micromodels of software

declarative modelling and analysis with Alloy

lecture 4: the Alloy language

Daniel Jackson
MIT Lab for Computer Science
Marktoberdorf, August 2002

the Alloy language

the Alloy language

syntactic constructs

- for packaging formulas
- for incremental description of structures

the Alloy language

syntactic constructs
- for packaging formulas
- for incremental description of structures

design principles
- remain first-order
- keep it simple (no partial functions, no subtypes)
- idiom-free (like PVS, ACL2)

look out for …

look out for …

key constructs

› signatures and functions

look out for …

key constructs

› signatures and functions

puns

› signatures like Java's classes
› dot like Java's dereferencing dot
› functions like PL functions

look out for …

key constructs

- signatures and functions

puns

- signatures like Java's classes
- dot like Java's dereferencing dot
- functions like PL functions

what's missing

- composites
- subtyping
- state machines

look out for …

key constructs

› signatures and functions

puns

› signatures like Java's classes
› dot like Java's dereferencing dot
› functions like PL functions

what's missing

› composites
› subtyping
› state machines

no notion of execution at all

gross structure

```
module Memory

sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}

fact {all m: Memory, a: Addr | sole m.data[a]}

fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d
  }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
     Write (m,m',a,d) => m.data[a] = d
  }

run Write
check WriteWorks
```

gross structure

- signature
  declares set & relations

```
module Memory

sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}

fact {all m: Memory, a: Addr | sole m.data[a]}

fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d
  }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
    Write (m,m',a,d) => m.data[a] = d
  }

run Write
check WriteWorks
```

gross structure

- signature
  declares set & relations
- fact
  global constraint

```
module Memory

sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}

fact {all m: Memory, a: Addr | sole m.data[a]}

fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d
  }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
    Write (m,m',a,d) => m.data[a] = d
  }

run Write
check WriteWorks
```

gross structure

- signature
  declares set & relations
- fact
  global constraint
- function
  parameterized constraint

```
module Memory

sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}

fact {all m: Memory, a: Addr | sole m.data[a]}

fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d
  }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
     Write (m,m',a,d) => m.data[a] = d
  }

run Write
check WriteWorks
```

gross structure

- signature
  - declares set & relations
- fact
  - global constraint
- function
  - parameterized constraint
- assertion
  - theorem to check

```
module Memory

sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}

fact {all m: Memory, a: Addr | sole m.data[a]}

fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d
  }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
     Write (m,m',a,d) => m.data[a] = d
  }

run Write
check WriteWorks
```

gross structure

- signature
  - declares set & relations
- fact
  - global constraint
- function
  - parameterized constraint
- assertion
  - theorem to check
- command
  - run function
  - check assertion

```
module Memory

sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}

fact {all m: Memory, a: Addr | sole m.data[a]}

fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d
  }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
     Write (m,m',a,d) => m.data[a] = d
  }

run Write
check WriteWorks
```

basic signature decls

basic signature decls

introduces set and basic type

sig s {} *set s with basic type (S, say)*

basic signature decls

introduces set and basic type

```
sig s {}                    set s with basic type (S, say)
```

set is immutable

› a global variable with one value

basic signature decls

introduces set and basic type

```
sig s {}                    set s with basic type (S, say)
```

set is immutable

- a global variable with one value

examples

```
sig Memory {}
sig Addr {}
sig Data {}
sig FileSystem {}
sig BirthdayBook {}
```

extension decls

extension decls

introduces subset of existing set

sig t **extends** s {} *t subset of s*

extension decls

introduces subset of existing set

sig t **extends** s {} *t subset of s*

subset is immutable

› a global variable with one value

extension decls

introduces subset of existing set

sig t **extends** s {}　　　*t subset of s*

subset is immutable

› a global variable with one value

examples

sig Cache **extends** Memory {}

sig Man **extends** Person {}

sig Dir **extends** FileSysObj {}

disjointness

disjointness

can mark subsigs as mutually disjoint

disj sig t1 **extends** s {}	*t1 and t2 disjoint …*
disj sig t2 **extends** s {}	
sig t3 **extends** s {}	*… but not nec t3*

disjointness

can mark subsigs as mutually disjoint

```
disj sig t1 extends s {}        t1 and t2 disjoint …
disj sig t2 extends s {}
sig t3 extends s {}             … but not nec t3
```

examples

```
sig Memory {}
disj sig Cache extends Memory {}
disj sig MainMemory extends Memory {}

sig Person {}
part sig Man, Woman extends Person {}
sig Employee extends Person {}
```

singletons

singletons

constrains set to be a singleton

static sig s {}   *s has one tuple*

singletons

constrains set to be a singleton

static sig s {}   *s has one tuple*

examples

static part sig Red, Green, Blue **extends** Colour {}

static sig Root **extends** Directory {}

fields

fields

introduces a relation

```
sig s {}
sig t {f: s}        f is relation of type (T,S)
                    with left set t and right set s
```

fields

introduces a relation

```
sig s {}
sig t {f: s}
```

f is relation of type (T,S)
with left set t and right set s

relation is immutable

› a global variable with one value

fields

introduces a relation

```
sig s {}
sig t {f: s}            f is relation of type (T,S)
                        with left set t and right set s
```

relation is immutable

- a global variable with one value

example

```
sig Date {}
sig Person {birthday: Date}
fact {some disj p,q: Person | p.birthday = q.birthday}
```

fields

introduces a relation

```
sig s {}
sig t {f: s}
```

f is relation of type (T,S) with left set t and right set s

relation is immutable

- a global variable with one value

example

```
sig Date {}
sig Person {birthday: Date}
fact {some disj p,q: Person | p.birthday = q.birthday}
```

birthday has type (PERSON,DATE)

overloading

overloading

resolved on left type

sig s {f: u}   *s$f has type (S,U)*

sig t {f: u}   *t$f has type (T,U)*

overloading

resolved on left type

sig s {f: u}   *s$f has type (S,U)*

sig t {f: u}   *t$f has type (T,U)*

example

sig Name {}
sig City {name: Name}
sig Person {name: Name}
fact {**no disj** p,q: Person | p.name = q.name}

overloading

resolved on left type

sig s {f: u}   *s$f has type (S,U)*

sig t {f: u}   *t$f has type (T,U)*

example

sig Name {}
sig City {name: Name}
sig Person {name: Name}
fact {**no disj** p,q: Person | p.name = q.name}

name
short for
Person$name

higher-arity fields

higher-arity fields

any expression can be used in field decl

```
sig s {f: e}
```

higher-arity fields

any expression can be used in field decl

```
sig s {f: e}
```

examples

```
sig Memory {data: Addr ->? Data}

sig FileSysObj {}
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files + dirs - root) ->! dirs
  }

sig BirthdayBook {records: Person ->? Date}
```

higher-arity fields

any expression can be used in field decl

```
sig s {f: e}
```

examples

```
sig Memory {data: Addr ->? Data}

sig FileSysObj {}
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files + dirs - root) ->! dirs
  }

sig BirthdayBook {records: Person ->? Date}
```

data has type (MEMORY,ADDR,DATA)

all m: Memory | m.data : Addr ->? Data

higher-arity fields

any expression can be used in field decl

```
sig s {f: e}
```

examples

```
sig Memory {data: Addr ->? Data}

sig FileSysObj {}
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files + dirs - root) ->! dirs
  }

sig BirthdayBook {records: Person ->? Date}
```

data has type (MEMORY,ADDR,DATA)

all m: Memory | m.data : Addr ->? Data

parent has type (FILESYSTEM,FILESYSOBJ,FILESYSOBJ)

all s: FileSystem | s.parent:(s.files+s.dirs-s.root)->! s.dirs

higher-arity fields

any expression can be used in field decl

```
sig s {f: e}
```

examples

```
sig Memory {data: Addr ->? Data}

sig FileSysObj {}
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files + dirs - root) ->! dirs
  }

sig BirthdayBook {records: Person ->? Date}
```

data has type (MEMORY,ADDR,DATA)

all m: Memory | m.data : Addr ->? Data

parent has type (FILESYSTEM,FILESYSOBJ,FILESYSOBJ)

all s: FileSystem | s.parent:(s.files+s.dirs-s.root)->! s.dirs

records has type (BIRTHDAYBOOK,PERSON,DATE)

all bb: BirthdayBook | bb.records: Person ->? Date

show me…

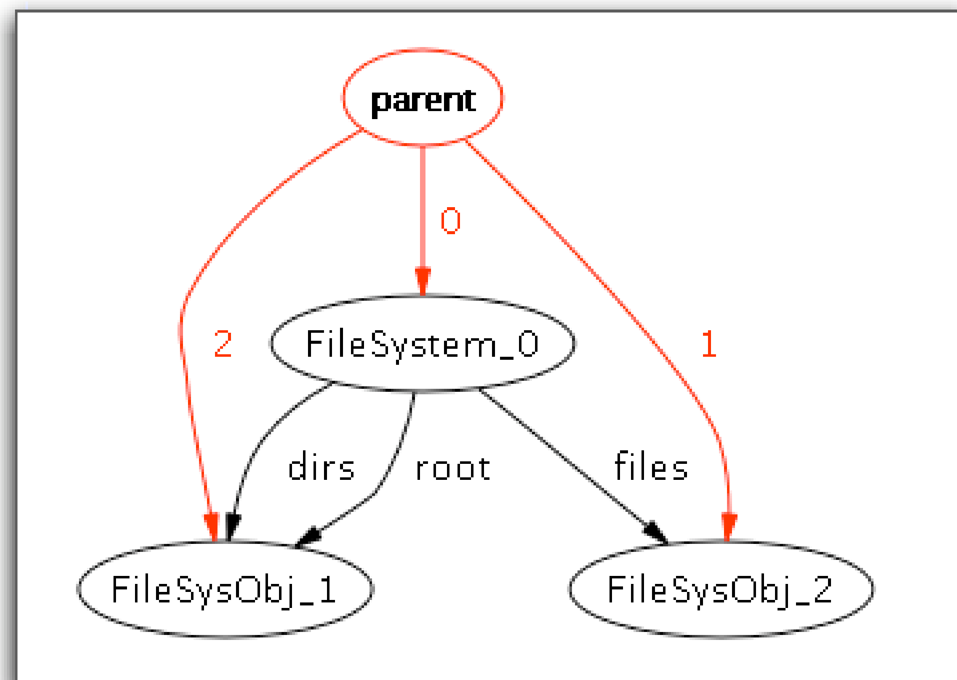
show me…

```
sig FileSysObj {}
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files+dirs-root) ->! dirs
  }

fun showMe () {
  some FileSystem
  }
run showMe
```

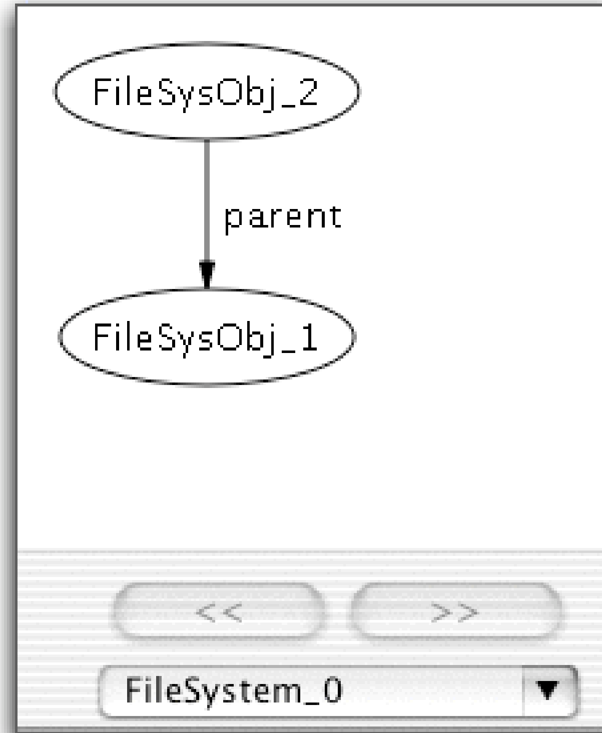
show me…

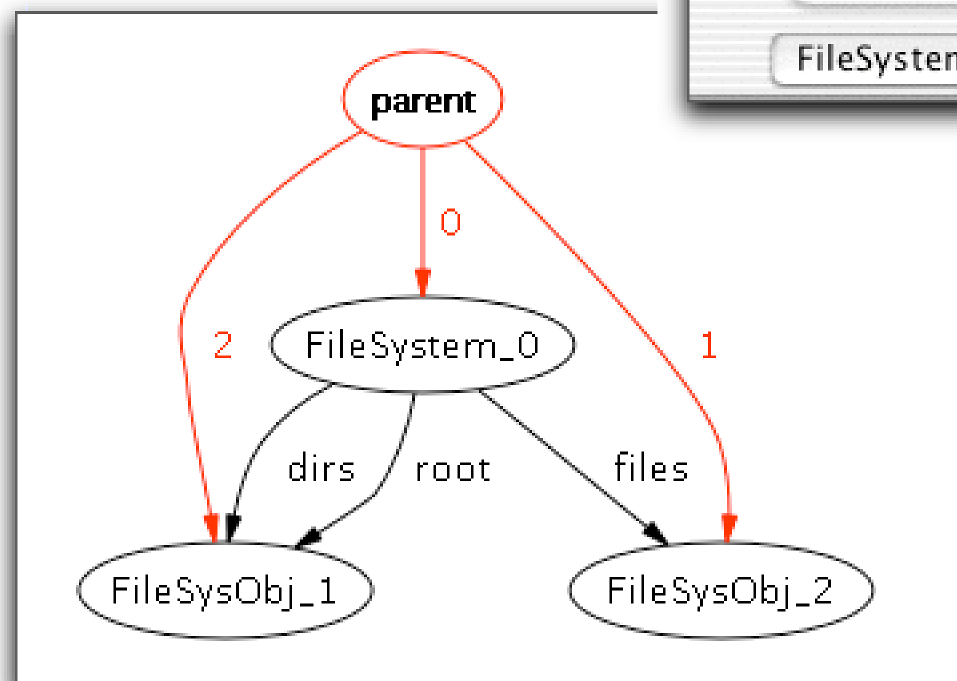
```
sig FileSysObj {}
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files+dirs-root) ->! dirs
  }
fun showMe () {
  some FileSystem
  }
run showMe
```



show me…

```
sig FileSysObj {}
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files+dirs-root) ->! dirs
  }
fun showMe () {
  some FileSystem
  }
run showMe
```





subsignature fields

subsignature fields

fields of subsignatures

- associated with basic type
- constrained to have subsig as left set

```
sig s {}
sig t extends s {f: u}        f has type (S,U) and left set t
```

subsignature fields

fields of subsignatures

- associated with basic type
- constrained to have subsig as left set

```
sig s {}
sig t extends s {f: u}        f has type (S,U) and left set t
```

example

```
sig Memory {data: Addr ->? Data}
sig Cache extends Memory {dirty: set Addr}
```

subsignature fields

fields of subsignatures

- associated with basic type
- constrained to have subsig as left set

sig s {}
sig t **extends** s {f: u}   *f has type (S,U) and left set t*

example

sig Memory {data: Addr ->? Data}
sig Cache **extends** Memory {dirty: **set** Addr}

dirty has type (MEMORY,ADDR)

dirty in Cache -> Addr

subsignature fields

fields of subsignatures

- associated with basic type
- constrained to have subsig as left set

sig s {}
sig t **extends** s {f: u} *f has type (S,U) and left set t*

example

sig Memory {data: Addr ->? Data}
sig Cache **extends** Memory {dirty: **set** Addr}

dirty has type (MEMORY,ADDR)

dirty in Cache -> Addr

out of domain application

if m is not in Memory, m.cache is **empty**

with

with

implicit deref

with e | F

is like F with each expression e′
whose left type matches the type of e
replaced by e′.e

with

implicit deref

```
with e | F
  is like F with each expression e′
  whose left type matches the type of e
  replaced by e′.e
```

example

```
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files+dirs-root) ->! dirs
  }
fact {all fs: FileSystem | with fs | files in root.*~parent
```

with

implicit deref

with e | F

is like F with each expression e′
whose left type matches the type of e
replaced by e′.e

example

```
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files+dirs-root) ->! dirs
  }
fact {all fs: FileSystem | with fs | files in root.*~parent
```

short for
fs.files in fs::root.*~fs::parent

signature facts

signature facts

signature fact has implicit *all* and *with*

```
sig s {...} {F}
  short for
sig s {...}
fact {all x: s | with x | F}
```

signature facts

signature fact has implicit *all* and *with*

```
sig s {...} {F}
  short for
sig s {...}
fact {all x: s | with x | F}
```

example

```
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files+dirs-root) ->! dirs
  } {files + dirs in root.*~parent}
```

signature facts

signature fact has implicit *all* and *with*

```
sig s {...} {F}
  short for
sig s {...}
fact {all x: s | with x | F}
```

example

```
sig FileSystem {
  disj files, dirs: set FileSysObj,
  root: dirs,
  parent: (files+dirs-root) ->! dirs
  } {files + dirs in root.*~parent}
```

short for
all this: FileSystem | **with this** |
files **in** root.*~parent

functions

functions

function

- a parameterized constraint
- ‘invocation’ is a formula
- meaning is just inlining/substitution
  (but recursion now supported)

functions

function

- a parameterized constraint
- 'invocation' is a formula
- meaning is just inlining/substitution (but recursion now supported)

example

```
fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
    Write (m,m',a,d) => m'.data[a] = d }
```

functions

function

- a parameterized constraint
- ‘invocation’ is a formula
- meaning is just inlining/substitution (but recursion now supported)

short for
m′.data = m.data ++ a->d

example

```
fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d }

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
    Write (m,m',a,d) => m'.data[a] = d }
```

checking assertions

checking assertions

form

> **check** *Assertion* [*scope*]

checking assertions

form

check *Assertion* [*scope*]

scope

for K **but** Number Type, …
for Number Type , …

checking assertions

form

check *Assertion* [*scope*]

scope

for K **but** Number Type, …
for Number Type , …

example

```
sig Memory {data: Addr ->? Data}

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
    Write (m,m',a,d) => m'.data[a] = d }

check WriteWorks for 3 but 2 Memory
```

checking assertions

form

check *Assertion* [*scope*]

scope

for K **but** Number Type, …
for Number Type , …

example

```
sig Memory {data: Addr ->? Data}

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
    Write (m,m',a,d) => m'.data[a] = d }

check WriteWorks for 3 but 2 Memory
```

find any value of
m, m′, a, d, data, Memory, Addr, Data
such that #Addr≤3 ,#Data≤3, ,#Memory≤2
and **Write (m,m',a,d) && m.data[a] != d**

checking assertions

form

check *Assertion [scope]*

scope

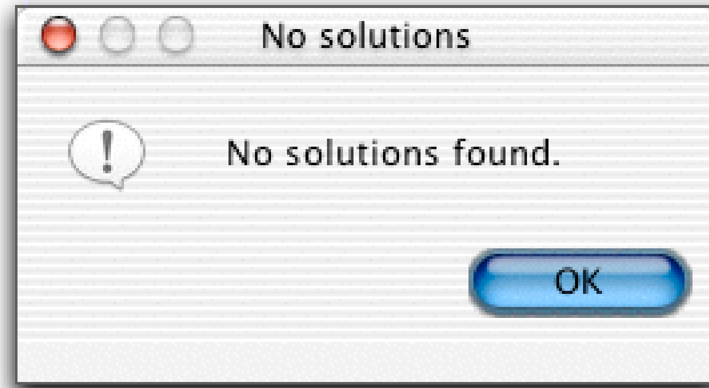
for K **but** Number Type, …
for Number Type , …

example

sig Memory {data: Addr ->? Data}

assert WriteWorks {
  all m,m': Memory, a: Addr, d: Data |
    Write (m,m',a,d) => m′.data[a] = d }

check WriteWorks **for** 3 **but** 2 Memory



find any value of
m, m′, a, d, data, Memory, Addr, Data
such that #Addr≤3 ,#Data≤3, ,#Memory≤2
and **Write (m,m',a,d) && m.data[a] != d**

running functions

running functions

form

run *Function* [*scope*]

running functions

form

```
run Function [ scope ]
```

example

```
fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d }
run Write for 2
```

running functions

form

run *Function* [*scope*]

example

```
fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d }
run Write for 2
```

find any value of
m, m′, a, d, data, Memory, Addr, Data
such that #Addr, #Data, #Memory≤2
and **Write (m,m',a,d)**

running functions

form

run *Function* [*scope*]

example

fun Write (m,m': Memory,a: Addr,d: Data) {
  m'.data = m.data ++ a->d }

run Write **for** 2

find any value of
m, m′, a, d, data, Memory, Addr, Data
such that #Addr, #Data, #Memory≤2
and **Write (m,m',a,d)**

simple memory model

simple memory model

```
module Memory
```

simple memory model

```
module Memory
sig Memory {data: Addr ->? Data}
```

simple memory model

```
module Memory
sig Memory {data: Addr ->? Data}
sig Addr {}
```

simple memory model

```
module Memory
sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}
```

simple memory model

```
module Memory
sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}
fun Write (m, m': Memory, a: Addr, d: Data) {
  m'.data = m.data ++ a->d
  }
```

simple memory model

```
module Memory
sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}
fun Write (m, m': Memory, a: Addr, d: Data) {
  m'.data = m.data ++ a->d
  }
fun Read (m: Memory, a: Addr, d: Data) {
  d = m.data[a]
  }
```

simple memory model

```
module Memory
sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}
fun Write (m, m': Memory, a: Addr, d: Data) {
  m'.data = m.data ++ a->d
  }
fun Read (m: Memory, a: Addr, d: Data) {
  d = m.data[a]
  }
assert ReadWrite {
  all m,m': Memory, a: Addr, d,d': Data |
     Write (m,m',a,d) && Read (m',a,d') => d = d’
  }
```

simple memory model

```
module Memory
sig Memory {data: Addr ->? Data}
sig Addr {}
sig Data {}
fun Write (m, m': Memory, a: Addr, d: Data) {
  m'.data = m.data ++ a->d
  }
fun Read (m: Memory, a: Addr, d: Data) {
  d = m.data[a]
  }
assert ReadWrite {
  all m,m': Memory, a: Addr, d,d': Data |
     Write (m,m',a,d) && Read (m',a,d') => d = d’
  }
check ReadWrite
```

cache model

cache model

```
sig Cache extends Memory {dirty: set Addr, main: Memory}
```

cache model

```
sig Cache extends Memory {dirty: set Addr, main: Memory}

fun CacheWrite (c, c': Cache, a: Addr, d: Data) {
  c'.dirty = c.dirty + a
  c'.main = c.main
  Write (c,c',a,d)
  }
```

cache model

```
sig Cache extends Memory {dirty: set Addr, main: Memory}

fun CacheWrite (c, c': Cache, a: Addr, d: Data) {
  c'.dirty = c.dirty + a
  c'.main = c.main
  Write (c,c',a,d)
  }

fun CacheRead (c, c': Cache, a: Addr, d: Data) {
  a in c.data.Data =>
     (Read (c,a,d) && c' = c),
     (Load (c,c',a) && Read (c',a,d))
  }
```

cache model

```
sig Cache extends Memory {dirty: set Addr, main: Memory}
fun CacheWrite (c, c': Cache, a: Addr, d: Data) {
  c'.dirty = c.dirty + a
  c'.main = c.main
  Write (c,c',a,d)
  }
fun CacheRead (c, c': Cache, a: Addr, d: Data) {
  a in c.data.Data =>
     (Read (c,a,d) && c' = c),
     (Load (c,c',a) && Read (c',a,d))
  }
fun Load (c, c': Cache, a: Addr) {
  some addrs: set Addr {
     c'.dirty = c.dirty - addrs
     c'.data = (c.data - addrs->Data) ++ a->c.main.data[a]
     c'.main.data = c.main.data ++ c.data & (addrs - c.dirty)->Data
     }
  }
```

even weaker drop rule

even weaker drop rule

web cache

```
sig Server {store: URL ->? Doc}
sig URL {}
sig Date {after: set Date}
sig Doc {expires: Date}

fun Request (u: URL, cache, cache', server: Server, now: Date) {
  cache'.store in (cache + server).store
  cache'.store[u] != server.store[u] =>
    cache.store[u].expires in now.after
  }
```

even weaker drop rule

web cache

```
sig Server {store: URL ->? Doc}
sig URL {}
sig Date {after: set Date}
sig Doc {expires: Date}

fun Request (u: URL, cache, cache', server: Server, now: Date) {
  cache'.store in (cache + server).store
  cache'.store[u] != server.store[u] =>
    cache.store[u].expires in now.after
  }
```

cache policy:
if a URL is not updated,
it can't have expired

sequencing checks

sequencing checks

```
assert CacheReadWrite {
  all c,c',c": Cache, a: Addr, d,d': Data |
    CacheWrite (c,c',a,d) && CacheRead (c',c",a,d') => d = d’
  }
```

sequencing checks

```
assert CacheReadWrite {
  all c,c',c": Cache, a: Addr, d,d': Data |
    CacheWrite (c,c',a,d) && CacheRead (c',c",a,d') => d = d’
  }
check CacheReadWrite
```

sequencing checks

```
assert CacheReadWrite {
  all c,c',c": Cache, a: Addr, d,d': Data |
    CacheWrite (c,c',a,d) && CacheRead (c',c",a,d') => d = d’
  }
check CacheReadWrite
assert CacheReadWrite2 {
  all c,c',c",c"": Cache, a,a': Addr, d,d',d": Data |
    {
    CacheWrite (c,c',a,d)
    CacheRead (c',c",a',d')
    CacheRead (c",c"",a,d")
    } => d = d”
  }
```

sequencing checks

```
assert CacheReadWrite {
  all c,c',c": Cache, a: Addr, d,d': Data |
    CacheWrite (c,c',a,d) && CacheRead (c',c",a,d') => d = d'
  }
check CacheReadWrite
assert CacheReadWrite2 {
  all c,c',c",c"": Cache, a,a': Addr, d,d',d": Data |
    {
    CacheWrite (c,c',a,d)
    CacheRead (c',c",a',d')
    CacheRead (c",c"",a,d")
    } => d = d"
  }
check CacheReadWrite2 for 3 but 4 Memory
```

a refinement check

a refinement check

```
fun Abstraction (c: Cache, m: Memory) {
  m.data = c.main.data ++ (c.data & c.dirty->Data)
  }
```

a refinement check

```
fun Abstraction (c: Cache, m: Memory) {
  m.data = c.main.data ++ (c.data & c.dirty->Data)
  }
fun Consistent (c: Cache) {
  c.data - c.dirty->Data in c.main.data
  }
```

a refinement check

```
fun Abstraction (c: Cache, m: Memory) {
  m.data = c.main.data ++ (c.data & c.dirty->Data)
  }
fun Consistent (c: Cache) {
  c.data - c.dirty->Data in c.main.data
  }
assert CacheReadOK {
  all c, c': Cache, m,m': Memory, a: Addr, d: Data |
    Consistent (c) && CacheRead (c,c',a,d)
    && Abstraction (c,m) && Abstraction (c',m') =>
      Read (m,a,d) && m.data = m'.data && Consistent (c')
  }
```

a refinement check

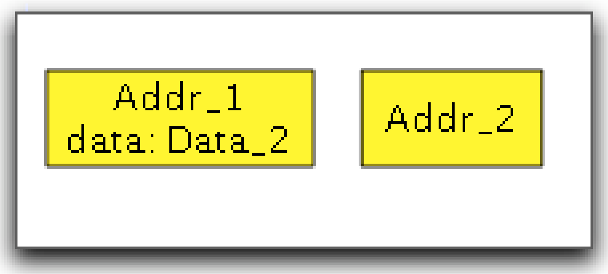
```
fun Abstraction (c: Cache, m: Memory) {
  m.data = c.main.data ++ (c.data & c.dirty->Data)
  }
fun Consistent (c: Cache) {
  c.data - c.dirty->Data in c.main.data
  }
assert CacheReadOK {
  all c, c': Cache, m,m': Memory, a: Addr, d: Data |
     Consistent (c) && CacheRead (c,c',a,d)
     && Abstraction (c,m) && Abstraction (c',m') =>
        Read (m,a,d) && m.data = m’.data && Consistent (c')
  }
check CacheWriteOK for 3 but 4 Memory
```

counterexample

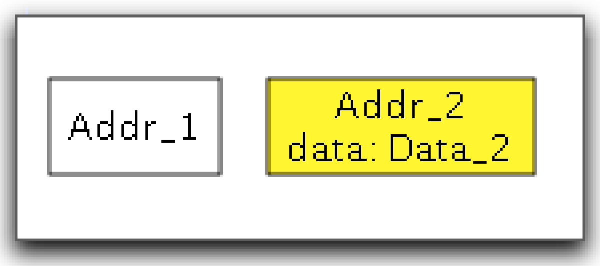
counterexample

a = Addr_2, d = Data_2

c

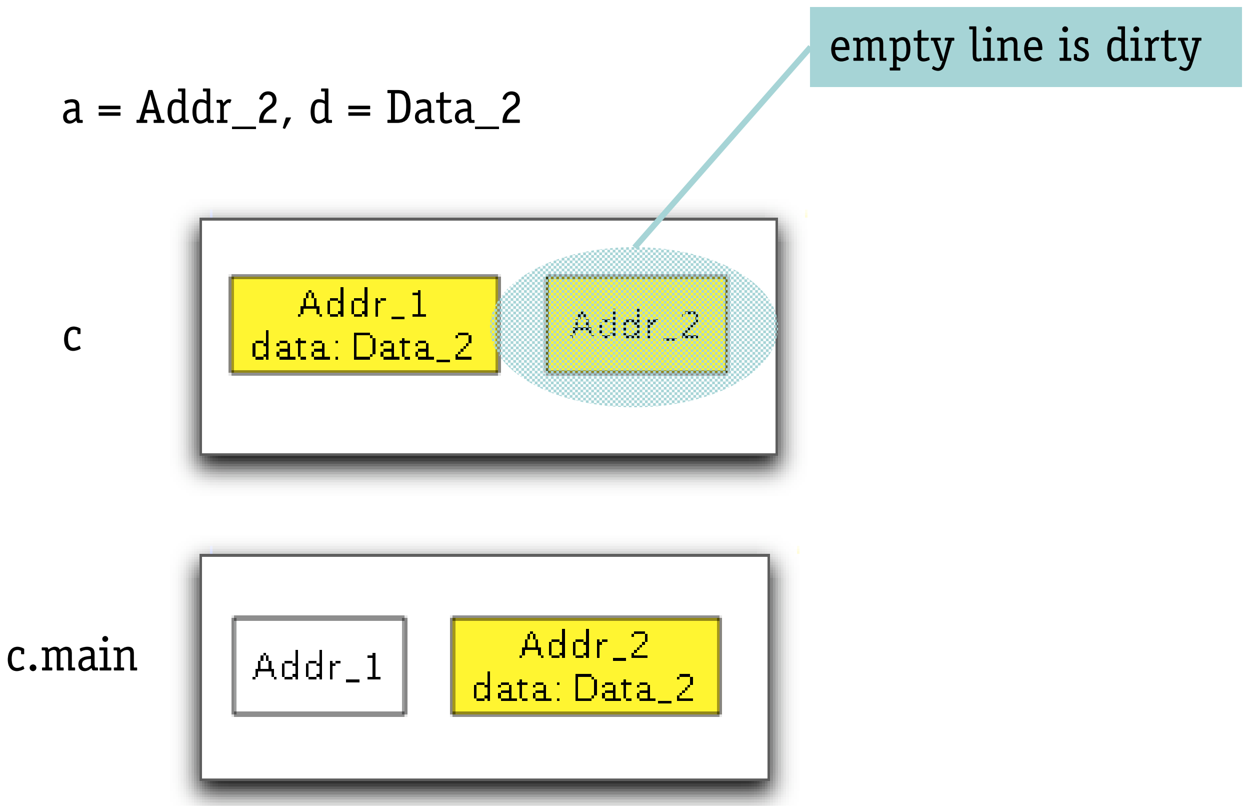


c.main



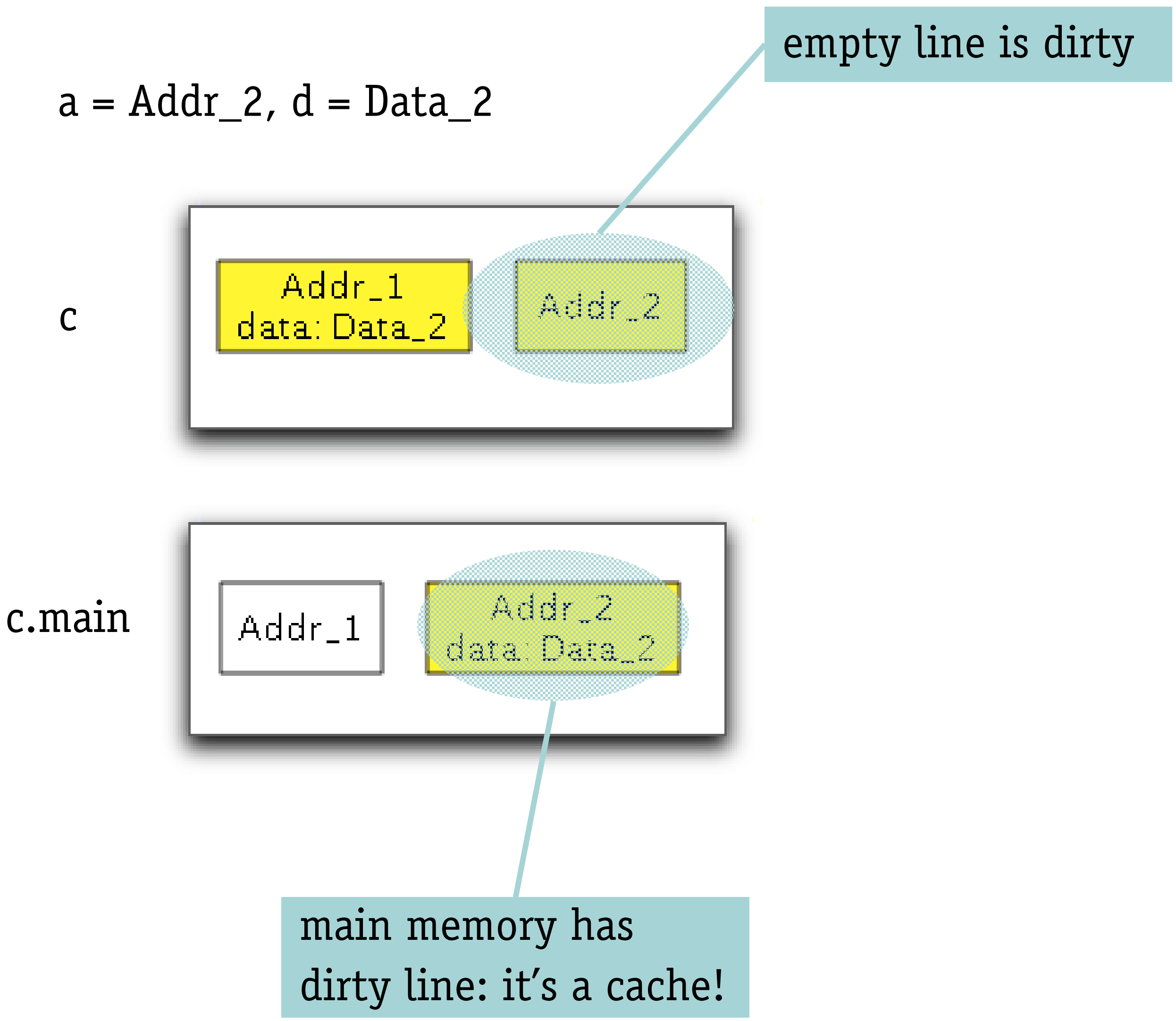
counterexample

a = Addr_2, d = Data_2



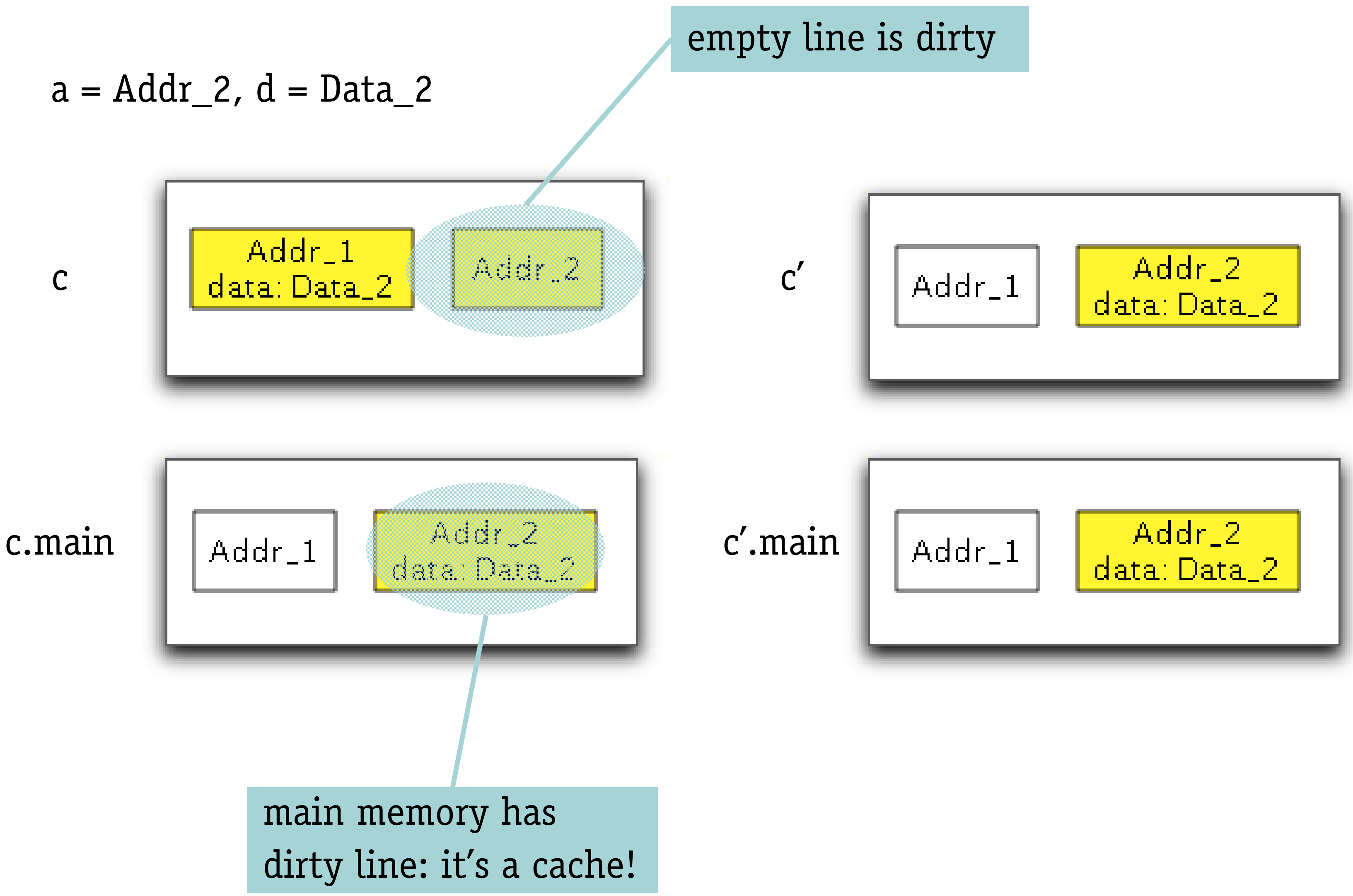
counterexample

a = Addr_2, d = Data_2



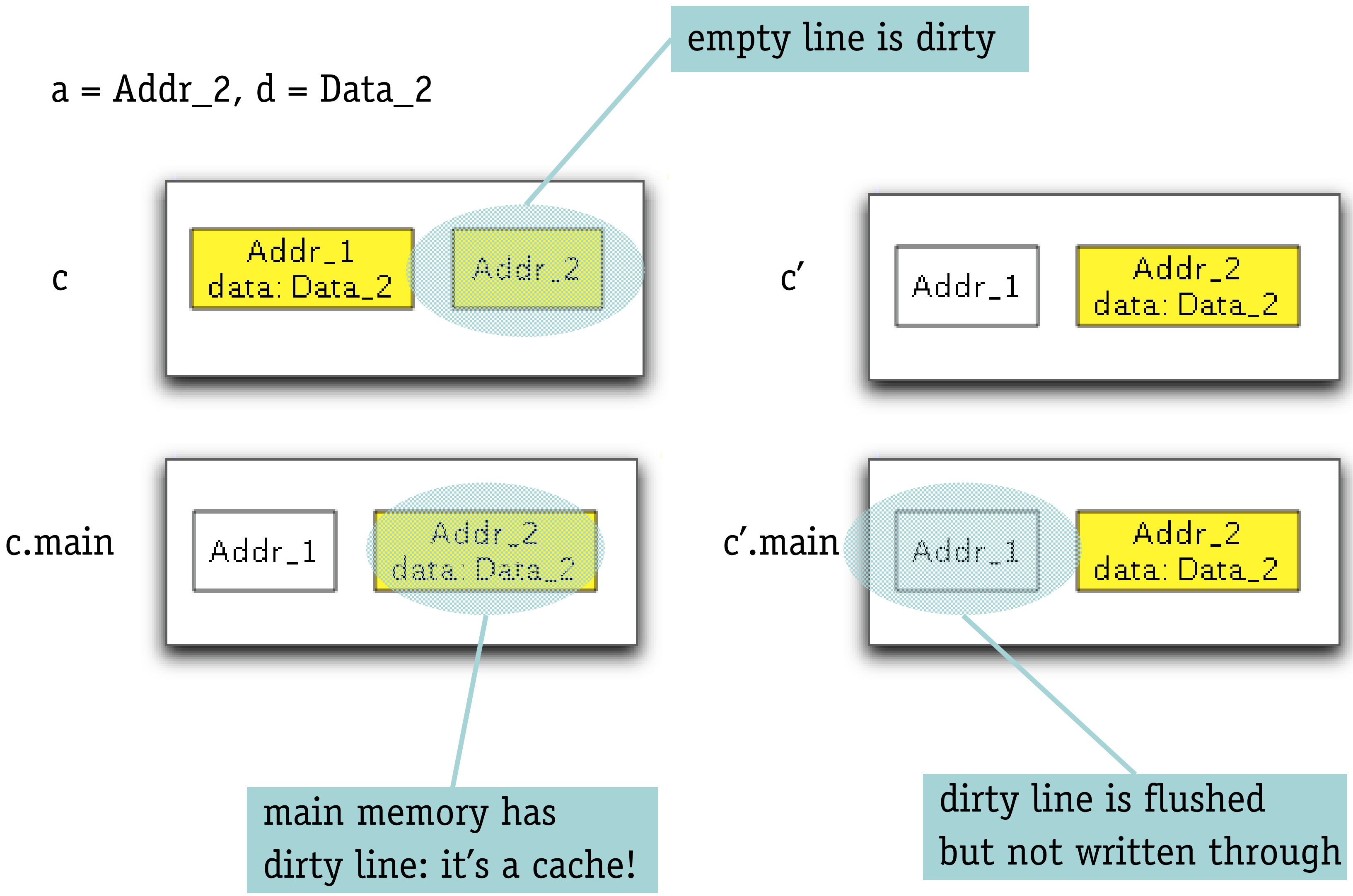
counterexample

a = Addr_2, d = Data_2



counterexample

a = Addr_2, d = Data_2



correction

correction

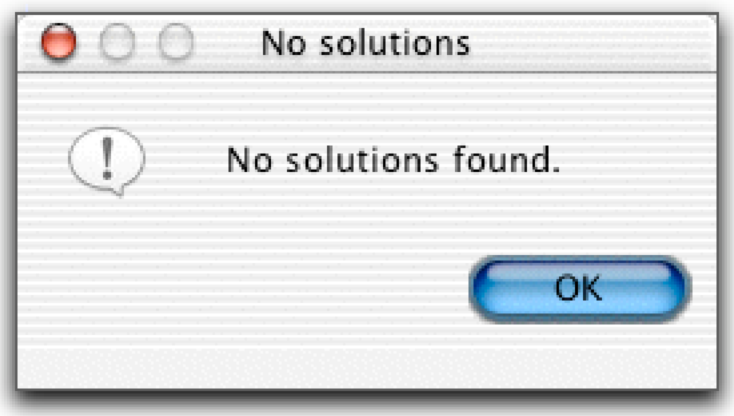
must write through dirty, not non-dirty lines:

```
fun Load (c, c': Cache, a: Addr) {
  some addrs: set Addr {
    c'.dirty = c.dirty - addrs
    c'.data = (c.data - addrs->Data) ++ a->c.main.data[a]
    c'.main.data = c.main.data ++ c.data & (addrs & c.dirty)->Data
    }
  }
```

correction

must write through dirty, not non-dirty lines:

```
fun Load (c, c': Cache, a: Addr) {
  some addrs: set Addr {
    c'.dirty = c.dirty - addrs
    c'.data = (c.data - addrs->Data) ++ a->c.main.data[a]
    c'.main.data = c.main.data ++ c.data & (addrs & c.dirty)->Data
    }
  }
```



other language features

other language features

function shorthands

```
fun f (a: A): B {... result...} == fun f (a: A, b: B) {...b...}
```

other language features

function shorthands

```
fun f (a: A): B {... result...} == fun f (a: A, b: B) {...b...}
```

polymorphism

```
module Lists
sig List [t] {}
sig NonEmptyList [t] extends List[t] {elt: t, next: List[t]}
fun elements [t] (l: List[t]): set t {result = l.*next.elt}
```

other language features

function shorthands

```
fun f (a: A): B {... result...} == fun f (a: A, b: B) {...b...}
```

polymorphism

```
module Lists
sig List [t] {}
sig NonEmptyList [t] extends List[t] {elt: t, next: List[t]}
fun elements [t] (l: List[t]): set t {result = l.*next.elt}
```

modules

```
open Lists
sig Op {}
sig Undo {ops: List[Op]}
```

doing more with less

doing more with less

no composites

› first order, so tractable

doing more with less

no composites

› first order, so tractable

no built-in idioms

› more flexible

doing more with less

no composites

- first order, so tractable

no built-in idioms

- more flexible

no separate property language -- allows masking

```
assert {System() => Property1()}
assert {System() && Property1() => Property2()}
```

doing more with less

no composites

- first order, so tractable

no built-in idioms

- more flexible

no separate property language -- allows masking

```
assert {System() => Property1()}
assert {System() && Property1() => Property2()}
```

no subtyping -- no casts

```
dir.contents.contents in File
dir.contents->forall{d | d.oclIsKindOf(Dir) implies
  d.oclAsType(Dir). contents in File}
```

challenges for you

challenges for you

solve Halmos's handshake problem using Alloy

Alice and Bob invite four couples for dinner.

When they arrive, they shake hands.

Nobody shakes their own or spouse's hand.

After some handshaking, Alice says "Stop shaking hands!", and then asks how many hands each person has shaken. All the answers are different.

How many hands has Bob shaken?

halmos’s handshaking

halmos's handshaking

```
sig Person {
  spouse: Person,
  shaken: set Person}{
    spouse != this
    no (this + spouse) & shaken }
fact {
  univ[Person] in Person
  spouse = ~spouse
  shakes = ~shakes }
fun Halmos () {
  some Alice: Person |
    all disj p1, p2: Person - Alice | # p1.shakes != # p2.shakes }
run Halmos for 10
```