
(under vector-nsi (lambda () (g a x)))
```