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 posn 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

```
type icae =
    INum of int
  | IAdd of icae * icae
  | ISub of icae * icae
  | IIfZ of icae * icae * icae
  | IArg
  | IThis
  | INew of string * icae list
  | IGet of icae * string
  | ISend of icae * string * icae
  | ISuper of string * icae

and idecl =
    IClass of string * string * ifield list * imeth list

and ifield = IField of string
and imeth = IMethod of string * icae
```

ICAE Interpreter

```
let iinterp = function
    (idecls, expr)  ->
      let expr
          = compileExpr(expr, IClass("bad", "bad", [], []))
      in let cdeclsNotFlat
          = (List.map (fun sdecl -> compileMethods(sdecl))
                idecls)
      in let cdecls
          = (List.map (fun cdecl -> flattenClass(cdecl,
                                                 idecls,
                                                 cdeclsNotFlat))
                cdeclsNotFlat)
      in interp(expr, cdecls, NumV(0), NumV(0))
```

ICAE Compiler: Expressions

```
let rec compileExpr = function
    (expr, thisClass) ->
      let recur = fun expr -> compileExpr(expr, thisClass)
      in match expr with
        INum(n) -> Num(n)
      | IAdd(l, r) -> Add(recur l, recur r)
      | ISub(l, r) -> Sub(recur l, recur r)
      | IIfZ(tst, thn, els) -> IfZ(recur tst,
                                   recur thn,
                                   recur els)
      | IArg -> Arg
      | IThis -> This
      | INew(s, exprs) -> New(s, List.map recur exprs)
      | IGet(expr, fname) -> Get(recur expr, fname)
      | ISend(expr, mname, argExpr) ->
          DSend(recur expr, mname, recur argExpr)
      | ISuper(mname, expr) ->
          let IClass(_, sname, _, _) = thisClass
          in SSend(This, sname, mname, recur expr)
```

ICAE Compiler: Methods

```
let rec compileMethods = function
    sdecl ->
      let IClass(name, superName, fields, methods) = sdecl
      in Class(name,
               List.map
                 (fun (IField(fname)) -> Field(fname))
                 fields,
               List.map
                 (fun (IMethod(name, expr)) ->
                   Method(name,
                          compileExpr(expr,
                                      sdecl)))
                 methods)
```

ICAE Compiler: Flatten Class

```
let rec flattenClass = function
    (Class(name, fields, methods), idecls, cdecls) ->
      let IClass(_, superName, _, _) = findIClass name idecls
      in let Class(_, superFields, superMethods)
          = if (superName = "object")
          then Class("object", [], [])
          else flattenClass(findClass superName cdecls,
                            idecls, cdecls)
      in Class(name,
               addFields(superFields, fields),
               addReplaceMethods(superMethods, methods))
```

ICAE Compiler: Flatten Class, Fields

```
let addFields = function
    (superFields, fields) -> List.append superFields fields
```

ICAE Compiler: Flatten Class, Methods

```
let rec addReplaceMethods = function
    (methods, []) -> methods
  | (methods, meth::mrest) ->
      addReplaceMethods(addReplaceMethod(methods, meth),
                        mrest)

and addReplaceMethod = function
    ([], bmeth) -> [bmeth]
  | ((Method(aname, aexpr) as ameth)::arest,
     (Method(bname, bexpr) as bmeth))
    -> if (aname = bname)
    then bmeth::arest
    else ameth::(addReplaceMethod (arest, bmeth))
```

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 **`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 -> 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> : <tyexpr>                          NEW
<meth>    ::=  {<mid> : <tyexpr> -> <tyexpr> <ICAE>}     NEW
<tyexpr>  ::=  num                                       NEW
           |   <cid>                                     NEW
```

TICAE Datatypes

```
type tdecl =
    TClass of string * string * tfield list * tmeth list

and tfield = TField of string * te

and tmeth = TMethod of string * te * te * ciae

and te =
    NumTE
  | ObjTE of string

and ty =
    NumT
  | ObjT of string
```

TICAE Type Checking

```
let typecheck = function
    (tdecls, expr)  ->
      let _ = (List.map
                 (fun tdecl ->
                   let TClass(_, _, _, methods) = tdecl
                   in (List.map
                         (fun m -> (typecheckMethod(m, tdecl, tdecls);
                                    checkOverride(m , tdecl, tdecls)))
                         methods))
                 tdecls)
      in typecheckExpr(expr, tdecls, NumT,
                       TClass("bad", "bad", [], []))
```

TICAE Type Checking: Methods

```
let typecheckMethod = function
    (TMethod(_, argTE, resultTE, body), tdecl, tdecls) ->
      let bodyT = typecheckExpr(body, tdecls, parseType argTE, tdecl)
      in if (isSubType(bodyT, (parseType resultTE), tdecls))
      then ()
      else raise(NoType(body, "result type mismatch"))
```

TICAE Type Checking: Method Overrides

```
exception BadOverride of string

let checkOverride = function
    (TMethod(mname, argTE, resultTE, _), tdecl, tdecls) ->
      let TClass(_, sname, _, _) = tdecl
      in try let TMethod(_, sArgTE, sResultTE, _)
          = findMethodInTree(mname,
                             findTClass sname tdecls,
                             tdecls)
      in if ((sArgTE = argTE) & (sResultTE = resultTE))
      then ()
      else raise(BadOverride(mname))
      with (NoSuch _) -> ()
```

TICAE Type Checker

```
let rec typecheckExpr = function
    (expr, tdecls, argTy, thisClass) ->
      let recur = fun expr ->
                    typecheckExpr(expr, tdecls, argTy, thisClass)
      in match expr with
        INum(n) -> NumT
      | IAdd(l, r) ->
          (match (recur l, recur r) with
            (NumT, NumT) -> NumT
          | _ -> raise (NoType(expr, "not numbers")))
      | ISub(l, r) ->
          (match (recur l, recur r) with
            (NumT, NumT) -> NumT
          | _ -> raise (NoType(expr, "not numbers")))
      …
```

TICAE Type Checker

```
let rec typecheckExpr = function
    (expr, tdecls, argTy, thisClass) ->
      let recur = fun expr ->
                    typecheckExpr(expr, tdecls, argTy, thisClass)
      in match expr with
      | IIfZ(tst, thn, els) ->
          (match (recur tst) with
            NumT ->
              let thnty = recur thn
              and elsty = recur els
              in if (isSubType(thnty, elsty, tdecls))
              then elsty
              else if (isSubType(elsty, thnty, tdecls))
              then thnty
              else raise (NoType (expr, "branch mismatch"))
          | _ -> raise (NoType(expr, "test is non-number")))
```

TICAE Type Checker

```
let rec typecheckExpr = function
    (expr, tdecls, argTy, thisClass) ->
      let recur = fun expr ->
                    typecheckExpr(expr, tdecls, argTy, thisClass)
      in match expr with
      | IArg -> argTy
      | IThis ->
          let TClass(name, _, _, _) = thisClass
          in ObjT(name)
```

TICAE Type Checker

```
let rec typecheckExpr = function
    (expr, tdecls, argTy, thisClass) ->
      let recur = fun expr ->
                    typecheckExpr(expr, tdecls, argTy, thisClass)
      in match expr with
      | INew(cname, exprs) ->
          let argTys = List.map recur exprs
          and fieldTys = getAllFieldTypes(cname, tdecls)
          in if (andmap2 (fun t1 t2 -> isSubType(t1, t2, tdecls))
                    argTys fieldTys)
          then ObjT(cname)
          else raise (NoType(expr, "field type mismatch"))
```

TICAE Type Checker

```
let rec typecheckExpr = function
    (expr, tdecls, argTy, thisClass) ->
      let recur = fun expr ->
                    typecheckExpr(expr, tdecls, argTy, thisClass)
      in match expr with
      | IGet(expr, fname) ->
          (match (recur expr) with
            ObjT(cname) ->
              let TField(_, fieldTE)
                  = findFieldInTree(fname,
                                    findTClass cname tdecls,
                                    tdecls)
              in parseType(fieldTE)
          | _ -> raise (NoType(expr, "not an object for get")))
```

TICAE Type Checker

```
let rec typecheckExpr = function
    (expr, tdecls, argTy, thisClass) ->
      let recur = fun expr ->
                    typecheckExpr(expr, tdecls, argTy, thisClass)
      in match expr with
      …
      | ISend(expr, mname, argExpr) ->
          (match (recur expr) with
            ObjT(cname) ->
              typecheckSend(cname, mname, argExpr, tdecls, recur)
          | _ -> raise (NoType(expr, "not an object for send")))
      | ISuper(mname, argExpr) ->
          let TClass(_, sname, _, _) = thisClass
          in typecheckSend(sname, mname, argExpr, tdecls, recur)
```

TICAE Type Checker: Sends

```
and typecheckSend = function
    (cname, mname, argExpr, tdecls, recur) ->
      let TMethod(_, argTE, resultTE, _)
          = findMethodInTree(mname,
                             findTClass cname tdecls,
                             tdecls)
      in if (isSubType(recur argExpr,
                       parseType(argTE),
                       tdecls))
      then parseType(resultTE)
      else raise (NoType(argExpr, "arg type mismatch"))
```

TICAE Type Checker: Subtypes

```
let rec isSubClass = function
    (aname, bname, tdecls) ->
      if (aname = bname)
      then true
      else if (aname = "object")
      then false
      else let TClass(_, sname, _, _)
                = findTClass aname tdecls
      in isSubClass(sname, bname, tdecls)

let isSubType = function
    (ObjT(aname), ObjT(bname), tdecls) ->
      isSubClass(aname, bname, tdecls)
  | (NumT, NumT, _) -> true
  | _ -> false
```

TICAE Type Checker: Tree Helpers

```
let rec findInTree = fun
  findInList extract (name, tdecl, tdecls) ->
    let items = extract tdecl
    and TClass(_, sname, _, _) = tdecl
    in try (findInList name items)
    with exn -> if (sname == "object")
    then raise exn
    else (findInTree findInList extract
            (name, findTClass sname tdecls, tdecls))

let findFieldInTree = (findInTree findTField
                         (fun (TClass(_, _, fields, _)) -> fields))

let findMethodInTree = (findInTree findTMethod
                          (fun (TClass(_, _, _, methods)) -> methods))
```

TICAE Type Checker: Other Helpers

```
let parseType = function
    NumTE -> NumT
  | ObjTE(s) -> ObjT(s)

let rec getAllFieldTypes = function
    (cname, tdecls) ->
      if (cname = "object")
      then []
      else let TClass(_, sname, fields, _) = findTClass cname tdecls
      in (List.append
            (getAllFieldTypes (sname, tdecls))
            (List.map (fun (TField(_, te)) -> parseType te)
               fields))

let rec andmap2 = fun f l1 l2
    -> match (l1, l2) with
      (a::arest, b::brest) ->
        (f a b) & (andmap2 f arest brest)
    | ([], []) -> true
    | _ -> false

exception NoType of icae * string
```

TICAE Interpreter

```
let stripTypes = function
    TClass(name, sname, fields, methods) ->
      IClass(name, sname,
             (List.map (fun (TField(n,te)) -> IField(n))
                fields),
             (List.map (fun (TMethod(n,ate,rte,body)) ->
                          IMethod(n, body))
                methods))

let tinterp = function
    (tdecls, expr) ->
      iinterp(List.map stripTypes tdecls, expr)
```