

# Iterations

Perna, Tannen, Wong

November 17, 2019

## 1 Limsoon's original proposal

$e_1 : \text{LTrack}[t_1]$   
 $e_2 : \text{Track}[t_2]$   
 $(x_1, X_2) \Rightarrow e : [t_1, \text{Track}[t_2]] \Rightarrow \text{Track}[t]$   
 $x_1 \Rightarrow \gamma_1 : t_1 \Rightarrow \text{Boolean}$   
 $(x_1, x_2) \Rightarrow \gamma_2 : [t_1, t_2] \Rightarrow \text{Boolean}$   
 $\vdash \bigcup \{ e \mid x_1 \in e_1 \text{ st } \gamma_1, X_2 \subseteq e_2 \ni x_2 \text{ st } \gamma_2 \} : \text{Track}[t]$

In the background we have the optimization predicate (“can see”):  $\xi : [t_1, t_2] \Rightarrow \text{Boolean}$

As well as nother optimization predicate (“join positions still possible ahead”):

$\beta : \text{Product}[t_1, t_2] \Rightarrow \text{Boolean}$

The “lzip” implementation:

```
lzip : (t1->bool)*(t1 * t2->bool)*(t1 * t2->bool)*
      (t2 * t'->t')*(t1 * t'->t')*(t'->{t})*t'*t'*{t1}*{t2} -> {t}

lzip (sx, sy, ay, h, g, f, a, e) ({}, Y) = f a
lzip (sx, sy, ay, h, g, f, a, e) (X, {}) = f a
lzip (sx, sy, ay, h, g, f, a, e) (x::X, y::Y) =
  if sx(x)
  then if sy(x, y)
    then if ay(x,y)
      then lzip (sx, sy, ay, h, g, f, h(y, g(x, a)), e) (x::X, Y)
      else lzip (sx, sy, ay, h, g, f, g(x, a), e) (x::X, Y)
    else f (g(x, a)) @ lzip (sx, sy, ay, h, g, f, e, e) (X, y::Y)
  else f a @ lzip (sx, sy, ay, h, g, f, e, e) (X, y::Y)
```

Then take  $\mathbf{sx}(x_1) = \gamma_1$ ,  $\mathbf{ay}(x_1, x_2) = \xi(x_1, x_2) \wedge \gamma_2$ ,  $\mathbf{sy}(x_1, x_2) = \beta(x_1, x_2) \vee \xi(x_1, x_2)$   
and  $\mathbf{h}(x_1, (X_1, X_2)) = (X_1, X_2 \cup \{x_2\})$ ,  $\mathbf{g}(x_1, (X_1, X_2)) = (X_1 \cup \{x_1\}, X_2)$ ,  
 $\mathbf{f}(X_1, X_2) = \bigcup \{e \mid x_1 \in X_1\}$  in  $\text{lzip}(\mathbf{sx}, \mathbf{sy}, \mathbf{ay}, \mathbf{h}, \mathbf{g}, \mathbf{f}, (\{\}, \{\}), (\{\}, \{\})) (e_1, e_2)$

## 2 A library function; assumptions

We will make the optimization predicates into explicit parameters. We also avoid bound variables ( $\mathbf{x} \Rightarrow$ ) using instead functions as parameters, resulting in a higher-order formulation of a **dependent join**. Moreover, we want to investigate under what assumptions do the optimization predicates work properly.

Let's also switch notation. Denote the landmark track type by  $\mathbf{LTrack}[\ell]$ , the track itself by  $e_\ell$  and its elements by  $x, x', x_1, x_2, \dots : \ell$ . Denote the experimental track by  $\mathbf{Track}[t]$ , the track itself by  $e_t$  and its elements by  $y, y', y_1, y_2, \dots : t$ . The result will be of type  $\mathbf{Track}[r]$ .

$e_\ell : \mathbf{LTrack}[\ell]$   
 $e_t : \mathbf{Track}[t]$   
 $f : [\ell, \mathbf{Track}[t]] \Rightarrow \mathbf{Track}[r]$   
 $\gamma_\ell : \ell \Rightarrow \mathbf{Boolean}$   
 $\gamma_t : [t] \Rightarrow \mathbf{Boolean}$   
 $\xi : [\ell, t] \Rightarrow \mathbf{Boolean}$   
 $\beta : [\ell, t] \Rightarrow \mathbf{Boolean}$   
 $\vdash \quad \mathbf{DJGenLs}(e_\ell, e_t, f, \gamma_\ell, \gamma_t, \xi, \beta) : \mathbf{Track}[r]$

### Assumptions:

1. The elements of types  $\ell, t$  and  $r$  are totally ordered, notation  $<$ .  $\mathbf{LTrack}$  is a subtype of  $\mathbf{Track}$  and  $\mathbf{Track}$  implements an “iterable” interface whose iterators traverse a collection of type  $\mathbf{Track}[s]$  (where  $s$  is  $\ell$  or  $t$ ) in the order given by  $<$ .
2. Introduce the notations

$$B(x) = \{y \mid \beta(x, y)\} \qquad J(x) = \{y \mid \xi(x, y)\}$$

3. Compatibility with the order on landmarks

$$x < x' \Rightarrow B(x) \subsetneq B(x') \quad \text{OR IS IT} \quad B(x) \subseteq B(x')$$

4.  $B(x)$  is *downwards closed* wrt  $<$  (“down” or “left”?)

$$y \in B(x) \wedge y' < y \Rightarrow y' \in B(x)$$

5. Crucial property

$$\neg \beta(x, y) \wedge \neg \xi(x, y) \Rightarrow \forall y' > y \quad \neg \xi(x, y')$$

Equivalently

$$y \notin B(x) \wedge y \notin J(x) \Rightarrow \forall y' > y \quad y' \notin J(x)$$

6. Note that neither of  $\beta$  or  $\xi$  completely determines the other, so both are needed.

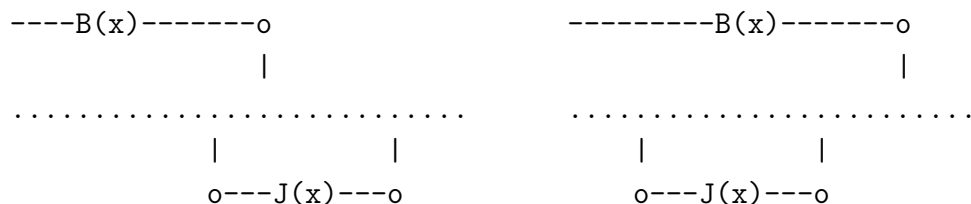
7. In many applications,  $J(x)$  will be *convex*

$$y < y' < y'' \wedge y \in J(x) \wedge y'' \in J(x) \Rightarrow y' \in J(x)$$

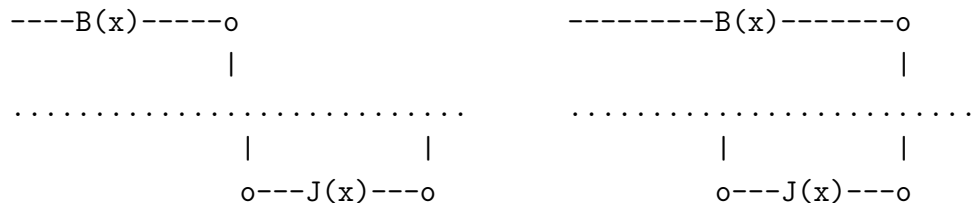
However, we do *not* want this to be a necessary condition for the dependent join to work correctly.

Here are some common cases and corner cases represented visually.

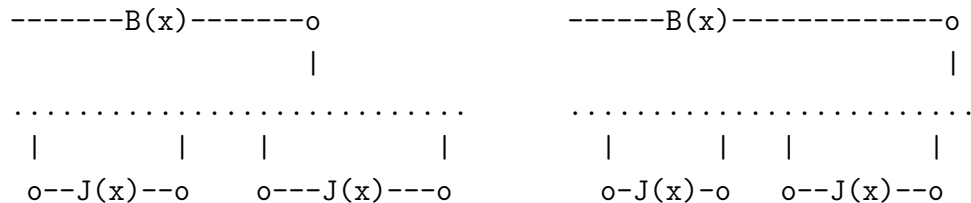
common situations on the experimental track:



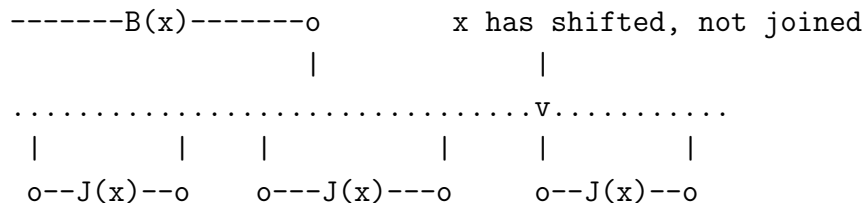
corner cases that must work also



J non-convex cases that should also work



what should the algorithm do with "late" J-elements?



### 3 Pseudocode: preamble with types and iterators

EL, ET, ER SUBTYPES OF OrderedElement

LTrack[OrderedElement] SUBTYPE OF Track[OrderedElement]

Track[OrderedElement] IMPLEMENTS Iterator

Iterators for Track[OrderedElement] traverse it in the order discussed in the assumptions. Here is the preamble.

```
DJGenLs(eL:LTrack[EL], eT:Track[ET],
        f:[EL,Track[ET]]=>Track[ER],
        gammaL:EL=>Boolean, gammaT:[EL,ET]>Boolean,
        cs:[EL,ET]>Boolean, be:[EL,ET]>Boolean
    ) : Track[E]
```

```
VAR iL = eL.iterator
```

```
VAR iT = eT.iterator
```

```
VAR output : Track[ER] = initially empty
```

```
VAR acc : Track[ET] = initially empty
```

```
VAR xL : EL = some safe initial value
```

```
VAR xT : ET = some safe initial value
```

```
CONTINUED
```

## 4 Pseudocode: the main loop

```
VAR shiftL : Boolean = true
VAR shiftT : Boolean = true
LOOP
  IF ( shiftL = true ) {
    IF ( iL.hasNext ) {
      xL = iL.next
      shiftL = false
    } ELSE {
      BREAK
    }
  }
  IF ( shiftT = true ) {
    IF ( iT.hasNext ) {
      xT = iT.next
      shiftT = false
    } ELSE {
      output.append(f(xL,acc))
      BREAK
    }
  }
  IF ( gammaL(xL) ) {
    IF ( be(xL,xT) OR cs(xL,xT) ) {
      IF ( cs(xL,xT) AND gammaT(xL,xT) ) {
        acc.appendOne(xT)
      }
      x2 = i2.next
      CONTINUE
    }
    output.append(f(xL,acc))
    acc = empty
  }
  shiftL = true
END_LOOP
clean up
RETURN
```

## 5 Pseudocode: how we got the main loop by refactoring

We start with a particularization of lzip and with the assumption that both tracks are infinite.

```
xL = iL.next
xT = iT.next
LOOP
  IF ( gammaL(xL) ) {
    IF ( be(xL,xT) OR  cs(xL,xT) ) {
      IF ( cs(xL,xT) AND gammaT(xL,xT) ) {
        acc.appendOne(xT)
        xT = xT.next
        CONTINUE
      } ELSE {
        xT = iT.next
        CONTINUE
      }
    } ELSE {
      output.append(f(xL,acc))
      acc = empty
      xL = iL.next
      CONTINUE
    }
  } ELSE {
    xL = iL.next
    CONTINUE
  }
END_LOOP
```

FIRST REFACTORING: IF code 1 ELSE code 2 becomes IF code 1 ; code 2. In addition, this allows rearranging some code that is common to both branches. From 4 CONTINUE statements we go to 2 total. Then the second one is not necessary since it's where the loop repeats anyway.

```
xL = iL.next
xT = iT.next
LOOP
  IF ( gammaL(xL) ) {
    IF ( be(xL,xT) OR cs(xL,xT) ) {
      IF ( cs(xL,xT) AND gammaT(xL,xT) ) {
        acc.appendOne(xT)
      }
      xT = iT.next
      CONTINUE
    }
    output.append(f(xL,acc))
    acc = empty
  }
  xL = iL.next
END_LOOP
```

SECOND REFACTORING: L-track is finite. The other track still infinite.

```
VAR shiftL : Boolean = true
xT = iT.next
LOOP
  IF ( shiftL = true ) {
    IF ( iL.hasNext ) {
      xL = iL.next
      shiftL = false
    } ELSE {
      BREAK
    }
  }
  IF ( gammaL(xL) ) {
    IF ( be(xL,xT) OR cs(xL,xT) ) {
      IF ( cs(xL,xT) AND gammaT(xL,xT) ) {
        acc.appendOne(xT)
      }
      x2 = i2.next
      CONTINUE
    }
    output.append(f(xL,acc))
    acc = empty
  }
  shiftL = true
END_LOOP
clean up
RETURN
```



THIRD REFACTORING: Both tracks are finite.

```
VAR shiftL : Boolean = true
VAR shiftT : Boolean = true
LOOP
  IF ( shiftL = true ) {
    IF ( iL.hasNext ) {
      xL = iL.next
      shiftL = false
    } ELSE {
      BREAK
    }
  }
  IF ( shiftT = true ) {
    IF ( iT.hasNext ) {
      xT = iT.next
      shiftT = false
    } ELSE {
      output.append(f(xL,acc))
      BREAK
    }
  }
  IF ( gammaL(xL) ) {
    IF ( be(xL,xT) OR cs(xL,xT) ) {
      IF ( cs(xL,xT) AND gammaT(xL,xT) ) {
        acc.appendOne(xT)
      }
      x2 = i2.next
      CONTINUE
    }
    output.append(f(xL,acc))
    acc = empty
  }
  shiftL = true
END_LOOP
clean up
RETURN
```

## 6 THREE TRACKS pseudocode: preamble

EL, ET1, ET2, E SUBTYPES OF OrderedElement

LTrack[OrderedElement] SUBTYPE OF Track[OrderedElement]

Track[OrderedElement] IMPLEMENTS Iterator

```
DJGenLs(eL:LTrack[EL], eT1:Track[ET1], eT2:Track[ET2],
        f:[EL,Track[ET1],Track[ET2]]=>Track[E],
        gammaL:EL=>Boolean, gammaT1:[EL,ET1]>Boolean,
        gammaT2:[EL,ET2]>Boolean,
        cs1:[EL,ET1]>Boolean, be1:[EL,ET2]>Boolean
        cs2:[EL,ET2]>Boolean, be2:[EL,ET2]>Boolean
    ) : Track[E]
```

```
VAR iL = eL.iterator
VAR iT1 = eT1.iterator
VAR iT2 = eT2.iterator
```

```
VAR output : Track[E] = initially empty
```

```
VAR accT1 : Track[ET1] = initially empty
VAR accT2 : Track[ET2] = initially empty
```

```
VAR xL : EL = some safe initial value
```

```
VAR xT1 : ET1 = some safe initial value
VAR xT2 : ET2 = some safe initial value
```

CONTINUED

START VERSION: ASSUMING ALL THREE TRACKS ARE INFINITE

```
xL = iL.next
xT1 = iT1.next
xT2 = iT2.next
LOOP
  IF ( gammaL(xL) ) {
    IF ( be1(xL,xT1) OR  cs1(xL,xT1) ) {
      IF ( cs1(xL,xT1) AND gammaT1(xL,xT1) ) {
        accT1.appendOne(xT1)
        xT1 = iT1.next
        CONTINUE
      } ELSE {
        xT1 = iT1.next
        CONTINUE
      }
    }
    IF ( be2(xL,xT2) OR  cs2(xL,xT2) ) {
      IF ( cs2(xL,xT2) AND gammaT2(xL,xT2) ) {
        accT2.appendOne(xT2)
        xT2 = iT2.next
        CONTINUE
      } ELSE {
        xT2 = iT2.next
        CONTINUE
      }
    }
    output.append(f(xL,accT1,accT2))
    accT1 = empty; accT2 = empty
    xL = iL.next;
    CONTINUE
  } ELSE {
    xL = iL.next
    CONTINUE
  }
END_LOOP
```

## FIRST REFACTORING: “ELSE” AND COMMON CODE REMOVAL

```
xL = iL.next
xT1 = iT1.next
xT2 = iT2.next;
LOOP
  IF ( gammaL(xL) ) {
    IF ( be1(xL,xT1) OR  cs1(xL,xT1) ) {
      IF ( cs1(xL,xT1) AND gammaT1(xL,xT1) ) {
        accT1.appendOne(xT1)
      }
      xT1 = iT1.next
      CONTINUE
    }
    IF ( be2(xL,xT2) OR  cs2(xL,xT2) ) {
      IF ( cs2(xL,xT2) AND gammaT2(xL,xT2) ) {
        accT2.appendOne(xT2)
      }
      xT2 = iT2.next
      CONTINUE
    }
    output.append(f(xL,accT1,accT2))
    accT1 = empty; accT2 = empty
  }
  xL = iL.next
END_LOOP
```

SECOND REFACTORING: landmark track finite the others infinite

```
VAR shiftL : Boolean = true
xT1 = iT1.next
xT2 = iT2.next
LOOP
  IF ( shiftL ) {
    IF ( iL.hasNext ) {
      xL = iL.next
      shiftL = false
    } ELSE {
      BREAK
    }
  }
  IF ( gammaL(xL) ) {
    IF ( be1(xL,xT1) OR  cs1(xL,xT1) ) {
      IF ( cs1(xL,xT1) AND gammaT1(xL,xT1) ) {
        accT1.appendOne(xT1)
      }
      xT1 = iT1.next
      CONTINUE
    }
    IF ( be2(xL,xT2) OR  cs2(xL,xT2) ) {
      IF ( cs2(xL,xT2) AND gammaT2(xL,xT2) ) {
        accT2.appendOne(xT2)
      }
      xT2 = iT2.next
      CONTINUE
    }
    output.append(f(xL,accT1,accT2))
    accT1 = empty; accT2 = empty
  }
  shiftL = true
END_LOOP
CLEAN UP
RETURN
```

THIRD REFACTORING: all tracks finite

```
VAR shiftL : Boolean = true
VAR shiftT1 : Boolean = true; VAR doneT1 : Boolean = false
VAR shiftT2 : Boolean = true
LOOP
  IF ( shiftL ) {
    IF ( iL.hasNext ) {
      xL = iL.next; shiftL = false
    } ELSE {
      BREAK
    }
  }
  IF ( shiftT1 ) {
    IF ( iT1.hasNext ) {
      xT1 = iT1.next; shiftT1 = false
    } ELSE {
      doneT1 = true;
      CONTINUE
    }
  }
  IF ( shiftT2 ) {
    IF ( iT2.hasNext ) {
      xT2 = iT2.next; shiftT2 = false
    } ELSE {
      output.append(f(xL,accT1,accT2))
      BREAK
    }
  }
  IF ( gammaL(xL) ) {
    IF ( ( be1(xL,xT1) OR cs1(xL,xT1) ) AND !doneT1 ) {
      IF ( cs1(xL,xT1) AND gammaT1(xL,xT1) ) {
        accT1.appendOne(xT1)
      }
      shiftT1 = true
      CONTINUE
    }
    IF ( be2(xL,xT2) OR cs2(xL,xT2) ) {
      IF ( cs2(xL,xT2) AND gammaT2(xL,xT2) ) {
        accT2.appendOne(xT2)
      }
    }
  }
```

```
        shiftT2 = true
        CONTINUE
    }
    output.append(f(xL,accT1,accT2))
    IF ( doneT1 ) {
        BREAK
    }
    accT1 = empty; accT2 = empty
}
shiftL = true
END_LOOP
clean up
RETURN
```

## 7 The generalization

### Assumptions:

1. The elements of types  $\ell, t$  and  $r$  are still totally ordered, notation  $<$ . **LTrack** is a subtype of **Track** and **Track** implements an “iterable” interface whose iterators traverse a collection of type **Track**[ $s$ ] (where  $s$  is  $\ell$  or  $t$ ) in the order given by  $<$ .
2.  $x : \ell$  and  $y : t$ .
3. Now let  $J(x)$  be the set of  $y$ ’s that actually joins with  $x$ . We should allow for  $J(x) = \emptyset$ .
4. Let  $R(x)$  be a superset of  $J(x)$  that is convex with respect to  $<$ . We should require  $R(x) \neq \emptyset$ .
5. Compatibility with the order. Suppose  $x < x'$ . We should not allow for the existence of some  $y' \in R(x')$  such that *for all*  $y \in R(x)$   $y' > x$ . This means that if we iterate in order through the  $t$ -track we encounter the first element of  $R(x)$  before we encounter the first element of  $R(x')$ , or maybe these first elements are the same element.
6. Synchronized scan of both tracks accumulates pairs  $(x, y)$  such that  $y \in J(x)$ . Starts looking to accumulate them when entering  $R(x)$  and stops looking when exiting  $R(x)$ , when it also removes them from the accumulator and processes them.