I. FORMAL SPECIFICATION LANGUAGES

A. DESCRIBING THE WORLD

Software systems are typically embedded in environments that can be arbitrarily strange and complex. In order to make a proposed software system work properly, it is necessary at a minimum to determine which aspects of its environment are relevant to its operation. A formal model of the environment can then be constructed and combined with a formal model of the proposed software to determine whether the software would interact with its environment to produce desirable behavior.

Constructing models of various aspects of the world is a sophisticated activity that long predates the advent of computers. While modeling computer behavior adds some new wrinkles to the general world modeling problem, it is essential to try to make a software modeling framework as compatible as possible with more established modeling traditions. Otherwise we create huge learning costs and tremendous communication barriers.

Traditional modeling is generally carried out in the language of mathematics because the business of mathematics is to elucidate 'absolute' knowledge that can be used repeatedly over a vast spectrum of modeling problems. In this traditional approach, practitioners collect 'empirical' (experience based) knowledge and use it to construct a plausible, basic mathematical model of the situation at hand. This basic model then entails additional mathematical consequences, which might, for example, take the form 'well assuming you've postulated a sound model of the world here, if you do X, then Y should happen.' These mathematical modeling predictions may be used as a basis for collecting additional empirical knowledge about the situation being modeled [think scientific method here], or they may be used to build artifacts [think engineering here] or to create vast human enterprises – [think business organization here].

As you surely know, this whole modeling process is fraught with potential problems. You rarely know whether you're modeling all the important aspects of a situation or whether you've got adequate empirical data, etc. Moreover, a simple, approximately correct model may be pragmatically preferable to a much more exacting model that takes years to work out and understand.

However, there's an important point to note here about the problem of describing software. Almost all the troublesome empirical modeling issues lie outside the software itself. This may not be much comfort to a system analyst trying to perform the requirements analysis for software that's to be embedded in some weird environment, but it does make life easier in the rest of the software development process. We only need to figure out how to describe our software components within the 'absolute' world of mathematics.

B. THE MATHEMATICS WORLD

At first encounter, most of the phenomena that interest us in the world overwhelm us with their apparent complexity. So the three watchwords of understanding are simplify, simplify, and simplify. For a long time mathematics seemed to be about a variety of disparate domains. There were the natural numbers and integers used for counting and accounting things, the planes, lines, spheres, etc. used for geometric descriptions, the real numbers used for describing measurements, etc. Eventually mathematicians discovered that these seemingly distinct domains, as well as all of the more recently introduced domains such as the trees and graphs used in computing, could be unified in a single domain – the domain of sets.

For most of the traditional disciplines that use mathematics, this change in the intellectual landscape of mathematics is largely irrelevant, since most of their modeling problems can be solved using the well understood mathematical domain of real numbers with a few extensions for vectors, functions, etc. While these extensions are based in set theory, they don't really require a serious grounding in the theory of sets. The situation is different for the computing discipline, since computing involves working in many different domains, most of which could only be shoehorned into the real numbers with great difficulty, if at all.

1. BASIC SET THEORY

Because of the unifying role of sets, we want to build standard machinery for describing them into our basic specification language. If the foundational thesis about the adequacy of set theory for describing all mathematical domains is correct, we should then have a framework for describing all modeling domains that might arise in the software specification process.

If sets are going to provide models for everything that we might ever want to talk about, then establishing a theory of sets presents something of a paradox. On the one hand, the description of the nature of sets must be simple enough that we can believe that it really identifies what we want. On the other hand, that same description should imply the existence of any arbitrarily strange set that we might ever conceive of. This isn't quite attainable, so we'll have to settle for getting at least the sets that people have needed so far, with possibility of augmenting the description of the nature of sets if we discover that something we need is missing.

A second conundrum arises from the problem of what domain to use to describe set theory. If the foundational thesis is correct, then set theory, being a mathematical theory, should be described using set theory! Having the collection of all the sets you'd ever want to talk about be a member of itself turns out to just as bad an idea as you'd expect it to be. The way out of this problem is to make a distinction between the formal set theory that we're trying to describe and an informal (bigger) set theory in our metalanguage. There's just no way to explain anything, starting from nothing at all.

The general approach to formulating a theory for any domain is to try to identify the basic operators and collections of objects that seem to be involved, to look for defining properties of the operations, and then to simplify the formulation as much as possible. For an adequate set theory, we know we're eventually going to need operators like $\bigcup$, $\bigcap$, $\subseteq$, $\in$, $\wp$, $\rightarrow$, etc., but the simplification process leads us to the conclusion that we can start out with just the 'is an element of' operator $\in$ and then use it to define all the rest.

At first blush there would appear to be two distinct sorts of object collections involved. The first would be a collection of individual entities usually called atoms, which we could symbolize as $\mathcal{A}$. They're all the apples, oranges, motorists, asteroids, etc. of which we want wish to form sets, and they're distinguished by not having members the way sets do. The second collection would then be all the possible sets formable from the atoms, which we could symbolize as $\mathcal{S}et(\mathcal{A})$. Here the simplification process leads us to the initially surprising conclusion that we can avoid the problems associated with saying what atoms are and where they come from by simply eliminating them from our set theory all together. The idea is that, as long as our remaining collection of sets contains sets that look like any possible collection of atoms that we might ever need (at least as far as anyone can tell using just the set operators), we can use these internal sets as proxies for the atom collections. The problem of connecting up the atoms with their set proxies is then left as part of the modeling activity. The business of discovering absolute mathematical knowledge can then proceed in an austere world with only pure sets and no strange atoms.

The obvious question is what, if anything, is left after you eliminate the apples and oranges that we employed when building our initial intuitions about sets. Well at least we still have the empty set $\varnothing$. From it we could get the one element set $\{\varnothing\}$, and then $\{\{\varnothing\}\}$, and $\{\varnothing, \{\varnothing\}\}$, and $\cdots$
We'll have to identify several structural properties in order to guarantee that this meager beginning will lead to enough sets to model everything we'll ever need, but at least it's not silly to start talking about a collection of pure sets $\mathcal{S}et$.

Our eventual language for developing the standard mathematics that we'll need in programming will be a more complex one with features like a type system to make it easier to express concisely the kinds of things we commonly want to say in mathematics. But in the interest of keeping the underlying machinery as simple as possible to examine for correctness, we'll formulate a basic foundational theory for sets in the traditional predicate calculus style, and then relate our more sophisticated language to this simpler system.

We begin by specifying that, as indicated above, we'll be formulating this foundational theory of sets in terms of just two basic concepts. The first is the collection $\mathcal{S}et$, which we intend to denote all possible sets, from which all the objects to be discuss will come. The second is the two-argument predicate $x \in y$, which we intend to denote that the set x is a member of the set y. Of course, in mathematical discussions, our intentions don't count for much unless they're recorded precisely, and to that end we must record enough of the properties that sets should have to ensure that all the other properties of sets are logical consequences of the properties we've recorded.

One such property is the *extensionality* property, which says that any two sets that contain precisely the same elements are equal. Formally, that's written:

$\forall$ A, $\forall$ B, **if** $\forall$ x, $x \in A$ **iff** $x \in B$, **then** $A = B$;

A second characterizing property is the *empty set* property, which says that the sets include a set with no members:

$\exists E \ni \forall x, \neg\, x \in E$

Since the extensionality property allows us to prove that there's exactly one empty set in $\mathcal{S}et$, we are able to add a name for this set to our language without altering what is true about sets. Naturally we choose the traditional symbol $\varnothing$ as the name for the empty set so that we can start to build back the expressive power which we have been accustomed to having when discussing sets. Formally we write:

Implicit Definition of $\varnothing$ **is** $\forall x, \neg x \in \varnothing$

The rest of the formal development of set theory is a little more complicated because we need to make sure that our universe $\mathcal{Set}$ is vast enough to include a smaller subuniverse $\mathbb{Set}$ that contains all the sets that are used in ordinary world modeling. This is necessary because truly general software involves creating components such as stacks that work for any arbitrary set of entries that might arise in any computing problem. We will also want to specify what behavior we expect from system software such as the compilers for our programming language. The collections of things necessary to describe this situation will turn out to be bigger than any collections of things we allow in our programming language. So if we want our set theory to cover everything, we need the universe our programming language uses, $\mathbb{Set}$, to be just a small chunk of the full collection of sets, $\mathcal{Set}$.

Appendix I presents the complete collection of basic properties we will use to characterize sets, together with the definitions of the usual operators on sets and the theorems that justify the standard manipulation rules we use with set theory formulas (e. g., $(A\cup B)\cap C = (A\cap C)\cup(B\cap C)$).

2. MATHEMATICAL TYPES

In our basic set theory development, the only object type under discussion is sets, so we don't feel the need for a type system. As we develop the various specialized domains we need for ordinary mathematics within this set theory framework, having a type system proves to be a great convenience. This is because, first, types are familiar and natural from informal usage, second, they shorten our mathematical expressions, and third, they provide an easy mechanical check that frequently highlights areas in our mathematical developments where our thinking has gotten jumbled.

Types are in common usage in at least three different activity spheres important to the software development process, and while they serve similar purposes in each area, they are really used in quite distinct ways. Not understanding these distinctions can lead to serious problems with types.

The world modeling activity normally involves using types to record situational distinctions that should be respected in the analytic process. For example, if you find a formula in physics that equates a volume in cubic centimeters to a temperature in degrees Celsius times a length in feet or if you find a formula in a business model that equates an inventory count of tables to the sum of several counts of chairs, then you can be certain that these typing errors indicate some sort of serious modeling error. This is true despite the fact that there's no mathematical typing problem with multiplying two real numbers together and expecting the result to be a real number or with adding several integers and expecting the result to be an integer. Of course, as is usual with all typing systems, the absence of typing errors doesn't ensure that a model is correct, but the presence of one does ensure that it's wrong.

The programming activity certainly involves using types to avoid breaking abstractions. You don't, for example, want to pass as a parameter a real number when an integer is expected or even to pass a chair count represented as short integer when a long integer chair count is expected. In addition, if you were using long integers to represent both chair counts and file lengths, you'd probably still like to be able to make a type distinction, so that silly crossover errors would be caught quickly by your type checker before they became big debugging problems.

For the mathematics development activity, we don't want to use types to make the situational distinctions of world modeling, since we wouldn't want to develop separate theories of temperatures in degrees Celsius and lengths in feet when the single real number theory will work for these and a host of other modeling domains. Neither do we want the types to make the representational distinctions that are important in computing. (I. e., we don't want to have distinct theories of integers represented in binary and decimal notations.) That said, there are still distinctions that are worth capturing in a type system. For example, if F is a set representing a function with domain D and range R, then for $x \in D$, the function application F(x) will be meaningful and always represent some specific value in R. On the other hand, if F were a relation or anything else, then F(x) would be a quite meaningless expression. If, as we do in programming, we insist that all variables have types, then, whenever we see F(x) in a mathematical expression, it will be easy to determine whether or not it's meaningful.

Our approach then will be to add a type specification scheme to our basic predicate calculus system and to insist that all variables in our mathematical language be explicitly introduced by a construct such as a quantifier or a definition, which will be used to assign them a type. The types will just be sets from our underlying set collection, and in fact, sets will find their typical usage in our mathematical language as type designators.

To preview how this will work, suppose that we'd already managed to define the integers to be a set named $\mathbb{Z}$. In terms of $\mathbb{Z}$, we might have defined the + and $\cdot$ functions to be infix expressed functions of type $\mathbb{Z}\boxtimes\mathbb{Z}\to\mathbb{Z}$. We could then express mathematically the distributive property by:

$\forall$ a, b, c: $\mathbb{Z}$, $(a + b)\cdot c = a\cdot c + b\cdot c$

and see that it was at least a well-typed expression. Such an expression could be viewed as just a shorthand for its translate back into our basic set theory:

$\forall$ a, $\forall$ b, $\forall$ c, **if** $a \in \mathbb{Z}$ **and** $b \in \mathbb{Z}$ **and** $c \in \mathbb{Z}$, **then** $(a + b)\cdot c = a\cdot c + b\cdot c$,

but as in this case, our new expressions will generally be more concise and comprehensible.

We can also talk about more complex types such as the function Inv in

$\exists$ Inv: $\mathbb{Z}\to\mathbb{Z} \ni \forall$ a: $\mathbb{Z}$, $a + \mathrm{Inv}(a) = 0$.

When defining the type for a variable, we will typically use some previously defined set such as $\mathbb{Z}$ or some combination of the set formation operators such as the function set former $A\to B$, the power set former $\wp(A)$, and the Cartesian product former $A\times B$ that were defined in our basic set theory. In principle, any set could be used as a type designator, but in practice, it's much more common to use variables whose types are sets such as $\mathbb{Z}$ or $\mathbb{R}$ for which we have available a rich collection of functions and relations.

Our type system will also adhere to a historical mathematical practice that seems to date to the period before pure set theory was recognized as an adequate basis for mathematical notation. The typical user of mathematical notation views entities such as integers or real numbers as something like the atoms discussed earlier that exist outside of set theory. In this view, it doesn't make sense to talk about an expression like $7 \in 5$. In the pure set theory approach, both 7 and 4 will be sets, and $7 \in 5$ will make logical sense and be either true or false. However, in order not to have a bunch of non standard mathematical results such as $7 \in 5$, we'll set up integer theory so that in some models of $\mathbb{Z}$, $7 \in 5$, while in others $\neg\, 7 \in 5$.

Now when it comes to type designators, typical users don't want to see quantification expressions which might make sense in pure set theory such as:

$\forall$ k: 5, $\cdots$

Luckily, it's easy enough to let the mathematical syntax checker pretend that the sets we're using admit the possibility of being impure and of containing atoms as elements. Our definition for the sets $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{R}$, etc. will turn out to make them potentially impure, so that in the syntax checker's view, 5 could potentially be an atom, and thus it is inappropriate to use as a type designator. With this scheme, we are able to remain consistent with traditional mathematical typing practices, while avoiding the philosophical uncertainties introduced by basing everything on an impure set theory.

C. MATHEMATICAL UNITS

When we contemplate using mathematics to specify our software, the advantages in terms of understandability and labor saving of reusing existing mathematics to the maximum practical extent are fairly obvious. It’s also pretty clear that if all known mathematics were somehow gathered together into a single gargantuan math package, it would be totally unusable. The standard solution to such problems of scale is a modular break down into tractable pieces, which contain definitions and theorems that are related to each other and are likely to be used simultaneously by clients.

A Further consideration in such a modular break down is that the bulk of a complex mathematical development consists of proofs of the various theorems presented, while the typical user of this mathematics has little interest in these proofs beyond being assured that they exist and are correct. Accordingly it’s logical to separate a given unit of mathematical development into two modules: a précis summarizing the results that an ordinary user would need and a proof unit containing the detailed proofs that a mathematical specialist can check.

It’s probably easiest to begin to see what these units involve if we lay out a simple example such as a précis providing familiar basic properties of binary relations that can be reused in several other précis and proof units.

```
Precis Basic_Binary_Relation_Properties;

    Def. Is_Reflexive( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x: D, x ρ x );

    Def. Is_Transitive( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x, y, z: D,
                                                        if x ρ y and y ρ z, then x ρ z );

    Def. Is_Symmetric( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x, y: D, if x ρ y, then y ρ x );

    Def. Is_Antisymmetric( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x, y: D,
                                                        if x ρ y and y ρ x, then x = y );

    Def. Is_Asymmetric( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x, y: D, if x ρ y, then ¬ y ρ x );

    Def. Is_Irreflexive( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x: D, ¬ x ρ x );
        Corollary 1: ∀ D: Set, ∀ ρ: D⊠D→𝔹, if Is_Transitive( ρ ), then
                                                  Is_Irreflexive( ρ ) iff Is_Asymmetric( ρ );

    Def. Is_Total( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x, y: D, x ρ y or y ρ x );
        Corollary 1: ∀ D: Set, ∀ ρ: D⊠D→𝔹, if Is_Total( ρ ) then Is_Reflexive( ρ );

    Def. Is_Trichotomous( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x, y: D, x ρ y or x = y or y ρ x );

end Basic_Binary_Relation_Properties;
```

Not much in the way of proofs is needed here but a matching proof unit would look like:
Proofs Obvious_Prfs **for** Basic_Binary_Relation_Properties;

```
  Def. Is_Irreflexive( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x: D, ¬ x ρ x );
     Corollary 1: ∀ D: Set, ∀ ρ: D⊠D→𝔹, if Is_Transitive( ρ ), then
                                                     Is_Irreflexive( ρ ) iff Is_Asymmetric( ρ );
     Proof
     Supposition D: Set and ρ: D⊠D→𝔹 and Is_Transitive( ρ )
        Goal Is_Irreflexive( ρ ) iff Is_Asymmetric( ρ )
        Goal if Is_Irreflexive( ρ ), then Is_Asymmetric( ρ ) and if Is_Asymmetric( ρ ),
                                                                          then Is_Irreflexive( ρ )
        Goal if Is_Irreflexive( ρ ), then Is_Asymmetric( ρ )
        Supposition Is_Irreflexive( ρ )
           Goal Is_Asymmetric( ρ )
           Goal ∀ x, y: D, if x ρ y, then ¬ y ρ x
           Supposition x, y: D and x ρ y
              Goal ¬ y ρ x
           (∗) y ρ x or ¬ y ρ x                      by excluded middle
              Supposition y ρ x
                 Goal ¬ y ρ x
                 x ρ y and y ρ x                     by Supp$^2$ & ‘and’ rule
                 x ρ x                               by Supp$^4$ & def Is_Transitive
                 False                               by Supp$^3$ & def Is_Irreflexive
                 ¬ y ρ x                             by contradiction rule
              deduction if y ρ x, then ¬ y ρ x
              ¬ y ρ x or ¬ y ρ x                     by (∗) & ‘implies or’ rule
              ¬ y ρ x                                by Is_Idempotent( or )
           deduction if x, y: D and x ρ y, then ¬ y ρ x
           ∀ x, y: D, if x ρ y, then ¬ y ρ x         by universal generalization
           Is_Asymmetric( ρ )                        by def Is_Asymmetric
   (∗∗) deduction if Is_Irreflexive( ρ ), then Is_Asymmetric( ρ )
        Goal if Is_Asymmetric( ρ ), then Is_Irreflexive( ρ )
        Supposition Is_Asymmetric( ρ )
           Goal Is_Irreflexive( ρ )
           Goal ∀ x: D, ¬ x ρ x
           Supposition x: D
              Goal ¬ x ρ x
           (∗) x ρ x or ¬ x ρ x                      by excluded middle
              Supposition x ρ x
                 Goal ¬ x ρ x
                 ¬ x ρ x                             by Supp$^3$ & def Is_Asymmetric
              deduction if x ρ x, then ¬ x ρ x
              ¬ x ρ x or ¬ x ρ x                     by (∗) & ‘implies or’ rule
              ¬ x ρ x                                by Is_Idempotent( or )
```

```
            deduction if x: D, then ¬ x ρ x
            ∀ x: D, ¬ x ρ x                              by universal generalization
            Is_Irreflexive( ρ )                          by def Is_Irreflexive
        deduction if Is_Asymmetric( ρ ), then Is_Irreflexive( ρ )
        If Is_Irreflexive( ρ ), then Is_Asymmetric( ρ ) and if Is_Asymmetric( ρ ),
                                                               then Is_Irreflexive( ρ )
                                                         by (**) & ‘and’ rule
        Is_Irreflexive( ρ ) iff Is_Asymmetric( ρ )       by def iff
    deduction if D: Set and ρ: D⊠D→𝔹 and Is_Transitive( ρ ), then
                                                Is_Irreflexive( ρ ) iff Is_Asymmetric( ρ )
    ∀ D: Set, ∀ ρ: D⊠D→𝔹, if Is_Transitive( ρ ), then
                                                Is_Irreflexive( ρ ) iff Is_Asymmetric( ρ )
                                                         by universal generalization
    QED;

  Def. Is_Total( ( □: D: Set ) ρ ( □: D ): 𝔹 ): 𝔹 = ( ∀ x, y: D, x ρ y or y ρ x );
    Corollary 1: ∀ D: Set, ∀ ρ: D⊠D→𝔹, if Is_Total( ρ ) then Is_Reflexive( ρ );
    Proof
    Supposition D: Set and ρ: D⊠D→𝔹 and Is_Total( ρ )
            ⋮
    QED;
end Obvious_Prfs;
```

We can then use the binary relation properties to develop a unit covering the basic notions of orderings.

```
Precis Basic_Ordering_Theory;
    uses Basic_Binary_Relation_Properties;

  Def. Is_Preordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = (  Is_Reflexive( ⊴ ) and
                                                           Is_Transitive( ⊴ )  );

  Def. Is_Partial_Ordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = (  Is_Preordering( ⊴ ) and
                                                           Is_Antisymmetric( ⊴ )  );

  Def. Is_Total_Preordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = (  Is_Transitive( ⊴ ) and
                                                           Is_Total( ⊴ )  );

  Def. Is_Total_Ordering( ( □: D: Set ) ⊴ ( □: D ): 𝔹 ): 𝔹 = (  Is_Total_Preordering( ⊴ ) and
                                                           Is_Antisymmetric( ⊴ )  );

  Def. Is_Strict_Partial_Ordering( ( □: D: Set ) ⊲ ( □: D ): 𝔹 ): 𝔹 = (  Is_Transitive( ⊲ ) and
                                                           Is_Irreflexive( ⊲ )  );
```

Def. Is_Strict_Ordering((□: D: ***Set***) ⊲ (□: D): 𝔹): 𝔹 = (Is_Strict_Partial_Ordering(⊲)
and Is_Trichotomous(⊲));

Def. Is_Dense_Ordering((□: D: ***Set***) ⊲ (□: D): 𝔹): 𝔹 = (Is_Strict_Partial_Ordering(⊲)
∀ x, z: D, if x ⊲ z, then ∃ y: D ∋ x ⊲ y and y ⊲ z);

end Basic_Ordering_Theory;

In a similar fashion, we can also define the basic properties of functions for future reference.
Precis Basic_Function_Properties;

Def. Is_Identity(F: (D: ***Set***)→D): 𝔹 = (∀ x: D, F(x) = x)

Def. Is_Injective(F: (D: ***Set***)→(R: ***Set***)): 𝔹 = (∀ x, y: D, **if** F(x) = F(y), **then** x = y);
Corollary 1: ∀ D, R, S: ***Set***, ∀ F: D→R, ∀ G: R→S, **if** Is_Injective(F) **and**
Is_Injective(G), **then** Is_Injective(G∘F);
Corollary 2: ∀ D, R: ***Set***~{∅}, ∀ F: D→R, Is_Injective(F) **iff** ∃ G: R→D ∋
Is_Identity(G∘F);
Corollary 3: ∀ C, D, R: ***Set***, ∀ F: D→R, **if** Is_Injective(F) **and** C ⊆ D, **then**
Is_Injective(F↾C);

Def. Is_Surjective(F: (D: ***Set***)→(R: ***Set***)): 𝔹 = (∀ y: R, ∃ x: D ∋ F(x) = y);
Corollary 1: ∀ D, R, S: ***Set***, ∀ F: D→R, ∀ G: R→S, **if** Is_Surjective(F) **and**
Is_Surjective(G), **then** Is_Surjective(G∘F);
Corollary 2: ∀ D, R: ***Set***, ∀ F: D→R, Is_Surjective(F) **iff** ∃ G: R→D ∋
Is_Identity(F∘G);

Def. Is_Bijective(F: (D: ***Set***)→(R: ***Set***)): 𝔹 = (Is_Injective(F) **and** Is_Surjective(F));
⋮
end Basic_Function_Properties;

Another collection of useful property names are those used in conjunction with binary operations.
Precis Basic_Binary_Operation_Properties;

Def. Is_Associative((□: D: ***Set***) ⊕ (□: D): D): 𝔹 = (∀ x, y, z: D,
x ⊕ (y ⊕ z) = (x ⊕ y) ⊕ z);

Def. Is_Commutator_for((□: D: ***Set***) ⊕ (□: D): D, c: D): 𝔹 = (∀ y: D, c ⊕ y = y ⊕ c);

Def. Is_Commutative((□: D: ***Set***) ⊕ (□: D): D): 𝔹 = (∀ x: D,
Is_Commutator_for(⊕, x));

Def. Is_Right_Cancelable_for((□: D: *Set*) ⊕ (□: D): D, c: D): 𝔹 = (∀ x, y: D,
if x ⊕ c = y ⊕ c, **then** x = y);

Def. Is_Right_Cancelative((□: D: *Set*) ⊕ (□: D): D): 𝔹 = (∀ z: D,
Is_Right_Cancelable_for(⊕, z));

Def. Is_Right_Identity_for((□: D: *Set*) ⊕ (□: D): D, i: D): 𝔹 = (∀ x: D, x ⊕ i = x);

Def. Is_Left_Identity_for((□: D: *Set*) ⊕ (□: D): D, j: D): 𝔹 = (∀ x: D, j ⊕ x = x);
Corollary 1: ∀ D: *Set*, ∀ ⊕: D⊠D→D, ∀ i, j: D, **if** Is_Right_Identity_for(⊕, i) **and**
Is_Left_Identity_for(⊕, j), **then** i = j;

Def. Is_Identity_for((□: D: *Set*) ⊕ (□: D): D, i: D): 𝔹 = (Is_Right_Identity_for(⊕, i)
and Is_Left_Identity_for(⊕, i));

Def. Is_Right_Invertable_for((□: D: *Set*) ⊕ (□: D): D, x: D): 𝔹 = (∃ y, i: D ∍
Is_Right_Identity_for(⊕, i) **and** x ⊕ y = i);
Corollary 1: ∀ D: *Set*, ∀ ⊕: D⊠D→D, ∀ x: D, **if** Is_Associative(⊕) **and**
Is_Right_Invertable_for(⊕, x), **then** Is_Right_Cancelable_for(⊕, x);

Def. Is_Right_Distributive_over((□: D: *Set*) ⊕ (□: D): D, (□: D)⊗(□: D): D): 𝔹 = (
∀ x, y, z: D, (x ⊕ y)⊗z = (x⊗z) ⊕ (y⊗z));

Def. Is_Left_Distributive_over((□: D: *Set*) ⊕ (□: D): D, (□: D)⊗(□: D): D): 𝔹 = (
∀ x, y, z: D, x⊗(y ⊕ z) = (x⊗y) ⊕ (x⊗z));

Def. Is_Idempotent((□: D: *Set*) ⊕ (□: D): D): 𝔹 = (∀ x: D, x⊗x = x);

Def. Is_Right_Zero_for((□: D: *Set*)⊗(□: D): D, z: D): 𝔹 = (∀ x: D, x⊗z = z);
⋮
end Basic_Binary_Operation_Properties;