Sireum/Kiasan

an extensible symbolic execution framework

Technical and Implementation Walkthrough

Robby

Funding

CIS @ K-STATE
COMPUTING AND INFORMATION SCIENCES

SANTOS LAB

Overview

- Pilar Language -- Sireum IR
  - expression and action (command) sub-languages
  - syntax, AST, etc.

- SymExe Components
  - state and value
  - expression and action evaluations
  - extensions
  - decision procedure calls
  - computation tree exploration

Pilar (Sub) Language Syntax

e **::**= c | x | **(** e **)** | e op e

a **::**= **assert** e; | x **:=** e;

Pilar (Sub) Language Syntax

e **::=** *c* | *x* | **(** *e* **)** | *e op e*

a **::=** **assert** *e*; | *x* **:=** *e*;

Examples:

- Expressions: `5`, `@@x > @@y + 5`
- Actions: `@@x := 4; assert @@x > 0;`

Note: `@@` prefixes a global variable

Pilar AST: Expression

$$e \; ::= \; c \;|\; x \;|\; \textbf{(}\; e \;\textbf{)} \;|\; e \; op \; e$$

abstract class Exp(...) **extends** ...

sireum-pilar/src/main/scala/org/sireum/pilar/ast/PilarAstNode.scala

case class LiteralExp(typ : LiteralType.Type, literal : Any, text : String) **extends** Exp

case class NameExp(name : NameUser) **extends** Exp

case class TupleExp(exps : ISeq[Exp]) **extends** Exp

case class BinaryExp(op : BinaryOp, left : Exp, right : Exp) **extends** Exp

Pilar AST: Expression

$$e ::= c \mid x \mid (\, e \,) \mid e \; op \; e$$

sireum-pilar/src/main/scala/org/sireum/pilar/ast/PilarAstNode.scala

```
abstract class Exp(...) extends ...

case class LiteralExp(typ : LiteralType.Type, literal : Any, text : String) extends Exp

case class NameExp(name : NameUser) extends Exp

case class TupleExp(exps : ISeq[Exp]) extends Exp

case class BinaryExp(op : BinaryOp, left : Exp, right : Exp) extends Exp
```

Examples:

- `5` LiteralExp(LiteralType.INT, 5, "5")
- `@@x > @@y + 5` BinaryExp(">",
  NameExp(NameUser("@@x")),
  BinaryExp(">", ..., ...))

Pilar AST: Action

a `::=` **`assert`** e`;` | x **`:=`** e`;`

abstract class Action **extends** ...

sireum-pilar/src/main/scala/org/sireum/pilar/ast/PilarAstNode.scala

case class AssignAction(..., lhs : Exp, ..., rhs : Exp) **extends** Action **with** ...

case class AssertAction(..., cond : Exp, ...) **extends** Action

Examples:

- `@@x := 4;` AssignAction(..., NameExp(...), ..., LiteralExp(..., ..., ...))
- `assert @@x > 0;` AssertAction(..., BinaryExp(">", ..., ...), ...)

State and Value: Semantic Domains

$$
\begin{array}{llllll}
c, d & \in & \textbf{Integer} & & \text{set of integers} \\
\alpha, \beta, \gamma & \in & \textbf{Symbol} & = & \text{set of integer symbols} \\
v, w, u & \in & \textbf{Value} & = & \textbf{ScalarValue} = \textbf{Integer} \uplus \textbf{IntegerSymbol} \\
x, y, z & \in & \textbf{Variable} & & \text{set of variables} \\
\sigma & \in & \textbf{Store} & = & \textbf{Variable} \rightharpoonup \textbf{Value} \\
\mu & \in & \textbf{Status} & = & \{\mathsf{Normal}, \mathsf{Error}, \mathsf{Infeasible}\} \\
\phi & \in & \textbf{PathCond} & & \text{list of predicates over } \textbf{Value} \\
s^{\mathcal{S}} & \in & \textbf{State}^{\mathcal{S}} & = & \textbf{Label} \times \textbf{Store} \times \textbf{PathCond} \times \textbf{Status}
\end{array}
$$

State and Value: Semantic Domains

$$
\begin{array}{llcll}
c, d & \in & \mathbf{Integer} & & \text{set of integers} \\
\alpha, \beta, \gamma & \in & \mathbf{Symbol} & = & \text{set of integer symbols} \\
v, w, u & \in & \mathbf{Value} & = & \mathbf{ScalarValue} = \mathbf{Integer} \uplus \mathbf{IntegerSymbol} \\
x, y, z & \in & \mathbf{Variable} & & \text{set of variables} \\
\sigma & \in & \mathbf{Store} & = & \mathbf{Variable} \rightharpoonup \mathbf{Value} \\
\mu & \in & \mathbf{Status} & = & \{\mathsf{Normal}, \mathsf{Error}, \mathsf{Infeasible}\} \\
\phi & \in & \mathbf{PathCond} & & \text{list of predicates over } \mathbf{Value} \\
s^{\mathcal{S}} & \in & \mathbf{State}^{\mathcal{S}} & = & \mathbf{Label} \times \mathbf{Store} \times \mathbf{PathCond} \times \mathbf{Status}
\end{array}
$$

```
trait Value

trait ConcreteValue extends Value

trait AbstractValue extends Value
```

```
trait State[Self <: State[Self]]
                               extends Immutable with … {
  …
  def variable(varUri : ResourceUri) : Value

  def variable(varUri : ResourceUri,
                         value : Value)           : Self
  …
}
```

sireum-pilar/src/main/scala/org/sireum/pilar/state/State.scala

Kiasan State

```
trait KiasanStatePart[Self <: KiasanStatePart[Self]] extends … {
  def pathConditions : ISeq[Exp]
  …
  def addPathCondition(e : Exp) : Self =  …
  …
  def counters : IMap[ResourceUri, Int]
  …
  def next(uri : ResourceUri) : (Self, Int) = …
}

final case class BasicKiasanState(…)
    extends State[BasicKiasanState] with KiasanStatePart[BasicKiasanState]
    with … {
  …
}
```

sireum-kiasan/src/main/scala/org/sireum/kiasan/state/KiasanState.scala

(Big-Step) Operational Semantic Rules

- Forms
  - Expression: $s \vdash e \Rightarrow \langle s', v \rangle$
  - Action: $s \vdash a \Rightarrow s'$
- Semantic rules are *relations*
  - multiple rules can apply
  - cause branches in SymExe computation tree
  - implemented as *functions* returning a *sequence*
  - deterministic ordering of results
  - thus, deterministic computation tree exploration

(Big-Step) Operational Semantic Rules

- Forms
  - Expression: $s \vdash e \Rightarrow \langle s'_0, v_0 \rangle \mid \ldots \mid \langle s'_n, v_n \rangle$
  - Action: $s \vdash a \Rightarrow s'_0 \mid \ldots \mid s'_n$
- Semantic rules are *relations*
  - multiple rules can apply
  - cause branches in SymExe computation tree
  - implemented as *functions* returning a *sequence*
  - deterministic ordering of results
  - thus, deterministic computation tree exploration

Expression Evaluation

LIT
$$\frac{}{s \vdash \mathsf{c} \Rightarrow \langle s, c \rangle}$$

VAR
$$\frac{s_\sigma(x) = v}{s \vdash x \Rightarrow \langle s, v \rangle}$$

BINOP
$$\frac{s \vdash e_1 \Rightarrow \langle s', v \rangle \quad s' \vdash e_2 \Rightarrow \langle s'', w \rangle \quad s'' \vdash v \odot w \rightarrow \langle s''', u \rangle}{s \vdash e_1 \odot e_2 \Rightarrow \langle s''', u \rangle}$$

Expression Evaluation

$$\text{LIT} \qquad \frac{}{s \vdash \mathsf{c} \Rightarrow \langle s, c\rangle}$$

$$\text{VAR} \qquad \frac{s_\sigma(x) = v}{s \vdash x \Rightarrow \langle s, v\rangle}$$

$$\text{BINOP} \qquad \frac{s \vdash e_1 \Rightarrow \langle s', v\rangle \quad s' \vdash e_2 \Rightarrow \langle s'', w\rangle \quad s'' \vdash v \odot w \rightarrow \langle s''', u\rangle}{s \vdash e_1 \odot e_2 \Rightarrow \langle s''', u\rangle}$$

sireum-pilar/src/main/scala/org/sireum/pilar/eval/EvaluatorImpl.scala

```
val evalExp : (S, Exp) --> R = {
   …
   case (s, LiteralExp(_, n : Int, _)) => sec.intLiteral(s, n)
   …
   case (s, e : NameExp) if sp.isVar(e) => sec.variable(s, e.name)
   …
   case (s, BinaryExp(op, e1, e2)) => …
       for {
         sv1 <- eval(s, e1)
         sv2 <- eval(re2s(sv1), e2)
         sv3 <- sec.binaryOp(op, re2s(sv2), re2v(sv1), re2v(sv2))
       } yield sv3
  … }
```

Binary Operator Evaluation

$$\text{ABINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{A}} d \;\rightarrow\; \langle s, [\![\odot_{\mathrm{A}}]\!](c,d)\rangle}$$

$$\text{RBINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{R}} d \;\rightarrow\; \langle s, [\![\odot_{\mathrm{R}}]\!](c,d) \;?\; 1 \;:\; 0\rangle}$$

$$\text{ABINOP\_S} \quad \frac{\{v,w\} \not\subset \textbf{Integer} \quad \{v,w\} \subset \textbf{ScalarValue} \quad \alpha \text{ is fresh}}{s \vdash v \odot_{\mathrm{A}} w \;\Rightarrow\; \langle s[\alpha = v \odot_{\mathrm{A}} w]^{+}_{\phi}, \alpha\rangle}$$

$$\text{RBINOP\_S} \quad \frac{\{v,w\} \not\subset \textbf{Integer} \quad \{v,w\} \subset \textbf{ScalarValue}}{s \vdash v \odot_{\mathrm{R}} w \;\rightarrow\; \langle s[v \odot_{\mathrm{R}} w]^{+}_{\phi}, 1\rangle \;\mid\; \langle s[v(\odot_{\tilde{\mathrm{R}}})w]^{+}_{\phi}, 0\rangle}$$

$\odot_{\mathrm{A}} \in \{\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{\%}\}$ $\qquad$ $\odot_{\mathrm{R}} \in \{\texttt{==},\ \texttt{!=},\ \texttt{<},\ \texttt{>},\ \texttt{<=},\ \texttt{>=}\}$

$$\odot_{\tilde{\mathrm{R}}} = \{(\texttt{==}, \texttt{!=}), (\texttt{!=}, \texttt{==}), (\texttt{<}, \texttt{>=}), (\texttt{>}, \texttt{<=}), (\texttt{<=}, \texttt{>}), (\texttt{>=}, \texttt{<})\}(\odot_{\mathrm{R}})$$

Binary Operator Evaluation

ABINOP_C $$\frac{}{s \vdash c \odot_{\mathrm{A}} d \rightarrow \langle s, [\![\odot_{\mathrm{A}}]\!](c, d)\rangle}$$

RBINOP_C $$\frac{}{s \vdash c \odot_{\mathrm{R}} d \rightarrow \langle s, [\![\odot_{\mathrm{R}}]\!](c, d) \ ?\ 1 \ :\ 0\rangle}$$

ABINOP_S $$\frac{\{v, w\} \not\subset \textbf{Integer} \quad \{v, w\} \subset \textbf{ScalarValue} \quad \alpha \text{ is fresh}}{s \vdash v \odot_{\mathrm{A}} w \Rightarrow \langle s[\alpha = v \odot_{\mathrm{A}} w]^{+}_{\phi}, \alpha\rangle}$$

RBINOP_S $$\frac{\{v, w\} \not\subset \textbf{Integer} \quad \{v, w\} \subset \textbf{ScalarValue}}{s \vdash v \odot_{\mathrm{R}} w \rightarrow \langle s[v \odot_{\mathrm{R}} w]^{+}_{\phi}, 1\rangle \ \mid \ \langle s[v(\tilde{\odot_{\mathrm{R}}})w]^{+}_{\phi}, 0\rangle}$$

$\odot_{\mathrm{A}} \in \{\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{\%}\}$ $\odot_{\mathrm{R}} \in \{\texttt{==},\ \texttt{!=},\ \texttt{<},\ \texttt{>},\ \texttt{<=},\ \texttt{>=}\}$

$$\tilde{\odot_{\mathrm{R}}} = \{(\texttt{==}, \texttt{!=}), (\texttt{!=}, \texttt{==}), (\texttt{<}, \texttt{>=}), (\texttt{>}, \texttt{<=}), (\texttt{<=}, \texttt{>}), (\texttt{>=}, \texttt{<})\}(\odot_{\mathrm{R}})$$

Binary Operator Evaluation

$$\text{ABINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{A}} d \rightarrow \langle s, [\![\odot_{\mathrm{A}}]\!](c,d)\rangle}$$

$$\text{RBINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{R}} d \rightarrow \langle s, [\![\odot_{\mathrm{R}}]\!](c,d)\ ?\ 1\ :\ 0\rangle}$$

$$\text{ABINOP\_S} \quad \frac{\{v,w\} \not\subset \textbf{Integer} \quad \{v,w\} \subset \textbf{ScalarValue} \quad \alpha \text{ is fresh}}{s \vdash v \odot_{\mathrm{A}} w \Rightarrow \langle s[\alpha = v \odot_{\mathrm{A}} w]^{+}_{\phi}, \alpha\rangle}$$

$$\text{RBINOP\_S} \quad \frac{\{v,w\} \not\subset \textbf{Integer} \quad \{v,w\} \subset \textbf{ScalarValue}}{s \vdash v \odot_{\mathrm{R}} w \rightarrow \langle s[v \odot_{\mathrm{R}} w]^{+}_{\phi}, 1\rangle \ \mid\ \langle s[v(\odot_{\tilde{\mathrm{R}}})w]^{+}_{\phi}, 0\rangle}$$

$\odot_{\mathrm{A}} \in \{+,\ -,\ *,\ /,\ \%\}$ $\qquad\qquad$ $\odot_{\mathrm{R}} \in \{==,\ !=,\ <,\ >,\ <=,\ >=\}$

$$\odot_{\tilde{\mathrm{R}}} = \{(==, !=), (!=, ==), (<, >=), (>, <=), (<=, >), (>=, <)\}(\odot_{\mathrm{R}})$$

Binary Operator Evaluation

$$\text{ABINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{A}} d \rightarrow \langle s, [\![\odot_{\mathrm{A}}]\!](c,d)\rangle}$$

$$\text{RBINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{R}} d \rightarrow \langle s, [\![\odot_{\mathrm{R}}]\!](c,d) \ ?\ 1 \ :\ 0\rangle}$$

$$\text{ABINOP\_S} \quad \frac{\{v,w\} \not\subset \textbf{Integer} \quad \{v,w\} \subset \textbf{ScalarValue} \quad \alpha \text{ is fresh}}{s \vdash v \odot_{\mathrm{A}} w \Rightarrow \langle s[\alpha = v \odot_{\mathrm{A}} w]^{+}_{\phi}, \alpha\rangle}$$

$$\text{RBINOP\_S} \quad \frac{\{v,w\} \not\subset \textbf{Integer} \quad \{v,w\} \subset \textbf{ScalarValue}}{s \vdash v \odot_{\mathrm{R}} w \rightarrow \langle s[v \odot_{\mathrm{R}} w]^{+}_{\phi}, 1\rangle \ \mid \ \langle s[v(\odot_{\tilde{\mathrm{R}}})w]^{+}_{\phi}, 0\rangle}$$

$\odot_{\mathrm{A}} \in \{\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{\%}\}$ $\quad\quad$ $\odot_{\mathrm{R}} \in \{\texttt{==},\ \texttt{!=},\ \texttt{<},\ \texttt{>},\ \texttt{<=},\ \texttt{>=}\}$

$$\odot_{\tilde{\mathrm{R}}} = \{(\texttt{==}, \texttt{!=}), (\texttt{!=}, \texttt{==}), (\texttt{<}, \texttt{>=}), (\texttt{>}, \texttt{<=}), (\texttt{<=}, \texttt{>}), (\texttt{>=}, \texttt{<})\}(\odot_{\mathrm{R}})$$

Pilar and Semantics Extension

- In Pilar, there is no "built-in" semantics
  - The AST literal expression is just a language feature that can be interpreted/evaluated in multiple ways
- Semantics are defined through extensions
  - a set of extensions make a language profile
- Thus, we need to define extensions for
  - looking up and update variables
  - representing integer values
  - applying binary operators on integer values
  - etc.

Variable Access Extension

```
object MyVariableAccessExtension extends ExtensionCompanion { … }

class MyVariableAccessExtension[S <: State[S]](config : EvaluatorConfiguration[…])
    extends Extension[S, Value, ISeq[(S, Value)], ISeq[(S, Boolean)], ISeq[S]] {
  …
  def varUri(x : NameUser) = if (x.hasResourceInfo) x.resourceUri else x.name

  implicit def re2r(p : (S, Value)) = ilist(p)

  @VarLookup
  def variableLookup : (S, NameUser) --> ISeq[(S, Value)] = {
   case (s, x) => (s, s.variable(varUri(x)))
  }
  …
}
```

$$\text{VAR} \quad \frac{s_\sigma(x) = v}{s \vdash x \Rightarrow \langle s, v \rangle}$$

stress12-kiasan/src/test/scala/stress12/MyVariableAccessExtension.scala

Integer Extension

```
object MyIntExtension extends ... {
  val URI_PATH = "stress12/MyIntegerExtension"
  …
  val KINT_TYPE_URI = "pilar://typeext/" + URI_PATH + "/KInt"
}

sealed abstract class I extends NonReferenceValue

case class CI(value : Int) extends I with ConcreteValue { …  }

case class KI(num : Int) extends I with KiasanValue {
  def typeUri = MyIntegerExtension.KINT_TYPE_URI
}

final class MyIntExtension[S <: KiasanStatePart[S]](
  config : EvaluatorConfiguration[…]) extends Extension[…] {
  …
}
```

stress12-kiasan/src/main/scala/stress12/MyIntExtension.scala

Integer Extension: Literal

$$\text{LIT} \quad \frac{}{s \vdash \mathsf{c} \Rightarrow \langle s, c \rangle}$$

```
@Literal(classOf[Int])
def literal : (S, Int) --> ISeq[(S, Value)] = {
  case (s, n) => (s, CI(n))
}
```

Integer Extension: Fresh Sym. Int.

$$\text{ABINOP\_S} \quad \frac{\{v, w\} \not\subset \textbf{Integer} \quad \{v, w\} \subset \textbf{ScalarValue} \quad \boxed{\alpha \text{ is fresh}}}{s \vdash v \odot_{\mathrm{A}} w \Rightarrow \langle s[\alpha = v \odot_{\mathrm{A}} w]^{+}_{\phi}, \alpha \rangle}$$

```
@FreshKiasanValue
def freshKI : (S, ResourceUri) --> (S, Value) = {
  case (s, KINT_TYPE_URI) => {
    val (nextS, num) = s.next(KINT_TYPE_URI)
    (nextS, KI(num))
  }
}
```

Integer Extension: $\odot_A$ ConExe Eval

```
@Binaries(Array("+", "-", "*", "/", "%"))
def opAEval : (S, Value, String, Value) --> ISeq[(S, Value)] =
{ case (s, c : CI, opA : String, d : CI) => (s, opASem(opA)(c, d))
  case (s, v : CI, opA : String, w : KI) => opAHelper(s, v, opA, w)
  case (s, v : KI, opA : String, w : CI) => opAHelper(s, v, opA, w)
  case (s, v : KI, opA : String, w : KI) => opAHelper(s, v, opA, w) }

def opASem(opA : String) : (CI, CI) => CI = { (c, d) =>
  opA match { case "+" => CI(c.value + d.value)
              case "-" => CI(c.value - d.value)          case …  }  }
```

$$\text{ABINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{A}} d \rightarrow \langle s, [\![\odot_{\mathrm{A}}]\!](c, d)\rangle}$$

Integer Extension: $\odot_A$ SymExe Eval

```
@Binaries(Array("+", "-", "*", "/", "%"))
def opAEval : (S, Value, String, Value) --> ISeq[(S, Value)] =
{ case (s, c : CI, opA : String, d : CI) => (s, opASem(opA)(c, d))
  case (s, v : CI, opA : String, w : KI) => opAHelper(s, v, opA, w)
  case (s, v : KI, opA : String, w : CI) => opAHelper(s, v, opA, w)
  case (s, v : KI, opA : String, w : KI) => opAHelper(s, v, opA, w) }
```

$$\text{ABINOP\_S}\quad \frac{\{v, w\} \not\subset \textbf{Integer} \quad \{v, w\} \subset \textbf{ScalarValue} \quad \alpha \text{ is fresh}}{s \vdash v \odot_{\mathrm{A}} w \Rightarrow \langle s[\alpha = v \odot_{\mathrm{A}} w]^{+}_{\phi}, \alpha \rangle}$$

```
implicit def v2e(v : Value) : Exp = ValueExp(v)

def opAHelper(s : S, v : Value, opA : String, w : Value) : ISeq[(S, Value)] =
{ val (nextS, a) = freshKI(s, KINT_TYPE_URI)
  (nextS.addPathCondition(BinaryExp("==", a, BinaryExp(opA, v, w))), a)   }
```

Integer Extension: $\odot_A$ Eval

```
@Binaries(Array("+", "-", "*", "/", "%"))
def opAEval : (S, Value, String, Value) --> ISeq[(S, Value)] =
{ case (s, c : CI, opA : String, d : CI) => (s, opASem(opA)(c, d))
  case (s, v : CI, opA : String, w : KI) => opAHelper(s, v, opA, w)
  case (s, v : KI, opA : String, w : CI) => opAHelper(s, v, opA, w)
  case (s, v : KI, opA : String, w : KI) => opAHelper(s, v, opA, w) }

def opASem(opA : String) : (CI, CI) => CI = { (c, d) =>
  opA match { case "+" => CI(c.value + d.value)
                    case "-" => CI(c.value - d.value)           case …  }  }

implicit def v2e(v : Value) : Exp = ValueExp(v)

def opAHelper(s : S, v : Value, opA : String, w : Value) : ISeq[(S, Value)] =
{ val (nextS, a) = freshKI(s, KINT_TYPE_URI)
  (nextS.addPathCondition(BinaryExp("==", a, BinaryExp(opA, v, w))), a)   }
```

Integer Extension: ConExe $\odot_R$ Eval

```
@Binaries(Array("==", "!=", ">", ">=", "<", "<="))
def opREval : (S, Value, String, Value) --> ISeq[(S, Value)] =
{ case (s, c : CI, opR : String, d : CI) =>  (s, opRSem(opR)(c, d))
  …
  case (s, v : KI, opR : String, w : KI) => opRHelper(s, opR, v, w) }

def opRSem(opR : String) : (CI, CI) => CI = { (c, d) =>
   if (opR match { case "==" => c.value == d.value
                   case "!=" => c.value != d.value
                   case …                          }) CI(1) else CI(0) }
```

$$\text{RBINOP\_C} \quad \frac{}{s \vdash c \odot_{\mathrm{R}} d \rightarrow \langle s, [\![\odot_{\mathrm{R}}]\!](c,d) \ ? \ 1 \ : \ 0 \rangle}$$

Integer Extension: SymExe $\odot_R$ Eval

```
@Binaries(Array("==", "!=", ">", ">=", "<", "<="))
def opREval : (S, Value, String, Value) --> ISeq[(S, Value)] =
{ case (s, c : CI, opR : String, d : CI) =>  (s, opRSem(opR)(c, d))
  …
  case (s, v : KI, opR : String, w : KI) => opRHelper(s, opR, v, w) }
```

$$\text{RBINOP\_S} \quad \frac{\{v, w\} \not\subset \textbf{Integer} \quad \{v, w\} \subset \textbf{ScalarValue}}{s \vdash v \odot_{\mathrm{R}} w \rightarrow \langle s[v \odot_{\mathrm{R}} w]^{+}_{\phi}, 1\rangle \mid \langle s[v(\tilde{\odot_{\mathrm{R}}})w]^{+}_{\phi}, 0\rangle}$$

```
val comp = Map("==" -> "!=",  "!=" -> "==", …)

def opRHelper(s : S, v : Value, opR : String, w : Value) : ISeq[(S, Value)] =
{ ilist((s.addPathCondition(BinaryExp(         opR,  v, w)).requestInconsistencyCheck, CI(1)),
        (s.addPathCondition(BinaryExp(comp(opR), v, w)).requestInconsistencyCheck, CI(0)))) }
```

Integer Extension: $\odot_R$ Eval

```
@Binaries(Array("==", "!=", ">", ">=", "<", "<="))
def opREval : (S, Value, String, Value) --> ISeq[(S, Value)] =
{ case (s, c : CI, opR : String, d : CI) =>  (s, opRSem(opR)(c, d))
  …
  case (s, v : KI, opR : String, w : KI) => opRHelper(s, opR, v, w) }

def opRSem(opR : String) : (CI, CI) => CI = { (c, d) =>
    if (opR match { case "==" => c.value == d.value
                    case "!=" => c.value != d.value
                    case …                          }) CI(1) else CI(0) }

val comp = Map("==" -> "!=",  "!=" -> "==", …)

def opRHelper(s : S, v : Value, opR : String, w : Value) : ISeq[(S, Value)] =
{ ilist((s.addPathCondition(BinaryExp(          opR,  v, w)).requestInconsistencyCheck, CI(1)),
        (s.addPathCondition(BinaryExp(comp(opR), v, w)).requestInconsistencyCheck, CI(0)))) }
```

Putting It All Together: Testing

```
@RunWith(classOf[JUnitRunner])
class MyIntExtensionExpTest extends StressTest[…] with ExpEvaluatorTestFramework[…] {

  val state = new BasicKiasanState()

  type S = BasicKiasanState
  type V = Value

  Evaluating expression "1" gives "1" satisfying { r : (S, V)  => r.value is 1 }

  Evaluating.expression("2 + 3").gives("5").satisfying({ r : (S, V) => r.value is 5 })

  Evaluating expression "2 * @@x" on (state("@@x" -> 3)) gives "6" satisfying {
    r : (S, V) => r.value is 6 }
  …
}
```

stress12-kiasan/src/test/scala/stress12/MyIntExtensionExpTest.scala

Putting It All Together: Testing

```
val alpha = KI(0)                    val beta = KI(1)                    val gamma = KI(2)
…
val rewriter = Rewriter.build[Exp](
                        { case NameExp(NameUser("alpha"))   => ValueExp(alpha)
                          case NameExp(NameUser("beta"))    => ValueExp(beta)
                          case NameExp(NameUser("gamma"))   => ValueExp(gamma)
                          case LiteralExp(_, n : Int, _)    => ValueExp(CI(n))    })
…
Evaluating expression "alpha * 2" gives "beta, where beta=alpha*2" satisfying
{ r : (S, V) => r.value is beta
          r.state.pathConditions is "beta == alpha * 2" }

Evaluating expression "@@y * 3 + 2" on (state("@@y" -> alpha)) gives
  "gamma, where gamma=beta+2 and beta=alpha*3" satisfying
{ r : (S, V) => r.value is gamma
          r.state.pathConditions is ("gamma == beta + 2", "beta == alpha * 3")          }
```

Decision Procedure (DP) Calls

- Each extension contributing to path condition should define translation to decision procedure back-ends (SMT solver: Z3, etc.)
    - only need to handle the *constraint forms* that *it contributes* and process *models* back from DP

- Z3 (process) translation example:

  "gamma == beta + 2", "beta == alpha * 3"

```
(declare-const i!0 Int)
(declare-const i!1 Int)
(assert (= i!1 (* i!0 3)))
(declare-const i!2 Int)
(assert (= i!2 (+ i!1 2)))
```

DP Calls: Issues

- Which theories?
  - e.g., bit-vector or arbitrary precision integer?
- What if the theory that we want is unsupported or supported but best-effort?
  - what if we get UNKNOWN answer?
  - e.g., non-linear arith., unbounded strings, etc.
  - weakening via uninterpreted functions with axioms (sound abstraction)
  - consequences
    - cannot depend on SAT answers
    - cannot dependably get models

DP Calls: Topi Interface

- check conjuncts satisfiability

Evaluating expression "@@y * 3 + 2" on (state("@@y" -> alpha)) gives
"gamma, where gamma=beta+2 and beta=alpha*3" satisfying

```
{ r : (S, V) =>  r.value is gamma
                 r.state.pathConditions is ("gamma == beta + 2", "beta == alpha * 3")
                 check(r.state) is TopiResult.SAT
}
```

DP Calls: Topi Interface

- check conjuncts satisfiability

```
Evaluating expression "@@y * 3 + 2" on (state("@@y" -> alpha)) gives
      "gamma, where gamma=beta+2 and beta=alpha*3" satisfying
  { r : (S, V) =>  r.value is gamma
                   r.state.pathConditions is ("gamma == beta + 2", "beta == alpha * 3")
                   check(r.state) is TopiResult.SAT
                   checkModel(r.state)                                              }
```

- if SAT, retrieve model and *confirm* it

```
def checkModel(s : S)
{ val m = getModel(s)
  val r = Rewriter.build[Exp]({ case ValueExp(ki : KI) => ValueExp(m(ki)) })
  val evaluator = newExpEvaluator(state)
  for (pc <- s.pathConditions) evaluator.evalExp(state, r(pc)).foreach(_.value is 1)  }
```

For You To Do

- Implement an integer extension for
    - the minus unary operator

Adding (Mutable) Integer List

e `::= … |` `` `[ ``e_0`,…,`e_n`]`
`| car` e `| cdr` e
a `::= … | update (`e_1`,`e_2`);`

Intents:

- `` `[ ``e_0`,…,`e_n`]`, builds a list containing $e_0,\dots,e_n$
- `car` e, retrieves the first int data from list e
- `cdr` e, retrieves the tail of list e
- `update (`e_1`,`e_2`);`, replace (mutate) the first int data of list e_1 with int e_2

Adding (Mutable) Integer List

```
e ::= … | `[e0,…,en]
        | car e | cdr e
a ::= … | update (e1,e2);
```

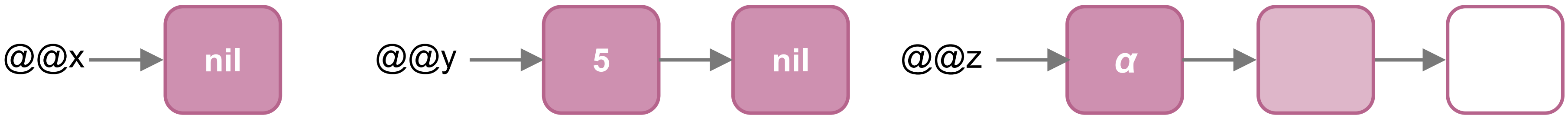
$$e ::= \ldots \mid \texttt{`[}e_0\texttt{,}\ldots\texttt{,}e_n\texttt{]} \mid \texttt{car}\ e \mid \texttt{cdr}\ e$$

$$a ::= \ldots \mid \texttt{update (}e_1\texttt{,}e_2\texttt{);}$$

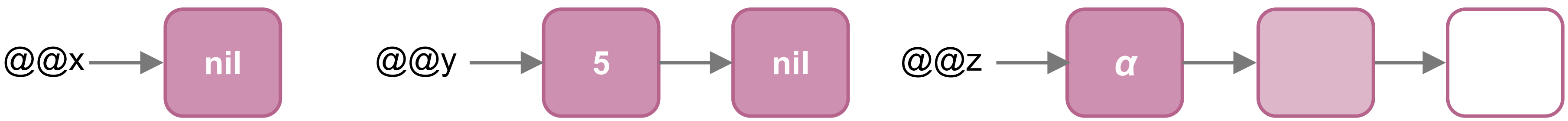
Examples:

- `car @@x` CallExp(NameExp(NameUser("car")), NameExp(NameUser("@@x")))
- ``1 + `[@@y]`` BinaryExp("+", LiteralExp(…, 1, …), ListExp(ilist(NameExp(…))))
- ``update (2, `[1]);`` ExtCallAction(
  CallExp(…, TupleExp(
  ilist(LiteralExp(…, 2, …),
  ListExp(ilist(LiteralExp(…, 1, …))))))

Integer List Representation



Integer List Representation



use heap to model mutable lists

Global Store

Var	Value
@@x	nil
@@y	(0,0)
@@z	(0,1)

Heap: 0

Address	Contents
0	data = 5 next = nil
1	data = α next = (0,2)
2	next = (0,3)
3	

...

Heap: N

...

Integer List Extension

```
object MyIntListExtension extends ExtensionCompanion {
  …
  val URI_PATH = "stress12/MyIntListExtension"
  val KINT_LIST_TYPE_URI = "pilar://typeext/" + URI_PATH + "/KIntList"

  val nextFieldUri = "next"
  val intDataFieldUri = "data"
  val listHeapIdKey = "ListHeapId"
}

object NilValue extends ConcreteValue with ReferenceValue

final class MyIntListExtension[S <: KiasanStatePart[S] with Heap[S]](
  config : EvaluatorConfiguration[…]) extends Extension[…] {
  …
}
```

Integer List Extension: `[e_0,…,e_n]

```
val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
val lhid = heapConfig.heapId(listHeapIdKey)

@NewList
def newList : (S, ISeq[Value]) --> ISeq[(S, Value)] = {
  case (s, vs) if vs.forall { _.isInstanceOf[I] } =>
    var l : ReferenceValue = NilValue
    var newS = s
    for (v <- vs.reverse) {
      val p = newS.newObject(lhid, nextFieldUri -> l, intDataFieldUri -> v)
      newS = first2(p)
      l = second2(p)
    }
    (newS, l)
}
```

Integer List Extension: `` `[e_0, …, e_n] ``

```
val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
val lhid = heapConfig.heapId(listHeapIdKey)

@NewList
def newList : (S, ISeq[Value]) --> ISeq[(S, Value)] = {
  case (s, vs) if vs.forall { _.isInstanceOf[I] } =>
    var l : ReferenceValue = NilValue
    var newS = s
    for (v <- vs.reverse) {
      val p = newS.newObject(lhid, nextFieldUri -> l, intDataFieldUri -> v)
      newS = first2(p)
      l = second2(p)
    }
    (newS, l)
}
```

`[1, 2, 3]

Integer List Extension: `[e_0, …, e_n]

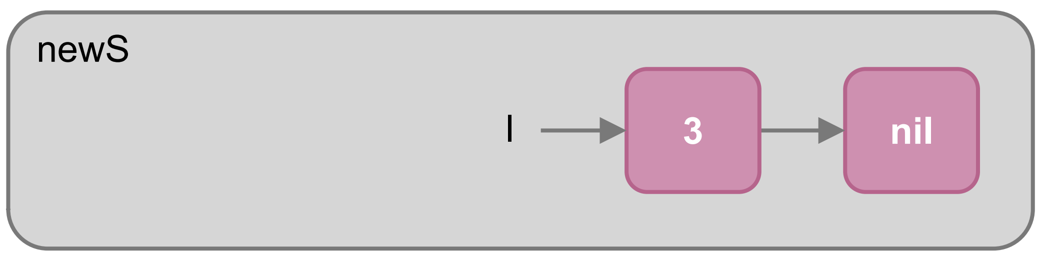
```
val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
val lhid = heapConfig.heapId(listHeapIdKey)

@NewList
def newList : (S, ISeq[Value]) --> ISeq[(S, Value)] = {
  case (s, vs) if vs.forall { _.isInstanceOf[I] } =>
    var l : ReferenceValue = NilValue
    var newS = s
    for (v <- vs.reverse) {
      val p = newS.newObject(lhid, nextFieldUri -> l, intDataFieldUri -> v)
      newS = first2(p)
      l = second2(p)
    }
    (newS, l)
}
```

`[1, 2, 3]

Integer List Extension: `[e_0,…,e_n]

```
val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
val lhid = heapConfig.heapId(listHeapIdKey)

@NewList
def newList : (S, ISeq[Value]) --> ISeq[(S, Value)] = {
  case (s, vs) if vs.forall { _.isInstanceOf[I] } =>
    var l : ReferenceValue = NilValue
    var newS = s
    for (v <- vs.reverse) {
      val p = newS.newObject(lhid, nextFieldUri -> l, intDataFieldUri -> v)
      newS = first2(p)
      l = second2(p)
    }
    (newS, l)
}
```

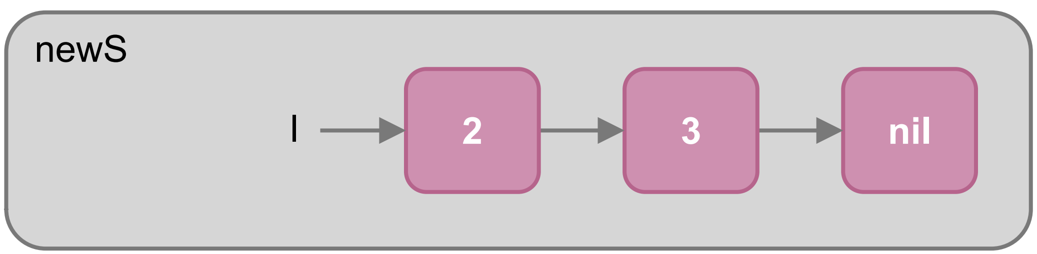
`[1, 2, 3]



Integer List Extension: `` `[$e_0$,…,$e_n$] ``

```
val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
val lhid = heapConfig.heapId(listHeapIdKey)

@NewList
def newList : (S, ISeq[Value]) --> ISeq[(S, Value)] = {
  case (s, vs) if vs.forall { _.isInstanceOf[I] } =>
    var l : ReferenceValue = NilValue
    var newS = s
    for (v <- vs.reverse) {
      val p = newS.newObject(lhid, nextFieldUri -> l, intDataFieldUri -> v)
      newS = first2(p)
      l = second2(p)
    }
    (newS, l)
}
```

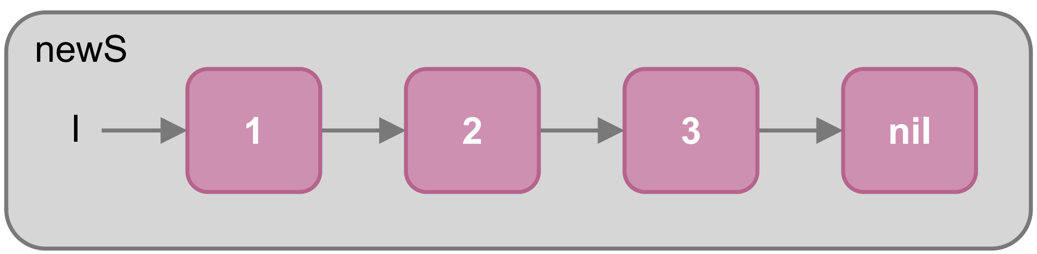
`[1, 2, 3]



Integer List Extension: `` `[$e_0$, …, $e_n$] ``

```
val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
val lhid = heapConfig.heapId(listHeapIdKey)

@NewList
def newList : (S, ISeq[Value]) --> ISeq[(S, Value)] = {
  case (s, vs) if vs.forall { _.isInstanceOf[I] } =>
    var l : ReferenceValue = NilValue
    var newS = s
    for (v <- vs.reverse) {
      val p = newS.newObject(lhid, nextFieldUri -> l, intDataFieldUri -> v)
      newS = first2(p)
      l = second2(p)
    }
    (newS, l)
}
```

`[1, 2, 3]



Integer List Extension Test: `` `[$e_0$, …, $e_n$] ``

```
@RunWith(classOf[JUnitRunner])
class MyIntListExtensionExpTest extends StressTest[…] with … {
  type S = KiasanStateWithHeap
  val config = KiasanEvaluatorTestUtil.newConfig[S](MyIntListExtension,  MyIntExtension, …)
  val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
  val lhid = heapConfig.heapId(MyIntListExtension.listHeapIdKey)
  val state = new KiasanStateWithHeap(Vector(Vector()))
  def newExpEvaluator(s : S) = config.evaluator.mainEvaluator

  Evaluating expression "`[1, 2, 3]" on state gives "[1, 2, 3]" satisfying { r : (S, V) =>


  }
  …
}
```

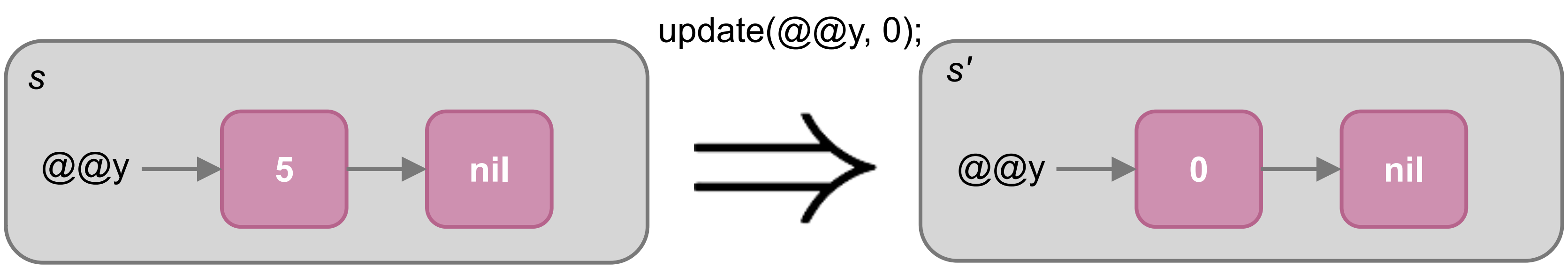
stress12-kiasan/src/test/scala/stress12/MyIntListExtensionTest.scala

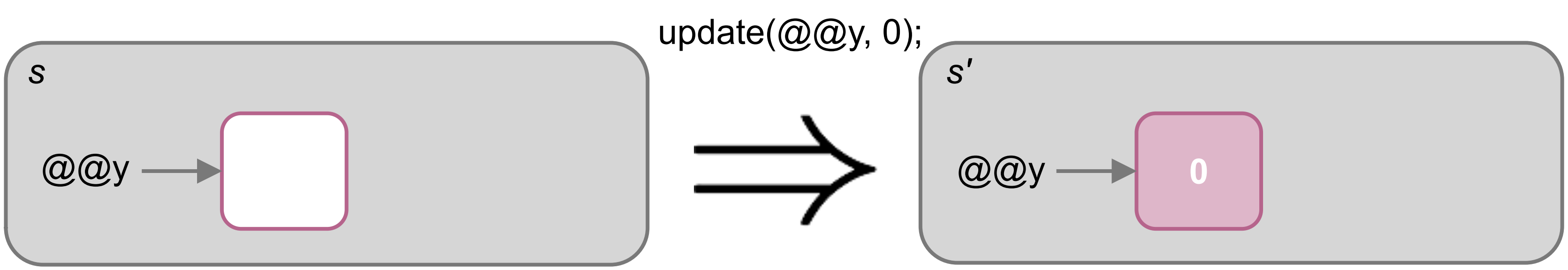
Integer List Extension Test: `` `[e_0, …, e_n] ``

```
@RunWith(classOf[JUnitRunner])
class MyIntListExtensionExpTest extends StressTest[…] with … {
  type S = KiasanStateWithHeap
  val config = KiasanEvaluatorTestUtil.newConfig[S](MyIntListExtension,  MyIntExtension, …)
  val heapConfig = config.adapter[EvaluatorHeapConfiguration[…]]
  val lhid = heapConfig.heapId(MyIntListExtension.listHeapIdKey)
  val state = new KiasanStateWithHeap(Vector(Vector()))
  def newExpEvaluator(s : S) = config.evaluator.mainEvaluator

  Evaluating expression "`[1, 2, 3]" on state gives "[1, 2, 3]" satisfying { r : (S, V) =>
    println(r.value)          RV(0,2)
    println(r.state)          KiasanStateWithHeap(Vector(Vector(
  }                             O(Map(next -> stress12.NilValue$@293bdd36,
  …                                   data -> CI(3))),
}                               O(Map(next -> RV(0,0),
                                      data -> CI(2))),
                                O(Map(next -> RV(0,1),
                                      data -> CI(1))))), …)
```

Integer List Extension: $\texttt{update}(e_1, e_2);$

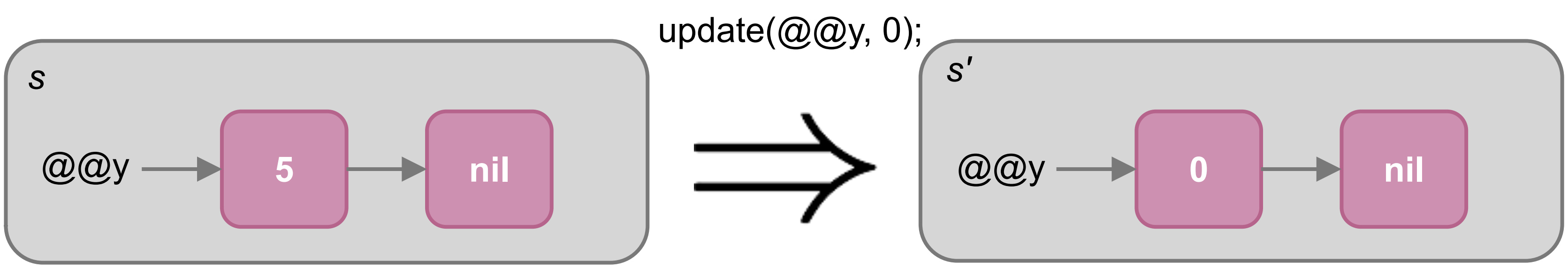
```
implicit def s2sr(s : S) = ilist(s)

@TopLevel @ActionExt
def update : (S, Value, Value) --> ISeq[S] = {
  case (s, rv @ Heap.RV(hid, _), v : I) if hid == lhid => s.update(rv, intDataFieldUri, v)
}
```





Integer List Extension: `update(e₁,e₂);`

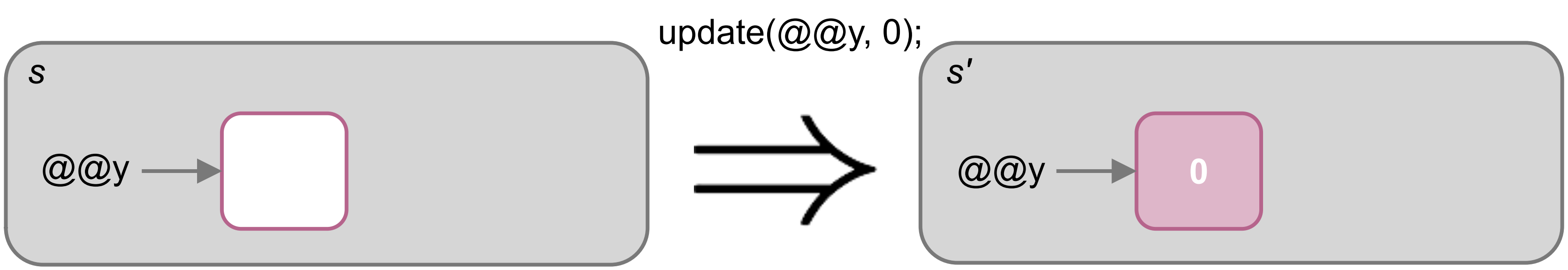
```
implicit def s2sr(s : S) = ilist(s)

@TopLevel @ActionExt
def update : (S, Value, Value) --> ISeq[S] = {
  case (s, rv @ Heap.RV(hid, _), v : I) if hid == lhid => s.update(rv, intDataFieldUri, v)
}
```




```
Evaluating action "update(@@y, 0);" on
{ val (s, rv) = state.newObject(lhid, intDataFieldUri -> CI(5), nextFieldUri -> NilValue)
  s.variable("@@y", rv) } gives "a new state" satisfying { s : S =>
    val rv = s.variable("@@y").asInstanceOf[ReferenceValue]
    s.lookup(rv, intDataFieldUri) is 0
    s.lookup(rv, nextFieldUri) is NilValue                          }
```

Integer List Extension: `update(e1,e2);`

```
implicit def s2sr(s : S) = ilist(s)

@TopLevel @ActionExt
def update : (S, Value, Value) --> ISeq[S] = {
  case (s, rv @ Heap.RV(hid, _), v : I) if hid == lhid => s.update(rv, intDataFieldUri, v)
}
Evaluating action "update(@@y, 0);" on {
  val (s, rv) = state.newObject(lhid); s.variable("@@y", rv)
} gives "a new state" satisfying { s : S =>
    val rv = s.variable("@@y").asInstanceOf[ReferenceValue]
    s.lookup(rv, intDataFieldUri) is 0
    s.hasFieldValue(rv, nextFieldUri) is false                    }
```

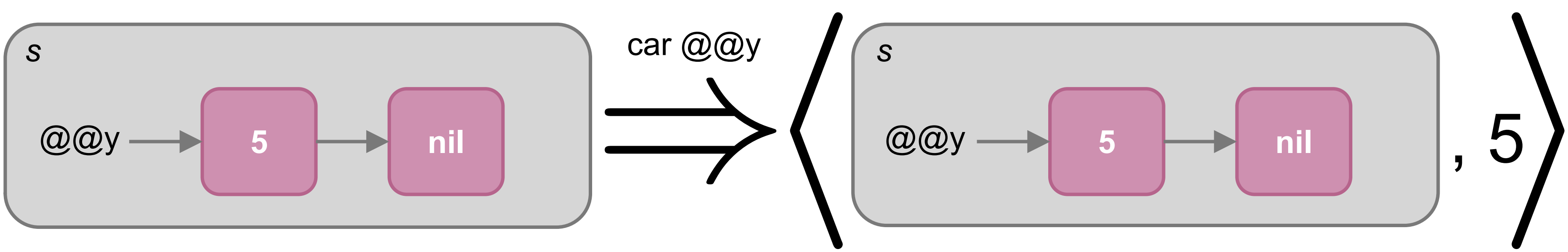


Integer List Extension: `car e`

```
@TopLevel @ExpExt
def car : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, intDataFieldUri))
      (s, s.lookup(rv, intDataFieldUri))
    else {
      val (s2, alpha) = sei.freshKiasanValue(s, MyIntExtension.KINT_TYPE_URI)
      val s3 = s2.update(rv, intDataFieldUri, alpha)
      (s3, alpha)
    }
}
```

Integer List Extension: `car e`

```
@TopLevel @ExpExt
def car : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, intDataFieldUri))
      (s, s.lookup(rv, intDataFieldUri))
    else {
      val (s2, alpha) = sei.freshKiasanValue(s, MyIntExtension.KINT_TYPE_URI)
      val s3 = s2.update(rv, intDataFieldUri, alpha)
      (s3, alpha)
    }
}
```



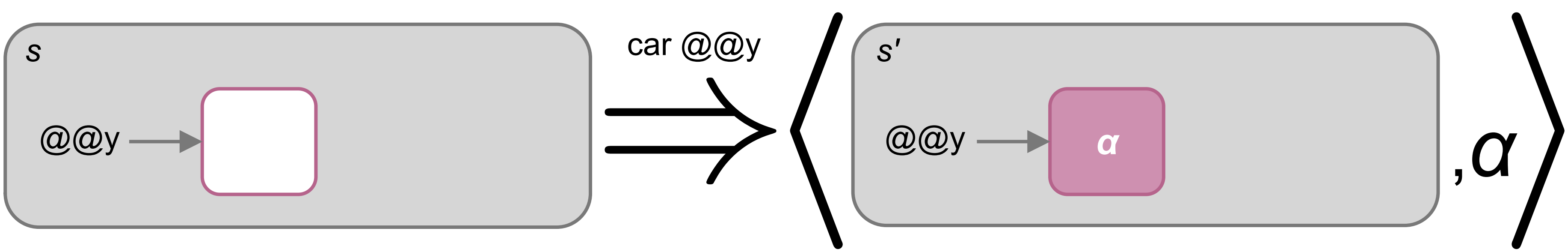
Integer List Extension: `car e`

```
@TopLevel @ExpExt
def car : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, intDataFieldUri))
      (s, s.lookup(rv, intDataFieldUri))
    else {
      val (s2, alpha) = sei.freshKiasanValue(s, MyIntExtension.KINT_TYPE_URI)
      val s3 = s2.update(rv, intDataFieldUri, alpha)
      (s3, alpha)
    }
}
```

```
{ val (s, rv) = {
    val (s0, v) = state.newObject(lhid, intDataFieldUri -> CI(5), nextFieldUri -> NilValue)
    (s0.variable("@@y", v), v) }

  Evaluating expression "car @@y" on s gives "5" satisfying { r : (S, V) =>
    r.value is 5; r.state is s
  }
}
```

Integer List Extension: `car e`

```
@TopLevel @ExpExt
def car : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, intDataFieldUri))
      (s, s.lookup(rv, intDataFieldUri))
    else {
      val (s2, alpha) = sei.freshKiasanValue(s, MyIntExtension.KINT_TYPE_URI)
      val s3 = s2.update(rv, intDataFieldUri, alpha)
      (s3, alpha)
    }
}
```



Integer List Extension: `car e`

```
@TopLevel @ExpExt
def car : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, intDataFieldUri))
      (s, s.lookup(rv, intDataFieldUri))
    else {
      val (s2, alpha) = sei.freshKiasanValue(s, MyIntExtension.KINT_TYPE_URI)
      val s3 = s2.update(rv, intDataFieldUri, alpha)
      (s3, alpha)
    }
}
```

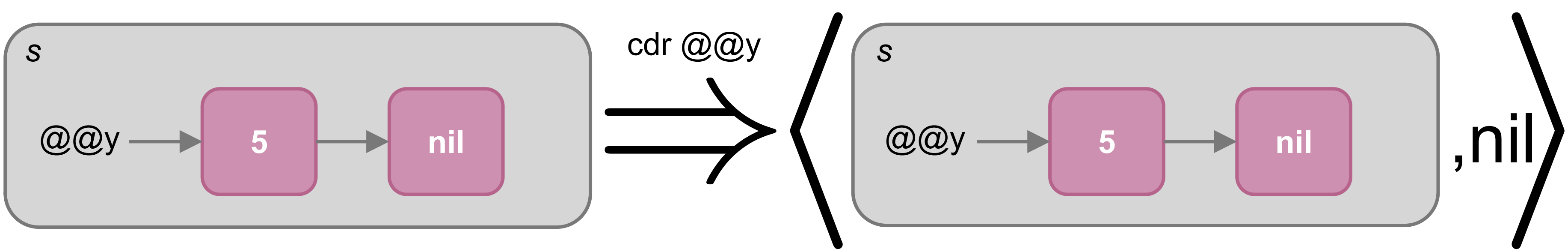
```
val alpha = KI(1)
Evaluating expression "car @@y" on
{ val (s, rv) = state.newObject(lhid); s.variable("@@y", rv) } gives "alpha" satisfying
{ r : (S, V) => val (s, rv) = (r.state, r.state.variable("@@y").asInstanceOf[ReferenceValue])
                s.lookup(rv, intDataFieldUri) is alpha
                s.hasFieldValue(rv, nextFieldUri) is false
                r.value is alpha                                                    }
```

Integer List Extension: `cdr e`

```
val sei = config.semanticsExtension.adapter[KiasanSemanticsExtensionConsumer[…]]

@TopLevel @ExpExt
def cdr : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, nextFieldUri))
      (s, s.lookup(rv, nextFieldUri))
    else {  val (newS, rv2) = s.newObject(lhid)
            ilist((s.update(rv, nextFieldUri, NilValue), NilValue),
                  (newS.update(rv, nextFieldUri, rv2), rv2)          )}
}
```

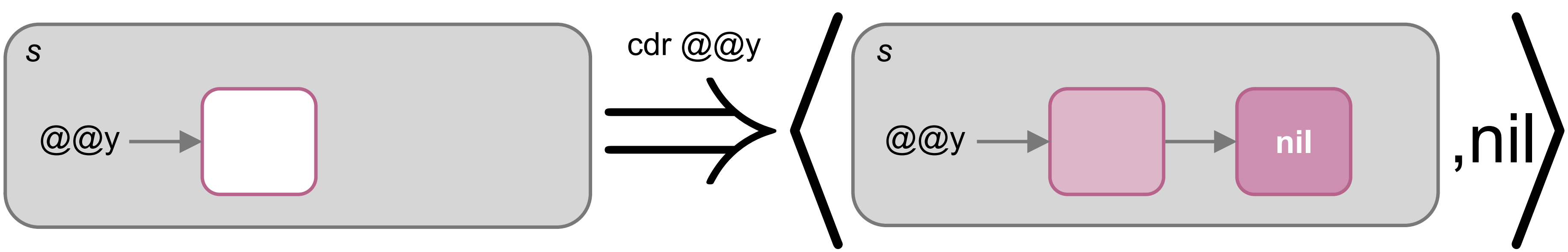
Integer List Extension: `cdr e`

```
val sei = config.semanticsExtension.adapter[KiasanSemanticsExtensionConsumer[…]]

@TopLevel @ExpExt
def cdr : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, nextFieldUri))
      (s, s.lookup(rv, nextFieldUri))
    else {  val (newS, rv2) = s.newObject(lhid)
              ilist((s.update(rv, nextFieldUri, NilValue), NilValue),
                    (newS.update(rv, nextFieldUri, rv2), rv2)          )}
}
```



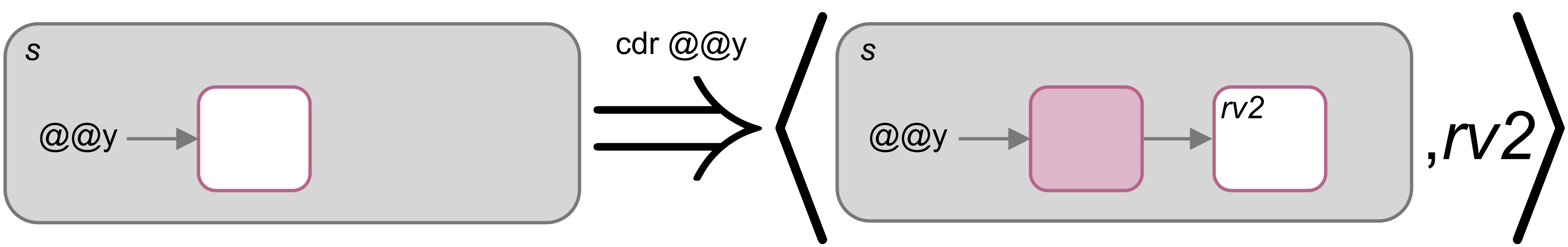
Integer List Extension: `cdr e`

```
val sei = config.semanticsExtension.adapter[KiasanSemanticsExtensionConsumer[…]]

@TopLevel @ExpExt
def cdr : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, nextFieldUri))
      (s, s.lookup(rv, nextFieldUri))
   else {  val (newS, rv2) = s.newObject(lhid)
             ilist((s.update(rv, nextFieldUri, NilValue), NilValue),
                (newS.update(rv, nextFieldUri, rv2), rv2)         )}
}
```

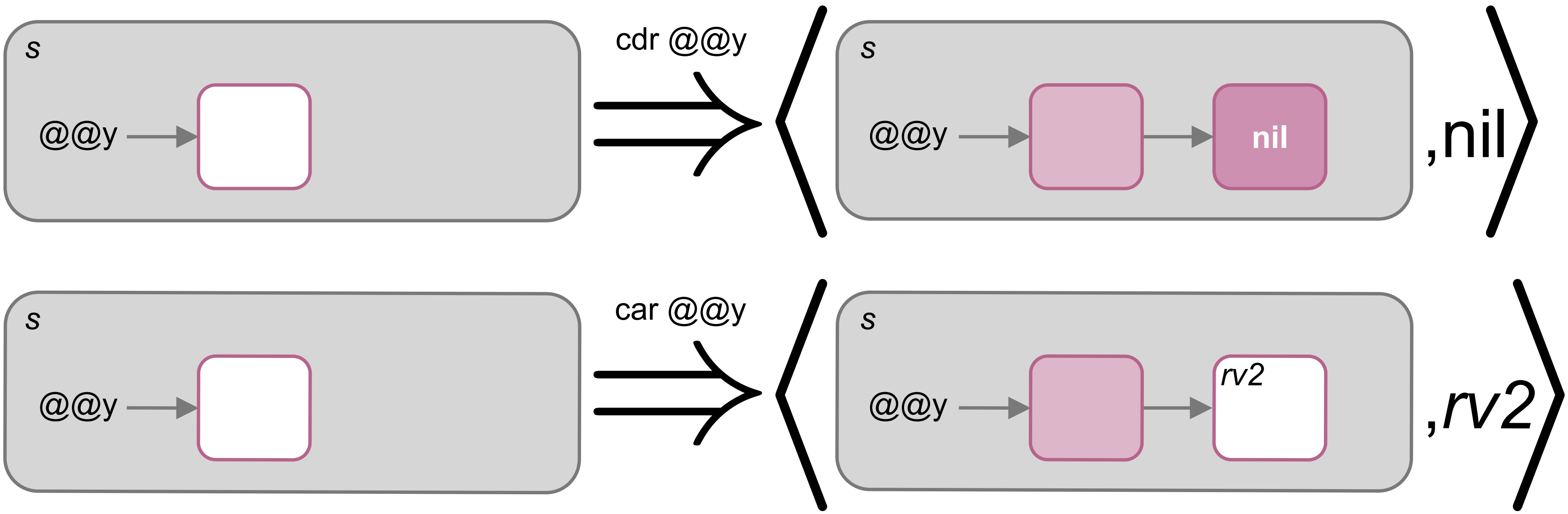


Integer List Extension: `cdr e`

```
val sei = config.semanticsExtension.adapter[KiasanSemanticsExtensionConsumer[…]]

@TopLevel @ExpExt
def cdr : (S, Value) --> ISeq[(S, Value)] = {
  case (s, rv @ Heap.RV(hid, _)) if hid == lhid =>
    if (s.hasFieldValue(rv, nextFieldUri))
      (s, s.lookup(rv, nextFieldUri))
   else {  val (newS, rv2) = s.newObject(lhid)
              ilist((s.update(rv, nextFieldUri, NilValue), NilValue),
                    (newS.update(rv, nextFieldUri, rv2), rv2)          )}
}
```



Integer List Extension: `cdr e`



Evaluating expression "cdr @@y" on

```
{ val (s, rv) = state.newObject(lhid); s.variable("@@y", rv) } gives "nil,node" satisfying
{ r : (S, V) => val (s, rv) = (r.state, r.state.variable("@@y").asInstanceOf[ReferenceValue])
   s.hasFieldValue(rv, intDataFieldUri) is false
   r.value match { case NilValue                  => s.lookup(rv, nextFieldUri) is NilValue
                   case rv2 : ReferenceValue => s.lookup(rv, nextFieldUri) is rv2
                                                s.hasFieldValue(rv2, nextFieldUri) is false
                                                s.hasFieldValue(rv2, intDataFieldUri) is false } }
```

Integer List Extension: `cdr e`

There are more cases to handle...

- termination
- aliasing, sharing of list tails
- cyclicity

For You To Do

- Implement integer list extensions for
  - the plus binary operator, e.g., `` 1 + `[1, 2] ``
  - looking up an element at a certain index
  - updating an element at a certain index

- For the last two, think about
  - when both the list and index are concrete
  - when the list is symbolic, but index is concrete
  - when the list is concrete, but index is symbolic
  - when both the list and index are symbolic

Kiasan Evaluators and Search

- provides evaluations for general Pilar transitions (jumps, procedure calls, etc.)
  - extensions, extensions, extensions!

- SymExe computation tree exploration
  - breadth-first search (BFS)
  - worklist algorithm
  - highly-parallel
  - distributable support (near future)

Kiasan BFS

```
trait KiasanBfs[S <: Kiasan.KiasanState[S], R] extends Kiasan {
  …
  def evaluator : Evaluator[S, R, ISeq[S]]
  def initialStatesProvider : KiasanInitialStateProvider[S]
  def reporter : KiasanReporter[S]

  def search {
    var workList : GenSeq[S] = initialStatesProvider.initialStates
    while (!workList.isEmpty) {
      val ps = inconNextStatesPairs(workList)
      val inconsistencyCheckRequested = ps.exists(first2)
      val nextStates = ps.flatMap(second2)
      workList = filterTerminatingStates(par(inconsistencyCheckRequested, nextStates))
    }
  }
  …
}
```

Summary

Kiasan provides a framework to build highly-parallel SymExe engines

- modular architecture
- no built-in interpretations/semantics
- easy to add semantics through extensions
- customizable to use different constraint solvers
- customizable to various application domains
- future: easy-to-deploy from laptops to clusters