

A User's Guide to the
KMT
Module and Type System

Jeffrey H. Kingston

Version 1.0 (Draft Only)
July 2008

Preface

This guide is a draft. KMT has not yet been released, nor used by the Nonpareil compiler.

This is the User's Guide to Version 1.0 of KMT, a C library for use in the front ends of compilers for statically typed object-oriented programming languages. KMT manages the hierarchical namespace of an object-oriented language, and it supplies the module and type operations needed by modern languages. KMT is released under the GNU General Public License.

KMT is motivated by a belief that programmers and compiler writers want languages with powerful and safe module and type systems, but do not have the time to become type systems experts. So compiler writers need to use off-the-shelf type systems in the same way that they use off-the-shelf lexing and parsing systems now. KMT aims to be such a system.

Accordingly, this guide does not assume that its readers are type systems experts. Familiarity with object-oriented languages, including generic classes and functions (templates in C++), is assumed. The type systems book by Pierce [5] is a good starting point for further study.

KMT is not likely to be useful in compilers for existing object-oriented languages, since they all have idiosyncratic features that KMT does not support. (Some features are unsupported for very good reasons: they conflict with other features considered more important.) However, language designers willing to be influenced by what KMT provides will find that there is enough here to support a very powerful, modern object-oriented language, including inheritance and import, access control, overloading and overriding, user-defined operators, renaming, multiple inheritance and meet types, generic modules, classes, and functions, function currying, and inference of missing actual generic parameters. Most features come with options, allowing many choices for their detailed behaviour, and it is always easy to elect not to use some features.

KMT was written by me between August 2007 and June 2008, starting from code I wrote earlier for the compiler of my Nonpareil language. I have since rewritten the Nonpareil compiler to use KMT, greatly simplifying it and providing a thorough test of KMT itself.

There are many ways to organize the inclusion of KMT into a compiler. One simple way is to make a subdirectory of the main compiler directory called `kmt` and place the KMT source files and makefile in there. Type `make` with no arguments to compile all the source files into object files. Ensure that these object files are linked into the main compiler binary, by adding `kmt/*.o` to the command that does the link. Add something like `-I kmt` to the compile command line to make the compiler search directory `kmt` for include files. Then place

```
#include "kmt.h"
```

at the start of any source files that require access to KMT's types and operations.

Backward compatibility will not be a high priority when releasing new versions of KMT; I want to be free to change it in the light of experience. If you are using KMT, I would love to hear from you. I am very willing to discuss changes that you may need to support your language, and if you can see a way to make KMT better, please let me know.

Jeffrey H. Kingston
jeff@it.usyd.edu.au

Contents

Chapter 1. Types	1
1.1. Introduction	1
1.2. Upcasting, downcasting, and traversal	4
1.3. Genus and <i>is_typic</i>	6
1.4. Evaluation and validity	7
1.5. Evaluation cycles	10
1.6. Lexical context and compilation order	13
1.7. Display	14
1.8. Equality, subtype, join, and meet	17
Chapter 2. Function types	20
2.1. Creation and query	20
2.2. Evaluation and validity	23
2.3. Display	24
2.4. Equality, subtype, join, and meet	25
Chapter 3. Parameter types	31
3.1. Creation and query	31
3.2. Display	31
3.3. Equality, subtype, join, and meet	32
Chapter 4. Call types and function matching	33
4.1. Creation and query	33
4.2. Evaluation	34
4.3. Display	35
4.4. Function matching	36
Chapter 5. Range types	44
5.1. The inference of actual generic parameters	44
5.2. Creation and query	45
5.3. Range policies and failure to infer	46
5.4. Confirming and cancelling changes	47
5.5. Evaluation and validity	49
5.6. Display	50
5.7. Equality, subtype, join, and meet	50

Chapter 6. Record types	52
6.1. Creation and query	52
6.2. Renaming	54
6.3. Evaluation and validity	56
6.4. Display	57
6.5. Equality, subtype, join, and meet	58
Chapter 7. Access types and retrieval	59
7.1. Creation and query	59
7.2. Evaluation	60
7.3. Display	61
7.4. Retrieval	61
7.5. Access control	69
7.6. Closure parameters	71
Chapter 8. Meet types	73
8.1. Creation and query	73
8.2. Evaluation and validity	74
8.3. Display	75
8.4. Equality, subtype, join, and meet	75
Chapter 9. Error types	77
9.1. Creation and query	77
9.2. Display	77
9.3. Equality, subtype, join, and meet	78
Chapter 10. Other features	79
10.1. A Boolean type	79
10.2. Memory allocation	79
10.3. Extensible arrays	80
10.4. String construction	81
10.5. Symbol tables	82
10.6. Behaviour on incorrect usage	84
10.7. Testing KMT	84
References	87
Index	88

Chapter 1. Types

1.1. Introduction

Formally, a type is a set of allowed values: the integer type is the set of all integers, for example. However, in practice types are represented by *type expressions*, such as *int* for the integer type, *util.list[int]* for the type of lists of integers taken from a *util* module, and so on, and these are what KMT is mainly concerned with.

Most languages contain module expressions, type expressions, and term expressions (ordinary expressions). KMT cannot hope to cover all the term expressions in all the programming languages of the world (conditional expressions, assignment statements, and so on). Instead, it offers operations, such as subtype testing, function matching, and retrieval, that the user can call on when checking term expressions. But type and module expressions are much less varied, and they are evaluated at compile time, so KMT takes complete control of those.

Familiar type expressions such as *util.list[int]*, which invoke existing modules and classes, are provided by KMT. What is perhaps less familiar (although standard in the type systems literature) is that the *definitions* of functions, classes, modules, and so on are also considered to be type expressions. In fact, the entire program (omitting its term expressions, which KMT does not handle directly) is a single large type expression.

A *type* in KMT is an object (a pointer to a private struct) of type `KMT_TYPE`, which in object-oriented terms is the abstract superclass of eight concrete subclasses:

<i>Kind of type</i>	<i>Type name</i>	<i>Syntax (not part of KMT)</i>
<i>Function type</i>	<code>KMT_FUNCTION_TYPE</code>	fun $[x_1: T_1, \dots, x_k: T_k](x_{k+1}: T_{k+1}, \dots, x_n: T_n): T$
<i>Parameter type</i>	<code>KMT_PARAMETER_TYPE</code>	x
<i>Call type</i>	<code>KMT_CALL_TYPE</code>	$T[T_1, \dots, T_k](T_{k+1}, \dots, T_n)$
<i>Range type</i>	<code>KMT_RANGE_TYPE</code>	$T..T$
<i>Record type</i>	<code>KMT_RECORD_TYPE</code>	inherit $R_1, \dots, R_m \{ u_1: T_1, \dots, u_n: T_n \}$
<i>Access type</i>	<code>KMT_ACCESS_TYPE</code>	$R.u$ (or just u)
<i>Meet type</i>	<code>KMT_MEET_TYPE</code>	R_1 meet ... meet R_n
<i>Error type</i>	<code>KMT_ERROR_TYPE</code>	?

The syntax is used as an aid in this guide, and it is defined formally in Section 10.7, but syntax is not part of KMT. Instead, type objects are created by the user by calling KMT functions while parsing whatever syntax the user prefers. Within the syntax, T stands for a type expression, and R for a type expression whose value must be a record type.

Each type object contains an *implementation pointer*, or just *implementation*, whose value is supplied by the user and not used by KMT. It is intended to point to whatever the type represents on the user side. For example, a type representing a class would contain an implementation pointer to the user-side representation of that class, and so on. In some cases, notably type expressions in the conventional sense (*list[int]* and so forth), the only information actually needed

on the user side may be a file position. This pointer may be accessed at any time, and also reset (although that would be unusual), by calling

```
void *KmtTypeImpl(KMT_TYPE type);
void KmtTypeSetImpl(KMT_TYPE type, void *impl);
```

Each concrete kind of type has its own version of these two functions.

For each of the eight kinds of types there is a chapter of this guide, explaining it in detail. The remainder of this section is an overview of the most important kinds.

Function types hold the formal parameter lists of functions and classes. Each formal parameter may be either *generic* (stand for a type) or *ordinary* (stand for a term). Generic parameters allow function types to represent *universal types*, in the terminology of the literature. The syntax shows generic formal parameters enclosed in brackets followed by ordinary formal parameters enclosed in parentheses, in the familiar way; but in actual use, each function type has a single sequence of formal parameters, each of which may be generic or not independently of the others. The (optional) type associated with each formal parameter is called its *upper constraint*. The concluding type is compulsory and is called the *result type* of the function type.

Parameter types invoke the formal parameters of lexically enclosing function types. A parameter type may appear within the upper constraints of the formal parameters of its formal parameter's function type, providing what the literature calls 'F-bounded polymorphism', as well as within the result type.

Call types are applications of actual parameters to function types. The initial type T (the *head*) must evaluate to a function type. The other types stand for zero or more *actual parameters*; generic actual parameters in brackets, and ordinary actual parameters in parentheses. (Strictly speaking, no call type has ordinary actual parameters, but the part of KMT that deals with call types may also be used to deal with call *expressions*, which do.) The value of a call type is the result type of its head, with invocations of the head's formal generic parameters replaced by the corresponding actual parameters. The actual parameters must satisfy the upper constraints of the corresponding formal parameters, where present.

Care is needed in distinguishing between ordinary functions, which KMT does not handle directly, and function types, which it does. Each ordinary function has an associated function type. For example, the ordinary call $\text{sqrt}(2)$ is evaluated at run time and not handled by KMT, but KMT can evaluate $(\text{fun}(x: \text{real}): \text{real})(\text{int})$ at compile time, to check that the ordinary call is well typed. Here $\text{fun}(x: \text{real}): \text{real}$ is the type of the sqrt function (it takes one real-valued parameter and returns a real result), and int is the type of the term expression 2.

Record types represent sets of named things. The notation ' $u_1: T_1, \dots, u_n: T_n$ ' stands for zero or more *components* of the record, each consisting of a *name* (a string) and a *value* (a type). The components could represent classes within a module, or fields and methods within a class, and so on. Components may optionally be inherited into a record type from any number of other record types, as usual in object-oriented languages.

Access types select record components by name; the value of an access type is the selected component's value. Anywhere within the record type itself, its components may be selected without any preceding 'R.'

As an example of these type expressions in action, consider the class

```

class nlist[x] inherit list[x]
  head: x
  tail: list[x]
end

```

representing non-empty generic lists. As a KMT type expression, this is

```

{
  nlist: fun[x]: inherit list[x] {
    head: x
    tail: list[x]
  }
}

```

Notice how the class breaks into three pieces: its name, which is a component name in an enclosing record type; its parameters, which produce a function type; and its inherit clause and components, which go into an inner record type which is the result type of the function type.

Type expressions get *evaluated*, including checking that they are *valid* and working out what they stand for. Continuing the example above, the type expression *nlist*[*int*] is evaluated by evaluating *int* (whose value is a record type, not shown), evaluating *nlist*, whose value is

```

fun[x]: inherit list[x] {
  head: x
  tail: list[x]
}

```

and then applying *int* to this function type to get

```

inherit list[int] {
  head: int
  tail: list[int]
}

```

as the result. This last step might seem to involve an expensive record type copy operation, but in fact no record type is ever copied. Instead, the result is represented by a reference to the original record type, plus some substitutions (in the example, *int* for *x*). These substitutions are applied to anything subsequently extracted from the record type, simulating the expensive full copy.

Two record types are considered to be equal when they refer to the same original record type and have equal substitutions. The literature calls this *nominal typing*. The alternative, *structural typing*, would require KMT to make the substitutions and compare the resulting record types component by component, which is more intuitive, but less efficient. The two approaches do not always produce the same result, for several reasons, notably the two given now.

First, record types whose original record types were different can be equal using structural typing, but they are never equal using nominal typing. This can be assimilated into the intuitive view by imagining that each record type contains an *identity*, a unique number assigned to it by the system when it is first declared, and copied when it is (notionally) copied, and defining equality of records to include equality of their identities.

Second, the substitutions tested by nominal typing affect the outcome even if the parameters they replace are never used. For example, suppose a second parameter *y* was added to the *nlist* class above, but not used within it. Then *nlist[int, int]* and *nlist[int, real]* would be equal using structural typing, but unequal using nominal typing.

KMT uses nominal typing only for record types; in all other cases it uses structural typing. For example, two function types are equal if they have the same formal parameters, with equal upper constraints and equal result types. Peculiar as it may seem to mix the two kinds of typing in this way, all mainstream object-oriented languages do it.

Many issues lurk below the surface of this system. For example, an access type may occur earlier in a type expression than the component it accesses, so type expressions cannot be evaluated in a single pass. The type expression *list[util]* invokes a module where a type is required, showing that not all type expressions are acceptable in all contexts. Term expressions, though not handled directly, need support. Retrieving a component raises the familiar issues of overloading, overriding, and access control. KMT is based on simple, pure principles, but it is not an ivory-tower system. It tackles these issues and many others, and offers many options for handling them in detail.

1.2. Upcasting, downcasting, and traversal

Upcasting and downcasting between `KMT_TYPE` and its eight concrete subtypes, using C type casts, is a normal part of using KMT. The eight subtypes are all represented by pointers to private structs, each of which begins with a tag field, of type `KMT_TAG`, defined by

```
typedef enum {
    KMT_FUNCTION_TYPE_TAG,
    KMT_PARAMETER_TYPE_TAG,
    KMT_CALL_TYPE_TAG,
    KMT_RANGE_TYPE_TAG,
    KMT_RECORD_TYPE_TAG,
    KMT_ACCESS_TYPE_TAG,
    KMT_MEET_TYPE_TAG,
    KMT_ERROR_TYPE_TAG,
    KMT_TRANSIENT_TYPE_TAG,
    KMT_FORMAL_PARAMETER_TAG,
    KMT_COMPONENT_TAG,
    KMT_PARENT_TAG
} KMT_TAG;
```

Tag value `KMT_TRANSIENT_TYPE_TAG` tags *transient types*, for which see Section 1.4. The last three tag values do not tag type objects; they are included because the objects they tag share abstract supertypes with type objects. For example, `KMT_COMPONENT` shares abstract supertype `KMT_CONTEXT` with `KMT_FUNCTION_TYPE` and `KMT_RECORD_TYPE` (Section 1.6).

The tag field of a type object may be retrieved by

```
KMT_TAG KmtTypeTag(KMT_TYPE type);
```

Any tag value may be displayed by

```
wchar_t *KmtTagShow(KMT_TAG tag);
```

Types themselves may also be displayed (Section 1.7).

One application of downcasting is in traversing a type. The pattern for this is

```
void DoItForType(KMT_TYPE type)
{
    switch( KmtTypeTag(type) )
    {
        case KMT_FUNCTION_TYPE_TAG:

            DoItForFunctionType((KMT_FUNCTION_TYPE) type);
            break;

        case KMT_PARAMETER_TYPE_TAG:

            DoItForParameterType((KMT_PARAMETER_TYPE) type);
            break;

        case KMT_RECORD_TYPE_TAG:

            DoItForRecordType((KMT_RECORD_TYPE) type);
            break;

        case KMT_MEET_TYPE_TAG:

            DoItForMeetType((KMT_MEET_TYPE) type);
            break;

        case KMT_RANGE_TYPE_TAG:

            assert(KmtRangeTypeResolvedTo((KMT_RANGE_TYPE) type) != NULL);
            DoItForType(KmtRangeTypeResolvedTo((KMT_RANGE_TYPE) type));
            break;

        case KMT_CALL_TYPE_TAG:
        case KMT_ACCESS_TYPE_TAG:
        case KMT_ERROR_TYPE_TAG:
        default:

            assert(KMT_FALSE);
    }
}
```

If traversal is being done to generate reflection code, by then each range type should be resolved (Section 5.4), and must be treated as the type it is resolved to, as shown. The existence of even one error type should have caused compilation to end before code generation, and call, access, and transient types should have been removed by evaluation (Section 1.4); so, although these

types can be visited like the others, they should not be encountered during code generation.

Although C type casts may be used safely for upcasting and downcasting, they may be used for other, unsafe things as well, making them somewhat error-prone. The user may prefer to avoid C casts altogether by using these operations for upcasting:

```
KMT_TYPE KmtFunctionTypeToType(KMT_FUNCTION_TYPE function_type);
KMT_TYPE KmtParameterTypeToType(KMT_PARAMETER_TYPE param_type);
KMT_TYPE KmtCallTypeToType(KMT_CALL_TYPE call_type);
KMT_TYPE KmtRangeTypeToType(KMT_RANGE_TYPE range_type);
KMT_TYPE KmtRecordTypeToType(KMT_RECORD_TYPE record_type);
KMT_TYPE KmtAccessTypeToType(KMT_ACCESS_TYPE access_type);
KMT_TYPE KmtMeetTypeToType(KMT_MEET_TYPE meet_type);
KMT_TYPE KmtErrorTypeToType(KMT_ERROR_TYPE error_type);
KMT_TYPE KmtTransientTypeToType(KMT_TRANSIENT_TYPE transient_type);
```

and these for downcasting:

```
KMT_FUNCTION_TYPE KmtTypeToFunctionType(KMT_TYPE type);
KMT_PARAMETER_TYPE KmtTypeToParameterType(KMT_TYPE type);
KMT_CALL_TYPE KmtTypeToCallType(KMT_TYPE type);
KMT_RANGE_TYPE KmtTypeToRangeType(KMT_TYPE type);
KMT_RECORD_TYPE KmtTypeToRecordType(KMT_TYPE type);
KMT_ACCESS_TYPE KmtTypeToAccessType(KMT_TYPE type);
KMT_MEET_TYPE KmtTypeToMeetType(KMT_TYPE type);
KMT_ERROR_TYPE KmtTypeToErrorType(KMT_TYPE type);
KMT_TRANSIENT_TYPE KmtTypeToTransientType(KMT_TYPE type);
```

Each applies the appropriate cast, with the downcasting operations first checking that it is safe to do so, and aborting if not. If C type casts are avoided altogether, the result will be a dynamically type-safe program.

1.3. Genus and *is_tpic*

KMT takes upon itself the task of ensuring that type expressions are valid, and so it must have some way of detecting problems in expressions such as *list[util]* and *list[sqrt]*. In these examples, a module or function name appears where a type name is expected.

One way to rule out *list[util]* at least is to have a top type (the usual *object*) for types and a top type for modules, and in places where formal parameters have no upper constraints, to insert one of these top types. Then *list[util]* has a type error, since *util* is not a subtype of *object*. This has the advantage of requiring nothing new, but it has disadvantages too: subtype constraints have to be added everywhere a type occurs, and top types can be very artificial.

Instead, in KMT, all evaluated types except error types have a *genus*, a small integer which classifies the type into a broad category. The user is free to define up to eight genera. Two obvious ones to have are *module* and *type*. A third, *type_fn*, is useful for function types, so that cases like *list[list]*, where a function type appears as a parameter, can be detected as genus errors. Each function type, formal parameter, and record type is declared to have a particular genus,

based on syntactic cues – **class** and **module** keywords, and so on, the other kinds of evaluated type deriving their genera from these ones. At each point where a type occurs, the user may state that a type of a particular genus, or one of a set of genera, is expected. The user must use powers of 2 as the genus values, to allow sets of genera to be defined, like this for a hypothetical VX language:

```
typedef enum {
    VX_GENUS_MODULE   = 1,
    VX_GENUS_TYPE     = 2,
    VX_GENUS_TYPE_FN  = 4
} VX_GENUS;
```

For example, in *R,u* the genus of *R* could be *module* or *type*, and the user would give

```
VX_GENUS_MODULE | VX_GENUS_TYPE
```

as the expected genus set in that case.

The genus of an evaluated, non-error type may be found by calling

```
char KmtTypeGenus(KMT_TYPE type);
```

This function will abort if *type* is not an evaluated non-error type.

Genus checking detects many problems, including inheriting a module rather than a class, forming the meet of a module with a class, and so on. There does remain one kind of error, however, which does not seem to be handled well by either genera or top types. Consider

```
emptylist: fun[x]: list[x] := ...
type mylist: fun[x]: list[x]
```

where *emptylist* is a function and *mylist* is a type definition which introduces a new name for *list*. There is no difference in genus between these two definitions, and yet *mylist[int]* (say) may be used in type expressions and *emptylist[int]* may not. This problem cannot be solved by any method which first evaluates the type expression and then examines the result, which is how subtype and genus checking work, because the result is *list[x]* in both cases.

To handle this, each formal parameter and record component has a Boolean attribute, *is_typic*, which is *KMT_TRUE* if it may appear in type expressions, and *KMT_FALSE* otherwise. The syntax indicates that a formal parameter is typic (that is, generic) by enclosing it in brackets rather than parentheses; a component is indicated to be typic by preceding it with the **type** keyword, although it is often omitted in obvious cases. There is something unsatisfactory about *is_typic*, but it is simple and it seems to work.

1.4. Evaluation and validity

A type expression may be in one of two states: *raw* or *evaluated*. The user's parser produces raw types, then *KMT* evaluates them, including determining what the names within the types refer to, checking that upper constraints are not violated, and so on. The resulting evaluated types have a known meaning, they are *valid* (have no problems), and they are able to participate in further

operations, such as equality and subtype testing, which are not applicable to raw types.

Two of the eight kinds of types exist only in the raw state: call types and access types. This is because, just as evaluation of term expressions such as `sqrt(2)` removes all call expressions, leaving only a value, so evaluation of type expressions removes all call and access types. And two of the eight kinds exist only in the evaluated state: parameter types and error types. There are no raw parameter types because it is usual for parameter types to be syntactically indistinguishable from access types that have no ‘R.’, and a parser that had to choose between the two would be seriously embarrassed. Instead, a raw access type is created in both cases, and the ambiguity is cleared up during evaluation. There are no raw error types because it turns out that there is no need for them. The other four kinds of types can be either raw or evaluated.

The state of a type may be tested by calling

```
KMT_BOOLEAN KmtTypeIsEvaluated(KMT_TYPE type);
```

but this will probably never be needed, because the answer can be predicted from where the compiler is up to. The operation

```
void KmtTypeEvaluate(KMT_TYPE *type, KMT_CONTEXT context,
    KMT_EVALUATION_POLICIES policies);
```

evaluates `*type`, replacing it by its value after evaluation. The `context` parameter defines the lexical context in which the names within `*type` are interpreted, and is the subject of Section 1.6. The `policies` parameter defines what to do when errors occur, and is detailed below.

`KmtTypeEvaluate` replaces `*type`, rather than returning its value, as a reminder that the type’s raw state is considered to be of no further interest once it has been evaluated. In some cases the raw type object becomes unlinked from the type structure; in others, evaluation may choose to save time and memory by mutating it in place into the corresponding evaluated type, or into an error type. Precise rules for each kind of type are given in the relevant chapters.

A raw type may contain evaluated elements; these are skipped over during evaluation. In other words, `KmtTypeEvaluate` does nothing when `*type` is already evaluated. This may be convenient when including types that do not appear explicitly in the program text, such as the Boolean type of an invariant, since it may be simpler to generate them in evaluated form. It also allows raw types to be linked into the surrounding type structure at two or more points, provided the lexical context is the same at each point. The first evaluation will either produce a fresh type, leaving the raw type unharmed, or else it will mutate the raw type in place. Either way, the second evaluation will succeed. For example, if a method is defined with two names, it is safe to add two components to the enclosing record type that share the method’s type for their value.

For convenience in situations where types are optional, `*type` may be `NULL`, in which case `KmtTypeEvaluate` also does nothing. There are several places like this in KMT where a type is optional, and the value `NULL` is used to indicate that no type is being supplied. However, `NULL` itself is not considered to be a type, and KMT will abort when given `NULL` in places where a type is not specifically said to be optional.

An evaluated type must be *valid*: it must satisfy certain conditions, called its *validity rules*. Specific rules for each kind of type are given in the relevant chapters, and there is one general rule: an evaluated type may not contain raw elements. Evaluated types are always valid because invalid ones cannot be created. If an attempt is made to create one directly, KMT aborts; if eval-

uation of a raw type would produce an invalid result, an error callback is generated, as explained below, and the result becomes an error type, which (paradoxically, perhaps) is always valid.

A raw type may be said to be valid if it generates no error callbacks during evaluation. However, the concept of validity is applied in this guide only to evaluated types.

The `policies` parameter of `KmtTypeEvaluate` contains a large collection of user-supplied callback functions that `KmtTypeEvaluate` calls when it finds an error. These calls allow the user to record the fact that an error has occurred, and to print an error message. They offer no control over error recovery; in all cases, after the callback returns, the problem type is replaced by an error type (Chapter 9), allowing type checking to continue in a reasonable manner. A `policies` object, of type `KMT_EVALUATION_POLICIES`, may be created by calling

```
KMT_EVALUATION_POLICIES KmtEvaluationPoliciesMake(
    KMT_EVALUATIONPATH_FUN          evaluation_cycle,
    KMT_FUNCTIONTYPE_INT_FUN        function_upper_wrong_genus,
    KMT_FUNCTIONTYPE_INT_FUN        function_default_wrong_genus,
    KMT_FUNCTIONTYPE_INT_FUN        function_default_wrong_type,
    KMT_FUNCTIONTYPE_FUN            function_result_wrong_genus,
    KMT_CALLTYPE_FUN                call_head_not_function,
    KMT_CALLTYPE_FUN                call_unmatched_parameters,
    KMT_CALLTYPE_INT_FUN            call_parameter_wrong_genus,
    KMT_CALLTYPE_INT_FUN            call_parameter_wrong_type,
    KMT_RANGETYPE_FUN               range_lower_wrong_genus,
    KMT_RANGETYPE_FUN               range_upper_wrong_genus,
    KMT_RANGETYPE_FUN               range_inconsistent,
    KMT_PARENT_FUN                  record_parent_not_record,
    KMT_PARENT_FUN                  record_parent_wrong_genus,
    KMT_RECORDTYPE3_FUN             record_ancestors_inconsistent,
    KMT_COMPONENT_FUN               record_component_wrong_genus,
    KMT_ACCESSTYPE_FUN              access_head_not_record,
    KMT_ACCESSTYPE_FUN              access_head_wrong_genus,
    KMT_ACCESSTYPE_FUN              access_name_unknown,
    KMT_ACCESSTYPE_NAMEDENTITY2_FUN access_name_overload,
    KMT_MEETTYPE_INT_FUN            meet_member_not_record,
    KMT_MEETTYPE_INT_FUN            meet_member_wrong_genus,
    KMT_MEETTYPE_RECORDTYPE2_FUN    meet_inconsistent,
    KMT_MEETTYPE_TYPE_FUN           meet_redundant
);
```

It is usual to call `KmtEvaluationPoliciesMake` just once, and share the resulting object among all calls to `KmtTypeEvaluate`. The parameters, which are pointers to callback functions, are described in detail in the relevant chapters, except for `evaluation_cycle`, which is the subject of Section 1.5.

As an aid to getting started, each callback function has a default value, obtained by passing `NULL`. These default values print rudimentary one-line English error messages onto `stderr` in wide characters, beginning with `L"KMT: "`, and including displays of the types involved. The `policies` parameter may itself be `NULL`, and then these default values will be used for all

callback functions. Ultimately, however, users have to write their own functions, if only to ensure that a file position is attached to each error message.

Evaluation is the most difficult operation of all, and behind the scenes it has to be carried out in stages. The remainder of this section explains some issues arising from this.

Within the bodies of the callback functions, it is safe to use the usual operations to query and display the types passed as parameters; but some of these types may be halfway between raw and evaluated, and some may be evaluated but not valid, so they should not be used in other operations, or preserved on the user side after the callback returns. Some may be *transient types*, with tag `KMT_TRANSIENT_TYPE_TAG`. In principle, these could appear anywhere during evaluation, but in the current implementation only the upper constraints of some formal parameters ever have a transient type for their value, and even then only while the enclosing function type is evaluated but they are not. By the end of evaluation, all types reachable from the updated `*type` parameter of the call to `KmtTypeEvaluate` will be non-transient, evaluated, valid types.

Anything potentially involving record type ancestors, including all subtype testing, is deferred until just before `KmtTypeEvaluate` returns. Some callbacks therefore occur later than expected, namely `function_default_wrong_type`, `call_parameter_wrong_type`, `range_inconsistent`, `record_ancestors_inconsistent`, `meet_inconsistent`, and `meet_redundant`, with surprising consequences.

For example, when a record type has inconsistent ancestors, as in

```
inherit list[int], list[real] { ... }
```

a call to `record_ancestors_inconsistent` will occur eventually, but before then retrievals in the record type may have searched the inconsistent list of parents.

When a call type has a parameter of the wrong type, this fact is discovered long after the call type's result has been computed and used, too late to set it to an error type, which would be preferable. For example, casual inspection of

```
sorted_list: fun[x: comparable[x]]: { ... }
```

might suggest that if people are not comparable then no evaluated type `sorted_list[person]` can exist. In fact, when this type expression is evaluated, the result is a valid record type (the validity rules for record types do not include any requirement that constraints in surrounding function types must be obeyed); callback `call_parameter_wrong_type` will occur eventually, but too late to recall the record type.

1.5. Evaluation cycles

Evaluation may *cycle*, which means that in the course of evaluating some type it becomes necessary to evaluate that same type again. Evaluation cycles always contain at least one access type, but otherwise they vary widely, from trivial examples such as the type definition

```
type mylist: mylist
```

to highly non-trivial ones such as

```

A: inherit BX { ... }
B: inherit A.Y { ... }

```

To evaluate *BX* it is necessary to retrieve *X* from *B*. Since *X* could be inherited into *B* from *A.Y*, this retrieval may involve evaluating *A.Y*. To evaluate *A.Y* it is necessary to retrieve *Y* from *A*. Since *Y* could be inherited into *A* from *BX*, this may involve evaluating *BX*, giving a cycle. This is not an inheritance cycle; those are legal. Java avoids cycling for modules by requiring that ‘canonical’ (uninherited) names be used when importing, but it seems to be legal when inheriting classes. One compiler (an early `gcj`) crashes on the Java translation of this example.

KMT detects all evaluation cycles, and reports them by calling the `evaluation_cycle` callback function. In the syntax that the user would use to define this function, it is

```
void evaluation_cycle(KMT_EVALUATION_PATH ep);
```

where `ep` gives access to the path that the cyclic evaluation took. After the callback returns, all types on the cycle are replaced by error types, and evaluation continues.

Type `KMT_EVALUATION_PATH` has three operations:

```

KMT_EVALUATION_NODE KmtEvaluationPathNode(KMT_EVALUATION_PATH ep);
KMT_EVALUATION_PATH KmtEvaluationPathPrev(KMT_EVALUATION_PATH ep);
KMT_BOOLEAN KmtEvaluationPathOnCycle(KMT_EVALUATION_PATH ep);

```

`KmtEvaluationPathNode` returns one node of the path. `KmtEvaluationPathPrev` returns the previous part of the path; calling it repeatedly until `ep` becomes `NULL` traces back through the sequence of evaluations that led to the cycle. This is only possible within the error callback function itself, because for efficiency the evaluation path is held in stack memory, and lost after the callback returns. `KmtEvaluationPathOnCycle` returns `KMT_TRUE` if `ep`’s node is a part of the cycle. For example, the loop

```

while( ep != NULL && KmtEvaluationPathOnCycle(ep) )
{
    visit(KmtEvaluationPathNode(ep));
    ep = KmtEvaluationPathPrev(ep);
}

```

visits each node on the cyclic part of the path. Its body is executed at least two times, and the first and last time have equal values of `KmtEvaluationPathNode(ep)`, closing the cycle.

`KMT_EVALUATION_NODE` is an abstract superclass of all kinds of types and other objects that may lie on an evaluation cycle:

```

KMT_EVALUATION_NODE
    KMT_FUNCTION_TYPE
    KMT_CALL_TYPE
    KMT_RANGE_TYPE
    KMT_ACCESS_TYPE
    KMT_MEET_TYPE
    KMT_COMPONENT
    KMT_PARENT

```

Its tag field may be retrieved by calling

```
KMT_TAG KmtEvaluationNodeTag(KMT_EVALUATION_NODE evaluation_node);
```

Upcasting and downcasting may be done with C casts as usual, and there are also explicit upcasting operations:

```
KMT_EVALUATION_NODE KmtFunctionTypeToEvaluationNode(
    KMT_FUNCTION_TYPE function_type);
KMT_EVALUATION_NODE KmtCallTypeToEvaluationNode(KMT_CALL_TYPE call_type);
KMT_EVALUATION_NODE KmtRangeTypeToEvaluationNode(KMT_RANGE_TYPE range_type);
KMT_EVALUATION_NODE KmtAccessTypeToEvaluationNode(
    KMT_ACCESS_TYPE access_type);
KMT_EVALUATION_NODE KmtMeetTypeToEvaluationNode(KMT_MEET_TYPE meet_type);
KMT_EVALUATION_NODE KmtComponentToEvaluationNode(KMT_COMPONENT component);
KMT_EVALUATION_NODE KmtParentToEvaluationNode(KMT_PARENT parent);
```

and downcasting operations:

```
KMT_FUNCTION_TYPE KmtEvaluationNodeToFunctionType(KMT_EVALUATION_NODE en);
KMT_CALL_TYPE KmtEvaluationNodeToCallType(KMT_EVALUATION_NODE en);
KMT_RANGE_TYPE KmtEvaluationNodeToRangeType(KMT_EVALUATION_NODE en);
KMT_ACCESS_TYPE KmtEvaluationNodeToAccessType(KMT_EVALUATION_NODE en);
KMT_MEET_TYPE KmtEvaluationNodeToMeetType(KMT_EVALUATION_NODE en);
KMT_COMPONENT KmtEvaluationNodeToComponent(KMT_EVALUATION_NODE en);
KMT_PARENT KmtEvaluationNodeToParent(KMT_EVALUATION_NODE en);
```

These work in the same way as the casting operations for types (Section 1.2), with the downcast operations checking that the downcast is safe and aborting if not.

Predicting evaluation cycles can be puzzling. The evaluation of a type includes the evaluation of all its parts, with two exceptions. First, to evaluate a record type, KMT makes an object containing a pointer to the original record plus some substitutions (Section 1.1), without evaluating anything else, so a record type ends its path and cannot lie on a cycle. Its parent types and component values are evaluated later. Second, the upper constraints (but not the default types or result type) of a function type are also evaluated later than the function type itself; they may be represented by transient types (Section 1.4) during the interval between the evaluation of the function type and their own evaluation.

Here are some examples of these rules. Record types, even recursive ones such as

```
list: fun[x]: {
    head: x
    tail: list[x]
}
```

never lead to evaluation cycles. Nor does

```
list: fun[x: list]: x
```

cycle, because evaluation of the upper constraint of *x* is delayed. A non-typic parameter's upper

constraint can never lie on a cycle. On the other hand,

```
list: fun[x]: list[x]
```

does cycle, as it must: the evaluation of the value of component *list* evaluates the function type, which evaluates its result type, which attempts to evaluate *list*, causing a cycle. And

```
list: fun[x: list]: x.anything
```

also cycles: the evaluation of the value of component *list* evaluates the function type, which evaluates its result type *x.anything*, causing a retrieval in the type of *x*, which in turn causes *x*'s upper constraint to be evaluated, which attempts to evaluate *list*, causing a cycle as before.

1.6. Lexical context and compilation order

A *lexical context* defines how names are interpreted when a raw type is evaluated (Section 1.4). The pure way to handle lexical contexts is to have a separate stack which grows as evaluation enters contexts and shrinks as it leaves them, ultimately disappearing without trace. However, KMT handles them in a different way, which will be called the *stored contexts* method. When evaluation enters a function type, record type, or record component, the current value of the `context` parameter is stored in that object, and that object becomes the context for interpretation of the types lying within it. In this way the stack of lexical contexts is held in the type objects themselves. It is never deleted, so stored contexts are accessible not just to raw types *during* evaluation, as with the pure method, but also to their evaluated versions *after* evaluation.

If a type expression is evaluated within a context which is a function type, the function type's parameters are visible, as well as anything else in scope within that function type. If the context is a record type, the record type's components (including inherited components) are visible, as well as anything else in scope within that record type. If the context is a component of a record type, the type expression is assumed to lie within that component. This is the same as lying within the enclosing record type, except for a slight difference to access control (Section 7.5). Alternatively, a context may be `NULL`, meaning that nothing is in scope.

Stored contexts have been preferred for three reasons. First, they are helpful when displaying record types (Section 6.4). Second, they make it easy to evaluate deeply nested types long after their contexts are evaluated. For example, the type expressions and other symbols scattered through the body of a function can be evaluated long after the enclosing context has been evaluated, by supplying the enclosing function type as their context. Third, behind the scenes it is sometimes necessary to interrupt the evaluation of one type in order to evaluate another, and stored contexts make it easy to switch from one context to another and back again.

A lexical context has type `KMT_CONTEXT`, an abstract supertype of three subtypes:

```
KMT_CONTEXT
  KMT_FUNCTION_TYPE
  KMT_RECORD_TYPE
  KMT_COMPONENT
```

representing a function type, a record type, and one component of a record type (not itself a type). There is an operation for finding the tag field of a context object:

```
KMT_TAG KmtContextTag(KMT_CONTEXT context);
```

Upcasting and downcasting may be done using C casts, or by using these upcasting operations:

```
KMT_CONTEXT KmtFunctionTypeToContext(KMT_FUNCTION_TYPE function_type);
KMT_CONTEXT KmtRecordTypeToContext(KMT_RECORD_TYPE record_type);
KMT_CONTEXT KmtComponentToContext(KMT_COMPONENT component);
```

and these downcasting operations:

```
KMT_FUNCTION_TYPE KmtContextToFunctionType(KMT_CONTEXT context);
KMT_RECORD_TYPE KmtContextToRecordType(KMT_CONTEXT context);
KMT_COMPONENT KmtContextToComponent(KMT_CONTEXT context);
```

These work in the same way as the casting operations for types (Section 1.2), with the downcast operations checking that the downcast is safe and aborting if not.

Finding a suitable ordering of the steps needed to compile an object-oriented program can be perplexing. `KmtTypeEvaluate` solves that problem, as the following examples show. Some care is needed, however, because the `context` parameter passed to `KmtTypeEvaluate` is permitted to give access only to *evaluated* types. This rule affects how `KmtTypeEvaluate` can be used, and how the types of an object-oriented program can be evaluated.

One strategy is to parse the whole program, producing one enormous raw type, which would typically be a record type with one component for each module, then evaluate that type in a `NULL` context. This checks all module, class, field, and method interfaces in one king hit, and allows all of them to refer to all of the others. The same approach can be used to compile a single module, using only the *interfaces* of other modules (obtained from C `.h` files, Java `.class` files, etc.). KMT does not know or care that all but one of the modules is a mere interface.

Alternatively, the programming language might specify an ordering of modules such that earlier modules do not refer to later ones, and it is desired to compile the modules one by one in the specified order. In that case, an empty root record type is created and evaluated in a `NULL` context, then each module in order is parsed, evaluated in the context of the root record type, then added to that record type as one of its components.

Function bodies are usually evaluated last. They are term expressions, and since KMT does not deal directly with those, it is not possible to interpret the type expressions and variables scattered through them in a single call to `KmtTypeEvaluate`. An efficient strategy here is to create an empty record type to hold the local definitions, and evaluate it in the context of the enclosing function type, which sets its context to that function type. Then, as each local definition comes into scope, it is evaluated in the context of this record type then added to it, and as each local definition goes out of scope, it is deleted from the record type.

1.7. Display

KMT offers an operation for displaying type expressions, flexible enough to produce the usual range of syntax seen in type expressions. If something else is needed, then the user will have to write a display function, following the template for traversing types given in Section 1.2.

An arbitrary type may be displayed by calling

```
wchar_t *KmtTypeShow(KMT_TYPE type);
```

The result is held in memory from the current memory arena (Section 10.2). For convenience there are also functions for displaying each concrete kind of type, documented in the chapters for those types. These merely upcast their parameter to `KMT_TYPE` and call `KmtTypeShow`.

The format used when displaying types is held privately within KMT in an object of type `KMT_TYPE_FMT` (a pointer to a private struct). It may be retrieved and reset by

```
KMT_TYPE_FMT KmtTypeFmtGet(void);
void KmtTypeFmtSet(KMT_TYPE_FMT fmt);
```

`KmtTypeFmtGet` always returns a non-NULL value; if no format has been assigned by a previous call to `KmtTypeFmtSet`, then the default format defined below will be returned. The `fmt` parameter of `KmtTypeFmtSet` must be non-NULL, or else the function will abort.

An object of type `KMT_TYPE_FMT` may be obtained by calling

```
KMT_TYPE_FMT KmtTypeFmtMake(wchar_t *overall_fmt, wchar_t *function_fmt,
    wchar_t *function_formal_list_fmt, wchar_t *function_formal_fmt,
    wchar_t *parameter_fmt, wchar_t *call_fmt, wchar_t *call_actual_list_fmt,
    wchar_t *call_actual_fmt, wchar_t *range_fmt, wchar_t *access_fmt,
    wchar_t *meet_fmt, wchar_t *error_fmt, wchar_t *transient_fmt);
```

Detailed descriptions of these *format strings* appear in the relevant chapters, except that `overall_fmt` and `transient_fmt` are described as part of the general introduction to format strings given below. They may be retrieved by

```
wchar_t *KmtTypeOverallFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeFunctionFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeFunctionFormalListFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeFunctionFormalFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeParameterFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeCallFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeCallActualListFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeCallActualFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeAccessFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeMeetFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeRangeFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeErrorFmt(KMT_TYPE_FMT fmt);
wchar_t *KmtTypeTransientFmt(KMT_TYPE_FMT fmt);
```

Alternatively, a default format may be created, by calling

```
KMT_TYPE_FMT KmtTypeFmtDefault();
```

This returns the format used by the author's Nonpareil language.

The format strings are modelled on C `wprintf` strings. They are displayed literally except for formatting commands consisting of a `%` character and the next character. For example, `overall_fmt` gives the format of the display overall; within it, `%T` displays the type expression.

The usual value of `overall_fmt` would be just

```
L"%T"
```

but if, say, type expressions always begin with `!`, then the appropriate format would be `L"!%T"`. To display a literal `%` character, use `L"%%"`.

The `transient_fmt` format string gives the format to use when displaying transient types. These occur only within the types returned by error callbacks during evaluation (Section 1.4). Their internal structure is not accessible, so `transient_fmt` takes no formatting commands. A reasonable value for it would be `L"..."`, indicating that something is being elided.

When there is a sequence of things to display, the arrangement is

```
L"%{ ... % | ... % | ... % | ... % }"
```

The first ellipsis stands for what to display if the sequence is empty, the second for how to display the first element if there is one, the third for how to display elements after the first, if there are any, and the fourth for what to display after the last element if there is one. For example, to display a sequence of actual parameters in C style, one would use

```
L"(%{|%P%|, %P%|%})"
```

This always displays the enclosing parentheses, displays nothing else if the sequence is empty, and otherwise displays each parameter, with parameters after the first preceded by a comma and space. Alternatively, `L"%{|(%P%|, %P%|)%}"` produces Pascal style. The sequence format may be used only in formats where there is a sequence of things to be printed. In every such case there is exactly *one* sequence of things to be printed, and it is the length of that one sequence which determines which parts of the format are used. There is never any use for nesting one of these formats within another.

In a similar way, when there is an optional element the arrangement is

```
L"%[ ... % | ... %]"
```

saying what to display if the element is not present, and if it is. For example, access types always have a name but their preceding type is optional, so a suitable format for them would be

```
L"%[%| %T.%]%N"
```

which precedes the name by the type and `.` when there is a type. There may be several optional elements within one type. For example, within a formal parameter the name, upper constraint, and default type are all optional. To handle these cases, the `L"%[... % | ... %]"` format looks ahead into its own second part to see what things are being printed there, and prints the second part only if all of those things are present. Nesting these formats is not allowed.

The display operation will abort if it finds that the rules have been broken. For example, an optional item should not appear outside the second part of `L"%[... %]"` unless the user is certain that it will always be present, and a list item should never appear outside the second and third parts of `L"%{ ... %}"`.

1.8. Equality, subtype, join, and meet

The equality, subtype, join, and meet operations play a central role in the theory of types. They apply only to evaluated types, never to raw types. This section defines these operations in general, and further details are provided for each kind of type in the relevant chapter.

The equality operation has already been discussed (Section 1.1), and the point made that KMT uses nominal typing for record types, and structural typing for all other types. Two types may be tested for equality by calling

```
KMT_BOOLEAN KmtTypeEqual(KMT_TYPE type1, KMT_TYPE type2);
```

This function aborts if either parameter is `NULL` or `raw`. The genus of the two types is not specifically consulted, but it is a fact that two types of different genus cannot be equal.

The C ‘==’ operator should never be used to compare types. If it returns `KMT_TRUE` the types are equal, but a `KMT_FALSE` result says nothing. If the aim is efficiency in simple cases, KMT aims for that too, and in fact tries ‘==’ before doing anything more elaborate.

Formally, type T_1 is a *subtype* of type T_2 , written $T_1 \leq T_2$, when T_1 ’s set of values is a subset of T_2 ’s set. It is clear from this definition that the relation is reflexive and transitive, so this is a reasonable notation for it. Another important consequence is the *principle of substitution*: a value of type T_1 may be substituted anywhere a value of type T_2 is expected, without risk of a type error. KMT defines subtyping on type expressions rather than sets of values, inevitably, but its definition does respect these principles. The elements of the subtype relation are given with the definition of each kind of type; the full relation is their reflexive and transitive closure.

Two types may be tested for a subtype relationship by calling

```
KMT_BOOLEAN KmtTypeSubType(KMT_TYPE lower, KMT_TYPE upper);
```

This aborts if either parameter is `NULL` or `raw`. The genus of the two types is not specifically consulted, but it is a fact that two types of different genus cannot lie in the subtype relation.

It is not true that $T_1 \leq T_2$ and $T_2 \leq T_1$ together imply $T_1 = T_2$ in general. For example, a type *int* of unboxed integers and a type *int_ref* of boxed integers may be subtypes of each other (presumably connected in the implementation by coercions), but they are not therefore considered to be equal. KMT permits such cycles in the inheritance relation, and users who wish to outlaw them have to detect them and take action on the user side.

The expression ‘ T_1 is a supertype of T_2 ’ means ‘ T_2 is a subtype of T_1 ’ and is written $T_1 \geq T_2$. It is merely a convenient form of expression; there is no distinct supertype operation.

Formally, the *join* of two types, T_1 and T_2 , written $T_1 \vee T_2$, is their union as sets of values. Following the usual type systems practice, KMT defines the join as the least common supertype of the two types. That is, it is a type with the three properties

1. $T_1 \vee T_2 \geq T_1$;
2. $T_1 \vee T_2 \geq T_2$;
3. If T is any type such that $T \geq T_1$ and $T \geq T_2$, then $T \geq T_1 \vee T_2$.

Joins are useful for typing conditional expressions such as

if cond then a else b end

The overall result could have the type of *a* or the type of *b*; that is, it is a supertype of both. The join is the most specific type with this property.

The join of two types need not exist. For example, the two types could have no supertypes in common, like *int* and *list[int]* (probably), or they could have some without there being one that is least. Types of different genus cannot have a join.

Formally, the *meet* of two types, T_1 and T_2 , written $T_1 \wedge T_2$, is their intersection as sets of values. KMT defines the meet as the greatest common subtype of the two types. That is, it is a type with the three properties

1. $T_1 \wedge T_2 \leq T_1$;
2. $T_1 \wedge T_2 \leq T_2$;
3. If T is any type such that $T \leq T_1$ and $T \leq T_2$, then $T \leq T_1 \wedge T_2$.

Meet types are closely related to multiple inheritance: the multiple inheritance of several types is much the same as the single inheritance of the meet of those types. Meet types are also occasionally useful as explicit types, for example as upper constraints on type variables.

The meet of two types need not exist. For example, since KMT does not permit inconsistent genericity, the types *list[int]* and *list[real]* have no meet, because it would be a subtype of both of these types if it did exist. Types of different genus also cannot have a meet.

The join and meet of two equal types are both that type; if one type is a subtype of another, the join is the higher of the two types, and the meet is the lower.

Joins and meets are used within the implementation of range types. A lower constraint is accumulated into a range type by joining it with any lower constraint already present, and an upper constraint is accumulated by meeting it with any upper constraint already present.

KMT offers join and meet operations:

```
KMT_BOOLEAN KmtTypeJoin(KMT_TYPE type1, KMT_TYPE type2, KMT_TYPE *res);
KMT_BOOLEAN KmtTypeMeet(KMT_TYPE type1, KMT_TYPE type2, KMT_TYPE *res);
```

In each case, if a type with the three properties can be found, `KMT_TRUE` is returned and `*res` is set to that type. If it cannot be found, `KMT_FALSE` is returned and `*res` is not modified. None of the parameters may be a raw type, but, for convenience, either or both of `type1` and `type2` may be `NULL`, and then `KMT_TRUE` will be returned and `*res` set to the other (to `NULL` if both are `NULL`). For example, to accumulate the join of several types one could write

```
join_type = NULL;
if( !KmtTypeJoin(join_type, type1, &join_type) )
    Error("type1 did not join");
if( !KmtTypeJoin(join_type, type2, &join_type) )
    Error("type2 did not join");
if( !KmtTypeJoin(join_type, type3, &join_type) )
    Error("type3 did not join");
```

which reports types that did not join but otherwise ignores them. The first error message will never be printed, because the first join has one `NULL` parameter, so cannot fail.

Detailed specifications of `KmtTypeJoin` and `KmtTypeMeet` for each kind of type are not part of the specification of KMT; they are so rigidly constrained by their relationship with subtyping that further definition would add nothing. Some detailed information is presented in the relevant sections of other chapters as background, and full information about the current implementation is available elsewhere [2].

Chapter 2. Function types

Function types are the types of functions, generic classes, and other entities with parameters. They were introduced in Section 1.1, using the syntax

$$\mathbf{fun}[x_1: T_1, \dots, x_k: T_k](x_{k+1}: T_{k+1}, \dots, x_n: T_n): T$$

After the **fun** keyword comes a sequence of formal parameters, each with an optional *upper constraint*. The syntax encloses typic (Section 1.3) formal parameters in brackets and non-typic ones in parentheses, but in actual use there is just one sequence of formal parameters, with the two kinds arbitrarily intermixed. The final type, which is compulsory, is called the *result type*.

2.1. Creation and query

A function type is created by a sequence of calls beginning with

```
KMT_FUNCTION_TYPE KmtFunctionTypeMakeBegin(KMT_BOOLEAN evaluated,
void *impl, KMT_FUNCTION_POLICIES policies);
```

The first parameter says whether the function type is to be created in an already evaluated state, and will usually be `KMT_FALSE`. After it comes the implementation pointer (Section 1.2), and a set of policies that control function type operations, described below. A sequence of formal parameters is then added by a sequence of zero or more calls to

```
KMT_FORMAL_PARAMETER KmtFunctionTypeMakeParameter(
KMT_FUNCTION_TYPE function_type, void *impl,
KMT_PARAMETER_POLICIES policies, wchar_t *name, char name_syntax,
KMT_TYPE upper_constraint, KMT_TYPE default_type,
KMT_FORMAL_PARAMETER *clash);
```

This both creates a formal parameter and adds it to `function_type`. The attributes are the usual implementation pointer, a set of policies that control function type operations at the parameter, described below, the parameter name (which may be `NULL`, indicating that the parameter has no name) and name syntax, an optional upper constraint, and an optional default type (for which see the `match_unmatched` parameter policy below). The new parameter will not be created if it would cause a name clash with an existing parameter; in that case, `NULL` is returned and `*clash` is set to that existing parameter. See below for the circumstances in which a name clash is considered to occur. Finally, the function type receives a result type and is completed by

```
void KmtFunctionTypeMakeEnd(KMT_FUNCTION_TYPE function_type,
KMT_TYPE result_type);
```

It may not be used in any way until after this last call has been made. If an evaluated function type is being created, all types passed by all these calls must be evaluated.

The attributes of a function type may be retrieved, and its implementation pointer reset, by

```

void *KmtFunctionTypeImpl(KMT_FUNCTION_TYPE function_type);
void KmtFunctionTypeSetImpl(KMT_FUNCTION_TYPE function_type, void *impl);
KMT_FUNCTION_POLICIES KmtFunctionTypePolicies(KMT_FUNCTION_TYPE function_type);
int KmtFunctionTypeParameterCount(KMT_FUNCTION_TYPE function_type);
KMT_FORMAL_PARAMETER KmtFunctionTypeParameter(KMT_FUNCTION_TYPE function_type,
    int index);
KMT_TYPE KmtFunctionTypeResultType(KMT_FUNCTION_TYPE function_type);
int KmtFunctionTypeGenus(KMT_FUNCTION_TYPE function_type);

```

`KmtFunctionTypeGenus` aborts if the function type is raw. The genus of an evaluated function type is as given in its policy set (see below). `KmtFunctionTypeParameter` returns the `index`'th formal parameter, counting from 0 as usual in C. It will abort if `index` is out of range. The attributes of a formal parameter may be retrieved, and its implementation pointer reset, by calling

```

void *KmtFormalParameterImpl(KMT_FORMAL_PARAMETER formal);
void KmtFormalParameterSetImpl(KMT_FORMAL_PARAMETER formal, void *impl);
KMT_PARAMETER_POLICIES *KmtFormalParameterPolicies(
    KMT_FORMAL_PARAMETER formal);
wchar_t *KmtFormalParameterName(KMT_FORMAL_PARAMETER formal);
char KmtFormalParameterNameSyntax(KMT_FORMAL_PARAMETER formal);
KMT_TYPE KmtFormalParameterUpperConstraint(KMT_FORMAL_PARAMETER formal);
KMT_TYPE KmtFormalParameterDefaultType(KMT_FORMAL_PARAMETER formal);

```

Both the upper constraint and the default type may be `NULL`.

The policies that control type operations on function types are divided into those related to the function type as a whole, stored in objects of type `KMT_FUNCTION_POLICIES`, and those related to individual parameters, stored in objects of type `KMT_PARAMETER_POLICIES`. Both types are pointers to immutable records, allowing sets of policies to be shared. A function policies object is created by calling

```

KMT_FUNCTION_POLICIES KmtFunctionPoliciesMake(
    char          genus,
    char          result_expected_genus_set,
    KMT_TYPEOP   match_result_typeop,
    KMT_VARIANCE subtype_result_variance,
    KMT_CLOSURE_POLICIES closure_policies);

```

The first policy defines the genus of the function type. The second is a set of genera (Section 1.3) that the genus of the result type of the function is expected to be a member of. The third influences the function matching operation (that is, the application of a set of actual parameters to the function), and is explained in Section 4.4. The fourth influences the subtype operation between function types (Section 2.4). The last is a set of policies used when inserting closure parameters (Section 7.6), or is `NULL` when these are not wanted. These attributes may be retrieved by calling

```

char KmtFunctionPoliciesGenus(KMT_FUNCTION_POLICIES policies);
char KmtFunctionPoliciesResultExpectedGenusSet(
    KMT_FUNCTION_POLICIES policies);
KMT_TYPEOP KmtFunctionPoliciesMatchResultTypeOp(
    KMT_FUNCTION_POLICIES policies);
KMT_VARIANCE KmtFunctionPoliciesSubtypeResultVariance(
    KMT_FUNCTION_POLICIES policies);
KMT_CLOSURE_POLICIES KmtFunctionPoliciesClosurePolicies(
    KMT_FUNCTION_POLICIES policies);

```

A parameter policies object, containing policies that control type operations at individual parameters, is created by calling

```

KMT_PARAMETER_POLICIES KmtParameterPoliciesMake(
    char                genus,
    KMT_BOOLEAN        is_typic,
    int                species,
    wchar_t            *species_label,
    int                match_kind,
    KMT_MATCH_STYLE    match_style,
    KMT_UNMATCHED      match_unmatched,
    KMT_TYPEOP         match_typeop,
    KMT_RANGE_POLICIES match_range_policies,
    KMT_VARIANCE       subtype_variance,
    KMT_BOOLEAN        subtype_extra
);

```

The first two policies are the *genus* and *is_typic* attributes defined in Section 1.3; the *genus* is also the expected genus set for any upper constraint and default type. The third is a more specific classification called the *species*, like the *genus* a small integer chosen by the user. The next is a label for the *species*, not used by KMT but handy when printing error messages. For example, an ordinary parameter would have *genus type*, *KMT_FALSE* for *is_typic*, a *species* used only for ordinary parameters, and *species label* L"parameter", whereas a generic parameter would have *genus type*, *KMT_TRUE* for *is_typic*, a *species* used only for generic parameters, and *species label* L"generic parameter". This makes it easy to generate error messages such as

```
generic parameter found where expression expected
```

which are more informative than just saying that something had the wrong *genus*.

The next five policies influence the function matching operation (that is, the application of a set of actual parameters to the function) at this parameter, and are explained in Section 4.4. The last two influence the function subtype operation, and are explained in Section 2.4.

The attributes of a parameter policies object may be retrieved by calling

```

char KmtParameterPoliciesGenus(KMT_PARAMETER_POLICIES policies);
KMT_BOOLEAN KmtParameterPoliciesIsTypic(KMT_PARAMETER_POLICIES policies);
int KmtParameterPoliciesSpecies(KMT_PARAMETER_POLICIES policies);
wchar_t *KmtParameterPoliciesSpeciesLabel(KMT_PARAMETER_POLICIES policies);
int KmtParameterPoliciesMatchKind(KMT_PARAMETER_POLICIES policies);
KMT_MATCH_STYLE KmtParameterPoliciesMatchStyle(KMT_PARAMETER_POLICIES policies);
KMT_UNMATCHED KmtParameterPoliciesMatchUnmatched(
    KMT_PARAMETER_POLICIES policies);
KMT_TYPEOP KmtParameterPoliciesMatchTypeOp(KMT_PARAMETER_POLICIES policies);
KMT_RANGE_POLICIES KmtParameterPoliciesMatchRangePolicies(
    KMT_PARAMETER_POLICIES policies);
KMT_VARIANCE KmtParameterPoliciesSubtypeVariance(
    KMT_PARAMETER_POLICIES policies);
KMT_BOOLEAN KmtParameterPoliciesSubtypeExtra(KMT_PARAMETER_POLICIES policies);

```

If `match_style` allows the formal parameter to match with a named actual parameter, then the name attribute of the formal parameter must be non-NULL, otherwise KMT will abort. If the name attribute of a formal parameter is non-NULL, it must be distinct from the names of all formal parameters of the function with the same `match_kind`, otherwise a name clash is considered to have occurred, with the consequences explained above.

2.2. Evaluation and validity

The evaluation operation, `KmtTypeEvaluate`, was introduced in Section 1.4. This section explains how it applies to function types.

Evaluation of a raw function type evaluates its constituent types (its parameters' upper constraints and default types, and its result type), mutating the raw object into an evaluated one in place. The `KMT_EVALUATION_POLICIES` object from Section 1.4 contains four error callback functions related to function types. In the syntax used to declare these functions, they are:

```
void function_upper_wrong_genus(KMT_FUNCTION_TYPE fun_type, int index);
```

The `index`'th formal parameter of `fun_type` has an upper constraint whose genus is not equal to the genus of the formal parameter.

```
void function_default_wrong_genus(KMT_FUNCTION_TYPE fun_type, int index);
```

The `index`'th formal parameter of `fun_type` has a default type whose genus is not equal to the genus of the formal parameter.

```
void function_default_wrong_type(KMT_FUNCTION_TYPE fun_type, int index);
```

The `index`'th formal parameter of `fun_type` has both an upper constraint and a default type, and the the default type does not type check against the upper constraint.

```
void function_result_wrong_genus(KMT_FUNCTION_TYPE fun_type);
```

The genus of the result type of `fun_type` does not lie in the expected genus set given by the

result_expected_genus_set policy of the function type.

The context parameter of `KmtTypeEvaluate` is stored in the function type, and remains available after evaluation. The context used when evaluating the constituent types is the function type itself, which means that its parameters are visible in the constituent types in addition to everything in the original context.

2.3. Display

The operation for displaying a type, `KmtTypeShow`, was introduced in Section 1.7. This section explains how it applies to function types. For convenience, a function

```
wchar_t *KmtFunctionTypeShow(KMT_FUNCTION_TYPE fun_type);
```

is defined which displays a function type by upcasting and calling `KmtTypeShow`.

`KMT_TYPE_FMT` objects contain three strings which define the format of function types: `function_fmt`, `function_formal_list_fmt`, and `function_formal_fmt`. The first defines the format of the function type as a whole; the second defines the format of a sequence of formal parameters; and the third defines the format of one formal parameter.

Parameters of different kinds usually have different syntax. For example, in

```
map[y](f : funI[x, y]): list[y]
```

the brackets around `y` indicate that it has one kind (it is a generic parameter), whereas the parentheses around `f : funI[x, y]` indicate that it has another (it is an ordinary parameter). Accordingly, the formal parameter list is analysed into a sequence of maximal non-empty sequences of formal parameters of equal kind, and each subsequence is displayed separately using the `function_formal_list_fmt` and `function_formal_fmt` strings used to display the subsequence as a whole, and its formal parameters individually.

Format string `function_fmt` may contain the formatting commands `L"%S"` for a sequence of formal parameters of equal kind, and `L"%R"` for the result type. For example,

```
L"%{ % | %S% | %S% | % } : %R"
```

produces a Pascal-style display, with the sequences of formal parameters printed adjacent to each other, and the result type at the end after a colon and space.

Format string `function_formal_list_fmt` may contain the formatting command `L"%P"` for one formal parameter. For example,

```
L"(%{ % | %P% | , %P% | % } )"
```

produces a sequence of formal parameters enclosed in parentheses and separated by commas. Owing to the way sequences of formal parameters are analyzed, the sequence printed by `function_formal_list_fmt` can never be empty.

Format string `function_formal_fmt` may contain the formatting commands `L"%N"` for the formal parameter's name, `L"%U"` for its upper constraint, and `L"%D"` for its default type. All three are optional. For example,

```
L"%[%| %N%]%[%| : %U%]"
```

prints the name, if any, followed by the upper constraint (preceded by a colon), if any.

Within `function_formal_list_fmt` and `function_formal_fmt` it is possible to vary the format depending on the parameter kind, by means of the commands `L"%0"`, `L"%1"`, ..., `L"%9"`, which cause everything following, up to the next command of the same kind or the end of the string, to be printed only if the kind equals the digit. Command `L"%-` removes this restriction. For example, the default formats expect up to two parameter kinds, 0 for ordinary parameters and 1 for generic parameters, and in them the left parenthesis above is replaced by

```
%0(%1[%-
```

and similarly for the right parenthesis. These commands may contain and be contained within sequences and options.

2.4. Equality, subtype, join, and meet

The equality, subtype, join, and meet operations were introduced in Section 1.8. They apply only to evaluated types, so this section explains their operation on evaluated function types.

All four operations may safely be given any two evaluated types. KMT aborts only if use is made of a variance policy (described below) with value `KMT_VARIANCE_ERROR`, which should never happen. In this section, then, a statement that some condition must hold, or that there is no join or meet, means that the operation returns `KMT_FALSE`, not that it aborts.

Operations involving a function type and a non-function type return `KMT_FALSE`, with a few obvious exceptions (when the non-function type is an error type, or perhaps a parameter type whose upper constraint is a function type). So it will be assumed that both types involved in each operation are function types.

2.4.1. Corresponding parameters

The first step in all four operations is to find a correspondence between the parameters of the two function types involved, so that corresponding pairs of parameters can be checked. Traditionally, the correspondence is positional: the first formal parameter of the first function type corresponds with the first of the second function type, the second with the second, and so on. But function matching (Section 4.4) goes beyond this. Each formal parameter has an integer `match_kind` policy (its *kind*), and a `match_style` policy (its *match style*) of type

```
typedef enum {
    KMT_MATCH_STYLE_NAMED,
    KMT_MATCH_STYLE_POSITIONAL,
    KMT_MATCH_STYLE_POSITIONAL_OR_NAMED
} KMT_MATCH_STYLE;
```

specifying that the formal parameter may match with a named actual parameter, a positional actual parameter, or either. Each actual parameter has a kind, and is either named or positional. Function matching matches named actual parameters with formal parameters with the same kind and name (in this section, ‘name’ means ‘name and name syntax’) whose match style is

not `KMT_MATCH_STYLE_POSITIONAL`, then matches positional actual parameters with the remaining formals with the same kind whose match style is not `KMT_MATCH_STYLE_NAMED`.

To agree with function matching, the four type operations use the correspondence between formal parameters defined by the following algorithm. Choose one of the two function types arbitrarily, and visit each of its formal parameters in order. If the parameter has match style `KMT_MATCH_STYLE_NAMED`, match it with an unmatched formal parameter of the other function type with the same kind, name, and match style (there can be at most one); otherwise, match it with the first remaining unmatched formal parameter of the other function type with the same kind whose match style is also not `KMT_MATCH_STYLE_NAMED`.

Formal parameters from either function type that fail to match are called *leftover formal parameters*. Whether they are permitted depends on the operation, but in all cases, a leftover formal parameter must have match style `KMT_MATCH_STYLE_NAMED`, and the user must specify that it may be left over by setting its `subtype_extra` policy to `KMT_TRUE`, otherwise the operation returns `KMT_FALSE` without going further. Allowing positional parameters to be left over causes problems when forming the meet of two function types, which is why they are forbidden regardless of the value of `subtype_extra`.

All four operations consider corresponding typic parameters to be equal as types. This makes the types `fun[x: person]: list[x]` and `fun[y: person]: list[y]` equal, for example.

2.4.2. Function type equality

Two function types are equal if they have equal policies, equal result types, and, when matched together as just defined, there are no leftover formal parameters, and corresponding formal parameters have equal elements. Two formal parameters have equal elements if they have equal policies, name (except as explained next), upper constraint, and default type.

The exception involving the name, which applies to the other operations as well as equality, is that when a parameter's match style is `KMT_MATCH_STYLE_POSITIONAL`, any name that it may have is considered to be not part of the type, and so all tests or actions involving the name are omitted. For example, this allows anonymous positional parameters in a function interface specification to match with named positional parameters in a function declaration. When the formal parameter's match style is not `KMT_MATCH_STYLE_POSITIONAL`, the formal parameter must have a name, and that name is part of the type and influences the four operations.

2.4.3. Function subtyping

There is a standard formula for function subtyping, but it does not handle the options of KMT's function types. So it is necessary to review the properties that function subtyping should have, and define an operation with those properties.

Deciding that f_2 is a subtype of f_1 means deciding that f_2 may replace f_1 safely. This will be so provided that, first, for all sets of actual parameters P , if $f_1(P)$ is type-safe, then so is $f_2(P)$; and second, the result type of $f_2(P)$ is a subtype of the result type of $f_1(P)$ for all P . Any definition of function subtyping must have these two properties, plus the usual reflexivity and transitivity.

The connection to function matching, assumed above without justification, is clearly evident here, since deciding whether $f_1(P)$ is type-safe is the job of function matching. For example, one

consequence of the relationship $f_2 \leq f_1$ is that if an attempt to match f_2 with P fails, then a subsequent attempt (which may well occur during retrieval) to match f_1 with P must also fail.

At the minimum, the subtype relation for function types could be the same as the equality relation, as it was initially in C++ [7]. The required conditions then hold trivially, but the resulting language lacks expressiveness, which is why C++ moved to a larger relation.

At the maximum, the two properties could be used to *define* the function subtype relation. This gives the most general relation and so the most expressive language, and reflexivity and transitivity are trivial. Problems may arise in checking the two conditions, which range over an infinite set of sets of actual parameters, and in finding concrete definitions of the join and meet of two function types which satisfy the conditions that characterise those operations.

Let f_1 have type $\mathbf{fun}[x: X_1]: T_1$, and let f_2 have type $\mathbf{fun}[x: X_2]: T_2$, showing only one parameter to keep the notation simple. The standard formula is

$$\frac{X_2 \geq X_1 \quad T_2 \leq T_1}{\mathbf{fun}[x: X_2]: T_2 \leq \mathbf{fun}[x: X_1]: T_1}$$

This is to be read ‘function type $\mathbf{fun}[x: X_2]: T_2$ is a subtype of function type $\mathbf{fun}[x: X_1]: T_1$ if $X_2 \geq X_1$ at each parameter, and $T_2 \leq T_1$ ’. It is easily seen to satisfy all the required conditions.

This formula assumes the traditional positional correspondence between parameters, whereas KMT uses the correspondence defined earlier. The point mentioned there, that corresponding typic parameters are considered equal as types, is indicated in the formula by its use of the same names x for the parameters of both function types. KMT also allows leftover formal parameters on request, but only in the lower function type. The rationale is that when the lower function type replaces the upper one, leftover parameters may receive a default value.

The major issue in function subtyping is whether the elements of a function type may change in going from the upper function type to the lower. Elements for which KMT makes no provision for change are *constant*; these must be the same in both types. The others are *variable*. Within function type objects, the policies are constant but the result types and leftover formal parameters are variable (although neither term applies well to leftover formal parameters, since they don’t exist at all in the upper type). Within corresponding formal parameters, the policies are constant except for `match_style`, and the `match_style` policies, names, upper constraints, and default types are variable. (The presence or absence of a default type is constant, because it is determined by the constant `match_unmatched` policy.) There may be cases where the `match_unmatched` policy could usefully vary, but none of these are allowed at present.

Variable elements may change in three ways. They may change to something more restricted in the lower type, in which case they are called *covariant*, because the function type itself may be viewed as changing in that way. They may change to something less restricted in the lower type, in which case they are *contravariant* – they change in the contrary direction to the function type. Or they may remain the same, that is, their freedom to change is not taken up, making them *invariant*. These terms are usually applied to the upper constraints of formal parameters and to result types. In KMT they are also applied to each formal parameter as a whole, because its variable elements are all *simultaneously* covariant, contravariant, or invariant; one cannot request that a formal parameter’s match style be invariant while its upper constraint be contravariant, for example.

The variance of result types is determined by the common `subtype_result_variance` policy of the function types involved. Its type is

```
typedef enum {
    KMT_VARIANCE_ERROR,
    KMT_VARIANCE_INVARIANT,
    KMT_VARIANCE_COVARIANT,
    KMT_VARIANCE_CONTRAVARIANT
} KMT_VARIANCE;
```

The values indicate that the function type should never participate in any subtype, join, or meet operation (KMT will abort if it does), or that a check should be made for invariance, covariance, or contravariance. The standard formula specifies that result types should be covariant, and in general the type-safe choices are `KMT_VARIANCE_ERROR`, `KMT_VARIANCE_INVARIANT`, and `KMT_VARIANCE_COVARIANT`. There is no option for not checking the variance at all, because that would leave the result types of joins and meets of function types unspecified.

The variance of corresponding formal parameters is set by their `subtype_variance` policy, which also has type `KMT_VARIANCE`. The remainder of this section is devoted to the effects of this policy. As mentioned above, it simultaneously determines the variance of the corresponding formal parameters' upper constraints, default types, match styles, and names.

The standard formula requires the upper constraints of corresponding formal parameters to be contravariant. This follows from the principle of substitution: if f_2 is to replace f_1 , then it must accept any values for its parameters that f_1 would accept, and possibly others.

Upper constraints are optional in formal parameters, so one or both of the values compared when testing the variance of the upper constraints of corresponding formal parameters may be `NULL`. The equality and subtype operations used in these comparisons do not normally accept `NULL` values, but here the interpretation 'NULL means unconstrained' is used to extend their definitions to cover these cases: invariance holds when either both of the values are `NULL` or both are non-`NULL` and equal; covariance holds when the upper value is `NULL`, or both are non-`NULL` and the lower is a subtype of the upper; and contravariance holds when the lower value is `NULL`, or both are non-`NULL` and the upper is a subtype of the lower.

Care is needed in choosing `subtype_variance`. For ordinary parameters the type-safe choices are `KMT_VARIANCE_INVARIANT` and `KMT_VARIANCE_CONTRAVARIANT`. Generic parameters have the same type-safe choices, but type checking when contravariance is requested has been shown to be an undecidable problem, so KMT cannot guarantee termination – a good reason for choosing `KMT_VARIANCE_INVARIANT`. If the 'self' or 'this' parameters of methods are included in the methods' function types, `KMT_VARIANCE_COVARIANT` is the right choice for them (covariance being safe because the implementation uses dynamic dispatch on them).

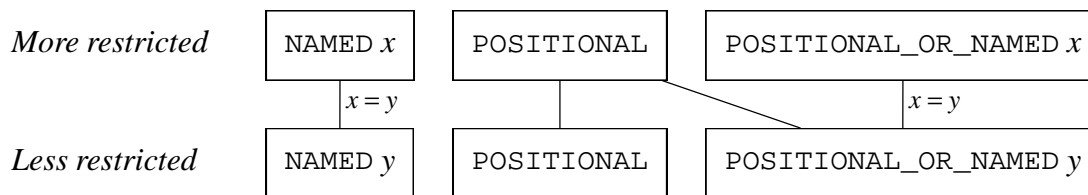
Parameter contravariance is incontrovertible, yet it has been argued over for years. Eiffel [3] actually has a covariant parameter rule. There seems to be a desire to somehow make covariant parameters work [1], perhaps stemming from a belief that they are useful in object-oriented design. This belief must be mistaken, because covariant parameters are not type-safe.

Whether default types are naturally contravariant is a nice point. Consider upper parameter x_1 with upper constraint X_1 and default type D_1 , and corresponding lower parameter x_2 with upper constraint X_2 and default type D_2 . By the nature of default types, $D_1 \leq X_1$ and $D_2 \leq X_2$, and

in the usual contravariant case, $X_2 \geq X_1$. When the lower function type replaces the upper in a call where default values are used, a value of type D_1 will be passed where a value of type X_2 is expected, which is safe without imposing any further conditions. On the other hand, if default types are to appear in types at all, some rule is needed for calculating them in joins and meets. KMT cuts this knot by treating default types (when present) as it treats upper constraints.

The other variable elements of formal parameters, their names and match styles, interact quite subtly with subtyping. Invariant parameters must have the same match style and name (with the exception for names explained above). Covariant parameters can be swapped and tested for contravariance, leaving just the contravariant case.

If either match style is `KMT_MATCH_STYLE_NAMED`, then, by the way the correspondence was constructed, both the names and match styles will be equal and there is nothing to check. Otherwise, contravariance requires that the match style be less restricted in the lower parameter, which rules out the combination of `KMT_MATCH_STYLE_POSITIONAL_OR_NAMED` in the upper with `KMT_MATCH_STYLE_POSITIONAL` in the lower. All other combinations are legal. The names must be equal when both styles are not `KMT_MATCH_STYLE_POSITIONAL`. These rules are summarized in this diagram:



Here x and y are the parameter names, which must be present where shown. All legal cases are indicated by lines, and $x = y$ means that the names must be equal.

2.4.4. Function type joins and meets

The rules for function subtyping lead to straightforward definitions of join and meet which satisfy the conditions placed on those operations, generalized to respect the variance options. In particular, the join of two function types is a supertype of both according to their own lights, and the meet is a subtype of both. These basic properties may be all the user needs to know; but, since the matter is complicated, for the record their detailed consequences are documented here.

The join or meet of two function types, if it exists, is a function type. The constant parts of the two must be equal, and their common values are given to their join or meet.

If the `subtype_result_variance` policy of the two function types specifies that result types are to be invariant, then the two result types must be equal and the result type of the join or meet is their common value. If the policy specifies covariance, then the result type of their join is the join of their result types, and the result type of their meet is the meet of their result types; this is reversed if the policy specifies contravariance (which it never should).

The join of two function types has one parameter for each pair of corresponding parameters in the two types; the meet has these parameters too, but it also has one parameter for each leftover parameter of the two types. These last will be simple copies of the original leftover parameters. As usual, leftover parameters must have match style `KMT_MATCH_STYLE_NAMED` and their `subtype_extra` policies must have value `KMT_TRUE`.

It remains to specify the variable parts of each parameter of the meet or join that represents a pair of corresponding parameters: its upper constraint, default type, match style, and name.

If the corresponding parameters have invariant variance, then they must have equal upper constraints, default types, match styles, and names (with the usual exception). The parameter that represents them in the join or meet has these values; it has a name only if the names are equal.

Consider two corresponding parameters with contravariant variance, lying in two function types being joined (or with covariant variance, lying in two function types being met). It may help to think of the two function types as alternative results of a conditional expression, this being the leading application of the join operation. The two parameters must accept everything that the parameter that represents them in the join accepts. So the join parameter's upper constraint is the meet of their upper constraints (NULL meaning unconstrained, as usual), and its default type (if required) is the meet of their default types. If either meet does not exist, the function types have no meet. The join parameter's name and match style are given by this table:

<i>Contravariant join</i>	POSITIONAL	POSITIONAL_OR_NAMED y_2
POSITIONAL	POSITIONAL	POSITIONAL
POSITIONAL_OR_NAMED y_1	POSITIONAL	POSITIONAL_OR_NAMED y_1 if $y_1 = y_2$, else POSITIONAL

The KMT_MATCH_STYLE_NAMED case is simple (corresponding parameters have the same name and match style) and separate, so has been omitted. If any entry in this table is positioned in the top row of the subtyping diagram above, and its column or row heading is positioned in the bottom row, then a line connects them, as required. When the table does not define a name for the parameter, that is, when the result has match style KMT_MATCH_STYLE_POSITIONAL, the parameter is given a name when both parameters have a name and the two names are equal.

Or take two corresponding parameters with contravariant variance, lying in two function types being met (or with covariant variance, lying in two function types being joined). The leading application here is to the multiple inheritance of function types. The parameter that represents the two parameters in the meet must accept everything that they accept. So its upper constraint is the join of their upper constraints (NULL meaning unconstrained, as usual), and its default type (if required) is the join of their default types. If either join does not exist, the function types have no meet. The meet parameter's name and match style are given by this table:

<i>Contravariant meet</i>	POSITIONAL	POSITIONAL_OR_NAMED y_2
POSITIONAL	POSITIONAL	POSITIONAL_OR_NAMED y_2
POSITIONAL_OR_NAMED y_1	POSITIONAL_OR_NAMED y_1	POSITIONAL_OR_NAMED y_1 if $y_1 = y_2$, else no meet

The KMT_MATCH_STYLE_NAMED case has been omitted as before. This time there is no remedy when both parameters have match style KMT_MATCH_STYLE_POSITIONAL_OR_NAMED, but their names differ: no parameter covers everything they both do, so there is no meet. This table may be checked by positioning any entry in the bottom row of the subtyping diagram, and its column or row heading in the top row. When the table does not define the parameter's name, the parameter is given a name when both parameters have a name and the two names are equal.

Chapter 3. Parameter types

A parameter type, as introduced in Section 1.1, is an invocation of a formal parameter of a function type, expressed by the syntax

x

appearing anywhere within the function type's upper constraints, default types, and result type. This chapter is a detailed guide to parameter types.

3.1. Creation and query

A parameter type is an object of type `KMT_PARAMETER_TYPE`. It invokes a formal parameter but is different from a formal parameter, which is defined in Section 2.1 to be an object of type `KMT_FORMAL_PARAMETER`.

There are no raw parameter types, because, as Section 1.4 remarked, parameter types are syntactically indistinguishable from access types, and a parser that had to distinguish between the two would be seriously embarrassed. Instead, an access type is created in both cases, and the ambiguity is cleared up during evaluation. If the raw type turns out to refer to a formal parameter, then the access type mutates in place into a parameter type.

It is also possible to create a parameter type directly, by calling

```
KMT_PARAMETER_TYPE KmtParameterTypeMake(void *impl,
    KMT_FORMAL_PARAMETER formal);
```

The attributes are the usual implementation pointer and the formal parameter being invoked. This function will abort if `formal` is `NULL`. A parameter type is valid if it invokes a typic formal parameter, and lies within that formal parameter's function type. The second part is not checked by `KMT`, but parameter types produced by evaluating raw types always satisfy both conditions.

The attributes may be retrieved, and the implementation pointer may be reset, by calling

```
void *KmtParameterTypeImpl(KMT_PARAMETER_TYPE param_type);
void KmtParameterTypeSetImpl(KMT_PARAMETER_TYPE param_type, void *impl);
KMT_FORMAL_PARAMETER KmtParameterTypeFormal(KMT_PARAMETER_TYPE param_type);
int KmtParameterTypeGenus(KMT_PARAMETER_TYPE param_type);
```

The genus of a parameter type is the genus of its formal parameter.

3.2. Display

The display operation, `KmtTypeShow`, was introduced in Section 1.7. This section documents the parts relevant to displaying parameter types. For convenience, a function

```
wchar_t *KmtParameterTypeShow(KMT_PARAMETER_TYPE param_type);
```

is defined which displays a parameter type, by upcasting and calling `KmtTypeShow`.

Each `KMT_TYPE_FMT` object contains one string, `parameter_fmt`, which defines the format of parameter types. Within it, the formatting command `L"%N"` displays the name of the parameter type's formal parameter, and one could use just that as the value of `parameter_fmt`, although, since the name is optional, it would be more robust to use something like

```
L"%[ <anon>% | %N% ]"
```

The format may be varied depending on the kind of the formal parameter, by using the `L"%0"`, `L"%1"`, ..., `L"%9"`, and `L"%-`" commands as described for function types in Section 2.3.

3.3. Equality, subtype, join, and meet

This section explains how the equality, subtype, join, and meet operations introduced in Section 1.8 apply to parameter types. The value of a parameter type could be anything at all, except that it must be a subtype of the upper constraint of the parameter type's formal parameter, if there is one. The definitions below follow naturally from this interpretation.

Two parameter types are equal if they refer to the same formal parameter. A parameter type is not equal to any other type. A parameter type is a subtype of another type if its formal parameter has an upper constraint which is a subtype of the other type.

The definitions of the join and meet operations follow from the definition of the subtype relation, as usual. The join of two parameter types is the higher of those two types if one is a subtype of the other, or else it is the join of their upper constraints if both have upper constraints; otherwise there is no join. A symmetrical definition would apply to the meet operation if formal parameters had *lower* constraints, but since they don't, two parameter types can have a meet only if one is a subtype of the other.

Chapter 4. Call types and function matching

A call type, as introduced in Section 1.1, is the type of a call on a function type:

$$T[T_1, \dots, T_k](T_{k+1}, \dots, T_n)$$

T , called the *head* of the call type, must evaluate to a function type. The other types, called the *actual parameters*, must match with the formal parameters of the function type, and satisfy their upper constraints where present. The syntax shows generic actual parameters enclosed in brackets and the types of ordinary actual parameters enclosed in parentheses, but in actual use a call type has a single sequence of actual parameters, and the two kinds may be arbitrarily mixed.

There are no evaluated call types, only raw ones, because evaluation removes all call types from type expressions, in the same way as evaluation of term expressions such as $\text{sqrt}(2)$ removes all call expressions, leaving only values. There are no equality, subtype, join, or meet operations for call types, since those operations apply only to evaluated types.

The evaluation of a call type is a normal part of the evaluation of a type expression. An example of evaluating $\text{nlist}[\text{int}]$ was given in Section 1.1. But there is another use for evaluation of call types, for type checking a *term* expression, by evaluating a call type representing its type structure. For example, the term expression $\text{sqrt}(2)$ is not handled directly by KMT, but it may be type checked by constructing and evaluating the call type

$$(\text{fun}(x: \text{real}): \text{real})(\text{int})$$

where $\text{fun}(x: \text{real}): \text{real}$ is the type of the sqrt function, and int is the type of the term expression 2. This use of call types to check term expressions is called *function matching*.

4.1. Creation and query

A call type object, of type `KMT_CALL_TYPE`, is created by beginning with

```
KMT_CALL_TYPE KmtCallTypeMakeBegin(void *impl);
```

followed by a sequence of zero or more calls to

```
KMT_ACTUAL_PARAMETER KmtCallTypeMakeParameter(KMT_CALL_TYPE call_type,  
void *impl, char kind, wchar_t *name, KMT_TYPE type);
```

and concluding with

```
void KmtCallTypeMakeEnd(KMT_CALL_TYPE call_type);
```

KMT will abort if these functions are called out of sequence, or if an attempt is made to use the call type before `KmtCallTypeMakeEnd` has been called.

The first function accepts the usual implementation pointer. The second makes and adds one actual parameter. It accepts an implementation pointer, the kind and name of the actual

parameter (it may match only with formal parameters with that kind, and with that name if name is non-NULL), and its type, which is compulsory. The last function ends the construction.

A separate function is provided to pass in the head of the call type:

```
void KmtCallTypeSetHead(KMT_CALL_TYPE call_type, KMT_TYPE head);
```

It may be called at any point within or after the sequence above, and it may be called more than once. This is convenient for all kinds of syntax (the head does not always come first), but it is done this way mainly to support the testing of multiple overloaded methods against the actual parameters of a call type. The call type may be created with its actual parameters but no head, then different heads may be tried. A head must be present whenever the call type is evaluated.

As usual there are functions to query a call type and reset its implementation pointer:

```
void *KmtCallTypeImpl(KMT_CALL_TYPE call_type);
void KmtCallTypeSetImpl(KMT_CALL_TYPE call_type, void *impl);
KMT_TYPE KmtCallTypeHead(KMT_CALL_TYPE call_type);
int KmtCallTypeParameterCount(KMT_CALL_TYPE call_type);
KMT_ACTUAL_PARAMETER KmtCallTypeParameter(KMT_CALL_TYPE call_type,
int index);
```

with the last aborting if index is out of range. The attributes of an actual parameter may also be queried, either individually by

```
void *KmtActualParameterImpl(KMT_ACTUAL_PARAMETER actual);
char KmtActualParameterKind(KMT_ACTUAL_PARAMETER actual);
wchar_t *KmtActualParameterName(KMT_ACTUAL_PARAMETER actual);
KMT_TYPE KmtActualParameterType(KMT_ACTUAL_PARAMETER actual);
```

or all at once by

```
void KmtActualParameterAll(KMT_ACTUAL_PARAMETER actual, void **impl,
char *kind, wchar_t **name, KMT_TYPE *type);
```

Other operations, needed mainly for function matching, appear in Section 4.4.

4.2. Evaluation

The evaluation operation was introduced in Section 1.4. This section explains how evaluation applies to call types, at a level of detail sufficient for evaluating type expressions. Many more details, mainly relevant to function matching, appear in Section 4.4.

Evaluation of a call type evaluates the actual parameters and head in the given lexical context, if they are not already evaluated. It then matches the actual parameters with the formal parameters from the head, checks that the actual parameters satisfy the upper constraints of the formal parameters where present, and finishes by taking a copy of the function type's result type, with all occurrences of the typic formal parameters within it replaced by the actual parameters. This type is the result of the evaluation of the call type.

When evaluation of a call type strikes a problem, it calls one of the callback functions of its

`policies` parameter as described in Section 1.4. Within their bodies, the query operations of Section 4.1 are available as usual. The actual parameters will have been reordered within the call type object to align with the formal parameters they matched with. In the syntax that the user would use to declare them, the callback functions are:

```
void call_head_not_function(KMT_CALL_TYPE call_type);
```

This call type's head did not evaluate to a function type (nor to an error type).

```
void call_unmatched_parameters(KMT_CALL_TYPE call_type);
```

This call type's head evaluated to a function type, but the formal parameters and actual parameters did not match. This could be for several reasons: too few or too many actual parameters, named actual parameters not finding formal parameters with the right names, and so on.

```
void call_parameter_wrong_genus(KMT_CALL_TYPE call_type, int index);
void call_parameter_wrong_type(KMT_CALL_TYPE call_type, int index);
```

This call type's head evaluated to a function type, and the parameters matched, but at position `index` the formal and actual parameters differ in genus, or the type checking operation specified in the formal parameter's `match_typeop` policy did not succeed. One of these callbacks will be generated for each index where there is a problem.

These error callbacks are designed to support the production of brief error messages, such as `L"wrong number of parameters"`, `L"variable where type expected"`, and the like. If more detailed error messages are desired, a fully detailed diagnosis of the problems found when evaluating the call type can be obtained within the error callback functions, by querying `call_type` in more detail, using the operations defined in Section 4.4.

4.3. Display

The operation for displaying a type, `KmtTypeShow`, was introduced in Section 1.7. This section explains how it applies to call types. For convenience, a function

```
wchar_t *KmtCallTypeShow(KMT_CALL_TYPE call_type);
```

is defined which displays a call type, by upcasting and calling `KmtTypeShow`.

Like function type display (Section 2.3), call type display groups maximal sequences of parameters of the same kind together. The `KMT_TYPE_FMT` object contains three format strings which define the format of call type displays.

Format string `call_fmt` is the overall format. Within it, `L"%H"` displays the head, and `L"%S"` displays one maximal subsequence of actual parameters of the same kind, as in

```
L"%H%{%( ) | %S% | %S% | %}"
```

which displays the head first in the conventional way, followed by the sequences of parameters, or by `()` if there are no parameters. Since a head is not always present it would be prudent to enclose `L"%H"` in `L"%[... %]"`.

Format string `call_actual_list_fmt` is the format of one maximal subsequence of

actual parameters of the same kind. Within it, `L"%P"` stands for one actual parameter, as in

```
L"(%{|%P%|, %P%|%})"
```

which produces actual parameters enclosed in parentheses and separated by commas in the conventional way. Although this format string will print just the parentheses if the sequence is empty, in fact, because the overall sequence of actual parameters is analyzed into maximal *non-empty* subsequences, the sequence will never be empty.

Format string `call_actual_fmt` is the format of one actual parameter. Within it, `L"%N"` displays the optional name of the actual parameter, and `L"%T"` displays its type, as in

```
L"%[|%N := %]|%T"
```

which produces the type, preceded by the name and `:=` when there is a name.

Within `call_actual_list_fmt` and `call_actual_fmt` the display format may be varied depending on the actual parameter kind, using the `L"%0"`, `L"%1"`, ..., `L"%9"`, and `L"%-` commands as described for function types in Section 2.3.

4.4. Function matching

The evaluation of call types is closely related to *function matching*, the type checking of term expressions which are function calls. This section explains function matching, including the many options that control both it and the evaluation of call types.

Most of the work done by `KmtTypeEvaluate` when given a call type is relevant to function matching, but two parts are not. First, the actual parameters and head will have already been evaluated, and evaluating them again would achieve nothing. Second, reporting problems via callback functions is not well suited to function matching. Callback functions are useful when `KmtTypeEvaluate` ranges widely over many types, finding many problems along the way; but when the aim is to evaluate just one call type whose elements have already been evaluated, producing a detailed diagnosis of any problems, it is more convenient if the evaluation is done completely and then its problems diagnosed after evaluation returns.

Accordingly, a large chunk of the work done when `KmtTypeEvaluate` evaluates a call type is encapsulated in this function, which the user may call directly:

```
KMT_BOOLEAN KmtCallTypeMatch(KMT_CALL_TYPE call_type, void *range_impl);
```

This section describes `KmtCallTypeMatch` in detail. The references to range types are for readers of Chapter 5, and can be ignored without invalidating the rest.

`KmtCallTypeMatch` assumes that the actual parameters of `call_type` are evaluated, and that the head is an evaluated function type. It matches the actual and formal parameters together, reordering the actual parameters if necessary so that they align with the formal parameters they match with, checks upper constraints, calculates the result type, and returns `KMT_TRUE` if everything was successful, and `KMT_FALSE` otherwise. Parameter `range_impl` is the implementation pointer to give to range types introduced during the match. Any problems encountered along the way are recorded in `call_type`, where they can be queried later.

Before examining `KmtCallTypeMatch`, it will be helpful to see how `KmtTypeEvaluate`

is implemented in terms of it. `KmtTypeEvaluate` first evaluates the actual parameters of the call type, then its head. If the head is an error type, it sets its result type to an error type and returns. Otherwise, if the head is not a function type, it calls the `call_not_function` callback, then again it sets its result type to an error type and returns. Otherwise it calls `KmtCallTypeMatch`, and afterwards it queries `call_type` just as the user can, generating `call_unmatched_params`, `call_param_wrong_genus`, and `call_param_wrong_type` callbacks which summarize the detailed problems found.

It is clear from this that everything described in the remainder of this section as applying to `KmtCallTypeMatch` applies to `KmtTypeEvaluate` as well, and that queries addressed to the call type object can be performed equally well after `KmtCallTypeMatch` returns, or within any of the callbacks generated by `KmtTypeEvaluate` except `call_not_function`. Although the call type object is not part of the evaluated result type, it remains available as a repository of diagnostic information about the match.

4.4.1. Setting up

`KmtCallTypeMatch` obeys policies from the head of `call_type`: `match_result_typeop` taken from the function type itself, and `match_kind`, `match_style`, `match_unmatched`, `match_typeop`, and `match_range_policies` from its formal parameters. Section 2.1 explained how to set these policies; this section explains what they do.

The call type object may contain a desired result type, offered in support of top-down type checking (not often used in practice). It must be an evaluated type if present, and is set by

```
void KmtCallTypeSetResultType(KMT_CALL_TYPE call_type, KMT_TYPE type);
```

When it is present, the `match_result_typeop` policy indicates the type operation to perform between the instantiated result type of the function type and this type. Its possible values are

```
typedef enum {
    KMT_TYPEOP_ERROR,
    KMT_TYPEOP_UNCHECKED,
    KMT_TYPEOP_EQUAL,
    KMT_TYPEOP_SUBTYPE,
    KMT_TYPEOP_SUPERTYPE
} KMT_TYPEOP;
```

The first value indicates that no operation should ever be requested here (presumably all type checking is bottom up, so no desired result type is ever supplied); if it is, KMT aborts. The others indicate that no type check should be performed, that the two types should be checked for equality, and that the instantiated function type result type should be checked for being a subtype of the desired result type, or a supertype. The only type-safe options are `KMT_TYPEOP_ERROR`, `KMT_TYPEOP_EQUAL`, and `KMT_TYPEOP_SUBTYPE`.

The `match_kind` policy of each formal parameter helps to guide the matching of actual parameters with formal parameters: actual parameters also have a kind, and will match only with formal parameters of the same kind. The kind of a formal parameter normally encodes the genus and *is_typic* attributes of the formal parameter, and possibly other syntax-oriented features such as being a left or right parameter of an operator. The kind of an actual parameter encodes similar

information about the actual parameter. For example, in the call

```
cons[int](5, null[int])
```

the first actual parameter, *int*, would have a kind indicating that it is a generic parameter, while the others would have kinds indicating that they are ordinary parameters.

The `match_style` policy of each formal parameter says what *parameter matching style* is permitted when matching the parameter with an actual parameter. Its possible values are:

```
typedef enum {
    KMT_MATCH_STYLE_NAMED,
    KMT_MATCH_STYLE_POSITIONAL,
    KMT_MATCH_STYLE_POSITIONAL_OR_NAMED
} KMT_MATCH_STYLE;
```

meaning only by name, only positional, or either. It would be possible for the user to encode this information in the value of `match_kind`, but it turns out that KMT needs to know it explicitly during the subtype operation on function types, hence its presence here.

The `match_unmatched` policy in each formal parameter has values

```
typedef enum {
    KMT_UNMATCHED_MISSING,
    KMT_UNMATCHED_CURRY_ATEND,
    KMT_UNMATCHED_CURRY_ANY,
    KMT_UNMATCHED_DEFAULT,
    KMT_UNMATCHED_INFER_GREEDY,
    KMT_UNMATCHED_INFER_RANGE
} KMT_UNMATCHED;
```

and determines what to do if no actual parameter matches with its formal parameter.

`KMT_UNMATCHED_MISSING` specifies that a missing actual parameter is an error, and outcome `KMT_MATCH_MISSING` should be declared at this formal parameter.

`KMT_UNMATCHED_CURRY_ATEND` specifies that a missing actual parameter indicates that currying should occur at this position, provided that no actual parameters matched with formal parameters to the right (a restriction commonly placed on currying for implementation reasons), and that the matching operation should declare as outcome either `KMT_MATCH_CURRY` or `KMT_MATCH_MISSING`.

`KMT_UNMATCHED_CURRY_ANY` is similar but currying is possible regardless of other positions, and so a missing actual always leads to outcome `KMT_MATCH_CURRY` at this position.

`KMT_UNMATCHED_DEFAULT` specifies that a missing actual parameter indicates that a default value should be used at this position, and outcome `KMT_MATCH_DEFAULT` declared. The default value may be a term expression, which KMT cannot handle directly; instead, its type must appear in the formal parameter's `default_type` attribute, and this is all KMT needs.

Care is needed in finding a suitable value for `default_type` when this policy is used. If the formal parameter is a generic parameter, then presumably it has been declared along with a type expression giving its default value, like *string* in

hash_table: **fun**[*value_type*, *key_type* **dft** *string*] ...

and the user can parse this type expression and pass it, raw, as `default_type`. If the formal parameter is an ordinary parameter, the case is trickier:

rotate: **fun**(*angle*: *real* **dft** 180)

There is no prospect of finding the type of the default expression, here *int*, while interfaces are still being instantiated, except by brutal means, such as requiring the programmer to declare it, or restricting default expressions to be literals. Nor is it safe to pass in the default type later, after type checking the default expression, because that would be to alter the type of an interface after type checking of expressions has begun. The obvious solution here is for the default type to be set to the same value as the upper constraint; although not quite truthful, it is safe. There is no problem with passing the same raw type twice (Section 1.4).

`KMT_UNMATCHED_INFER_GREEDY` and `KMT_UNMATCHED_INFER_RANGE` specify that a missing actual parameter indicates that inference of an actual generic parameter should occur at this position. Accordingly, the matching operation introduces a range type here, leading to outcome `KMT_MATCH_INFERRED`. The two values differ only in that the first sets the `finalizing` attribute of the range type to `KMT_TRUE` initially, while the second sets it to `KMT_FALSE` initially. The second is recommended.

Policy `match_typeop` determines the type check to perform between the type of the actual parameter and the upper constraint of the formal parameter. Its possible values are as for `match_result_typeop`; only `KMT_TYPEOP_ERROR`, `KMT_TYPEOP_EQUAL`, and `KMT_TYPEOP_SUBTYPE` are type-safe. If the check fails, `KMT_PARAMETER_WRONG_TYPE` is the outcome. When the upper constraint is `NULL`, the interpretation is that there is no constraint, so no type check is performed.

The `match_range_policies` policy is given to a range type which replaces a missing actual parameter corresponding to this formal parameter. When `match_unmatched` is neither `KMT_UNMATCHED_INFER_GREEDY` nor `KMT_UNMATCHED_INFER_RANGE` it will not be used and should be `NULL`.

4.4.2. The matching algorithm

This section explains what `KmtCallTypeMatch` does in detail.

As an aid to disambiguating overloaded method calls, `KmtCallTypeMatch` may be called repeatedly on the same call type, using `KmtCallTypeSetHead` between calls to install the next method. Whenever `KmtCallTypeSetHead` is called it ensures that all trace of any previous call to `KmtCallTypeMatch` is removed from the call type, including restoring the original order of the actual parameters and removing any inserted actual parameters (default values and range types). This does not remove side-effects on range types in the types within `call_type`, so if range types are in use, each call to `KmtCallTypeMatch` on the same call type will need to be enclosed in `KmtInferBegin` and `KmtInferEnd`, as explained in Section 5.4.

`KmtCallTypeMatch` checks that `call_type` has a head which is an evaluated function type, and that the actual parameters are all evaluated, aborting if these checks fail. The main algorithm then begins, and consists of the following steps:

1. Match actual parameters with formal parameters, taking account of name, kind, and match style, but not genus and type. First match each named actual parameter, from first to last, with the first remaining unmatched formal parameter with the same name and kind whose match style accepts named actual parameters (i.e. is either `KMT_MATCH_STYLE_NAMED` or `KMT_MATCH_STYLE_POSITIONAL_OR_NAMED`). Then match each unnamed actual parameter, from first to last, with the first remaining unmatched formal parameter with the same kind whose match style accepts positional parameters (i.e. is either `KMT_MATCH_STYLE_POSITIONAL` or `KMT_MATCH_STYLE_POSITIONAL_OR_NAMED`). Each actual parameter that matched is moved so that its index within `call_type` agrees with the index of its formal parameter within the function type; each actual parameter that did not match is moved so that its index within `call_type` is greater than the index of any formal parameter of the function type.
2. For each formal parameter that did not match with an actual parameter, use the formal parameter's `match_unmatched` policy, described above, to decide what to do about this. If the decision is to insert an actual parameter (using either a default type or a fresh, initially unconstrained range type), then do that.
3. Instantiate the function type. First, resolve each typic formal parameter which has a corresponding actual parameter (either present in `call_type` initially, or inserted by the previous step) to the type of that actual parameter. Then copy the upper constraints and result type of the function type, making the substitutions indicated by the resolutions just done. These copy operations do not copy range types; range types are shared. The original types within the function type will not be used from here on; only the copies, called the *instantiated* types, will be used. Finally, remove the resolutions inserted at first.
4. If the call type contains a result type supplied by `KmtCallTypeSetResultType` above, check the instantiated result type from the function type against this type, using the type checking operation specified by the function type's `match_result_typeop` policy.
5. At each index where there is both a formal parameter and an actual parameter, first check that their genera are equal, then (if they are) check their types using the type checking operation specified by the formal parameter's `match_typeop` policy.
6. Finalize any range types introduced earlier.

`KmtCallTypeMatch` increases `KmtCallTypeParameterCount` as required to ensure that every formal parameter and every actual parameter (including unmatched ones) has an index. Its Boolean result indicates whether it considers the match to have succeeded, as explained below.

4.4.3. Diagnosing outcomes

Detailed information about the outcome of a function matching operation may be obtained from the call type object after `KmtCallTypeMatch` returns, via the functions described in this section. These all abort if not called after `KmtCallTypeMatch`.

`KmtCallTypeResultOutcome` reports what happened with the result type:

```
KMT_RESULT_OUTCOME KmtCallTypeResultOutcome(KMT_CALL_TYPE call_type);
```

`KMT_RESULT_OUTCOME` is an enumerated type with the values listed below. An asterisk against a value indicates that KMT considers a match with this result outcome to have failed.

`KMT_RESULT_WRONG_GENUS *`

The call type contained a result type, supplied by `KmtCallTypeSetResultType` above, whose genus differed from the genus of the instantiated function type's result type.

`KMT_RESULT_WRONG_TYPE *`

The call type contained a result type, supplied by `KmtCallTypeSetResultType` above, whose genus was equal to the genus of the instantiated function type's result type, but the operation specified by the `match_result_typeop` policy of the function type failed.

`KMT_RESULT_MATCHED`

Neither of the previous two problems arose.

`KmtCallTypeParameterOutcome` reports what happened at each parameter position:

```
KMT_PARAMETER_OUTCOME KmtCallTypeParameterOutcome(
    KMT_CALL_TYPE call_type, int index);
```

It will abort if `index` is out of range. `KMT_PARAMETER_OUTCOME` is an enumerated type with the values listed below. An asterisk indicates that KMT considers a match with this outcome in any position to have failed.

`KMT_PARAMETER_MISSING *`

There is a formal parameter at this position, but no actual parameter matched with it, and the formal parameter's `match_unmatched` policy is `KMT_UNMATCHED_MISSING`. So the actual parameter is confirmed missing.

`KMT_PARAMETER_NOACCESS`

There is a formal parameter at this position, but no actual parameter is permitted to match with it, because it is not accessible in the current scope. In most cases, formal parameters are as accessible as their enclosing function types, so this case does not arise; if it is allowed, the only reasonable way to handle it is with a private default value.

`KMT_PARAMETER_CURRY`

There is a formal parameter at this position, but no actual parameter matched with it. However, the formal parameter's `match_unmatched` policy indicates that currying is to be used in this case, either because the parameter can always be curried, or because it can be curried when there are no matched parameters after it and that was the case here. So there is still no actual parameter, but it is expected that currying will solve this problem.

`KMT_PARAMETER_DEFAULT`

There is a formal parameter at this position, but no actual parameter matched with it. However, the formal parameter's `match_unmatched` policy indicates that a default value is to be used in this case, so KMT inserted an actual parameter whose type was the default type of the formal parameter.

`KMT_PARAMETER_INFERRERD`

There is a formal parameter at this position, but no actual parameter matched with it.

However, the formal parameter's `match_unmatched` policy indicates that inference of actual generic parameters is to be used in this case, so KMT inserted an actual parameter whose type was a fresh range type with the same genus as the formal parameter.

`KMT_PARAMETER_WRONG_GENUS *`

There is a formal parameter at this position, and an actual parameter supplied by the user. The genus of the actual parameter's type is not equal to the genus of the formal parameter.

`KMT_PARAMETER_WRONG_TYPE *`

There is a formal parameter at this position, and an actual parameter supplied by the user. The formal parameter has an upper constraint, and although its genus (always equal to the genus of the formal parameter) equals the genus of the actual parameter's type, the two types fail to type check, using the operation specified by the formal parameter's `match_typecheck` policy.

`KMT_PARAMETER_MATCHED`

There is a formal parameter at this position, and an actual parameter supplied by the user, and none of the above cases occurred.

`KMT_PARAMETER_LEFTOVER_NAME_UNKNOWN *`

The function type has no formal parameter at this position; this is an actual parameter which did not match with any formal parameter, because it had a name which did not equal the name of any formal parameter with the same kind as the actual parameter and a match style that accepts named actual parameters.

`KMT_PARAMETER_LEFTOVER_NAME_REPEAT *`

Like `KMT_PARAMETER_LEFTOVER_NAME_UNKNOWN`, except that there was at least one formal parameter with the same name and kind and a match style accepting named actual parameters, but all such formal parameters were already matched when this actual parameter's turn came to be matched.

`KMT_PARAMETER_LEFTOVER_POSITIONAL *`

Like `KMT_PARAMETER_LEFTOVER_NAME_UNKNOWN`, except that the actual parameter had no name but all the formal parameters of its kind accepting positional named parameters were already matched when this actual parameter's turn came to be matched.

The following code examines the outcome at each parameter, and is permitted at any time after `KmtCallTypeMatch` returns and before `KmtCallTypeSetHead` resets the call type:

```
for( i = 0; i < KmtCallTypeParameterCount(call_type); i++ )
    switch( KmtCallTypeParameterOutcome(call_type, i) )
    {
        /* one case for each outcome above */
    }
```

`KmtCallTypeParameter(call_type, i)` may be called for any `i` in this range, but will return `NULL` at positions where the outcome indicates that there is no actual parameter.

Two other functions may also be called during this period:

```
KMT_TYPE KmtCallTypeInstantiatedResultType(KMT_CALL_TYPE call_type);
KMT_TYPE KmtCallTypeInstantiatedParameterType(KMT_CALL_TYPE call_type,
        int index);
```

As usual, these abort if `call_type` has not been matched, or if `index` is out of range. They return the elements of the instantiated function type: its result type, and its parameters' upper constraints. For example, if the type of function

flatten: **fun**[*x*](*l*: list[list[*x*]): list[*x*]

is matched with a call type representing the call

flatten[*int*]([[1], [2, 3]])

then `KmtCallTypeInstantiatedResultType` would return the type `list[int]` (not the original result type, `list[x]`), while `KmtCallTypeInstantiatedParameterType` would return `int` for the first parameter and `list[list[int]]` for the second. The instantiated result type will be needed for later type checking, unless top-down type checking is being used (it is the value used as the result type by `KmtTypeEvaluate`, for example); the instantiated parameter types may be needed for code generation. For example, when some subtype relationships are implemented by coercions, it is necessary to compare the types of the actual parameters with these instantiated parameter types, to see whether any actual parameters need to be coerced.

There may be range types in the types returned by these two functions, waiting on a later type operation involving the result type for their resolution. Section 5.4 discusses strategies for ensuring that all range types are eventually resolved.

Chapter 5. Range types

Range types, with syntax

$T..T$

are types whose values are in the process of being gradually determined. They are used during function matching, for inferring the values of actual generic parameters.

5.1. The inference of actual generic parameters

The aim of actual generic parameter inference is to allow the programmer to omit actual generic parameters in function calls, where the information they convey can be recovered from the actual ordinary parameters. For example, in

`cons[real](5.0, emptylist[real])`

both actual generic parameters are deducible from the presence of 5.0; the programmer should be able to write just `cons(5.0, emptylist)`. KMT offers a form of actual generic parameter inference based on the method pioneered by Pierce and Turner [4] and subsequently added to Java. KMT tries to clarify this subject by giving the name ‘range type’ to the key new idea, and differentiating range types from parameter types.

Actual generic parameter inference begins with the insertion of a *range type* in place of the missing actual generic parameter, when the policy of the corresponding formal parameter permits it (Step 2 of the algorithm given in Section 4.4). A range type is a type containing two other types, a *lower constraint*, L , and an *upper constraint*, U . It stands for some type t , not yet finalized, in the range $L \leq t \leq U$ (some type which is a supertype of L and a subtype of U). Both L and U are optional; if either is omitted, the corresponding constraint on t does not apply. If both are absent, t is free to take on any value at all. If both are present, the condition $L \leq U$ must hold; that is, L must be a subtype of U . The syntax above means $L..U$, with both types optional.

A range type may also contain a third type, T . If T is present, then the range type is said to be *resolved to T* ; its value is T , and L and U don’t matter. However, before resolving to T a check is always made that $L \leq T \leq U$, so T does not ride roughshod over L and U .

As type operations such as equality and subtype tests occur, questions are asked of a range type: are you a subtype of this other type, are you equal to it? The range type tries to answer ‘Yes’ to all these questions, but it remembers the commitments it has made, either by *accumulating constraints* (updating L or U or both), or by resolving to some type T . If its constraints forbid the operation, or it does not know what to do (Section 5.3), it answers ‘No’.

In addition to L , U , and T , a range type contains a Boolean *finalizing* flag. When it is set, the range type assumes that the next type operation it undergoes will be its last. This is valuable information, because then the range type does not have to worry about making a choice that fails to work out later: anything that works for that one operation will do. In particular, a range type is more likely to resolve itself to another type when its *finalizing* flag is set.

It is usual to adjust the introduced range types in some way at the end of the matching operation (Step 6 of the algorithm of Section 4.4). At one extreme, KMT could do nothing at all, and let the range types continue to accumulate constraints indefinitely. This is said to lead to type errors far from the point where the range types were introduced. At the other extreme, it could resolve the range types at that time, based on whatever constraints they have accumulated during the match. This fails to say what to do with range types that have accumulated no constraints at all, and it prevents the literal list expression

```
[4, 5.0]
```

from type checking, when it clearly has type *list[real]*.

KMT should probably offer a policy specifying what to do at the end of the match, taking into account what constraints (if any) the range type has accumulated, how many times the range type appears in the result type (this affects how many more type operations it will undergo), and whether top-down type checking was used in the match. All of this is straightforward to implement, but messy to specify, so at present KMT has only one policy (that is, it offers no choice): each range type introduced during function matching has its *finalizing* flag set at the end of the match. Of course, the range type may already be finalizing, or even resolved, by then. This is the author's pet policy. It has many attractive features, including the ability to type check the literal list above; indeed, it will produce type *list[x]* for a manifest list, where *x* is the join of the list element types. Its main deficiency is that when a range type appears more than once in the result type, the *finalizing* flag causes the range type to resolve itself before the constraints on occurrences after the first have been taken into account, occasionally leading to the conclusion that an expression is not well-typed, when in reality it is.

Range types left unresolved by this procedure are usually resolved shortly afterwards, during the type checking of the result type. Still, there is no absolute guarantee of this. For example, in the expression

```
emptylist.length
```

which calculates the length of an empty list, the list element type will remain completely unconstrained. If it is important to ensure that all range types are resolved eventually, then the `KmtInferBegin` and `KmtInferEnd` functions are the answer, as Section 5.4 explains.

5.2. Creation and query

The user is unlikely to ever need to create range types explicitly, since function matching is their main application, and KMT does the creation itself there. Range types are always shared when copying, so a range type in (say) a result type or inherit type would always be wrong. However, for the record, a range type can be created by calling

```
KMT_RANGE_TYPE KmtRangeTypeMake(KMT_BOOLEAN evaluated, void *impl,
    KMT_RANGE_POLICIES policies, wchar_t *name, KMT_BOOLEAN finalizing,
    KMT_TYPE lower_constraint, KMT_TYPE upper_constraint);
```

The first parameter says whether the range type is to be created in an already evaluated state. After that comes the usual implementation pointer, some policies (Section 5.3), an optional name

used only by the display operation (Section 5.6), the finalizing flag described above, and initial values (possibly NULL) for the lower and upper constraints. These may be retrieved, and the implementation pointer may be reset, by calling

```
void *KmtRangeTypeImpl(KMT_RANGE_TYPE range_type);
void KmtRangeTypeSetImpl(KMT_RANGE_TYPE range_type, void *impl);
KMT_RANGE_POLICIES KmtRangeTypePolicies(KMT_RANGE_TYPE range_type);
wchar_t *KmtRangeTypeName(KMT_RANGE_TYPE range_type);
KMT_BOOLEAN KmtRangeTypeFinalizing(KMT_RANGE_TYPE range_type);
KMT_TYPE KmtRangeTypeLowerConstraint(KMT_RANGE_TYPE range_type);
KMT_TYPE KmtRangeTypeUpperConstraint(KMT_RANGE_TYPE range_type);
int KmtRangeTypeGenus(KMT_RANGE_TYPE range_type);
KMT_TYPE KmtRangeTypeResolvedTo(KMT_RANGE_TYPE range_type);
```

The genus of a range type is the value of the genus policy stored in its `policies` attribute, when evaluated. `KmtRangeTypeResolvedTo` returns the type that the range type is currently resolved to, or NULL if it is not resolved. It is important to use this function when traversing a type (Section 1.2), since resolved range types must be taken to be the types they are resolved to, otherwise the integrity of the system is compromised.

5.3. Range policies and failure to infer

In most cases, there is no doubt about the outcome of a type operation even though range types are involved. For example, testing equality of `list[int..]` with `list[real]`, where `int..` denotes a range type constrained below by `int`, clearly must succeed resolving `int..` to `real`.

There are cases, however, where an operation could succeed but there is no best way to make it do so. For example, `int..` can be made a subtype of `..real` in many ways: let x be any type such that $int \leq x \leq real$, and constrain `int..` to `int..x`, and `..real` to `x..real`. But x could prove to be a bad choice later, when these types become involved in other operations. For this reason, KMT does not make these kinds of arbitrary choices unless the `finalizing` flag is set, so that there is no need to worry about their effect on future operations. Instead, in these cases the operation fails but in addition a callback is executed informing the user that there has been a *failure to infer*.

The callback function will have one of these two types:

```
typedef void (*KMT_RANGE_RANGE_FUN)(KMT_RANGE_TYPE range_type1,
    KMT_RANGE_TYPE range_type2);
typedef void (*KMT_RANGE_OTHER_FUN)(KMT_RANGE_TYPE range_type,
    KMT_TYPE other_type);
```

These pass the two types involved, either two range types or one range type and one non-range type. They accept nothing in return, and offer the user no chance to patch up the outcome in any way. They are merely opportunities for printing error messages.

These callbacks are somewhat different from the callbacks that occur during evaluation. Calls on these functions may occur whenever type operations involving range types are going on, which could be during evaluation, but could also be at other times. And they do not cause an error type to be inserted; instead, the operation under way at the time has result `KMT_FALSE`.

The functions are held in a `KMT_RANGE_POLICIES` object, passed as the `policies` attribute of the range type. An object of this type may be created by a call to

```
KMT_RANGE_POLICIES KmtRangePoliciesMake(
    char                genus,
    KMT_RANGE_RANGE_FUN range_subtype_range_failure,
    KMT_RANGE_RANGE_FUN range_join_range_failure,
    KMT_RANGE_OTHER_FUN range_join_other_failure,
    KMT_RANGE_RANGE_FUN range_meet_range_failure,
    KMT_RANGE_OTHER_FUN range_meet_other_failure);
```

The first parameter defines the genus of the range type, and must agree with the genus of the lower and upper constraints, if present. The functions are called when failure to infer occurred when trying to establish a subtype relation between two range types (as in the example above), when trying to join two range types, when trying to join a range type to some non-range type, when trying to meet two range types, and when trying to meet a range type with a non-range type. It may be useful for debugging to distinguish these cases, but for the programmer it is probably sufficient to merely report failure to infer in each case, by means of an error message such as

```
missing generic parameter (cannot infer)
```

A suitable file position to attach to this error message should be obtainable via the implementation pointer of the range type.

5.4. Confirming and cancelling changes

Until now, types have always had fixed values. It is worrying to discover that range types experience side effects during operations. It could be that asking the same question twice might give two different answers, because of these side effects. For example, consider the operation of testing several overloaded function types to see whether they match the actual parameters of a call type. There is a real danger that side effects from one match in the types of the actual parameters will affect another, even if the first match is rejected.

Another issue of practical importance, especially in systems that offer reflection or coercion, is whether all range types eventually do get resolved to a specific type. Some range types, such as the list element type in *emptylist.length*, are inherently indeterminate. Others accumulate some constraints, not enough to fix a definite type, but as long as either *L* or *U* is present the range type can reasonably be resolved to either of them.

Both of these issues are addressed by the `KmtInferBegin` and `KmtInferEnd` functions:

```
void KmtInferBegin();
void KmtInferEnd(KMT_INFER infer, KMT_TYPE type);
```

Calls to these functions must occur in matching pairs. The pairs may be arbitrarily nested, although it will be convenient to first specify their behaviour without nesting, then generalize.

Consider a matching pair of `KmtInferBegin` and `KmtInferEnd` calls, not contained within or containing any other calls to these functions. Let the *introduced range types* be those range types created between these two calls, and let the *altered range types* be those range types

already in existence when `KmtInferBegin` was called whose values changed (that is, there was some change to *L*, *U*, *T*, or *finalizing*) between the two calls.

The only effect of `KmtInferBegin` is to mark the starting point, and so help to define the two sets of range types. The real work is carried out by `KmtInferEnd`. What it does depends on its first parameter, `infer`, whose value has type

```
typedef enum {
    KMT_INFER_CANCEL,
    KMT_INFER_CONFIRM,
    KMT_INFER_CONFIRM_AND_RESOLVE
} KMT_INFER;
```

`KMT_INFER_CANCEL` means that the work done since the matching `KmtInferBegin` is to be undone. Introduced range types don't matter since they will become unreachable, but altered range types must be reset to their values at the time of the matching `KmtInferBegin`.

`KMT_INFER_CONFIRM` means that the work done since the matching `KmtInferBegin` will continue to be used after `KmtInferEnd` returns. `KmtInferEnd` has nothing to do in this case except wave everyone through.

`KMT_INFER_CONFIRM_AND_RESOLVE` means that, in addition to confirming, an attempt should be made to resolve all the range types introduced since the matching `KmtInferBegin` which are not already resolved. Any of these with a lower or upper constraint will be resolved to that constraint; failing that, if the `type` parameter of `KmtInferEnd` is non-NULL, it will be used in cases where its genus is appropriate; and failing that, resolution will not occur. The `type` parameter is used only in this case.

`KmtInferBegin` and `KmtInferEnd` have two main uses. One is to enclose any collection of typing operations that are tentative and might need to be undone, as in this example:

```
KmtInferBegin();
if( KmtTypeEqual(type1, type2) )
{
    /* success so confirm and carry on */
    KmtInferEnd(KMT_INFER_CONFIRM, NULL);
    ...
}
else
{
    /* failure so undo any changes and try something else */
    KmtInferEnd(KMT_INFER_CANCEL, NULL);
    ...
}
```

This is only needed where there are *alternative* typings, to insulate the second alternative against side-effects of the first; if the types must be equal else it is an error, one simply writes

```

if( !KmtTypeEqual(type1, type2) )
{
    Error("types should have been equal");
    ...
}

```

with no need for `KmtInferBegin` and `KmtInferEnd`.

Alternative typings occur when testing overloaded function types against a call type object. When range types are in use, each call to `KmtCallTypeMatch` (Section 4.4) needs to be enclosed in `KmtInferBegin` and `KmtInferEnd`, so that if side-effects are introduced into the actual parameters, they will be removed before the next matching operation begins.

The second main use of `KmtInferBegin` and `KmtInferEnd` is to enclose, say, the entire type checking of a function body:

```

KmtInferBegin();
...
KmtInferEnd(KMT_INFER_CONFIRM_AND_RESOLVE, int_type);

```

This ensures that every range type is resolved to some definite type, as typically needed for code generation, reflection, etc. In this example the user has decided that totally unconstrained types such as the list element type in `emptylist.length` should be arbitrarily assigned type `int`.

When calls to `KmtInferBegin` and `KmtInferEnd` are nested, introductions and changes to range types occurring within any inner pair of calls count for the outer pair if the inner pair ended with a confirmation (since then the inner pair waves them on), and don't count for the outer pair if the inner pair ended in a cancellation (since then the inner pair undoes them). A change confirmed by an inner pair could be cancelled later by an outer pair. If confirmation with resolution is required for more than one genus of type, it can be achieved with nested calls, since each type supplied by `KmtInferEnd` is only applied to range types of its own genus.

Range types may be introduced and altered outside of any pair of `KmtInferBegin` and `KmtInferEnd` calls. There will then be no way to cancel unnested alterations.

`KmtInferBegin` and `KmtInferEnd` have been implemented carefully and have negligible cost. For example, the memory they use is recycled through free lists by `KmtInferEnd`, so memory allocation and deallocation time is virtually zero.

5.5. Evaluation and validity

This section describes how the operation of evaluating a raw type applies to range types, and defines the validity rules for range types.

A raw range type is evaluated by evaluating its lower and upper constraints, if present, then checking the validity rules and mutating the raw range type in place into an evaluated range type. A lexical context is not stored in range types; their lower and upper constraints, if any, are evaluated in the same lexical context as the range type as a whole.

There are two validity rules for range types. First, the lower and upper constraints, when present, must have the same genus as the genus contained in the range type's policy set. Second, if both a lower and upper constraint are present, then the lower constraint must be a subtype

of the upper constraint. The range types inserted automatically during function matching are evaluated range types; they never suffer from validity problems.

When evaluation detects a validity problem, it calls one of the three callback functions related to range types held in its `policies` parameter, as described in Section 1.4. In the syntax that the user would use to declare these functions, they are:

```
void range_lower_wrong_genus(KMT_RANGE_TYPE range_type);
void range_upper_wrong_genus(KMT_RANGE_TYPE range_type);
```

The genus of the lower or upper constraint of `range_type` differs from the genus stored in the policy set of `range_type`.

```
void range_inconsistent(KMT_RANGE_TYPE range_type);
```

This raw range type has both a valid lower constraint and a valid upper constraint, but the lower constraint is not a subtype of the upper constraint.

5.6. Display

The operation for displaying a type, `KmtTypeShow`, was introduced in Section 1.7. This section explains how it applies to range types. For convenience, a function

```
wchar_t *KmtRangeTypeShow(KMT_RANGE_TYPE range_type);
```

is defined which displays a range type, by upcasting and calling `KmtTypeShow`.

If a range type is resolved to another type, that other type is displayed, as it must be, because the meaning is that the range type really is that other type. Otherwise, if a range type appears more than once within the overall type being displayed, then to emphasize the fact that the two occurrences have a shared value, the range type's name is displayed, using the `parameter_fmt` format string. When a range type is inserted by KMT, its name is set to the name of the formal parameter it matches with, so using the `parameter_fmt` format string is appropriate.

Otherwise (if the range type is not resolved and appears exactly once within the overall type being displayed) the `range_fmt` format string of the `KMT_TYPE_FMT` object determines how the range type will be displayed. The range type contains two types, its lower and upper constraints, and these are displayed by the formatting commands `L"%L"` and `L"%U"`. Both are optional, so they may only appear within the second part of a `L"%[...%]"` formatting command. For example, the syntax

```
L"%[%|%L%]..%[%|%U%]"
```

is recommended. It prints the lower constraint if there is one, followed by two dots, followed by the upper constraint if there is one.

5.7. Equality, subtype, join, and meet

A range type is equal to another type if it can be resolved to the other type without violating its lower and upper constraints. In that case, when the question is asked (by `KmtTypeEqual`), the

resolution will actually occur as a side-effect of the operation.

A range type is a subtype of another type if its upper constraint can be reduced to a subtype of the other type without inconsistency; it is a supertype of another type if its lower constraint can be increased to a supertype of the other type without inconsistency. In these cases, these constraint accumulations will actually occur as side effects of the subtype test operation. Analogous but more complex rules apply to joins and meets.

When a range type's *finalizing* flag is set, it uses a somewhat different approach which is more likely to cause resolution to another type than constraint accumulation. There are obscure cases where resolution is possible but constraint accumulation leads to failure to infer. A full description is available elsewhere [2].

Chapter 6. Record types

Record types were introduced in Section 1.1, using the syntax

```
inherit  $R_1, \dots, R_m \{ u_1: T_1, \dots, u_n: T_n \}$ 
```

where R_1, \dots, R_m are optional *parent types*, which must evaluate to record types, and $u: T$ stands for zero or more *components*, each consisting of a *name* (a string) and a *value* (a type). A record type typically represents an entire module or class, except for the name and generic parameters.

Each record type also has an *identity*, which may be thought of as a unique integer kept in the record. When a record type is copied, the copy has the same identity as the original; but two record types derived from different originals have different identities, so can never be equal.

6.1. Creation and query

A record type is created by a sequence of calls beginning with

```
KMT_RECORD_TYPE KmtRecordTypeMakeBegin(void *impl, int genus);
```

followed by any number of calls on these functions, in any order:

```
KMT_PARENT KmtRecordTypeMakeParent(KMT_RECORD_TYPE record_type,  
    void *impl, KMT_TYPE type, char expected_genus_set, KMT_PARENT_KIND kind);  
KMT_COMPONENT KmtRecordTypeMakeComponent(KMT_RECORD_TYPE record_type,  
    void *impl, wchar_t *name, char name_syntax, KMT_BOOLEAN is_typic,  
    KMT_CONTEXT access_context, KMT_ACCESS_LEXICAL access_lexical,  
    KMT_BOOLEAN access_inherit, KMT_TYPE value, char expected_genus_set);  
void KmtRecordTypeDeleteComponent(KMT_RECORD_TYPE record_type,  
    KMT_COMPONENT component);
```

and ending with a call to

```
void KmtRecordTypeMakeEnd(KMT_RECORD_TYPE record_type);
```

As usual, forgetting `KmtRecordTypeMakeEnd` wastes a seriously large amount of memory.

`KmtRecordTypeMakeBegin` begins the construction of a raw record type with a fresh identity. Its parameters are the usual implementation pointer, and the type's genus. These attributes may be queried, and the implementation pointer reset, by

```
void *KmtRecordTypeImpl(KMT_RECORD_TYPE record_type);  
void KmtRecordTypeSetImpl(KMT_RECORD_TYPE record_type, void *impl);  
int KmtRecordTypeGenus(KMT_RECORD_TYPE record_type);
```

`KmtRecordTypeMakeParent` makes a new parent, adds it to `record_type`, and returns it. The parameters are the usual implementation pointer, the parent type (which must evaluate to a

record type), an expected genus set for the parent type (an error callback will occur if the parent type evaluates to a type whose genus is not in this set), and the kind, for which see below. The usual queries are available:

```
void *KmtParentImpl(KMT_PARENT parent);
void KmtParentSetImpl(KMT_PARENT parent, void *impl);
KMT_RECORD_TYPE KmtParentRecordType(KMT_PARENT parent);
KMT_TYPE KmtParentType(KMT_PARENT parent);
int KmtParentExpectedGenus(KMT_PARENT parent);
KMT_PARENT_KIND KmtParentKind(KMT_PARENT parent);
```

and a record type may be queried to find its parents:

```
int KmtRecordTypeParentCount(KMT_RECORD_TYPE record_type);
KMT_PARENT KmtRecordTypeParent(KMT_RECORD_TYPE record_type, int index);
```

counting from 0 as usual. `KmtRecordTypeParent` will abort if `index` is out of range.

The `kind` parameter determines what kind of parent is wanted; its type is

```
typedef enum {
    KMT_PARENT_SINGLE,
    KMT_PARENT_COMPONENTS,
    KMT_PARENT_SUBTYPE
} KMT_PARENT_KIND;
```

`KMT_PARENT_SINGLE` specifies that the parent record type itself is the only thing inherited. This is the ‘single type import’ of Java, not often used. (Import and inheritance have very different meanings at run time, but they are much the same to KMT.) `KMT_PARENT_COMPONENTS` specifies that the components of the parent type are inherited. This is used when importing one module into another. Finally, `KMT_PARENT_SUBTYPE` is like `KMT_PARENT_COMPONENTS`, but in addition a subtype relationship is established. This is used when inheriting classes. Access control (Section 7.5) may limit the visibility of some of the inherited components.

`KmtRecordTypeMakeComponent` makes a new component, adds it to `record_type`, and returns it. Parameter `name` is the component’s name; `name_syntax` is intended to store an enumerated value giving a name syntax (ordinary name, prefix or postfix operator, etc.); it is effectively an additional character in the name. The value of `name` may be `L"`, but it may not be `NULL`. Two components may have the same name and name syntax; they are said to be *overloaded*. Parameter `is_typic` says whether the component is typic (Section 1.3). The next three parameters, `access_context`, `access_lexical`, and `access_inherit`, are concerned with access control and are described in Section 7.5. Parameter `value` is the value of the component, and `expected_genus_set` is a set of genera that `value`’s genus is expected to belong to, otherwise it is an error (Section 1.3).

`KmtRecordTypeDeleteComponent` deletes a component from a record type. It aborts if `component` is not present in `record_type`.

The attributes of a component may be retrieved separately by calling

```

void *KmtComponentImpl(KMT_COMPONENT component);
void KmtComponentSetImpl(KMT_COMPONENT component, void *impl);
wchar_t *KmtComponentName(KMT_COMPONENT component);
char KmtComponentNameSyntax(KMT_COMPONENT component);
KMT_BOOLEAN KmtComponentIsTypic(KMT_COMPONENT component);
KMT_CONTEXT KmtComponentAccessContext(KMT_COMPONENT component);
KMT_ACCESS_LEXICAL KmtComponentAccessLexical(KMT_COMPONENT component);
KMT_BOOLEAN KmtComponentAccessInherit(KMT_COMPONENT component);
KMT_TYPE KmtComponentValue(KMT_COMPONENT component);
int KmtComponentIndex(KMT_COMPONENT component);

```

or all at once by calling

```

void KmtComponentAll(KMT_COMPONENT component, void **impl,
    wchar_t **name, char *name_syntax, KMT_BOOLEAN *is_typic,
    KMT_CONTEXT *access_context, KMT_ACCESS_LEXICAL *access_lexical,
    KMT_BOOLEAN *access_inherit, KMT_TYPE *value, int *index);

```

The index attribute is set automatically; it is 0 for the first component added to the record type, 1 for the second, and so on, not counting inherited components, and ignoring deletions.

Two functions provide for visiting every uninherited component:

```

int KmtRecordTypeComponentTableSize(KMT_RECORD_TYPE record_type);
KMT_BOOLEAN KmtRecordTypeComponent(KMT_RECORD_TYPE record_type,
    int pos, KMT_COMPONENT *component);

```

The components are stored in a linear probing hash table. The first function returns the size of this table, while the second tells whether position `pos` holds a component, and sets `*component` to it if so. To visit every component, use

```

for( pos = 0; pos < KmtRecordTypeComponentTableSize(record_type); pos++ )
    if( KmtRecordTypeComponent(record_type, pos, &component) )
        visit(component);

```

As usual with hash tables, the components will be visited in an essentially random order.

6.2. Renaming

KMT supports *renaming* during inheritance. For example, an inherit clause might say

```
class vip_person inherit person rename address as residence ...
```

specifying that the component known as *address* in class *person* will be referred to as *residence* in class *vip_person*. The rename does not create a new component, it merely changes the name by which an existing component is retrieved. This example is not very convincing, but renaming is one means of overcoming name clashes caused by multiple inheritance, and a change of name may also be desirable when there is a change of interpretation.

By default, parents do not have any renaming. A set of *renames* may be added by

```
void KmtParentRenameBegin(KMT_PARENT parent);
```

followed by any number of calls to

```
KMT_RENAME KmtParentMakeRename(KMT_PARENT parent,
    void *old_impl, wchar_t *old_name, char old_name_syntax,
    void *new_impl, wchar_t *new_name, char new_name_syntax,
    KMT_RENAME_CLASH_FUN old_clash_abort,
    KMT_RENAME_CLASH_FUN new_clash_abort);
```

and finishing with

```
void KmtParentRenameEnd(KMT_PARENT parent);
```

As usual, the penalty for forgetting `KmtParentRenameEnd` is a lot of wasted memory.

`KmtParentMakeRename` accepts the parent, a trio of implementation pointer, name, and name syntax representing the old name (from the parent type), and another trio representing the new name (that the old name is being renamed to). It creates a `KMT_RENAME` object containing these attributes, adds it to `parent`, and returns it; or, if the call is aborted by `old_clash_abort` or `new_clash_abort` (see below), it returns `NULL`. The usual queries are available:

```
void *KmtRenameOldImpl(KMT_RENAME rename);
void KmtRenameSetOldImpl(KMT_RENAME rename, void *old_impl);
wchar_t *KmtRenameOldName(KMT_RENAME rename);
char KmtRenameOldNameSyntax(KMT_RENAME rename);
void *KmtRenameNewImpl(KMT_RENAME rename);
void KmtRenameSetNewImpl(KMT_RENAME rename, void *new_impl);
wchar_t *KmtRenameNewName(KMT_RENAME rename);
char KmtRenameNewNameSyntax(KMT_RENAME rename);
```

There are no operations for visiting all the renames of a parent.

Renaming causes a component in the parent record type with the old name (for convenience, ‘name’ means ‘name plus name syntax’ from here on) to be visible under the new name in the child record type, instead of the old name. This is not as simple as it may sound.

When a call to `KmtParentMakeRename` discovers that the rename it is about to insert has the same old name as an existing rename, it calls the callback function passed as parameter `old_clash_abort`. This has signature

```
typedef KMT_BOOLEAN (*KMT_RENAME_CLASH_FUN)(
    KMT_RENAME incoming_rename, KMT_RENAME existing_rename);
```

and passes back to the user the incoming rename (not yet inserted) and the existing rename (previously inserted) which clash. The user may take the opportunity to print an error message, if desired, and in any case must return a `KMT_BOOLEAN` result which is `KMT_TRUE` if the rename insertion is to be aborted, and `KMT_FALSE` if it is to continue. If it does continue, more callbacks reporting clashes with other existing renames may occur. If the insertion goes ahead, the interpretation is that the component with the old name in the parent record type is accessible via either new name in the child record type. For example,

```
class vip_person inherit person
  rename address as residence, rename address as home
```

if not aborted means that within *vip_person*, the inherited component *address* may be referred to as either *residence* or *home*.

A similar arrangement is made when new names clash: the function passed in parameter *new_clash_abort* is called, with the same parameters, and the same meaning for its return value. Once again, a clash is not necessarily an error. For example,

```
class vip_person inherit person
  rename address as residence, rename telephone as residence
```

if not aborted means that the components *address* and *telephone* from *person* are overloaded in *vip_person* under the name *residence*.

No callback occurs when an old name clashes with a new name:

```
class vip_person inherit person
  rename address as residence, rename residence as home
```

It may look dangerous, but there is in fact no interaction between these renames and no reason to consider them to be a problem.

The meaning of inheritance is that the components of the inherited type are considered to be present in the child type, just as though they had been declared there. Accordingly, the operation of retrieving a component with a given name in the child type searches each parent type as well. When doing that, for each rename with the given name for new name, it searches the parent type for the corresponding old name; and then, if there is no rename whose old name is the given name, it searches the parent type for the given name.

The first part of this, searching for each old name, implements the interpretation that clashing new names indicate that the old names become overloaded. The second part is perhaps surprising. If there is a rename whose old name is the given name, it would be wrong to search the parent type for the given name, since that might find something that, being renamed, it should not find. Conversely, if there is no rename with the given name for old name, then any components of the parent type with the given name have not been renamed and should be found, even if there are also renames with the given name for new name.

KMT makes it easy to prohibit rename clashes within individual parents, but there is no support for global consistency checks, for example to ensure that all components are inherited under at most one name. Suppose record type *D* inherits *B* and *C*, which both inherit *A*, but that *C* renames *x* in *A* to *y*. Then *x* in *A* is inherited into *D* as both *x* and *y*. Assigning a meaning to this is not KMT's problem, but it is a problem for the user.

6.3. Evaluation and validity

The evaluation operation, `KmtTypeEvaluate`, was introduced in Section 1.4. This section explains how it applies to record types.

Evaluation of a raw record type evaluates all the types within the record (all the parent record types and component values) and carries out some major work behind the scenes (the

construction of the ancestor sets described in Section 8.4). The context for evaluating component values is the record type itself, ensuring that the components are visible in each others' values. The context for evaluating parent types is the enclosing context of the record type; the syntax used in this guide makes this difference visible by placing the parent types outside the braces. Evaluating parent types in the context of the record type would produce evaluation cycles.

Record types are one of four kinds of type that may be either raw or evaluated. Unlike the other three (function types, range types, and meet types), record types may not be created in the evaluated state.¹ However, they may be evaluated at any time, even immediately after creation:

```
record_type = KmtRecordTypeMakeBegin(impl, genus);
KmtTypeEvaluate((KMT_TYPE *) &record_type, context, policies);
```

Evaluation may be followed by the insertion and deletion of components (provided they have *evaluated* values, since an evaluated type may contain only evaluated elements), and eventually by `KmtRecordTypeMakeEnd`. But a parent may not be added to an evaluated record type, since the ancestor sets mentioned above would be spoiled by the late arrival of another parent.

The `KMT_EVALUATION_POLICIES` object from Section 1.4 contains four error callback functions related to record types. In the syntax used to declare these functions, they are:

```
void record_parent_not_record(KMT_PARENT parent);
```

The type of `parent` did not evaluate to a record type.

```
void record_parent_wrong_genus(KMT_PARENT parent);
```

The type of `parent` evaluated to a record type, but its genus differs from the expected genus.

```
void record_ancestors_inconsistent(KMT_RAW_RECORD_TYPE record_type,
    KMT_RECORD_TYPE ancestor_type1, KMT_RECORD_TYPE ancestor_type2);
```

Inconsistent genericity: evaluated record types `ancestor_type1` and `ancestor_type2` are ancestors of `record_type` with equal identities and unequal substitutions.

```
void record_component_wrong_genus(KMT_COMPONENT component);
```

The value of `component` is not an error type, but its genus does not lie in the expected genus set of the component. Evaluated record types, being valid, do not suffer from these problems.

6.4. Display

The operation for displaying a type, `KmtTypeShow`, was introduced in Section 1.7. This section explains how it applies to record types. For convenience, a function

```
wchar_t *KmtRecordTypeShow(KMT_RECORD_TYPE record_type);
```

is defined which displays a record type, by upcasting and calling `KmtTypeShow`.

¹This is because an evaluated record type must know its lexical context, to access the formal parameters in scope. These are needed when handling substitutions (Section 1.1).

`KmtTypeShow` does not display the contents of a record type. Instead, it displays a type expression whose value is the record type in question. For example, given a record type `nlist[x]` lying within a `util` module, the record type which is the value of the call expression `nlist[int]` would display as `util.nlist[int]`.

These displays are constructed by following the stored context pointer of the record type (Section 1.4) outwards through the chain of enclosing contexts (function types, components, and record types) until it ends. No format strings are provided for record types. Instead, each component is displayed using the `access_fmt` format string, with its `"%H"` formatting command producing everything in the enclosing context, or being considered not present if the record type enclosing the component has a `NULL` stored context. Each function type is displayed using the format strings for call types, with the `L"%H"` command again producing everything in the enclosing context. This display format requires one actual parameter for each formal parameter in the context. When displaying a raw record type, the names of the formal parameters are used. When displaying an evaluated record type, the corresponding substitutions are used.

Section 1.4 states that contexts are stored during evaluation, suggesting that the above method will fail on unevaluated types, and on types created in evaluated form. However, contexts are also stored whenever one context object is placed directly within another: when a component is added to a record type, when a record type becomes the result type of a function type or the value of a component, and so on. (Such contexts are replaced during `KmtTypeEvaluate` and do not influence evaluation.)

This display method works best when record types appear in the usual places: as component values, and as result types of function types which are themselves component values. It produces what Java calls a ‘canonical type expression’, that is, one with all inheritance (or import) removed. Display of record types appearing in other places is hampered by the fact that no type expression exists which denotes them exactly. This method will produce a misleading result in those cases. In practice the external language is not likely to permit them anyway.

6.5. Equality, subtype, join, and meet

Two evaluated record types are equal if they have the same identity and their substitutions are equal. This is discussed in Section 1.1, where examples are given which show that the outcome differs in several ways from that produced by structural typing.

One evaluated record type is a subtype of another if, when the first evaluated record type’s substitutions are made into its parent record types, one of those parent record types is equal to the other type. There is no way to influence KMT to use anything other than an equality test here. When combined with the general transitivity of subtyping, this implies that an evaluated record type is a subtype of all its ancestor record types, with appropriate substitutions.

Definitions of the join and meet operations follow in the usual way from this definition of the subtype relation. Two evaluated record types have a join if and only if some ancestor of one is equal to some ancestor of the other; the join will be a subtype of each such ancestor. They have a meet if and only if it is not the case that some ancestor of one invokes a record type with the same identity as some ancestor of the other, and the two ancestors have different substitutions. Section 8.4 contains a more detailed explanation.

Chapter 7. Access types and retrieval

Access types were introduced in Section 1.1, using the syntax

$R.u$

The *head* of the access type, ‘ R ’, is optional; when present, it must evaluate to a record type. The *name*, u , is a string.

There are no evaluated access types, only raw ones, because evaluation removes all access types from type expressions. There are no equality, subtype, join, or meet operations for access types, since they apply only to evaluated types. The value of $R.u$ is the value of the component with name u from record type R ; or, if R is absent, it is the value of the component with name u from the nearest lexically enclosing record type containing such a component.

The evaluation of access types is a normal part of the evaluation of type expressions; indeed, all names appearing in type expressions, except for invocations of formal parameters, are the names of access types. But there is another use for evaluating names, in type checking *term* expressions. For example, type checking of the term expression

rocket.launch

must first find *rocket* in the enclosing lexical context, then find *launch* in the type of the value of *rocket*. This operation of searching for a name in a context or type is called *retrieval*, and is also covered in this chapter.

7.1. Creation and query

An access type is created by calling

```
KMT_ACCESS_TYPE KmtAccessTypeMake(void *impl, KMT_TYPE head,
    char expected_genus_set, wchar_t *name, char name_syntax);
```

This accepts the usual implementation pointer, the head, an expected genus set, the name, and a name syntax (Section 6.1). It will abort if name is NULL. These attributes may be queried, and the implementation pointer may be reset, by calling

```
void *KmtAccessTypeImpl(KMT_ACCESS_TYPE access_type);
void KmtAccessTypeSetImpl(KMT_ACCESS_TYPE access_type, void *impl);
KMT_TYPE KmtAccessTypeHead(KMT_ACCESS_TYPE access_type);
char KmtAccessTypeExpectedGenusSet(KMT_ACCESS_TYPE access_type);
wchar_t *KmtAccessTypeName(KMT_ACCESS_TYPE access_type);
char KmtAccessTypeNameSyntax(KMT_ACCESS_TYPE access_type);
```

A head is optional, so head may be NULL; expected_genus_set is unused in this case. If there is a head, then when evaluated its genus must lie within expected_genus_set.

7.2. Evaluation

The evaluation operation was introduced in Section 1.4. This section explains how evaluation applies to access types, at a level of detail sufficient for evaluating type expressions. There is in fact much more to evaluating access types than this. Since this extra material is mainly relevant to retrieval, it appears in Section 7.4 and following sections.

If there is a head, it is evaluated first. If it evaluates to an error type, evaluation returns immediately with an error type for result. Otherwise, if it does not evaluate to a record type with a suitable genus, an error callback is generated, and again evaluation returns with an error type for result. Otherwise, a typic component with the access type's name and name syntax is retrieved from the record type, and the result is the value of that component.

If there is no head, the access type could represent an invocation of a formal parameter as well as of a component, because the two are syntactically indistinguishable, as discussed in Section 1.4. In this case, evaluation searches its `context` parameter, retrieving the nearest typic formal parameter or typic record component with the access type's name and name syntax. The result is either a parameter type invoking the formal parameter, or the component's value.

Retrieval from a record type always includes the appropriate type substitutions, and it always searches inherited record types, taking due account of renaming where present. These issues and many others are covered in Section 7.4.

When evaluation of an access type strikes a problem, it calls one of the callback functions of its `policies` parameter as described in Section 1.4. In the syntax that the user would use to declare these functions, they are:

```
void access_head_not_record(KMT_ACCESS_TYPE access_type);
```

This access type has a non-NULL head which has been successfully evaluated (or was already evaluated), but that head is neither a record type nor an error type. The result is an error type as usual; the head itself is not considered erroneous, rather the access type is in error for using it.

```
void access_head_wrong_genus(KMT_ACCESS_TYPE access_type);
```

As for `access_head_not_record`, except that the head is a record type but its genus is not a member of the expected genus set.

```
void access_name_unknown(KMT_ACCESS_TYPE access_type);
```

Retrieval failed to find either a typic formal parameter or a typic component with the name and name syntax of the access type.

```
void access_name_overload(KMT_ACCESS_TYPE access_type,
    KMT_NAMED named1, KMT_NAMED named2);
```

Retrieval found two or more distinct typic formal parameters or typic components with this access type's name and name syntax; `named1` and `named2` are two examples of such named entities. Although overloading is acceptable in general, it is not acceptable in type expressions. See Section 7.4.2 for type `KMT_NAMED`.

An access type that attempts to retrieve a parameter from its head is always erroneous. For example, given a record type `list[x]` the type expression `list[int].x` is an error. There is no callback

function for this case, because retrieval from record type *list[int]* does not search lexically enclosing scopes, so will not find parameter *x*, which lies in an enclosing function type. Instead, an `access_name_unknown` callback will occur.

7.3. Display

The operation for displaying a type, `KmtTypeShow`, was introduced in Section 1.7. This section explains how it applies to access types. For convenience, a function

```
wchar_t *KmtAccessTypeShow(KMT_ACCESS_TYPE access_type);
```

is defined which displays an access type, by upcasting and calling `KmtTypeShow`.

Each `KMT_TYPE_FMT` object contains just one format string for displaying access types, `access_fmt`. Within it, format command `L"%H"` displays the head, and `L"%N"` displays the name. Since the head is optional it should appear only in the second part of a `L"%[. . . %]"` format command. For example,

```
L"%[% | %H. ]%N"
```

produces the usual format, with the head followed by a dot followed by the name when there is a head, or the name alone otherwise.

7.4. Retrieval

The evaluation of access types is closely related to *retrieval*, the type checking of term expressions which are invocations of formal parameters and the components of record types.

7.4.1. The retrieval operations

Just as most of the work of evaluating a call type is done by the function matching operation, so most of the work of evaluating an access type is done by these two functions, which the user can call directly:

```
void KmtRetrieveFromType(KMT_TYPE type, KMT_CONTEXT context,
    wchar_t *name, char name_syntax, int genus_set, int species_set,
    KMT_RETRIEVE_TEST_FUN retrieve_test_fun, void *impl);
void KmtRetrieveFromContext(KMT_CONTEXT context,
    wchar_t *name, char name_syntax, int genus_set, int species_set,
    KMT_RETRIEVE_TEST_FUN retrieve_test_fun, void *impl);
```

Parameters `name`, `name_syntax`, `genus_set`, and `species_set` tell the two functions what to search for: any *named entity* (formal parameter or record component) with the given name and name syntax, a genus from the given genus set, and a species from the given species set. The symbol `KMT_ANY`, a synonym for `~0`, will match any non-zero genus or species.

In both cases, the result is conceptually a list of all named entities satisfying the conditions of the retrieval. However, instead of returning such a list, the two functions execute `retrieve_test_fun`, a callback function, each time they find one of them. The retrieve test

function (and the `impl` parameter, which is associated with it) is the subject of the next section.

`KmtRetrieveFromType` is used when there is a head, for example when retrieving f in $e.f(2)$; the `type` parameter is the type of e , and must be an evaluated type. The second parameter, `context`, is the lexical context (Section 1.6) within which the entire expression occurs. Although the type is searched, not the context, the context is needed for access control (Section 7.5). This kind of retrieval can only ever find a record component, not a formal parameter. Its behaviour on the various kinds of evaluated types is as follows.

Retrieval from a function type produces a null outcome (i.e. no calls on the retrieve test function). Retrieval from a parameter type is equivalent to retrieval from its formal parameter's upper constraint if there is one, and produces a null outcome otherwise. Retrieval from a record type searches the record type. Retrieval from a meet type retrieves from each of its record types in an unspecified order. Retrieval from an error type produces a null outcome.

Retrieval from a resolved range type is equivalent to retrieval from the type it is resolved to. Retrieval from an unresolved range type with at least one constraint first causes the range type to be resolved, to the lower constraint if there is one, otherwise to the upper constraint, then proceeds as before. (Resolution is necessary in some cases, and doing it whenever possible is at least simple and definite.) Retrieval from an unconstrained range type produces a null outcome.

`KmtRetrieveFromContext` is used when there is no head, for example when retrieving f in $f(2)$; the `context` parameter is both the lexical context in which the expression occurs and the starting point for the search. `KmtRetrieveFromContext` searches for formal parameters in the function types of the context, and components in the record types of the context, from the inside out as usual.

In conformity with the meaning of inheritance, which is that inherited components are to be treated just as though they were defined in the child record, the search of a record type by both functions includes searching its parent record types, and their parent record types and so on. The retrieve test function provides detailed control over this process (Section 7.4.2).

7.4.2. The retrieve test function

A simple way to think about retrieval in the presence of overloading is as a pipeline with two stages: first retrieve all relevant named entities, then test them for applicability and choose the best. Unfortunately, this is usually too simple, since the first stage would return some named entities which are irrelevant because they are overridden or shadowed. It is usually necessary for the user to test the results of a retrieval for applicability while the retrieval is going on.

The `retrieve_test_fun` parameter of the two retrieval functions supports this interleaving of retrieval with testing. It is a callback function, invoked when a named entity is found that has the right name, name syntax, genus, and species. This function has signature

```
typedef void (*KMT_RETRIEVE_TEST_FUN)(KMT_NAMED named,
    KMT_TYPE type, KMT_RETRIEVE_INFO retrieve_info,
    KMT_BOOLEAN *hides_enclosing, KMT_BOOLEAN *hides_inherited);
```

Parameter `named` contains the named entity that has been retrieved. Type `KMT_NAMED` is the abstract superclass of objects that have a name, and hence could be the result of a retrieval. It has two concrete subtypes:

```
KMT_NAMED
  KMT_FORMAL_PARAMETER
  KMT_COMPONENT
```

There are operations for returning the tag, name, name syntax, and the *is_typic* attribute (Section 1.3) possessed by all named entities:

```
KMT_TAG KmtNamedTag(KMT_NAMED named);
wchar_t *KmtNamedName(KMT_NAMED named);
char KmtNamedNameSyntax(KMT_NAMED named);
KMT_BOOLEAN KmtNamedIsTypic(KMT_NAMED named);
```

plus upcasting operations:

```
KMT_NAMED KmtFormalParameterToNamed(KMT_FORMAL_PARAMETER formal);
KMT_NAMED KmtComponentToNamed(KMT_COMPONENT component);
```

and downcasting operations:

```
KMT_FORMAL_PARAMETER KmtNamedToFormalParameter(KMT_NAMED named);
KMT_COMPONENT KmtNamedToComponent(KMT_NAMED named);
```

As usual, C casts may be used if preferred.

The *type* parameter of the retrieve test function is the type of the retrieved entity. If the entity is a formal parameter, it will be a parameter type invoking that parameter; if the entity is a record component, it will be the component's value with appropriate type substitutions.

The *retrieve_info* parameter of the retrieve test function is a pointer to a private struct containing miscellaneous useful information about the retrieval:

```
void *KmtRetrieveInfoImpl(KMT_RETRIEVE_INFO retrieve_info);
KMT_TYPE KmtRetrieveInfoType(KMT_RETRIEVE_INFO retrieve_info);
KMT_CONTEXT KmtRetrieveInfoContext(KMT_RETRIEVE_INFO retrieve_info);
KMT_RETRIEVE_PATH KmtRetrieveInfoPath(KMT_RETRIEVE_INFO retrieve_info);
```

KmtRetrieveInfoImpl returns the *impl* parameter of the original *KmtRetrieveFromType* or *KmtRetrieveFromContext* call, *KmtRetrieveInfoType* returns the original type parameter (which will be non-NULL if and only if the original call was to *KmtRetrieveFromType* and not *KmtRetrieveFromContext*), and *KmtRetrieveInfoContext* returns the original context parameter. *KmtRetrieveInfoPath* returns the retrieve path, explained in Section 7.4.3. The info struct is held in stack memory and lost after the retrieve test function returns, so anything needing to be kept must be extracted from it within the retrieve test function.

The *hides_enclosing* and *hides_inherited* parameters of the retrieve test function are pointers to Boolean variables whose values are undefined at the time of the call. These variables must be assigned values within the body of the retrieve test function, and KMT will abort immediately after it returns if this is not done.

If a call on a retrieve test function returns with **hides_enclosing* set to *KMT_TRUE*, then the search will not proceed to the enclosing lexical context. *KmtRetrieveFromType* does not search lexically enclosing contexts in any case, but **hides_enclosing* must be set anyway.

If a call on a retrieve test function returns with `*hides_inherited` set to `KMT_TRUE`, and a record type is currently being searched, then the parents of that record type will not be searched. Searches of the parents of other record types are not affected. If a function type is being searched, `*hides_inherited` has no effect, but it must be set anyway.

Whatever truncations in the search these flags produce, it remains possible for the omitted areas to be reached along other paths. The flags prevent certain transitions from one area of the search to another; they do not prohibit searching the omitted areas altogether.

It is possible for two or more overloaded components to be found within a single record type, and for two or more overloaded formal parameters to be found within a single function type. In that case, the functions call back the user for all of them, in order of declaration for formal parameters and in an unspecified order for record components, before the search leaves that record or function type. The rules given above for the meaning of `*hides_enclosing` and `*hides_inherited` have been written with this possibility in mind.

Some examples of retrieve test functions appear below. When writing them, the user has two tasks. First, the entity and type need to be saved. Typically, the user would set the `impl` parameter of the retrieve operation to point to the node of the abstract syntax tree on whose behalf this retrieval is being carried out, and store `entity` and `type` into that node. Second, values must be assigned to `*hides_enclosing` and `*hides_inherited`.

Here is a simple example of a retrieve test function that might be useful in a system with no overloading. Type `VX_CALL_NODE` is supposed to be the user's type representing a call expression syntax tree node.

```
void SimpleRetrieveTest(KMT_NAMED named, KMT_TYPE type,
    KMT_RETRIEVE_INFO retrieve_info,
    KMT_BOOLEAN *hides_enclosing, KMT_BOOLEAN *hides_inherited)
{
    VX_CALL_NODE call_node;
    call_node = (VX_CALL_NODE) KmtRetrieveInfoImpl(retrieve_info);
    call_node->named = named;
    call_node->type = type;
    *hides_enclosing = *hides_inherited = KMT_TRUE;
}
```

This function considers any named entity with the correct name, syntax, genus, and species to be applicable, and takes its existence as sufficient justification for not searching any further into lexically enclosing or inherited contexts.

Here is a more elaborate retrieve test function, which accepts a named entity only if its signature matches some function application, in the manner of C++ and Java:

```

void FunRetrieveTest(KMT_NAMED named, KMT_TYPE type,
    KMT_RETRIEVE_INFO retrieve_info,
    KMT_BOOLEAN *hides_enclosing, KMT_BOOLEAN *hides_inherited)
{
    VX_CALL_NODE call_node;
    call_node = (VX_CALL_NODE) KmtRetrieveInfoImpl(retrieve_info);
    KmtRawCallTypeSetHead(call_node->call_type, type);
    if( !KmtRawCallTypeMatch(call_node->call_type) )
    {
        /* does not match, so ignore it and keep searching */
        *hides_enclosing = *hides_inherited = KMT_FALSE;
    }
    else if( call_node->named != NULL )
    {
        /* matches, but so did a previous callback */
        if( call_node->named != named )
            Error("ambiguity");
        *hides_enclosing = *hides_inherited = KMT_TRUE;
    }
    else
    {
        /* matches, so accept and truncate the search */
        call_node->named = named;
        call_node->type = type;
        *hides_enclosing = *hides_inherited = KMT_TRUE;
    }
}

```

This records the first matching named entity and its type, printing an error message if there are others, although not when the same entity is retrieved twice along different paths (Section 7.4.3). If range types are in use, arrangements must be made to reset them between matches (Section 5.4).

When only one named entity is retrieved and its type fails to match, it is good practice to print a detailed diagnosis. This can be done by storing `named` and `type` even when the match fails, and if nothing better comes along, traversing the failed call type as in Section 4.4.

There are many other possible enhancements. For example, concrete entities could be preferred to abstract ones. The `impl` pointers of the retrieved entities should give access to this information on the user side. A realistic retrieve test function is likely to be significantly longer than the one given above.

`KmtRetrieveFromType` and `KmtRetrieveFromContext` can be called from within the retrieve test function. Although deciding whether one named entity is applicable by retrieving others does not seem useful, such calls are handled correctly (that is, independently).

7.4.3. The retrieve path

The *retrieve path* of a retrieved named entity is the path that the retrieval followed to reach that entity, beginning at the `type` or `context` parameter of the retrieve function, and ending at the type that directly encloses the entity passed to the retrieve test function. At a minimum the path contains one element, which is both the `type` or `context` parameter of the retrieve function and immediately encloses the retrieved entity.

Retrievals never search raw types, and error types never lead to calls on the retrieve test function, so the types that may be elements of a retrieve path are function types, parameter types, range types, record types, and meet types. An element of a retrieve path may also be a component or a parent (including an optional rename), making seven kinds of element altogether.

Two retrieve paths are *distinct* if they pass through a different sequence of elements. A retrieve path is *acyclic* if it does not contain two attempts to retrieve the same name and name syntax from the same type. (In the absence of renaming, this just means that the path does not contain the same record type twice.) Retrieval explores all distinct acyclic paths, except when the retrieve test function instructs it to truncate its search, and it calls the retrieve test function whenever it finds a suitable named entity in the course of exploring all these paths.

An example of revisiting a type occurs in the classic diamond, where *D* inherits *B* and *C*, both of which inherit *A*. A retrieval initiated at *D* which finds an *x* in *A* will find it twice, once via *B*, then again via *C*. By stacking *n* diamonds it is possible to construct a case of $3n + 1$ types which cause retrieval to return the same entity 2^n times.

There are rare cases where the search revisits a type because it is inherited by two distinct lexically enclosing types. Consider the following legal Java program:

```
class A
{
    class B extends A { ... }
}
```

A search begun within *B* will visit *A* during the course of searching type *B* and its inherited types, then visit *A* again because it lexically encloses *B*. A private component of *A* (one not accessible via inheritance, but accessible throughout its lexical scope) would be found on the second search of *A* but not the first.

The above examples visit a type twice, but on different paths. Renaming can cause a type to be visited twice on the *same* path. Consider this example:

```
A: inherit B { ... }
```

```
B: inherit A rename x as y { ... }
```

and suppose that *B* contains a component *x*. Then a retrieval of *y* in *B* will succeed, via a path that visits *B* twice. This is not a cycle, because the two visits search for different names.

Occasionally it might be desired to examine the retrieve path leading to a retrieved named entity. The value returned by a call to `KmtRetrieveInfoPath(retrieve_info)` within the retrieve test function represents the last element of this path, and from there it is possible to follow the path all the way back to the start. The following operations are available on objects

of its type, `KMT_RETRIEVE_PATH`:

```
wchar_t *KmtRetrievePathName(KMT_RETRIEVE_PATH rp);
char KmtRetrievePathNameSyntax(KMT_RETRIEVE_PATH rp);
KMT_RENAME KmtRetrievePathRename(KMT_RETRIEVE_PATH rp);
KMT_RETRIEVE_PATH KmtRetrievePathPrev(KMT_RETRIEVE_PATH rp);
KMT_RETRIEVE_PATH KmtRetrievePathCopy(KMT_RETRIEVE_PATH rp);
KMT_RETRIEVE_NODE KmtRetrievePathNode(KMT_RETRIEVE_PATH rp);
```

`KmtRetrievePathName` and `KmtRetrievePathNameSyntax` give the name and name syntax being retrieved at this point (in the absence of renaming they will be constant throughout the path), and `KmtRetrievePathRename` is the rename that the path is currently passing through, or `NULL` if none (only relevant if `KmtRetrievePathNode` has type `KMT_PARENT`).

`KmtRetrievePathPrev` is the previous element of the retrieve path, or `NULL` if `rp` is the first element. `KmtRetrievePathCopy` makes a deep copy of the retrieve path. This, or something like it, will be necessary if it is desired to use the retrieve path after the whole retrieval ends, because the retrieve path pointed to by the `rp` parameter of the retrieve test function is kept in stack memory, for efficiency, and so becomes unavailable after the retrieve test function returns.

`KmtRetrievePathNode` returns the object being searched at this point of the path. Its result type, `KMT_RETRIEVE_NODE`, is an abstract supertype of seven concrete subtypes, one for each of the seven kinds of element that may lie on a retrieve path:

```
KMT_RETRIEVE_NODE
  KMT_FUNCTION_TYPE
  KMT_PARAMETER_TYPE
  KMT_RANGE_TYPE
  KMT_RECORD_TYPE
  KMT_MEET_TYPE
  KMT_PARENT
  KMT_COMPONENT
```

The tag of a retrieve node may be found as usual by

```
KMT_TAG KmtRetrieveNodeTag(KMT_RETRIEVE_NODE rn);
```

and upcasting and downcasting may be done with C casts, or by calling upcasting operations

```
KMT_RETRIEVE_NODE KmtFunctionTypeToRetrieveNode(
  KMT_FUNCTION_TYPE function_type);
KMT_RETRIEVE_NODE KmtParameterTypeToRetrieveNode(
  KMT_PARAMETER_TYPE param_type);
KMT_RETRIEVE_NODE KmtRangeTypeToRetrieveNode(KMT_RANGE_TYPE range_type);
KMT_RETRIEVE_NODE KmtRecordTypeToRetrieveNode(KMT_RECORD_TYPE
  record_type);
KMT_RETRIEVE_NODE KmtMeetTypeToRetrieveNode(KMT_MEET_TYPE meet_type);
KMT_RETRIEVE_NODE KmtParentToRetrieveNode(KMT_PARENT parent);
KMT_RETRIEVE_NODE KmtComponentToRetrieveNode(KMT_COMPONENT component);
```

and downcasting operations

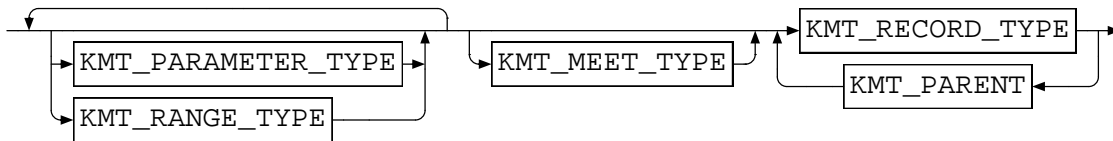
```
KMT_FUNCTION_TYPE KmtRetrieveNodeToFunctionType(KMT_RETRIEVE_NODE rn);
KMT_PARAMETER_TYPE KmtRetrieveNodeToParameterType(KMT_RETRIEVE_NODE rn);
KMT_RANGE_TYPE KmtRetrieveNodeToRangeType(KMT_RETRIEVE_NODE rn);
KMT_RECORD_TYPE KmtRetrieveNodeToRecordType(KMT_RETRIEVE_NODE rn);
KMT_MEET_TYPE KmtRetrieveNodeToMeetType(KMT_RETRIEVE_NODE rn);
KMT_PARENT KmtRetrieveNodeToParent(KMT_RETRIEVE_NODE rn);
KMT_COMPONENT KmtRetrieveNodeToComponent(KMT_RETRIEVE_NODE rn);
```

Since the retrieve path is never empty, it may be traversed by

```
rp = KmtRetrieveInfoPath(retrieve_info);
do
{
    rn = KmtRetrievePathNode(rp);
    switch( KmtRetrieveNodeTag(rn) )
    {
        case KMT_FUNCTION_TYPE_TAG: ...
        case KMT_PARAMETER_TYPE_TAG: ...
        case KMT_RANGE_TYPE_TAG: ...
        case KMT_RECORD_TYPE_TAG: ...
        case KMT_MEET_TYPE_TAG: ...
        case KMT_PARENT_TAG: ...
        case KMT_COMPONENT_TAG: ...
    }
    rp = KmtRetrievePathPrev(rp);
} while( rp != NULL );
```

downcasting `rn` to the appropriate type within each case.

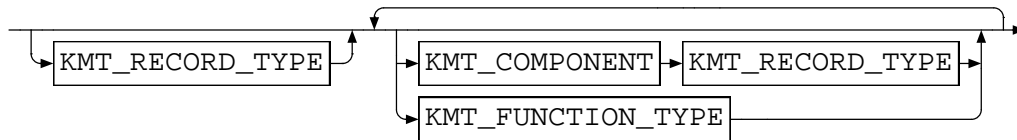
A retrieve path which begins with `KmtRetrieveFromType` and ends with a call on the retrieve test function has this structure:



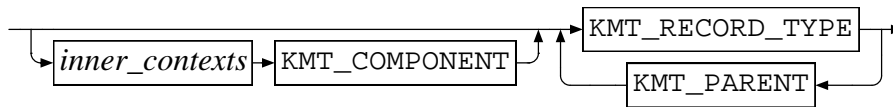
The path may pass through several parameter and range types at first, then possibly one meet type, but it must eventually reach a record type; from there it may pass via parents to other record types, ending with the record type containing the component retrieved.

A retrieve path which begins with `KmtRetrieveFromContext` and ends with a call on the retrieve test function has a somewhat more complex structure. The path may begin by exploring a sequence of inner lexical contexts, none of which is the source of the entity returned by the retrieve test function. This part of the path has this structure:

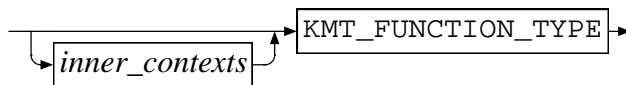
inner_contexts



At each context level, the path either passes through a component and then its enclosing record type, or through a function type; but it may begin at a record type without having previously passed through a component. A path which finds a component has this structure:



It ends with a record type, possibly reached via several parents. Before that it may have searched some inner contexts, in which case it emerges from them through some component. A path which finds a parameter has this structure:



It ends with a function type, and may also have searched inner contexts beforehand.

7.5. Access control

Access control is the term commonly used for language features that restrict access to record components. In KMT terms, access control causes a retrieval operation to not call the retrieve test function when it otherwise would.

There are essentially no design constraints on language features for access control, as examples such as friend functions in C++ [6] and selective export and object-based privacy in Eiffel [3] demonstrate. Stroustrup somewhere gives two guiding principles: access should be granted not taken, and access control should be independent of the semantics of the things accessed. A third is that modules can replace unusual features such as those just listed.

In the absence of further guidance, KMT offers a conservative system consistent with these principles and sufficient to implement the Java access control system. The user who needs other kinds of access control can bypass or supplement KMT's system by implementing them within the retrieve test function, utilizing `retrieve_info` and arbitrary data held on the user side. However, the retrieve test function called when evaluating type expressions cannot be replaced, and this forces KMT's access control to be used within type expressions.

Access control depends on the *retrieved component*, which is the component that is under consideration for passing to the user via the retrieve test function; on the *original context of the retrieval*, which is the `context` parameter of the original call to `KmtRetrieveFromContext` or `KmtRetrieveFromType`; and on the *enclosing record type*, which is the record type immediately enclosing the retrieved component. It also depends on three attributes stored in the retrieved component: `access_context`, `access_lexical`, and `access_inherit`. Section 6.1 explains how to set these attributes.

Attribute `access_context` is a context. It should either be `NULL` or else it should enclose the retrieved component, directly or indirectly. Attribute `access_lexical` has type

```
typedef enum {
    KMT_ACCESS_THROUGHOUT,
    KMT_ACCESS_HERE_AND_AFTER,
    KMT_ACCESS_AFTER,
} KMT_ACCESS_LEXICAL;
```

and `access_inherit` is a Boolean. A component is visible throughout its `access_context`, or has unrestricted visibility if `access_context` is `NULL`, except that `access_lexical` allows its visibility to be restricted to at or after the point of definition (something that is not often wanted in modern object-oriented languages), and `access_inherit` allows its visibility to be extended to all record types which inherit the enclosing record type. More formally:

- (1) Access is granted when `access_context` is `NULL` or the original context of the retrieval lies within `access_context`, except as modified by rule (2).
- (2) When the original context of the retrieval lies within one of the components of the enclosing record type, two exceptions to rule (1) apply: access is denied when `access_lexical` is `KMT_ACCESS_HERE_AND_AFTER` and the original context of the retrieval precedes the component, and when `access_lexical` is `KMT_ACCESS_AFTER` and the original context of the retrieval either precedes the component or lies within its value. These tests are implemented by checking whether there is a node of the retrieve path immediately before the enclosing record type which is a component of the enclosing record type, and if so comparing its index with the index of the retrieved component.
- (3) In addition to (1), if `access_inherit` is `KMT_TRUE` then access is granted if the original context of the retrieval is lexically enclosed in a record type which inherits the enclosing record type directly or indirectly.

These rules support the access modes of Java:

Java mode	<code>access_context</code>	<code>access_lexical</code>	<code>access_inherit</code>
<code>private</code>	Enclosing class	<code>KMT_ACCESS_THROUGHOUT</code>	<code>KMT_FALSE</code>
<code>package</code>	Enclosing module	<code>KMT_ACCESS_THROUGHOUT</code>	<code>KMT_FALSE</code>
<code>protected</code>	Enclosing module	<code>KMT_ACCESS_THROUGHOUT</code>	<code>KMT_TRUE</code>
<code>public</code>	<code>NULL</code>	<code>KMT_ACCESS_THROUGHOUT</code>	<code>KMT_TRUE</code>

but give little insight into the design space of access control overall. Rule (3) seems anomalous when there is a head, because in addition to the main search, the one beginning at the head to find the component, another search, beginning at the original context, may be made to check rule (3). When there is no head, the two searches would follow the same track so the second is omitted.

During evaluation, some parent types may need to be evaluated while the second search is being carried out, just as they would have needed to be evaluated by the main search if there had been no head. Thus, not only can the main search contribute to an evaluation cycle (Section 1.5) during evaluation, checking rule (3) can do so as well.

7.6. Closure parameters

This section and its implementation are under construction

A *local function* is a function whose definition is lexically enclosed within another function definition. Although object-oriented programming languages tend not to offer local functions, many others do, including Pascal and functional languages.

When compiling local functions the problem arises of what to do about invocations within the bodies of these functions of local variables and parameters declared outside them, but not globally. For example, take the Nonpareil function

```
quadratic_roots(a: real, b: real, c: real): list{real} :=
let
  disc := b * b - 4 * a * c
in
  if disc < 0 then
    [ (-b + sqrt(disc))/2*a, (-b - sqrt(disc))/2*a ]
  else if disc = 0 then
    [ -b / 2*a ]
  else
    []
  end
end
```

Suppose we are compiling to C and it is desired to compile *disc* into a separate function. (This is not actually necessary unless *disc* has a parameter.) The function cannot be

```
float disc()
{
  return b * b - 4 * a * c;
}
```

because *a*, *b*, and *c* are undefined.

There are many solutions to this problem, starting with the traditional one of compiling to assembly language and including a ‘static pointer’ in each activation record to the lexically enclosing activation record. The whole question could very well be left to the back end of the compiler. KMT offers a solution, particularly useful when compiling to C, which is based on a clever conversion of the interface of *disc* in the front end, to something like

```
disc(a: real, b: real, c: real)
```

where there is one new parameter for every non-global variable or parameter invoked within *disc* but defined outside it. These added parameters are called *closure parameters*.

A new closure parameter is needed whenever a retrieval originating within the body of *disc* passes through *disc* to a lexically enclosing record and retrieves from that record something that is not a global symbol. KMT adds the new parameter during the retrieval: it notices that the right conditions have occurred, inserts the new parameter, and modifies the result of the retrieval to return the new parameter rather than the original result. The caller of `KmtTypeRetrieve` is not

aware that this has happened.

Subsequent retrievals also find the new parameter, because it is given the same name and now shadows the component it originally intercepted. When functions are nested within functions nested within functions, a new parameter is added to every function on the search path. It may seem to be too clever, but there really are no catches.¹

Each function type contains a policy of type `KMT_CLOSURE_POLICIES` which must be non-NULL in those function types, such as that for *disc* above, which are open to receiving closure parameters. An object of this type is created by calling

```
KMT_CLOSURE_POLICIES KmtClosurePoliciesMake(
    int                genus_set,
    int                species_set,
    KMT_PARAMETER_POLICIES policies
);
```

When a retrieval passes through a function type with a non-NULL value of this policy, it goes on alert. When it retrieves a component, but before passing it to the user for testing, it checks whether the genus and species of the value of the component it has found lie in the `genus_set` and `species_set` of the policy. If so, it adds a new formal parameter to the function type holding the policy, with the same name as the retrieved component, and the `policies` attribute for its policies, and switches to retrieving that new parameter. This is done repeatedly back down the chain of lexically enclosing types.

Careful limitation of the function types containing these policies (to only those representing functions lexically enclosed in other functions), and the kinds of components intercepted (to only those representing local variables or parameters of functions), is needed when using closure parameters. Another point requiring careful notice is that, for simplicity of use, closure parameters are added before there is any confirmation from retrieval testing that the retrieved symbol is really wanted. For the kinds of symbols of interest, this is not a problem.

callback function allowing new parameter to have an implementation pointer still to do effect on function matching still to do

¹Actually, there is a catch. If the function type which is accumulating closure parameters is retrieved before all its closure parameters have been added (which may happen if it is recursive, for example), then that retrieval will not return the correct final type, which is not known yet. Watch this space.

Chapter 8. Meet types

A *meet type* is a set of record types, expressed in Section 1.1 by the syntax

$$R_1 \text{ meet } \dots \text{ meet } R_n$$

where $n \geq 2$ and each *member*, R_i , is a record type. It stands for the largest type which is a subtype of each member. For example, *author meet editor* is the type of authors who are also editors. Even if a language design omits meet types, KMT's type operations produce them occasionally, when multiple inheritance and joins or meets (as used in conditional expressions or the inference of actual generic parameters) are present.

Meet types are an instance of what the literature calls *intersection types*. The name 'meet type' was chosen because of the close connection with the meet operation: the meet of record types R_1 and R_2 , when neither is a subtype of the other, is $R_1 \text{ meet } R_2$ when it exists.

A *meet operation* may be applied to any two types, but the members of a *meet type* may only be record types. Case analysis makes the reason clear. For example, a parameter type in a meet would be constrained in a way that there is no provision for, as consideration of parameter x in

$$\text{list[int] meet } x$$

shows. A function type can only meet with a function type, and in that case the result is another function type, so there is no need for a representation of the meet of two function types.

8.1. Creation and query

A meet type is created by a sequence of calls beginning with

```
KMT_MEET_TYPE KmtMeetTypeMakeBegin(KMT_BOOLEAN evaluated, void *impl);
```

The first parameter says whether the meet type is to be created in an already evaluated state, and will usually be `KMT_FALSE`. After it comes the usual implementation pointer (Section 1.2). A sequence of members is then added by at least two calls to

```
void KmtMeetTypeMakeMember(KMT_MEET_TYPE meet_type, KMT_TYPE member);
```

When an evaluated meet type is being created, each member must be a record type, otherwise `KmtMeetTypeMakeMember` aborts. When a raw meet type is being created, each member may have any type, but an error callback will be generated later if the member evaluates to anything other than a record type. The sequence concludes with

```
void KmtMeetTypeMakeEnd(KMT_MEET_TYPE meet_type);
```

When an evaluated meet type is being created, this performs the validity checks described in Section 8.2 and aborts if they fail, so the user must be certain that the type is valid. Calling these functions out of sequence, calling `KmtMeetTypeMakeMember` less than twice before

calling `KmtMeetTypeMakeEnd`, and attempting to make use of a meet type before calling `KmtMeetTypeMakeEnd`, all cause an abort.

A meet type may be queried, and its implementation pointer reset, by calling

```
void *KmtMeetTypeImpl(KMT_MEET_TYPE meet_type);
void KmtMeetTypeSetImpl(KMT_MEET_TYPE meet_type, void *impl);
short KmtMeetTypeMemberCount(KMT_MEET_TYPE meet_type);
KMT_TYPE KmtMeetTypeMember(KMT_MEET_TYPE meet_type, int index);
int KmtMeetTypeGenus(KMT_MEET_TYPE meet_type);
```

`KmtMeetTypeMemberCount` returns the number of members. `KmtMeetTypeMember` returns the `index`'th member, counting from 0, aborting if `index` is out of range. It is safe to downcast its result to `KMT_RECORD_TYPE` when the meet type is evaluated. `KmtMeetTypeGenus` is available only when the meet type is evaluated; it returns the common genus of the members.

8.2. Evaluation and validity

The evaluation operation, `KmtTypeEvaluate`, was introduced in Section 1.4. This section explains how it applies to raw meet types, and gives the validity rules for meet types.

A raw meet type is evaluated by evaluating its members and carrying out the validity checks explained below. The members may be reordered during evaluation. There is no stored context in a meet type, and its members are evaluated in the same lexical context as the meet type itself.

The validity rules for an evaluated meet type are that it must have at least two members, all of which are evaluated record types with the same genus, and it must be neither inconsistent nor redundant. A meet type is *inconsistent* if it is a subtype of two record types with the same identity but different substitutions. For example,

```
list[int] meet list[real]
```

is obviously inconsistent, but there are also less obvious examples. A meet type is *redundant* if it is expressible using fewer members. For example,

```
list[int] meet list[int]
```

is obviously redundant, but again there are less obvious examples.

Excluding meet types with less than two members imposes no significant restriction. A meet type with one member can only mean the same type as that member, and a meet type with no members can only mean a type which is a supertype of all record types. This 'top' type has some theoretical interest, but since it has no components it is useless in practice.

When evaluation detects a validity problem, it calls one of the callback functions of its `policies` parameter as described in Section 1.4. In the syntax that the user would use to declare these functions, they are:

```
void meet_member_not_record(KMT_MEET_TYPE meet_type, int index);
```

The `index`'th member of `meet_type` did not evaluate to a record type.

```
void meet_member_wrong_genus(KMT_MEET_TYPE meet_type, int index);
```

The genus of the `index`'th member of `meet_type` is not equal to the genus of member 0.

```
void meet_inconsistent(KMT_MEET_TYPE meet_type,
    KMT_RECORD_TYPE record_type1, KMT_RECORD_TYPE record_type2);
```

The members are valid, but the meet type is inconsistent. The two record types `record_type1` and `record_type2` prove this: `meet_type` is a subtype of both, and they have the same identity but different substitutions.

```
void meet_redundant(KMT_MEET_TYPE meet_type, KMT_TYPE alt_type);
```

The members are valid, and the meet type is consistent, but it is redundant. Type `alt_type`, a simpler, equivalent, and non-redundant record or meet type, proves this.

There is no error callback for reporting that a meet type has less than two members. Instead, `KmtMeetTypeMakeEnd` aborts if an attempt is made to create a meet type with less than two members; and if there are less than two after evaluation, that is a case of redundancy.

8.3. Display

The operation for displaying a type, `KmtTypeShow`, was introduced in Section 1.7. This section explains how it applies to meet types. For convenience, a function

```
wchar_t *KmtMeetTypeShow(KMT_MEET_TYPE meet_type);
```

is defined which displays a meet type, by upcasting and calling `KmtTypeShow`.

The `meet_fmt` format string in the `KMT_TYPE_FMT` object affects the display of meet types. Meet types consist of a sequence of members, and within this format string the sequence formatting command `L"%{ ... %}"` displays them. Within its second and third parts, the formatting command `L"%T"` displays one member. For example, setting `meet_fmt` to

```
L"%{|%T%| meet %T%|%}"
```

produces the format used in this document, with the members separated by **meet**. The part of the sequence formatting command that is printed when the sequence is empty will never be used, because the sequence of members of a meet is never empty; but it must be present anyway.

8.4. Equality, subtype, join, and meet

The equality, subtype, join, and meet operations were introduced in Section 1.8. This section explains how they apply to evaluated meet types.

Two meet types are equal if their members are equal; the members' order does not matter. One meet type is a subtype of another meet type if for every member U of the upper type, there exists a member L of the lower type such that $L \leq U$. A meet type is a subtype of a non-meet type if some member is a subtype of that type; a non-meet type is a subtype of a meet type if it is a subtype of every member of the meet type.

The join and meet operations for meet types follow from the subtype operation in the usual way. For background (i.e. what follows is not part of the specification of KMT), here is how joins and meets between record and meet types are found, when the simple cases (types equal, or one a subtype of the other) do not apply. During evaluation, the inherit types of each record type are expanded into the set of all ancestors of the record type, including the record type itself, the record types it inherits, the record types they inherit, and so on, taking care not to include any record type twice. The join of two record types is found by taking the intersection of their ancestor sets, then going back to a record or meet type which has that set of ancestors. The meet of two record types is found in the same way, except that the union of the ancestor sets is taken instead of the join. The same method applies for meet types, except that the ancestor set to start from is the intersection of the ancestor sets of the record types of the meet type.

Ancestor sets are also used to detect inconsistency and redundancy. A record or meet type is inconsistent if, during the construction of its ancestor set, an attempt is made to add a record type when another record type is already present with the same identity and different substitutions. (Equal substitutions are fine; the incoming record type is simply left out.) A meet type is redundant if, when its ancestor set is converted back into the simplest possible meet type, the result has fewer members than the original (possibly just one, making it a record type).

Chapter 9. Error types

An *error type*, mentioned in Section 1.1 and given syntax

?

there, may be inserted wherever a type could not be supplied because of an error. Both the user and KMT may insert error types; KMT performs a user callback just before doing so, allowing the user to record the fact that an error occurred and print an error message.

9.1. Creation and query

An error type is an evaluated type object of type `KMT_ERROR_TYPE`, created by a call to

```
KMT_ERROR_TYPE KmtErrorTypeMake(void *impl, wchar_t *name,
    char name_syntax);
```

Parameter `impl` is the usual implementation pointer. When the error was caused by an access type whose name is not known, `name` and `name_syntax` should hold the failed name and name syntax, otherwise `name` should be `NULL` and `name_syntax` does not matter. Functions

```
void *KmtErrorTypeImpl(KMT_ERROR_TYPE error_type);
void KmtErrorTypeSetImpl(KMT_ERROR_TYPE error_type, void *impl);
wchar_t *KmtErrorTypeName(KMT_ERROR_TYPE error_type);
char KmtErrorTypeNameSyntax(KMT_ERROR_TYPE error_type);
```

retrieve these attributes and reset the implementation pointer. An error type is always valid, but it has no genus, and KMT will abort if an attempt is made to find the genus of an error type.

9.2. Display

The operation for displaying a type, `KmtTypeShow`, was introduced in Section 1.7. This section explains how it applies to error types. For convenience, a function

```
wchar_t *KmtErrorTypeShow(KMT_ERROR_TYPE error_type);
```

is defined which displays an error type, by upcasting and calling `KmtTypeShow`.

`KMT_TYPE_FMT` objects contain an `error_fmt` format string which determines the format used to display an error type. Within the format string, the formatting command `L"%N"` displays the name stored in the error type. This name is optional, so `L"%N"` should only appear within the second part of a `L"%[... %]"` formatting command. For example, setting `error_fmt` to

```
L"?[%| %N%]"
```

prints the name preceded by a question mark if present, or just the question mark otherwise.

9.3. Equality, subtype, join, and meet

The purpose of error types is to allow type checking to continue in a reasonable manner after an error has occurred. Type operations affected by the error should be allowed to succeed, to avoid a cascade of error messages after the first. Hence the following definitions.

An error type is equal to every other type, and it is also a subtype and a supertype of every other type. The join or meet of an error type with any other type is the other type. These rules override all other rules defining the equality, subtype, join, and meet operations.

Chapter 10. Other features

10.1. A Boolean type

The type `KMT_BOOLEAN` used throughout this guide is defined in `kmt.h` to be

```
typedef enum {
    KMT_FALSE = 0,
    KMT_TRUE = 1
} KMT_BOOLEAN;
```

It is the Boolean type that C forgot.

10.2. Memory allocation

KMT maintains a stack of *memory arenas*. Its memory allocations go into the arena on top of the stack. There is no limit to the capacity of an arena other than the usual operating system limit.

If the stack is empty when memory is needed, an arena is created and pushed onto it, allowing the user to ignore memory allocation. Alternatively, the user may influence memory allocation by making, pushing, popping, and freeing arenas:

```
KMT_MEMORY_ARENA KmtMemoryArenaMake();
void KmtMemoryArenaPush(KMT_MEMORY_ARENA arena);
void KmtMemoryArenaPop(KMT_MEMORY_ARENA arena);
void KmtMemoryArenaFree(KMT_MEMORY_ARENA arena);
```

`KmtMemoryArenaPush` aborts if given a `NULL` arena, or if the arena is already on the stack or appears to have been freed. `KmtMemoryArenaPop` aborts if the arena it is given is not currently on top of the stack. `KmtMemoryArenaFree` will abort if arena is `NULL`, or the arena appears to be still on the stack or already freed.

It is up to the user to use these facilities safely. One simple approach is to have a single arena holding the types of all the interfaces used in compiling a single compilation unit, plus a separate arena for analysing the body of each method, by enclosing the code for that as follows:

```
method->arena = KmtMemoryArenaMake();
KmtMemoryArenaPush(method->arena);
... analyse method ...
KmtMemoryArenaPop(method->arena);
KmtMemoryArenaFree(method->arena);
```

Most of the memory consumed by interfaces is needed for the entire compilation, and most of the memory allocated by KMT goes on analysing the bodies of methods, so, simple as this is, it

should use memory efficiently in most cases.

When a KMT operation needs memory, it calls

```
void *KmtMemoryAlloc(size_t nbytes);
```

This returns a pointer to at least `nbytes` bytes of memory, or aborts if the operating system memory limit is reached. The pointer is aligned with type `KMT_ALIGN`, defined to be `long`, but it can be changed (in `kmt.h`) to a wider type. The user is free to use this function.

10.3. Extensible arrays

Also available are four functions which allocate extensible arrays of pointers (or other values of the same width as a pointer):

```
void KmtArrayBegin(void ***array, int *avail_len, int min_len);
void KmtArrayGrow(void ***array, int *avail_len, int min_len);
void KmtArrayEnd(void **array, int *avail_len, int used_len);
void KmtArrayReGrow(void ***array, int *avail_len, int min_len);
```

The first three functions are tuned for both space and time, and should be more efficient than a general-purpose memory allocator. Any number of arrays may be growing at one time.

`KmtArrayBegin` sets `*array` to point to a new extensible array aligned with `KMT_ALIGN`, of length `min_len` at least, and sets `*avail_len` to its length.

`KmtArrayGrow` accepts `*array` and `*avail_len` values as set by a previous call to `KmtArrayBegin`, `KmtArrayGrow`, or `KmtArrayReGrow`, and if `min_len` exceeds `*avail_len` it changes `*array` to point to a new extensible array aligned with `KMT_ALIGN`, of length `min_len` at least, sets `*avail_len` to its length, copies the old array's contents into the new array, zeroes the rest of the new array, and reclaims the old array.

`KmtArrayEnd` reduces `*avail_len` to `used_len` and reclaims the unused memory. It is not possible to call `KmtArrayGrow` after `KmtArrayEnd`.

`KmtArrayReGrow` does what `KmtArrayGrow` does, except that it may be called after `KmtArrayGrowEnd`. It does not reclaim any memory.

For example, suppose there is an object of type `A` which must contain an extensible array of objects of pointer type `B`, among other things. The declaration of `A` would be:

```
typedef struct {
    ...
    B    *b_array;
    int  b_avail;
    int  b_count;
    ...
} *A;
```

The code for initializing an object `a` of type `A` would include

```
a->b_count = 0;
KmtArrayBegin(&a->b_array, &a->b_avail, 0);
```

The code for adding one element is

```
if( a->b_count == a->b_avail )
    KmtArrayGrow(&a->b_array, &a->b_avail, a->b_count + 1);
a->b_array[a->b_count++] = new_b;
```

and the code for finishing off is

```
KmtArrayEnd(a->b_array, &a->b_avail, a->b_count);
```

If `KmtArrayEnd` is omitted, a seriously large amount of memory may be wasted.

`KmtArrayGrow` is not very suitable for enlarging hash tables, because it copies the old array, it does not rehash it. Hash tables can be enlarged by calling `KmtArrayBegin` to obtain a new table without losing the old, rehashing the old into the new, then calling `KmtArrayEnd` on the old table with `used_len` set to 0. KMT also offers hash tables directly (Section 10.5).

Behind the scenes, each arena contains an arbitrary number of large blocks of memory obtained from `malloc`. Whenever a new array is begun or grown, it grabs all of the remaining unallocated portion of the block currently receiving allocations, or all of a new large block. When the array ends, the unallocated portion of it becomes available again. For example, if type objects are being created while parsing a method header within a class within a module within a root raw record, four arrays will be growing behind the scenes (three of them holding hash tables), each occupying the remaining unallocated portion of one large memory block. As each array ends, the unallocated part of it becomes available again. Only a small portion of each block ultimately goes unallocated, and the number of arrays growing at one time is at most the lexical depth. The memory allocator can be profiled by changing

```
#undef KMT_MEMORY_PROFILE
```

to

```
#define KMT_MEMORY_PROFILE
```

in `kmt.h`, and placing a call to

```
void KmtMemoryProfile(FILE *fp);
```

at the end of the main program. This prints a report on memory usage to `fp` in wide characters: the amount received from `malloc`, the amount given to the user, lost to fragmentation, etc. It does not track the effect of `KmtArenaFree`; if that function is in use, the true numbers will all be smaller than shown in the report.

10.4. String construction

KMT uses extensible arrays to hold the type expression strings returned by `KmtTypeDisplay`. The functions which build these strings are available to the user:

```

KMT_STRING_FACTORY KmtStringBegin();
void KmtStringAddChar(KMT_STRING_FACTORY sf, wchar_t ch);
void KmtStringAddInt(KMT_STRING_FACTORY sf, int i);
void KmtStringAddFloat(KMT_STRING_FACTORY sf, wchar_t *fmt, float x);
void KmtStringAddString(KMT_STRING_FACTORY sf, wchar_t *s);
wchar_t *KmtStringEnd(KMT_STRING_FACTORY sf);

```

`KmtStringBegin` returns a ‘string factory’ object which organizes the construction of one wide character string. The next four operations add one character, one integer in format `L"%d"`, one float in format `fmt`, and one string to the end of the factory’s string. The final operation ends the string (including adding the final `L'\0'`) and returns it.

10.5. Symbol tables

KMT takes over the traditional uses for symbol tables within compilers, leaving few reasons to create one directly. Nevertheless, the KMT symbol table module is accessible to the user. The module implements linear probing hash tables which double in size when they reach 80% capacity. It uses KMT’s extensible arrays, which should be more efficient than implementing this kind of table using a general-purpose memory allocator. Care needs to be taken to end the table when no further elements are expected, to release memory as described for extensible arrays.

The module assumes that the entries to be inserted have type

```

typedef struct kmt_table_entry_rec {
    void          *unused;
    wchar_t       *name;
} *KMT_TABLE_ENTRY;

```

where `name` is the name of the entry. The user may cast other types to this if they are pointers to structs whose second word contains their name. The first word has been left free for a tag field.

A symbol table has type `KMT_TABLE`, defined in `kmt.h` to be a struct, not a pointer; it is however passed to the table operations by reference. The fields of the struct are visible in `kmt.h`, but should not be accessed directly. The operations are as follows.

```
void KmtTableBegin(KMT_TABLE *table, int initial_size);
```

Initialize `*table` to size `initial_size` with no entries.

```
void KmtTableInsert(KMT_TABLE *table, KMT_TABLE_ENTRY entry);
void KmtTableDelete(KMT_TABLE *table, KMT_TABLE_ENTRY entry);
```

Insert or delete `entry`. Overloading is permitted; when deleting, the entry must be present or KMT will abort. The table will be enlarged automatically if required when inserting, but not reduced when deleting.

```
void KmtTableEnd(KMT_TABLE *table);
```

Let the table know that no more entries will be inserted. It is important to do this, otherwise a seriously large amount of memory will be wasted.

```
KMT_BOOLEAN KmtTableOpen(KMT_TABLE *table);
```

KMT_TRUE if `*table` is open to insertions, that is, if `KmtTableEnd` has not yet been called.

```
KMT_BOOLEAN KmtTableRetrieve(KMT_TABLE *table, wchar_t *name,
    int *cursor, KMT_TABLE_ENTRY *entry);
```

If `*table` contains an entry with this name, set `*cursor` to the index in the table of the first such entry found, set `*entry` to the entry at `*cursor`, and return KMT_TRUE. Otherwise set `*cursor` to -1, set `*entry` to NULL, and return KMT_FALSE.

```
KMT_BOOLEAN KmtTableRetrieveHashed(KMT_TABLE *table, wchar_t *name,
    int hash_code, int *cursor, KMT_TABLE_ENTRY *entry);
```

Similar, except that a hash code (before reduction modulo the table size) obtained from

```
int KmtTableHash(wchar_t *name);
```

is also passed, allowing it to be calculated only once when retrieving from multiple tables.

```
KMT_BOOLEAN KmtTableRetrieveNext(KMT_TABLE *table, int *cursor,
    KMT_TABLE_ENTRY *entry);
```

Assuming that `*cursor` was set by `KmtTableRetrieve`, `KmtTableRetrieveHashed`, or `KmtTableRetrieveNext`, find the next entry with the same name as the one lying under the cursor, setting `*cursor`, `*entry`, and the return value as for `KmtTableRetrieve`.

`KmtTableRetrieve` and `KmtTableRetrieveNext` make it easy to construct a loop that visits every entry with a given name. This loop is packaged in a macro called as follows:

```
KmtTableForEach(&table, name, &cursor, &entry)
{
    ... visit entry ...
}
```

A variant

```
KmtTableForEachHashed(&table, name, hash_code, &cursor, &entry)
{
    ... visit entry ...
}
```

is also available. Visiting every entry in the table can be done using operations

```
int KmtTableSize(KMT_TABLE *table);
KMT_BOOLEAN KmtTableGet(KMT_TABLE *table, int index, KMT_TABLE_ENTRY *entry);
```

The first returns the table size. The second returns KMT_TRUE if there is an entry at position `index`, and if so it sets `*entry` to that entry. So visiting every entry can be done by the loop

```

for( i = 0; i < KmtTableSize(&table); i++ )
    if( KmtTableGet(&table, i, &entry) )
        ... visit entry ...

```

The entries will be visited in an essentially random order, as usual with hash tables.

10.6. Behaviour on incorrect usage

When KMT is used incorrectly, it prints a brief message beginning with ‘KMT usage error’ in wide characters on `stderr`, for example

```
KMT usage error: attempt to find genus of raw type
```

then calls the C `abort()` function. This guide lists all conditions under which this happens, using phrases such as ‘This function aborts if ...’.

There are many cases where functions cannot do what they are asked to do; they might return `KMT_FALSE`, or invoke a callback function, to indicate that something has gone wrong. These cases typically arise when type checking an incorrect program. This is quite normal, and different from using KMT incorrectly.

When KMT detects that its own implementation is defective, it prints a similar message beginning with ‘KMT internal error’, for example

```
KMT internal error: expected formal parameter here
```

and aborts. Please let me know if you get any of these. I will make fixing them a high priority.

10.7. Testing KMT

Typing `make` or `make kmt` in the main distribution directory compiles KMT but does not produce an executable. Typing `make kmt_test` adds a C source file (`kmt_test.c`) which contains a main program. The resulting executable, `kmt_test`, may be used to test KMT.

Binary `kmt_test` reads the files given as its command-line arguments, and runs the tests in those files. A few files suitable for passing to `kmt_test` appear in the main KMT directory; their names all begin with `test`. File name ‘-’ stands for standard input and causes `kmt_test` to read commands from standard input using its interactive mode. For example, after running `make kmt_test`, Unix command

```
./kmt_test test1 -
```

causes `kmt_test` to first read commands from `test1` and then enter interactive mode; any definitions created by `test1` remain available during the interactive phase. Interactive mode differs from non-interactive mode in that it prints a prompt before each command, but does not echo back the input it receives.

All input files are assumed to be in UTF8. Spaces and newlines are not significant, except that comments begin with ‘#’ and end at end of line.

The testing function maintains a record type, called the root record type, which forms the

context for all the testing. If the first token of the first file read is '{', it is taken to be the beginning of this record type, and it is read in completely and then evaluated. If not, an empty record type is created and evaluated. After this optional record type comes a sequence of zero or more commands beginning with a keyword and ending with a semicolon:

```

c component ; Evaluate component then add it to the root record type
d ; Display the components of the root record type
t type ; Evaluate type and display its value
i type ; Evaluate type and inspect its value (display it in detail)
r name type ; Evaluate type and retrieve name from it, showing all hits
e type type ; Evaluate the types and perform an equality operation
s type type ; Evaluate the types and perform a subtype operation
j type type ; Evaluate the types and perform a join operation
m type type ; Evaluate the types and perform a meet operation
h ; Display the list of available commands (this list)
q ; Quit reading this file

```

If a syntax error occurs, `kmt_test` skips to just after the next semicolon and resumes. Output goes to `stdout`; when printing types, the default print format is used. Evaluation is done using the default policies, which print error messages to `stderr`.

In general, the name of a component is visible within the component's own value (and earlier); but this is not true of components inserted by `c` commands, because their components are not created and added to the root record type until after their values have been evaluated. To get mutually dependent components, place them within an initial root record type as described above.

The `i` command invokes function

```
wchar_t *KmtTypeInspect(KMT_TYPE type);
```

which is available to the user but not documented elsewhere in this guide. For some kinds of types it is the same as `KmtTypeShow`, but for others it produces more detail. For example, when given an evaluated record type it will print its context, parents, ancestor set, and components.

The grammar of types is shown in Figure 10.1. Here is a deterministic version of category `type`, more suited to a recursive descent parser than the version in the figure:

```

type ::=
    typea [ "meet" typea { "meet" typea } ]
typea ::=
    typeb [ ".." typeb ]
typeb ::=
    typec { suffix }
typec ::=
    function_type | record_type | error_type | name | "(" type ")"
suffix ::=
    generic_actuals [ ordinary_actuals ] | ordinary_actuals | "." name

```

```

type ::=
    function_type | call_type | range_type | record_type
    | access_type | meet_type | error_type | "(" type ")"
function_type ::=
    "fun" [ generic_formals ] [ ordinary_formals ] ":" type
generic_formals ::=
    "[" formal_parameter { "," formal_parameter } "]"
ordinary_formals ::=
    "(" formal_parameter { "," formal_parameter } ")"
formal_parameter ::=
    [ name ] [ ":" type ] [ "dft" type ]
call_type ::=
    type [ generic_actuals ] [ ordinary_actuals ]
generic_actuals ::=
    "[" actual_parameter { "," actual_parameter } "]"
ordinary_actuals ::=
    "(" actual_parameter { "," actual_parameter } ")"
actual_parameter ::=
    [ "$" name ] type
range_type ::=
    type ".." type
record_type ::=
    [ "inherit" parent { "," parent } ] "{" { component } "}"
parent ::=
    type [ "rename" "(" rename { "," rename } ")" ]
rename ::=
    name "as" name
component ::=
    [ "type" ] name ":" type
access_type ::=
    [ type "." ] name
meet_type ::=
    type "meet" type { "meet" type }
error_type ::=
    "?"

```

Figure 10.1. The grammar of type expressions used by the testing module. In this figure, `"` encloses literal tokens, `|` separates alternatives, `[]` encloses optional parts, `{ }` encloses parts that may appear zero or more times, and `name` stands for a string of ASCII letters, digits, and underscores, beginning with a letter, and not equal to `fun`, `dft`, `inherit`, `rename`, `as`, `type`, or `meet`. This is just the syntax from Section 1.1 used throughout this guide, only defined more precisely and with some extra parts. As usual, a parameter type may be entered using the syntax for access types; evaluation will distinguish the two cases. Components prefixed by `type` have their `is_typic` flags set. Generic parameters are `typic`, and `match_unmatched` is `KMT_UNMATCHED_INFER_RANGE` if there is no default type, or `KMT_UNMATCHED_DEFAULT` if there is. Ordinary parameters are not `typic`, and `match_unmatched` is `KMT_UNMATCHED_MISSING` if there is no default type, or `KMT_UNMATCHED_DEFAULT` if there is.

References

- [1] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object-Oriented Systems* (1996).
- [2] Jeffrey H. Kingston. KMT Implementation Notes. Tech. Rep. (August 2007).
- [3] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
- [4] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.
- [5] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [6] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley. Second Edition (corrected), 1993.
- [7] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

Index

This index includes the types defined by KMT, but not the functions.

- Abort conditions, 84
- Access control, 69
- Access type, 59
- Actual parameter (of call type), 33
 - inference of *see* range types
- Altered range types, 47
- Alternative typings, 48
- Ancestor set (of record or meet type), 76
- Arena (for memory allocation), 79
- Array (extensible data structure), 80

- Boolean type, 79

- C++, 27, 64, 69
- Call type, 33
- Closure parameter, 71
- Coercion, 17, 43
- Compilation strategies, 14
- Component (of record type), 52
- Conditional expression, 17
- Context, 13
- Contravariance, 27
- Corresponding parameters, 25
- Covariance, 27
- Currying, 38
- Cycle
 - evaluation, 10, 70
 - inheritance, 11, 17

- Default type (of formal parameter), 20, 38
 - variance of, 28
- Display, 14
 - of access types, 61
 - of call types, 35
 - of error types, 77
 - of function types, 24
 - of meet types, 75
 - of parameter types, 31
 - of range types, 50
 - of record types, 57
- Display (*ctd.*)
 - of transient types, 16
- Downcasting, 4

- Eiffel, 28, 69
- `emptylist`, 7
- Equality, 17
 - of error types, 78
 - of function types, 26
 - of meet types, 75
 - of parameter types, 32
 - of range types, 50
 - of record types, 58
- Error messages, 9
- Error type, 77
- Evaluated type, 7
- Evaluation, 7
 - of access types, 60
 - of call types, 34
 - of function types, 23
 - of meet types, 74
 - of range types, 49
 - of record types, 56
- Evaluation cycle, 10
- Evaluation path, 11
- Evaluation policies, 9
- Extensible array, 80

- Failure to infer (range type), 46
- File position, 2, 10, 47
- finalizing* flag (of range type), 44
- Format strings (for display), 15
- Function bodies, 14
- Function matching, 36
- Function type, 20

- Genus, 6
- GNU General Public License, ii

- Hash table, 54, 82

Head (of call type), 33

Identity of record type, 52

Implementation pointer, 1

Import, 53

Inconsistent genericity, 18, 57, 74, 76

Instantiated parameter type, 43

Instantiated result type, 43

Internal error (abort message), 84

Intersection types *see* meet types

Introduced range types, 47

Invariance, 27

`is_typic`, 7

Java, 44, 53, 58, 64

Join, 17

 of error types, 78

 of function types, 29

 of meet types, 76

 of parameter types, 32

 of range types, 51

 of record types, 58

Kind (of parameter), 37

Kingston, Jeffrey H., ii

KMT, ii

`KMT_ACCESS_LEXICAL`, 70

`KMT_ACCESS_TYPE`, 59

`KMT_ACTUAL_PARAMETER`, 33

`KMT_ALIGN`, 80

`KMT_ANY`, 61

`KMT_BOOLEAN`, 79

`KMT_CALL_TYPE`, 33

`KMT_CLOSURE_POLICIES`, 72

`KMT_COMPONENT`, 52

`KMT_CONTEXT`, 13

`KMT_ERROR_TYPE`, 77

`KMT_EVALUATION_NODE`, 11

`KMT_EVALUATION_PATH`, 11

`KMT_EVALUATION_POLICIES`, 9

`KMT_FORMAL_PARAMETER`, 20

`KMT_FUNCTION_TYPE`, 20

`KMT_INFER`, 48

`KMT_MEET_TYPE`, 73

`KMT_MEMORY_ARENA`, 79

`KMT_MEMORY_PROFILE`, 81

`KMT_NAMED`, 62

`KMT_PARAMETER_OUTCOME`, 41

`KMT_PARAMETER_TYPE`, 31

`KMT_PARENT`, 52

`KMT_PARENT_KIND`, 53

`KMT_RANGE_OTHER_FUN`, 46

`KMT_RANGE_POLICIES`, 47

`KMT_RANGE_RANGE_FUN`, 46

`KMT_RANGE_TYPE`, 45

`KMT_RECORD_TYPE`, 52

`KMT_RENAME`, 55

`KMT_RENAME_CLASH_FUN`, 55

`KMT_RESULT_OUTCOME`, 41

`KMT_RETRIEVE_INFO`, 63

`KMT_RETRIEVE_NODE`, 67

`KMT_RETRIEVE_PATH`, 67

`KMT_STRING_FACTORY`, 81

`KMT_TABLE`, 82

`KMT_TAG`, 4

`KMT_TYPE`, 1

`KMT_TYPE_FMT`, 15

`KMT_TYPEOP`, 37

`KMT_UNMATCHED`, 38

`KMT_VARIANCE`, 28

`kmt.h`, ii

`kmt_test`, 84

Leftover formal parameter, 26

Lexical context, 13

Local definition, 14, 71

Lower constraint (of range type), 44

Makefile, ii

`malloc`, 81

Match style, 38

 variance of, 29

Meet, 18

 of error types, 78

 of function types, 29

 of meet types, 76

 of parameter types, 32

 of range types, 51

 of record types, 58

Meet type, 73

Memory allocation, 79

Memory arena, 79

- Module expression, 1
- Multiple inheritance, 18, 54
- `mylist`, 7

- Name of formal parameter, 20
 - variance of, 29
- Name of record component, 53
- `nlist` class, 3
- Nominal typing, 3
- Nonpareil programming language, ii
- `NULL` where type expected, 8

- Operations on types
 - display, 14
 - equality, 17
 - evaluation, 7
 - function matching, 36
 - join, 17
 - meet, 18
 - retrieval, 61
 - subtype, 17
 - supertype, 17
- Overloading, 39, 49, 53, 60, 64

- Parameter type, 31
- Parent type (of record type), 52
- Parsing, 1
- Pierce, Benjamin C., ii, 44
- Policies
 - closure, 72
 - evaluation, 9
 - function type, 21
 - parameter, 22
 - range, 47
- Principle of substitution, 17

- Range type, 44
- Raw type, 7
- Record type, 52
- Redundancy (of meet types), 74, 76
- Reflection, 5, 49
- Renaming, 54
- Resolved value (of range type), 44
- Result type (of function type), 20
 - variance of, 28
- Retrieval, 61
- Retrieve path, 66
- Retrieve test function, 62

- Side effects during type operations, 47
- Single type import, 53
- Species, 22
- Stored context, 13, 58
- String construction, 81
- Stroustrup, Bjarne, 69
- Structural typing, 3
- Subtype, 17
 - of error types, 78
 - of function types, 26
 - of meet types, 75
 - of parameter types, 32
 - of range types, 51
 - of record types, 58
- Supertype, 17
- Symbol table, 82
- Syntax, 1

- Term expression, 1
- `test` subdirectory, 84
- Testing KMT, 84
- Top types, 6
- Top-down type checking, 37
- Transient type, 10
 - display of, 16
- Traversal of types, 5
- Turner, David N., 44
- Type, 1
 - access type, 59
 - call type, 33
 - error type, 77
 - function type, 20
 - meet type, 73
 - parameter type, 31
 - range type, 44
 - record type, 52
 - transient type, 10

- Upcasting, 4
- Upper constraint of formal parameter, 20
 - variance of, 28
- Upper constraint of range type, 44
- Usage error (abort message), 84

UTF8 character encoding, 84

Validity, 8

Variance, 28

VX language, 7