KerMeta Type System and well-formedness rules

Franck Fleurey

KerMeta Type System and well-formedness rules
Franck Fleurey
Published Build date: 10-October-2005

Table of Contents

List of Figures

CHAPTER 1

Kermeta well-formedness rules

1.1. Opposite properties

Coherence of opposites

1.2. Redefinition

KerMeta does not support operation specialization but only invariant operation overriding.

1.3. Inheritance

KerMeta supports multiple inheritance. Name uniqueness.

CHAPTER 2

Overview of the type system

Figure 2.1.

TODO: Present the model

TODO: Briefly discuss function types.

CHAPTER 3

KerMeta sub-typing rules

3.1. Class sub-typing

Figure 3.1.

A class type C1can be a sub-type of some Class TypeVariable ProductType or SelfType

3.2. FunctionType sub-typing

ProductType sub-typing

DataType

TODO : Discuss subTyping here

CHAPTER 4

Type of KerMeta expressions

This section details the static computation of type of each kind of expression.

4.1. Literals

Figure 4.1.

Expression	Type
VoidLiteral	**result** := kermeta::standard::Void
BooleanLiteral	**result** := kermeta::standard::Boolean
IntegerLiteral	**result** := kermeta::standard::Integer
StringLiteral	**result** := kermeta::standard::String

TypeLiteral	**if** expression.typeref.upper == 1 **then**
	result := expression.typeref.type
	else
	if expression.typeref.unique **and** expression.typeref.ordered **then**
	result := OrderedSet<expression.typeref.type>
	end
	if expression.typeref.unique **and not** expression.typeref.ordered **then**
	result := Set<expression.typeref.type>
	end
	if not expression.typeref.unique **and** expression.typeref.ordered **then**

Expression	Type
	result := Sequence<expression.typeref.type> **end** **if not** expression.typeref.unique **and not** expression.typeref.ordered **then** **result** := Bag<expression.typeref.type> **end** end

4.2. Control Structures

Figure 4.2.

Loop expression

Constraints :

-
  -
    - *TypeOf(*Loop.stopCondition*)* <: kermeta::standard::Boolean

Returned Type :

In the literature 3 solution are defined for the return value of a loop expression:

-
  -
    - Nothing. This is the simplest solution.
    - The object returned by the expression “body” for the last iteration.
    - The collection of object returned by the expression “body” for all iterations.

Among these solution, and for the sake of simplicity, we have chosen to implement the first solution in KerMeta.

Expression	Type
Loop	**result** := kermeta::standard::Void

Block expression

A block is contains a set of statements and a set of rescue block. Each rescue block handles a particular type of exception. The type of an exception can only be a Class.

Constraints :

-
  -
    - Block.rescueBlock.exceptionType.type <: kermeta::structure::Class
    - Block.rescueBlock.exceptionType.upper = 1

Returned Type :

Two alternatives exists in the literature :

-
  -
    - Nothing. This is the simplest solution.
    - The object returned by the last statement of the block.

In KerMeta we have chosen the second solution because it is the choice made in the OCL. If a rescue block is executed, the block expression returns Void.

Expression	**Type**
Block	*TypeOf(* Block.statement.last *)*

Conditional expression

Constraints :

-
  -
    - *TypeOf(*Conditional.stopCondition*)* <: kermeta::standard::Boolean

Returned Type :

2 solutions

-
  -
    - Nothing. This is the simplest solution.
    - Support for union types

The first solution 1 has been rejected because it does not allow some very convenient OCL-like expressions.

Expression	Type
Conditional	*UNION (TypeOf(* Conditional.thenBody *), TypeOf(* Conditional.elseBody *))*

4.3. Variables

The scope of a variable is the block in which it is defined.

A variable can be initialized with an initialization expression. If there is no initialization expression then its value is initialized to *void*.

Figure 4.3.

Constraints :

-
  -
    - *TypeOf(*VariableDecl.initialization*)* <: *AsType(* VariableDecl.type *)*

Returned Type :

Expression	Type
VariableDecl	*AsType(* VariableDecl.type *)*

4.4. Call Expressions

Call expression allows reading / writing variable and properties but also calling operations, super-operations and lambda expressions.

Figure 4.4.

Call variable / Call result

A Call variable represents a call to an operation parameter, a interpreter variable or a local variable. In any case, the type of this variable might be either a simple type or a function type.

If it is a simple type, the returned value of the expression is the value of the variable: the type of the

expression is the type of the variable.

If the type of the variable is a function type then there are two possible behaviour depending on the presence of parameters. In no parameters are supplied then the returned value of the expression is the function but if parameter are provided, the function is applied and the value returned by the expression is the value returned by the function. In that case the return type of the expression is the return type of the function.

Constraints :

-
  -
    - *Presence of parameters => function types*
    - Number of parameters
    - Type of parameters

SelfExpression

The self expression allow accessing the current instance. The object returned by a self expression is the instance in the context of which the operation is executed. The type of this object can be any type associated with the class definition in which the operation is defined.

Thus the type of a self expression is obtained by binding all type variables of the class to the least derived compatible type. If the type variable has a superType then this superType is used, otherwise the type Object is used.

CallFeature

CallFeature expression allows calling operation and reading/writing features. If no target is specified the self is used.

Returned Type :

Expression	**Type**
CallVariable	XXX

4.5. Assignment

Figure 4.5.

4.6. Lambda Expression

Figure 4.6.

CHAPTER 5

Conclusion