// A Datalog++ Program. The syntax is inspired by Datalog, Scala and Coq.

// In practice we would probably implement as small Scala DSL which mimics this style.

// The purpose of this example is to show the kinds of properties we want to express.

// We want a type-safe variant of Datalog.

// We begin by declaring the existence of some types which we will use in the following.

// In this example we are interested in points-to information and numbers.

Type Variable, Object Field, Number.

// We can now declare the four well-known points-to relations.

// These are the relations which will hold the input *facts*.

// In Datalog++ relations are typed and each component in a relation has a name.

Relation New(lhs: Variable, obj: Object).

Relation Assign(lhs: Variable, rhs: Variable).

Relation Load(lhs: Variable, base: Variable , field: Field).

Relation Store(base: Variable, field: Field, rhs: Variable).

// We declare two additional input relations.

// One for storing constant integers into a variable

// and one for computing the sum of two variables.

// Notice that the type Int is a built-in type and not a declared type.

// This is important since we rely on SMT knowledge of integers.

Relation Number(lhs: Variable, number: Int).

Relation Plus(lhs: Variable, left: Variable, right: Variable).

// In this example we want to abstract integers using the interval lattice.

// We begin by declaring its elements as an ADT:

// (NB: Like in Haskell the namespace for types and constructors are separate).

Lattice Interval = Bot

| Interval(b: Int, e: Int) **where** b <= e

| Top.

// Notice the *where* clause. Without this restriction we won’t be able to

// (automatically) prove some of the properties we require.

// (NB: We assume, for now, that an element named Bot is the bottom element).

// The lattice declaration automatically gives us the

// Equality relation defined by structural equality.

// Next, we must define the partial order. The system will prove that it is:

// - Reflexive.

// - Anti-symmetric.

// - Transitive.

// - That the bottom element is the least element.

Order Interval(i1: Interval, i2: Interval) = (i1, i2) **match** {

(Bot, \_) => true;

(Interval(b1, e1), Interval(b2, e2)) => b2 <= b1 && e1 <= e2;

(\_, Top) => true;

}

// We can now define the least upper bound (or join) of two intervals.

// This time the system will prove that:

// - Join is a total function.

// - Join actually produces an upper bound.

// - And that upper bound is the least upper bound.

Join Interval(i1: Interval, i2: Interval) = (i1, i2) **match** {

(Bot, i) => i;

(i, Bot) => i;

(Interval(b1, e1), Interval(b2, e2)) => Interval(min(b1, b2), max(e1, e2));

(Top, \_) => Top;

(\_, Top) => Top;

}

// Here the min and max functions are defined – by the system – for integers.

// The next thing we need is an abstraction function

// which can lift an integer into an interval.

// The system will prove that this function is total.

Function lift(n: Int): Interval = Interval(n, n).

// Finally, since the interval lattice has infinite ascending chains,

// we need a widening function. The widening must be:

// - Monotonic.

// - Greater (or equal) to its inputs.

// - Provide a termination function (a bounding function) which can be used to prove

// that there are no infinite chains.

Widening widen(n1: Interval, n2: Interval): Interval = (n1, n2) **match** {

(Bot, i) => i;

(i, Bot) => i;

(Interval(b1, e1), Interval(b2, e2)) **if** max(e1, e2) - min(b1, b2) < 10

=> Interval(min(b1, b2), max(e1, e2));

(Interval(b1, e1), Interval(b2, e2)) => Top;

(Top, \_) => Top;

(\_, Top) => Top;

}

// The termination function:

// - It is always non-negative.

// - It is always decreasing.

Bound widen(n: Interval): Int = n **match** {

Bot => 100;

Interval(b, e) => 100 – (e - b);

Top => 0;

}

// The abstract sum function takes two intervals and adds them together.

// The system proves that the function is monotone.

Function sum(n1: Interval, n2: Interval): Interval = (n1, n2) **match** {

(Bot, \_) => Bot;

(\_, Bot) => Bot;

(Interval(b1, e1), Interval(b2, e2)) => Interval(b1 + b2, e1 + e2);

(Top, \_) => Top;

(\_, Top) => Top;

}

// We now define an abstract value as the product

// lattice of a set of objects and an interval.

Lattice Value(pointers: Set[Object], interval: Interval).

// Finally, we can declare the variable points-to and heap points-to relations.

// These are completely standard, except that a variable (or field) can point to a value.

Relation VarPointsTo(variable: Int, value: Value).

Relation HeapPointsTo(base: Object, field: Field, value: Value).

// These are the standard points-to constraints:

Constraint VarPointsTo(var1, value) :- Assign(var1, var2),

VarPointsTo(var2, value).

Constraint VarPointsTo(var, value) :- Load(var1, var2, field),

VarPointsTo(var2, Value(o, \_)),

HeapPointsTo(o, field, value).

Constraint HeapPointsTo(base, field, value) :- Store(var1, field, var2),

VarPointsTo(var1, Value(o, \_)),

VarPointsTo(var2, value).

// Notice the use of pattern matching in VarPointsTo.

// Next we add two rules for dealing with integers using the interval lattice:

Constraint VarPointsTo(var, value) :- Number(var, n),

value <- lift(n).

Constraint VarPointsTo(var, value) :- Plus(var, Value(\_, n1), Value(\_, n2)),

value <- sum(n1, n2).

// Here we use the new syntax “x <- f(...)” to make clear that

// lift and sum are functions and not relations.

// We are now able to find certain errors in the program.

// For instance, let’s find all field loads where the base

// is not actually an object, but a number.

Relation NumberDeref(base, field).

Constraint NumberDeref(base, field) :- Load(\_, base, field),

VarPointsTo(base, Value(\_, number)),

Bot < number.

// Here the constraint “Bot < number” expresses

// that the interval part of an abstract value must be

// non-bottom for the rule to match.

// Notice that this is monotonic. Furthermore, notice

// that < is easy to derive from Equal and Order.

// Finally, finally, we should add a ton of facts to get everything started...

FactNew(3, 17).

Fact New(4, 5).

Fact Load(5, “foo”, 6).

// and so on...

// TODO: @Unchecked and @Distributive.