Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа прикладной математики и информатики
Кафедра дискретной математики

Выпускная квалификационная работа бакалавра по направлению 010900
«Прикладные математика и информатика»

Верификация асимптотической оценки временной сложности в задачах динамического программирования

Студент 7910б группы
Григорянц С. А.

Научный руководитель
Дашков Е. В.

Долгопрудный
2021

Содержание

Глава 1

Введение

Сегодня, все больше отраслей компьютерных технологий нуждаются в формальной верификации. В первую очередь, в этот список входят программы, связанные с транспортом, коммуникацией, медициной, компьютерной безопасностью, криптографией и банковским делом. В этих критических сферах, последствия отсутствия формальной верификации в некоторых проектах привели к очень плачевным последствиям. Например, миссия НАСА Mars Climate Orbiter сорвалась из-за довольно банальной ошибки в софте [5]. Есть и другие, гораздо более пугающие примеры [5]. Фундаментальным отличием формальной верификации от классического тестирования ПО заключается в том, что, в то время как классическое тестирование проверяет систему лишь на каком-то подмножестве возможных входов, формальная верификация ПО позволяет утвеждать о корректности системы на всех возможных входах. Такого вида гарантии конечно же дают несравнимо большую уверенность в корректности работы данного ПО, даже учитвая то, что остальные части системы, такие как компилятор, аппаратное обеспечение, и, в конце концов, само ПО, используемое для верификации, могут дать сбои. Ибо мы таким образом убеждаемся в том, что сам код, который является частью ПО, ошибок не содержит. Можно привести в пример много проектов, использующих формальную верификацию для довольно сложных систем. Например, в верификации компиляторов – проект Jinja [11, 12], проект Verisoft [13, 21], а также проект CompCert [1, 14]. В блокчейне – проект Scilla [20]. Таким образом, мы можем убедиться в том, что формальная верификация ПО и вправду очень востребованна.

Одним из основных инструментов верификации программ является интерактивное программное средство доказательства теорем Coq [23]. Coq представляет из себя богатый формальный язык, основанный на исчислении индуктивных конструкций(Calculus of Inductive Constructions, CIC) [17] – формальной системе, которая способна представлять как функциональные программы в стиле ML [16], так и доказательства в логике высшего порядка. CIC в свою очередь основан на чистой системе типов(Pure type system [18]) – исчислении конструкций (The Calculus of Constructions) [4]. Coq, как и любое другое средство доказательств, основанное на теории типов, использует для представления утверждений и доказательств Соответствие Карри — Ховарда [10] – изоморфизм, сопоставляющий логическому высказыванию тип, а доказательству данного утверждения – терм данного типа. Используя это соответствие, задача проверки доказательства сводится к задаче проверки типа выражения. Эта задача решается специальном модулем Coq, ядром – отностительно небольшой программой, очень тщательно написанной, т.к. это самая важная часть системы – допущенная в ней ошибка может привести к доказательствам неверных утверждений. Для формализации внутри Coq математических концепций и спецификации ПО Coq предоставляет богатый язык команд Vernacular. С помощью него можно вводить новые обозначения, отношения, порядки, предикаты, и т.д.

Для построения фреймворка для верификации императивных программ, нужно каким-то образом инкорпорировать в этот фреймвор состояние программы, а также создать инструменты для рассуждения о том, как программы влияют на эти самые состояния. Стандартным подходом для данной задачи является логика Хоара [9]. Логика Хоара представляет из себя формальную систему, которая позволяет рассуждать о корректности императивных программ. Основным инструментом логики Хоара является тройка Хоара $\{P\}C\{Q\}$, где P и Q – это предикаты на множестве состояний, а C – это некоторая поседовательность команд. Вывод данной тройки в логике Хоара означает, что если текущее состояние удовлетворяет утверждению P, то после выполнения на нем программы C она будет удовлетворять утверждению Q. Правила вывода в логике Хоара строятся довольно естественно. Например, ясно, что пустая команда ничего не делает с состоянием, поэтому можно добавить правило вывода $\{P\}\textbf{skip}\{P\}$.

Логиках Хоара является отличным инструментом для рассуждения о программах, но она не может нам помочь в случае, когда нам нужно иметь общее состояние в программе(то есть указатели на те же участки памяти). Формально говоря, состояние в логике Хоара – это всего лишь отображение переменных в их значения, но в реальных языках программирования присутствуют также и указатели, которые хранят не значение, а ссылку на него. С такого вида состояниями логиха Хоара работать не позволяет. Например, пусть утверждение имеет вид $x \mapsto 3 \wedge y \mapsto 3$, то есть говорит о двух указателях, которые указывают на ячейку с числом 3 внутри. Из этого утверждения никак не понять, является ли ячейка, на которую указывают x и y, одной и той же, или же это разные ячейки, у которых просто совпадают значения? Очевидно, что от ответа на этот вопрос зависит эффект, который произведет, скажем, команда $\{*x = 4\}$ – присваивание ячейки, на которую указывает x, значение 3.

Для рассуждения подобного вида нам приходит на помощь Сепарационная Логика [19]. Это расширение логики Хоара, которое вводит дополнительную логическую операцию $P * Q$ – логическую конъюкцию. Выражение $P * Q$ утверждает, что текущее состояние может быть разделено на два непересекающихся в адресном пространстве состояния, каждое из которых удовлетворяет P и Q соответственно. Данная логика также вводит одно очень важное правило вывода, называемое правилом фрейма(frame rule)

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}$$

Это правило вывода позволяет обобщать локальные рассуждение о маленьком состоянии на большее состояние. В данной логике предикат $x \mapsto 3 * y \mapsto 3$ уже имеет четкий смысл. Библиотека CFML [3] формализует Сепарационную Логику в Coq для верификации программ на OCaml.

Но наряду с верификацией программ, также встает вопрос верификации асимптотики применяемого алгоритма. Действительно, довольно часто мы хотим убедиться не только в том, что алгоритм корректен, но также и в том, что его асимптотика соответствует нашим ожиданиям. На самом деле, баги, связанные с асимтотикой алгоритма, могут возникать довольно часто только для конкретных входов, что делает классический подход тестирования неприемлемым для их отыскания. например, рассмотрим следующую реализацию бинарного поиска (на языке Python).

Рис. 1. Проблемный бинарный поиск

```
# Requires t to be a sorted array of integers.
# Returns k such that i <= k < j and t[k] = v
# or -1 if there is no such k.
def bsearch(t, v, i, j):
    if j <= i:
        return -1
    k = i + (j-i) // 2
    if v == t[k]:
        return k
    elif v < t[k]:
        return bsearch(t, v, i, k)
    return bsearch(t, v, i+1, j)
```

Проблема этого кода состоит в том, что при попадании в правую часть списка, вместо того, чтобы рассматривать интервал $[k+1; j)$, мы рассматриваем интервал $[i+1; j)$. Конечно, сам алгоритм корректно реализует бинарный поиск, и его корректность можно формально доказать, но асимптотика при вводе, скажем, последнего элемента массива, превращается из логарифмической в линейную. На этом примере хорошо видно, что даже формальной верификации алгоритма и классического тестирования его на проверку асимптотики работы не всегда гарантирует нам корректность этой самой асимптотики. В связи с этим, возникает потребность в том, чтобы иметь также возможность формально верифицировать не только корректность алгоритма, но и его асимптотику.

В данной работе мы покажем, как формальная верификация алгоритмов и их асимптотик может быть реализована с помощью Сепарационной Логики с временными кредитами [8]. С помощью этой теории, мы формально верифицируем алгоритм нахождения наибольшей общей подпоследовательности в системе интерактивных доказательств Coq [23].

Глава 2

Результаты

В этой главе мы изложим полученный результат, состоящий в верификации алгоритма LCS.

2.1 Реализация алгоритма

Начнем изложение с представления реализации самого алгоритма. Ввиду использования фреймворка CFML [8], алгоритм реализован на языке OCaml [15].

Рис. 2. Реализация LCS

```
let lcs (a : int array) (b : int array) : int array =
  let n = Array.length a in
  let m = Array.length b in
  let c = Array.make ((n+1)*(m+1)) [] in
  for i = 1 to n do
    for j = 1 to m do
      if a.(i-1) = b.(j-1)
      then c.(i*(m+1) + j) <- List.append c.((i-1)*(m+1) + j - 1) [a.(i-1)]
      else if List.length c.((i-1)*(m+1) + j) > List.length c.(i*(m+1) + j - 1)
        then c.(i*(m+1) + j) <- c.((i-1)*(m+1) + j)
        else c.(i*(m+1) + j) <- c.(i*(m+1) + j - 1)
      done
    done;
  Array.of_list c.((n+1)*(m+1)-1);;
```

Здесь приведена реализация классического алгоритма LCS, в котором с помощью динамического программирования вычисляется массив $c[i][j]$, который содержит $LCS(A_i, B_j)$, где $A_i = a_1 \ldots a_i$, $B_j = b_1 \ldots b_j$. Формула для вычисления $LCS(A_i, B_j)$ из предыдущих значений следующая:

$$LCS\left(A_i, B_j\right) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS\left(A_{i-1}, B_{j-1}\right) x_i & \text{if } i, j > 0 \text{ and } a_i = b_j \\ \max\left\{LCS\left(A_i, B_{j-1}\right), LCS\left(A_{i-1}, B_j\right)\right\} & \text{if } i, j > 0 \text{ and } a_i \neq b_j. \end{cases}$$

Но стоит отметить, что вместо двумерного массива c используется одномерных массив размера $|a| \times |b|$. Это сделано из-за того, что в оригинальной своей версии CFML [8], в языке пропозициональных высказываний о куче (тип **hprop**) не поддерживается квантора всеобщности, а только квантор существования (**HExists**). В случае двумерного массива квантов всеобщности необходим для построения утвержений инварианте цикла. Добавления квантора всеобщности в эту систему является не такой простой задачей, как может показаться на первый взгляд, поэтому был использован одномерный массив.

2.2 Формализация утверждений об Lcs

Для того, чтобы сформулировать теорему об корректности работы и асимптотики алгоритма LCS, описанного выше, кроме инструментария CFML, также необходима и некая формализация в Coq самого понятия LCS. Делается это с помощью так называемых индуктивных высказываний. Определим сначала высказывание подпоследовательность, то есть SubSeq l_1 l_2 означает, что l_1 является подпоследовательностью l_2. Это утверждение формализуется следующим способом:

Рис. 3. Определение SubSeq

```
Inductive SubSeq {A:Type} : list A -> list A -> Prop :=
 | SubNil (l:list A) : SubSeq nil l
 | SubCons1 (x:A) (l1 l2:list A) (H: SubSeq l1 l2) : SubSeq l1 (x::l2)
 | SubCons2 (x:A) (l1 l2:list A) (H: SubSeq l1 l2) : SubSeq (x::l1) (x::l2).
```

Теперь мы также можем определить утверждение Lcs l l_1 l_2, которое будет означать, что l является наибольшей общей подпоследовательностью l_1 и l_2:

Рис. 4. Определение Lcs

```
Definition Lcs {A: Type} l l1 l2 :=
  SubSeq l l1 /\ SubSeq l l2 /\
  (forall l': list A, SubSeq l' l1 /\ SubSeq l' l2 -> length l' <= length l).
```

В этом утверждении просто проверяется, что во-первых l и вправду является подпоследовательностью l_1 и l_2, а также то, что любая другая их общая подпоследовательность l' имеет длину, не большую l. Далее, для доказательства корректности работы алгоритма, нужно будет убедиться в правильности построения $LCS(A_i, B_j)$. Вспомним правило построения:

$$LCS\left(A_i, B_j\right) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS\left(A_{i-1}, B_{j-1}\right) x_i & \text{if } i, j > 0 \text{ and } a_i = b_j \\ \max\left\{LCS\left(A_i, B_{j-1}\right), LCS\left(A_{i-1}, B_j\right)\right\} & \text{if } i, j > 0 \text{ and } a_i \neq b_j. \end{cases}$$

По большому счету, нам просто нужно доказать два утверждения об Lcs, в зависимости от равенства последних символов строк. Эти утверждения формализуются следующим способом:

Рис. 5. Необходимые свойства Lcs

```
Lemma lcs_app_eq: forall (l1 l2 l: list int) (x: int),
  Lcs l l1 l2 -> Lcs (l & x) (l1 & x) (l2 & x).

Lemma lcs_app_neq: forall (l1 l2 l l': list int) (x y : int),
  x <> y -> Lcs l (l1&x) l2 -> Lcs l' l1 (l2&y) -> length l' <= length l ->
  Lcs l (l1&x) (l2&y).
```

Для доказательства этих основных утверждений об Lcs, были использованы следующие вспомогательные леммы, устанавливающие свойства SubSeq и Lcs:

Рис. 6. Вспомогательные утверждения

```
Lemma subseq_cons: forall A l1 l2 (x : A), SubSeq (x::l1) l2 -> SubSeq l1 l2.

Lemma subseq_app: forall A l1 l2 (x : A), SubSeq l1 l2 -> SubSeq (l1 & x) (l2 & x).

Lemma subseq_nil: forall A (l : list A), SubSeq l nil -> l = nil.

Lemma subseq_length: forall (l a: list int), SubSeq l a -> length l <= length a.

Lemma subseq_cons_l: forall A l1 l2 (x : A),  SubSeq (x :: l1) l2 <->
  exists l2' l2'', l2 = l2' & x ++ l2'' /\ SubSeq l1 l2''.

Lemma subseq_last_head: forall l1 l2 (x y : int),
  SubSeq (l1 & x) (l2 & y) -> SubSeq l1 l2.

Lemma subseq_app_r: forall (l l1 l2: list int), SubSeq l l1 -> SubSeq l (l1 ++ l2).

Lemma subseq_last_case: forall l l1 (x : int), SubSeq l (l1 & x) ->
  (SubSeq l l1) \/ (exists l', l = l' & x /\ (SubSeq l' l1)).

Lemma subseq_last_case: forall l l1 (x : int), SubSeq l (l1 & x) ->
  (SubSeq l l1) \/ (exists l', l = l' & x /\ (SubSeq l' l1)).

Lemma subseq_last_neq: forall l l1 l2 (x y : int), x <> y -> SubSeq l (l1 & x) ->
  SubSeq l (l2 & y) -> (SubSeq l l1) \/ (SubSeq l l2).

Lemma lcs_nil_nil: forall A (l: list A), Lcs nil nil l.

Lemma lcs_symm: forall A (l l1 l2 : list A), Lcs l l1 l2 <-> Lcs l l2 l1.
```

2.3 Формулировка основной теоремы

Теперь мы практически готовы сформулировать основную теорему. Осталось только задать отношение на фильтре, т.к. для формулировки нам нужен фильтр над $\mathbb{Z}^2$. Нам необходим стандартный фильтр на $\mathbb{Z}^2$, получающийся как произведение двух стандартных фильров над $\mathbb{Z}$:

Рис. 7. Определение фильтра

```
Definition ZZle (p1 p2 : Z * Z) :=
  let (x1, y1) := p1 in
  let (x2, y2) := p2 in
  1 <= x1 <= x2 /\ 0 <= y1 <= y2.
```

Теперь, можно сформулировать основную теорему:

Рис. 8. Формулировка основной теоремы

```
Lemma lcs_spec:
  spec0
    (product_filterType Z_filterType Z_filterType)
    ZZle
  ( fun cost =>
  forall (l1 l2 : list int) p1 p2,
  app lcs [p1 p2]
  PRE (\$(cost (LibListZ.length l1, LibListZ.length l2)) \*
  p1 ~> Array l1 \* p2 ~> Array l2)
  POST (fun p => Hexists (l : list int), p ~> Array l \* \[Lcs l l1 l2]))
  (fun '(n,m) => n * m).
```

2.4 Доказательство основной теоремы

Перейдем к изложению доказательства основной теоремы. Начнем с изложения основных идей при доказательстве алгоритмической корректности алгоритма

2.4.1 Доказательство алгоритмической корректности

Ясно, что ключ к ее доказательству лежит в правильном выборе инвариантов двух основных циклов алгоритма. Как было изложено ранее, эти циклы отвечают за вычисление $c[i][j] = LCS(A_i, B_j)$, то есть инвариантами цикла является утверждение о том, что во всех предыдущих ячейках $c[i][j]$ записаны верные значения $LCS(A_i, B_j)$. Может показаться, что этого достаточно, но на самом деле для формальной верификации нужно еще кое-что. Мы ведь не учли, что наш инвариант должен оставаться верным и при первой итерации цикла, когда в $c[i][j]$ записан пустой список по умолчанию (что верно, т.к. мы берем пустой префикс у списка). Поэтому, нам также необходимо добавить в инвариант утверждение о том, что все во всех последующих ячейках $c[i][j]$ записан пустой список. В нашем случае, инвариант будет выглядеть следующим образом:

Рис. 9. Инвариант внешнего цикла

```
xfor_inv (fun (i:int) =>
Hexists (x' : list (list int)),
p1 ~> Array l1 \*
p2 ~> Array l2 \*
c ~> Array x' \*
\[length x' = (n+1)*(m+1)] \*
\[forall i1 i2 : int, 0 <= i1 < i -> 0 <= i2 <= m ->
    Lcs x'[i1*(m+1) + i2] (take i1 l1 ) (take i2 l2) ] \*
\[forall i', i*(m+1) <= i' < (n+1)*(m+1) ->
    x'[i'] = nil ]).
```

Два последних коснтантных предиката кучи в этом выражении как раз и постулируют описанные выше условия. Остальные требования переносят предыдущие переменные неизменными в контекст цикла, а также требование $c \sim>$ Array x' как раз постулирует, что c указывает на список с нужными свойствами. Написанный выше инвариант относится к внешнему циклу, аналогичный инвариант нужно написать и для внутреннего цикла:

Рис. 10. Инвариант внутреннего цикла

```
xfor_inv (fun (j:int) =>
  Hexists (x' : list (list int)),
  p1 ~> Array l1 \*
  p2 ~> Array l2 \*
  c ~> Array x' \*
  \[length x' = (n+1)*(m+1)] \*
  \[forall i1 i2 : int, 0 <= i1 <= i -> 0 <= i2 <= m ->
      i1*(m+1) + i2 < i*(m+1) + j ->
      Lcs x'[i1*(m+1) + i2] (take i1 l1 ) (take i2 l2) ] \*
  \[forall i', i*(m+1) + j <= i' < (n+1)*(m+1) ->
      x'[i'] = nil ]).
```

После формулировки данных инвариантов, оставшаяся часть доказательства представляет из себя довольно занудную процедуру разбора случаев, в которых применяется *lcs_app_eq* либо *lcs_app_neq* в зависимости от самого случая. Также постоянно приходится убеждаться в том, что при обращении к массиву мы не выходим за его рамки.

2.4.2 Доказательство асимптотической корректности

Как было изложено раньше, фреймворк CFML [8] позволяет нам построить функцию стоимости алгоритма в процессе доказательства алгоритмической корректности. После этого, нам нужно показать, что полученная функция является монотонной, а также проверить, что она доминируется заявленной функцией, то есть в нашем случае функцией $f(n, m) = nm$. В процессе доказательства корректности генерируется следующая функция стоимости:

Рис. 11. Функция стоимости

```
fun '(x, y) =>
     (1 +
      (1 +
       (1 +
        ((((x + 1)%I * (y + 1)%I)%I + 1)%I +
         (cumul 1 (x + 1)
            (fun _ : int =>
             1 +
             cumul 1 (y + 1)
               (fun _ : int =>
                1 +
                (1 +
                 (1 +
                  (0 +
                   Z.max (1 + (1 + (0 + 1)%I)%I)
                     (1 + (0 + (1 + (0 + Z.max (1 + 1) (1 + 1))%I)%I)%I))%I)%I)
          (1 + 1)%I)%I)%I)%I)%I)%I
```

Это выражение можно сильно упростить, если заметить, что в аргументе cumul стоит константная функция. После упрощения получаем:

Рис. 12. Упрощенная функция стоимости

```
(fun '(x, y) =>
 (((((x * y)%I + x)%I + y)%I +
   (Z.max 0 ((x + 1)%I - 1) *
    (1 +
     (Z.max 0 ((y + 1)%I - 1) *
      (1 + (1 + (1 + Z.max (1 + (1 + 1)%I)
      (1 + (1 + (1 + 1)%I)%I))%I)%I)%I)%I)%I)%I)%I + 7)%I)
```

Из ее вида уже сразу видно, что она монотонна, и доминируется. Монотонность проверяется непосредственно тактикой math_nia. Для проверки того, что она доминируется функцией $f(n, m) = n * m$, мы для начала раскладываем нашу функцию на слагаемые, и показываем, что каждое из них доминируется. Разложение на слагаемые делается с помощью дистрибутивности по сложению, то есть утверждения: $f_1 = \mathcal{O}(g) \wedge f_2 = \mathcal{O}(g) \implies f_1 + f_2 = \mathcal{O}(g)$, которое формализуется следующим образом:

Рис. 13. Дистрибутивность доминирования по сложению

```
Theorem dominated_sum_distr_nary
: forall (domain : list Type) (M : Filter.mixin_of (Rtuple domain))
      (f g h : Arrow domain int),
    dominated (nFilterType domain M) (Uncurry f) (Uncurry h) ->
    dominated (nFilterType domain M) (Uncurry g) (Uncurry h) ->
    dominated (nFilterType domain M)
      (Fun' (fun p : Rtuple domain => (App f p + App g p)%I))
      (Uncurry h)
```

После разложения на слагаемые получаются довольно простые функции, доминируемость которых мы показываем с помощью дистрибутивности доминирования умножению, то есть утверждения о том, что
$f_1 = \mathcal{O}(g_1) \wedge f_2 = \mathcal{O}(g_2) \implies f_1 f_2 = \mathcal{O}(g_1 g_2)$, которое формально выглядит так:

Рис. 14. Дистрибутивность доминирования по умножению

```
Theorem dominated_mul_nary
    : forall (domain : list Type) (M : Filter.mixin_of (Rtuple domain))
    (f1 f2 g1 g2 : Rarrow domain int),
  dominated (nFilterType domain M) (Uncurry f1) (Uncurry g1) ->
  dominated (nFilterType domain M) (Uncurry f2) (Uncurry g2) ->
  dominated (nFilterType domain M)
    (Fun' (fun p : Rtuple domain => (App f1 p * App f2 p)%I))
    (Fun' (fun p : Rtuple domain => (App g1 p * App g2 p)%I))
```

С помощью этих утверждений, а также некоторых утверждений о свойствах фильтров, мы завершаем доказательство. С полной формализацие результата можно ознакомиться в соответствующей главе, а также в репозитории [7].

Глава 3

Заключение

В нашей работе мы на примере формальной верификации корректности алгоритма LCS и его асимптотики применили методы, описанные в [8] на практике, формально верифицировав утверждение 8. В процессе применения этой библиотеки, возникли некоторые проблемы с отсутствием квантора всеобщности для предиката кучи, а именно, без него не получается строить утверждения о двумерных, и, следовательно, многомерных списках. Одним из вариантов развития этой работы может послужить реализация предиката Hforall, а также доказательство его различных свойств и интеграция в тактику hsimpl. Этот предикат был бы очень полезен не только при работе с многомерными массивами, но и во многих других случаях. Еще одним интересным вариантом развития идей формальной верификации асимптотики алгоритмов является формальная верификация асимптотики вероятностных алгоритмов. В то время как формализация проверки корректности вероятностных алгоритмов изучается довольно активно ([2], [22], [6]), проверка асимптотики вероятностных алгоритмов до сих пор практически не рассматривалась. Рассмотренные в этой работе методы могут быть обобщены и на вероятностные алгоритмы.

Глава 4

Полная формализация результата

```
Set Implicit Arguments.
Require Import TLC.LibTactics.
Require Import TLC.LibListZ.
(* Load the CFML library, with time credits. *)
Require Import CFML.CFLibCredits.
Require Pervasives_ml.
Require Array_ml.
Require Import Pervasives_proof.
Require Import ArrayCredits_proof.
(* Load the big-O library. *)
Require Import Dominated.
Require Import UltimatelyGreater.
Require Import Monotonic.
Require Import LibZExtra.
Require Import DominatedNary.
Require Import LimitNary.
Require Import Generic.
(* Load the custom CFML tactics with support for big-Os *)
Require Import CFMLBigO.
(* Load the CF definitions. *)
Require Import Lcs_flat_ml.

Open Scope liblist_scope.

Local Ltac auto_tilde ::= try solve [ auto with maths | false; math ].

Local Ltac my_invert H := inversion H; subst; clear H.

Lemma cons_injective: forall {A} x (l1 l2: list A), l1 = l2 -> x :: l1 = x :: l2.
```

```
Proof.
  intros. rewrite H. reflexivity.
Qed.


Lemma take_plus_one: forall (i : nat) (l: list int),
  1 <= i <= length l -> take i l = take (i - 1) l & l[i-1].
Proof.
  intros. generalize dependent i. induction l.
  - intros. rewrite length_nil in H. auto_false~.
  - intros. rewrite take_cons_pos. destruct i. auto_false~. destruct i.
    rewrite take_zero. rewrite take_zero. rewrite app_nil_l.
    rewrite read_zero.  reflexivity. rewrite take_cons_pos.
    rewrite read_cons_case. case_if. auto_false~. rewrite last_cons.
    apply cons_injective. remember (S (S i) - 1) as i'.
    remember (to_nat i') as i''.
    assert (i' = i''). rewrite Heqi''. symmetry. apply to_nat_nonneg. math.
    rewrite H0. apply IHl. rewrite <- H0. subst. rewrite length_cons in H. math.
    math. math.
Qed.

Lemma last_head: forall (l: list int), length l > 0 ->
  exists l' x, l = l' & x.
Proof.
  intros. exists (take (length l - 1) l) l[(length l) - 1].
  rewrite <- take_full_length at 1. apply take_plus_one. math.
Qed.

Inductive SubSeq {A:Type} : list A -> list A -> Prop :=
 | SubNil (l:list A) : SubSeq nil l
 | SubCons1 (x:A) (l1 l2:list A) (H: SubSeq l1 l2) : SubSeq l1 (x::l2)
 | SubCons2 (x:A) (l1 l2:list A) (H: SubSeq l1 l2) : SubSeq (x::l1) (x::l2).

Lemma subseq_cons: forall A l1 l2 (x : A), SubSeq (x::l1) l2 -> SubSeq l1 l2.
Proof.
  intros. remember (x::l1) as l1'. induction H.
  - discriminate Heql1'.
  - subst. constructor. apply IHSubSeq. reflexivity.
  - my_invert Heql1'. constructor. assumption.
Qed.
```

```
Lemma subseq_app: forall A l1 l2 (x : A), SubSeq l1 l2 -> SubSeq (l1 & x) (l2 & x).
Proof.
  intros. induction H.
  - induction l.
    + rewrite last_nil. apply SubCons2. apply SubNil.
    + rewrite last_cons. apply SubCons1. assumption.
  - rewrite last_cons. apply SubCons1. assumption.
  - rewrite last_cons. rewrite last_cons. apply SubCons2. assumption.
Qed.

Lemma subseq_nil: forall A (l : list A), SubSeq l nil -> l = nil.
Proof.
  intros. my_invert H. reflexivity.
Qed.

Lemma subseq_length: forall (l a: list int), SubSeq l a -> length l <= length a.
Proof.
  intros l. induction l.
  - intros. rewrite length_nil. math.
  - intros. my_invert H.
    * apply subseq_cons in H0. apply IHl in H0.
      rewrite length_cons. rewrite length_cons. math.
    * apply IHl in H3.
      rewrite length_cons. rewrite length_cons. math.
Qed.

Lemma subseq_cons_l: forall A l1 l2 (x : A),  SubSeq (x :: l1) l2 <->
  exists l2' l2'', l2 = l2' & x ++ l2'' /\ SubSeq l1 l2''.
Proof.
  split. generalize dependent x. generalize dependent l2.
  {induction l1.
  - intros l2. induction l2.
    + intros. my_invert H.
    + intros. my_invert H.
      * apply IHl2 in H2. destruct H2 as [l2' [l2'' [H2 H3]]].
        exists (a :: l2') l2''. rewrite H2. auto.
      * exists (@nil A) l2. auto.
  - intros l2. induction l2.
    + intros. my_invert H.
```

```
    + intros. my_invert H.
      * apply IHl2 in H2. destruct H2 as [l2' [l2'' [H2 H3]]].
        exists (a0 :: l2') l2''. rewrite H2. auto.
      * exists (@nil A) l2. auto.
  }
  {
    intros H. destruct H as [l2' [l2'' [H1 H2]]]. rewrite H1. generalize dependent l2. i
    - intros. rewrite last_nil. apply SubCons2. auto.
    - intros. destruct l2. discriminate.
      assert ((a :: l2') & x ++ l2'' = a :: (l2' & x ++ l2'')). reflexivity.
      rewrite H in H1. injection H1 as H1. apply IHl2' in H0. rewrite H.
      apply SubCons1. auto.
  }
Qed.


Lemma subseq_last_head: forall l1 l2 (x y : int),
  SubSeq (l1 & x) (l2 & y) -> SubSeq l1 l2.
Proof.
  intros l1.
  induction l1.
  - constructor.
  - intros. rewrite last_cons in H. apply subseq_cons_l in H.
    destruct H as [l2' [l2'' [H1 H2]]]. rewrite subseq_cons_l.
    lets H3: subseq_length H2. rewrite length_last in H3.
    assert (length l2'' > 0) by math.
    lets H5: last_head l2'' H. destruct H5 as [l' [z H5]]. rewrite H5 in H1.
    rewrite H5 in H2. apply IHl1 in H2. exists l2' l'. split.
    assert (l2' & a ++ l' & z = (l2' & a ++ l') & z). rewrite last_app.
    reflexivity.
    rewrite H0 in H1. apply last_eq_last_inv in H1. destruct H1 as [H1 _].
    rewrite H1.
    reflexivity. auto.
Qed.

Lemma subseq_app_r: forall (l l1 l2: list int), SubSeq l l1 -> SubSeq l (l1 ++ l2).
Proof.
  intros l. induction l.
  - constructor.
  - intros. rewrite subseq_cons_l in H. rewrite subseq_cons_l.
```

```
    destruct H as [l2' [l2'' [H1 H2]]]. apply (IHl l2'' l2) in H2.
    exists l2' (l2'' ++ l2). split. rewrite H1. rew_list. reflexivity. auto.
Qed.

Lemma subseq_last_case: forall l l1 (x : int), SubSeq l (l1 & x) ->
  (SubSeq l l1) \/ (exists l', l = l' & x /\ (SubSeq l' l1)).
Proof.
  intros l. induction l.
  - left. constructor.
  - intros. rewrite subseq_cons_l in H.
    destruct H as [l2' [l2'' [H1 H2]]]. destruct l2''.
    * apply subseq_nil in H2. rewrite app_nil_r in H1. subst.
      apply last_eq_last_inv in H1. right. exists (@nil int).
      destruct H1 as [H1 H2]. rewrite H2. split. rew_list. auto. constructor.
    * remember (z :: l2'') as ll. assert (length ll > 0).
      rewrite Heqll. rewrite length_cons. math.
      lets M: (last_head ll H). destruct M as [l' [y H3]]. rewrite H3 in H1.
      assert (l2' & a ++ l' & y = (l2' & a ++ l') & y). rewrite last_app. reflexivi
      rewrite H0 in H1. apply last_eq_last_inv in H1. destruct H1 as [H1 H4].
      rewrite H3 in H2. apply IHl in H2. destruct H2.
      + left. rewrite subseq_cons_l. exists l2' l'. split; auto.
      + destruct H2 as [l'0 [H2 H2']]. right. exists (a :: l'0). split.
        rewrite last_cons. f_equal. rewrite H4. auto. rewrite subseq_cons_l.
        exists l2' l'. split; auto.
Qed.

Lemma subseq_last_neq: forall l l1 l2 (x y : int), x <> y -> SubSeq l (l1 & x) ->
  SubSeq l (l2 & y) -> (SubSeq l l1) \/ (SubSeq l l2).
Proof.
  intros. apply subseq_last_case in H0. destruct H0.
  - left. auto.
  - destruct H0 as [l' [H01 H02]]. apply subseq_last_case in H1. destruct H1.
    + right. auto.
    + destruct H0 as [l'' [H11 H12]]. rewrite H01 in H11.
      apply last_eq_last_inv in H11. destruct H11 as [H1 H2]. auto_false.
Qed.

Definition Lcs {A: Type} l l1 l2 :=
  SubSeq l l1 /\ SubSeq l l2 /\
  (forall l': list A, SubSeq l' l1 /\ SubSeq l' l2 -> length l' <= length l).
```

```
Lemma lcs_nil_nil: forall A (l: list A), Lcs nil nil l.
Proof.
  intros. unfold Lcs. split. constructor. split. constructor. intros. destruct H as [H _
  apply subseq_nil in H. rewrite H. rewrite length_nil. math.
Qed.

Lemma lcs_symm: forall A (l l1 l2 : list A), Lcs l l1 l2 <-> Lcs l l2 l1.
Proof.
  intros. split.
  - unfold Lcs. intros[H1 [H2 H3]]. split. auto. split. auto.
    intros l' [H4 H5]. specialize (H3 l'). apply H3. split; auto.
  - unfold Lcs. intros[H1 [H2 H3]]. split. auto. split. auto.
    intros l' [H4 H5]. specialize (H3 l'). apply H3. split; auto.
Qed.

Lemma lcs_app_eq: forall (l1 l2 l: list int) (x: int),
  Lcs l l1 l2 -> Lcs (l & x) (l1 & x) (l2 & x).
Proof.
  unfold Lcs. intros. destruct H as [H1 [H2 H3]]. split.
  apply subseq_app. assumption. split.
  apply subseq_app. assumption.
  intros. destruct l'. rewrite length_nil. math.
  remember (z :: l') as l''.
  assert (HM: length l'' > 0). rewrite Heql''. rewrite length_cons. math.
  lets M: last_head l'' HM. destruct M as [ll [y M]]. rewrite M.
  rewrite length_last. rewrite length_last. destruct H as [Hl1 Hl2].
  rewrite M in Hl1. rewrite M in Hl2.
  apply subseq_last_head in Hl1. apply subseq_last_head in Hl2.
  assert (Hll: length ll <= length l) by auto. math.
Qed.

Lemma lcs_app_neq: forall (l1 l2 l l': list int) (x y : int),
  x <> y -> Lcs l (l1&x) l2 -> Lcs l' l1 (l2&y) -> length l' <= length l ->
  Lcs l (l1&x) (l2&y).
Proof.
  unfold Lcs. intros. destruct H0 as [Hl_1 [Hl_2 Hl_3]].
  destruct H1 as [Hl'_1 [Hl'_2 Hl'_3]].
  split. auto. split. apply subseq_app_r. auto.
  intros. destruct H0 as [H01 H02].
```

```
  assert (H' := H01).
  eapply subseq_last_neq in H01.
  3: {apply H02. } 2: {auto. } destruct H01.
  - apply H1_3. split; auto.
  - assert (length l'0 <= length l' -> length l'0 <= length l) by math.
    apply H1. apply H1'_3. split; auto.
Qed.



Definition ZZle (p1 p2 : Z * Z) :=
  let (x1, y1) := p1 in
  let (x2, y2) := p2 in
  1 <= x1 <= x2 /\ 0 <= y1 <= y2.

Lemma lcs_spec:
  specO
    (product_filterType Z_filterType Z_filterType)
    ZZle
  ( fun cost =>
  forall (l1 l2 : list int) p1 p2,
  app lcs [p1 p2]
  PRE (\$(cost (LibListZ.length l1, LibListZ.length l2)) \*
  p1 ~> Array l1 \* p2 ~> Array l2)
  POST (fun p => Hexists (l : list int), p ~> Array l \* \[Lcs l l1 l2]))
  (fun '(n,m) => n * m).
Proof.
  xspecO_refine straight_line. xcf.
  xpay.  xapp~. intros. xapp~. intros. rewrite <- H. rewrite <- H0. xapp~.
  assert (0 <= length l1) by (apply length_nonneg).
  assert (0 <= length l2) by (apply length_nonneg).
  rewrite <- H in H1.
  rewrite <- H0 in H2.
  { math_nia. }
  intros. weaken.
  xfor_inv (fun (i:int) =>
    Hexists (x' : list (list int)),
    p1 ~> Array l1 \*
    p2 ~> Array l2 \*
    c ~> Array x' \*
```

```
    \[length x' = (n+1)*(m+1)] \*
    \[forall i1 i2 : int, 0 <= i1 < i -> 0 <= i2 <= m ->
        Lcs x'[i1*(m+1) + i2] (take i1 l1 ) (take i2 l2) ] \*
    \[forall i', i*(m+1) <= i' < (n+1)*(m+1) ->
        x'[i'] = nil ]
    ).
  { math_nia. }
  2: {
    hsimpl.
    - intros. rewrite H1. rewrite read_make. reflexivity. apply~ int_index_prove.
    - intros. assert (0 <= i1 < 1 -> i1 = 0) by math_nia. apply H4 in H2.
      rewrite H2. rewrite take_zero. rewrite H1. rewrite read_make.
      apply lcs_nil_nil. apply~ int_index_prove. math_nia.
    - rewrite H1. apply length_make. math_nia.
  }
  2: {
    intros. xapp~. apply~ int_index_prove. math_nia.
    intros. xapp~. hsimpl_credits. specialize (H3 n m).
    rewrite take_ge in H3. rewrite take_ge in H3.
    assert (((n * (m + 1)%I)%I + m)%I = (((n + 1)%I * (m + 1)%I)%I - 1)%I) by math_nia.
    rewrite H6 in H3. rewrite H5. apply H3. math_nia. math_nia. math_nia. math_nia.
  }
  intros. xpay. xpull. intros.
  xfor_inv (fun (j:int) =>
    Hexists (x' : list (list int)),
    p1 ~> Array l1 \*
    p2 ~> Array l2 \*
    c ~> Array x' \*
    \[length x' = (n+1)*(m+1)] \*
    \[forall i1 i2 : int, 0 <= i1 <= i -> 0 <= i2 <= m ->
        i1*(m+1) + i2 < i*(m+1) + j ->
        Lcs x'[i1*(m+1) + i2] (take i1 l1 ) (take i2 l2) ] \*
    \[forall i', i*(m+1) + j <= i' < (n+1)*(m+1) ->
        x'[i'] = nil ]
    ).
  { math_nia. }
  2: {
    hsimpl.
    - intros. apply H5. math_nia.
    - intros.
```

```
    remember (to_nat i1) as i1'.
    remember (to_nat i) as i'.
    assert (i1 = i1'). rewrite Heqi1'. symmetry. apply to_nat_nonneg. math.
    assert (i = i'). rewrite Heqi'. symmetry. apply to_nat_nonneg. math.
    assert ((i1' <= i')%nat) by math.
    apply le_lt_eq_dec in H11.
    destruct H11.
    + assert (i1 < i) by math. clear Heqi1' Heqi' l H9 H10 i' i1'.
      apply H4; math.
    + assert (i1 = i) by math. clear Heqi1' Heqi' e H9 H10 i' i1'.
      rewrite lcs_symm. assert (x0[((i1 * (m + 1)%I)%I + 0)%I] = nil).
      apply H5. math_nia. asserts_rewrite (i2 = 0). math_nia.
      rewrite H9. apply lcs_nil_nil.
  - assumption.
}
2: {
  hsimpl.
  - intros. apply H9. math_nia.
  - intros. apply H8; math_nia.
  - assumption.
}
intros. xpay. xpull. intros.
xapp~. { apply~ int_index_prove. }
intros. xapp~. { apply~ int_index_prove. }
intros. xret. intros. xif.
{
  xapp~. { apply~ int_index_prove. }
  intros. xapp~.
  { apply~ int_index_prove. math_nia. math_nia. }
  intros. xret. intros. xapp~.
  { apply~ int_index_prove. math_nia. math_nia. }
  xpull. hsimpl_credits.
  - intros.
    remember (((i * (m + 1)) + i0)) as j.
    remember x11__ as v.
    assert ((x1[j:=v])[i'] = x1[i']).
    (* TODO: WHY THE HECK read_update_neq does not work??? *)
    rewrite read_update_case. case_if; auto_false~.
    apply~ int_index_prove. math_nia.
    rewrite H16. apply H9. math_nia.
```

```
    - intros.
      remember (to_nat i1) as i1'.
      remember (to_nat i) as i'.
      assert (i1 = i1'). rewrite Heqi1'. symmetry. apply to_nat_nonneg. math.
      assert (i = i'). rewrite Heqi'. symmetry. apply to_nat_nonneg. math.
      assert ((i1' <= i')%nat) by math.
      apply le_lt_eq_dec in H20.
      destruct H20.
      + assert (i1 < i) by math.
        rewrite read_update_case. case_if.
        assert (((i * (m + 1)%I)%I + i0)%I > ((i1 * (m + 1)%I)%I + i2)%I) by math_nia.
        auto_false~. apply H8; math_nia. apply int_index_prove; math_nia.
      + remember (to_nat i2) as i2'.
        remember (to_nat (i0 + 1)) as i0'.
        assert (i2 = i2'). rewrite Heqi2'. symmetry. apply to_nat_nonneg. math.
        assert (i0 + 1 = i0'). rewrite Heqi0'. symmetry. apply to_nat_nonneg. math.
        assert ((i2' < i0')%nat) by math_nia.
        apply le_lt_eq_dec in H22.
        destruct H22.
        * assert (i2 < i0) by math.
          rewrite read_update_case. case_if.
          assert (((i * (m + 1)%I)%I + i0)%I > ((i1 * (m + 1)%I)%I + i2)%I) by math_nia.
          auto_false~. apply H8; math_nia. apply int_index_prove; math_nia.
        * assert (i1 = i) by math. assert (i2 = i0) by math.
          rewrite read_update_case. case_if. rewrite H14.
          rewrite H22. rewrite H19. rewrite take_plus_one.
          rewrite <- H19. rewrite H20. rewrite take_plus_one. rewrite <- H20.
          rewrite H23. rewrite H12.
          rewrite <- H11. rewrite C. rewrite H10. rewrite <- H10.
          apply lcs_app_eq.
          rewrite H13.
          asserts_rewrite (((((i - 1)%I * (m + 1)%I)%I + i0)%I - 1)%I = (((i - 1)%I * (m
          math. apply H8; math_nia. math_nia. math_nia. apply~ int_index_prove; math_nia
    - rewrite <- H7. apply length_update.
  }
  {
    xapp~. { apply~ int_index_prove. math_nia. math_nia. }
    intros. xret. intros. xapp~.
    { apply~ int_index_prove. math_nia. math_nia. }
    intros. xret. intros. xif.
```

```
{
  xapp~.
  { apply~ int_index_prove. math_nia. math_nia. }
  intros. xapp~.
  { apply~ int_index_prove. math_nia. math_nia. }
  hsimpl_credits.
  {
    intros.
    rewrite read_update_case. case_if; auto_false~.
    apply int_index_prove; math_nia.
  }
  - intros.
    remember (to_nat i1) as i1'.
    remember (to_nat i) as i'.
    assert (i1 = i1'). rewrite Heqi1'. symmetry. apply to_nat_nonneg. math.
    assert (i = i'). rewrite Heqi'. symmetry. apply to_nat_nonneg. math.
    assert ((i1' <= i')%nat) by math.
    apply le_lt_eq_dec in H22.
    destruct H22.
    + assert (i1 < i) by math.
      rewrite read_update_case. case_if.
      assert (((i * (m + 1)%I)%I + i0)%I > ((i1 * (m + 1)%I)%I + i2)%I) by math
      auto_false~. apply H8; math_nia. apply int_index_prove; math_nia.
    + remember (to_nat i2) as i2'.
      remember (to_nat (i0 + 1)) as i0'.
      assert (i2 = i2'). rewrite Heqi2'. symmetry. apply to_nat_nonneg. math.
      assert (i0 + 1 = i0'). rewrite Heqi0'. symmetry. apply to_nat_nonneg. mat
      assert ((i2' < i0')%nat) by math_nia.
      apply le_lt_eq_dec in H24.
      destruct H24.
      * assert (i2 < i0) by math.
        rewrite read_update_case. case_if.
        assert (((i * (m + 1)%I)%I + i0)%I > ((i1 * (m + 1)%I)%I + i2)%I) by ma
        auto_false~. apply H8; math_nia. apply int_index_prove; math_nia.
      * assert (i1 = i) by math. assert (i2 = i0) by math.
        rewrite read_update_case. case_if. rewrite H16.
        rewrite H20. rewrite take_plus_one. rewrite <- H20.
        rewrite H22. rewrite take_plus_one. rewrite <- H22.
        rewrite H24. rewrite H25.
        rewrite <- H10. rewrite <- H11.
```

```
        (* rewrite <- H11. rewrite C. rewrite H10. rewrite <- H10.  *)
        rewrite lcs_symm.
        eapply lcs_app_neq. auto. rewrite H10.
        rewrite <- H25. rewrite H22. rewrite <- take_plus_one. rewrite lcs_symm.
        apply H8; math_nia. math_nia. rewrite lcs_symm. rewrite H11. rewrite H21.
        rewrite <- take_plus_one. rewrite <- H21. apply H8; math_nia. math_nia.
        asserts_rewrite (((i * (m + 1)%I)%I + (i0%I - 1)%I) = (((i * (m + 1)%I)%I +
        rewrite <- H12. rewrite <- H14. rewrite <- H13. rewrite <- H15. math. math.
        apply int_index_prove; math_nia.
    - rewrite <- H7. apply length_update.
  }
  {
    xapp~.
    { apply~ int_index_prove. math_nia. math_nia. }
    intros. xapp~.
    { apply~ int_index_prove. math_nia. math_nia. }
    hsimpl_credits.
    {
      intros.
      rewrite read_update_case. case_if; auto_false~.
      apply int_index_prove; math_nia.
    }
    - intros.
      remember (to_nat i1) as i1'.
      remember (to_nat i) as i'.
      assert (i1 = i1'). rewrite Heqi1'. symmetry. apply to_nat_nonneg. math.
      assert (i = i'). rewrite Heqi'. symmetry. apply to_nat_nonneg. math.
      assert ((i1' <= i')%nat) by math.
      apply le_lt_eq_dec in H22.
      destruct H22.
      + assert (i1 < i) by math.
        rewrite read_update_case. case_if.
        assert (((i * (m + 1)%I)%I + i0)%I > ((i1 * (m + 1)%I)%I + i2)%I) by math_nia.
        auto_false~. apply H8; math_nia. apply int_index_prove; math_nia.
      + remember (to_nat i2) as i2'.
        remember (to_nat (i0 + 1)) as i0'.
        assert (i2 = i2'). rewrite Heqi2'. symmetry. apply to_nat_nonneg. math.
        assert (i0 + 1 = i0'). rewrite Heqi0'. symmetry. apply to_nat_nonneg. math.
        assert ((i2' < i0')%nat) by math_nia.
        apply le_lt_eq_dec in H24.
```

```
        destruct H24.
        * assert (i2 < i0) by math.
          rewrite read_update_case. case_if.
          assert (((i * (m + 1)%I)%I + i0)%I > ((i1 * (m + 1)%I)%I + i2)%I) by ma
          auto_false~. apply H8; math_nia. apply int_index_prove; math_nia.
        * assert (i1 = i) by math. assert (i2 = i0) by math.
          rewrite read_update_case. case_if. rewrite H16.
          rewrite H20. rewrite take_plus_one. rewrite <- H20.
          rewrite H22. rewrite take_plus_one. rewrite <- H22.
          rewrite H24. rewrite H25.
          rewrite <- H10. rewrite <- H11.
          (* rewrite <- H11. rewrite C. rewrite H10. rewrite <- H10.  *)
          eapply lcs_app_neq. auto. rewrite H11. rewrite H21. rewrite <- take_plu
          asserts_rewrite ((((i * (m + 1)%I)%I + i0)%I - 1)%I = ((i * (m + 1)%I)%
          apply H8; math.  math.
          rewrite H10.
          rewrite <- H25. rewrite H22. rewrite <- take_plus_one. rewrite <- H22.
          apply H8; math_nia. math. rewrite <- H12. rewrite <- H14. rewrite <- H1
          apply int_index_prove; math_nia.
    - rewrite <- H7. apply length_update.
  }
}
reflexivity.
cleanup_cost.
{ equates 1; swap 1 2.
  { instantiate (1 := (fun '(x, y) => _)). apply fun_ext_1. intros [x y].
    rewrite !cumul_const'. rew_cost. reflexivity. }
  intros [x1 y1] [x2 y2] [H1 H2]. math_nia. }
apply_nary dominated_sum_distr_nary; swap 1 2.
dominated.
apply_nary dominated_sum_distr_nary.
apply_nary dominated_sum_distr_nary.
apply_nary dominated_sum_distr_nary.
dominated.
{ apply dominated_transitive with (fun '(x, y) => x * 1).
  - (* TODO: improve using some setoid rewrite instances? *)
    apply dominated_eq. intros [? ?]. math.
  - apply_nary dominated_mul_nary; dominated.
}
{ apply dominated_transitive with (fun '(x, y) => 1 * y).
```

```
    - (* TODO: improve using some setoid rewrite instances? *)
      apply dominated_eq. intros [? ?]. math.
    - apply_nary dominated_mul_nary; dominated.
  }

  { eapply dominated_transitive.
    apply dominated_product_swap.
    apply Product.dominated_big_sum_bound_with.
    { apply filter_universe_alt. intros. rewrite~ <-cumul_nonneg. math_lia. }
    { monotonic. }
    { limit.  }
    simpl. dominated.

    now repeat apply_nary dominated_sum_distr_nary; dominated.
    repeat apply_nary dominated_sum_distr_nary; dominated.
    etransitivity. apply Product.dominated_big_sum_bound_with.
    intros. apply filter_universe_alt. math_lia.
    monotonic. limit. dominated. apply_nary dominated_sum_distr_nary; dominated. }
Qed.
```

Список литературы

[1] Xavier Leroy et al. *The CompCert verified compiler.* URL: http://compcert.inria.fr/.

[2] Philippe Audebaud, Christine Paulin-Mohring. “Proofs of randomized algorithms in Coq”. *Science of Computer Programming* **74** 8 (2009). Special Issue on Mathematics of Program Construction (MPC 2006), с. 568—589. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.09.002. URL: https://www.sciencedirect.com/science/article/pii/S0167642309000240.

[3] Arthur Charguéraud. “Separation Logic for Sequential Programs (Functional Pearl)”. *Proc. ACM Program. Lang.* **4** ICFP (авг. 2020). DOI: 10.1145/3408998. URL: https://doi.org/10.1145/3408998.

[4] Thierry Coquand, Gérard Huet. “The calculus of constructions”. *Information and Computation* **76** 2 (1988), с. 95—120. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(88)90005-3. URL: https://www.sciencedirect.com/science/article/pii/0890540188900053.

[5] N. Dershowitz. *SOFTWARE HORROR STORIES.* URL: https://www.cs.tau.ac.il/~nachumd/verify/horror.html.

[6] Allyx Fontaine, Akka Zemmari. “Certified Impossibility Results and Analyses in Coq of Some Randomised Distributed Algorithms”. *Theoretical Aspects of Computing – ICTAC 2016.* под ред. Augusto Sampaio, Farn Wang. Cham: Springer International Publishing, 2016, с. 69—81. ISBN: 978-3-319-46750-4.

[7] Sergey Grigoryants. *LCS algorithm formal verification.* https://github.com/isergeyam/coq-bigO/blob/main/examples/proofs/Lcs_flat_proof.v. 2021.

[8] Armaël Guéneau, Arthur Charguéraud, François Pottier. “A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification”. *Programming Languages and Systems.* под ред. Amal Ahmed. Cham: Springer International Publishing, 2018, с. 533—560. ISBN: 978-3-319-89884-1.

[9] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. *Commun. ACM* **12** 10 (окт. 1969), с. 576—580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[10] William Alvin Howard. “The Formulae-as-Types Notion of Construction”. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* под ред. Haskell Curry и др. Academic Press, 1980.

[11] Gerwin Klein, Tobias Nipkow. “A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler”. *ACM Trans. Program. Lang. Syst.* **28** 4 (июль 2006), с. 619—695. ISSN: 0164-0925. DOI: `10.1145/1146809.1146811`. URL: `https://doi.org/10.1145/1146809.1146811`.

[12] Gerwin Klein, Tobias Nipkow. “Verified Bytecode Verifiers”. *TCS* **298** (2003), с. 583—626.

[13] Dirk Leinenbach, Wolfgang Paul, Elena Petrova. “Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctnes”. *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods.* SEFM ’05. USA: IEEE Computer Society, 2005, с. 2—12. ISBN: 0769524354. DOI: `10.1109/SEFM.2005.51`. URL: `https://doi.org/10.1109/SEFM.2005.51`.

[14] Xavier Leroy. “Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant”. *SIGPLAN Not.* **41** 1 (янв. 2006), с. 42—54. ISSN: 0362-1340. DOI: `10.1145/1111320.1111042`. URL: `https://doi.org/10.1145/1111320.1111042`.

[15] Xavier Leroy и др. “The OCaml system: Documentation and user’s manual”. *INRIA* **3** (), с. 42.

[16] Robin Milner, Mads Tofte, David Macqueen. *The Definition of Standard ML.* Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.

[17] Christine Paulin-Mohring. “Introduction to the Calculus of Inductive Constructions”. *All about Proofs, Proofs for All.* под ред. Bruno Woltzenlogel Paleo, David Delahaye. т. 55. Studies in Logic (Mathematical logic and foundations). College Publications, янв. 2015. URL: `https://hal.inria.fr/hal-01094195`.

[18] Benjamin C. Pierce. *Types and Programming Languages.* 1st. The MIT Press, 2002, с. 466. ISBN: 0262162091.

[19] J.C. Reynolds. “Separation logic: a logic for shared mutable data structures”. *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* 2002, с. 55—74. DOI: `10.1109/LICS.2002.1029817`.

[20] Ilya Sergey, Amrit Kumar, Aquinas Hobor. *Scilla: a Smart Contract Intermediate-Level LAnguage.* 2018. arXiv: `1801.00687 [cs.PL]`.

[21] Martin Strecker. *Compiler Verification for C0 (intermediate report).*

[22] Joseph Tassarotti. “Verifying concurrent randomized algorithms”. дис. … док. Carnegie Mellon University Pittsburgh, PA, 2017.

[23] The Coq Development Team. *The Coq Proof Assistant.* вер. 8.13. янв. 2021. DOI: `10.5281/zenodo.4501022`. URL: `https://doi.org/10.5281/zenodo.4501022`.