

From: [Molnár, Vince](#) on behalf of [Vince Molnár Budapest University of Technology](#)
To: [SWG](#)
Subject: [Semantics WG] Abstract thoughts about "abstract"
Date: Friday, June 20, 2025 12:38:27 PM

Definitions

According to the non-normative section 7 in the KerML spec:

"A type is specified as *abstract* by placing the keyword **abstract** before the keyword **type**. A type that is not abstract is called a *concrete* type. Declaring a type to be abstract means that all instances of the type must also be instances of at least one concrete type that directly or indirectly specializes the abstract type."

As recorded by [KerML-7](#), there is no mathematical semantics defined to capture the intent above. This would be relatively easy to add, e.g.,

$$t.isAbstract \Rightarrow (t)^T = \bigcup_{t_s \in t.specialization.specific} (t_s)^T$$

$$(t.isAbstract \Rightarrow (t)^T = \bigcup_{t_s \in t.specialization.specific} (t_s)^T)$$

That is, if t is abstract, its interpretation is equal to the union of the extents of its specializations, which will be the empty set if there is no specialization. It is relatively easy to show that this is equivalent to saying there is no element in the extent of t that is not also in the extent of at least one of its specializations (which does not have to be concrete because the same rule will apply to specializing abstract types, leading to a recursive proof).

Implications

1. Since `isAbstract` is defined for `Type`, a concrete feature typed by an abstract classifier seems to be able to have a value without any further specialization (as already pointed out by some people), as the feature itself is a concrete specialization of the abstract classifier. However, the mathematical semantics actually rules this case out.
 - a. The value of the feature will be a sequence in the extent of the feature.
 - b. Due to the axiom about specialization (axiom 1 in 8.4.3.2, "All sequences in the interpretation of a `Type` are in the interpretations of the `Types` it specializes."), this sequence will be in the extent of the abstract classifier as well, because "typed by" is a kind of specialization.
 - c. According to axiom 1 in 8.4.3.3 ("If the interpretation of a `Classifier` includes a sequence, it also includes the 1-tail of that sequence."), the 1-tail of the sequence (which is **not** in the extent of the feature) must be in the extent of the abstract classifier.
 - d. However, according to the rule for abstract types, that value (the 1-sequence) should also be in the extent of a specializing type (which has to be either a concrete type or ultimately be specialized by a concrete type). If it *is* in the extent of such a type, the strongest type of the value is not abstract and we are OK, otherwise we have a contradiction, which means that the feature cannot have the sequence in its extent.

2. Types that have abstract features with lower multiplicity greater than zero can be instantiated only with types that are not mentioned in the definition of the type.
 - a. This might be interesting in execution because we cannot blindly instantiate the type of the feature to get a value, but have to explore its specializations to choose a suitable type for the value (if any).
 - b. However, there are cases in which these types **can be instantiated**, contrary to types in OO programming languages, where a class must be abstract if it has an abstract member. Again, this is **not** the case here.
 - c. What happens if there is no concrete type in the model that directly or indirectly specializes the abstract one? Are we “so open-world” that we can infer the existence of types, that is, we can just come up with a concrete specialization that is not in the model?
 - i. If so, what is the use of semantic limitations (we could just as well invent new specialization relationships or anything)?
 - ii. If not, then this means some types with abstract features **cannot be instantiated**, but to decide, we have to traverse all the specializations of the abstract type (and potentially some of the values will be excluded by other constraints, so finding a concrete type does not guarantee there will be a legal value).

Uses of ‘abstract’

As I mentioned at the Denver meeting, I believe an abstract feature or type is implicitly a derived union of all of its specializations (this is also captured in the formalization above). This could be quite useful in some scenarios:

1. Package-level usages that serve to “collect” values (most of the ones in the library) could be abstract, which would emphasize the goal of aggregating the content of user-defined features and actually prevent values that are not coming from the modeler. It would also force users to subset or redefine them (for example, ‘lengths’), helping them acknowledge that the values of these features will be “local” to the type they are working with (as opposed to treating package-level features as global variables).
 - a. [subjective] For the same reasons, I will personally teach that package-level usages should be abstract as a best practice, to emphasize the pattern that they are “feature prototypes” that can be brought into types (which is the only pattern that is acceptable to me at this time).

2. Even features that are not package-level can implement such an “aggregation”, in which case, making them abstract communicates the intent that they should not have their own values.
 - a. Because of this, I proposed that perhaps ‘derived’ could imply ‘abstract’, as derived features are expected to be bound to something, which also entails specialization (so it does not hurt existing use cases). Without a binding, a derived feature that is implicitly also abstract would have all the values from all of its subsets, similarly to “derived union” in UML, but without any extra keywords. This is a separate topic about isDerived and should not influence the discussion about isAbstract.
 - b. Also, note that we should avoid assuming any kind of causality in these discussions. That is, the superset and the subset mutually define each other – a subsetting relationship does not specify whether we “pick” values for the subset from the superset, or we aggregate and potentially extend values from the subset to get the extent of the superset.
3. Many (if not all) references should be abstract to force their values to come from composite features. This would promote a clean execution pattern that starts with the instantiation of composite features, then finds suitable values out of the already existing ones for the references. I acknowledge that there might be justified exceptions.
4. A special case of this is event occurrence usages (including perform action, exhibit state, and include use case). These constructs make the most sense if used along with a reference subsetting that points to the set of values to pick from, but the language allows not to specify a reference subsetting, in which case the event occurrence can have its own value. I believe it is more sensible to model these as ‘abstract event occurrences’, which can then be redefined to actually refer to something once the target of the reference is known (see “realizing events”). Adding the ‘abstract’ keyword here clarifies that we do not yet know what will realize the event, and there should be no “random” values in the extent of the feature.

Since the above could already easily cover several meetings, I propose to start a discussion also in ~~email~~ [JIRA] – written arguments tend to be more precise anyway, which is kind of useful when discussing semantics.