

Multi-Language Library Development

From Haskell Type Classes to C++ Concepts

Marcin Zalewski¹, Andreas Priesnitz¹, Cezar Ionescu²,
Nicola Botta², and Sibylle Schupp¹

¹ Chalmers University of Technology
Gothenburg, Sweden

{zalewski|priesnit|schupp}@cs.chalmers.se

² Potsdam Institute for Climate Impact Research,
Potsdam, Germany
{ionescu|botta}@pik-potsdam.de

Abstract. We define a mapping from generic Haskell specifications to C++ with concepts, a recent extension to C++, that can ultimately be automated. More specifically, we provide a translation from Haskell multi-parameter type classes with functional dependencies to ConceptC++. Our translation consists of three major parts: the division of Haskell class variables into ConceptC++ concept parameters and associated types, the corresponding division of superclasses in the context of a type class, and the linearization of Haskell ASTs to the concrete syntax of ConceptC++. We also discuss cases in which there is no single correct translation from classes with functional dependencies to concepts. Our translation handles these cases in a reasonable way and is well-defined for the cases most common in practice. The translation is motivated by an ongoing project for distributed adaptive finite volume methods, in which software components are modeled in Haskell and implemented in C++.

1 Introduction

The goal of the S project at the Potsdam Institute for Climate Impact Research [1–3] is to provide reusable, generic software components for distributed adaptive finite volume methods. In the development process, Haskell is used as high-level modeling language and C++ as implementation language: Haskell allows reasoning about the soundness of constructs and algorithms, whereas C++ allows for an efficient implementation, which is crucial in numerical applications. The S project started in 2004, its software is released since early 2007. Up to now, however, the transition from Haskell to C++ was not formally specified.

The purpose of this paper is to define the mapping from generic Haskell specifications to C++ so, that it ultimately can be automated. The principal constituents of these generic specifications are abstractions of types and restrictions on possible parameter types to these abstractions. In Haskell, abstractions on *type variables* are represented as *type classes* that are parameterized by these variables [17]. *Superclasses* impose restrictions on the parameters. In *multi-parameter*

type classes, relationships among the parameters are stated as *functional dependencies* [13]. In `ConceptC++` [10], on the other hand, which anticipates the upcoming revision of the C++ standard, abstractions are expressed by *concepts* [9, 16], which take type parameters and require *associated types*. Restrictions on concept parameters and associated types are stated as *refinements* and as *requirements*.

In previous work, these language features have been compared with each other and with similar techniques for expressing *constrained genericity* in the same or the other language [5, 11, 12, 14], and Haskell language elements and constructs have been emulated in C++, for example in the `FC++` library [15]. In this paper, we go one step further by developing rules for translating Haskell specifications into the corresponding `ConceptC++` code.

The presentation of this paper is based on a particular part of the `S` project software, namely the generic Relation-Based Algorithm (RBA) pattern. We first provide the necessary terminology of constructs in Haskell and `ConceptC++`, using the RBA as example (Sect. 2). The translation rules themselves are discussed in Sect. 3. In Sect. 4, we outline future work and conclude.

2 Background and Terminology

The two corresponding features considered in this paper are Haskell *type classes* and `ConceptC++` *concepts*. We introduce the necessary terminology by way of the RBA example. First we briefly describe the idea of RBAs, then we discuss a generic Haskell specification and the corresponding `ConceptC++` code, which are based on type classes and concepts.

The Relation Based Algorithm pattern is a simple, yet powerful, computation pattern that has been developed within the `S` project and now plays a central role in the design of the software. In its original form, the RBA can be described by the following snippet of Haskell code [1]:

```
rba :: [[Int]] -> [a] -> ([a] -> b) -> [Int] -> [b]
rba rss fs h js = [ h [ fs!!i | i <- rss!!j ] | j <- js ]
```

RBA takes as input a relation `rss`, a function `f`, a user-defined function `h`, and the points `js` at which to evaluate the relation `rss`. The details of the workings of an RBA and how it is used in the numerical software, are beyond the scope of this paper.

Presenting the RBA using list comprehension syntax has the advantage of simplicity, but also limits the RBA to specific types and hides some semantic nuances, including the precise requirements on the underlying container type. Most importantly for our purpose, however, it complicates the mapping from Haskell to `ConceptC++` since the details of the desired implementations differ considerably. As a first step towards the translation, we therefore lift the RBA to a generic version. The translation itself can then take place at a higher level of abstraction that captures the essence of the RBA and is not cluttered with inessential details of a problem. The generic RBA is shown in Listing 1.

Lines 2–10 of the listing define the necessary type classes and the polymorphic function expressed in terms of these type classes. Each type class defines

Listing 1 Specification of generic RBA in Haskell

```
1 -- Generic RBA
2 class Functor f where
3   fmap :: (a -> b) -> f a -> f b
4 class Set f => LambdaRelation r f a b | r -> f a b where
5   lambda :: r -> a -> f b
6 class DiscreteFunction f a b | f -> a b where
7   value :: f -> a -> b
8 rba rss fs h js = fmap f js
9   where f j = h (fmap g (rss 'lambda' j))
10          where g i = fs 'value' i
11
12 -- Making it work with lists
13 instance LambdaRelation [[a]] [] Int a where
14   lambda rss x = rss !! x
15 instance DiscreteFunction [a] Int a where
16   value fs i = fs !! i
```

class methods, one per class in this case, which must have a unique name in the top-level namespace. This restriction is necessary because of Haskell's *type inference* mechanism that attempts to automatically deduce the type of each term. Because the name `value` is reserved for the `DiscreteFunction` class, the Haskell compiler can produce the corresponding constraint on line 10 without any explicit annotations from the user.

Each type class introduces at least one (only one in Haskell 98) type variable. The type variables may be used to specify class methods and must all be mentioned in the type of each class method. For example, the type of the `value` function on line 7 mentions all three variables `f`, `a`, and `b` of the `DiscreteFunction` class and, therefore, is legal. Classes can require a certain *context*. For example, the `LambdaRelation` class requires that there must be an instance of the class `Set` (`Set` is not defined in Listing 1) with variable `f`. In other words, `Set` is a *superclass* of `LambdaRelation`.

The classes `LambdaRelation` and `DiscreteFunction` specify *functional dependencies* [13]: `r -> f a b` and `f -> a b`, respectively. A functional dependency of the form `lhs -> rhs` has two important implications. First, it states that all type variables in `rhs` are *reachable* given the variables in `lhs`; the notion of reachability extends the restriction on types of class methods having to mention all class variables—it is enough that all variables are reachable from that type. Second, a functional dependency restricts the possible *instances*. For example, once the instance on lines 15–16 is defined, no other instance is allowed with `[[a]]` as the first argument.

Type classes implement *ad-hoc* polymorphism [4, 17], in which the implementation of an overloaded function must be specified for each set of possible types

Listing 2 Translation of RBA type classes to ConceptC++

```
1 // concept example
2 concept DiscreteFunction<typename F>
3 : Function<F> // refinement example
4 {
5     typename domain_t;
6     typename codomain_t;
7     requires CopyConstructible<codomain_t>; // requirement example
8     codomain_t value(const F&, const domain_t&);
9 };
10
11 // concept_map example
12 template<typename T>
13 concept_map DiscreteFunction<std::vector<T>> {
14     typedef std::vector<T> function_t;
15     typedef typename function_t::size_t domain_t;
16     typedef typename function_t::value_t codomain_t;
17     codomain_t value(const function_t& f, const domain_t& d)
18         { return f[d]; }
19 };
20
21 // RBA algorithm
22 template<typename Rel, typename Fun, typename H, typename Js>
23 requires FunctConstr<Js>,
24         Funct<Js, FFun<H, Rel, GFun<Fun>, FunctConstr<Js>::value_t>>
25 Funct<Js, FFun<H, Rel, GFun<Fun>, FunctConstr<Js>::value_t>>::result_t
26 rba(Rel& r, Fun& f, H& h, Js& js) {
27     GFun<Fun> gh(f);
28     FFun<H, Rel, GFun<Fun>, FunctConstr<Js>::value_t> fh(h, r, gh);
29     return fmap(fh, js);
30 }
```

with which the function may get called. Each instance declaration specifies one case for one set of arguments, some of them possibly polymorphic themselves. For example, the instance declaration on lines 15–16 provides the definition of the overloaded function `value` from the `DiscreteFunction` class for any list `[a]` (where `a` is a type variable), the type `Int`, and the type `a` of the elements of the list. When the function `value` is called with the corresponding arguments, the Haskell compiler finds that instance and the specific definition of the `value` function.

Listing 2 contains parts of (simplified) ConceptC++ code corresponding to the Haskell code presented in Listing 1. *Concepts* in ConceptC++ correspond to type classes in Haskell. The `DiscreteFunction` concept, listed in lines 2–9, represents the same abstraction as the Haskell type class with the same name. A concept has *type parameters*, enclosed in angle brackets after the name of the concept, just

```

TypeClass:
  class [SuperClass+ =>] TypeClassId TypeVarId+ [1 FunDeps]
Superclass:
  TypeClassId TypeExp+
TypeExp:
  TypeVarId | TypeConstr | ConcreteType
TypeConstr:
  TypeConstrId TypeExp+
FunDeps:
  Dependency+
Dependency:
  TypeVarId+ -> TypeVarId+

```

Fig. 1. Grammar of type classes without class methods

as a Haskell class has type variables. In Listing 2, the `DiscreteFunction` concept, for example, has one parameter `F`. The example is extended with a *refinement* clause: any `DiscreteFunction` must also be a `Function` and, therefore, provide all associated types and operations required by the `Function` concept.

Concepts also have *associated types* written as a `typename` declaration in the concept body. Associated types are determined by the parameters of a concept; for example, the `domain_type` and `codomain_type` associated types of the `DiscreteFunction` concept depend on the concept parameter `F`. Related to associated types are *inline requirements*, for example, the requirement that `codomain_t` be `CopyConstructible` (line 7). Inline requirements are similar to refined concepts but, in difference from refined concepts, may mention associated types (for further distinctions between refinements and requirements see [8]). Finally, *concept operations*, declared within a concept body, correspond to Haskell class methods; each operation has a signature specifying its argument and result types.

Lines 12–19 give an example of a *concept map*, more specifically, a *template concept map*. Without getting into details, the concept map states that `std::vector`, with any element type, models the `DiscreteFunction` concept and provides a definition of the associated types and operations required by the concept. The *constrained template rba* (lines 22–30) corresponds to the polymorphic Haskell function `rba`. The two important things to notice are that all requirements must be explicitly given in a `requires` clause of the template and that Haskell `where` expressions are represented by a `ConceptC++` object. The need to specify requirements explicitly and to represent Haskell `where` expressions with objects stems from the fact that `ConceptC++`, unlike Haskell, does not perform type inference.

3 Translation Rules

In this section we describe the translation from Haskell multi-parameter type classes with non-constructor type variables and functional dependencies to concepts in `ConceptC++`. The grammar of the sublanguage of Haskell we consider

```

[[ class superClasses => C typeVars | funDeps ]] =
concept  $\tilde{C}$  <reduce(paramTypeVars(typeVars, funDeps), typeParamDecl, ',')>
: reduce(refinementClasses(paramTypeVars(typeVars, funDeps), superClasses),
         conceptInstance,
         ',')
{
  reduce(assocTypeVars(funDeps), assocTypeDecl,  $\epsilon$ )
  require reduce(requirementClasses(assocTypeVars(funDeps), superClasses),
                conceptInstance,
                ',');
}

```

Fig. 2. Essence of translation rules

is specified in Fig. 1. Throughout the translation process, we assume that the Haskell code given as input is valid in Haskell 98 with Glasgow extensions.

A class declaration, as specified by the grammar, consists of an optional context, the name of the class, the type variables of the class, and optional functional dependencies. The context consists of superclass requirements where each superclass can be thought of as a prerequisite for the currently declared class. A superclass takes an appropriate number of arguments where each of the arguments can be a type variable, an applied type constructor, or a concrete type. For example, for a type class with one variable, `a`, one can require that the type variable is equality-comparable and that it can be reduced to an `Int` by requiring the context (`Eq a, Reducible a Int`). Haskell puts some additional semantic restrictions on the superclasses in a class context [7] but we do not enumerate them here.

In addition to superclass requirements, functional dependencies are a feature crucial to generic programming in Haskell. Each class can introduce a set of dependencies of the form `lhs -> rhs` where both `lhs` and `rhs` are sets of type variables. As discussed in Sect. 2, functional dependencies allow one to tie types on the right hand side of a dependency to the types on the left hand side. For example, in Listing 1, the functional dependency `f -> a b` is crucial to properly express the concept of a `DiscreteFunction` on line 15 since one function type `f` should always be tied to only one domain and codomain type, represented by the type variables `a` and `b`.

In the remainder of this section we describe the translation rules. The presentation is divided into two parts. First, in Sect. 3.1, we discuss the four top-level translation steps, related to the four syntactic parts of a C++ concept. Then, in Sect. 3.2, we describe the internals of the translation: the computations necessary to process functional dependencies and to split a Haskell class context into either refined concepts or inline requirements, and the linearization process.

3.1 Top-Level Translation Steps

The top-level, logical view of the translation from Haskell type classes to ConceptC++ concepts is given in Fig. 2 as the composition of translation functions; the functions themselves are defined in Appendix A. The concrete syntax is typeset in a `typewriter` font; all expressions typeset in *italics* evaluate to a set of abstract syntax trees, and all *underlined* identifiers name a linearization function that produces concrete ConceptC++ syntax. The tilde “~” signifies identifier translation. The input to the translation is an abstract syntax tree of the grammar in Fig. 1.

The translation can be divided into four steps, marked by four *reduce* linearizations in Fig. 2, each of which is related to one part of a concept:

1. the parameters of the concept,
2. the list of refined concepts,
3. the associated types of the concept,
4. the requirements on concept parameters and associated types.

Given a Haskell type class with multiple parameters, some of the class variables will be translated into concept type parameters and some into associated types of the concept; the functional dependencies of a class control the division. In general, every class type variable could be translated into a concept parameter. However, that translation would make generic software less practical because quite a large number of parameters would have to be passed around [6]. In ConceptC++, and even earlier in plain C++, associated types are a natural and widely used way of decreasing the number of parameters. Whenever possible, we therefore target associated types (steps 1 and 3); the next section discusses the details. An extension of Haskell with a feature corresponding to associated types in ConceptC++ has been proposed [5] but is neither widely used yet nor included in Glasgow extensions to Haskell 98. If it were, it could directly guide the translation.

The division of superclasses in a class context into either refined concepts or inline requirements (steps 2 and 4) depends on the corresponding division of class variables into concept parameters and associated types (steps 1 and 3). Each superclass must mention at least one type variable in its type expression. Those superclasses that depend on variables translated into associated types obviously cannot be translated into refined concepts.

3.2 Internal translation steps

The four high-level translation steps from the previous section Sect. 3.1 depend on three kinds of internal computations: computations for processing functional dependencies, computations for splitting a Haskell class context into refined concepts and inline requirements, and the linearization process. We further elaborate on each of the three computations.

In most practical cases, a class specifies only one functional dependency `lhs -> rhs` with all class variables mentioned either in `rhs` or `lhs`, see the classes

in Listing 1 for an example. In this case, the variables on the right hand side of the dependency can safely be translated into associated types of a concept, the variables on the left hand side can be translated into concept parameters, and the functional dependency is upheld: only one concept map with particular arguments for the left hand side variables is possible and thus only one corresponding assignment of types to variables on the right hand side can be made. In the cases when more than one functional dependency is present or not all class variables are mentioned in the functional dependencies, however, it is not clear which class variables should become associated types and, generally, not possible to translate that type class to a single concept in a semantics-preserving way. Any exact solution would have to ensure that only one assignment of the types on the right hand side of every dependency in a class is possible for a corresponding assignment of types to variables on the left hand side, but such solution is not possible with the translation of a class into a single concept where class variables are divided only into the two sets of concept parameters and associated types. Instead of providing an exact solution, we choose an approximation of translating all variables that occur only on the right hand sides of the functional dependencies into the corresponding associated types. That translation does not preserve the semantic condition of functional dependencies but it safely chooses those variables that can always be translated into associated types. Associated type synonyms, recently proposed for Haskell [5], share with associated types in ConceptC++ the problem that they cannot be used to correctly represent functional dependencies.

Once the division of type variables into concept parameters and associated types of a concept is complete, the superclasses in the class context must be divided in the corresponding way. Because of the scoping rules in ConceptC++, every type class that mentions at least one type variable that has become an associated type must be translated into an inline requirement—not all concept parameters corresponding to the variables mentioned in the superclass are in the scope of the refine clause. To decide whether a superclass will become a ConceptC++ requirement, we compute the set of all type variables mentioned in the type expressions of the superclass (see the grammar in Fig. 1) and check if any type variable that will be translated into an associated type is an element of that set. The classes that are then translated into refined concepts are simply the difference between all superclasses and the ones translated into inline requirements.

The final kind of internal helper functions are the linearization functions. These functions comprise the *reduce* function and the specific functions corresponding to the syntactic parts of a concept. The reduce function takes a set of abstract syntax trees (ASTs) of the grammar in Fig. 1, a particular linearization function, and a delimiter that can be an empty string. The particular linearization function is applied to every AST and the results are concatenated to a single string in an undefined order using the delimiter as separator. All linearizations are rather straightforward except the *conceptInstance* function. The type expressions of a superclass in Haskell may involve a mix of type variables and particular

types. For example, a superclass `Collection c [a]` may be used to state that `c` must be a collection of lists of `as`. To translate this superclass into a refined concept or an inline requirement, one needs to first decide how to translate `[a]`. In general, there is no “correct” translation between particular types of the two languages—one could, for example, target in `ConceptC++` both `Collection<c, std::list<a>>` and `Collection<c, std::vector<a>>` to translate the Haskell class `Collection c [a]`. We therefore do not specify how to translate particular types and rather leave the particular mapping up to the application. An exception is the Haskell unit type `()`, a particularly important concrete type that closely corresponds to the C++ `void` type, thus can always be mapped to it.

4 Conclusions and Future Work

We have considered a practical example of multi-paradigm library development, namely the `S` project. In the `S` project, Haskell is used for modeling, specifications, and test implementations, while C++ is used for the actual implementation. Currently, the C++ code is not directly derived from specifications in Haskell and must be kept in sync with the specification manually. We have described a mapping between Haskell and C++ that is based on two kinds of abstractions, Haskell type classes on the one hand and C++ extended with concepts (`ConceptC++`) on the other hand.

The important parts of the mapping between type classes and concepts are the division of Haskell class variables into `ConceptC++` concept parameters and concept associated types, the corresponding division of superclasses in the context of a type class, and the linearization of Haskell ASTs to the concrete syntax of `ConceptC++`. Our translation correctly handles the cases most common in practice when there is one functional dependency and the dependency mentions all class variables. In the other cases, our translation chooses an approximate solution that does not completely preserve the semantics of dependencies but results in a solution that will fit many practical cases. Our translation also correctly handles superclass declarations. When they involve a particular Haskell type, however, our translation does not specify how such type should be translated. Since there is no “correct” mapping between particular types of Haskell and `ConceptC++`, it is best to leave that translation to the developers, who can decide on a case-by-case basis. An exception is the Haskell unit type `()`, which can be translated into the `void` type in `ConceptC++`.

Ultimately, our goal is to translate Haskell generic algorithms to constrained templates in `ConceptC++`. As the immediate next step towards that goal, we plan to tackle the translation of Haskell class methods into `ConceptC++`. Since class methods can introduce new type variables and have their own context, their translation is not trivial. Interesting questions here center around the type computations that might be required to determine the result type of an operation. Briefly, in `ConceptC++`, type variables in class methods must become concept type parameters, so that clients of a concept can provide the necessary type computation. We also need to extend our translation by constructor classes.

Such extension requires the representation of higher-kinded types and, again, type computations in `ConceptC++`, which both are non-trivial tasks. A possible solution could be to introduce concepts that represent applied type constructors; such concepts would allow for ‘pattern matching’ on an applied constructor to obtain its arguments.

Acknowledgments

The authors acknowledge the Swedish Foundation for International Cooperation in Research and Higher Education (STINT) and the Deutsche Akademische Austauschdienst (DAAD) for their partial support of that work. We also thank Gustav Munkby for discussions and Doug Gregor for answering questions on `ConceptC++`.

References

1. N. Botta and C. Ionescu. Relation-Based Computations in a Monadic BSP Model. <http://www.pik-potsdam.de/research/research-domains/transdisciplinary-concepts-and-methods/s/PARCO-S-06-00112.fdf>, Sept. 2006. Submitted for publication to *Parallel Computing*, Elsevier.
2. N. Botta, C. Ionescu, R. Klein, and C. Linstead. S—Software Components for Distributed Adaptive Finite Volume Methods. <http://www.pik-potsdam.de/research/research-domains/transdisciplinary-concepts-and-methods/s>, 2007.
3. N. Botta, C. Ionescu, C. Linstead, and R. Klein. Structuring distributed relation-based computations with SCDRC. Technical Report 103, Potsdam Institute for Climate Impact Research, Sept. 2006.
4. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.
5. M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated Type Synonyms. In *Proc. of the 10th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 241–253, New York, NY, USA, 2005. ACM Press.
6. R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A Comparative Study of Language Support for Generic Programming. In *Proc. of the 2003 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’03)*, Oct. 2003.
7. Glasgow Haskell Compiler. <http://haskell.org/ghc/>.
8. P. Gottschling. Fundamental Algebraic Concepts in Concept-Enabled C++. Technical Report 638, Indiana University, 2006.
9. D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. *SIGPLAN Notices*, 41(10):291–310, 2006.
10. D. Gregor and B. Stroustrup. Proposed Wording for Concepts. Technical Report N2193=07-0053, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Mar. 2007.
11. J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An Analysis of Constrained Polymorphism for Generic Programming. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming 2003: Proc. of the MPOOL Workshop at OOPSLA’03*,

- John von Neumann Institute of Computing Series, pages 87–107, Anaheim, CA, Oct. 2003.
12. J. Järvi, J. Willcock, and A. Lumsdaine. Concept-Controlled Polymorphism. In F. Pfennig and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 228–244. Springer Verlag, Sept. 2003.
 13. M. P. Jones. Type Classes with Functional Dependencies. In *Proc. 9th European Symp. on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.
 14. S. Kothari and M. Sulzmann. C++ Templates/Traits versus Haskell Type Classes. Technical Report TRB2/05, The National Univ. of Singapore, 2005.
 15. B. McNamara and Y. Smaragdakis. Functional Programming with the FC++Library. *Journal of Functional Programming*, 14(4):429–472, July 2004.
 16. G. D. Reis and B. Stroustrup. Specifying C++ Concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 295–308, New York, NY, USA, 2006. ACM Press.
 17. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 60–76, New York, NY, USA, 1989. ACM Press.

A Complete Translation Rules

```

paramTypeVars(typeVars, funDeps) =
  typeVars - assocTypeVars(funDeps)
assocTypeVars(funDeps) =
  rhs(funDeps) - lhs(funDeps)
rhs(funDeps) =
  join(map(pickRhs, funDeps))
lhs(funDeps) = as rhs...
pickLeft([leftTypeVars -> rightTypeVars]) =
  leftTypeVars
pickRight([leftTypeVars -> rightTypeVars]) = as pickLeft...
join(aSetOfSets) = as monadic join, removes one level of set structure...
map(fun, aSet) =
  {fun(el) | el ∈ aSet}
refinementClasses(assocTypeVars, superClasses) =
  superClasses - requirementClasses(assocTypeVars, superClasses)
requirementClasses(assocTypeVars, superClasses) =
  = join({mentionedInClasses(v, superClasses) | v ∈ assocTypeVars})
mentionedInClasses(typeVar, superClasses) =
  {c | c ∈ superClasses, typeVar ∈ mentionedIn(typeExpsOfClass(c))}
typeExpsOfClass([typeClassId typeExps]) =
  typeExps
mentionedInExps(typeExps) =
  join(map(mentionedInExp, typeExps))
mentionedInExp([typeVarId]) =
  {typeVarId}
mentionedInExp([concreteType]) =
  ∅
mentionedInExp([typeConstr]) =
  mentionedInTypeConstr(typeConstr)
mentionedInTypeConstr([typeConstrId typeExps]) =
  mentionedInExps(typeExps)

```

Fig. 3. Translation functions

```

reduce(ASTs, linearization, delimiter) =
  linearization(AST1) ++ delimiter ++ ... ++ delimiter ++ linearization(ASTn)
  where n = |ASTs|
typeParamDecl(typeVarId) =
  typeVarId
conceptInstance(typeClassId typeExps) =
  typeClassId ++ < ++ conceptArguments(typeExps) ++ >
conceptArguments(typeExps) =
  reduce(typeExps, conceptArgument, ‘,’)
conceptArgument(concreteType) =
  if concreteType is unit type () then void else undefined
conceptArgument(typeConstrId typeExps) =
  reduce(typeExps, conceptArgument, ‘,’)
conceptArgument(typeVarId) =
  typeVarId
assocTypeDecl(typeVarId) =
  typename typeVarId ;

```

Fig. 4. Linearization functions