But keeping these restrictions in mind, our approach is sound and simple.

0.1 Basic concepts

0.1.1 Definition basic concepts

The conceptional, semantic universe of Haskell can be separated into four distinct kind of entities:

- Values, comprising all things in the most basic sense: numbers, characters, strings, lists, tuples, and functions.
- Types. Haskell is strongly typed in the sense that every value has a well–defined type, and it is statically typed in the sense that this doesn't change.
- Type classes which is the Haskell terminology for what mathematicians would call a (single–carrier) theory or model class. It is not a class in the object-oriented terminology, but a signature and axiom set for a class of structures.
- Modules. A module is a combination of values, types and type classes, well–defined in itself and encapsulated in a separated namespace by a module name.

0.1.2 Remark

If we take this semantic approach, there are also certain things in Haskell that fit in only after some adjustments.

- Actions, i.e. IO (input-output) interactions can't be understood in a purely functional language like Haskell, because they don't behave determinstically. User input is not a function, because it differs each time. But the Haskell designers have developed a mechanism to abstract this away and to integrate actions into the value universe. This is the famous monad concept and there is a `Monad` type class with several instances.
- There is a universal kind of social equality among all the values and functions are *first–class* in the sense that they can be arguments and results of other functions, just as any other value can. But we can only develop a full semantical perspective, if we also consider certain Haskell syntax constructions as functions. For example the ternary "`if..then..else..`" function is usually not considered a function in Haskell and is not treated "first–class".

0.1.3 Remark

Functional programming languages are really different to declarative ones and it is often said that they are more similar to mathematics. That is true in many respects and we will approach Haskell in this perspective. But there are also certain things in Haskell, which are really different from the traditional mathematical point of view.

- In mathematics, a type is a set and saying that the value c is of type X (written "$c : X$" in mathematics and "c`::`X" in Haskell), is saying that c is a member of the set X. But in Haskell, sets don't exist at all, at least not as values, only as types. And some of the usual type constructors like the Cartesian product $X \times Y$ (in Haskell, that is "`(`X`,` Y`)`") or function constructor $X \longrightarrow Y$ (i.e. "X `->` Y" in Haskell) do exist, but others like the union $X \cup Y$ or intersection $X \cap Y$ don't. The type system of Haskell only allows certain kind of constructions, some of them are less often used in mathematics.
- Note, that values, types, type classes and modules are really different things in Haskell. Also in the sense that they cannot be converted into each other. In mathematics, it is trivial to take a function $f : X \longrightarrow Y$ (which is a value) and extract its domain $\mathbf{dom}(f) = X$ (which is a type). But there is no way to do these things in Haskell!
- Haskell tries to abstract the value from its computation and the time and memory necessary to evaluate, is abstracted away, as it is in mathematics. But there have been a lot of compromises as well. For example, there is an `Integer` type for integers without any limit, but also an `Int` type for integers in a limited range with fast built-in arithmetic.
- And there is of course, the decision for a *lazy* order of evaluations, rather than the *strict* way. The final results are the same. But sometimes, lazy evaluation has results, where the strict strategy has none and this is creatively exploited by the adoption of infinite lists.
- Another consequence, closely related to the lazy strategy is the fact that each type has one more member than the mathematical set would have: `undefined` (or $\perp$) is fine in Haskell.