CAE

Subclasses with CAE:

```
{class posn
  x y
  {mdist {+ {get this x} {get this y}}}
  {addDist {+ {dsend arg mdist 0}
              {dsend this mdist 0}}}}
{class posn3D
  x y z
  {mdist {+ {get this z}
            {ssend this posn mdist arg}}}
  {addDist {ssend this posn addDist arg}}}
{send {new posn3D 1 2 3} addDist {new posn 3 4}}
```

Programmer manually

- duplicates fields
- implements method inheritance

ICAE

ICAE adds ***implementation inheritance***:

```
{class posn extends object
  x y
  {mdist {+ {get this x} {get this y}}}
  {addDist {+ {send arg mdist 0}
              {send this mdist 0}}}}
{class posn3D extends posn
  z
  {mdist {+ {get this z}
            {super mdist arg}}}}
{send {new posn3D 1 2 3} addDist {new posn 3 4}}
```

We can compile ICAE programs to CAE

ICAE Grammar

```
<prog>  ::=  <decl>* <ICAE>
<decl>  ::=  {class <cid> <fid>* <meth>*}
<meth>  ::=  {<mid> <ICAE>}
<ICAE>  ::=  <num>
         |   {+ <ICAE> <ICAE>}
         |   {- <ICAE> <ICAE>}
         |   {if0 <ICAE> <ICAE> <ICAE>}
         |   arg
         |   this
         |   {new <cid> <ICAE>*}
         |   {get <ICAE> <fid>}
         |   {send <ICAE> <mid> <ICAE>}     NEW
         |   {super <mid> <ICAE>}           NEW
```

ICAE Datatypes

```
(define-type ICAE
  [inum (n : number)]
  [istr (s : string)]
  [iadd (lhs : ICAE)
        (rhs : ICAE)]
  [isub (lhs : ICAE)
        (rhs : ICAE)]
  [iif0 (test-expr : ICAE)
        (then-expr : ICAE)
        (else-expr : ICAE)]
  [iarg]
  [ithis]
  [inew (class : symbol)
        (args : (listof ICAE))]
  [iget (obj-expr : ICAE)
        (field-name : symbol)]
  [isend (obj-expr : ICAE)
         (method-name : symbol)
         (arg-expr : ICAE)]
  [isuper (method-name : symbol)
          (arg-expr : ICAE)])
```

ICAE Datatypes

```
(define-type IDecl
  [iclass (name : symbol)
          (super : symbol)
          (fields : (listof IField))
          (methods : (listof IMethod))])

(define-type IField
  [ifield (name : symbol)])

(define-type IMethod
  [imethod (name : symbol)
           (body-expr : ICAE)])
```

ICAE Interpreter

```
(define (iinterp idecls iexpr)
  (local [(define expr (compile-expr iexpr
                                     (iclass 'bad 'bad empty empty)))
          (define cdecls-not-flat
            (map compile-methods idecls))
          (define cdecls
            (map (lambda (cdecl)
                   (flatten-class cdecl idecls cdecls-not-flat))
                 cdecls-not-flat))]
    (interp expr  cdecls (numV 0) (numV 0))))
```

ICAE Compiler: Expressions

```
(define compile-expr : (ICAE IDecl -> CAE)
  (lambda (icae this-class)
    (local [(define (recur expr)
              (compile-expr expr this-class))]
      (type-case ICAE icae
        [inum (n) (num n)]
        [istr (s) (str s)]
        [iadd (r l) (add (recur l) (recur r))]
        [isub (r l) (sub (recur l) (recur r))]
        [iif0 (t th el)
              (if0 (recur t) (recur th) (recur el))]
        [iarg () (arg)]
        [ithis () (this)]
        [inew (class-name field-exprs)
              (new class-name (map recur field-exprs))]
        [iget (expr field-name)
              (get (recur expr) field-name)]
        ...))))
```

ICAE Compiler: Expressions

```
(define compile-expr : (ICAE IDecl -> CAE)
  (lambda (icae this-class)
    (local [(define (recur expr)
              (compile-expr expr this-class))]
      (type-case ICAE icae
        ...
        [isend (expr method-name arg-expr)
               (dsend (recur expr)
                      method-name
                      (recur arg-expr))]
        [isuper (method-name arg-expr)
                (type-case IDecl this-class
                  [iclass (name super-name fields method)
                          (ssend (this)
                                 super-name method-name
                                 (recur arg-expr))])]))))
```

ICAE Compiler: Methods

```
(define (compile-methods idecl)
  (type-case IDecl idecl
    [iclass (name super-name fields methods)
            (class name
              (map (lambda (f)
                     (type-case IField f
                       [ifield (name) (field name)]))
                   fields)
              (map (lambda (m)
                     (type-case IMethod m
                       [imethod (name body-expr)
                                (method name
                                        (compile-expr body-expr
                                                      idecl))]))

                   methods))]))
```

ICAE Compiler: Flatten Class

```
(define (flatten-class cdecl idecls cdecls)
  (type-case CDecl cdecl
    [class (name fields methods)
      (type-case IDecl (find-iclass name idecls)
        [iclass (name super-name ifields imethods)
                (type-case CDecl (if (equal? super-name 'object)
                                     (class 'object empty empty)
                                     (flatten-class
                                      (find-class super-name cdecls)
                                      idecls
                                      cdecls))
                  [class (super-name super-fields super-methods)
                    (class name
                      (add-fields super-fields fields)
                      (add/replace-methods super-methods
                                           methods))])])]))
```

ICAE Compiler: Flatten Fields

```
(define (add-fields super-fields fields)
  (append super-fields fields))
```

ICAE Compiler: Flatten Methods

```
(define (add/replace-methods methods new-methods)
  (cond
   [(empty? new-methods) methods]
   [else (add/replace-methods
          (add/replace-method methods (first new-methods))
          (rest new-methods))]))

(define (add/replace-method methods new-method)
  (cond
   [(empty? methods) (list new-method)]
   [else
    (if (equal? (method-name (first methods))
                (method-name new-method))
        (cons new-method (rest methods))
        (cons (first methods)
              (add/replace-method (rest methods)
                                  new-method)))]))
```

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {send {get this x} mdist 0}
              {send {get this y} mdist 0}}}}
10
```

No – the `x` and `y` fields are not objects

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this z}}}}
10
```

No – `posn` has no `z` field

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {send this get-y 0}}}}
10
```

No – **`posn`** has no **`get-y`** method

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> posn
           {+ {get this x} {get this y}}}}
10
```

No – result type for **`mdist`** does not match body type

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}}
10
```

Yes

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}}
{new posn 12}
```

No – wrong number of fields in **`new`**

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}}
{new posn 12 {new posn 1 2}}
```

No – wrong field type for first **new**

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}
  {clone : num -> posn
         {new posn {get this x} {get this y}}}}
{send {new posn 1 2} clone 0}
```

Yes

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}
  {clone : num -> posn
         {new posn {get this x} {get this y}}}}
{class posn3D extends posn
  z : num
  {mdist : num -> num
         {+ {get this z} {super mdist arg}}}}
{new posn3D 5 7 3}
```

Yes

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}
  {clone : num -> posn
         {new posn {get this x} {get this y}}}}
{class posn3D extends posn
  z : num
  {mdist : num -> posn
         {new posn 10 10}}}
{new posn3D 5 7 3}
```

No – override of **`mdist`** changes result type

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}
  {clone : num -> posn
         {new posn {get this x} {get this y}}}}
{class posn3D extends posn
  z : num
  {mdist : num -> num
         {+ {get this z} {super mdist arg}}}
  {clone : num -> posn3D
         {new posn3D
              {get this x}
              {get this y}
              {get this z}}}}
{new posn3D 5 7 3}
```

No – override of **`clone`** changes result type

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  x : num  y : num
  {mdist : num -> num
           {+ {get this x} {get this y}}}
  {clone : num -> posn
         {new posn {get this x} {get this y}}}}
{class posn3D extends posn
  z : num
  {mdist : num -> num
         {+ {get this z} {super mdist arg}}}
  {clone : num -> posn
         {new posn3D
              {get this x}
              {get this y}
              {get this z}}}}
{new posn3D 5 7 3}
```

Yes – which means that we need subtypes

Typechecking Programs with Classes

Is this program well-formed?

```
{class posn extends object
  ...}
{class posn3D extends posn
  ...}
{if0 ... {new posn 1 2} {new posn3D 5 7 3}}
```

Yes – and expression type is **`posn`**

TICAE Grammar

```
<prog>   ::=  <decl>* <ICAE>
<decl>   ::=  {class <cid> <field>* <meth>*}
<field>  ::=  <fid> : <TE>                          NEW
<meth>   ::=  {<mid> : <TE> -> <TE> <ICAE>}         NEW
<TE>     ::=  num                                   NEW
          |   <cid>                                 NEW
```

TICAE Datatypes

```
(define-type TDecl
  [tclass (name : symbol)
          (super-name : symbol)
          (fields : (listof TField))
          (methods : (listof TMethod))])

(define-type TField
  [tfield (name : symbol)
          (te : TE)])

(define-type TMethod
  [tmethod (name : symbol)
           (arg-te : TE)
           (result-te : TE)
           (body-expr : ICAE)])
```

TICAE Datatypes

```
(define-type TE
  [numTE]
  [strTE]
  [objTE (class-name : symbol)])

(define-type Type
  [numT]
  [strT]
  [objT (class-name : symbol)])
```

TICAE Type Checking

```
(define (typecheck tdecls expr)
  (begin
    (map (lambda (tdecl)
           (type-case TDecl tdecl
             [tclass (name super-name fields methods)
                     (map (lambda (m)
                            (begin
                              (typecheck-method m tdecl tdecls)
                              (check-override m tdecl tdecls)))
                          methods)]))
         tdecls)
    (typecheck-expr expr tdecls (numT)
                    (tclass 'bad 'bad empty empty))))
```

TICAE Type Checking: Methods

```
(define (typecheck-method method this-tdecl tdecls)
  (type-case TMethod method
    [tmethod (name arg-te result-te body-expr)
             (if (is-subtype?
                  (typecheck-expr body-expr tdecls
                                  (parse-type arg-te) this-tdecl)
                  (parse-type result-te)
                  tdecls)
                 (values)
                 (type-error body-expr
                             (to-string (parse-type result-te))))]))
```

TICAE Type Checking: Method Overrides

```
(define (check-override method this-tdecl tdecls)
  (local [(define super-name
            (tclass-super-name this-tdecl))
          (define super-method
            (try
             (find-method-in-tree (tmethod-name method)
                                  (find-tclass super-name tdecls)
                                  tdecls)
             (lambda () method)))]
    (if (and (equal? (tmethod-arg-te method)
                     (tmethod-arg-te super-method))
             (equal? (tmethod-result-te method)
                     (tmethod-result-te super-method)))
        (values)
        (error 'typecheck (string-append
                           "bad override of "
                           (to-string method-name))))))
```

TICAE Type Checker

```
(define typecheck-expr : (ICAE (listof TDecl) Type TDecl -> Type)
  (lambda (expr tdecls arg-type this-class)
    (local [(define (recur expr)
              (typecheck-expr expr tdecls arg-type this-class))]
      (type-case ICAE expr
        ...
        [inum (n) (numT)]
        [istr (s) (strT)]
        [iadd (l r)
              (type-case Type (recur l)
                [numT ()
                      (type-case Type (recur r)
                        [numT () (numT)]
                        [else (type-error r "num")])]
                [else (type-error l "num")])]
        [isub (l r) ...]
        [iarg () arg-type]
        ...))))
```

TICAE Type Checker

```
(define typecheck-expr : (ICAE (listof TDecl) Type TDecl -> Type)
  (lambda (expr tdecls arg-type this-class)
    (local [(define (recur expr)
              (typecheck-expr expr tdecls arg-type this-class))]
      (type-case ICAE expr
        ...
        [iif0 (test-expr then-expr else-expr)
              (local [(define test-type (recur test-expr))
                      (define then-type (recur then-expr))
                      (define else-type (recur else-expr))]
                (type-case Type (recur test-expr)
                  [numT ()
                        (cond
                         [(is-subtype? then-type else-type tdecls)
                          else-type]
                         [(is-subtype? else-type then-type tdecls)
                          then-type]
                         [else
                          (type-error else-expr
                                      (to-string then-type))])]
                  [else (type-error test-expr "num")]))]
        ...))))
```

TICAE Type Checker

```
(define typecheck-expr : (ICAE (listof TDecl) Type TDecl -> Type)
  (lambda (expr tdecls arg-type this-class)
    (local [(define (recur expr)
              (typecheck-expr expr tdecls arg-type this-class))]
      (type-case ICAE expr
        ...
        [ithis ()
               (type-case TDecl this-class
                 [tclass (name super-name fields methods)
                         (objT name)])]
        ...))))
```

TICAE Type Checker

```
(define typecheck-expr : (ICAE (listof TDecl) Type TDecl -> Type)
  (lambda (expr tdecls arg-type this-class)
    (local [(define (recur expr)
              (typecheck-expr expr tdecls arg-type this-class))]
      (type-case ICAE expr
        ...
        [inew (class-name exprs)
              (local [(define arg-types (map recur exprs))]
                (if (andmap2 (lambda (t1 t2)
                               (is-subtype? t1 t2 tdecls))
                             arg-types
                             (get-all-field-types class-name tdecls))
                    (objT class-name)
                    (type-error expr "field type mismatch")))]
        ...))))
```

TICAE Type Checker

```
(define typecheck-expr : (ICAE (listof TDecl) Type TDecl -> Type)
  (lambda (expr tdecls arg-type this-class)
    (local [(define (recur expr)
              (typecheck-expr expr tdecls arg-type this-class))]
      (type-case ICAE expr
        ...
        [iget (obj-expr field-name)
              (type-case Type (recur obj-expr)
                [objT (class-name)
                      (local [(define field
                                (find-field-in-tree
                                 field-name
                                 (find-tclass class-name tdecls)
                                 tdecls))]
                        (type-case TField field
                          [tfield (name te)
                                  (parse-type te)]))]
                [else (type-error obj-expr "object")])]
        ...))))
```

TICAE Type Checker

```
(define typecheck-expr : (ICAE (listof TDecl) Type TDecl -> Type)
  (lambda (expr tdecls arg-type this-class)
    (local [(define (recur expr)
              (typecheck-expr expr tdecls arg-type this-class))]
      (type-case ICAE expr
        ...
        [isend (obj-expr method-name arg-expr)
               (local [(define obj-type (recur obj-expr))
                       (define arg-type (recur arg-expr))]
                 (type-case Type obj-type
                   [objT (class-name)
                         (typecheck-send class-name method-name
                                         arg-expr arg-type tdecls)]
                   [else
                    (type-error obj-expr "object")]))]
        ...))))
```

TICAE Type Checker

```
(define typecheck-expr : (ICAE (listof TDecl) Type TDecl -> Type)
  (lambda (expr tdecls arg-type this-class)
    (local [(define (recur expr)
              (typecheck-expr expr tdecls arg-type this-class))]
      (type-case ICAE expr
        ...
        [isuper (method-name arg-expr)
                (local [(define arg-type (recur arg-expr))]
                  (typecheck-send (tclass-super-name this-class)
                                  method-name
                                  arg-expr arg-type tdecls))]
        ...))))
```

TICAE Type Checker: Sends

```
(define typecheck-send
  (lambda (class-name method-name arg-expr arg-type tdecls)
    (type-case TMethod (find-method-in-tree
                        method-name
                        (find-tclass class-name tdecls)
                        tdecls)
      [tmethod (name arg-te result-te body-expr)
               (if (is-subtype? arg-type (parse-type arg-te) tdecls)
                   (parse-type result-te)
                   (type-error arg-expr
                               (to-string (parse-type arg-te))))])))
```

TICAE Type Checker: Subtypes

```
(define (is-subclass? name1 name2 tdecls)
  (cond
   [(equal? name1 name2) true]
   [(equal? name1 'object) false]
   [else
    (type-case TDecl (find-tclass name1 tdecls)
      [tclass (name super-name fields methods)
              (is-subclass? super-name name2 tdecls)])]))

(define (is-subtype? t1 t2 tdecls)
  (type-case Type t1
    [objT (name1)
          (type-case Type t2
            [objT (name2)
                  (is-subclass? name1 name2 tdecls)]
            [else false])]
    [else (equal? t1 t2)]))
```

TICAE Type Checker: Tree Helpers

```
(define find-in-tree
  (lambda (find-in-list extract)
    (lambda (name tdecl tdecls)
      (local [(define items (extract tdecl))
              (define super-name
                (tclass-super-name tdecl))]
        (if (equal? super-name 'object)
            (find-in-list name items)
            (try (find-in-list name items)
                 (lambda ()
                   ((find-in-tree find-in-list extract)
                    name
                    (find-tclass super-name tdecls)
                    tdecls)))))))))

(define find-field-in-tree
  (find-in-tree find-tfield tclass-fields))

(define find-method-in-tree
  (find-in-tree find-tmethod tclass-methods))
```

TICAE Type Checker: Other Helpers

```
(define (parse-type te)
  (type-case TE te
    [numTE () (numT)]
    [strTE () (strT)]
    [objTE (name) (objT name)]))

(define (get-all-field-types class-name tdecls)
  (if (equal? class-name 'object)
      empty
      (type-case TDecl (find-tclass class-name tdecls)
        [tclass (name super-name fields methods)
                (append
                 (map (lambda (f) (parse-type (tfield-te f)))
                      fields)
                 (get-all-field-types super-name tdecls))])))

(define (andmap2 f l1 l2)
  (cond
   [(and (empty? l1) (empty? l2)) true]
   [(and (cons? l1) (cons? l2))
    (and (f (first l1) (first l2))
         (andmap2 f (rest l1) (rest l2)))]
   [else false]))
```

TICAE Interpreter

```
(define (tinterp tdecls expr)
  (iinterp (map strip-types tdecls)
           expr))

(define (strip-types tdecl)
  (type-case TDecl tdecl
    [tclass (name super-name fields methods)
            (iclass
             name
             super-name
             (map (lambda (f)
                    (type-case TField f
                      [tfield (name te) (ifield name)]))
                  fields)
             (map (lambda (m)
                    (type-case TMethod m
                      [tmethod (name arg-te result-te body-expr)
                               (imethod name body-expr)]))
                  methods))]))
```