Welder-Inox Documentation

Release 1

Rodrigo Raya

May 01, 2017

Contents:

CHAPTER

ONE

DEFINITIONS

1.1 Flags

The current implementation of Inox supports including certain flags while defining types.

```
abstract class Flag(val name: String, args: Seq[Any]) extends Printable
case class Variance(variance: Boolean) extends Flag("variance", Seq(variance))
case class HasADTInvariant(id: Identifier) extends Flag("invariant", Seq(id))
case class HasADTEquality(id: Identifier) extends Flag("equality", Seq(id))
case class Annotation(override val name: String, val args: Seq[Any]) extends Flag(name, args)
```

The flag Variance determines the variance of a [[Types.TypeParameter TypeParameter]] and should only be attached to those.

The flag HasADTInvariant denotes that the ADT is refined by invariant ‘’id’‘.

The flag HasADTEquality denotes that the ADT has an overriden equality relation given by ‘’id’‘.

Finally, the flag Annotation denotes compiler annotations given in the source code as @annot.

The flag HasADTInvariant allows us to do useful things. For instance, we can define ADT types from other types. Here is an example on how we would do that for constructing natural numbers from the IntegerType type:

```
val natural = FreshIdentifier("nat")
val element = FreshIdentifier("e")
val naturalInvariant = FreshIdentifier("natinvariant")

val naturalADT = new ADTConstructor(natural,Seq(),None,Seq(
                      ValDef(element, IntegerType),Set(HasADTInvariant(naturalInvariant)))

val naturalInvariantFunction: FunDef = mkFunDef(naturalInvariant)(){ case Seq() =>
  val args: Seq[ValDef] = Seq("i" :: T(natural)())
  val retType: Type = BooleanType
  val body: Seq[Variable] => Expr = { case Seq(i) =>
    i.getField(element) >= E(BigInt(0))
  }

  (args, retType, body)
}
```

1.2 ADT definition

Algebraic datatype definitions can be recursive and support type-parametric polymorphism, as in the named function case.

The following trait represents an ADT definition

```
sealed trait ADTDefinition extends Definition{
  val id: Identifier
  val tparams: Seq[TypeParameterDef]
  val flags: Set[Flag]

  def root(implicit s: Symbols): ADTDefinition // root of the class hierarchy
  def isInductive(implicit s: Symbols): Boolean
  def isWellFormed(implicit s: Symbols): Boolean
  def invariant(implicit s: Symbols): Option[FunDef] // An invariant that refines this [[ADTDefinitio
  def hasInvariant(implicit s: Symbols): Boolean // checks if this [[ADTDefinition]] has a defined in
  def equality(implicit s: Symbols): Option[FunDef] //An equality relation defined on this [[ADTDefin
  def hasEquality(implicit s: Symbols): Boolean // checks if this [[ADTDefinition]] has a defined equ
  val isSort: Boolean //checks if this [[ADTDefinition]] has a Sort
  def typeArgs
  def typed(tps: Seq[Type])(implicit s: Symbols): TypedADTDefinition
  def typed(implicit s: Symbols): TypedADTDefinition
}
```

An ADT can either correspond to a sort, or a constructor.

```
class ADTSort(
  val id: Identifier,                 /* The symbol associated with this ADT sort. */
  val tparams: Seq[TypeParameterDef], /* The ADT's type parameters. */
  val cons: Seq[Identifier],          /* The symbols of the sort's constructors. */
  val flags: Set[Flag]                /* Annotations associated to this definition. */)

class ADTConstructor(
  val id: Identifier,                 /* The symbol associated with this ADT sort. */
  val tparams: Seq[TypeParameterDef], /* The ADT's type parameters. */
  val sort: Option[Identifier],       /* The symbol of the constructor's (optional) sort. */
  val fields: Seq[ValDef],            /* The fields associated to this constructor. */
  val flags: Set[Flag]                /* Annotations associated to this definition. */)
```

Sorts can be seen as abstract classes with constructors resembling final classes as follows

```
abstract class id(tparams)
final case class cons[0](fields: tparams) extends id
...
```

1.3 Typed ADT definitions

Inox provides the utility types TypedADTSort and TypedADTConstructor (see file inox/ast/Definitions scala) that correspond to ADT definitions whose type parameters have been instantiated with concrete types. One can use these to access parameters, fields and enclosed expressions with instantiated types.

```
/** Represents an [[ADTDefinition]] whose type parameters have been instantiated to ''tps'' */
sealed abstract class TypedADTDefinition extends Tree

/** Represents an [[ADTSort]] whose type parameters have been instantiated to ''tps'' */
case class TypedADTSort(definition: ADTSort, tps: Seq[Type])(implicit val symbols: Symbols) extends T

/** Represents an [[ADTConstructor]] whose type parameters have been instantiated to ''tps'' */
case class TypedADTConstructor(definition: ADTConstructor, tps: Seq[Type])(implicit val symbols: Symb
```

1.4 Function definition

In inox/ast/Definitions.scala we can see the following definition

```
class FunDef(
  val id: Identifier,                  //The identifier which will refer to this function.
  val tparams: Seq[TypeParameterDef], //The type parameters this function takes.
  val params: Seq[ValDef],            //The functions formal arguments.
  val returnType: Type,               //The function's return type.
  val fullBody: Expr,                 //The body of this function.
  val flags: Set[Flag]                //Flags that annotate this function with attributes.
) extends Definition
```

We have to note that params' type and the return type may depend on [[tparams]]. In any case, FunDef is not a Expression but a Definition.

```
def mkFunDef(id: Identifier)
   (tpNames: String*)
   (builder: Seq[TypeParameter] => (Seq[ValDef], Type, Seq[Variable] => Expr)): FunDef
```

```
val sum = FreshIdentifier("sum")

val sumFunction = mkFunDef(sum)() { case _ =>
  val args: Seq[ValDef] = Seq("n" :: IntegerType)
  val retType: Type = IntegerType
  val body: Seq[Variable] => Expr = { case Seq(n) =>
    if_ (n === E(BigInt(0))) {
      E(BigInt(0))
    } else_ {
      val predN = n - E(BigInt(1))
      E(sum)(predN) + n
    }
  }
  (args, retType, body)
}
```

```
val contentFunction = mkFunDef(contentID)("A") { case Seq(aT) =>
  val args: Seq[ValDef] = Seq("l" :: T(list)(aT))
  val retType: Type = SetType(aT)
  val body: Seq[Variable] => Expr = { case Seq(l) =>
    if_ (l.isInstOf(T(cons)(aT))) {
      SetAdd(E(contentID)(aT)(l.asInstOf(T(cons)(aT)).getField(tail)), l.asInstOf(T(cons)(aT)).getFie
    } else_ {
      FiniteSet(Seq.empty, aT)
    }
  }
  (args, retType, body)
}
```

1.5 Typed function definition

```
/** Represents a [[FunDef]] whose type parameters have been instantiated with the specified types */
case class TypedFunDef(fd: FunDef, tps: Seq[Type])(implicit val symbols: Symbols) extends Tree
```

1.6 Valdef

See file inox/ast/Definitions.scala

Inox uses the Identifier type to represent symbols in the program. Note that no type is attached to an instance of Identifier. Values and variables will attach a type to their identifier, and function invocations will rely on the symbol table to compute their type.

class ValDef(v: Variable) extends Definition with VariableSymbol

A ValDef declares a formal parameter (with symbol [[id]]) to be of a certain type. It has its counterpart expression in the Variable construct. (See inox/Expressions.scala)

CHAPTER

TWO

TYPES

This file describes inox/ast/Types.scala.

The types that Inox admits go as follows:

```
abstract class Type extends Tree with Typed
case object Untyped     extends Type
case object BooleanType extends Type
case object UnitType    extends Type
case object CharType    extends Type
case object IntegerType extends Type
case object RealType    extends Type
case object StringType  extends Type
case class BVType(size: Int) extends Type
object Int32Type extends BVType(32)
case class TypeParameter(id: Identifier, flags: Set[Flag]) extends Type
case class TupleType(bases: Seq[Type]) extends Type
case class SetType(base: Type) extends Type
case class BagType(base: Type) extends Type
case class MapType(from: Type, to: Type) extends Type
case class FunctionType(from: Seq[Type], to: Type) extends Type
case class ADTType(id: Identifier, tps: Seq[Type]) extends Type
```

Here the Typed trait is defined as follows:

```
trait Typed extends Printable {
  def getType(implicit s: Symbols): Type
  def isTyped(implicit s: Symbols): Boolean = getType != Untyped
}
```

CHAPTER THREE

EXPRESSIONS

See inox/Expressions.scala

```
/** Represents an expression in Inox. */
abstract class Expr extends Tree with Typed
```

An expression in Inox can be one of the following:

3.1 Assume expression

```
case class Assume(pred: Expr, body: Expr) extends Expr with CachingTyped
```

Represents a local assumption where pred is the predicate to be assumed and body is the expression following the statement *assume(pred)*

3.2 Variable expression

```
case class Variable(id: Identifier, tpe: Type, flags: Set[Flag]) extends Expr with Terminal with Var
```

Represents a variable given by identifier *id*.

3.3 Let expression

```
case class Let(vd: ValDef, value: Expr, body: Expr) extends Expr with CachingTyped
```

This is the encoding of a statement of the form *val ...* = ...*;* .... Here vd is the ValDef used in body, defined just after *val*. value is the value assigned to the identifier, after the = sign. And body is the expression that follows to *val ...* = ... *;*. See also [[SymbolOps.let the let constructor]].

3.4 Choose expression

This encodes the *choose()* statement which returns a value satisfying the provided predicate.

```
case class Choose(res: ValDef, pred: Expr) extends Expr with CachingTyped
```

3.5 Expressions for control flow

```
case class FunctionInvocation(id: Identifier, tps: Seq[Type], args: Seq[Expr]) extends Expr with Cach
case class IfExpr(cond: Expr, thenn: Expr, elze: Expr) extends Expr with CachingTyped
```

The class for IfExpr computes the type as the least upper bound of the then and the else part.

3.6 Other expressions

```
case class Application(callee: Expr, args: Seq[Expr]) extends Expr with CachingTyped
case class Lambda(args: Seq[ValDef], body: Expr) extends Expr with CachingTyped
case class Forall(args: Seq[ValDef], body: Expr) extends Expr with CachingTyped
case class ADT(adt: ADTType, args: Seq[Expr]) extends Expr with CachingTyped
case class IsInstanceOf(expr: Expr, tpe: Type) extends Expr with CachingTyped
case class AsInstanceOf(expr: Expr, tpe: Type) extends Expr with CachingTyped
```

3.7 isInstanceOf and asInstanceOf

```
In inox/SymbolOps.scala

/** $encodingof expr.asInstanceOf[tpe],
returns `expr` if it already is of type `tpe`.  */
def asInstOf(expr: Expr, tpe: Type) = {
  if (symbols.isSubtypeOf(expr.getType, tpe)) {
    expr
  } else {
    AsInstanceOf(expr, tpe)
  }
}

def isInstOf(expr: Expr, tpe: Type) = {
  if (symbols.isSubtypeOf(expr.getType, tpe)) {
    BooleanLiteral(true)
  } else {
    IsInstanceOf(expr, tpe)
  }
}
```

3.8 Literals

```
sealed abstract class Literal[+T] extends Expr with Terminal
case class CharLiteral(value: Char) extends Literal[Char] //character literal
case class BVLiteral(value: BitSet, size: Int) extends Literal[BitSet] //bit-vector literal
case class IntegerLiteral(value: BigInt) extends Literal[BigInt] //infinite precision integer literal
case class FractionLiteral(numerator: BigInt, denominator: BigInt) extends Literal[(BigInt, BigInt) /
case class BooleanLiteral(value: Boolean) extends Literal[Boolean] // boolean literal
case class UnitLiteral() extends Literal[Unit] // unit literal
case class StringLiteral(value: String) extends Literal[String] // string literal
```

3.9 Expressions for propositional logic

```
case class And(exprs: Seq[Expr]) extends Expr with CachingTyped // &&
case class Or(exprs: Seq[Expr]) extends Expr with CachingTyped // ||
case class Implies(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // ==>
case class Not(expr: Expr) extends Expr with CachingTyped // !
```

3.10 Expressions for string theory

```
case class StringConcat(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // +
case class SubString(expr: Expr, start: Expr, end: Expr) extends Expr with CachingTyped
case class StringLength(expr: Expr) extends Expr with CachingTyped
```

3.11 Expressions for general arithmetic

```
case class Plus(lhs: Expr, rhs: Expr) extends Expr with CachingTyped //+
case class Minus(lhs: Expr, rhs: Expr) extends Expr with CachingTyped //-
case class UMinus(expr: Expr) extends Expr with CachingTyped // - for BigInt
case class Times(lhs: Expr, rhs: Expr) extends Expr with CachingTyped //*
case class Division(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // /
case class Remainder(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // %
case class Modulo(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // mod
case class LessThan(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // <
case class GreaterThan(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // >
case class LessEquals(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // <=
case class GreaterEquals(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // >=
```

3.12 Expressions for bit-vectors operations

```
case class BVNot(e: Expr) extends Expr with CachingTyped // ~
case class BVOr(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // |
case class BVAnd(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // &
case class BVXor(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // ^
case class BVShiftLeft(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // <<
case class BVAShiftRight(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // >>
case class BVLShiftRight(lhs: Expr, rhs: Expr) extends Expr with CachingTyped // >>>
```

3.13 Expressions for tuples operations

```
case class Tuple(exprs: Seq[Expr]) extends Expr with CachingTyped // (..., ....)
case class TupleSelect(tuple: Expr, index: Int) extends Expr with CachingTyped // (tuple)._i
```

3.14 Expressions for sets operations

```
case class FiniteSet(elements: Seq[Expr], base: Type) extends Expr with CachingTyped // Set[base](ele
case class SetAdd(set: Expr, elem: Expr) extends Expr with CachingTyped // set + elem
case class ElementOfSet(element: Expr, set: Expr) extends Expr with CachingTyped // set.contains(elem
case class SubsetOf(set1: Expr, set2: Expr) extends Expr with CachingTyped // set.subsetOf(set2)
case class SetIntersection(set1: Expr, set2: Expr) extends Expr with CachingTyped // set & set2
case class SetUnion(set1: Expr, set2: Expr) extends Expr with CachingTyped // set ++ set2
case class SetDifference(set1: Expr, set2: Expr) extends Expr with CachingTyped // set -- set2
```

3.15 Expressions for bags operations

```
case class FiniteBag(elements: Seq[(Expr, Expr)], base: Type) extends Expr // Bag[base](elements)
case class BagAdd(bag: Expr, elem: Expr) extends Expr with CachingTyped // bag + elem
case class MultiplicityInBag(element: Expr, bag: Expr) extends Expr with CachingTyped // bag.get(elem
case class BagIntersection(bag1: Expr, bag2: Expr) extends Expr with CachingTyped // bag1 & bag2
case class BagUnion(bag1: Expr, bag2: Expr) extends Expr with CachingTyped // bag1 ++ bag2
case class BagDifference(bag1: Expr, bag2: Expr) extends Expr with CachingTyped // bag1 -- bag2
```

3.16 Expressions for total map operations

```
 case class FiniteMap(pairs: Seq[(Expr, Expr)], default: Expr, keyType: Type, valueType: Type) extend
 case class MapApply(map: Expr, key: Expr) extends Expr with CachingTyped
case class MapUpdated(map: Expr, key: Expr, value: Expr) extends Expr with CachingTyped
```

CHAPTER

FOUR

ERRORS

```
welder.Theory$AttemptException: TypeError(prove,<untyped>)

 p.asInstanceOf[finitePoint[element]].second * p.asInstanceOf[finitePoint[element]].second ==

 (p.asInstanceOf[finitePoint[element]].first * p.asInstanceOf[finitePoint[element]].first) * p.asInst

 No reconocia el tipo de retorno de una funcion.
```

Encontramos el siguiente error:

```
java.lang.ExceptionInInitializerError
at Main$.main(Main.scala:4)
at Main.main(Main.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
```

Caused by: welder.Theory$AttemptException: TypeError(prove,<untyped>) at welder.Theory$class.attemptToValue(Theory.scala:352) at welder.package$$anon$1.attemptToValue(package.scala:10) at Curve$.<init>(curve.scala:154) at Curve$.<clinit>(curve.scala) at Main$.main(Main.scala:4) at Main.main(Main.scala) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at java.lang.reflect.Method.invoke(Method.java:498)

Cuando definimos en el archivo curve.scala:

```
def opp(e: Expr): Expr = E(Field.oppositeID)(F)(e)
val z = T(Field.zero)()()
implicit class ExprOps(private val lhs: Expr) extends AnyVal{
  def +(rhs: Expr): Expr = E(Field.addID)(F)(lhs, rhs)
  def *(rhs: Expr): Expr = E(Field.multiplicationID)(F)(lhs, rhs)
  def /(rhs: Expr): Expr = E(Field.inverseID)(F)(lhs, rhs)
  def -(rhs: Expr): Expr = E(Field.addID)(F)(lhs, E(Field.oppositeID)(F)(rhs))
}

def double(e: Expr): Expr = e + e
def triple(e: Expr): Expr = e + e + e
```

Mientras que situamos las funciones:

```
implicit class ExprOperands(private val lhs: Expr) extends AnyVal{
  def +(rhs: Expr): Expr = E(addID)(T(element)())(lhs, rhs)
  def *(rhs: Expr): Expr = E(multiplicationID)(T(element)())(lhs, rhs)
  def /(rhs: Expr): Expr = E(inverseID)(T(element)())(lhs, rhs)
  def -(rhs: Expr): Expr = E(addID)(T(element)())(lhs, E(oppositeID)(T(element)())(rhs))
}
```

en el archivo field.scala no encontramos ningun error.

CHAPTER

FIVE

DSL

5.1 Type invocation through the DSL

```
def T(tp1: Type, tp2: Type, tps: Type*) = TupleType(tp1 :: tp2 :: tps.toList)
def T(id: Identifier) = new IdToADTType(id)
def T(str: String) = TypeParameter.fresh(str)

class IdToADTType(id: Identifier) {
  def apply(tps: Type*) = ADTType(id, tps.toSeq)
}
```

The way to declare types corresponding to ADT types is therefore ‘T(identifierOfType)(type parameters)

5.2 Function invocation through the DSL

We can invoke a function using the DSL using the E() construct. Note that in any case DSL will give back a Function-Invocation which is an Expression.

```
def E(id: Identifier) = new IdToFunInv(id)
class IdToFunInv(id: Identifier) {
  def apply(tp1: Type, tps: Type*)(args: Expr*) =
    FunctionInvocation(id, tp1 +: tps.toSeq, args.toSeq)
  def apply(args: Expr*) =
    FunctionInvocation(id, Seq.empty, args.toSeq)
}
```

If function has arguments and type parameters then call

```
E(functionID)(tps)(args)
```

If function has only arguments then call

```
E(functionID)(args)
```

Don’t call:

```
E(functionID)()(args)
```

5.3 ADT definitions through the DSL

```
def mkSort(id: Identifier)
  (tpNames: String*)
  (cons: Seq[Identifier]): ADTSort

def mkSort(id: Identifier, flags: Flag*)
          (tParamNames: String*)
          (cons: Seq[Identifier]) = {
  val tParams = tParamNames map TypeParameter.fresh
  val tParamDefs = tParams map (TypeParameterDef(_))
  new ADTSort(id, tParamDefs, cons, flags.toSet)
}

def mkConstructor(id: Identifier)
  (tpNames: String*)
  (sort: Option[Identifier])
  (fieldBuilder: Seq[TypeParameter] => Seq[ValDef]): ADTConstructor

def mkConstructor(id: Identifier, flags: Flag*)
                 (tParamNames: String*)
                 (sort: Option[Identifier])
                 (fieldBuilder: Seq[TypeParameter] => Seq[ValDef]) = {
  val tParams = tParamNames map TypeParameter.fresh
  val tParamDefs = tParams map (TypeParameterDef(_))
  val fields = fieldBuilder(tParams)
  new ADTConstructor(id, tParamDefs, sort, fields, flags.toSet)
}
```

The problem with this implementation is that some class hierarchies cannot be implemented directly in Inox. For details see “Modelling a class hierarchy in Inox” on SO. Here is an example of class hierarchy that cannot be modelled:

```
abstract class Element()
abstract class nonZero() extends Element
final case class Zero() extends Element
final case class One() extends nonZero()
final case class notOne() extends nonZero()
```

5.4 If-else construct through the DSL

To have an if-expr we require to provide an *else_*. This is the dangling else problem in which one could have ambiguity if no else were required. More details for *IfExpr* in Expressions section.

```
class DanglingElse private[DSL] (theCond: Expr, theThen: Expr) {
  def else_ (theElse: Expr) = IfExpr(theCond, theThen, theElse)
}

def if_ (cond: Expr)(thenn: Expr) = new DanglingElse(cond, thenn)
```

5.5 Expressions for arithmetic, comparisons and boolean from the DSL

It is important to know what expressions the DSL defines in order to know what expressions we can override in our programs.

```
def + = Plus(e, _: Expr)
def - = Minus(e, _: Expr)
def * = Times(e, _: Expr)
def % = Modulo(e, _: Expr)
def / = Division(e, _: Expr)
def unary_- = UMinus(e)

def <   = LessThan(e, _: Expr)
def <=  = LessEquals(e, _: Expr)
def >   = GreaterThan(e, _: Expr)
def >=  = GreaterEquals(e, _: Expr)
def === = Equals(e, _: Expr)
def !== = (e2: Expr) => Not(Equals(e, e2))

def &&  = And(e, _: Expr)
def ||  = Or(e, _: Expr)
def ==> = Implies(e, _: Expr)
def unary_! = Not(e)
```

5.6 Expressions for tuples, sets and ADTTypes from the DSL

Here are expressions to deal with tuples, sets and ADTTypes.

```
def _1 = TupleSelect(e, 1)
def _2 = TupleSelect(e, 2)
def _3 = TupleSelect(e, 3)
def _4 = TupleSelect(e, 4)

def subsetOf = SubsetOf(e, _: Expr)
def insert   = SetAdd(e, _: Expr)
def ++ = SetUnion(e, _: Expr)
def -- = SetDifference(e, _: Expr)
def &  = SetIntersection(e, _: Expr)
def contains = ElementOfSet(_: Expr, e)

def isInstOf(tp: ADTType) = IsInstanceOf(e, tp)
def asInstOf(tp: ADTType) = AsInstanceOf(e, tp)
def getField(selector: Identifier) = ADTSelector(e, selector)
object C {
  def unapplySeq(expr: Expr): Option[(Identifier, Seq[Expr])] = expr match {
    case ADT(adt, exprs) => Some((adt.id, exprs))
    case _ => None
  }
}
```

The last object defines an ADT extractor object. See paper “ Matching objects with patterns ”.

CHAPTER

SIX

WELDER

6.1 welder/Solvers.scala

Looking at file welder/Solvers.scala reveals what welder is doing when it calls an Inox solver with a Theory. Basically it defines the following three functions:

```
def prove(expr: Expr): Attempt[Theorem]
def prove(expr: Expr, first: Theorem, rest: Theorem*): Attempt[Theorem]
def prove(expr: Expr, assumptions: Seq[Theorem]): Attempt[Theorem]
```

Where the first two are just rewrittings of the third one who does the real work. It first ensures that the type of the expression to be proven is *BooleanType*. It then constructs the formula

Not(h1 ^ h2 ^ ... ^ hn ==> expr)

where the hi are the expressions corresponding to the assumptions theorems. It checks that this formula is well constructed and then calls to a solver setting its time out. Of course if this formula is not satifaisable then we know that our goal is valid. In other case we inform of the error to the user.

```
val hypotheses = assumptions.map(_.expression)
val negation = Not(Implies(and(hypotheses : _*), expr))
program.ctx.reporter.debug(negation)
val solver = factory.getNewSolver.setTimeout(5000L)
try {
  solver.assertCnstr(negation)
  val result = solver.check(SolverResponses.Model) // TODO: What to do with models?

  program.ctx.reporter.debug(result)

  if (result.isUNSAT) {
    // Impossible to satisfy the negation of the expression,
    // thus the expression follows from the assumptions.
    Attempt.success(new Theorem(expr).from(assumptions))
  } else {
    Attempt.fail("SMT solver could not prove the property: " + expr + ", from hypotheses: " + hypothe
  }
}
```

So at the very bottom level, Welder is just a simplifier that can modify the structure of the expression we want to proof applying logical transformations. At the bottom level, the system is going to feed Inox with an expression that he needs to prove using its particular unrolling algorithm.

6.2 welder/Theory.scala

Using the cake pattern we put together all the elements of the implementation.

It manages the theorems proved by the systems. A *Theorem* is a wrapper around expressions of type *BooleanType* . Users of the library can not build a *Theorem* using the constructor directly. Instead, they must rely on the combinators provided by the various mixed-in traits.

```
class Theorem private[welder] (val expression: Expr,
                               private[welder] val markings: Set[Mark] = Set())
```

Having a theorem asserts the given *expression* as true in some scope. We also get a set of markings, indicating in which scopes this theorem is valid. An empty set indicates that this theorem is valid in the global scope. There are some tools to deal with the markings:

```
private[welder] type Mark = BigInt
private[welder] def mark: (Theorem, Mark)
private[welder] def mark(marking: Mark): Theorem
private[welder] def unmark(marking: Mark): Theorem
private[welder] def unmark(markings: Seq[Mark]): Theorem
def isGloballyValid: Boolean
```

The construct from indicates that this Theorem is derived from other theorems. The resulting Theorem is additionally marked by all markings of the others theorems. isGloballyValid checks if this *Theorem* is valid in the global scope. Any time that we use a deduction rule we will call *from* in order to mark the new theorem with the marks of the previous theorems:

```
private[welder] def from(that: Theorem): Theorem =
  new Theorem(expression, markings ++ that.markings)
private[welder] def from(those: Seq[Theorem]): Theorem = {
  if (those.isEmpty) this
  else new Theorem(expression, markings ++ those.map(_.markings).reduce(_ ++ _))
}
```

The file further defines the derivations rules that can be found in the Handbook of Practical Logic by John Harrison. This rules are implemented in the file welder/Rules.scala

```
def andE: Attempt[Seq[Theorem]] // Conjunction elimination.
def implE(proof: Goal => Attempt[Witness]): Attempt[Theorem] // Implication elimination
def forallE(first: Expr, rest: Expr*): Attempt[Theorem] // Universal quantification elimination
def existsI(path: Path, name: String): Attempt[Theorem] // Existantial quantification introduction
def existsI(expr: Expr, name: String): Attempt[Theorem] // Existantial quantification introduction
def existsE: Attempt[(Variable, Theorem)] // Existantial quantification elimination
def orE(conclusion: Expr)
  (cases: (Theorem, Goal) => Attempt[Witness]): Attempt[Theorem] // Disjunction elimination
def notE: Attempt[Theorem] // Negation elimination
def instantiateType(tpeParam: TypeParameter, tpe: Type): Theorem // Type instantiation
```

6.2.1 Goals

```
class Goal(val expression: Expr)
```

Syntax

Goals are wrappers around BooleanType expressions.

Semantic

A goal indicates the current expression to be proven.

Details on implementation

They are never created by the user of the library, but are often passed as arguments to high order functions. For instance in the call to negation introduction:

```
def notI(hypothesis: Expr)(contradiction: (Theorem, Goal) => Attempt[Witness]): Attempt[Theorem]
```

It basically contains two methods to prove it.

```
def by(theorem: Theorem): Attempt[Witness]
```

The by method tries to proof the goal using a theorem. If the expression of the theorem and the goal are the same, it produces a witness that certifies that the theorem was proven. Else it calls prove on the expression using the provided theorem as a hint and producing the corresponding witnesses.

```
def trivial: Attempt[Witness] = by(truth)
```

The trivial method just tries to solve the goal without any other assumptions.

6.2.2 Witnesses

```
trait Witness {
  def extractTheorem(goal: Goal): Attempt[Theorem]
}
private class ActualWitness(theorem: Theorem) extends Witness
```

6.2.3 Attempt

```
sealed abstract class Attempt[+A]
case class Success[A](value: A) extends Attempt[A]
case class Failure(reason: FailureReason) extends Attempt[Nothing]
```

Semantic

Represents possibly failing computations. In case of non-successful attempts, returns the reasons of failure.

6.3 welder/Rules.scala

This file contains introduction and elimination rules for many constructions.

6.3.1 Inference rules without quantifiers

```
def notI(hypothesis: Expr)(contradiction: (Theorem, Goal) => Attempt[Witness]): Attempt[Theorem]
def notE(thm: Theorem): Attempt[Theorem]
def andI(theorems: Seq[Theorem]): Theorem
def andI(first: Theorem, rest: Theorem*): Theorem
def andE(conjunction: Theorem): Attempt[Seq[Theorem]]
def orI(expressions: Seq[Expr])(cases: Goal => Attempt[Witness]): Attempt[Theorem]
def orI(first: Expr, rest: Expr*)(cases: Goal => Attempt[Witness]): Attempt[Theorem]
def orE(disjunction: Theorem, conclusion: Expr)(cases: (Theorem, Goal) => Attempt[Witness]): Attempt
def orToImpl(disjunction: Theorem): Attempt[Seq[Theorem]]
def implI(assumption: Expr)(conclusion: Theorem => Attempt[Theorem]): Attempt[Theorem]
def implE(implication: Theorem)(proof: Goal => Attempt[Witness]): Attempt[Theorem]
```

The construct *notI* proves the negation of some *hypothesis* by showing that it implies *false*. The idea is to proof the formula ∀ *x. (P[x] → false) →* ¬ *P[x]*. So basically, here the *proof* argument is a proof of *false*, given the *hypothesis*. The function returns a theorem for the negation of the hypothesis.

```
def notI(hypothesis: Expr)
        (contradiction: (Theorem, Goal) => Attempt[Witness]): Attempt[Theorem] = {

  val (hyp, mark) = new Theorem(hypothesis).mark
  val goal = new Goal(BooleanLiteral(false))

  catchFailedAttempts {
    for {
      witness <- contradiction(hyp, goal)
      theorem <- witness.extractTheorem(goal)
    } yield new Theorem(Not(hypothesis)).from(theorem).unmark(mark)
  }
}
```

The construct *andI* will generate a new theorem that is conjunction of its parameters and will mark this theorem with the marking of each of the conjunctions.

The construct *implI* proves that *assumption* implies one or more conclusions.

assumption is an expression of type *BooleanType*. The method returns the *Theorem* of the implication between the assumption and the attempted

by proving that the conclusion holds assuming the *assumption*. Here to assume. Should be *conclusion* is the derivation of a *Theorem*, assuming the *assumption*.

```
def implI(assumption: Expr)(conclusion: Theorem => Attempt[Theorem]): Attempt[Theorem] = {
  if (assumption.getType != BooleanType) return Attempt.fail(/*...*/)
  val (hypothesis, mark) = new Theorem(assumption).mark
  conclusion(hypothesis) map {
    (thm: Theorem) => new Theorem(Implies(assumption, thm.expression)).from(thm).unmark(mark)
  }
}
```

The construct *implE* given a proven *implication*, and a *proof* of the premise of the implication, returns the conclusion as a *Theorem*. The *implication* should be of the form *Implies(...)*.

6.3.2 Inference rules with quantifiers

```
def forallI(vd: ValDef)(theorem: Variable => Attempt[Theorem]): Attempt[Theorem]
def forallE(quantified: Theorem)(first: Expr, rest: Expr*): Attempt[Theorem]
def forallE(quantified: Theorem, terms: Seq[Expr]): Attempt[Theorem]
def existsI(expr: Expr, name: String)(theorem: Theorem): Attempt[Theorem]
def existsI(path: Path, name: String)(theorem: Theorem): Attempt[Theorem]
def existsE(quantified: Theorem): Attempt[(Variable, Theorem)]
```

The construct *forallI* has versions for up to four variable and then a version for a sequence of variables. Note that we have to provide a theorem in order to obtain a quantified theorem. The *forallE* takes a quantified theorem and then a sequence of terms to which we instantiate the rule. So basically, there is an obvious inference rule that allows to go from a *forall* quantified theorem into a theorem with instantiations.

The construct *existsI*

6.4 welder/Relational.scala

Welder provides an equational reasoning DSL similar to Stainless one (stainless/proof/package.scala). It allows chains of equalities to be written in the following way:

```
{{{
  expression1 ==| proof1is2 |
  expression2 ==| proof2is3 |
  expression3
}}}
```

6.4.1 Relations on Expressions

We state five types of binary relations on expressions:

```
sealed abstract class Rel
case object LE extends Rel
case object LT extends Rel
case object EQ extends Rel
case object GT extends Rel
case object GE extends Rel
```

together with a composition operation that always select the more exigent condition.

```
case (EQ, b) => Some(b)
case (a, EQ) => Some(a)
case (LT, LT | LE) => Some(LT)
case (LE, LT) => Some(LT)
case (LE, LE) => Some(LE)
case (GT, GT | GE) => Some(GT)
case (GE, GT) => Some(GT)
case (GE, GE) => Some(GE)
case _ => None
```

6.4.2 Deduction chains and nodes

Then we introduce a notion of chain of relations made up of nodes. This nodes are defined as follows:

```
trait Node {
  private[welder] val nextRel: Rel
  private[welder] val first: Expr //First expression
  private[welder] val next: Goal => Attempt[Witness] //Justification for the next step
  //Sets the next expression, and following justification, in the chain
  private[welder] def append(node: Node): Chain =
  // Sets the last expression in the equality chain
  private[welder] def append(end: Expr): Attempt[Theorem]
}
```

Then we model the notion of chain:

```
// Chain of equalities, with proof for the following equality
trait Chain extends Node {
  private[welder] val last: Expr // Last seen expression
  private[welder] val rel: Rel
  //Theorem stating the relation between the `first` and `last` expressions
  private[welder] def relation: Attempt[Theorem]
  // Sets the next expression, and following justification, in the equality chain
  override private[welder] def append(node: Node): Chain =
  // Sets the last expression in the equality chain
  override private[welder] def append(end: Expr): Attempt[Theorem]
}
```

We can use some implicit constructions to make proofs that will set the type of the chain we are working in and will set up the node of the current step.

```
// Adds a theorem or a proof as justification for the next equality
def ==|(theorem: Theorem): EqChain
def ==|(proof: Goal => Attempt[Witness]): EqChain

// Adds a theorem or proof as justification for the next relation
def >=|(theorem: Theorem): GreaterEqChain
def >=|(proof: Goal => Attempt[Witness]): GreaterEqChain
def >>|(theorem: Theorem): GreaterChain
def >>|(proof: Goal => Attempt[Witness]): GreaterChain
def <=|(theorem: Theorem): LessEqChain
def <=|(proof: Goal => Attempt[Witness]): LessEqChain
def <<|(theorem: Theorem): LessChain
def <<|(proof: Goal => Attempt[Witness]): LessChain
```

The trait HasNode models a logical step in our chains. Depending of the type of our chain we will be able to produce other type of chains. For instance, if we have an EqChain then we can construct any other type of chain. But, if we are given a LessChain we will be able to build just LessChains. Note the power of this approach that lets you combine chains of reasonings.

```
trait HasNode {
  val node: Node
  def |(end: Expr): Attempt[Theorem] = node.append(end)
}

class EqChain(val node: Node) extends HasNode {
  def |(chain: LessChain): LessChain =
  def |(chain: LessEqChain): LessEqChain
  def |(chain: EqChain): EqChain
  def |(chain: GreaterChain): GreaterChain
  def |(chain: GreaterEqChain): GreaterEqChain
}

class LessEqChain(val node: Node) extends HasNode {
  def |(chain: LessChain): LessChain
  def |(chain: LessEqChain): LessEqChain
  def |(chain: EqChain): LessEqChain
}
```

6.5 welder/Arithmetic.scala

This file contains methods and theorems related to *Arithmetic*. This will be included in the general *Theory* by means of the cake pattern:

Here is important to note that types are assumed to be numeric types from Inox. That is, they are either integer, real or bitvectors.

```
//Creates the corresponding theorem for arguments of type tpe on demand
def plusCommutativity(tpe: Type): Attempt[Theorem]
def plusAssociativity(tpe: Type): Attempt[Theorem]
def timesCommutativity(tpe: Type): Attempt[Theorem]
def timesAssociativity(tpe: Type): Attempt[Theorem]
def distributivity(tpe: Type): Attempt[Theorem]
//Proves the theorem (n+d)/d === (n/d) + 1 for integers
lazy val divisionDecomposition: Theorem

// Induction hypotheses for natural induction
trait NaturalInductionHypotheses {
  val variable: Variable                   //Inductive variable
  val variableGreaterThanBase: Theorem     //Theorem: inductive variable is greater than base case
  val propertyForVar: Theorem              //Theorem: property holds for the variable
  val propertyForLessOrEqualToVar:Theorem //Theorem: property holds for variable and smaller values
}
```

Now we can turn to the actual procedures to perform natural induction:

```
//Tries to prove a property by natural induction.
//property, the property to be proved.
//base, the base expression. Must be of type `IntegerType`.
//baseCase, the proof that property holds in the base case.
//inductiveCase, the proof that property holds in the inductive case assuming induction hypotheses.
// Returns a forall-quantified theorem of the property.

def naturalInduction(property: Expr => Expr, base: Expr, baseCase: Goal => Attempt[Witness])
    (inductiveCase: (NaturalInductionHypotheses, Goal) => Attempt[Witness]): Attempt[Theorem]
```

Steps in the function definition:

1. Proof base case by recovering the expression of property for base with a Goal construction and then applying the proof for the base case. Catch failed attempts.
2. Create marked induction hypothesis and construct the *NaturalInductionHypotheses* from them. Then formulate the required property for *n+1* and recover it as a Goal. Then try to use the inductive hypothesis on that goal assumming the inductionHypothesis. Quite interestingly, this hypothesis and goal are created internally so the user has no control on them.
3. Creates the corresponding theorem unmarkig when necessary.

A variation of the above may not require *property* to be a function from *Expr* to *Expr*:

```
def naturalInduction(property: Expr, base: Expr, baseCase: Goal => Attempt[Witness])
    (inductiveCase: (NaturalInductionHypotheses, Goal) => Attempt[Witness]): Attempt[Theorem]
```

It relies on the forallToPredicate construct of the *Theory.scala* file to call the basic natural induction method. It requires the expression to be of the kind ∀ *n: BigInt. ...*.

```
forallToPredicate(property, IntegerType) flatMap { (f: Expr => Expr) =>
  naturalInduction(f, base, baseCase)(inductiveCase)
}
```

CHAPTER

SEVEN

THE CAKE PATTERN

7.1 Self reference

You can use this in a method to refer to the enclosing instance, which is useful for referencing another member of the instance. Explicitly using this is not usually necessary for this purpose, but it 's occasionally useful for disambiguating a reference when several items are in scope with the same name.

Self-type annotations let you specify additional type expectations for this, and they can be used to create aliases for this. Let 's consider the latter case first:

```
class C1 { self =>
  def talk(message: String) = println("C1.talk: " + message)
  class C2 {
    class C3 {
      def talk(message: String) = self.talk("C3.talk: " + message)
    }
    val c3 = new C3
  }
  val c2 = new C2
}
val c1 = new C1
c1.talk("Hello")
c1.c2.c3.talk("World")
```

Prints:

```
C1.talk: Hello
C1.talk: C3.talk: World
```

We give the outer scope (C1) this the alias self, so we can easily refer to it in C3. We could use self within any method inside the body of C1 or its nested types. Note that the name self is arbitrary, but it is somewhat conventional. In fact, you could say this =>, but it would be completely redundant.

If the self-type annotation has types in the annotation, we get some very different benefits:

```
trait Persistence {def startPersistence: Unit}
trait Midtier {def startMidtier: Unit}
trait UI {def startUI: Unit}
trait Database extends Persistence {def startPersistence = println("Starting Database")}
trait ComputeCluster extends Midtier {def startMidtier = println("Starting ComputeCluster")}
trait WebUI extends UI {def startUI = println("Starting WebUI")}
trait App {
  self: Persistence with Midtier with UI =>
  def run = {
    startPersistence
    startMidtier
    startUI
  }
}
object MyApp extends App with Database with ComputeCluster with WebUI
MyApp.run
```

Each abstract trait declares a " start " method that does the work of initializing the tier. (We 're ignoring issues like success versus failure of startup, etc.) Each abstract tier is implemented by a corresponding concrete trait (not a class, so we can use them as mixins). We have traits for database persistence, some sort of computation cluster to do the heavy lifting for the business logic, and a web-based UI.

The App trait wires the tiers together. For example, it does the work of starting the tiers in the run method.

Note the self-type annotation, self: Persistence with Midtier with UI =>. It has two practical effects:

1. The body of the trait can assume it is an instance of Persistence, Midtier, and UI, so it can call methods defined in those types, whether or not they are actually defined at this point.
2. The concrete type that mixes in this trait must also mix in these three other traits or descendants of them.

In other words, the self type in App specifies dependencies on other components. These dependencies are satisfied in MyApp, which mixes in the concrete traits for the three tiers.

7.2 Cake pattern

The dependency injection (DI) pattern allows to avoid hard-coded dependencies and to substitute dependencies either at run-time or at compile time. The pattern is a special case of inversion of control (IoC) technique.

Dependency injection is used to choose among multiple implementations of a particular component in application, or to provide mock component implementations for unit testing.

Apart from IoC containers, the simplest Java implementation uses constructor arguments to pass required dependencies. So, we use composition to express dependency requirements:

```
public interface Repository {
  void save(User user);
}

public class DatabaseRepository implements Repository { /* ... */ }

public class UserService {
  private final Repository repository;

  UserService(Repository repository) {
      this.repository = repository;
  }

  void create(User user) {
      // ...
      repository.save(user);
  }
}

new UserService(new DatabaseRepository());
```

In addition to composition (" HAS-A ") and inheritance (" IS-A "), Scala offers a special kind of object relation – requirement (" REQUIRES-A "), expressed as self-type annotations. Self-types let us specify additional type expectations for an object, without exposing them in the inheritance hierarchy.

We can use self-type annotations together with traits to implement dependency injection:

```
trait Repository {
  def save(user: User)
}

trait DatabaseRepository extends Repository { /* ... */ }

trait UserService { self: Repository => // requires Repository
  def create(user: User) {
    // ...
    save(user)
  }
}

new UserService with DatabaseRepository
```

Unlike constructor injection, this approach requires a single reference for each kind of dependency in configuration. The full implementation of this technique is known as Cake pattern (and there are many other ways to implement DI in Scala).

Since mixing in traits is done statically in Scala, this approach is limited to compile-time dependency injection. However, in practice, run-time re-configuration is rarely needed, while static checking of configuration is a definite advantage (over an XML-based configuration).

7.3 References

http://www.nescala.org/2013#t-62625592 https://dl.dropboxusercontent.com/u/1679797/NE%20Scala/Bakery%20from%20the%20Blac
https://pavelfatin.com/design-patterns-in-scala/#dependency-injection Programming Scala, O'Reilly (2009)

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search