- $[\![cd_1]\!]_{inv,mod}$ means "the encoding of cd_1 if its invariant of the current class verified is inv, its set of modifiable fields is mod and the access modifier of the current method is $accmod$".

- $[\![cd_1]\!]^{accmod,C}_{inv,mod}$ means "the encoding of cd_1 if the current class is C, its invariant is inv, its set of modifiable fields is mod and the access modifier of the current method is $accmod$".

- $\forall\ e\ S\ stmt$ indicates that we we repeat the statement $stmt$ for each element e of the set S. Not that a $\forall$ is encoded in Viper

- The fonts are used to differentiate between litteral text and identifiers. `assert` means that assert will be literally encoded while *fieldname* means that fieldname is an identifier or part of some formula.

- $[\![C]\!]$ where C is a class identifier encodes to `Int` and `Bool` for integers and boolean, it encodes to `Ref`

- by combining the last two rules we that: $\forall\ e\ fieldofC, [\![e.class]\!] ==$ `Ref` $stmt$ means that for each field e class C whose declared class encodes to `Ref` repeat the statement *stmt*.

- Blue are the changes.
- highlighted changes might be optional, e.g. Users could use invariant instead
- Hypotheses that need further thought are written in violet

conc ::= concealed:={fieldname$_1$,... fieldname$_k$}

(*Programs*)	*Prog*	::=	$cd_1 ... cd_n$
(*Class definition*)	*cd*	::=	`class` C `{` *inv mod* conc $fd_1 ... fd_k$ *cnd* $md_1 ... md_n$ `}`
(*Modifiable list*)	*mod*	::=	`modifiable := {` $fieldname_1, ..., fieldname_k$ `}`
(*Field definition*)	*fd*	::=	*accmod* C *fname*
(*Constructor definition*)	*cnd*	::=	*accmod* C `(` $C_1\ arg_1, ..., C_n\ arg_n$`)` `)` *spec* `{` $s_1 ... s_n$ `}`
(*Method definition*)	*md*	::=	*accmod* C *mname* `(` $C_1\ arg_1, ..., C_n\ arg_n$ `)` *spec* `{` $s_1 ... s_n$ `}`
(*Statements*)	*s*	::=	(*ifstmt* | *fieldacc* `=` x_2 | $C\ x$ | x_1 `=` *exp* | $x_3 = mcall$ | *leakstmt* | `return` x);
(*If statements*)	*ifstmt*	::=	`if(` x `)` `{` $s_1 ... s_k$ `}` `else` `{` $s_{k+1} ... s_n$ `}`
(*Leak statements*)	*leakstmt*	::=	`leak` x
(*Method call*)	*mcall*	::=	x.*mname* `(` $x_1, ..., x_k$ `)` | `new` C `(` $x_1, ..., x_n$ `)`
(*Specification*)	*spec*	::=	`requires` `assert` `ensures` *assertion*
(*Invariant*)	*inv*	::=	`invariant :=` *assertion*
(*Access modifier*)	*accmod*	::=	`private` | `public`
(*Field access*)	*fieldacc*	::=	x.*fieldname*
(*Assertion*)	*assertion*	::=	(*assertion op assertion*) | *sing*
(*Operation*)	*op*	::=	`+` | `-` | `|` | `&` | `*` | `/` | `%` | `!=` | `==` | `>` | `>=`
(*Singleton*)	*sing*	::=	*exp* | *spex*
(*Expression*)	*exp*	::=	(*exp op exp*) | *fieldacc* | x | `true` | `false` | *intLit*
(*Specification expression*)	*spex*	::=	`hidden(` x, *perm* `)` | `leakable(` x, *perm* `)` | `acc(` x, *perm* `)`
(*Permission amount*)	*perm*	::=	`write` | `none` | `wildcard` | *frac*
(*Fraction*)	*frac*	::=	*intLit* `/` *intLit*

Figure 3.2: Target language syntax

$[\![cd_1 ... cd_n]\!]$ ::= $[\![cd_1]\!] ... [\![cd_n]\!]$

$[\![class\ C\ \{inv\ mod\ fd_1 ... fd_k\ \ cnd\ \ md_1 ... md_n\}]\!]$ ::= $[\![fd_1]\!] ... [\![fd_k]\!]\ \ [\![cnd]\!]_{inv,mod}$ $[\![md_1]\!]_{inv,mod} ... [\![md_n]\!]_{inv,mod}$

$[\![accmod\ C\ fn]\!]$ ::= `field` $fn : [\![C]\!]$

$[\![$ `public` $C\ (C_1\ arg_1, ..., C_n\ arg_n)\ spec\ \{s_1 ... s_n\}]\!]_{inv,mod}$ ::= `method` C `_cons_h` $([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$ $spec\ \{bsch_C [\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}\}$

leaked_spec := ensures low(\result)
ensures leakable(\result))

`method` C `_cons_l` $([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$ leaked_spec
$\{bscl^C_{arg1,..,argn} [\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}$
$[\![$ `leak this` $]\!]^{public,C}_{inv,mod}\}$

$[\![$ `private` $C\ (C_1\ arg_1, ..., C_n\ arg_n)\ spec\ \{s_1 ... s_n\}]\!]_{inv,mod}$ ::= `method` C `_cons` $([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$ $spec\ \{bsch_C\ [\![s_1]\!]^{private,C}_{inv,mod} ... [\![s_n]\!]^{private,C}_{inv,mod}\}$

$[\![$ `public` $C\ mn\ (C_1\ arg_1, .., C_n\ arg_n)\ spec\ \{s_1 .. s_n\}]\!]_{inv,mod}$ ::= `method` mn `_h` $([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$ $returns$ (`ret:` $[\![C]\!]$) $spec'$ $\{[\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}\}$

leaked_spec := ensures low(\result)
ensures leakable(\result))

~~spec' := spec ++ ensures leakable(this)~~
~~=> low(result) ∧ leakable(result)~~
↳ not needed

`method` mn `_l` $([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$ $returns$ (`ret:` $[\![C]\!]$) leaked_spec
$\{bsml_{arg_1,...,arg_n}$
$[\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}$
~~`assert perm(leakable(ret))> 0)`~~ }
↳ moved to postcondition

$[\![$ `private` $C\ mn\ (C_1\ arg_1, .., C_n\ arg_n)\ spec\ \{s_1 .. s_n\}]\!]_{inv,mod}$ ::= `method` mn $([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$ $returns$ (`ret:` $[\![C]\!]$) $spec$ $\{[\![s_1]\!]^{private,C}_{inv,mod} ... [\![s_n]\!]^{private,C}_{inv,mod}\}$

where

$bsch_C$ = $\forall\ f\ fieldof\ C$ `inhale acc(this.` f `, write)`
`inhale acc(hidden(this),write)`

$bscl^C_{arg1,..,argn}$ = $\forall\ f\ fieldof\ C$
`inhale acc((this.` f `),write)`
$\forall\ arg\ in\ (arg_1, .., arg_n)$, $[\![arg.class]\!]$ == `Ref`
`inhale acc(leakable(` arg `), write)`
`inhale acc(hidden(this),write)`

assume low(arg1)
...
assume low(argn)
assume low(this)

$bsml_{arg1,..,argn}$ = `inhale acc(leakable(` arg_1 `), write)`
...
`inhale acc(leakable(` arg_n `),write)`
`inhale acc(leakable(this),write)`

assume low(arg1)
...
assume low(argn)
assume low(this)

∀f fieldof C,
f ∉ conc:
inhale acc(leakable(this.f)) ∧ low(this.f)

Figure 3.3: Here ::= means " the encoding of ... is" while = means "is"
∀ f fieldof C stmt means "for each field f of the class C do stmt"

$[\![$ `if(` x `)` `{` $s_1 ... s_k$ `}` `else` `{` $s_{k+1} ... s_n$ `}` $]\!]^{accmod,C}_{inv,mod}$ ::= `if(` x `)` `{` $[\![s_1]\!]^{accmod,C}_{inv,mod} ... [\![s_k]\!]^{accmod,C}_{inv,mod}$ `}`
`else` `{` $[\![s_{k+1}]\!]^{accmod,C}_{inv,mod} ... [\![s_n]\!]^{accmod,C}_{inv,mod}$ `}`

$[\![C\ x]\!]^{accmod,C}_{inv,mod}$ ::= `var` x : $[\![C]\!]$

$[\![x_0 = x.mn(x_1, ..., x_n)]\!]^{accmod,C}_{inv,mod}$
where x is verified ::= x_0 `=` $x.mn(x_1, ..., x_n)$

$[\![x_0 = x.mn(x_1, ..., x_n)]\!]^{accmod,C}_{inv,mod}$
where x is not verified ::= $\forall\ i\ in(1-n), [\![x_i.class]\!] ==$ `Ref`
`assert perm((leakable(` x_i `)) > 0`
x_0 `=` $x.mn(x_1, ..., x_n)$

assert low Event
assert low (x)
$\forall i \in (1-n)$
assert low(x_i)

$[\![x_0 = x.fn]\!]^{accmod,C}_{inv,mod}$
where fn $\in$ mod,
fn is a private field ::=

```
assert perm(leakable( x )) > 0
  || perm(hidden) > 0
if(perm(leakable( x )) > 0) {
```
$\forall\ f,\ f \in mod$
`var` f`_temp` : $[\![f.class]\!]$
`inhale` $inv[\forall f', f' \in mod, f'/f'$`_temp`$]$
$x_0 = fn$`_temp`
```
} else {
```
x_0 `=` $x.fn$
```
}
```

if (fn ∉ conc) {
assume low (fn_temp)
}

$[\![x_0 = x.fn]\!]^{accmod,C}_{inv,mod}$
where fn $\notin$ mod,
fn is a private field ::=

```
assert perm(leakable( x )) > 0
  || perm(hidden) > 0
if(perm(leakable( x )) > 0) {
```
$\forall\ f,\ f \in mod$
`var` f`_temp` : $[\![f.class]\!]$
`inhale` $inv[\forall f', f' \in mod, f'/f'$`_temp`$]$
$x_0 = x.fn$
```
} else {
```
x_0 `=` $x.fn$
```
}
```

if (fn ∉ conc) {
assume low (x.fn)
}

$[\![x_0 = x.fn]\!]^{accmod,C}_{inv,mod}$
where fn is a public field ::=

```
assert perm(leakable( x )) > 0
  || perm(hidden) > 0
if(perm(leakable( x )) > 0) {
  var pubvar : [[fn.class]]
  inhale acc(leakable(pubvar),wildcard)
  x = pubvar
```
assume low (pubvar)
```
} else {
```
x_0 `=` $x.fn$
```
}
```

$[\![x.fn = x_0]\!]^{accmod,C}_{inv,mod}$
where fn $\notin$ mod,
fn is a private field ::=

assert perm(leakable(x)) > 0
  || perm(hidden) > 0
if(perm(leakable(x)) > 0) {
  $\forall\ f,\ f \in mod$
    var f_temp : $[\![f.class]\!]$
  inhale $inv[\forall f', f' \in mod, f'/f'$_temp]
  fn_temp $= x_0$
  assert $inv[\forall f', f' \in mod, f'/f'$_temp]
} else {
  $x.fn$ = x_0
}

if (fn $\notin$ conc) {
assert low Event
assert low(x)
assert low(x_0)
}

→ maybe already taken care of by invariant? ⇒ one could also think of assert leakable (x_0) ⇒ handled by inv

$[\![x.fn = x_0]\!]^{accmod,C}_{inv,mod}$
where fn $\notin$ mod,
fn is a private field ::=

assert perm(leakable(x)) > 0
  || perm(hidden) > 0
if(perm(leakable(x)) > 0) {
  $\forall\ f,\ f \in mod$
    var f_temp : $[\![f.class]\!]$
  inhale $inv[\forall f', f' \in mod, f'/f'$_temp]
  $x.fn = x_0$
  assert $inv[\forall f', f' \in mod, f'/f'$_temp]
} else {
  $x.fn$ = x_0
}

if (fn $\notin$ conc) {
assert low Event
assert low(x)
assert low(x_0)
}

→ maybe already taken care of by invariant? ⇒ one could also think of assert leakable (x_0) ⇒ handled by inv

$[\![x.fn = x_0]\!]^{accmod,C}_{inv,mod}$
where fn is a public field, ::=

assert perm(leakable(x)) > 0
  || perm(hidden) > 0
if(perm(leakable(x)) > 0) {
  assert acc(leakable(x_0),wildcard)
} else {
  $x.fn$ = x_0
}

assert low Event
assert low(x)
assert low(x_0)

Figure 3.5: ∀ f fieldof C, f ∈ mod means "for every field of C that is in the set mod"

moved to post-condition of methods (2nd verif.)

$[\![$ `return` $x]\!]^{private,C}_{inv,mod}$ ::= `return` x

$[\![$ `return` $x]\!]^{public,C}_{inv,mod}$ ::= ~~`assert perm(leakable(` x `)) > 0`~~
`return` x

$[\![$ `leak` $x]\!]^{public,C}_{inv,mod}$ ::= `assert perm(leakable(` x `)) > 0`
`|| perm(hidden) > 0`
`if perm(leakable(` x `)) == 0 {`
`exhale acc(hidden(` x `),write)`
`inhale acc(leakable(` x `),write)`
`assert` inv
$\forall f$ $fieldof\ C, f \in mod \ || \ f.accmod == public$
`exhale acc(` f `,write)`
$\forall f'$ $fieldof\ C,\ f'.accmod == public, [\![f'.class]\!] ==$ `Ref`
`assert acc(leakable(` f' `),wildcard)`

$\forall f$ fieldof C, $f \notin$ conc :
assert low(x.f)

this way it is ensured that leakable(x) implies that all readable (non-concealed) contents of x are low

$[\![predName(x)]\!]^{public,C}_{inv,mod}$ ::= `acc(predName(` x `),write)`

where *predName* is the name of a predicate `}`

otherwise

$[\![$ `e` $]\!]^{private,C}_{inv,mod}$ ::= e

Figure 3.6: $inv[\forall f \in mod/f\_temp]$ means
"the assertion inv with every field f in mod replaced by f_temp"

The encoding is mostly a 1 to 1 application of the design we presented in the earlier sections with the notable additions of the fact that every time we branch on whether or not a given object is leakable we first assert that we have non-zero permission on either leakable or hidden of said object. The fact that an object is either held or not by unverified code is trivially true, but a careless specifier could lose either predicate by calling a method that required one or the other and ensured neither. To remain sound we require that at any time we use an object hidden or leakable status we must know whether the object is hidden or leakable.

Apart from the above addition, we do apply the design described earlier. For every assignment to a private field we verify that the invariant is maintained by inhaling it before