CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Sections 10.1-10.3

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Fundamental Discrete Structures

Goal: to reason about ordered pairs, options, sets, relations, functions, and maps as fundamental data types in computer science, e.g., for modeling and verification of digital systems.

- ordered pairs
- options
- sets, relations, and functions
- *maps*

Ordered pairs

- Intuitively, an ordered pair of two objects a and b, often written in conventional notation as (a, b), is a pair of a and b in which order is significant:
  - a is the *first* element of the pair and
  - b is the second.
- It follows that, if $a \neq b$, then (a, b) and (b, a) are two distinct ordered pairs, even though they contain the same elements.
- Indeed, the fundamental property of ordered pairs is that (a_1, b_1) is the same pair as (a_2, b_2) iff $a_1 = a_2$ and $b_1 = b_2$.
- When we formalize ordered pairs as an algebraic datatype, this property follows directly from the datatype's axioms.

Ordered pairs

- We introduce ordered pairs with a polymorphic datatype `Pair`, which has only one constructor, the binary symbol `pair`.
- We also declare two selectors, one for retrieving the left and another for retrieving the right element of an ordered pair:

```
datatype (Pair S T) := (pair pair-left:S pair-right:T)
```

- All of the code for ordered pairs is introduced inside a predefined module `Pair` that is loaded when Athena first starts:

```
module Pair {
    ...
}
```

- Because ordered pairs are so widely used, this module is by default opened at the top level.

Ordered pairs

- We introduce the symbol `@` as an alias for the constructor `pair`, so we can write pairs (a, b) in infix notation as $(a$ `@` $b)$.
- We want the pair constructor to bind quite tightly, so we give it a high precedence level:

```
set-precedence pair 310

define @ := pair
```

- Occasionally we also want to write ordered pairs using square-bracket notation, writing `[1 2]` for `(pair 1 2)`, for example.
- Conversely, we will want to transform `(pair` s t`)` terms into two-element Athena lists of the form `[`s t`]`.
- The procedures `lst->pair` and `pair->lst` let us do both.

Ordered pairs

- Finally, we define some variable abbreviations and assert the relevant datatype and selector axioms:

```
define [x y z a b w p p1 p2] :=
       [?x ?y ?z ?a ?b ?w ?p:(Pair 'S 'T) ?p1:(Pair 'S1 'T1)
        ?p2:(Pair 'S2 'T2)]

assert pair-axioms :=
  (datatype-axioms "Pair" joined-with selector-axioms "Pair")
```

- No-confusion, no-junk axioms, as well as axioms for `pair-left` and `pair-right`.

Ordered pairs

Let us now prove a few theorems:

```
conclude pair-theorem-1 :=
    (forall p . p = (pair-left p) @ (pair-right p))
  datatype-cases pair-theorem-1 {
    (p as (pair x y)) =>
     (!chain [p = ((pair-left p) @ (pair-right p))  [pair-axioms]])
  }

conclude pair-theorem-2 :=
    (forall x y z w . x @ y = z @ w <==> y @ x = w @ z)
  pick-any x y z w
    (!chain [(x @ y = z @ w) <==> (x = z & y = w)    [pair-axioms]
                             <==> (y = w & x = z)    [prop-taut]
                             <==> (y @ x = w @ z)    [pair-axioms]])
```

Ordered pairs

We introduce a swap function that swaps the elements of an ordered pair:

```
declare swap:  (S, T) [(Pair S T)] -> (Pair T S)
assert* swap-def := [(swap x @ y = y @ x)]

> (eval swap swap 1 @ 'a)
Term: (pair 1 'a)

> (eval swap swap swap 1 @ 'a)
Term: (pair 'a 1)

conclude swap-theorem-1 :=
  (forall x y . swap swap x @ y = x @ y)
pick-any x y
  (!chain [(swap swap x @ y) = (swap y @ x) [swap-def]
                             = (x @ y)      [swap-def]])
```

Options

Option values are generated by the following polymorphic datatype:

```
datatype (Option S) := NONE | (SOME option-val:S)
```

Intuitively, this definition says that a value of sort (Option S) is either

- NONE or
- (SOME v) for some value v of sort S.

In the latter case, applying the selector function option-val to (SOME v) retrieves the value v.

The symbol SOME is given low precedence (110), as we want its scope to extend as far to the right as possible, as illustrated below.

```
> (SOME 'foo :: nil)

Term: (SOME (:: 'foo
                nil:(List Ide)))
```

Options

Options are useful in a number of contexts, most notably for error handling:

- If a function f that is normally supposed to produce a value of sort S generates an error, it can instead return `NONE` as its output; that will explicitly indicate the occurrence of an error.
- If, on the other hand, all goes well and f successfully computes a result v of sort S, it can return the output `(SOME` v`)`, indicating that the computation went well and resulted in v.
- Hence, to make f deal with errors explicitly, we modify its output sort by lifting it from S to `(Option` S`)`.

Options

It is important to keep in mind the datatype axioms for `Option`:

1. `NONE` is distinct from any value of the form `(SOME` v`)`;
2. `(SOME` v_1`)` is identical to `(SOME` v_2`)` iff v_1 is identical to v_2; and
3. every option value is either equal to `NONE` or to a value of the form `(SOME` v`)`.

For reference convenience, we give the name `opt-axioms` to the list of these axioms:

```
assert opt-axioms := (datatype-axioms "Option")
```

```
define [opt-no-confusion opt-some-injective opt-no-junk] := opt-axioms
```

Options

Lemmas derived inside `Options` module opened at the top level.

```
define [x y z] := [?x ?y ?z]

conclude option-lemma-1 := (forall x y . x = SOME y ==> x =/= NONE)
  pick-any x y
    assume hyp := (x = SOME y)
      (!chain-> [true ==> (NONE =/= SOME y)  [opt-no-confusion]
                      ==> (NONE =/= x)       [hyp]
                      ==> (x =/= NONE)       [sym]])

conclude option-lemma-2 :=
          (forall x . x =/= NONE ==> exists y . x = SOME y)
  pick-any x
    assume hyp := (x =/= NONE)
      (!dsyl (!uspec opt-no-junk x) hyp)

define option-lemma-2-conv :=
  (forall x . (forall y . x =/= SOME y) ==> x = NONE)
```

Options

```
define option-lemma-3 :=
  (forall x . x = NONE ==> ~ exists y . x = SOME y)

define option-lemma-4 :=
   (forall x y . x = NONE ==> x =/= SOME y)

conclude option-lemma-5 :=
  (forall x y z . x = SOME y & y =/= z ==> x =/= SOME z)
 pick-any x y z
   assume h := (x = SOME y & y =/= z)
     (!chain-> [h ==> (y =/= z)           [right-and]
                  ==> (SOME y =/= SOME z) [opt-some-injective]
                  ==> (x =/= SOME z)      [h]])

define opt-lemmas := [option-lemma-1 option-lemma-2 option-lemma-2-conv
                      option-lemma-3 option-lemma-4 option-lemma-5]
define option-results := (join opt-axioms opt-lemmas)
```

Sets

Finite sets are introduced through a polymorphic structure `(Set S)` that is inductively generated from two constructors:

- a nullary constructor `null`, representing the empty set of sort `S`, and
- a binary constructor `insert`, which adds an element of sort `S` into a set of sort `(Set S)`.

```
structure (Set S) :=  null | (insert S (Set S))
```

This structure definition, and all subsequent code and proofs are in an Athena module named `Set`, part of the standard Athena library. Unlike `Options` and `Pair`, this module is not open by default.

Sets

Observe that the `Set` definition appears to be structurally identical to that of finite *lists*:

```
datatype (List S) :=  nil | (:: S (List S))
```

with the roles of `nil` and `::` played by `null` and `insert`, respectively. There is one crucial difference, however:

- Lists are freely generated (hence the `datatype` keyword), whereas
- Sets are only inductively generated (hence the `structure` keyword).

Free generation means that two lists are equal iff they are identical as terms. Thus, `(:: 1 (:: 2 nil))` and `(:: 2 (:: 1 nil))` represent different lists.

Sets

Free generation is reflected in the `datatype-axioms` of `List`, one of which is:

```
(forall h1 t1 h2 t2 . h1 :: t1 = h2 :: t2 ==> h1 = h2 & t1 = t2)
```

and whose contrapositive is:

```
(forall h1 t1 h2 t2 . h1 =/= h2 | t1 =/= t2 ==> h1 :: t1 =/= h2 :: t2)
```

By contrast, the analogous axiom for sets does not hold.
Indeed, we will have to ensure that `(insert 1 (insert 2 null))` and

`(insert 2 (insert 1 null))`

are identical in order to respect the *extensionality principle*, which states that two sets are equal iff they contain the same elements.

Sets

It will be convenient to be able to transform square-bracket-enclosed Athena lists into sets and vice versa, so we can write, for instance, `[1 2 3]` as a shorthand for the set

`(1 insert 2 insert 3 insert null)`.

The `Set` module introduces two such procedures, `alist->set` and `set->alist` (the latter also removes duplicates from the final list.) We expand the input space of `insert` so that it can accept Athena lists as second arguments:

```
expand-input insert [id alist->set]
```

And in the interest of brevity and readability, we define `++` as an alias for `insert`:

```
define ++ := insert
```

Sets

We can now write sets in a succinct and fairly natural notation:

```
> (1 ++ [2 3])

Term: (insert 1
              (insert 2
                      (insert 3
                              null:(Set Int))))
```

We also set the precedence of ++ to be relatively high, and define some abbreviations for term variables: x, y, z and h, h' as polymorphic variables of completely general sorts, and the rest specifically as (polymorphic) set variables:

```
set-precedence ++ 210

define [x y z h h' s s' t t' s1 s2 s3 A B C D E] :=
       [?x ... ?s:(Set 'T1) ... ?t:(Set 'T3) ... ?A:(Set 'T8) ...]
```

Set membership

The first relation we introduce is the most fundamental one: the set membership relation.

We expand its input space so that it can take square-bracket-enclosed lists as second arguments (which will be converted to set terms via `alist->set`).

And we define the relation recursively, much in the same way that a similar relation for list membership would be defined:

```
declare in: (S) [S (Set S)] -> Boolean [[id alist->set]]

assert* in-def :=
  [(~ _ in null)
   (x in h ++ t <==> x = h | x in t)]

define [in-not-in-null in-def-insert] := in-def
```

Set membership

The following two lemmas are simple but handy:

```
conclude null-characterization := (forall x . x in [] <==> false)
  pick-any x
    (!equiv
      assume hyp := (x in [])
        (!absurd hyp (!uspec in-not-in-null x))
      assume false
        (!from-false (x in [])))

conclude in-lemma-1 := (forall x A . x in x ++ A)
  pick-any x A
    (!chain-> [(x = x) ==> (x in x ++ A) [in-def-insert]])
```

We often use upper-case shorthands for characterization theorems:

```
define NC := null-characterization
```

We call this a “characterization” result for `null`, by the way, because it provides necessary and sufficient membership conditions for `null`.

The subset relation

- We continue with another fundamental relation, built on top of the membership relation: the *subset* relation, customarily represented by the symbol $\subseteq$.
- A set A is a subset of a set B iff every element of A is also an element of B:

$$\forall\, A, B \;.\; A \subseteq B \Leftrightarrow (\forall\, x \;.\; x \in A \Rightarrow x \in B). \qquad (1)$$

- We could define `subset` directly by asserting (1). However, such a definition would have a drawback:
  - It would not be constructive, due to the unrestricted universal quantifier that occurs on the right-hand side of the biconditional.

The subset relation

- Without a constructive definition, typically given by recursion, evaluation will not work; for example, `eval` will not be able to reduce a term like `([1] subset [2 1 3])` to `true`.
- We want `eval` to be able to handle such terms, not only in order to test our axioms, but also in order to test conjectures.
- For that reason, we will instead give a constructive definition of `subset`. Once we have such a definition, we will then *derive* (1) from it.
- This is a general strategy that we use, for instance, instead of defining set-theoretic union by directly asserting that an element x belongs to the union of A and B iff it belongs to A or it belongs to B, we instead give a recursive definition of union and then inductively derive the said property from it.

The subset relation

- In general, when there is a choice between a constructive definition D_1 and a nonconstructive definition D_2, we will typically prefer D_1 and then proceed to derive D_2 from D_1, usually by induction.
- In this case the required recursive definition can be given as:

```
declare subset: (S) [(Set S) (Set S)] -> Boolean [[alist->set alist->set]]

assert* subset-def :=
  [([] subset _)
   (h ++ t subset A <==> h in A & t subset A)]

> (eval [1 2] subset [3 2 4 1 5])
Term: true

> (eval [1 2] subset [3 2])
Term: false
```

The subset relation

- We now go on to derive the nonconstructive characterization of the subset relation from this definition.
- This is done in two steps, beginning with the left-to-right direction stating that if A is a subset of B (according to `subset-def`) then for all x, if x belongs to A then x also belongs to B. It is proved by structural induction, with basis case:

```
define subset-characterization-1 :=
  by-induction (forall A B . A subset B ==> forall x . x in A ==> x in B)
  {
    null => pick-any B
              assume (null subset B)
                pick-any x
                  (!chain [(x in null) ==> false        [NC]
                                       ==> (x in B)     [prop-taut]])
  | (A as (insert h t)) => ...
  }
```

The subset relation

and inductive step:

```
  | (A as (insert h t)) =>
      pick-any B
        assume hyp := (A subset B)
          pick-any x
            let {IH := (forall B . t subset B ==>
                                      forall x . x in t ==> x in B);
                 _ := (!chain-> [hyp ==> (t subset B) [subset-def]])}
              assume (x in A)
                (!cases (!chain<- [(x = h | x in t)
                                   <== (x in A)            [in-def]])
                  assume (x = h)
                    (!chain-> [hyp ==>  (h in B)       [subset-def]
                                   ==>  (x in B)       [(x = h)]])
                  (!chain [(x in t) ==> (x in B)       [IH]]))
```

The subset relation

The converse direction is likewise proved by structural induction. With both directions available, the classical (nonconstructive) subset characterization is easy to derive.

```
define subset-characterization-2 :=
  (forall A B . (forall x . x in A ==> x in B) ==> A subset B)

define subset-characterization :=
  (forall s1 s2 . s1 subset s2 <==> forall x . x in s1 ==> x in s2)
```

We define `SC` as a handy abbreviation for `subset-characterization`:

```
define SC := subset-characterization
```

The subset relation

It will be very useful to have an introduction method for goals of the form (*s* subset *t*), based on subset-characterization. The method will take a premise of the form

(forall x . x in *s* ==> x in *t*)

and will use SC to derive the desired goal (*s* subset *t*):

```
define subset-intro :=
  method (p)
    match p {
      (forall x ((x in A) ==> (x in B))) =>
        (!chain-> [p ==> (A subset B) [SC]])
    }
```

Set identity

Recall that when an inductively generated structure is not free, we need to define the identity relation on that structure; that is, we need to state exactly when two terms of that structure are to count as identical. In this case, the identity relation on sets is defined as should be expected: Two sets are equal iff each is a subset of the other.

```
assert* set-identity := (A = B <==> A subset B & B subset A)
```

Recall also that when a structural identity axiom of this kind is asserted, Athena automatically adjusts `eval` to ensure that it adheres to the new definition. Thus, we now have:

```
> (eval 1 ++ 2 ++ [] = 2 ++ 1 ++ [])

Term: true
```

If we did not have a constructive definition of `subset`, we would also not have a constructive set identity, and the `eval` call above would fail.

Set identity

An alternative and nonconstructive characterization of set identity is:

```
define set-identity-characterization :=
  (forall A B . A = B <==> forall x . x in A <==> x in B)

define SIC := set-identity-characterization
```

We can readily derive SIC from set-identity and SC.

Similar to subset-intro, we define a method set-identity-intro that takes two premises of the form (s subset t) and (t subset s) and derives (s = t):

```
define set-identity-intro :=
  method (p1 p2)
    match [p1 p2] {
      [(A subset B) (B subset A)] =>
        (!chain-> [p1 ==> (p1 & p2) [augment]
                      ==> (A = B)   [set-identity]])
    }
```

Set identity

A more direct introduction method, based on `SIC`, is also useful.

This one takes a premise of the form

$$\forall x \,.\, x \in A \Leftrightarrow x \in B$$

and derives the conclusion $A = B$:

```
define set-identity-intro-direct :=
  method (premise)
    match premise {
      (forall x ((x in A) <==> (x in B))) =>
        (!chain-> [premise ==> (A = B) [SIC]])
    }
```

Empty set characterizations

Using SIC we can also derive a couple of handy alternative characterizations of the empty set:

```
conclude NC-2 := (forall A . A = null <==> forall x . ~ x in A)
pick-any A
  (!chain [(A = null)
       <==> (forall x . x in A <==> x in null)   [SIC]
       <==> (forall x . x in A <==> false)       [NC]
       <==> (forall x . ~ x in A)                [prop-taut]])

define NC-3 := (forall A . A =/= null <==> exists x . x in A)
```

The following will be a convenient abbreviation, defined as a procedure:

```
define (non-empty s) := (s =/= null)
```

Subset relation properties

Some standard results about the subset relation can be readily derived now: reflexivity, antisymmetry, and transitivity.

```
define subset-reflexivity := (forall A . A subset A)

define subset-antisymmetry :=
  (forall A B . A subset B & B subset A ==> A = B)

conclude subset-transitivity :=
 (forall A B C . A subset B & B subset C ==> A subset C)
   pick-any A B C
     assume (A subset B & B subset C)
       (!subset-intro
          pick-any x
           (!chain [(x in A)
                ==> (x in B) [SC]
                ==> (x in C) [SC]]))
```

Proper subset relation

The *proper subset* relation is defined as follows:

```
declare proper-subset: (S) [(Set S) (Set S)] -> Boolean
                           [[alist->set alist->set]]

assert* proper-subset-def :=
  [(s1 proper-subset s2 <==> s1 subset s2 & s1 =/= s2)]

> (eval [1 2] proper-subset [2 3 1])

Term: true
```

Proper subset relation

A characterization theorem and some useful lemmas follow:

```
define proper-subset-characterization :=
  (forall s1 s2 . s1 proper-subset s2 <==>
                  s1 subset s2 & exists x . x in s2 & ~ x in s1)

define PSC := proper-subset-characterization

define proper-subset-lemma :=
  (forall A B x . A subset B & x in B & ~ x in A ==> A proper-subset B)

define in-lemma-2 := (forall h t . h in t ==> h ++ t = t)

define in-lemma-3 := (forall x h t . x in t ==> x in h ++ t)

define in-lemma-4 :=
  (forall A x y . x in A ==> y in A <==> y = x | y in A)
```

Singletons

We introduce an operation for forming singletons:

```
declare singleton: (S) [S] -> (Set S)

assert* singleton-def := [(singleton x = x ++ null)]

conclude singleton-characterization :=
  (forall x y . x in singleton y <==> x = y)
 pick-any x y
  (!chain [(x in singleton y)
       <==> (x in y ++ null)       [singleton-def]
       <==> (x = y | x in null)    [in-def]
       <==> (x = y | false)        [NC]
       <==> (x = y)                [prop-taut]])
```

Set operations

Next, we introduce three important binary set operations: union, intersection, and difference:

```
declare union, intersection, diff:
   (S) [(Set S) (Set S)] -> (Set S) [120 [alist->set alist->set]]

define [\/ /\ \] := [union intersection diff]
```

As we did with `subset`, we will define these operations constructively and then deductively derive their classical nonconstructive characterizations. We begin with union:

```
assert* union-def :=
 [([] \/ s = s)
  (h ++ t \/ s = h ++ (t \/ s))]
```

Set operations

We transform the output of `eval` so that sets are printed out in square-bracket notation:

```
transform-output eval [set->alist]
```

We use `eval` to test the definition:

```
> (eval [1 2 3] \/ [4 5 6 3])

List: [1 2 3 4 5 6]
```

The classical characterization of union is:

```
(forall A B x . x in A \/ B <==> x in A | x in B)
```
(2)

Set operations

We derive (2) (with shortcut UC) using equivalence chaining:

```
conclude union-characterization :=
  (forall A B x . x in A \/ B <==> x in A | x in B)
 by-induction union-characterization {
   null => pick-any B x
           (!chain [(x in null \/ B)
                    <==> (x in B)                 [union-def]
                    <==> (false | x in B)         [prop-taut]
                    <==> (x in null | x in B)     [NC]])
 | (A as (h insert t)) =>
    let {IH := (forall B x . x in t \/ B <==> x in t | x in B)}
      pick-any B x
        (!chain [(x in A \/ B)
                 <==> (x in h ++ (t \/ B))        [union-def]
                 <==> (x = h | x in t \/ B)       [in-def]
                 <==> (x = h | x in t | x in B)   [IH]
                 <==> ((x = h | x in t) | x in B) [prop-taut]
                 <==> (x in A | x in B)           [in-def]])
}
```

Set operations

We continue with intersection:

```
assert* intersection-def :=
 [(null /\ s = null)
  (h ++ t /\ A = h ++ (t /\ A) <== h in A)
  (h ++ t /\ A = t /\ A <== ~ h in A)]

> (eval [1 2 1] /\ [5 1 3])

List: [1]
```

The classical characterization of intersection follows:

```
(forall A B x . x in A /\ B <==> x in A & x in B)
```
(3)

Set operations

We derive (3) in two steps by induction, with the basis case and inductive case following:

```
conclude intersection-characterization-1 :=
  (forall A B x . x in A /\ B ==> x in A & x in B)
by-induction intersection-characterization-1 {
  null =>
    pick-any B x
      (!chain [(x in null /\ B)
           ==> (x in null)                    [intersection-def]
           ==> false                          [NC]
           ==> (x in null & x in B)           [prop-taut]])
| (A as (insert h t)) =>
    ...
}
```

Set operations

```
| (A as (insert h t)) =>
  let {IH := (forall B x . x in t /\ B ==> x in t & x in B)}
    pick-any B x
      (!two-cases
        assume (h in B)
          (!chain
            [(x in (h ++ t) /\ B)
          ==> (x in h ++ (t /\ B))                    [intersection-def]
          ==> (x = h | x in t /\ B)                   [in-def]
          ==> (x = h | x in t & x in B)               [IH]
          ==> ((x = h | x in t) & (x = h | x in B))   [prop-taut]
          ==> (x in A & (x in B | x in B))            [in-def (h in B)]
          ==> (x in A & x in B)                       [prop-taut]])
        assume (~ h in B)
          (!chain [(x in A /\ B)
                ==> (x in t /\ B)                     [intersection-def]
                ==> (x in t & x in B)                 [IH]
                ==> (x in A & x in B)                 [in-def]]))
```

Set operations

The `two-cases` analysis in the inductive case is dictated by the corresponding case split in `intersection-def`.

The classical characterization of intersection, which we will abbreviate as `IC`, is then readily derived from the two directions.

```
define intersection-characterization-2 :=
 (forall A B x . x in A & x in B ==> x in A /\ B)

define intersection-characterization :=
 (forall A B x . x in A /\ B <==> x in A & x in B)

define IC := intersection-characterization
```

Set operations

The following is an immediate corollary of `IC`:

```
conclude intersection-subset-theorem :=
  (forall A B . A /\ B subset A)
pick-any A B
  (!subset-intro
    pick-any x
      (!chain [(x in A /\ B)
           ==> (x in A)       [IC]]))
```

Set operations

Finally, we go through the same process for the set-theoretic difference operation:

```
assert* diff-def :=
 [(null \ _ = null)
  (h ++ t \ A = t \ A <== h in A)
  (h ++ t \ A = h ++ (t \ A) <== ~ h in A)]

> (eval [1 2 3] \ [3 1])

List: [2]
```

Set operations

We derive the following:

```
define diff-characterization-1 :=
  (forall A B x . x in A \ B ==> x in A & ~ x in B)

define diff-characterization-2 :=
  (forall A B x . x in A & ~ x in B ==> x in A \ B)

conclude diff-characterization :=
  (forall A B x . x in A \ B <==> x in A & ~ x in B)
  pick-any A B x
    (!equiv
      (!chain [(x in A \ B)
               ==> (x in A & ~ x in B) [diff-characterization-1]])
      (!chain [(x in A & ~ x in B)
               ==> (x in A \ B)        [diff-characterization-2]]))

define DC := diff-characterization
```

Set operations

We also introduce an operation that removes an element from a set, using - as an alias for it:

```
declare remove: (S) [(Set S) S] -> (Set S) [- [alist->set id]]

assert* remove-def :=
  [(null - _ = null)
   (h ++ t - x = t - x <== x = h)
   (h ++ t - x = h ++ (t - x) <== x =/= h)]

(eval [1 2 3 2 5] - 2)

> List: [1 3 5]
```

Set operations

The corresponding characterization theorem for this operation follows:

```
define remove-characterization :=
  (forall A x y . y in A - x <==> y in A & y =/= x)

define RC := remove-characterization
```

Some simple but useful corollaries of RC follow:

```
define remove-corollary :=   (forall A x  . ~ x in A - x)

define remove-corollary-2 := (forall A x  . ~ x in A ==> A - x = A)

define remove-corollary-3 :=
  (forall A x y . x in A & y =/= x ==> x in A - y)

define remove-corollary-4 := (forall A x y  . ~ x in A ==> ~ x in A - y)

define remove-corollary-5 :=
  (forall A B x . A subset B & ~ x in A ==> A subset B - x)
```

Set operations

We have now derived characterization theorems for most of the fundamental set-theoretic functions and relations: union, intersection, set difference, the subset and identity relations, and removal.
We can proceed to prove some important results, starting with the commutativity and associativity of union and intersection:

```
conclude intersection-commutes := (forall A B . A /\ B = B /\ A)
 pick-any A B
   (!set-identity-intro-direct
     pick-any x
       (!chain [(x in A /\ B) <==> (x in A & x in B)  [IC]
                              <==> (x in B & x in A)  [prop-taut]
                              <==> (x in B /\ A)      [IC]]))
```

The proof of union commutativity is similar.

```
define union-commutes := (forall A B . A \/ B = B \/ A)
```

Set operations

We proceed with the associativity of intersection and union:

```
conclude intersection-associativity :=
  (forall A B C . A /\ (B /\ C) = (A /\ B) /\ C)
 pick-any A B C
   (!set-identity-intro-direct
     pick-any x
       (!chain [(x in A /\ B /\ C)
            <==> (x in A & x in B /\ C)       [IC]
            <==> (x in A & x in B & x in C)   [IC]
            <==> ((x in A & x in B) & x in C) [prop-taut]
            <==> ((x in A /\ B) & x in C)     [IC]
            <==> (x in (A /\ B) /\ C)         [IC]]))

define union-associativity := (forall A B C . A \/ B \/ C = (A \/ B) \/ C)
```

Set operations

Idempotence is also useful to derive for both operations:

```
conclude /\-idempotence := (forall A . A /\ A = A)
  pick-any A
    (!set-identity-intro-direct
      pick-any x
        (!chain [(x in A /\ A)
                 <==> (x in A & x in A)  [IC]
                 <==> (x in A)           [prop-taut]]))
```

Union idempotence is likewise established, and defined as `\/-idempotence`.

Set operations

The following is a useful result specifying necessary and sufficient conditions for the union of two sets to be empty:

```
conclude union-null-theorem :=
  (forall A B . A \/ B = null <==> A = null & B = null)
pick-any A B
  (!chain [(A \/ B = null)
      <==> (forall x . x in A \/ B <==> x in null)       [SIC]
      <==> (forall x . x in A \/ B <==> false)           [NC]
      <==> (forall x . x in A | x in B <==> false)       [UC]
      <==> (forall x . ~ x in A & ~ x in B)              [prop-taut]
      <==> ((forall x . ~ x in A) & (forall x ~ x in B)) [quant-dist]
      <==> (A = null & B = null)                         [NC-2]])
```

Set operations

We continue with two distributivity theorems. The first one shows that union distributes over intersection:

$$\forall\, A, B, C \;.\; A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

```
conclude distributivity-1 :=
  (forall A B C . A \/ (B /\ C) = (A \/ B) /\ (A \/ C))
  pick-any A B C
    (!set-identity-intro-direct
       pick-any x
         (!chain [(x in A \/ (B /\ C))
             <==> (x in A | x in B /\ C)                  [UC]
             <==> (x in A | x in B & x in C)              [IC]
             <==> ((x in A | x in B) & (x in A | x in C)) [prop-taut]
             <==> (x in A \/ B & x in A \/ C)             [UC]
             <==> (x in (A \/ B) /\ (A \/ C))             [IC]]))
```

Set operations

Our next distributivity result shows that intersection distributes over union.

$$\forall\, A, B, C \;.\; A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

```
define distributivity-2 :=
  (forall A B C . A /\ (B \/ C) = (A /\ B) \/ (A /\ C))
```

Set operations

We continue with some theorems about set-theoretic difference:

```
conclude diff-theorem-1 := (forall A . A \ A = null)
  pick-any A
    (!set-identity-intro-direct
      pick-any x
        (!chain [(x in A \ A)
                 <==> (x in A & ~ x in A) [DC]
                 <==> false               [prop-taut]
                 <==> (x in null)         [NC]]))
```

Set operations

```
conclude diff-theorem-2 :=
  (forall A B C . B subset C ==> A \ C subset A \ B)
pick-any A B C
  assume (B subset C)
    (!subset-intro
      pick-any x
        (!chain [(x in A \ C)
                 ==> (x in A & ~ x in C) [DC]
                 ==> (x in A & ~ x in B) [SC]
                 ==> (x in A \ B)        [DC]]))
```

Set operations

We go on to derive a number of additional results relating the three fundamental set operations of union, intersection, and difference:

```
define diff-theorem-3 := (forall A B . A \ (A /\ B) = A \ B)

conclude diff-theorem-4 :=
  (forall A B . A /\ (A \ B) = A \ B)
    pick-any A B
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in A /\ (A \ B))
                <==> (x in A & x in A \ B)          [IC]
                <==> (x in A & x in A & ~ x in B)   [DC]
                <==> (x in A & ~ x in B)            [prop-taut]
                <==> (x in A \ B)                   [DC]]))
```

Set operations

```
define diff-theorem-5 := (forall A B . (A \ B) \/ B = A \/ B)
define diff-theorem-6 := (forall A B . (A \/ B) \ B = A \ B)
define diff-theorem-7 := (forall A B . (A /\ B) \ B = null)

conclude diff-theorem-8 :=
  (forall A B . (A \ B) /\ B = null)
    pick-any A B
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in (A \ B) /\ B)
              <==> (x in A \ B & x in B)          [IC]
              <==> ((x in A & ~ x in B) & x in B) [DC]
              <==> false                          [prop-taut]
              <==> (x in null)                    [NC]]))

define diff-theorem-9 :=
  (forall A B C . A \ (B \/ C) = (A \ B) /\ (A \ C))
```

Set operations

```
conclude diff-theorem-10 :=
  (forall A B C . A \ (B /\ C) = (A \ B) \/ (A \ C))
    pick-any A B C
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in A \ (B /\ C))
              <==> (x in A & ~ x in B /\ C)                    [DC]
              <==> (x in A & ~ (x in B & x in C))              [IC]
              <==> (x in A & (~ x in B | ~ x in C))          [prop-taut]
              <==> ((x in A & ~ x in B) | (x in A & ~ x in C)) [prop-taut]
              <==> (x in A \ B | x in A \ C)                   [DC]
              <==> (x in (A \ B) \/ (A \ C))                   [UC]]))

define diff-theorem-11 := (forall A B . A \ (A \ B) = A /\ B)
define diff-theorem-12 := (forall A B . A subset B ==> A \/ (B \ A) = B)
```

Set operations

```
conclude subset-theorem-1 :=
  (forall A B . A subset B ==> A \/ B = B)
pick-any A B
  assume (A subset B)
    (!set-identity-intro-direct
      pick-any x
        (!chain [(x in A \/ B)
            <==> (x in A | x in B) [UC]
            <==> (x in B | x in B) [prop-taut SC]
            <==> (x in B)          [prop-taut]]))

define subset-theorem-2 := (forall A B . A subset B ==> A /\ B = A)

define intersection-subset-theorem' :=
  (forall A B C . A subset B /\ C <==> A subset B & A subset C)
```

Set operations

```
conclude union-lemma-2 :=
  (forall A B x . x ++ (A \/ B) = A \/ x ++ B)
pick-any A B x
  (!chain [(x ++ (A \/ B))
         = (x ++ (B \/ A)) [union-commutes]
         = ((x ++ B) \/ A) [union-def]
         = (A \/ (x ++ B)) [union-commutes]])
```

Set operations

```
conclude absorption-1 :=
  (forall x A . x in A <==> x ++ A = A)
  pick-any x A
    (!equiv
      assume hyp := (x in A)
        (!set-identity-intro-direct
          pick-any y
            (!chain [(y in x ++ A)
                <==> (y = x | y in A)     [in-def]
                <==> (y in A | y in A)    [hyp prop-taut]
                <==> (y in A)             [prop-taut]]))
      assume (x ++ A = A)
        (!chain-> [true ==> (x in x ++ A) [in-lemma-1]
                        ==> (x in A)      [SIC]]))
```

Note that on the second step of the **equivalence** chain, we specify both hyp and `prop-taut` as justifiers. Neither would be sufficient by itself.

Set operations

The union analogue of `intersection-subset-theorem'` does not hold:

```
> (falsify (forall A B C . A subset B \/ C <==> A subset B | A subset C) 10)
List: ['success
        |{?A:(Set Int) :=
           insert 1
                  (insert 2
                          null:(Set Int)))
          ?B:(Set Int) := (insert 1
                                  null:(Set Int))
          ?C:(Set Int) := (insert 2
                                  null:(Set Int))
      }|]
```

However, a weaker version does hold:

```
define union-subset-theorem :=
  (forall A B C . A subset B | A subset C ==> A subset B \/ C)
```

Cartesian products

We develop the theory of cartesian products, which will become the foundation for a theory of relations and functions.
The core cartesian-product operation will be a polymorphic binary function `X` that

- accepts as inputs two sets of sorts S and T and
- produces an output set of sort `(Pair` S T`)`, i.e., a set of ordered pairs of elements from S and T.

We give a constructive definition of this operation in terms of a simpler operation, `paired-with`, that

- takes a single element x of sort S and a set A of sort T and
- produces a set of sort `(Pair` S T`)`, by prepending x in front of every element of A.

Cartesian products

Thus, for instance, applying `paired-with` to 3 and the set $\{2, 8\}$ will produce the set of pairs $\{(3, 2), (3, 8)\}$ (using conventional notation for sets and ordered pairs).

```
declare paired-with: (S, T) [S (Set T)] -> (Set (Pair S T))
                            [130 [id alist->set]]

assert* paired-with-def :=
  [(_ paired-with null = null)
   (x paired-with h ++ t = x @ h ++ (x paired-with t))]

> (eval 3 paired-with [2 8])

List: [(pair 3 2) (pair 3 8)]
```

Cartesian products

The set-theoretic characterization of `paired-with` from its constructive definition, by induction with basis case:

```
conclude paired-with-characterization :=
   (forall B x y a . x @ y in a paired-with B <==> x = a & y in B)
  by-induction paired-with-characterization {
    null => pick-any x y a
                (!chain [(x @ y in a paired-with null)
                    <==> (x @ y in null)              [paired-with-def]
                    <==> false                        [NC]
                    <==> (x = a & false)              [prop-taut]
                    <==> (x = a & y in null)          [NC]])
  | (B as (insert h t)) =>
      ...
  }

define PWC := paired-with-characterization
```

Cartesian products

and inductive case:

```
  | (B as (insert h t)) =>
      pick-any x y a
        let {IH := (forall x y a .
                      x @ y in a paired-with t <==> x = a & y in t)}
          (!chain
            [(x @ y in a paired-with h ++ t)
          <==> (x @ y in a @ h ++ (a paired-with t))     [paired-with-def]
          <==> (x @ y = a @ h | x @ y in a paired-with t) [in-def]
          <==> (x = a & y = h | x @ y in a paired-with t) [pair-axioms]
          <==> (x = a & y = h | x = a & y in t)           [IH]
          <==> (x = a & (y = h | y in t))                 [prop-taut]
          <==> (x = a & y in B)                           [in-def]])
```

Cartesian products

A useful lemma:

```
define paired-with-lemma-1 :=
  (forall A  x . x paired-with A = null ==> A = null)
```

We can now define cartesian products constructively in terms of `paired-with`:

```
declare product: (S, T) [(Set S) (Set T)] -> (Set (Pair S T))
                                                  [150 [alist->set alist->set]]
define X := product

assert* product-def :=
  [(null X _ = null)
   (h ++ t X A = h paired-with A \/ t X A)]

> (eval [1 2] X ['a 'b])

List: [(pair 1 'a) (pair 2 'a)  (pair 1 'b) (pair 2 'b)]
```

Cartesian products

The classical set-theoretic characterization of this operation can be derived from the preceding definition by induction.

```
define cartesian-product-characterization :=
  (forall A B a b . a @ b in A X B <==> a in A & b in B)

define CPC := cartesian-product-characterization
```

An alternative characterization is occasionally useful:

```
define cartesian-product-characterization-2 :=
  (forall x A B . x in A X B <==> exists a b . x = a @ b & a in A & b in B)

define CPC-2 := cartesian-product-characterization-2
```

Cartesian products

We define a number of theorems:

```
define [product-theorem-1 ... product-theorem-7] := [

  (forall A B . A X B = null ==> A = null | B = null)

  (forall A B . non-empty A & non-empty B ==> A X B = B X A <==> A = B)

  (forall A B C . non-empty A & A X B subset A X C ==> B subset C)

  (forall A B C . B subset C ==> A X B subset A X C)

  (forall A B C . A X (B /\ C) = A X B /\ A X C)

  (forall A B C . A X (B \/ C) = A X B \/ A X C)

  (forall A B C . A X (B \ C) = A X B \ A X C)
]
```

Cartesian products

We show the derivation of a couple of the theorems:

```
conclude product-theorem-1 :=
  (forall A B . A X B = null ==> A = null | B = null)
datatype-cases product-theorem-1 {
  null =>
     pick-any B
       (!chain [(null X B = null)
             ==> (null = null)                        [product-def]
             ==> (null = null | B = null)             [prop-taut]])
 | (A as (insert h t)) =>
     pick-any B
       (!chain
         [(h ++ t X B = null)
       ==> (h paired-with B \/ t X B = null)          [product-def]
       ==> (h paired-with B = null & t X B = null) [union-null-theorem]
       ==> (B = null)                               [paired-with-lemma-1]
       ==> (h ++ t = null | B = null)               [prop-taut]])
}
```

Cartesian products

```
conclude product-theorem-5 :=
  (forall A B C . A X (B /\ C) = A X B /\ A X C)
pick-any A B C
  (!set-identity-intro-direct
    datatype-cases
      (forall p . p in A X (B /\ C) <==> p in A X B /\ A X C)
      {(pair x y) =>
        (!chain [(x @ y in A X (B /\ C))
            <==> (x in A & y in B /\ C)                    [CPC]
            <==> (x in A & y in B & y in C)                [IC]
            <==> ((x in A & y in B) & (x in A & y in C))   [prop-taut]
            <==> (x @ y in A X B & x @ y in A X C)         [CPC]
            <==> (x @ y in A X B /\ A X C)                 [IC]])})
```

Relations

We will understand *a binary relation from A to B* precisely as *a set of ordered pairs (a, b) with $a \in A$ and $b \in B$*. This is the usual mathematical definition of relations.

Thus, in what follows, instead of "binary relations" we will be talking about sets of ordered pairs.

We introduce a few convenient variables for binary relations:

```
define [R R1 R2 R3 R4] :=
       [?R:(Set (Pair 'T14 'T15)) ?R1:(Set (Pair 'T16 'T17)) ...]
```

Relations

Let us now begin by defining the *domain* and *range* of a given relation. The domain of a relation R is the set of all left elements of all the ordered pairs in R; the range of R is the set of all right elements of those pairs. In conventional mathematical notation:

$$dom(R) = \{a \mid \exists\, b \,.\, (a, b) \in R\} \tag{4}$$

and

$$range(R) = \{b \mid \exists\, a \,.\, (a, b) \in R\}. \tag{5}$$

Relations

We give constructive definitions of both as follows:

```
declare dom: (S, T) [(Set (Pair S T))] -> (Set S) [150 [alist->set]]

assert* dom-def :=
  [(dom null = null)
   (dom x @ _ ++ t = x ++ dom t)]

> (eval dom [('a @ 1) ('b @ 2) ('c @ 98)])
List: ['a 'b 'c]

declare range: (S, T) [(Set (Pair S T))] -> (Set T) [150 [alist->set]]

assert* range-def :=
  [(range null = null)
   (range _ @ y ++ t = y ++ range t)]

> (eval range [('a @ 1) ('b @ 2) ('c @ 98)])
List: [1 2 98]
```

Relations

From these definitions, the conventional characterizations (4) and (5) can be derived by induction. These characterizations are defined below along with a few simple but useful lemmas relating the membership relation to dom and range:

```
conclude in-dom-lemma-1 :=
  (forall R a x y . a = x ==> a in dom x @ y ++ R)
pick-any R a x y
  (!chain [(a = x) ==> (a in x ++ dom R)      [in-def]
                   ==> (a in dom x @ y ++ R)  [dom-def]])

define in-range-lemma-1 :=
  (forall R a x y . a = y ==> a in range x @ y ++ R)
```

Relations

```
conclude in-dom-lemma-2 :=
  (forall R x a b . x in dom R ==> x in dom a @ b ++ R)
pick-any R x a b
  (!chain   [(x in dom a @ b ++ R)
         <== (x in a ++ dom R)     [dom-def]
         <== (x in dom R)          [in-def]])

define in-range-lemma-2 :=
  (forall R y a b . y in range R ==> y in range a @ b ++ R)

define dom-characterization :=
  (forall R x . x in dom R <==> exists y . x @ y in R)

define range-characterization :=
  (forall R y . y in range R <==> exists x . x @ y in R)

define [DOMC RANC] := [dom-characterization range-characterization]
```

Relations

We now define or derive a number of results about domains and ranges.

We first note that both operations distribute over unions:

```
conclude dom-theorem-1 :=
  (forall R1 R2 . dom (R1 \/ R2) = dom R1 \/ dom R2)
pick-any R1 R2
  (!set-identity-intro-direct
    pick-any x
      (!chain
        [(x in dom (R1 \/ R2))
    <==> (exists y . x @ y in R1 \/ R2)                          [DOMC]
    <==> (exists y . x @ y in R1 | x @ y in R2)                  [UC]
    <==> ((exists y . x @ y in R1) | (exists y . x @ y in R2)) [quant-dist]
    <==> (x in dom R1 | x in dom R2)                             [DOMC]
    <==> (x in dom R1 \/ dom R2)                                 [UC]]))

define range-theorem-1 :=
  (forall R1 R2 . range (R1 \/ R2) = range R1 \/ range R2)
```

Relations

We may conjecture that the two operations distribute over intersections:

```
define dom-intersection-conjecture :=
  (forall R1 R2 . dom (R1 /\ R2) = dom R1 /\ dom R2)

> (falsify dom-intersection-conjecture 10)
List: ['success
|{
?R1:(Set (Pair Int Int)) := (insert (pair 1 1)
                                    null:(Set (Pair Int Int)))
?R2:(Set (Pair Int Int)) := (insert (pair 1 2)
                                    null:(Set (Pair Int Int)))
}|]
```

However, the counterexample shows that the conjecture does not hold when $R_1 = \{(1,1)\}$ and $R_2 = \{(1,2)\}$. In that case $dom(R_1 \cap R_2) = dom(\emptyset) = \emptyset$, while $dom(R_1) \cap dom(R_2) = \{1\} \cap \{1\} = \{1\}$, so the two sides are unequal.

Relations

Weaker versions of the conjecture do hold for both operations:

```
conclude dom-theorem-2 :=
  (forall R1 R2 . dom (R1 /\ R2) subset dom R1 /\ dom R2)
pick-any R1 R2
  (!subset-intro
    pick-any x
      (!chain
        [(x in dom (R1 /\ R2))
     ==> (exists y . x @ y in R1 /\ R2)                          [DOMC]
     ==> (exists y . x @ y in R1 & x @ y in R2)                  [IC]
     ==> ((exists y . x @ y in R1) & (exists y . x @ y in R2)) [quant-dist]
     ==> (x in dom R1 & x in dom R2)                             [DOMC]
     ==> (x in dom R1 /\ dom R2)                                 [IC]]))

define range-theorem-2 :=
  (forall R1 R2 . range (R1 /\ R2) subset range R1 /\ range R2)
```

Relations

Weaker versions of distribution of both operations over set difference also hold:

```
define dom-theorem-3 :=
  (forall R1 R2 . dom R1 \ dom R2 subset dom (R1 \ R2))

define range-theorem-3 :=
  (forall R1 R2 . range R1 \ range R2 subset range (R1 \ R2))
```

Relations

Next, we introduce the *converse* of a binary relation R. This is simply the relation that holds between a and b iff R holds between b and a. We introduce the double-minus sign `--` as an alias for this operation:

```
declare conv: (S, T) [(Set (Pair S T))] -> (Set (Pair T S))
                                                  [210 [alist->set]]
define -- := conv

assert* conv-def :=
  [(-- null = null)
   (-- x @ y ++ t = y @ x ++ -- t)]

> (eval -- [(1 @ 'a) (2 @ 'b) (3 @ 'c)])

List: [(pair 'a 1) (pair 'b 2) (pair 'c 3)]
```

Relations

The characterization theorem for converse is derived by induction:

```
conclude converse-characterization :=
  (forall R x y . x @ y in -- R <==> y @ x in R)
by-induction converse-characterization {
  null => pick-any x y
          (!chain [(x @ y in -- null)
                   <==> (x @ y in null)     [conv-def]
                   <==> false               [NC]
                   <==> (y @ x in null)     [NC]])
| (R as (insert (pair a b) t)) =>
    let {IH := (forall x y . x @ y in -- t <==> y @ x in t)}
      pick-any x y
        (!chain [(x @ y in -- R)
                 <==> (x @ y in b @ a ++ -- t)          [conv-def]
                 <==> (x @ y = b @ a | x @ y in -- t)   [in-def]
                 <==> (y @ x = a @ b | x @ y in -- t)   [pair-theorem-2]
                 <==> (y @ x = a @ b | y @ x in t)      [IH]
                 <==> (y @ x in R)                      [in-def]])
}
```

Relations

A number of useful results can be derived:

```
define converse-theorem-1 := (forall R . -- -- R = R)

conclude converse-theorem-2 :=
  (forall R1 R2 . -- (R1 /\ R2) = -- R1 /\ -- R2)
pick-any R1 R2
  (!set-identity-intro-direct
    datatype-cases
      (forall p . p in -- (R1 /\ R2) <==> p in -- R1 /\ -- R2)
        {(pair x y) =>
          (!chain [(x @ y in -- (R1 /\ R2))
              <==> (y @ x in R1 /\ R2)              [CC]
              <==> (y @ x in R1 & y @ x in R2)      [IC]
              <==> (x @ y in -- R1 & x @ y in -- R2) [CC]
              <==> (x @ y in -- R1 /\ -- R2)        [IC]])})

define converse-theorem-3 := (forall R1 R2 . -- (R1 \/ R2) = -- R1 \/ -- R2)
define converse-theorem-4 := (forall R1 R2 . -- (R1 \ R2) = -- R1 \ -- R2)
```

Relation composition

The composition of a relation R_1 from S_1 to S_2 with a relation R_2 from S_2 to S_3, which we will denote by $(R_1$ `o` $R_2)$, is the set of all and only those ordered pairs $(x$ `@` $z)$ for which there is a y such that $(x$ `@` $y)$ is in R_1 and $(y$ `@` $z)$ is in R_2.

Roughly speaking, the composition relates two elements x and z iff there is some "intermediate" node y between x and z.

For example, after we define composition we will have:

```
> let {R1 := [('nyc @ 'boston) ('austin @ 'dc)];
       R2 := [('boston @ 'montreal) ('dc @ 'chicago)
              ('chicago @ 'seattle)]}
    (eval R1 o R2)

List: [(pair 'nyc 'montreal) (pair 'austin 'chicago)]
```

Relation composition

We give a constructive definition of composition in two stages. First we define a simpler operation, `composed-with`, that can only compose a given ordered pair $(a$ `@` $b)$ with a binary relation R. The result is another relation that contains all and only those pairs $(a$ `@` $c)$ such that $(b$ `@` $c)$ is in R:

```
declare composed-with: (S1, S2, S3)
   [(Pair S1 S2) (Set (Pair S2 S3))] -> (Set (Pair S1 S3))
   [200 [id alist->set]]

assert* composed-with-def :=
  [(_ composed-with null = null)
   (x @ y composed-with z @ w ++ t =
    x @ w ++ (x @ y composed-with t) <== y = z)

   (x @ y composed-with z @ w ++ t =
    x @ y composed-with t <== y =/= z)]
```

Relation composition

An example and a characterization of `composed-with` follow:

```
> (eval 1 @ 2 composed-with [(2 @ 5) (7 @ 8) (2 @ 3)])

List: [(pair 1 5) (pair 1 3)]

define composed-with-characterization :=
  (forall R x y z w .
    w @ z in x @ y composed-with R <==> w = x & y @ z in R)
```

Relation composition

We can now define the main composition operator, a sample evaluation, and a characterization theorem (abbreviated as OC) as follows:

```
declare o: (S1, S2, S3) [(Set (Pair S1 S2)) (Set (Pair S2 S3))]
                      -> (Set (Pair S1 S3)) [200 [alist->set alist->set]]

assert* o-def :=
 [(null o _ = null)
  (x @ y ++ t o R = x @ y composed-with R \/ t o R)]

> (eval [('nyc @ 'boston) ('houston @ 'dallas) ('austin @ 'dc)] o
        [('boston @ 'montreal) ('dallas @ 'chicago) ('dc @ 'nyc)] o
        [('chicago @ 'seattle) ('montreal @ 'london)])

List: [(pair 'nyc 'london) (pair 'houston 'seattle)]

define o-characterization :=
  (forall R1 R2 x z . x @ z in R1 o R2 <==>
                      exists y . x @ y in R1 & y @ z in R2)
```

Relation composition

Various useful results about composition can now be derived:

```
define [compose-theorem-1 ... compose-theorem-6
        composition-associativity] := [

  (forall R1 R2 . dom R1 o R2 subset dom R1)

  (forall R1 R2 R3 R4 . R1 subset R2 & R3 subset R4
                        ==> R1 o R3 subset R2 o R4)

  (forall R1 R2 R3  . R1 o (R2 \/ R3) = R1 o R2 \/ R1 o R3)

  (forall R1 R2 R3  . R1 o (R2 /\ R3) subset R1 o R2 /\ R1 o R3)

  (forall R1 R2 R3 . R1 o R2 \ R1 o R3 subset R1 o (R2 \ R3))

  (forall R1 R2 . -- (R1 o R2) = -- R2 o -- R1)

  (forall R1 R2 R3 . R1 o R2 o R3 = (R1 o R2) o R3)
]
```

Relation composition

A sample proof:

```
conclude compose-theorem-6 :=
  (forall R1 R2 . -- (R1 o R2) = -- R2 o -- R1)
pick-any R1 R2
  (!set-identity-intro-direct
    datatype-cases
      (forall p . p in -- (R1 o R2) <==> p in -- R2 o -- R1)
      {(pair x y) =>
        (!chain [(x @ y in -- (R1 o R2))
             <==> (y @ x in R1 o R2)                          [CC]
             <==> (exists z . y @ z in R1 & z @ x in R2)      [OC]
             <==> (exists z . z @ y in -- R1 & x @ z in -- R2) [CC]
             <==> (exists z . x @ z in -- R2 & z @ y in -- R1) [prop-taut]
             <==> (x @ y in -- R2 o -- R1)                    [OC]])})
```

Relation restriction

Restricting a relation R to a set of elements A yields a relation consisting of all and only those pairs $(x \texttt{ @ } y)$ in R such that $x \in A$. We again define this relation incrementally, starting with an operation that restricts a relation to a single element:

```
declare restrict1: (S, T) [(Set (Pair S T)) S] -> (Set (Pair S T))
                                                    [200 [alist->set id]]

define ^1 := restrict1

assert* restrict1-def :=
  [(null ^1 _ = null)
   (x @ y ++ t ^1 z = x @ y ++ (t ^1 z) <== x = z)
   (x @ y ++ t ^1 z = t ^1 z           <== x =/= z)]

> (eval [(1 @ 'a) (2 @ 'b) (1 @ 'c)] ^1 1)

List: [(pair 1 'a) (pair 1 'c)]
```

Relation restriction

A characterization theorem can be derived by structural induction.

```
define restrict1-characterization :=
 (forall R x y a . x @ y in R ^1 a <==> x @ y in R & x = a)
```

We now define the regular restriction operator as follows:

```
declare restrict: (S, T) [(Set (Pair S T)) (Set S)] ->
                          (Set (Pair S T)) [200 [alist->set alist->set]]

define ^ := restrict

assert* restrict-def :=
[(R ^ null = null)
 (R ^ h ++ t = R ^1 h \/ R ^ t)]

> (eval [(1 @ 'a) (2 @ 'b) (3 @ 'c) (1 @ 'f)] ^ [1 2])

List: [(pair 1 'a) (pair 1 'f) (pair 2 'b)]
```

Relation restriction

A characterization theorem and some useful results follow:

```
define restrict-characterization :=
  (forall A R x y . x @ y in R restrict A <==> x @ y in R & x in A)

define RSC := restrict-characterization

define [restriction-theorem-1 ... restriction-theorem-5] := [

  (forall R A B . A subset B ==> R ^ A subset R ^ B)

  (forall R A B . R ^ (A /\ B) = R ^ A /\ R ^ B)

  (forall R A B . R ^ (A \/ B) = R ^ A \/ R ^ B)

  (forall R A B . R ^ (A \ B) = R ^ A \ R ^ B)

  (forall R1 R2 A . (R1 o R2) ^ A = (R1 ^ A) o R2)
]
```

Relation restriction

A sample proof:

```
conclude restriction-theorem-3 :=
  (forall R A B . R ^ (A \/ B) = R ^ A \/ R ^ B)
pick-any R A B
  (!set-identity-intro-direct
    datatype-cases
      (forall p . p in R ^ (A \/ B) <==> p in R ^ A \/ R ^ B)
      {(pair x y) =>
        (!chain [(x @ y in R ^ (A \/ B))
            <==> (x @ y in R & x in A \/ B)                    [RSC]
            <==> (x @ y in R & (x in A | x in B))              [UC]
            <==> ((x @ y in R & x in A) | (x @ y in R & x in B)) [prop-taut]
            <==> (x @ y in R ^ A | x @ y in R ^ B)             [RSC]
            <==> (x @ y in R ^ A \/ R ^ B)                    [UC]])})
```

Relation image

The *image* of a relation R on a set A is the set of all y such that $(x \mathbin{@} y) \in R$ for some $x \in A$. This is the same as the range of the restriction of R on A:

```
declare image: (S, T) [(Set (Pair S T)) (Set S)] -> (Set T)
                      [** 200 [alist->set alist->set]]

assert* image-def :=  [(R ** A = range R ^ A)]

> (eval [(1 @ 'a) (2 @ 'b) (3 @ 'c)] ** [1 3])

List: ['a 'c]
```

We derive a characterization theorem:

```
define image-characterization :=
  (forall R A y . y in R ** A <==> exists x . x @ y in R & x in A)

define IMGC := image-characterization
```

Relation image theorems

Some useful results:

```
define [image-theorem-1 ... image-theorem-7] := [

  (forall R A B . R ** (A \/ B) = R ** A \/ R ** B)

  (forall R A B . R ** (A /\ B) subset R ** A /\ R ** B)

  (forall R A B . R ** A \ R ** B subset R ** (A \ B))

  (forall R A B . A subset B ==> R ** A subset R ** B)

  (forall R A . R ** A = null <==> dom R /\ A = null)

  (forall R A . dom R /\ A subset -- R ** R ** A)

  (forall R A B . (R ** A) /\ B subset R ** (A /\ -- R ** B))
]
```

Set cardinality

We define the size or *cardinality* of a set as follows:

```
declare card: (S) [(Set S)] -> N [[alist->set]]

define S := N.S  # Natural-number successor

assert* card-def :=
  [(card null = zero)
   (card h ++ t = card t   <== h in t)
   (card h ++ t = S card t <== ~ h in t)]

transform-output eval [nat->int]

> (eval card [1 2 3] \/ [4 7 8])

Term: 6
```

Set cardinality

Some results involving cardinality are proven in the Set module:

```
define card-theorem-1 := (forall x . card singleton x = S zero)

conclude card-theorem-2 :=
  (forall A x . ~ x in A ==> card A < card x ++ A)
pick-any A x
  assume hyp := (~ x in A)
     (!chain-> [true ==> (card A < S card A)    [N.Less.<S]
                     ==> (card A < card x ++ A) [card-def]])

define minus-card-theorem :=
  (forall A x . x in A ==> card A = S card A - x)

define subset-card-theorem :=
  (forall A B . A subset B ==> card A <= card B)

define proper-subset-card-theorem :=
  (forall A B . A proper-subset B ==> card A < card B)
```

Set cardinality

```
conclude intersection-card-theorem-1 :=
  (forall A B . card A /\ B <= card A)
pick-any A B
  (!chain-> [true ==> (A /\ B subset A) [intersection-subset-theorem]
                  ==> (card A /\ B <= card A) [subset-card-theorem]])

define intersection-card-theorem-2 :=
  (forall A B . card A /\ B <= card B)

define intersection-card-theorem-3 :=
  (forall A B x . ~ x in A & x in B ==> card (x ++ A) /\ B = S card A /\ B)

define union-card :=
  (forall A B . card A \/ B = ((card A) + (card B)) - card A /\ B)

define diff-card-lemma :=
  (forall A B . card A = (card A \ B) + (card A /\ B))
```

Set cardinality

```
conclude diff-card-theorem :=
  (forall A B . card A \ B =  (card A) - card A /\ B)
pick-any A B
  (!chain->
    [true
 ==> (card A = (card A \ B) + card A /\ B) [diff-card-lemma]
 ==> ((card A \ B) + card A /\ B = card A) [sym]
 ==> (card A \ B = (card A) - card A /\ B) [N.Minus.Plus-Minus-properties]])
```

Powersets

The powerset of a set A is the set of all subsets of A. We introduce this operation as a function symbol with the following signature:

```
declare powerset: (S) [(Set S)] -> (Set (Set S)) [[alist->set]]
```

We will define this operation constructively in two stages.
First we introduce a binary operation `in-all` that takes an arbitrary element x of sort S and an arbitrary set A of sets of sort S, and produces the set of sets obtained from A by inserting x in every set in A:

```
declare in-all: (S) [S (Set (Set S))] -> (Set (Set S)) [[id alist->set]]

assert* in-all-def :=
  [(_ in-all null = null)
   (x in-all A ++ t = (x ++ A) ++ (x in-all t))]

> (eval 7 in-all [[1 2] [] [4 5 6]])
List: [[7 1 2] [7] [7 4 5 6]]
```

Powersets

The following is a characterization theorem for `in-all`:

```
define in-all-characterization :=
  (forall U s x . s in x in-all U <==> exists B . B in U & s = x ++ B)
```

We can now define `powerset` recursively in terms of this operation:

```
assert* powerset-def :=
  [(powerset null = singleton null)
   (powerset x ++ t = (powerset t) \/ (x in-all powerset t))]

> (eval powerset [1 2 3])

List: [[] [3] [2] [2 3] [1] [1 3] [1 2] [1 2 3]]
```

Powersets

The corresponding characterization theorem is this:

```
define powerset-characterization :=
  (forall A B . B in powerset A <==> B subset A)

define POSC := powerset-characterization
```

A number of useful theorems involving the powerset operator are derived:

```
conclude ps-theorem-1 := (forall A . null in powerset A)
  pick-any A
   (!chain-> [true ==> (null subset A)      [subset-def]
                   ==> (null in powerset A) [POSC]])

conclude ps-theorem-2 := (forall A . A in powerset A)
  pick-any A
    (!chain-> [true ==> (A subset A)      [subset-reflexivity]
                    ==> (A in powerset A) [POSC]])
```

Powersets

```
conclude ps-theorem-3 :=
  (forall A B . A subset B <==> powerset A subset powerset B)
pick-any A B
  (!equiv assume (A subset B)
          (!subset-intro
            pick-any C
              (!chain [(C in powerset A)
                       ==> (C subset A)              [POSC]
                       ==> (C subset B)              [subset-transitivity]
                       ==> (C in powerset B)         [POSC]]))
          assume (powerset A subset powerset B)
            (!chain-> [true ==> (A in powerset A)  [ps-theorem-2]
                            ==> (A in powerset B)  [SC]
                            ==> (A subset B)       [POSC]]))
```

Powersets

```
conclude ps-theorem-4 :=
  (forall A B . powerset A /\ B = (powerset A) /\ (powerset B))
pick-any A B
  (!set-identity-intro-direct
    pick-any C
      (!chain
        [(C in powerset A /\ B)
    <==> (C subset A /\ B)                    [POSC]
    <==> (C subset A & C subset B)        [intersection-subset-theorem']
    <==> (C in powerset A & C in powerset B) [POSC]
    <==> (C in (powerset A) /\ (powerset B)) [IC]]))
```

Powersets

Remarkably, the union analogue of `ps-theorem-4` does not hold:

```
define p := (forall A B . powerset A \/ B = (powerset A) \/ (powerset B))

> (falsify p 10)
List: ['success
|{
?A := (insert 1 null)

?B := (insert 2 null)

}|]

> (eval powerset [1] \/ [2])
List: [[] [2] [1] [1 2]]

> (eval (powerset [1]) \/ (powerset [2]))
List: [[] [1] [2]]
```

Powersets

But a weaker version of the result does hold:

```
define ps-theorem-5 :=
  (forall A B . (powerset A) \/ (powerset B) subset powerset A \/ B)
```