理论任务选题列表

- WhileDCL 的指称语义理论，和语义等价性质证明

  WhileDCL 语言是指在 While 语言上增加：(D) 解引用和取地址；(C) 控制流指令 continue、break 与 for 循环、do-wile 循环；(L) 引入局部变量。你需要在 Coq 中

  - 定义程序语言的语法；
  - 定义程序语言的指称语义；
  - 定义程序状态的 well-defineness 条件；
  - 基于上面定义再定义表达式和程序语句的行为等价；
  - 证明行为等价是等价关系，并证明所有语法算子都保持行为等价。

  第一档难度：在 WhileD 语言上完成上述任务。第二档难度：在 WhileDL 或 WhileDC 语言上完成上述任务。第三档难度：在 WhileDCL 语言上完成上述任务。第四档难度：在 WhileDCL 语言的基础上增加过程调用，并完成上述任务。

- 带输出序列的 Monad 上的 Hoare 逻辑推理规则

  - 第一档难度：不考虑计算过程中异常退出的情况，定义 ret 算子、bind 算子、choice 算子、assume 算子以及 repeat-break 算子，并证明它们的霍尔逻辑推理规则；
  - 第二档难度：在上面的基础上证明 ret 算子是 bind 算子的单位元，bind 算子具有结合律；
  - 第三档难度：在上面的基础上，(1) 应用 Kleen 不动点定理证明 repeat-break 算子与其展开一层之后等价 (2) 或额外考虑计算过程中可能异常退出的情况，重新定义所有算子并完成证明。

  基本定义见`SetMonadO.v`。

- Monad 上的证明 Dijkstra 图论算法的正确性

  第二档难度：证明算法的输出确实是最短路径的长度；第三档难度：在上面的基础上证明，如果 e 是 v 到 u 的一条边，并且 $d(s,u) = d(s,v) + length(e)$，那么存在一条 s 到 u 的最短路径，其最后一条边是 e。第四档难度：修改程序，额外记录从源点出发到每个点的最短路中的前一个节点，证明最后可以构建一条最短路。详细信息请查看 `GraphAlg.zip`，算法的 monad 描述可在 algorithms 目录下找到。

- Monad 上的证明 Floyd 图论算法的正确性

  第二档难度：证明算法的输出确实是最短路径的长度；第三档难度：在上面的基础上证明，如果 e 是 v 到 u 的一条边，并且 $d(s,u) = d(s,v) + length(e)$，那么存在一条 s 到 u 的最短路径，其最后一条边是 e。第四档难度：修改程序，额外记录一些信息使得最后可以构建出最短路，证明它的正确性。详细信息请查看`GraphAlg.zip`，算法的 monad 描述可在 algorithms 目录下找到。

- Monad 上的证明 Prim 图论算法的正确性

  第一档难度：证明如果原图连通，那么 Prim 算法最终找到的是树；第三档难度：证明如果原图连通，那么 Prim 算法找到的一定是最小生成树。详细信息请查看`GraphAlg.zip`，算法的 monad 描述可在 algorithms 目录下找到。

- Monad 上的证明 Kruskal 图论算法的正确性

  第一档难度：证明如果原图连通，那么 Kruskal 算法最终找到的是树；第三档难度：证明如果原图连通，那么 Kruskal 算法找到的一定是最小生成树。详细信息请查看 `GraphAlg.zip`，算法的 monad 描述可在 algorithms 目录下找到。

- Monad 上的证明最长升序子序列的计算正确性

  求最长上升子序列的 monad 描述如下：

```
Definition LIS (l: list Z) (least: Z): program (Z * list Z) :=
  st <- list_iter
          (fun n st =>
             '(n0, len0, l0) <- max_object_of_subset
                                  Z.le
                                  (fun '(n0, len0, l0) => In (n0, len0, l0) st /\ n0 < n)
                                  (fun '(n0, len0, l0) => len0);;
             ret (st ++ [(n, len0 + 1, l0 ++ [n])]))
          l
          [(least, 0, [])];;
  '(n0, len0, l0) <- max_object_of_subset
                       Z.le
                       (fun '(n0, len0, l0) => In (n0, len0, l0) st)
                       (fun '(n0, len0, l0) => len0);;
  ret (len0, l0).
```

  该程序用 `list_iter` 算子进行动态规划，其中 `(n0, len0, l0)` 表示以 `n0` 为末尾元素的最长上升子序列为 `l0` 其长度为 `n0` 。本题证明中可能要用到子序列 `is_subsequence` 的定义，这可以在附件 ListLib 目录 General 子目录下的 IndexedElements.v 文件中找到。本题定义的“取最大值”来自于一个简单的最大值最小值库（见附件中 MaxMinLib），库中也有一些有用的性质。另外，附件中提供的 monadlib 包含了一些 monad 验证的自动化指令。具体信息见附件中根目录 README 和子目录 README。上述算法定义在 algorithms 目录下。详见`ListAlgTasks.zip`。

  第二档难度：证明算法的第一项输出确实是最长上升子序列的长度；第三档难度：在上面的基础上证明，算法的第二项输出确实是最长上升子序列之一；第四档难度：修改算法，并证明算法的输出是所有最长上升子序列中下标字典序最小的。

- Monad 上的证明算法正确性：选择最多的区间使其不相交

  算法的输入是右端点递增的一列闭区间，算法使用贪心法求出其中的闭区间，使得这些闭区间两两不交。以下是算法的 monad 描述：

```
Definition max_interval (l: list (Z * Z)) (leftmost: Z):
  program (Z * list (Z * Z)) :=
  '(leftmost0, size0, ans0) <- list_iter
                                 (fun '(l, r) =>
                                   fun '(leftmost0, size0, ans0) =>
                                     choice
                                       (assume (l <= leftmost0);;
                                        ret (leftmost0, size0, ans0))
                                       (assume (l > leftmost0);;
                                        ret (r, size0 + 1, ans0 ++ [(l, r)])))
                                 l
                                 (leftmost, 0, []);;
  ret (size0, ans0).
```

该算法中 `leftmost0` 表示现在已经选出的区间中，最右一个的右端点；`size0` 表示目前为止选出的闭区间数量；`ans0` 表示具体选出的闭区间。本题证明中可能要用到子序列 `is_subsequence` 的定义，这可以在附件 ListLib 目录 General 子目录下的 IndexedElements.v 文件中找到。本题的证明中可以使用一个我们提供的最大值最小值库（见附件中 MaxMinLib），库中也有一些有用的性质。另外，附件中提供的 monadlib 包含了一些 monad 验证的自动化指令。具体信息见附件中根目录 README 和子目录 README。上述算法定义在 algorithms 目录下。详见`ListAlgTasks.zip`。

第二档难度：证明算法的第一项确实是可选区间的最大数量；第三档难度：在上面的基础上证明，算法的第二项输出确实是使得所选区间数量最多的一组方案；第四档难度：证明算法的第二项输出是所有最佳方案中，区间编号字典序最小的方案。

- Monad 上的证明算法正确性：选择序列中互不相邻的一组元素使其和最大

  算法的输入是一个整数序列，算法求出一个子序列使得其中任意两个在原序列中互不相邻。以下是算法的 monad 描述：

```
Definition max_sum (l: list Z): program (Z * list Z) :=
  '(max1, ans1, _, _) <- list_iter
                           (fun n =>
                              fun '(max1, ans1, max2, ans2) =>
                                choice
                                  (assume (max1 <= max2 + n);;
                                   ret (max2 + n, ans1 ++ [n], max1, ans1))
                                  (assume (max1 >= max2 + n);;
                                   ret (max1, ans1, max1, ans1)))
                           l
                           (0, [], 0, []);;
  ret (max1, ans1).
```

该算法中 `max1` 表示目前为止能选出的子序列和最大值，`ans1` 表示取到这个最大值的子序列，`max2` 表示在不能取当前整数的情况下能选出的子序列和最大值，`ans2` 表示取到这个最大值的解。本题证明中可能要用到子序列 `is_subsequence` 的定义，这可以在附件 ListLib 目录 General 子目录下的 IndexedElements.v 文件中找到。本题的证明中可以使用一个我们提供的最大值最小值库（见附件中 MaxMinLib），库中也有一些有用的性质。另外，附件中提供的 monadlib 包含了一些 monad 验证的自动化指令。具体信息见附件中根目录 README 和子目录 README。上述算法定义在 algorithms 目录下。详见`ListAlgTasks.zip`。

第二档难度：证明算法的第一项确实是满足条件的子序列中子序列和的最大值；第三档难度：在上面的基础上证明，算法的第二项输出确实是一组使得和最大的可行方案；第四档难度：修改算法，并证明算法的输出是所有最优解中下标字典序最小的。

- 在验证工具上验证大整数加减运算 C 程序实现

  本项目将提供需要验证的 C 程序，该 C 程序是 mini-gmp 库的一部分。minigmp 是一个大整数运算库，你需要验证该库中的大整数加法和减法实现中部分 C 函数的正确性。代码中的主要数据结构如下：

```
typedef struct __mpz_struct
{
    int _mp_alloc;        /* Number of *limbs* allocated and pointed
                             to by the _mp_d field.  */
    int _mp_size;         /* abs(_mp_size) is the number of limbs the
                             last field points to.  If _mp_size is
                             negative this is a negative number.  */
    unsigned int *_mp_d;  /* Pointer to the limbs.  */
} __mpz_struct;

typedef __mpz_struct * mpz_t;
```

  具体的数据结构定义参见附件中的 mini-gmp.h。本项目将提供验证需要用到的一些谓词定义，其中最主要的定义如下：

```
(** 一列整数存储在数组中 *)
Definition mpd_store_list (ptr: addr) (data: list Z): Assertion :=
  UIntArray.full ptr (Zlength data) data.
```

```
(** 一列整数在Base进制下表示的大整数 *)
Fixpoint list_to_Z (data: list Z): Z :=
  match data with
    | nil => 0
    | a :: l0 => (list_to_Z l0) * Base + a
  end.
```

```
(** 一列整数都是Base进制下的数码 *)
Fixpoint list_within_bound (data: list Z): Prop :=
  match data with
   | nil => True
   | a :: l0 => 0 <= a < Base /\ (list_within_bound l0)
  end.
```

  另外，本项目还提供了这些谓词的一些重要性质证明。例如：

```
Lemma list_to_Z_app: forall l1 l2,
  list_to_Z (l1 ++ l2) =
  list_to_Z l1 + list_to_Z l2 * (Base ^ (Zlength l1)).
```

```
Lemma list_within_bound_concat: forall (l1 l2: list Z),
  list_within_bound l1 ->
  list_within_bound l2 ->
  list_within_bound (l1 ++ l2).
```

  本项目需要你完成高精度减法的验证，或完整最终 minigmp 对外大整数运算接口函数的验证。验证过程中你需要为 C 程序添加必要的标注，并在 Coq 中完成最终的证明。

  - 任务一第二档难度：完成 `mpn_sub_1` 与 `mpn_sub_n` 的验证；

- 任务一第三档难度：在上面的基础上完成 `mpn_sub` 的验证；
- 任务二第三档难度：完成 `mpz_abs_add` 、 `mpz_abs_sub` 、 `mpz_add` 与 `mpz_sub` 的验证；
- 第四档难度：额外验证一个关于乘法的 C 函数；

详细信息请查看`minigmp.zip`。

- 在验证工具上验证哈希表的 C 程序实现

  本项目将提供需要验证的 C 程序，代码中的主要数据结构如下：

```
struct blist {
  char *key;
  unsigned int val;
  struct blist *next;
  struct blist *down;
  struct blist *up;
};
```

```
struct hashtbl {
  struct blist **bucks;
  struct blist *top;
};
```

在哈希表中，哈希值相同的元素用一个单链表存储（`next` 域），而所有元素又用一个双链表串在一起（`up` 域和 `down` 域），具体的数据结构实现参见附件中的 hashtbl.h 和 hashtbl.c。本项目将提供验证需要用到的一些谓词定义和它们的基本性质，这些内容的主要部分是关于要用到的单链表和双链表的。需要验证的 C 程序代码已经标注了前后条件。在这个项目中，你需要为分离逻辑谓词设计必要的 strategies，你需要为用到的基础函数标注前后条件（C 程序无实现，不需要证明），你还需要在验证中为 C 程序添加必要的标注，并在 Coq 中完成证明。请注意，附件材料中的 `store_hashtbl` 谓词只有第四档任务才需用到。

- 任务一第二档难度：完成 `create_hashtbl` 与 `hashtbl_add` 的验证；
- 任务一第三档难度：除了上面要求之外，完成 `hashtbl_find` 的验证；
- 任务一第二档难度：完成 `free_hashtbl` 与 `hashtbl_remove` 的验证；
- 任务二第三档难度：完成 `hashtbl_findref` 的验证；
- 第四档难度：在上面基础上，在 C 代码中标注关于 `store_hashtbl` 的前后条件以及断言推导方向，并在 Coq 中完成规约推导的验证。

详细信息请查看`HashTable.zip`。

- 本项目将提供需要验证的 C 程序，代码中的主要数据结构如下：

```
struct sll {
    unsigned int data;
    struct sll * next;
};
```

```
struct sllb {
    struct sll * head;
    struct sll ** ptail;
};
```

链表合并的关键是以下函数，其他都是普通的链表实现：

```
struct sllb * app_list_box(struct sllb * b1, struct sllb * b2)
{
    * (b1 -> ptail) = b2 -> head;
    if (b2 -> head != NULL)
    {
        b1 -> ptail = b2 -> ptail;
    }
    free_sllb(b2);
    return b1;
}
```

在本项目中，你需要自己定义分离逻辑谓词、为分离逻辑谓词设计必要的 strategies、为 C 函数标注前后条件、为 C 程序添加必要的标注（特别是循环不变量），并在 Coq 中完成证明（个别只提供了 C 函数接口而无实现的 C 函数不需要证明）。详细信息请查看`MergableList.zip`。

- 第二档难度：完成 `nil_list_box` 、 `cons_list_box` 、 `free_list_box` 以及 `app_list_box` 的验证；
- 第三档难度：除了上面要求之外，完成 `map_list_box` 的验证；
- 第四档难度：除了上面要求之外，完成 `sllb2array` 的验证；