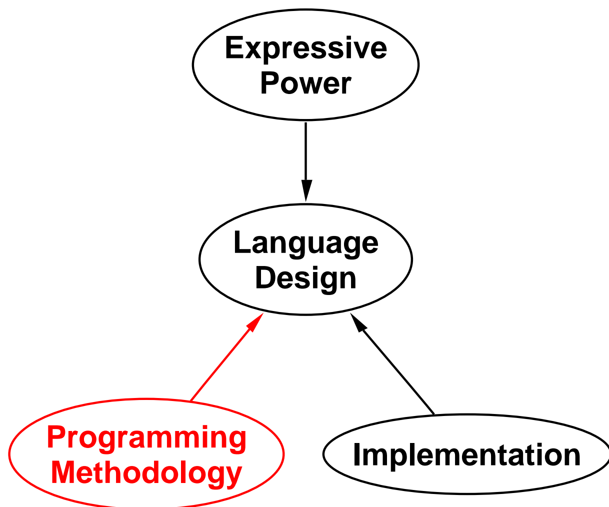
Programming Methodology (Software Engineering) and Language



1960s: (structured) control flow
1970s: data abstraction
1980s: reuse

CLU — Abstract Data Types

Abstract data types

- hide concrete representation (Parnas)
- expose only operations
- often specified with algebraic semantics

  looks like rewrite rules

When code depends on concrete representation, coupling forbids changes

- penalties: massive rewrites, massive recompilations

Methodology can be applied in any language

- greatest benefits come with language support

Static type discipline is essential to achieve reliability goals of ADTs

Can prove that ADT meets its spec.
Can prove that two ADTs are equivalent.

Data abstraction

Integers: concrete rep as 32-bit words.

```
if (n & 0x80000000)
  printf("n is negative\n");
else
  printf("n is positive\n");
```

What happens if I get 64-bit machine?
What happens if I use 16-bit machine?

Integers have predefined oprtions that do not require programmer to look at bits. C gives view of integers that is (somewhat) independent of concrete rep.

```
if (n < 0)
  printf("n is negative\n");
else
  printf("n is positive\n");
```


Data abstraction

Not

```
struct bintree {
  char *label;
  struct bintree *left_child;
  struct bintree *right_child;
}
```

Instead functions:

```
bool is_empty(bintree tree);
bool is_leaf(bintree tree);
bintree make_node(char* lab, bintree left, bintree
char *label_of(bintree tree);
bintree left_child_of(bintree tree);
bintree right_child_of(bintree tree);
void set_label_of(bintree tree, char *lab);
void set_left_child_of(bintree tree, bintree child)
void set_right_child_of(bintree tree, bintree child
```

Tree representation can change (balanced binary tree, array, string...)

Interface remains the same.

Data abstraction

Can't quite do this in C

Have to know that `bintree` represented by a `struct`

```
typedef struct bintree *bintree;

extern bool is_empty(bintree tree);
extern bool is_leaf(bintree tree);
...
```

C++ allows the abstraction (as a class with private data).
Where did C++ get it from?

- Simula-67 (class based language of late 60s.
- CLU (1970s experimental language).

CLU	clusters	ADTs hide data, export interface.
Ada	packages	
Modula	modules	

Java / C++ / Smalltalk: classes = ADTs + reusability

Data abstraction

Other data hiding:

SML:
- abstype keyword (deprecated)
- structures with opaque signatures(recommended)

Any higher-order functional language: use closures.
For example Scheme

```
(define (bintree)
  (define (dispatch m . args)
    (cond ((and (eq? m 'empty)
                (null? args))
           '())
          ((and (eq? m 'make_node)
                (= 3 (length args))
                (string? (car args)))
           args)
          ...
          ))
  dispatch)

((bintree) 'make_node "node1"
                      ((bintree) 'empty)
                      ((bintree) 'empty))
```

Works. Ugly. Luckily Scheme has macros.

Example — Environments

In interpreter, want not to depend on list representation

To specify operations, give signatures:

```
emptyEnv  : 'a env
bind      : name * 'a * 'a env -> 'a env
find      : name * 'a env -> 'a RAISES {NotFound}
```

Semantics may be informal or formal.

Good informal classification:

- constructors
  - creators (emptyEnv)
    create value of T, using values of other types
  - producers (bind)
    produce value of T, using existing T (plus ...)
- observers, include "selectors" (find)
  take a T, produce something else
- mutators
  change a value of T
  sensible only with *mutable state*

Equational Specification for environments

```
find(n, bind(n', v, rho)) =
  if n = n' then v else find(n, rho)

find(n, emptyEnv) = error: raises NotFound
```

Must specify compatible error behavior for all implemetations of ADT.

CLU — Clusters

Cluster introduces

- abstract type
- exported operations
- hidden representation

  can be basic types or other clusters
- private operations

No formal semantics in book.

(In current version, no signatures!
Will be fixed one day.
Book version of uCLU different from slides.
Book will migrate towards slides.

CLU — abstract and concrete

Implementation uses explicit abstraction functions,
(not Kamin's settor, selector, constructor functions)

- `up` takes rep into abstract domain
- `down` takes abstract value into rep

So, to construct:

construct rep in the ordinary way, apply `up`

to observe or mutate:

apply `down`, operate directly on rep
(may need to apply `up` to result)

Special declaration `cvt` automates most `down`, `up`

- `down` on entry, `up` on return

`up` and `down` are coercions change the type.

The representation type (`down`) only makes sense in cluster implementing the ADT.

Points on the plane—Cartesian coordinates

```
Point = cluster is new, abscissa, ordinate,
          reflect, rotate, compare
  rep = record [x:real, y:real]

  new = proc (x, y: real) returns (Point)
    return (up(rep${x:x, y:y})) end new

  abscissa = proc(p : Point) returns (real)
    return (down(p).x) end abscissa

  ordinate = proc(p : Point) returns (real)
    return (down(p).y) end ordinate

  reflect = proc(p : cvt)
    p.x := - p.x p.y := - p.y end reflect

  rotate = proc(p : cvt)
    tmp : real := p.x p.x := p.y p.y := - tmp
    end rotate

  sqrdist = proc(p : cvt) returns (real)
    return (p.x * p.x + p.y * p.y) end sqrdist

  compare = proc(p1, p2 : Point) returns (bool)
    return (sqrdist(p1) < sqrdist(p2)) end compare

end Point
```


Points on the plane — Equational Specification

```
abscissa(new(x, y)) = x
ordinate(new(x, y)) = y
reflect'(new(x, y)) = new(-x, -y)
rotate' (new(x, y)) = new(y, -x)
compare (new(x, y), new(x', y'))
```

$= x^2 + y^2 < x'^2 + y'^2$

...

Equational specification is purely functional

- what to do about mutators (side effects)?

Answer: two-level specifications

- language-independent or "shared" level

  functional, equational
- language-dependent or "interface" level

  handles imperative features: state, exceptions

Example

```
reflect = proc (p : cvt) ...
  ; modifies at most p
  ; ensures out p = reflect' (in p)
```

Points implementation — Polar coordinates

```
Point = cluster is new, abscissa, ordinate,
          reflect, rotate, compare
  pi = 3.1415926535
  rep = record [r:real, theta:real]

  new = proc (x, y: real) returns (Point)
    return (up(rep${r: sqrt(x * x + y * y),
                    theta: atan2(y, x)})) end new

  abscissa = proc(p : cvt) returns (real)
    return (p.r * cos(p.theta)) end abscissa

  ordinate = proc(p : cvt) returns (real)
    return (p.r * sin(p.theta)) end ordinate

  reflect = proc(p : cvt)
    p.theta := p.theta + pi end reflect

  rotate = proc(p : cvt)
    p.theta := p.theta + pi / 2.0 end rotate

  sqrdist = proc(p : cvt) returns (real)
    return (p.r * p.r) end sqrdist

  compare = proc(p1, p2 : Point) returns (bool)
    return (sqrdist(p1) < sqrdist(p2)) end compare
end Point
```


Implementing Abstract Data Types

Primary act is information hiding

- is almost all environments and scopes
- what name is visible where?
- some names introduced implicitly

Usability through syntactic sugar:

- `e1 + e2` ⟹ `T$add(e1, e2)`
- `e1 * e2` ⟹ `T$mul(e1, e2)`
- `-e` ⟹ `T$minus(e)`
- `a[e]` ⟹ `T$fetch(a, e)` where `T` is type of `a`
- `r.name` ⟹ `T$get_name(r)`

Example:

```
cluster bignum ...
  rep = { digits : array[int] , digitSize : int }

big.digits[4] ==>
    array[int]$fetch(bignum$get_digits(big), 4)
```

Also for assignment (`x` has type RT, `a` is array of int):

- `x.first := 6` ⟹ `RT$set_first(x, 6)`
- `a[23] := n` ⟹ `array[int]$store(a, 23, n)`

Full CLU: exceptions

Invented by the CLU team

- motivated by ADT methodology
  - `lookup(n, empty)` is never a value
  - picking special rep makes no sense

Classic paper (Liskov and Snyder 1979)

Too much terminology :-(

ML	CLU
exception	exception/signal
`raise`	`signal`
`e handle Exn => ...`	`s except when Exn ...`

Every procedure labelled with type of exceptions raised

major aid to reliability
(significant weakness in ML, but hard with H.O.F.)

Every exception must be caught by immediate caller

- leads to very efficient, portable implementation
- restriction relaxed in later languages
  (Ada, Modula-3, Standard ML)

Full CLU: mutability

Mutability:

- each ADT is mutable or immutable
  (support both *imperative* and *applicative* style)
- “small” values usually immutable
- “container” types (stack, tree, table, array) usually mutable

  mutation can be more efficient than copying
- many advantages to immutable strings

Full CLU: polymorphism

Polymorphic clusters combine two elements:

- Type parameters as in Typed μScheme
- Function parameters as in Typed μScheme

Type abstraction and application alone

```
Env = cluster [T: type] is ...
rho : Env [Value]
Gamma : Env [Ty]
```

Type abstraction combined with HO functions:

```
sort = proc [t : type] (a: array[t]; lo, hi: int)
          returns (array[t])
          where t has
            lt : proctype (t, t) returns (bool)
```

Is a bit like this Typed μScheme

```
(val sort (type-lambda ('t)
  (lambda (((function ('t 't) bool) lt))
     ... sort function ...)))
```

And has this type

$sort : \forall\tau.\{lt : \tau \times \tau \rightarrow \texttt{bool}\} \rightarrow (\tau\,\texttt{array} \times \texttt{int} \times \texttt{int}) \rightarrow \tau\,\texttt{array}$

(becomes type classes in Haskell).

Full CLU: reference semantics

All values are located in "objects,"

which are allocated on the heap

Variables are simply references to those objects

- only objects change (by mutation)
- assignment points variable to different object

Reference semantics: "everything is like a pointer in C"

```
p1 := Point$new(3.0, 4.0)
p2 := p1
Point$rotate(p1) % p2 is also changed
Point$abscissa(p2)
4.0
```

Equality under reference semantics

`equal`	objects can't be distinguished, even by mutation
`similar`	objects have isomorphic values
`copy`	create an object that is similar but not equal (useful only for mutable types)

Reference semantics

- can be confusing (loss of referential transparency)
- avoids need to define assignment, equality for ADTs

assignment & equality are always on pointers

Full CLU: Iterators

Iterators:

- like procedure, but can be executed repeatedly
- provides access to "contents" of "container class"
- enables `for (p = l->head; p; p=p->link)` ... without exposing representation

Example: inside cluster

```
elements = iter (l) yields T
             while (~null(l))
               yield(l.car)
               l := l.cdr
             end
```

outside cluster

```
for x : T in List$elements(l) do
```
code to fiddle with `x`

implementation treats *code* ... as a nested function!
passed as extra parameter to iterator

Digression: iterators in ML

Higher-order function `app` executes function for side effect:

```
fun app f (h::t) = (f h; app f t)
  | app f []     = ()
```

Can rephrase `app` to look like a CLU iterator:

```
val elements = fn yield =>
  let val rec loop =
        fn l => if null l then ()
                else ( yield (hd l)
                     ; loop (tl l)
                     )
  in  loop
  end
```

CLU

```
for x : T in List$elements(l) do
```
code to fiddle with `x`

becomes in ML

`elements (fn x =>` *code to fiddle with* `x)`

Designing with ADTs

More difficult than it looks

Basic data structures work well

- stacks, queues, tables (trees), priority queues (heaps)

More serious examples are very rare

- I/O streams in Modula-3

For discipline, a bit of formalism:

- representation invariant
  - heap invariant for arrays
  - order invariant for binary-search trees
  - color invariant for red-black trees
- abstraction function

  maps concrete representation to abstract domain

EVERY operation must maintain invariants.

- writing down data invariants pays off in any language

Proof of correctness must work on representation, map up to abstraction

- classic paper by Hoare (1972)

Assessment of CLU

Major, major contributions

- ADTs, representation hiding
- exceptions

Implementations never took off

- not a player commercially

Many successors worse off (Ada, C++)

Much to study and learn from if working in imperative family

ML example: queues

Ideal queue: sequence with fifo behavior.

queue $= \langle x_1, \ldots, x_n \rangle$

empty $= \langle \rangle$

insert $(x, \langle x_1, \ldots, x_n \rangle) = \langle x_1, \ldots, x_n, x \rangle$

remove $\langle x_1, \ldots, x_n \rangle = \langle x_1, \langle x_2, \ldots, x_n \rangle \rangle$

remove $= \langle \rangle$ is an error "Empty"

Algebra of queues:

remove(insert x_n,(insert x_{n-1},(... insert(x_1,empty) ...))) =
if n = 0 then error "Empty"
if n ≥ 1 then
$\langle$ x_1, insert x_n,(insert x_{n-1},(... insert(x_2,empty) ...))$\rangle$

ML example: queues

ML: old, deprecated

```
exception Empty
abstype queue = Q of int list
with
  val empty = Q nil
  fun insert (x, Q xs) = Q (x :: xs)
  fun remove (Q xs) =
    (let val (x :: xs') = rev xs
     in  (x, Q (rev xs'))
     end)
    handle Match => raise Empty
end
- type queue
  val empty = - : queue
  val insert = fn : int * queue -> queue
  val remove = fn : queue -> int * queue
```

Feature: always treat representation as datatype
fun foo (Q xs) ... use xs ... down xs
let val q = Q ys ... up ys

MisFeature: Can not associate `Empty` exception with abstraction.

Problem "nonce type" (compiler picks unique name for each implementation of ADT). Abstraction barrier for nonce types has some problems.

ML example: queues

```
signature QUEUE =
 sig
  type queue
  exception Empty
  val empty : queue
  val insert : int * queue -> queue
  val remove : queue -> int * queue
 end
structure QL :> QUEUE =
 struct
  type queue = int list;
  exception Empty
  fun insert (x, xs) = x :: xs
  fun remove xs =
    (let val (x :: xs') = rev xs
     in  (x, rev xs')
     end)
    handle Match => raise Empty
  ... more stuff not visible to client ...
 end
```

"`:>`" opaque signature matching, provides abstraction barrier.

"`:`" alternative: translucent signature matching does not provide abstraction barrier. See Module systems lecture (later in semester).

Correctness of ADT w.r.t ideal ADT

ideal (Abstract) queue is sequence $\langle x_1, \ldots, x_n \rangle$
with constant empty$_A = \langle\rangle$
functions insert$_A$ $(x, \langle x_1, \ldots, x_n \rangle) = \langle x_1, \ldots, x_n, x \rangle$

remove$_A$ $\langle x_1, \ldots, x_n \rangle = \langle x_1, \langle x_2, \ldots, x_n \rangle\rangle$

remove$_A = \langle\rangle$ is an error "Empty"

Often have representation invariant: an ideal array has indices only in in the right range.

A homomorphism "same shape" is a structure preserving function. Idea: homomorphism commutes with operations

For some concrete implementation of a queue, must have homomorphism

`abs : concrete queue` $\rightarrow$ `abstract queue`

abs(empty$_C$) = empty$_A$
I(q) $\Rightarrow$ abs(insert$_C$(x,q)) = insert$_A$(x,abs(q))
I(q) $\Rightarrow$ abs(remove$_C$(q)) = remove$_A$(abs(q))

Have no invariant so can ignore I(q).

Correctness of ADT w.r.t ideal ADT

Claim

abs([]) = $\langle\rangle$
abs([x$_n$,...,x$_1$]) = $\langle x_1, \ldots, x_n \rangle$

So prove it:

abs(empty$_C$) = abs([]) = $\langle\rangle$ = empty$_A$

abs(insert$_C$(x,[x_n,...,x_1])) = abs([x, x_n,...,x_1])
= $\langle x_1, \ldots, x_n, x \rangle$ = insert$_A$(x,$\langle x_1, \ldots, x_n \rangle$)
= insert$_A$(x, abs([x_n,...,x_1]))

abs(remove$_C$([x_n,...,x_1]) = $\langle x_1$, abs([x_n,...,x_2])$\rangle$
= $\langle x_1, \langle x_2, \ldots, x_n \rangle$ = remove$_A$(abs([x_n,...,x_1]))

The right way to code a queue

```
structure QFB :> QUEUE =
 struct
  type queue = int list * int list
  exception Empty
  fun insert (x, (bs,fs)) = (x :: bs, fs)
  fun remove (nil,nil) = raise Empty
    | remove (bs, nil) = remove (nil, rev bs)
    | remove (bs, f::fs) = (f, (bs,fs))
 end
```

Why right:

- less space than doubly linked list normally used in C.
- same average cost: cost of reversing bs $\leq$ cost of building back links in doubly linked list.
- Way cheaper than QL impelemtation, which has two reverses per remove.

insert(3,insert(2,insert(1,empty))) = ([3,2,1],[]) $\langle 1, 2, 3 \rangle$

remove([3,2,1],[]) = (1, ([], [2,3]) $\langle 2, 3 \rangle$

insert(4, ([], [2,3]) = ([4], [2,3]) $\langle 2, 3, 4 \rangle$

Q: what is abs for this version?

Proving two ADTs equivalent

Bisimulation "anything that one can do, the other can do".

Establish relation R between representation values. Show that

- R holds on constants
- if R holds before an operation it holds after that operation.

Example relation between QL and QFB: relation l = b @ rev(f)

- QL.empty: l = nil, QFB.empty: (b,f) = (nil, nil)
  nil = nil @ (rev nil)
- QL.insert(x, l) and QFB.insert(x, (b,f)). Assuming that l = b @ rev(f)
  x::l = x::(b @ rev(f)) = (x::b) @ rev(f) by rules for append.
- QL.remove(l) and QFB.remove(b,f). Assuming that l = b @ rev(f)
  by cases:

Proving two ADTs equivalent

- case l = l' @ [x] related to b @ rev(x::f')
  = b @ ((rev f') @ [x]) so l' = b @ rev(f')
  QL.remove(l' @ [x]) = (x, l') and QFB.remove(b,x::f')
  = (x, (b,f'))
- case l = l' @ x related to ((b'@[x]) @ []) ...
- case l = [] related to ([], []) ...

Bisimulation proof may be difficult: in case of data in representation being polymorphic, or including ADTs internally. We want to get bisimulations even when the hidden parts have different representations. (See Robert Harper "Programming Languages Theory and Practice".)

Types for ADTs: $\exists$

New type scheme $\exists\alpha.\sigma$

σ is the type of the cluster code where the type variable α replaces the representation type τ.

New value: pack τ with v as $\exists\alpha.\sigma$

pack up my value (cluster code) v, which has representation type τ exposing only the abstract type for my cluster $\exists\alpha.\sigma$

New expressions:

- pack τ with e as $\exists\alpha.\sigma$

  Pack up any expression, not just a value.

- open e_1 as β with $x:\sigma$ in e_2

  use expression e_1 which has type $\exists\beta.\sigma$ under the name x in e_2.

Variations: (1) Could drop the σ since type checking will have type of e_1.
(2) Could use a pattern rather than the variable x to keep from needing to use projections to get elements of cluster.

$\exists$ types example

Queue with

```
val empty : queue
val insert : int * queue -> queue
val remove : queue -> int * queue
```

Has type
$\exists q.\{\text{empty}: q, \text{insert}: \text{int}\times q \rightarrow q, \text{delete}: q \rightarrow \text{int}\times q\}$ Using ML notation for types in records.

open
    (pack int list with Queue Code as $\exists q.\{\text{empty}: q, \ldots\}$)
as q with Q: $\{\text{empty}: q, \ldots\}$
in
    (#insert Q) (1, (#empty Q))

Inserts 1 into the empty queue.

$\exists$ vs $\forall$:

- $\forall$ polymorphism: some data is not being used so its type could be anything.
- $\exists$ polymorphism: data has a fixed type, but since only accessed through interface the client does not need to know the type.

Type rules and semantics of pack, open

pack rule without up or down:

$$\frac{\Gamma \vdash e : \sigma[\tau/\alpha]}{\Gamma \vdash \text{pack } \tau \text{ with } e \text{ as } \exists\alpha.\sigma : \exists\alpha.\sigma} \quad (\text{pack}_{\exists 1})$$

where α does not appear free in Γ

pack rule with up and down:

$$\frac{\Gamma\{\text{up} : \tau \to \alpha, \text{down} : \alpha \to \tau\} \vdash e : \sigma[\tau/\alpha]}{\Gamma \vdash \text{pack } \tau \text{ with } e \text{ as } \exists\alpha.\sigma : \exists\alpha.\sigma} \quad (\text{pack}_{\exists 2})$$

open rule:

$$\frac{\Gamma \vdash e_1 : \exists\alpha.\sigma \quad \Gamma\{x : \sigma[\beta/\alpha]\} \vdash e_2 : \tau}{\Gamma \vdash \text{open } e_1 \text{ as } \beta \text{ with } x : \sigma \text{ in } e_2 : \tau} \quad (\text{open}_{\exists})$$

where β does not appear free in Γ (import restriction) and β does not appear free in τ (export restriction).

Semantics: (types are not abstract at runtime!)

$$\frac{\langle e_1, \rho\rangle \Downarrow \text{pack } \tau \text{ with } v_p \text{ as } \exists\alpha.\sigma \quad \langle e_2[\tau/\beta], \rho\{x \mapsto v_p\}\rangle \Downarrow v}{\langle \text{open } e_1 \text{ as } \beta \text{ with } x : \sigma \text{ in } e_2, \rho\rangle \Downarrow v} \quad (\text{ADT}_{\exists})$$

More about $\exists$ rules

Import restriction "where β does not appear free in Γ" is easy to satisfy by α-conversion of variables.

Export restriction "where β does not appear free in τ" is essential for sound semantics.

Without export restriction one could write

```
let val empty =  open QFB as 'q
                 with {empty,...} in empty
    val insert = open QL as 'q
                 with {insert,...} in insert
in
    insert(1,empty)
end
```

which would attempt to cons 1 (QL's implementation of insert) onto the pair ([],[]) (QF's implementation of empty)

Note that the export restriction means that all operations that need to pass abstract data around must occur within a single open.

This is too restrictive for most programming: In a system with multiple modules want to use data from ADTs between modules.

Nonce types

Alternative:

Have the compiler pick a unique constant (a nonce) ν for the type of each cluster. In a system with multiple modules, the same ν is associated with the cluster in each module.

TYPE RULES HERE

Problems:

(1) For semantics to be sound, representation type may not be polymorphic. (Fancier set of rules relaxes this).

(2) If a cluster contains a free variable that affects its behavior, we still have only one nonce type for the cluster, but different invocations of function defining the variable can result in different behavior of the cluster based on the value of the variable. This will not cause a program to crash – representation type is not changed – but can cause it to give unexpected results.

(3) The nonce type has no representation in the program, which may be undesirable

ML's (deprecated) abstype construct uses nonce types.

Other solutions...

Problem with existential types: could not globally express "abstract type of this cluster".

Problem with nonce types: program can use "abstract type exported by this cluster" but can not refer to it in source code.

Solution might be construct "the type of this cluster".

Odd situation: if we use this then types include references to code. Such types are called "dependent types"

Using dependent types for ADTs, (and module systems...) is ongoing research. Competition in the '90s for simpler explanation of ML module system, still ongoing

Other way of doing "the type of this cluster": get away from structured types, and just use types as names – standard in OO programming. (inherits from, implements). But OO never just passes representation, always code and representation together.

Other dependent types that are used in some experimental languages: distinguish the list of α of length 0 from list of α of length 1, etc. The 0 and 1 are expressions that occur inside types.

ADT wrapup (redux)

Data abstraction is good: use it.

Getting the right abstractions can be difficult (refactoring in OO?)

CLU examples: `up` and `down` for representation vs abstract type.

ML abstype example: Pattern-match for `down`, inject for `up`.

Can reason about correctness of an ADT with respect to ideal implementation.

Can reason about whether two implementations are really the same ADT.

Can extend ML-like type systems with types for ADTs – but end up with complex solutions.

Credits: Norman Ramsey: original CLU slides. Ramsey & Kamin draft book: ideal implemetations. Robert Harper draft book: bisimulation proofs of ADT equivalence. Robert Harper, John Reynolds, Benjamin Pierce: existential types. Mark Sheldon & Franklyn Turbak (in Turbak & Gifford draft book): nonce types.