Chapter 8:

Getting Rid of Functions and Equality

Taking a bird's-eye view of Chapter 3, in that chapter we have shown, within the context of Propositional Logic, that there exists a simple syntactic procedure to modify sets of "rich formulas" (in that chapter: formulas that use arbitrary operators) to sets of "less-rich formulas" (say, those using only the operators '$\rightarrow$' and '$\sim$') without losing any expressiveness, in the sense that a set of rich formulas has a model if and only if the corresponding set of less-rich formulas has a model (and that chapter, it in fact turned out to be the exact same model). In this chapter, we will follow a conceptually similar (yet technically quite different) path, and see that we can essentially eliminate two ingredients from our predicate-logic expressions with essentially no loss of any expressiveness. Specifically, you will show that we can syntactically replace all function invocations with invocations of appropriate relations, and that we can syntactically replace any equality with the invocation of an essentially equivalent relation. The point with all these replacements will be the same: that we do not lose expressive power in the same sense that if we take a set of formulas that may contain function invocations and equalities ("rich formulas"), and the corresponding set of formulas that results from syntactically replacing all these with corresponding relation invocations ("less-rich formulas"), then while they do not share the same models (as was the case in Chapter 3), models can still be translated back and forth between these sets of formulas, and in particular, the former set of formulas has a model if and only if the latter does, so the question of "does a given set of formulas have a model" can encode the same set of problems whether or not we allow the usage of functions and equalities. This will allow us to use functions and equalities whenever we find it convenient to do so (e.g., in Chapter 10), and to ignore them when we find it inconvenient to use them (in Chapter 12).

1 Getting Rid of Functions

In this section, we will show that we can eliminate all function invocations from predicate-logic formulas without losing expressive power. The idea is that functions $f : \Omega^k \rightarrow \Omega$ can be "encoded" as relations $F \subset \Omega^{k+1}$ where $y = f(x_1, \ldots, x_k) \Leftrightarrow F(y, x_1, \ldots, x_k)$.[1] Of course, such an "encoding" relation F is not arbitrary but has the special property that for every $x_1, \ldots, x_k$ there exists a single (no more and no less) y such that $F(y, x_1, \ldots, x_k)$. So, we will transform any formula ϕ that contains any invocations of function names $f_1, \ldots, f_t$ to an essentially equivalent formula that instead contains invocations of corresponding relation names $F_1, \ldots, F_t$. Specifically, each function invocation will be replaced by an invocation of a corresponding relation that has the same name as the function, except that the first letter is capitalized. For example, an invocation of the function named 'f'

[1] In fact, this is how functions are formally defined in the mathematical field known as **Set Theory**, where all mathematical objects, including functions, are defined as sets.

will be replaced by an invocation of the corresponding relation named 'F', an invocation of the function named 'f7' by an invocation of the corresponding relation named 'F7', and an invocation of the function named 'plus' by an invocation of the corresponding relation named 'Plus'. The file `predicates/functions.py`, in which all of the functions that you are asked to implement in this chapter should be implemented, contains Python functions that perform this translation from function names to relation names and back.

predicates/functions.py

```
def function_name_to_relation_name(function: str) -> str:
    """Converts the given function name to a canonically corresponding relation
    name.

    Parameters:
        function: function name to convert.

    Returns:
        A relation name that is the same as the given function name, except that
        its first letter is capitalized.
    """
    assert is_function(function)
    return function[0].upper() + function[1:]

def relation_name_to_function_name(relation: str) -> str:
    """Converts the given relation name to a canonically corresponding function
    name.

    Parameters:
        relation: relation name to convert.

    Returns:
        A function name `function` such that
        `function_name_to_relation_name(function)` is the given relation name.
    """
    assert is_relation(relation)
    return relation[0].lower() + relation[1:]
```

Definition (Relation Corresponding to Function; Function-Free Analog of Model)**.**

- Given an n-ary function f defined over a universe Ω, we say that a given $n+1$-ary relation F defined over the same universe Ω is the relation that **corresponds to** the function f if for all $y, x_1, \ldots, x_n \in \Omega$ we have that $F(y, x_1, \ldots, x_n)$ is *True* if and only if $f(x_1, \ldots, x_n)$ is y.
- Given a model M that contains functions, we say that a model M' is the **function-free analog** of the model M if M' is identical to M except that each function meaning in M is replaced in M' with the relation meaning that corresponds to it (as just defined), where the latter meaning is assigned to the relation name that is the same as the function name to which the former meaning is assigned, except that the first letter is capitalized.

For example, the following model with a single unary function and a single unary relation…:

- Universe $\Omega = \{0, 1, 2\}$.
- The meaning of a constant called '0', defined as $0 \in \Omega$.

- The meaning of a single unary function called 'inverse', defined as inverse(1) = 0, inverse(2) = 1, and inverse(0) = 2.
- The meaning of a single unary relation 'IsPrimitive', defined as $\{(2)\}$.

...is encoded by its following function-free analog model with a single binary relation that corresponds to the above unary function, and the same single unary relation:

- Universe, as above, $\Omega = \{0, 1, 2\}$.
- The meaning of a constant called '0', defined as above as $0 \in \Omega$.
- The meaning of a single binary relation called 'Inverse', defined as $\big\{(0,1), (1,2), (2,0)\big\}$.
- The meaning of a single unary relation 'IsPrimitive', defined as above as $\{(2)\}$.

In the next two task, you will programmatically transform a given model to its function-free analog model and back.

Task 1. Implement the missing code for the function `replace_functions_with_relations_in_model(model)`, which returns the function-free analog model of the given model that may contain function invocations.

predicates/functions.py

```
def replace_functions_with_relations_in_model(model: Model[T]) -> Model[T]:
    """Converts the given model to a canonically corresponding model without any
    function meanings, replacing each function meaning with a canonically
    corresponding relation meaning.

    Parameters:
        model: model to convert, such that there exist no canonically
            corresponding function name and relation name that both have
            meanings in this model.

    Returns:
        A model obtained from the given model by replacing every function
        meaning of a function name with a relation meaning of the canonically
        corresponding relation name, such that the relation meaning contains
        any tuple (`x1`,...,`xn`)  if and only if `x1` is the output of the
        function meaning for the arguments (`x2`,...,`xn`).
    """
    for function in model.function_meanings:
        assert function_name_to_relation_name(function) not in \
               model.relation_meanings
    # Task 8.1
```

Task 2. Implement the missing code for the function `replace_relations_with_functions_in_model(model, original_functions)`, which returns the model that has meanings for the given function names (and not for any other function names), and whose function-free analog is the given model.

predicates/functions.py

```
def replace_relations_with_functions_in_model(model: Model[T],
                                              original_functions:
                                              AbstractSet[str]) -> \
        Union[Model[T], None]:
```

```
    """Converts the given model with no function meanings to a canonically
    corresponding model with meanings for the given function names, having each
    new function meaning replace a canonically corresponding relation meaning.

    Parameters:
        model: model to convert, that contains no function meanings.
        original_functions: function names for the model to convert to,
            such that no relation name that canonically corresponds to any of
            these function names has a meaning in the given model.

    Returns:
        A model `model` with the given function names such that
        `replace_functions_with_relations_in_mode(model)` is the given model, or
        ``None`` if no such model exists.
    """
    for function in original_functions:
        assert is_function(function)
        assert function not in model.function_meanings
        assert function_name_to_relation_name(function) in \
               model.relation_meanings
    # Task 8.2
```

Hint: You should return `None` unless for every one of the given function names there is a correspondingly named relation meaning in the given model that corresponds to a function, i.e., if this relation meaning $k+1$-ary, then for every $x_1, \ldots, x_k$ in the universe of the given model, there exists a single (no more and no less) y in the universe of the given model such that $(y, x_1 \ldots, x_k)$ is in this relation meaning.

We now move on to the task of converting a formula that may contain function invocations to a function-free analog formula that instead contains invocation of the corresponding relations. Our goal is that the truth value of the former in any model M be the same as that of the latter in the function-free analog model of the same model M.

Definition (Function-Free Analog of Formula)**.** Given a formula ϕ that may contain function invocations, we say that a formula ϕ' that contains no function invocations is a **function-free analog**[2] of the formula ϕ if all of the following hold.

1. ϕ' uses the same relation names as ϕ, and in addition, for each function name in ϕ, the formula ϕ' also uses a relation with the same name except that the first letter is capitalized.

2. ϕ' has the same set of free variables as ϕ.

3. For every model M and for every assignment of elements from the universe of M to the free variables of ϕ, the truth value of ϕ in the model M with the given assignment is the same as the truth value of ϕ' in the model M' that is the function-free analog of M, with the same assignment.

The basic idea is to introduce a new variable, say z, to hold the value of every function invocation '$f(x)$' in the original formula. We will then replace the usage of the value of

[2]While we have defined above *the* function-free analog of a model, we define here *a* function-free analog of a formula. While a formula may indeed have more than one function-free analog, we will for simplicity focus below on a specific function-free analog for any given formula.

the function invocation by the new variable (i.e., replace '$f(x)$' with 'z'), and add an assumption that indeed 'z=$f(x)$', however this assumption will be specified using the corresponding relation, i.e., it will be specified as '$F(z,x)$'. For example, a function-free analog of the formula 'IsPrimitive(inverse(x))' (that uses the function and relation names as in the first model in the above example) is '∀z[(Inverse(z,x)→IsPrimitive(z))]' (which uses the two relation names as in the second model in the above example).

With some more care, we can also handle formulas that contain function invocations whose arguments are themselves function invocations, such as 'R(f(g(x))'. In such a case, we will also introduce new variables for all "intermediate" values of the contained function invocations. To understand how to handle such formulas, let us first formalize the task of taking a term that contains multiple (hierarchical) function invocations, and breaking it into steps while introducing explicit variable names for intermediate values. (This is exactly what a compiler for any programming language does when parsing such a term!)

Task 3. Implement the missing code for the function `_compile_term(term)`, which takes a term whose root is a function invocation (possibly with nested function invocations further down the term tree) and returns "steps" that "break down" the given term, each of which is a formula of the form 'y=$f(x_1,\ldots,x_k)$', where y is a new variable name, f is a function name, and each of the x_is is either a constant name or a variable name, such that the left-hand-side variable of the last step (the "y of the last step") evaluates to the value of the whole given term.

predicates/functions.py

```
def _compile_term(term: Term) -> List[Formula]:
    """Syntactically compiles the given term into a list of single-function
    invocation steps.

    Parameters:
        term: term to compile, whose root is a function invocation, and which
            contains no variable names starting with ``z``.

    Returns:
        A list of steps, each of which is a formula of the form
        '`y`=`f`(`x1`,...,`xn`)', where `y` is a new variable name obtained by
        calling `next(fresh_variable_name_generator)`, `f` is a function name,
        and each of the `x`i is either a constant name or a variable name. If
        `x`i is a new variable name, then it is also the left-hand side of a
        previous step, where all of the steps "leading up to" `x1` precede those
        "leading up" to `x2`, etc. If all the returned steps hold in any model,
        then the left-hand-side variable of the last returned step evaluates in
        that model to the value of the given term.
    """
    assert is_function(term.root)
    for variable in term.variables():
        assert variable[0] != 'z'
    # Task 8.3
```

Example: Given the term 'f(g(g(0)),h(x))', the function should return the formulas 'z1=g(0)', 'z2=g(z1)', 'z3=h(x)', and 'z4=f(z2,z3)', in this order, corresponding to the following break-down of the term:

$$\underbrace{\text{'f(}\underbrace{\text{g(}\underbrace{\text{g(0)}}_{\text{z1}}\text{)}}_{\text{z2}}\text{,}\underbrace{\text{h(x)}}_{\text{z3}}\text{)'}}_{\text{z4}}$$

Guidelines: The new variable names used in the returned steps, in the order of the steps, should be generated by calling `next(fresh_variable_name_generator)` (this generator, which makes sure that numbering continues to increase between successive calls to your function, is imported for you from `logic_utils.py`[3]), and you may assume that variable names generated this way do not occur anywhere in the given term.

We can now see how the "compiler" that you implemented in Task 3 can be used to find a function-free analog of a formula whose root is a relation invocation whose arguments contain multiple nested function invocations, such as 'R(f(g(x)),h(2,y),3)'. Using this formula as an example, note that for the term 'f(g(x))', your "compiler" returns the two steps 'z1=g(x)' and 'z2=f(z1)', and for the term 'h(2,y)', your "compiler" subsequently returns the single step 'z3=h(2,y)'. The function-free analog formula will go over all the new variables (where it will go over the lists corresponding to the relation invocation arguments in left-to-right order, and for each argument it will go over the list of new variables created for it in the order of the steps returned by your "compiler"). For each new variable over which it goes, the function-free analog formula will existentially (∃) quantify over it, stating that relation invocation that corresponds to the function invocation in the step for this variable should be satisfied, and that all of these, as well as the root relation being satisfied, should hold simultaneously. For example, in our example the equivalent resulting formula is

'∃z1[(G(z1,x)&∃z2[(F(z2,z1)&∃z3[(H(z3,2,y)&R(z2, z3, 3))])])]'

Finding a function-free analog of a formula whose root is an equality is very similar (in this context, equality is no different than any other relation), and finally finding function-free analogs of composite formulas (i.e., formulas whose root is a Boolean operator or a quantification) can be done recursively, by respectively compositing the function-free analogs of their subformulas.

Task 4. Implement the missing code for the function `replace_functions_with_relations_in_formula(formula)`, which returns the function-free analog formula (constructed as described above) of the given formula that may contain function invocations.

predicates/functions.py

```
def replace_functions_with_relations_in_formula(formula: Formula) -> Formula:
    """Syntactically converts the given formula to a formula that does not
    contain any function invocations, and is "one-way equivalent" in the sense
    that the former holds in a model if and only if the latter holds in the
    canonically corresponding model with no function meanings.

    Parameters:
        formula: formula to convert, which contains no variable names starting
            with ``z``, and such that there exist no canonically corresponding
            function name and relation name that are both invoked in this
            formula.

    Returns:
        A formula such that the given formula holds in any model `model` if and
        only if the returned formula holds in
```

[3]So a second call to `_compile_term()` with the same term 'f(g(g(0)),h(x))' from the example above will return the formulas 'z5=g(0)', 'z6=g(z5)', 'z7=h(x)', and 'z8=f(z6,z7)' if implemented using this generator, which is the behavior that we seek.

```
        `replace_function_with_relations_in_model(model)`.
    """
    assert len({function_name_to_relation_name(function) for
                function,arity in formula.functions()}.intersection(
                    {relation for relation,arity in formula.relations()})) == 0
    for variable in formula.variables():
        assert variable[0] != 'z'
    # Task 8.4
```

Hints: As with many (but not all) other tasks in this half of the book, here too it is possible to have the exact same code that handles relation names also handle equality, and you should do precisely this. Use recursion for formulas that have quantifiers or Boolean operators at their root.

Your solution to Task 4 guarantees the following: for every model `model`, if we take `new_formula=replace_functions_with_relations_in_formula(formula)` and `new_model=replace_functions_with_relations_in_model(model)`, then `new_model` is a valid model of `new_formula` if and only if `model` is a valid model of `formula`. Note that this only gives us a guarantee regarding the truth values of the function-free `new_formula` in a model that is a function-free analog of some model (as opposed to in a model that is not the function-free analog of any model), that is, in a model where the meanings of the new relations correspond to meanings of functions. In particular, it is possible that `formula` has no valid model whereas `new_formula` has a valid model (in which some new relations have meanings that do not correspond to meanings of functions). We will now add a verification of this property of meanings of relations, thereby ensuring that the meaning of each new relation, in *any* valid model for the function-free `new_formula` (and not only in those returned by `replace_functions_with_relations_in_model()`) indeed corresponds to *some* function meaning.

Note that for a binary relation F to indeed correspond to some unary function f, we need both that (a) for every x there exists a value z such that $z{=}f(x)$: '$\forall$x[$\exists$z[F(z,x)]]', and (b) there exists only one such value: '($\forall$x[($\forall$z1[($\forall$z2[((F(z1,x)&F(z2,x))$\rightarrow$z1=z2)])])])'. So, we will construct a set of formulas by starting with the function-free equivalent of some original formula. For each relation name that corresponds to a function name in the original formula, we will then add to this set a formula (the conjunction of the two formulas above) that ensures that the meaning of this relation name indeed corresponds to a function. This will guarantee that the obtained set of formulas has the property that any valid model for this *set* can be decoded into a valid model for the original formula (i.e., is returned by `replace_functions_with_relations_in_model()` given some model). We will now implement this procedure, not only for a single original formula, but more generally for a set of original formulas.

Task 5. Implement in the missing code for the function `replace_functions_with_relations_in_formulas(formulas)`, which returns formulas containing a) function-free analogs of all the given formulas, obtained via Task 4, and b) for every function name that appears in the given formulas, a formula that verifies that the meaning of the corresponding relation that replaced it indeed corresponds to a meaning of a function.

predicates/functions.py

```
def replace_functions_with_relations_in_formulas(formulas:
                                                 AbstractSet[Formula]) -> \
```

```
        Set[Formula]:
    """Syntactically converts the given set of formulas to a set of formulas
    that do not contain any function invocations, and is "two-way
    equivalent" in the sense that:

    1. The former holds in a model if and only if the latter holds in the
       canonically corresponding model with no function meanings.
    2. The latter holds in a model if and only if that model has a
       canonically corresponding model with meanings for the functions of the
       former, and the former holds in that model.

    Parameters:
        formulas: formulas to convert, which contain no variable names starting
            with ``z``, and such that there exist no canonically corresponding
            function name and relation name that are both invoked in these
            formulas.

    Returns:
        A set of formulas, one for each given formula as well as one additional
        formula for each relation name that replaces a function name from the
        given formulas, such that:

        1. The given formulas hold in a model `model` if and only if the
           returned formulas hold in
           `replace_functions_with_relations_in_model(model)`.
        2. The returned formulas hold in a model `model` if and only if
           `replace_relations_with_functions_in_model(model,original_functions)`,
           where `original_functions` are all the function names in the given
           formulas, is a model and the given formulas hold in it.
    """
    assert len(set.union(*[{function_name_to_relation_name(function) for
                            function,arity in formula.functions()}
                           for formula in formulas]).intersection(
                               set.union(*[{relation for relation,arity in
                                            formula.relations()} for
                                           formula in formulas]))) == 0
    for formula in formulas:
        for variable in formula.variables():
            assert variable[0] != 'z'
    # Task 8.5
```

For every formulas set `formulas` and its corresponding function-free formulas set `new_formulas=replace_functions_with_relations_in_formulas(formulas)`, we now have a two-way assurance. First, as before, we have for every valid model `model` of `formulas` that `replace_functions_with_relations_in_model(model)` is a valid model of `new_formulas`. Second, we now also have the other direction: if `functions` is the set of function names used in `formulas`, then we also have for every valid model `model` of `new_formulas` that `replace_relations_with_functions_in_model(model, functions)` is a valid model of `formulas`. Therefore, there exists a valid model for the set `formulas` (that may contain function invocations) if and only if there exists a valid model the function-free set `new_formulas`. Your solutions in this chapter so far therefore programmatically prove the following theorem.

Theorem (Redundance of Functions)**.** *For every set F of formulas (which may contain function invocations), there exists a set F' of function-free formulas obtainable from F*

via an efficient syntactic procedure, such that F has a valid model if and only if F' has a valid model. Moreover, given a valid model for F' there is a simple natural way to convert it into a valid model for F and vice versa.

Admittedly, the above theorem is stated in a somewhat informal manner. While as computer scientists we have a reasonable understanding of the word "efficient" in "efficient syntactic procedure," the phrase "simple natural way" justifiably seems less grounded and precise. We intentionally do not want to dive too deeply into the precise meaning of this phrase in this book, but will note that the idea is that there is a one-to-one correspondence between models for F and models for F' (so in a strong sense the sets of the models that F and F' can represent are "as rich"), which has many additional desirable properties beyond the ability to convert models back and forth. For example, this correspondence is *local* in the sense that given a model for F', to find the meaning of a single function name (or other element name) in the corresponding model for F we need only look at a single meaning in F' and do not need any "global" information about F'. Interested readers are advised to seek out a text on **Model Theory**, and specifically read about **homomorphisms** between models, for a deeper look into the beautiful world that here we merely peek into.

2 Getting Rid of Equality

We now move to the second simplification of this chapter, eliminating equalities in formulas. Since you have already shown how to eliminate function invocations from formulas without losing any expressive power, we will assume henceforth that our formulas and models are function-free. Unlike in the previous section, we will start with transforming formulas as in Task 5, and only then move to converting models back and forth as in Tasks 1 and 2. Our strategy here for transforming formulas is very simple: replace any equality 'τ=σ' (where τ and σ are terms) with a matching invocation of the relation name 'SAME', i.e., 'SAME(τ,σ)'. Once again, we must also add some formulas that force 'SAME' to have the intended meaning, but the catch is that we cannot use the equality symbol in these formulas (otherwise, we will have gained nothing). So, which conditions that can be expressed without using equalities capture the meaning of the equality relation? First, it is clear that 'SAME' should have the basic properties of reflexivity (SAME(x,x)), symmetry (SAME(x,y) iff SAME(y,x)), and transitivity (SAME(x,y) and SAME(y,z) imply SAME(x,z)). These do not suffice, however. Indeed, the main requirement that will connect this new relation to its intended meaning is that 'SAME(τ,σ)' implies that in *every* formula where we have τ we can replace it by σ. For a unary relation 'R', this would mean requiring that '∀x[∀y[(SAME(x,y)→(R(x)→R(y)))]]', for a binary relation 'R' it would mean requiring that '∀x1[∀x2[∀y1[∀y2[((SAME(x1,y1)&SAME(x2,y2))→(R(x1,x2)→R(y1,y2)))]]]]', etc. If we introduce such a formula for each relation name that our formulas contain, then we are assured that in every formula we can replace any term by any "SAME term," without changing the truth value of the formula.

Task 6. Implement the missing code for the function `replace_equality_with_SAME_in_formulas(formulas)`, which returns equality-free formulas containing a) the equality-free analogs of the given formulas (obtained by replacing every equality with a matching invocation of the relation 'SAME'), and

b) additional formulas that ensure that 'SAME' has the required semantics (reflexivity, symmetry, transitivity, and being respected by all relations).

predicates/functions.py

```
def replace_equality_with_SAME_in_formulas(formulas: AbstractSet[Formula]) -> \
        Set[Formula]:
    """Syntactically converts the given set of formulas to a canonically
    corresponding set of formulas that do not contain any equalities, consisting
    of the following formulas:

    1. A formula for each of the given formulas, where each equality is
       replaced with a matching invocation of the relation name 'SAME'.
    2. Formula(s) that ensure that in any model for the returned formulas,
       the meaning of the relation name 'SAME' is reflexive, symmetric, and
       transitive.
    3. For each relation name from the given formulas, formula(s) that ensure
       that in any model for the returned formulas, the meaning of this
       relation name respects the meaning of the relation name 'SAME'.

    Parameters:
        formulas: formulas to convert, that contain no function names and do not
            contain the relation name 'SAME'.

    Returns:
        The converted set of formulas.
    """
    for formula in formulas:
        assert len(formula.functions()) == 0
        assert 'SAME' not in \
               {relation for relation,arity in formula.relations()}
    # Task 8.6
```

Having defined the transformation for formulas, similarly to the previous section we now wish to convert a model of the original formulas to a model of the equality-free formulas, in a way that preserves the truth value.

Task 7. Implement the missing code for the function `add_SAME_as_equality_in_model(model)`, which takes a (finite) model and returns a model that is identical to the given one, with the addition of a meaning of the 'SAME' relation that behaves like equality.

predicates/functions.py

```
def add_SAME_as_equality_in_model(model: Model[T]) -> Model[T]:
    """Adds a meaning for the relation name 'SAME' in the given model, that
    canonically corresponds to equality in the given model.

    Parameters:
        model: model that has no meaning for the relation name 'SAME', to add
            the meaning to.

    Returns:
        A model obtained from the given model by adding a meaning for the
        relation name 'SAME', that contains precisely all pairs (`x`,`x`) for
        every element `x` of the universe of the given model.
    """
    assert 'SAME' not in model.relation_meanings
    # Task 8.7
```

Your solutions to Tasks 6 and 7 guarantee that for every formulas set `formulas` and its corresponding equality-free formulas set `new_formulas= replace_equality_with_SAME_in_formulas(formulas)`, for every model `model` it is the case that `model` is a valid model of `formulas` if and only if `add_SAME_as_equality_in_model(model)` is a valid model of `new_formulas`. Once again, though, similarly to the previous section, we need to make sure that we have not introduced any additional models when removing equality. As in Task 2, we will do so by showing that every valid model of the new set of equality-free formulas can be converted into a valid model of the original set of formulas. When going in this "opposite direction," a conceptual difficulty arises: the semantics of equality is that for any two constant or variable names x and y, the truth value of the formula 'x=y' is *True* if and only if these two elements are indeed the same in the sense that the *meanings* (assigned by a model or by an assignment) of x and of y are both the same element of Ω. There is no way to force such semantics without the equality symbol! This means that in some sense, the new set of formulas supports richer models not supported by the original set of formulas. For example, consider the formula '∀x[∀y[x=y]]'. It is easy to see that every model that satisfies it must have only a single element. There is no way to force a model to have only a single element without using equality: the formula '∀x[∀y[SAME(x,y)]]' can be satisfied by a model with two elements (and in fact, can be satisfied by a model with any number of elements!), even if we demand that the meaning of 'SAME' be reflexive, transitive, and commutative. Nevertheless, the different elements in such a model really are *the same* as each other as we have put formulas in place requiring that they act in exactly the same way in every possible formula.

In general, whenever a model contains different elements that are "the SAME" as each other, if we keep only a single representative in our model for each such set of items that are "the SAME" as each other, then we force equality to behave like 'SAME'. Formally, the 'SAME' relation partitions the elements of the universe into **equivalence classes**, and our new model will have a single representative for each equivalence class. As an example, consider the following model:

- Universe $\Omega = \{0, \ldots, 10\}$.
- The meaning of a constant called '0', defined as $0 \in \Omega$.
- The meaning of a constant called '4', defined as $4 \in \Omega$.
- The meaning of a ternary relation 'Plus', defined such that for every $\alpha, \beta, \gamma \in \Omega$, we have that (γ, α, β) is in the meaning of 'Plus' if and only if $\gamma = \alpha + \beta$ modulo 4.
- The meaning of a ternary relation 'Times', defined such that for every $\alpha, \beta, \gamma \in \Omega$, we have that (γ, α, β) is in the meaning of 'Times' if and only if $\gamma = \alpha \cdot \beta$ modulo 4.
- The meaning of the relation 'SAME', defined such that for every $\alpha, \beta \in \Omega$, we have that (α, β) is in the meaning of 'SAME' if and only if $\alpha = \beta$ modulo 4.

It is straightforward to verify that the meaning of 'SAME' in this model is reflexive, symmetric, transitive, and respected by both other relations. For the latter property, note that indeed if $\alpha, \alpha' \in \Omega$ are "the SAME," $\beta, \beta' \in \Omega$ are "the SAME," and $\gamma, \gamma' \in \Omega$ are "the SAME," then (γ, α, β) is in the meaning of one of these relations if and only if $(\gamma', \alpha', \beta')$ is in the meaning of that relation. It is straightforward to see that the meaning of the relation 'SAME' in the above model has four equivalent classes:

$\{0,4,8\},\{1,5,9\},\{2,6,10\},\{3,7\}$. So, taking one representative from each equivalence class, say, 8 for the first, 1 for the second, 6 for the third, and 3 for the fourth, we obtain the following model, where the meaning of all relations are preserved, but equality behaves like 'SAME' does in the original model:

- Universe $\Omega = \{8,1,6,3\}$.
- The meaning of a constant called '0', and the meaning of a constant called '4', both defined as $8 \in \Omega$ (since 8 is the representative of the equivalence class that contains both original meanings $0 \in \Omega$ and $4 \in \Omega$).
- The meaning of a ternary relation 'Plus', defined such that for every $\alpha,\beta,\gamma \in \Omega$, we have that (γ,α,β) is in the meaning of 'Plus' if and only if $\gamma = \alpha+\beta$ modulo 4.
- The meaning of a ternary relation 'Times', defined such that for every $\alpha,\beta,\gamma \in \Omega$, we have that (γ,α,β) is in the meaning of 'Times' if and only if $\gamma = \alpha\cdot\beta$ modulo 4.

Of course, had we chosen 0 as the representative of the first equivalence class, 1 of the second, 2 of the third, and 3 of the fourth, we would have gotten what seems like a "nicer" model with $\Omega = \{0,1,2,3\}$, with '0' defined as $0 \in \Omega$, and with 'Plus' and 'Times' corresponding to addition and multiplication modulo 4, but looking more closely we notice that these two models are in fact identical up to "renaming" the four elements of the model, so the latter only seems "nicer" to us, but mathematically both are equivalent and we will be as happy with either of them.

It is worth examining what would have gone wrong had we just removed the relation 'SAME' from the original model *without* shrinking the universe to contain only one representative from each equivalence class. First and more obviously, we would have had a problem with formulas that contain both constants: the formula 'SAME(0,4)' holds in the original model while the corresponding formula '0=4' would not have held in the new model had we not shrunk its universe. (This formula *does* hold in the new model with only one representative from each equivalence class, since the meanings of both constants '0' and '4' in that model are the same: the universe element $8 \in \Omega$.) There is another more subtle yet more fundamental problem with not shrinking the universe, though: consider the formula '∃v[∃w[∃x[∃y[∃z[(~SAME(v,w)&(~SAME(v,x)&$\cdots$(~SAME(x,z)&~SAME(y,z))$\cdots$))]]]]]' where the conjunction goes over all 10 pairs of distinct variable names out of 'v'…'z', checking that the meanings no two of these variables are "the SAME." This formula does not hold in the original model. Nonetheless, the corresponding formula '∃v[∃w[∃x[∃y[∃z[(~v=w&(~v=x&$\cdots$(~x=z&~y=z)$\cdots$))]]]]]' would have held in the new model had we not shrunk its universe. (This formula does *not* hold in the new model with only one representative from each equivalence class, since it universe does not contain five distinct elements.) Indeed, shrinking the universe to contain only one representative from each equivalence class is the only way to force equality in the new model to behave like 'SAME' does in the original model. Your final task in this chapter is to implement this conversion.

Task 8. Implement the missing code for the function `make_equality_as_SAME_in_model(model)`, which returns an analog of the given model without the meaning for 'SAME', and containing exactly one representative from each equivalence class of the meaning of 'SAME' in the given model. It is required

that for every formulas set `formulas` and its corresponding equality-free formulas set `new_formulas=replace_equality_with_SAME_in_formulas(formulas)`, for every model `model` it is the case that `model` is a valid model of `new_formulas` if and only if `make_equality_as_SAME_in_model(model)` is a valid model of `formulas`.

predicates/functions.py

```
def make_equality_as_SAME_in_model(model: Model[T]) -> Model[T]:
    """Converts the given model to a model where equality coincides with the
    meaning of 'SAME' in the given model, in the sense that any set of formulas
    holds in the returned model if and only if its canonically corresponding set
    of formulas that do not contain equality holds in the given model.

    Parameters:
        model: model to convert, that contains no function meanings, and
            contains a meaning for the relation name 'SAME' that is reflexive,
            symmetric, transitive, and respected by the meanings of all other
            relation names.

    Returns:
        A model that is a model for any set `formulas` if and only if
        the given model is a model for
        `replace_equality_with_SAME(formulas)`. The universe of the returned
        model corresponds to the equivalence classes of the 'SAME' relation in
        the given model.
    """
    assert 'SAME' in model.relation_meanings and \
           model.relation_arities['SAME'] == 2
    assert len(model.function_meanings) == 0
    # Task 8.8
```

The guarantees of your solutions to Tasks 7 and 8 programmatically prove the following theorem.

Theorem (Redundance of Equality)**.** *For every set F of formulas (which may contain equality), there exists a set F' of equality-free formulas obtainable from F via an efficient syntactic procedure, such that F has a valid model if and only if F' has a valid model. Moreover, given a valid model for F' there is a simple natural way to convert it into a valid model for F and vice versa.*

Once again, we resort to an admittedly informal statement of the above theorem. Recall that we have remarked that in the Theorem on Redundance of Functions, "simple natural way" in particular meant that the correspondence between models for F and models for F' was one-to-one. To make our informality in the above theorem even worse, by now you know that when removing equality, the correspondence between models for F and models for F' is not one-to-one (indeed, several models for F' will be mapped to the same model for F since we are collapsing every equivalence class into a single representative, and for example it is impossible to unambiguously recover the size of each equivalence class after it has been collapsed). Nonetheless, it turns out that this correspondence maintains many of the natural properties of one-to-one correspondences—and in particular still, in a precise strong sense, the models that F and F' can represent are "as rich"—as well as many additional desirable properties such as the *locality* property discussed above. Once again, interested readers are advised to seek out a text on **Model Theory** (and specifically on **homomorphisms** between models) for a more detailed and formal discussion.