# Nonpareil Implementation Notes

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

*15 September 2005*

## Abstract

This report describes how the Nonpareil compiler compiles Nonpareil to C. It was written mainly to explain the implementation to its author, so it omits many of the easy things and is often informal in style.

## Contents

## 1. Overview

The Nonpareil compiler is written in ANSI C. Its overall design is the subject of this section.

### 1.1. Object-oriented design

The compiler follows a pure (well, almost pure) object-oriented coding style. The usual pattern is for a `THING` type to be declared publicly as a pointer to a `struct` called `thing_rec`, whose implementation is private to a file called `thing.c`, which also contains routines for operations on `THING` objects. Thus, one `.c` file implements one class in the object-oriented sense, with perfect data abstraction. Single inheritance is also used frequently, implemented in the usual way with a tag field at the start of each record, and type casts for obtaining polymorphism.

The overall design has been refactored at least three times. After type checking it was the most difficult thing to get right, fundamentally because of the distance from Nonpareil to C, especially in module structure, object-orientation, and predefined objects.

The terms *front-end* and *back-end* are used frequently in this document. In general, the front end is that part of the compiler which is related to reading and understanding the Nonpareil source, while the back end is that part which is related to generating the compiled code. Some classes are exclusively one or the other; others have aspects of both. The author previously used the terms *view of x* and *x* where he now prefers *front-end x* and *back-end x*, and this older terminology survives in a few places, for example in ‘class views’ and ‘classes’, which would be better termed ‘front-end classes’ and ‘back-end classes’.

The compiler’s classes will be presented in five groups: library classes, expression classes, front-end function classes, back-end function classes, and other classes.

### *1.1.1. Library classes*

*Library classes* implement abstractions that could be used in other projects. They are therefore defined independently of the rest of the compiler, with their own separate `.h` files.

`BOOLEAN` implements a boolean type, missing from C. `ARRAY` is a generic resizable array class. `TRIE` is a trie class, used for storing Unicode character attributes. `ISET` implements sets of integer intervals $\{[a_1, b_1], [a_2, b_2], \ldots\}$. It is used to record which offsets in class records are in use, and also which code numbers in enumerated classes are in use.

The compiler reads Unicode (strictly, ISO10646 UTF-8) source files, accepts arbitrary UTF-8 file names, and writes UTF-8 error messages, but it generates files named in alphanumeric ASCII and containing ASCII C code, creating a need for handling both Unicode and ASCII. `UCHAR` is a Unicode character class, offering access to Unicode character attributes. `USTRING` is a Unicode string class. `SYMTAB` is a generic hash table indexed by Unicode strings. `USTRING_POOL` is a string pool class, for uniqueifying Unicode strings. `ASTRING` offers ASCII string functions, and `ASYMTAB` is a generic hash table indexed by ASCII strings.

`CODEGEN` offers features for generating code; it includes definitions of back-end files, types, functions, and constants. Unlike all the other classes, it is packaged as a record containing pointers to functions, so that one can select dynamically from a range of back-end implementations. At present only one, generating C code, has been written, but it should be possible to add another that generates, say, intermediate code.

### *1.1.2. Expression classes*

*Expression classes* implement the various kinds of expressions found in Nonpareil programs. Most of the compiler is only aware of the abstract supertype of these classes, `EXPR`. Only the expression classes have access to its subclasses, via header file `expr.h` which is included in expression class files but not other class files.

`EXPR_LIT` respresents literals. `EXPR_PAREN` represents expressions enclosed in parentheses. `EXPR_TUPLE`, `EXPR_LIST`, and `EXPR_ARRAY` represent manifest tuples, lists, and arrays. `EXPR_CALL` represents function calls of all kinds. `EXPR_LET` represents **let** expressions, `EXPR_IF` represents **if** expressions, and `EXPR_CASE` represents **case** expressions.

An `EXPR_FNHEAD` expression is inserted at the root of every expression which is the body of a function. It holds any default values for its parameters. Default values are treated by Nonpareil as part of the body of a function, making this the logical place to store them. `EXPR_DEFAULT` expressions are dummy expressions which are inserted at points where actual parameters of functions are missing, and the function is expected to replace the actual parameters with default values.

### *1.1.3. Front-end function classes*

A *front-end function* is a function or variable that may be called from within a Nonpareil expression. Its name is a Unicode identifier. This is as opposed to a *back-end function*, which is a function or variable that may be called from within the generated code, with an ASCII identifier for its name.

For every front-end function there must be a corresponding back-end function which implements the front-end function in the back end. However, this does not mean that front-end functions and back-end functions are in a simple one-to-one correspondence. The leading example where this is not the case is that of features: all the front-end functions defining that feature in various classes share the same back-end function.

In general, front-end function classes reflect the Nonpareil source code, and are concerned mainly with checking the rules of the Nonpareil language, while back-end function classes reflect the code that is to be generated. Given Nonpareil source it is fairly easy to build objects representing front-end functions, and given objects representing back-end functions it is fairly easy to generate code for them; the mapping between them is the complicated part.

There are several variations on the basic idea of a front-end function, which we will come to shortly. These are all represented by objects with a common supertype, called `FEFN`:

```
typedef struct fefn_rec {
  KIND_TAG              kind_tag;
  FILE_POS              file_pos;
  NAME                  name;
  TYPE_VARS             type_vars;
  ARRAY_FEFN_PARAM      parameters;
  TYPE                  result_type;
  BEFN                  befn;
} *FEFN;
```

We see here a tag field for distinguishing subtypes, the position of this function in the Nonpareil source code, the function's name, its signature (type variables, parameters, and result type), and the back-end function that the back end has to call in order to implement a call on this front-end function.

There is no field here holding an `EXPR` representing the body of a front-end function, because the way bodies are defined varies with the different function types. For example, builtin functions have no body, since only calls to them are ever generated.

Expressions are at home in both front-end and back-end functions, and indeed it is common for an expression to be shared between one front-end function and one back-end function. The front-end function controls the *manifesting* of the expression, that is, assigning meaning to the symbols within it, while the back-end function controls the generation of code which implements the expression.

Type `FEFN` is an abstract supertype, i.e. there are no objects of that exact type. It has the following concrete subtypes.

`FEFN_CREATION` is a front-end class creation function. It contains one parameter for each creation feature of its class that is visible in the given module. It also has a body, containing the default values of these parameters.

`FEFN_CREDFT` is a front-end function representing a default value of a creation feature. Since these are inherited, one default value may be used by several creation functions, and to avoid code duplication each is packaged in its own function.

`FEFN_FEATURE` is a front-end function type representing one feature from the Nonpareil source code (not all its redefinitions, just one limb). It contains the signature of that feature insofar as it is visible at that point in the source code, including a `self` parameter added by the compiler, and any body defined there.

`FEFN_PRECOND` is a feature precondition. Like creation feature default values, these may be inherited, so are packaged into functions to avoid code duplication.

`FEFN_BUILTIN` is a front-end function type representing a Nonpareil builtin function. It has no body, since although it may be called it does not need to be defined. `FEFN_LETDEF` is a front-end function type representing a let definition. Anonymous functions are also represented by this type, being converted to let definitions during parsing. `FEFN_DOWNDEF` is a front-end function type representing a downcast variable. `FEFN_PARAM` is a front-end function type representing a parameter.

Finally, `FEFN_INVARIANT` is a front-end function representing one class invariant in one class view (that is, if the class view has an **invariant** keyword, an object of this type will be created holding the invariant expression(s) following the keyword).

### *1.1.4. Back-end function classes*

Back-end functions were defined in the previous section, and distinguished there from front-end functions. In this section the fields of back-end functions are described, and the variants of the basic idea are introduced.

The common supertype of all back-end functions is `BEFN`:

```
typedef struct c_funct_rec {
  KIND_TAG              kind_tag;
  CODEGEN_TYPE          be_type;
  CODEGEN_OBJ           be_obj;
  ARRAY_BEFN            inner_functs;
  ARRAY_BEFN_PARAM      parameters;
  CODEGEN_OBJ           inline_be_obj;
  BOOLEAN               utilized;
  BOOLEAN               cached;
} *BEFN;
```

These fields are somewhat less easy to explain than front-end ones, but here goes.

The `be_type` and `be_obj` fields contain values that are created by the code generation module. They represent the result type of the back-end function and its name, to be passed back to the code generation module during code generation of this function and calls on it. When the back-end function is first created, these fields are given null values. The first phase of code generation visits all the back-end functions and assigns these fields. Earlier passes could do this, but we prefer not to give these passes access to the code generation module, to promote separation of concerns.

If an expression contains a **let** definition with parameters, then a back-end function corresponding to that definition must be generated just before the back-end function containing the original expression. The next field, `inner_functs`, contains all the back-end functions representing **let** definitions lying in expressions lying within this back-end function. Thus, the first step in generating code for this back-end function will be to generate code for all the `inner_functs` functions.

Next comes the list of parameters of the back-end function. These have type BEFN_PARAM, in contrast to the parameters of front-end functions, which have type FEFN_PARAM. As they are identified and inserted, they are sorted so that optional parameters precede compulsory ones.

The Nonpareil compiler leaves most optimizations to the C compiler, since most C compilers do them very well, at least when invoked at the higher optimization levels, and there seems to be little point in re-implementing them, given their complexity. However, to simplify the generated code it does inline two kinds of functions, when not curried: calls on creation features, and calls on noncreation features that are merely wrappers for builtin functions with exactly the same parameters in the same order. In these two cases, inline_be_obj will be non-null and will contain the backend object that is to be invoked in order to carry out inlining. Both types of inlining have the convenient property that the inlined version is just a function call with the same actual parameters as the non-inlined version, so the implementation has to merely use the inline_be_obj field instead of the be_obj field.

The utilized field records whether a call on this back-end function has been code generated so far or not. This could be used to identify unused code, but actually it is there to detect the sequencing problem with let definitions explained in the Language Specification, which occurs when a recursive call on a let definition precedes the first occurrence of a local variable defined outside it.

The cached field is true when calls on this function are to be cached. It is set for features and let definitions, but not for creation functions since object caching does the job for them. When set it causes code to be generated for consulting the function cache before executing the body of the function, and for storing a new cache object holding the computed result just before exit.

The various kinds of back-end function types are now described, together with their relationship with the corresponding front-end function types.

BEFN_CREATION represents the back-end creation function of one class. It has one parameter for every creation feature of the class, unlike FEFN_CREATION objects, which have one parameter for each *visible* creation feature. All the FEFN_CREATION objects for a given class refer to the same BEFN_CREATION object; there is only one creation function for each class in the back end.

BEFN_FEATURE represents the back-end function which implements one feature, including all its dynamically dispatched limbs. All the FEFN_FEATURE objects for a given feature refer to a common BEFN_FEATURE object, which contains a whole set of bodies, one for each redefinition of the feature, and knows which versions are effective in which features, so that code generation can produce the large switch statement that implements the dynamic dispatch.

BEFN_BUILTIN, BEFN_CREDFT, BEFN_LETDEF, BEFN_PRECOND, and BEFN_DOWNDEF are back-end images of front-end functions and are in one-to-one correspondence with the corresponding front-end function objects.

BEFN_PARAM represents a back-end parameter. A front-end parameter always refers to a back-end parameter lying in the back-end function referred to by the front-end function containing the front-end parameter. The back-end parameters referred to by a list of front-end parameters are always distinct. However, this still allows the possibility that the back-end function may have more parameters than the front-end function, and this case does actually occur in creation functions and in features. When a call on a front-end function requires that code for a back-end function call be generated, any such extra parameters are given `EXPR_DEFAULT` actual parameters whose value indicates that, in the body of the back-end function, default values associated with these parameters should be substituted in place of the passed value.

`BEFN_INVARIANT` represents one invariant, currently in one-to-one correspondence with an `FEFN_INVARIANT` object.

Finally come three back-end functions that implement the initialization of predefined objects, so do not correspond with any front-end functions. These are `BEFN_CLASS_INIT`, for initializing the *all_predefined* array or class trie of one class, `BEFN_ENUM_INIT` for initializing the *all_enumerated* array of one enumerated class, and `BEFN_SYSTEM_INIT` for initializing the whole system by making a sequence of calls to instances of the other two functions.

### *1.1.5. Other classes*

'Other classes' denotes the miscellaneous remnant, not classifiable under the previous headings.

`TOKEN` represents one input token, and `FILE_POS` represents a file position (file name, line number, and column). These are defined in *lexer.c*.

`NAME` is the name of something. It contains the name as a Unicode string, its kind (ordinary, infix, infixr, prefix, or postfix), and a variant of the name which is used to distinguish infix from prefix operators in symbol tables.

There is a `NAMED` type, the abstract supertype of all named entities (all types that have a `NAME`). A `CONTEXT` is a stack of symbol tables mapping Unicode strings to named entities, used to work out what entity the source code is referring to when a name appears in it.

`TYPE` represents a Nonpareil type, and `TYPE_VAR` represents a type variable. There are several other related types, described in Section 2. A `COERCION` is a sequence of front-end features which can be used to coerce a value of one type into another.

A `CLASS_VIEW` is the view of a class from a particular module, an image of one source class or class extension, oriented towards implementing the definition of a Nonpareil class as seen from one module. A `CLASS` represents one Nonpareil class in its entirety, and is oriented towards code generation. Each module has one class view for each class that is either defined in that module, imported into it without extension, or imported and extended. Thus, there will in general be many class views of the same class.

A `SYSTEM_VIEW` is the view of the entire system from a particular module, and in fact represents a module within the compiler. A `SYSTEM` object represents the entire system.

## 1.2. The compilation algorithm

The compilation algorithm divides into five stages: *establish*, *validate interface types*, *inherit features*, *validate expressions*, and *generate code*. These stages are described in detail below.

The overall algorithm takes each module in turn and applies the first three stages to it. After that, expressions are validated through the whole system, and finally code generation of the whole system occurs. Expression validation within a module could occur as soon as feature inheritance is complete in that module, but delaying it until after the first three stages are complete in all modules has an efficiency advantage that will be explained later. (Separate compilation is not a goal and would be virtually impossible for Nonpareil.)

It would be too confusing to describe here every activity carried out by every stage. Instead, the main outlines are given here, and later sections describe how the activities required for specialized aspects (invariants, the *clone* feature, and so on) are grafted onto this structure.

### *1.2.1. Establish*

The aim of this stage is to read in one module, parse it, and establish the entities (classes, features, etc.) discovered there. This is done as follows.

Establish a new system view (module). Lex and parse the module file, and establish, within the new system view, class views of all its imported classes, and within those, front-end features for all the imported features, with names reflecting any renaming. While this is going on, build an operator table containing all the operators imported into the module, and use this table to check the module-wide name consistency rule.

Lex the class files of the module, adding to the operator table whenever the lexer notices an operator name definition (easily spotted during lexical analysis, thanks to its **prefix**, **postfix**, **infix**, or **infixr** keyword). Then parse the lexed files, using the operator table to handle operators during the parsing.

It is quite clear during parsing what classes there are, and consequently a class object is established at this time for each new (not imported or extended) class, plus one class view object referring to that class in each module where that class is visible. However, two textual features might turn out to be limbs of the same feature later on, when inheritance is processed. Therefore only front-end features, not back-end features, are established during this stage. One front-end feature is created for each front-end feature imported from another module, and for each front-end feature encountered in the text of the module's class files.

### *1.2.2. Validate interface types*

The aim of this stage is to validate type expressions appearing in the interfaces of the classes and features of a module (everywhere but within expressions) – that is, work out what type variables and classes they refer to, and check their validity. This is carried out as follows.

Class views and front-end features imported into the module have already had their types validated. Thus the following validity checks are carried out only on type expressions appearing within the text of the current module.

Establish Level 1, then Level 2, then Level 3 validity of the types in the inherit clauses of all new classes and class extensions, as described in Section 2. This includes the creation of a parent set for each class, which contains all parent classes, whether appearing in the inherit clause or in the result type of coercion features, together with the coercion, if any, required to reach that parent class. These parent sets are then subjected to a reflexive transitive closure operation to find the ancestor sets. In the parent sets, the coercions are either null or else consist of just the one coercion feature; but the reflexive transitive closure operation builds longer chains of coercions for the ancestor set.

The unvalidated result types of coercion features are copied before insertion into the parent set, to avoid the awkwardness of having the result types of some features instantiated and others not.

Establish the validity of all type expressions appearing in the interfaces of features (parameter types and return types). These can be validated directly through all the levels, given that class types are now validated completely.

*1.2.3. Inherit features*

The aim of this stage is to ensure that the class views of one module contain all the front-end features they should. Before this point, class views contain only the non-inherited front-end features, either imported from other modules or defined in classes and class extensions of the current module. No inherited front-end features are present, coerced or otherwise, even from other modules, and no back-end features have been created, only front-end features. For a given module, feature inheritance is carried out as follows.

Inject into every class view in the current module, imported or otherwise, all front-end features that belong in it owing to inheritance, but not coercion. The class views are visited in top-down topological order, so that parent classes are visited before child classes. This is possible at this point since the inheritance hierarchy is acyclic if classes inherited with coercion are ignored. Top-down order ensures that this stage will have been completed on the parent of every class view before it is begun on that class view.

It is important to visit parents rather than ancestors during this stage. Otherwise there will be a problem with the case of three class views, *a*, *b*, and *c*, such that *b* inherits *a* and *c* inherits *b*. The problem is that a front-end feature defined in *a* and redefined in *b* will appear to have inconsistent definitions in *c* if ancestors are consulted; consulting parents will not give this problem, because the 'conflict' will be resolved in *b* and hence not affect *c*.

For each parent, then, take all front-end features currently in the parent (and these will be all front-end features known in this module that are either defined directly in this class view or inherited into it without coercion) and inject them into the current class view. Each front-end feature's signature receives the appropriate type variable substitution during this copy, based on information stored in the parent set.

If a front-end feature is injected that shares a name with a front-end feature already present in the class, the usual checks are carried out: they must have the same base front-end feature, etc. After injection of all front-end features from all parents, it is clear whether each front-end feature from the text of the class is a base feature or redefinition (and in that case what its base front-end feature is). Create a feature for each non-imported front-end feature that is not a redefinition.

It remains to inject into the class view those front-end features that are made available to it via coercions of the target. When dealing with coercion it is not possible to visit parents before children, because there may be (and usually are) cycles in the inheritance graph when coercion is included. The only practical alternative is to visit all the proper ancestors of each class, rather than its parents. Since ancestors reachable without coercion have already been done, this stage visits every proper ancestor reachable by a non-empty coercion. To ensure that redefinitions of features are preferred where available, the proper ancestors are visited in order of children before parents where possible – that is, in reverse sort order.

For each ancestor class view that is reached by a non-empty coercion, take all of its original front-end features (those appearing in the class as a result of import or definition, not inheritance) and inject them into the current class view, with the coercion needed to reach the ancestor class attached.

Because the ancestor set contains every class that is reachable by any combination of inheritance and coercion, this traversal will reach every original feature that is in the current class by virtue of any such inheritance path containing a non-null coercion. The same method could have been used for non-coercion inheritance, were it not for the need to distinguish between inheriting directly from a parent and inheriting indirectly from further up, as described earlier.

### *1.2.4. Validate expressions*

The aim of this stage is to validate expressions. An expression could be a feature body, parameter default value, precondition, or class invariant. Make a single traversal over the expression, working out the type of each subexpression in postorder. When interpreting names that appear in the expression, utilize a name context that initially contains the class names in scope plus the features of the current class. As let definitions etc. are entered, expand the context to include their names, and shrink it again as they are left.

To reduce the number of traversals of expressions, three kinds of preparation for code generation are also made during this stage. These preparations do not require access to the actual back end module – that is, they do not involve back end types, file names, etc. The first is deciding whether each subexpression is large-scale (requires implementing with C statements) or small-scale (can be implemented with a C expression); the second is placing let definitions with parameters into the `inner_functs` lists of their enclosing backend functions, to be code generated first, producing static helper functions.

The third preparation is to sort lists of actual parameters into the correct order for code generation. In principle, one could sort them according to the signature they have to conform with, and do all checks necessary for validity without ever consulting the corresponding backend function; but that would amount to sorting them once according to the signature, and again later according to the backend function. For efficiency they are sorted just once, directly into code generation order. This prevents this preparation step from being done until the parameters of all backend functions are finalized, which in general is not until all modules have been read: the parameters of a creation function depend on all extensions of the corresponding class, and the parameters of a feature depends on all the parameters declared in all redefinitions of that feature in all modules. This is why expression validation is delayed until after the first three stages are complete in all modules: only then are the parameters of the backend functions finalized.

This stage cannot precede feature inheritance, because it relies on class views containing all their front-end features, when validating calls on targets.

### *1.2.5. Generate code*

The generated code consists mainly of a type tag, record declaration, creation function, and optional invariant function for each class, and one function for each noncreation feature, usually holding a large switch on the type tag of the *self* parameter. The switch has one limb for each redefinition, preceded by a set of case labels, one for each non-abstract class in which that redefinition applies.

All generated functions may be preceded by static helper functions, needed to implement let expressions. Since these helper functions could be mutually recursive, when one let expression is nested inside a definition appearing in another, any list of two or more static helper functions associated with some function is preceded by a list of its own prototypes, just in case mutual recursion makes that necessary.

Generating this code is reasonably straightforward off the class and feature objects, which are specifically designed to bring together all the fragments scattered through the modules that together define one class or one feature. The compiler generates a single `typedefs.h` file containing all the typedefs, type tags, and structs, plus one file per module containing one section for each class originally defined in that module. The section contains the class's creation function, class swizzle, unswizzle, and invariant functions, plus any features originally defined in that class. The file as a whole ends with a few module-level functions, for swizzling, unswizzling, and initialization of predefined objects.

Code generation begins with a rather extensive initialization phase. This initializes the fields of each class related to the back end, such as the backend type and type tag names of the class, the names of the swizzle and unswizzle functions, and so on. Then it initializes the backend variables of the creation features of each class, principally their backend names, types, and offsets in the class struct. Finally it initializes the backend types and names of all backend functions (creation functions, noncreation features, class invariants, and their helper functions). Only backend types, not names, are assigned to backend parameters at this stage. Builtin functions also contain backend functions, but they are very simple and are initialized when the builtin functions are set up initially.

Phase 2 of code generation actually generates the code, including assigning backend names to parameters, and backend types and names to local variables in inner blocks. Delaying the assignment of backend names of parameters to this point simplifies ensuring that all local names are unique. A consequence of this is that when printing forward headers for possibly mutually recursive helper functions, parameter names are not available; fortunately C does not require parameter names to appear in these headers. Header (`.h`) files are generated in parallel with code files.

## 2. Implementing the type system

This section explains in detail how types are represented in the Nonpareil compiler, and how the various operations are implemented using this representation.

The definitions in this section will naturally be very concrete. Rather than defining 'concrete type', 'concrete variable,' and so on, we will re-use the existing terms with the understanding that the definitions given here apply in this section only.

### 2.1. Representing types and type variables

A *type* is a globally visible typedef which is a pointer to a *type record*:

```
typedef struct type_rec *TYPE;
typedef ARRAY(TYPE) ARRAY_TYPE;
```

The second typedef uses a macro called *ARRAY* to define a type *ARRAY_TYPE* representing a pointer to an array of types.

The type record itself is private to the type module. It is a tagged record containing either a variable invocation or else a pointer to an array of one or more *ctypes*:

```
struct type_rec {
  KIND_TAG      kind_tag;
  FILE_POS      file_pos;
  union {
    TYPE_VAR    variable;
    ARRAY_CTYPE ctypes;
  } u;
};
```

*CTYPE* is a private typedef (not known outside the type module) which represents one class type. It is a pointer to a *class type record*:

```
typedef struct ctype_rec *CTYPE;
typedef ARRAY(CTYPE) ARRAY_CTYPE;
```

A class type record contains an indication of where this class type occurred in the input, a pointer to the class being invoked (or just a name before the type has been instantiated), and an optional array of types, the actual generic parameters of the class type:

```
struct ctype_rec {
  KIND_TAG      kind_tag;
  FILE_POS      file_pos;
  USTRING       uninstantiated;
  CLASS         class;
  ARRAY_TYPE    generics;
};
```

These definitions follow exactly the options offered by the syntax of types.

There is a publicly visible typedef, *TYPE_VAR*, representing a type variable:

```
typedef struct type_var_rec *TYPE_VAR;
typedef ARRAY(TYPE_VAR) TYPE_VARS;
```

When such a pointer appears in a type record, as above, it represents a variable invocation, which is a type. When it appears in other contexts it may represent the variable itself, which is not a type. For example, a class record contains an array of its formal generic parameter type variables, of type *TYPE_VARS*; when accessed in this way (for example, to make a substitution) they are type variables, not types. A *variable record* is defined as follows:

```
typedef enum { Ordinary, Unify, Range } VAR_STATE;

struct type_var_rec {
  KIND_TAG      kind_tag;
  FILE_POS      file_pos;
  NAME          name;
  int           occurrences;
  int           swizzle_index;
  TYPE          resolved_to;
  TYPE          lower_constraint;
  TYPE          upper_constraint;
  VAR_STATE     state;
};
```

The kind tag marks this object as a type variable when it is stored in contexts. Then comes its file position and name, for error message printing only. The *occurrences* field is a temporary variable, used to calculate how many occurrences of this variable there are in a type, when implementing the type printing algorithm described in the appendix of the Language Specification which presents types in detail.

The `swizzle_index` field gives the index of the swizzle bit used in object records to indicate whether creation features of this variable type are actually pointers or not.

When non-null, the *resolved_to* field holds a type, and the meaning is that this variable has been resolved to this type. The variable is effectively defunct in this case: all invocations of it are interpreted to mean the resolved-to type, and no other fields apply. However, it is common for the `resolved_to` field to be subsequently cleared, making the other fields relevant again, and so their values must be preserved while the `resolved_to` field is set.

The `resolved_to` field is used to implement substitution. For example, given the type

*nlist*{*x*}

where *x* is a variable, if the `resolved_to` field of *x* is set to some type, say *list*{*int*}, then all algorithms on types will perceive the type to be

*nlist*{*list*{*int*}}

because they all follow `resolved_to` fields as a first priority. They may chain through several resolved variables before reaching an unresolved variable or class or meet type.

Substitutions are rarely permanent, so a typical algorithmic pattern is to set the `resolved_to` fields in some variables, carry out some operation, then clear the fields again. If the operation is a type copy those variables will be eliminated from the copy, being replaced wherever they occur by copies of the substituted values. ‘Substitute and copy’ as we call this pattern is the only way to effect a permanent substitution. With care, then, we may treat types as immutable objects, and share them rather than copy them, on the understanding that any substitutions will be strictly temporary, and will be undone before any other operation gains access to any type containing the resolved variables. If we do need a permanent version of a type containing substitutions, we must obtain it by substitute and copy.

When a variable is not resolved, the three other fields of its record have a currently active meaning. The *upper_constraint* field, if non-null, contains a type which is an upper constraint on the value of the variable. If null, there is no upper constraint. Similarly, the *lower_constraint* field records an optional lower constraint on the variable. The *state* field records the state of the variable, and may have one of three values:

*Ordinary.* The variable is an ordinary variable, such as parameter *x* of the *nlist*{*x*} class.

*Unify.* The variable is a unification variable, an invocation of which has been introduced in place of an absent actual generic parameter, and its upper and lower constraints are currently open to modification.

*Range.* The variable is a type range, an ex-unification variable whose optional upper and lower constraints are now fixed.

An *Ordinary* variable remains ordinary for life, apart from brief interludes when it is temporarily resolved. A *Unify* variable eventually either gets resolved or becomes a *Range* variable; and a *Range* variable eventually gets resolved.

### 2.2. Weighted ancestor sets

A class record, pointed to by the *class* field of a ctype, contains various items of information about a class, including its name, its features, and so on. The *parent set* of a class record is a set of class types, the union of the inherit clause and the result types of its coercion features (it holds the → relation). The parent set is just a stepping stone to the calculation of the ancestor set. The *ancestor set* of a class record is another set of class types, holding all the ancestors of the class (the ≤ relation). Eventually the class types of the ancestor set will appear in sorted order, but this is of course not possible until that ordering is established.

Actually, to say that parent sets and ancestor sets are sets of class types is somewhat simplified. In fact, they have type *ARRAY_ITYPE*, where *ITYPE* is a pointer to another private record:

```
typedef struct itype_rec *ITYPE;
typedef ARRAY(ITYPE) ARRAY_ITYPE;

struct itype_rec {
  CTYPE         ctype;
  int           weight;
  COERCION      coercion;
};
```

The *ctype* field points to the ctype that is this element of the parent set or ancestor set. The *weight* field is an integer weight assigned to the ctype, used to bias the algorithm for converting ancestor sets back to types as described below. Every class ancestor set naturally contains an itype for its own class, and that itype has weight 1 while all the others have weight 0. Algorithms given later for finding the union and intersection of ancestor sets, and mapping ancestor sets back to types, will use these weights. Finally, the *coercion* field records the sequence of coercions needed to reach this class type. For example, the ancestor set of class *int* contains *real_ref*, and the *coercion* field of its itype in the ancestor set would contain a coercion representing the sequence *to_real*.*to_real_ref*.

The class types in parent sets and ancestor sets may contain invocations of the type variables of the class, which are also accessible directly by the *type_vars* field. To obtain the ancestor set corresponding to a particular class type, the algorithm is to substitute actual values for the type variables of the class, copy the ancestor set, then remove the substitutions. For example, the ancestor set of *nlist*{*x*} might be

$$\{nlist\{x\}^1, list\{x\}^0, object^0\}$$

where the superscripts represent weights and the coercions are not shown. To obtain the ancestor set of, say, *nlist*{*int*} we use our access to the variables of the class (here just *x*) to set the *resolved_to* field of *x* to *int*, then copy the ancestor set, producing

$$\{nlist\{int\}^1, list\{int\}^0, object^0\}$$

which is the ancestor set of *nlist*{*int*}.

The class also contains two integers which define a standard ordering of classes. The *eq_class_id* field is an integer identifying the equivalence class of the class. These integers are assigned in topological order from the top down, so the *eq_class_id* of *real* will be smaller than that for *int*, and so on. The *eq_class_offset* field is a small integer identifying an arbitrary but unique position for the class within its equivalence class. For example,

$$\{int, int\_ref\}$$

is an equivalence class, so *int* and *int_ref* will have the same values for *eq_class_id* and arbitrary but distinct values for *eq_class_offset*. When classes are held in ancestor sets they are always sorted by the key (*eq_class_id*, *eq_class_offset*), and so elements of equivalence classes appear in ancestor sets adjacent to one another in some arbitrary but fixed order, and if the elements of one equivalence class are supertypes of the elements of another, the supertype classes precede the others in the order.

A basic question is how eagerly to calculate and hold ancestor sets. In the implementation described here, the ancestor set of every class is held but no others. When some other ancestor set is needed in the course of an operation on types it will be calculated temporarily and discarded afterwards. Holding the upper and lower bounds on unification type variables as ancestor sets might be beneficial, since these undergo frequent joins and meets. (Indeed, the sole use for join in the current system is to calculate an update on a lower bound, and the sole use for meet is to calculate an update on an upper bound, except that it is also used to calculate the meet of the result types of meeting features.) However the present implementation does not hold ancestor sets for these bounds, because they are involved in other operations as well, and it is simpler to provide these operations on a single representation only. The main drawbacks of holding ancestor sets generally are their larger size and the non-uniqueness of the map back to types. Even weighted ancestor sets have a non-unique map back when the weights are tied.

Each class already has a weighted ancestor set; the class itself appears in its own ancestor set with weight 1, and all other class types have weight 0. So we need no algorithms for building ancestor sets initially; rather, we need algorithms for finding the union and intersection of two weighted ancestor sets, and for converting a weighted ancestor set back to a type.

Apart from class ancestor sets, ancestor sets are constructed and used only temporarily during operations whose inputs and outputs are types. Thus it may be possible to implement ancestor set operations destructively (saving time by cannibalizing the parameters). In particular, union and intersection might take the form of replacing an existing ancestor set destructively with its union or intersection with another ancestor set.

Since ancestor sets are sorted we may find the union and intersection of two ancestor sets using the well-known algorithms for finding the union and intersection of two sorted lists. When a class type appears on both lists we use the itype with the higher weight. For example, again showing the weights as superscripts:

$$\{int^1, int\_ref^0\} \cup \{int^0, int\_ref^1\} = \{int^1, int\_ref^1\}$$

In this case an arbitrary choice is going to have to be made when mapping back to a type, but in cases where the originating classes all agree on which element of an equivalence class they want, the map back to types will preserve this agreement.

The algorithm for converting a weighted ancestor set back to a type depends on the fact that all the elements of any equivalence class are sorted together in the ancestor set, and that any subtypes of those classes must appear later in the sorted list than they do. One can tell which classes are equivalent by comparing their *eq_class_id* values for equality. For each equivalence class in order from left to right (going from ancestors towards descendants, that is), find the set of classes in the equivalence class and identify an element of that class of maximal weight, $c_i$. For every class following $c_i$'s equivalence class, check whether $c_i$ is a supertype of that class. If this check fails in every case, $c_i$'s equivalence class is a generator for the type, so add $c_i$ to the growing result type. Repeat until the entire ancestor set has been examined.

## 2.3. Validity

Setting up the types as defined above requires several passes, establishing validity at successively higher levels simultaneously for all type expressions in the inherit clauses and feature headers of each class. Other type expressions (those local to feature bodies and invariants) are done separately later following the same steps.

***Level 1 validity.*** Establishing Level 1 validity requires two passes. In the first pass, the class files are read and class records and type variable records are created. At the end of this pass, the class records hold type variable lists, uninstantiated inherit clauses, and uninstantiated features. In the second pass, class and variable names occurring in type expressions are resolved to the appropriate class records and variable records; after this point, class and variable names are no longer used except in error messages. The parent set of each class is built during this pass, holding the concatenation of all the relevant class types, but not checked beyond Level 1.

***Level 2 validity.*** Establishing Level 2 validity is principally about building the ancestor set of each class – the reflexive and transitive closure of its parent set, which is then held permanently in the class record's *ancestor_set* field. To build an ancestor set one must be able to substitute actual types for variables, and to test whether a class type with a given class name is already present in the growing ancestor set and if so whether its actual generic parameters are equal to the actual generic parameters of another class type referring to the same class. These operations are all available at this point.

In the first Level 2 pass, the ancestor set is generated by a search through the parent sets, their parent sets, and so on. Problems with consistent genericity are found and reported, so by the end of this pass every class type is known to have consistent genericity. As soon as ancestor sets are built the subtype operation becomes available. In the second Level 2 pass, the ancestor sets are used to partition the classes into equivalence classes, an *eq_class_id* and *eq_class_offset* value is assigned to each class, and the ancestor sets are sorted by these values. When this point is reached, the join and meet algorithms (which depend on sorting) become available. Notice that although class types are certified as Level 2 valid at this point, meet types are not. Nevertheless we regard this as the end of Level 2.

***Level 3 validity.*** Establishing Level 3 validity is a matter of visiting every actual generic parameter and checking that it is a subtype of the corresponding formal generic parameter. During this pass we also expand every meet type into an ancestor set and then contract it back into a type, detecting any violations of consistent genericity and non-redundancy while doing so (a redundant meet type will contract back into fewer class types than the original type).

We use the term *immediate validation* for the process of taking a type expression and validating it through all three levels immediately. This is only possible after the completion of Level 2 validation of the class types generally. The algorithm simply executes each step in validation directly after the former on the one type, rather than simultaneously through the levels for many types.

## 2.4. Operations

Two class or meet types are equal if they have the same class types (in any order) with equal actual generic parameters. This condition is not quite correct if there is redundancy in the types, but equality testing is needed very early in the setting up process, before redundancy testing is possible, so we tolerate this slight imperfection.

In meet types we do not assume that the class types are sorted as they are in ancestor sets, even though normalized type expressions will be sorted. Again, this is because equality testing is needed very early in the setting up process, before sorting is available. Although it would be possible to offer two versions of equality testing, for unsorted and sorted types, it seems not worthwhile to do so, since most type expressions are not meet types anyway and those that are rarely contain more than two class types. This situation is very different from that in ancestor sets, where the number of elements is larger, and sorting greatly simplifies the algorithm for mapping back to types by bringing equivalent classes together.

Testing that class type $a$ is a subtype of class type $b$ is done by searching the ancestor set of class $a$ for a class type for class $b$. If not present, then there is no subtype relationship. If present, substitute the actual generic parameters of $a$ for the type variables of $a$ and test the class type found in $a$'s ancestor set for equality with $b$, removing the substitution after the test (note that substitute and copy is not required). Testing two general meet types for the subtype relation is done by following the definition which extends class subtyping to meet subtyping.

The general algorithm for calculating a join or meet is slow, so it is worth while prefixing it with a test to see if either parameter is a subtype of the other. These tests will often succeed, and when they do, one of the two parameters is also the result, and time has been saved and copying avoided.

If neither type is a subtype of the other, then the result of the join or meet will differ from both parameters, and serious work is required. Convert each parameter to a weighted ancestor set (for a meet type this requires taking the union of the ancestor sets of its class types). Then, if the operation is meet, take the union of the two weighted ancestor sets, failing if the union reveals inconsistent genericity. If the operation is join, take their intersection, failing if the result is empty. Finally, convert the resulting weighted ancestor set back to a type.

## 3. Classes and objects

### 3.1. Record layout

For each class, abstract and concrete, one C *struct* declaration is generated, containing a type tag field plus one field for each creation feature of the class, inherited or not. Each creation feature has the same offset in each struct in which it appears, so that accessing that feature can be done in a single record access, without reference to the type tag. The only difficulty in doing this is in deciding what offset to give to each creation feature, a problem called *record layout*.

In the presence of multiple inheritance, record layouts may contain some gaps. *Optimal record layout* – minimizing these gaps – is a hopeless quest for two reasons: the problem is well known to be an NP-complete vertex colouring problem, and information about how many records of each type will be created at run time is not available.

Instead of trying for optimal record layout, Nonpareil uses a simple greedy vertex colouring heuristic, with the vertices ordered so that parent classes (ignoring coercion) are laid out before child classes. Each class contains a set of bit positions that are known to be unavailable for any subsequent assignments to the creation features of that class, because to use them would cause clashes somewhere. These sets are initially empty in all classes, except that the first 32 + 16 bits are unavailable owing to the object signature and type tag. Each type knows how many bits a field of that type will occupy in a record (usually 32, but enumerated types can be 8, 16, or 32).

Offsets are allocated to creation features in parent classes before child classes. When layout reaches a class *c*, all its inherited creation features will already have offsets, and the bit positions they occupy (plus possibly others) will be in *c*'s unavailable set. Based on this information, *c*'s original creation features are then allocated the smallest offsets that can accept their widths without overlapping any of the unavailable bit positions. This could cause some reordering, for example if there is room for a Boolean earlier in a record, but not for a wider field.

After all the original creation features are allocated offsets, the combined set of bit positions made unavailable by these allocations is notified to all classes that share a common descendant with *c*. In the case of single inheritance this is just *c*'s descendants, but in general all ancestors of all descendants must be notified. This is a peculiar set of classes, given that the underlying relation is not transitive. For example, suppose there are five classes, *a*, *b*, *c*, *d*, and *e*, and that *d* inherits *a* and *b*, and *e* inherits *b* and *c*. Then any positions given to features of *a* are unavailable to *b* because of the common descendant *d*; any positions given to features of *b* are unavailable to *c* because of the common descendant *e*; yet the positions given to features of *a* are available to *c*, because *a* and *c* have no common descendant.

This notification is implemented in two phases. In the first phase, after *c* completes its own allocation, it notifies all its proper descendants that the positions occupied by these allocations are now unavailable to them. In the second phase, before some other *c* begins its allocation, it queries all its proper descendants, adding their unavailable sets to its own.

### 3.2. Enumerated classes

The implementation of enumerated classes is not very different from ordinary classes. The corresponding C type is `unsigned char`, `unsigned short`, or `unsigned int`, depending on the number of code points. There is a second C type which is a reference to the object proper, including a type tag. This C type has no corresponding Nonpareil type.

The code generation for access to creation variables includes a lookup in a predefined array or (in general) trie. Since a trie is just an array of (arrays of …) arrays, it can be constructed during initialization using the usual array constructor functions, which include uniquefying by checking the array against the object hash table, and its nodes swizzled and unswizzled just like other arrays. The compiler needs to inject the three special features (*int_code*, *with_code*, and *legal_code*), sort the predefined objects by code number, and generate code for building these arrays of arrays bottom up – a tedious but routine affair. Tries are built simultaneously with *all_enumerated* arrays. Since the objects containing the creation features are referenced only from the tries, they are created at this time also.

The main remaining issue with enumerated classes is assigning suitable code numbers to those predefined object features that do not already have them. A curious duality with the problem of assigning offsets to creation features comes to our aid here. A creation feature must have an offset that is different from that of any creation feature with which it shares a class – any creation feature introduced in any ancestor of any descendant of the class in which it is introduced. An enumerated object must have a code number that is different from that of any enumerated object with which it shares a class – any enumerated object introduced in any descendant of any ancestor of the class in which it is introduced. Some kind of dual of the class record layout algorithm is therefore indicated, in which ancestors and descendants are reversed. Code number assignment is simpler in that code numbers are integers, whereas record offsets are integer ranges; it is more complex in that some features have preassigned code numbers, which must be respected, and which cause 'duplicate code number' errors when two predefined objects with the same code number have a common ancestor class.

We dispose of the last problem first. Sets of code numbers are represented by the same interval set data structure that is used for sets of offsets. This represents just the numbers themselves, not the entities they derive from, and so, although it can detect clashes, it cannot produce very useful error messages when it does. Therefore we do not worry about detecting clashes in this set; we merely insist that each class contain an interval set containing all the code numbers of all the predefined objects in that class that currently have a code number. Later, when generating code for building the trie for the class, the first step is to sort its predefined object features by code number, and clashes may be easily detected and reported intelligibly then. They can derive only from clashes in preassigned code numbers, not compiler-assigned ones.

Initialization of the interval sets for code number assignment is more complex than for record layout, owing to the presence of preassigned code numbers. We will consider this later, after we have examined the algorithm obtained by mechanically interchanging 'ancestors' and 'descendants' in the record layout algorithm. Here is that converted description.

'Code numbers are assigned to predefined object features in child classes before parent classes. When assignment reaches a class *c*, all its proper descendants' predefined object features will already have code numbers, and these numbers (plus possibly others) will be in *c*'s unavailable set. Based on this information, *c*'s original predefined object features are then allocated the smallest code numbers that do not conflict with any numbers in the unavailable set.

'After all the original predefined object features are allocated code numbers, the combined set of code numbers made unavailable by these allocations is notified to all classes that share a common ancestor with *c*. All descendants of all proper ancestors must be notified. This is a peculiar set of classes, given that the underlying relation is not transitive, as described in detail for the record layout algorithm.

'This notification is implemented in two phases. In the first phase, after *c* completes its own allocation, it notifies all its proper ancestors that the positions occupied by these allocations are now unavailable to them. In the second phase, before some other *c* begins its allocation, it queries all its proper ancestors, adding their unavailable sets to its own.'

Examination of this algorithm shows that it will work correctly provided that each class's unavailable set initially holds all the declared code numbers of preassigned objects defined in all the descendants of its own class. This is arranged as follows. By the time this algorithm runs, each class contains a list of its predefined object features, each of which may be queried to find out whether it has a code number interval already, and if so what interval. Initially the code sets of all classes are null, signalling that this initialization has not occurred for them yet. When initialization is invoked on such a class, it first ensures that all its children are initialized, then it sets its code set to the union of the code sets of all its children and the existing code intervals of its own predefined object features. As explained above, code number clashes are ignored at this stage.

Since all classes also contain an array of all their predefined objects, including predefined objects defined in descendant classes, and since this array has been set up by this point, an alternative method of initialization would be simply to run through this set and accumulate the code numbers it contains. However, this would visit each feature once for each of its ancestors, rather than just once, so the previous method is preferred.

### 3.3. Arrays and strings

The bulk of the data in any document will be carried in arrays and strings, so this is one place we need to optimize memory usage. The *array* class is builtin:

```
class array{X}
    builtin
noncreation
    length: int := builtin array_length(self)
    apply1(i: int): X := builtin array_get(self, i)
    coerce to_array_ref: array_ref{X} := array_ref(self)
    …
end
```

Many operations will be provided; only the basic ones are shown here.

Use of the name *apply1* permits the user to write *a*(*i*) instead of, say, *a*.*get*(*i*), which is convenient. However it does not by itself permit an array to be passed as a function; coercion to *array_ref* is needed for that:

```
class array_ref{X} inherit fun1{int, X}
    coerce val: array{X}
noncreation
    apply1(i: int) := val(i)
end
```

Thanks to *array_ref*, an array may be used in a polymorphic context, such as where a function is expected, just as *int_ref* allows *int* objects to be used in polymorphic contexts. However the cost in space is paid only on the rare occasions when it is required.

The builtin representation consists of a reference to a one-word length field followed by that many words holding the values. This arrangement is secure (every array knows its own length, and the type system knows the type of its elements) and maximally space-efficient.

The same representation is used for strings, naturally, since they inherit from *array*. Constant strings appearing in the program text are treated as predefined objects, and initialized during global initialization. Higher-dimension arrays could be given their own builtin implementation, beginning with two or more length fields; or they could be jerry-built using the well-known reduction to arrays of arrays.

Memory management requires that the size of every allocated piece of memory be identifiable, and swizzling requires that those fields that contain references be distinguishable from those that contain immediate values. How this is done for arrays will be explained in the section that describes memory handling.

## 4. Features

### 4.1. Feature inheritance

Associated with every type $a$ is a set of features, which we call the type's *feature signature*. Given a value of some type, these are the features that may be called on that type.

For every type $b$ such that $a \leq b$, all of the features of $b$ must also be features of $a$. In this section we define this relationship precisely for those features made available either by direct definition or by inheritance (but not by coercion). We call these features the *inherit signature* of the type. In the following section we extend to the full signature including features made available by coercion.

The term 'feature' is not precise enough for our needs in this section. Instead we will be using the terms 'feature instance' and 'feature signature.'

A *feature instance* is a declaration of a feature written explicitly in some class. It may be a *concrete feature instance* (have a value) or an *abstract feature instance* (have no value, because it is a noncreation feature without a body).

Consider a typical feature, such as

$$map\{y\}(f\!:\!fun1\{x, y\})\!:\,list\{y\}$$

In order to call this feature, one needs to know its name (including whether it has an ordinary or operator name), formal generic parameters (including any constraints on them), formal parameters, and result type. These items are part of the *feature signature.* Unlike feature instances, which are never copied or inherited, feature signatures are copied from parent classes to child classes and merged with other signatures.

Apart from information about the interface of the feature, a feature signature contains information relevant to checking inheritance and calling the feature. Altogether it has these items:

1. The name of the feature;
2. A *creation flag*, indicating whether the feature is a creation or noncreation feature;
3. A *coercion flag*, indicating whether the feature is a coercion feature or not;
4. A sequence of formal generic parameters and any constraints on them. The particular names by which the formal generic parameters are identified are not part of the feature signature; renaming any of these generic parameters throughout the feature signature to any other non-clashing name does not change the feature signature;
5. A sequence of *compulsory parameters*, each with its name and type;
6. A set of *optional parameters*, each with its name and type (but not any default value);
7. A result type;
8. A feature instance called the *base feature instance*;
9. A feature instance called the *effective feature instance.*
10. An *inherit state* which can have one of three values: *no* (the represented feature is delared in the child class but not currently known to be inherited), *yes* (the represented feature is inherited and is not redefined in the child class), and *redef* (the represented feature is both inherited and redefined).

11. The chain of zero or more coercions that must be applied to the target before this feature is called on it.

Associated with each type there is a set of feature signatures with distinct names, defining what features may be called on objects of that type. This is the *signature of the type.* If we consider only features made available by direct definition within a class or by inheritance (not by coercion), this is the *inherit signature of the type.*

Most of the items just echo information from feature instances. The need for item 8, the base feature instance, arises because, traditionally, if class $a$ contains a feature $f$, and class $b$ contains an unrelated feature which happens to have the same name $f$, then $a$ **meet** $b$ is considered to be a mistake: features conceived separately should not be merged accidentally. We check for this by recording in each signature the feature instance it originated from, which is what the base feature instance is, and refusing to merge signatures when their base feature instances differ.

There is a *base feature instance invariant* which can simplify the comparison of feature signatures at times. It states that two signatures with the same base feature instance must have the same values for items 1, 2, 4 (subject to renaming), and 5 in the list above. This can be proved inductively based on the rules for constructing and merging feature signatures given later in this section.

The effective feature instance is the feature instance that will be evaluated during execution if a feature call with this name is made on an object whose dynamic type's signature includes this feature signature. This is also the feature instance that supplies default values for any missing optional parameters (slogan: *default values are part of the body*). If the effective feature instance is abstract or absent, then objects with this dynamic type may not be created.

There is an *effective feature instance invariant* which also can simplify the comparison of feature signatures. It states that two signatures with equal non-absent effective feature instances must have the same values for all items in the list above. This too can be proved inductively.

The inherit state arguably does not have to be in the signature at all, since, in the end, it does not matter why a feature comes to be there. However, it is an important part of the implementation, and very helpful in understanding what is happening in a concrete way. The coercion chain relates to coercion signatures, defined in the next section, and will be empty in all the signatures created in this section.

Our goal in the rest of this section is to explain how to calculate the inherit signature of any type, starting from the feature instances, in slightly more detail than users actually need. We will do this first for class types, then for meet types, and finally for variable types.

Consider an arbitrary instantiated child class inheriting zero or more instantiated parent classes. Since the inheritance graph is acyclic, we may assume inductively that the inherit signatures of the parent class types are already known. Form a set consisting of all feature names in the inherit signatures of these class types, plus the names of all feature instances in the child instantiated class. The inherit signature of the child class type contains one feature signature with each of these names. We need to define, for each name, how the feature signature with that name is calculated.

There are three cases: (i) when the name occurs among the feature instances of the child class but not in any parent type's inherit signature; (ii) when the name occurs in one or more parent types' inherit signatures but not in the child class; and (iii) when the name occurs in both a parent type's inherit signature and in the child class.

In the first case, when the name occurs among the feature instances of the child class but not in any parent type's inherit signature (making the feature instance in the child class an *original definition*), the elements of its signature are all taken from the feature instance in the child. Both the base feature instance and the effective feature instance are set to this feature instance, the inherit state is set to *no*, and the coercion chain is empty (as it will be throughout this section, since coercion is not yet being considered).

In the second case, when the name occurs in one or more parent types' inherit signatures but not in the child, the effective feature instance in all these inherit signatures must be the same (if not it is an error). The effective feature instance invariant then implies that all the other items are the same too, and these common values go into the name's signature in the child class type. This case is handled by copying the inherited signature into the type with inherit state *yes* when first encountered in a parent, then checking only that subsequently encountered versions have the same effective feature instance. The coercion chain of each parent is empty, since coercion has not yet been considered for *any* class, and the coercion chain of the child is empty.

In the third case, when the name occurs in one or more parent types' inherit signatures and in the child class (making the feature instance in the child class a *redefinition*), once again the items are mainly taken from the child feature instance, but now there are constraints requiring these values to be related to those of the parent signatures. Here, in complete detail, are the rules for constructing and checking the child signature in this case:

1. The name of the child signature is the name of the child feature instance, which by construction is the same as the name of all the parent signatures.

2. The child signature's creation flag is that of the child feature instance. This must equal the creation flag in every parent (this may be relaxed to the Eiffel rule in future).

3. If the coercion flag is set in any parent, then the child feature instance must be marked **coerce** and its coercion flag is set. If the coercion flag is not set in any parent, then the child feature instance is free to either be marked **coerce** or not, and this determines the value of its coercion flag.

4. The child signature's formal generic parameters and their constraints are those of the child feature instance. These must equal those in every parent signature, subject to the freedom to rename described in the definition of feature signatures.

5. The child signature's sequence of compulsory parameters is that of the child feature instance. This sequence must match that in every parent in containing the same number of parameters, with the same types (but not necessarily the same names) in the same order.

6. The child signature's set of optional parameters is that of the child feature instance. For every optional parameter in every parent signature, there must be an optional parameter in the child signature with the same name and equal type.

7. The result type of the child signature is the result type of the child feature instance. In a creation feature this must be equal to the result type in every parent signature; in a noncreation feature it must be a subtype of the result type of every parent signature.

8. All parent signatures must have the same base feature instance, and this common value becomes the base feature instance of the child signature.

9. The child signature's effective feature instance is the child feature instance, without regard to the parent signatures' effective feature instances.

10. The child feature's inherit state is *redef*.

11. The child feature's coercion chain, like that in all its parents, is empty.

This case is handled by initially creating a signature for the redefinition as for an original definition. When the first parent is discovered, all the checks above are made, the child's base feature instance is reset to the parent's, and its inherit state is changed to *redef*. If further parents are discovered, we compare base feature instances, use the base feature instance invariant to skip some of the checks, but make the others.

The rule for optional parameters is somewhat unusual in that it allows a redefined feature to declare additional optional parameters. When called by dynamic dispatch, actual parameters corresponding to these additional ones may be absent – but that does not matter, the default values are available in that case. This design is essential for cloning an object and updating some of its fields without having to know exactly what its type is, a vital operation in the document formatting application.

The rule for result types allows for 'covariant redefinition' of the result types of noncreation features. This also is essential for cloning with update. Ironically, it is the need to support cloning that necessitates prohibiting covariant redefinition of the result types of creation features, since this would imply covariant redefinition of the corresponding parameters of *clone*, which is not type safe, as any small example involving two classes and one such redefined variable will quickly reveal. It is not a problem for creation since creation functions are not inherited.

This completes our definition of the inherit signature of a class type. We must now define the inherit signature of meet types and variable types.

Our definition of the inherit signature of a meet type is based on merging the inherit signatures of its component class types. There will be one feature signature for each distinct name among the feature signatures of these contributing class types. We call this the child signature, and we call each signature with this name from the inherit signatures of the class types being merged a parent signature. Here are the rules for constructing and checking each child signature:

1. The name of the child signature is the common name of all the parent signatures.

2. The child signature's creation flag is equal to the parents', which must all be equal. *!!! is this necessary? !!!*

3. The child signature's coercion flag is set if it is set in any parent.

4. The child signature's formal generic parameters and their constraints are those of the parents, which must be equal subject to the usual freedom to rename.

5. The child signature's sequence of compulsory parameters is that of the parents, which must be equal in their names as well as their number and types.

6. The child signature's set of optional parameters is the union of the parents', which must be compatible in the sense that whenever some parents have an optional parameter with the same name, the types in those parents must be equal.

7. If the feature signature is a creation feature, all its parents' result types must be equal and this common value is the result type of the child signature. If the feature is a noncreation feature, the result type of the the child signature is the meet of the result types of the parents. It is an error if this meet does not exist.

8. All parent signatures must have the same base feature instance, and this common value becomes the base feature instance of the child signature.

9. The child signature's effective feature instance is absent, and no compatibility rules are placed on the effective features of the parents.

Having no effective feature instance does not matter as long as no objects with this type as dynamic type exist. This is guaranteed in Nonpareil by providing no means of creating an object of variable or meet type. Compulsory parameters are required to have the same names in all parents because otherwise we would not know what names to give them in the child signature, or in other words what names could be used for the named compulsory parameters in calls.

The inherit signature of a variable type is just the inherit signature of the type that the variable is constrained to be a subtype of. If it is not constrained, the inherit signature is empty and nothing can be done with a value of that variable type except to pass it around. Not even a test for equality is available.

The soundness of this system is based on the fact that for every pair of types $a$ and $b$ such that $a \le b$, and for every feature signature $f$ in the inherit signature of $b$, $f$ is in the inherit signature of $a$, where it contains the same generic parameters and constraints as in $b$, the same sequence of compulsory parameters, all the optional parameters and possibly others, and a result type which is a subtype of the result type in $b$. Therefore $f$ in $a$ may be safely substituted for $f$ in $b$. Furthermore, every feature signature of every type for which objects may be created contains an effective feature instance which implements that signature.

## 4.2. Default values of creation features

Correct handling of the default values of creation features raises two problems, which this section explains and solves.

The first problem is to manifest the values in the correct context. As the Language Specification defines, a default value of a creation feature may not reference **self** or any creation feature defined after it is. This derives from the way creation features are repackaged as parameters of creation functions. A suitable context, then, is the usual outer level, visible throughout the module, of the visible classes, plus the class type variables, plus all the visible creation features of the current class that are defined before the current feature.

An algorithm for manifesting the creation features of a class view, after inheritance has been completed, is as follows. First, load the context with the visible classes and class variables. Then proceed through the creation features of the current class view in canonical order, inserting each in turn into the context. Just before doing that, if the feature's default value needs manifesting (because it is defined in this class view), manifest it.

One of our principles is that an expression should be linked to the front-end structures at one point only, where it is manifested. Accordingly, imported and inherited front-end creation features should drop their default values, so that only the original front-end feature has an expression to manifest. The test for whether a front-end creation feature's value needs manifesting is then simply a test for whether it *has* a value.

Adding a creation feature to context *as* a creation feature would not ordinarily work, because manifesting would prefix it with **self**, which will then fail when **self** cannot be found. This may be avoided by adding creation function parameters to the context instead of creation features, which is logical enough, since we are after all manifesting a creation function.

The second problem is that default values are inherited, and so they may be invoked within several creation functions.

Noncreation features avoid copying by building one large function. Each distinct body is code generated just once, below a list of case labels, one for each class in which the body is current. To adopt such a solution for creation functions would be highly unintuitive, because unlike noncreation functions they are not polymorphic.

Another popular way to handle polymorphic functions is to compile each distinct body as a separate function, and place a pointer to that function in the type record referenced by each object in which that body is effective. This, minus the polymorphism, is how Nonpareil handles default creation feature values: each distinct body is packaged into a separate *creation feature default value function*, and expression copying is avoided by calling that function when that body is required. The C compiler will inline these calls in most cases.

To determine what parameters the function must have, the mechanism for finding the hidden parameters of **let** definitions may be used. For example, given the creation feature

*angle*: *real* := *sides*/180.0

one would first package the body in a **let** definition, initially

**let** *angle*: *real* := *sides*/180.0 **in** *angle* **end**

but after manifesting and discovery of hidden parameters this becomes

**let** *angle*(*sides*: *int*): *real* := *sides*/180.0 **in** *angle*(*sides*) **end**

The definition here is the required creation feature default value function, usable as a global function. The enclosing back-end function passed to the manifesting would be this definition itself. In practice no **let** expression is created, but the equivalent work is done.

The hidden parameters mechanism adds to each formal parameter of the creation feature default value function a record of what outer symbol caused it to be created. As explained earlier, this would be a formal parameter of a creation function. The corresponding actual value to be substituted is a call on a parameter of a creation function, but not the same creation function if the default value is inherited.

There are several possible solutions here. One is to share parameters between creation functions. We prefer not to do this, because sharing front-end parameters implies sharing back-end parameters, which in turn implies sharing back-end parameter names – which is dangerous, since each back-end function has its own parameters defined in its own scope.

Our solution is a little clumsy but quite workable. Each hidden formal parameter refers to a formal creation function parameter; each formal creation function parameter refers to a creation feature. So it is easy to work out, for each formal parameter of the creation feature default value function, which creation feature it is for. One then needs to scan the list of formal parameters of the current creation function, to find the formal parameter that is for this same creation feature. The actual value to supply to the formal parameter is then just a call on this formal parameter.

### 4.3. Preconditions

A list of expressions may follow the **require** keyword. The whole list is stored in a single expression of type `EXPR_PRECOND` and treated as a unit thereafter. Each expression is enclosed in a call on the `assert()` macro before code generation, although some other signalling mechanism, such as a long jump on failure, could easily be used.

The `EXPR_PRECOND` object has a `subexpression` field which allows the preconditions to guard an arbitrary expression. It is used in let definitions to make the preconditions an integral part of the body of the let definition, where they can basically be forgotten; we will not mention preconditions of let definitions again. For features, in which preconditions are inherited, precondition expressions are stored separately from feature bodies, as will be explained next, and have no associated subexpressions.

The original implementation had preconditions of features as part of the feature body, like default values of parameters, as just described for let definitions. This had several advantages, but two disadvantages outweighed those. The first was that a feature may have a precondition even when it has no body; and the second was that burying preconditions inside bodies made them hard to get at when they needed to be inherited.

Each front-end feature holds an optional local precondition, and an optional list of inherited preconditions. The local precondition is manifested at the same time as the body of the front-end feature (or if there is no body, at the time it would be manifested if there was one). The others are inherited so will have been manifested at their points of origin.

To implement inheritance of preconditions is not difficult. Whenever a feature is discovered to inherit from another feature, references to the preconditions held in the parent front-end feature are copied into the list of inherited preconditions in the child front-end feature. Any references already present in the list are not copied again (these arise from grandparent features inherited twice).

The preconditions of features are treated as front-end functions, and the hidden parameter mechanism is used to work out what their parameters should be, exactly as described above for default values of parameters. Each feature instance (limb of the corresponding back-end function) is given a list of the back-end versions of all the preconditions that apply to that feature instance, so that it knows to call them. They are also added to the list of inner functions of the back-end feature, like let definitions are, and this ensures that they will be generated as local functions just preceding the feature.

When calling a back-end feature precondition function, it is necessary to supply actual parameters that are calls on the formal parameters of the enclosing feature. Which formal parameters to call can be deduced, as usual with hidden parameters, by carefully following the information recorded in each hidden parameter saying what caused it to be created. These calls are inserted into the code generated for each limb of the back-end feature, after default value substitution but before the body of the feature is evaluated.

### 4.4. Invariants

There is a connection between preconditions and invariants, which at first sight offers an easy means of implementing the latter. If we convert the invariant

**invariant**
*expr*

into the feature

*invariant*: *bool* **require** *expr* **end** := **true**

then inheritance of invariants maps automatically into inheritance of preconditions.

Unfortunately, pursued literally this produces an enormous C function, containing every invariant in the system, whose massive polymorphism is never needed because it is called only from within creation functions, where the exact type is known.[1] Even more seriously, the connection breaks down in class extensions: features appearing in class extensions are always new features, but invariants appearing in class extensions need to be added to the existing invariants. So the connection obscures as much as it illuminates, and we look for some other implementation.

Each class *c* is responsible for building a function named *c_invt*, with one parameter, **self**. The function evaluates the invariants that appear locally in that class and all its extensions, but not in ancestor classes. Each class view of that class manifests and contributes zero or more expressions to this function. The function is constructed at the time the first invariant is contributed; if no invariants are contributed, no function is constructed. This function appears in the generated code just before the class's creation function. Its result type is `void`, and it returns if and only if the invariant holds. At present failure is signalled by an assert error, but in future it will cause a long jump to a point where the most recent top-level editing operation can be rolled back.

An advantage of this method is that it allows invariants to call on features defined within the same class. These need access to a `self` variable, which this method provides. Of course, such calls cannot assume the invariant as an implicit precondition, as ordinary feature calls can. This has been pointed out in the Language Specification.

For each class, the compiler keeps track of the set of invariant functions that apply to that class, whether from the class itself or from its ancestors. For non-enumerated classes it inserts calls to each of these functions at the end of the creation function, passing as a parameter a reference to the object just created and inserted into the cache. (The cache must hold every created object, not just objects known to satisfy their invariants, otherwise equality tests will misbehave inside invariants.)

For objects of enumerated classes a different scheme is needed. Immediately after code is generated to create the objects and build their trie, more code is generated that loops through all the code numbers and calls the invariant functions for each number. This code is only generated if there is an invariant. As explained in the Language Specification, an invariant could depend

[1] Omitting *invariant* from the *object* class, thereby breaking the feature into smaller ones, fails when multiple inheritance brings together two *invariant* features with different base features. If this error is suppressed in this special case, the resulting feature has two base features and it is not clear what code to generate.

on the code number, so it would not be correct in general to check just one object per code number range.

### 4.5. The *clone* feature

The definition of the *clone* feature given by the Language Specification indicates a simple approach to implementing it: inject an artificially created *clone* feature into each non-builtin class and handle it in the usual way. This simple plan is followed by the compiler, although there are several problems, which this section explains and solves.

Carrying out the simple plan is a matter of choosing the right points to intervene in feature creation, import, and inheritance. For each non-enumerated, non-builtin class view there needs to be an appropriate *clone* front-end feature. If the class view is imported from another module, it will already have a *clone* feature; otherwise, one is created just before feature inheritance begins, initially with just a **self** parameter. During front-end feature inheritance, if any *clone* features are inherited, the current one is declared to be a redefinition of those in the usual way.

Some of the parameters of an imported *clone* front-end feature may derive from private creation features. These parameters are marked private, so like private feature parameters in general they are omitted from the signature when importing.

A problem when redefining is that the redefinition has only a **self** parameter initially, and if the parent features have other parameters (which they usually do), this is an error. However a special arrangment is made for *clone*: although it does get run through the inheritance mechanism, for the most part it is just ushered through without checking. Whenever a creation feature is added to a class view, either by import, inheritance, or original definition, a corresponding parameter is added to that class's *clone* feature. If the creation feature is original (neither imported nor inherited), the corresponding back-end function gains a parameter as well. This is similar to how creation functions are built, except that since creation functions are not inherited, a new back-end parameter is added for each original or inherited creation feature.

In addition to getting the parameters right, the implementation must also ensure that the right body is produced, including default parameter values. One way to do this would be to literally build the default values and bodies shown in the Language Specification, for example

> *clone*(*sides*: *int* := *sides*, *radius*: *real* := *radius*, *angle*: *real* := *angle*): *regular_polygon* :=
> *regular_polygon*(*sides*, *radius*, *angle* := *angle*)

and manifest and code generate these as usual. This fails when there is no scope in which all the default values are visible, that is, when some of the corresponding creation features are defined in class extensions. (The statement made in the Language Specification, that *clone* could be defined by the user, is thus not strictly true when there are class extensions, since there may be no point in the source code where every creation feature is visible.) Instead, the back end code generates the appropriate code from a traversal of the creation features of the class.

The problems so far have been minor, but now comes the major problem with the simple plan, to which the rest of this section is devoted: if a branch of a single *clone* feature is injected into every non-enumerated class, the resulting backend function will be excessively large. The large number of branches (one for each non-enumerated, non-builtin class in the system) is one problem, which could be solved as it was for the swizzle functions (Section 6.3), although for a function as frequently called as *clone* this would not be ideal. But there is a worse problem:

the backend function has a huge number of parameters, one for every creation feature in each non-enumerated class in the system.

Some way is needed to partition the backend function into a set of smaller functions, each covering only a subset of the classes. Each of these will then have only a subset of the parameters. The danger in doing this, as was noted in a footnote to the section on invariants, is that if the partitioning is too fine there might be several relevant effective features.

An abstract description of the partitioning method will be given first, which makes its correctness clear but needs some refinement in practice.

Define a graph with one node for each class needong a *clone* feature (each non-enumerated, non-builtin class in the system). Connect two nodes by an edge whenever the corresponding classes are related either by inheritance, or by appearing together in a meet type. As an exception, do not connect the node corresponding to the *object* class to any other node.

Take each connected component of this graph except the one containing just the node corresponding to *object*. Add a new empty class, inheriting only from *object*, to each component. In each of the other classes of the component that inherits only from *object* (normally indicated by an empty inheritance clause), change the inheritance to inherit from this new class. Then inject a *clone* feature into every class except *object*, as in the simple method. Each component has its own *clone* feature, with one parameter for each creation feature in the classes of the component, which should be a manageable number.

After all the inheritance is done, add a *clone* feature to the *object* class. Since the *object* class has no creation features, this *clone* feature has no parameters other than **self**, can update nothing, and so may be implemented as the identity function. It is useless, but it is put there so that its absence does not have to be mentioned.

As described, this algorithm wastes time introducing many dummy classes. There is also a problem with the constraints it places on the staging of the compilation: it requires that every type expression in the system be validated before it begins.[1] The following refined version eliminates these problems.

There are no problems in the front end; the problem is with the backend feature generated. So the refined algorithm continues to use the simple plan in the front end, but builds the features of the component method in the back end.

Inheritance of *clone* from *object* to any child class is ignored. The function which carries out the inheritance of one front-end feature begins with a check to see if this is what is being requested, and does nothing in that case. This allows a *clone* front-end feature to be added to each *object* class view at the usual time, rather than at the end of the algorithm as described above, which would be awkward to arrange.

At any moment during compilation there is a *current partitioning* of the classes, based on the type expressions validated up to that moment. The following invariant is maintained:

[1] An argument can be made that only inherit types need to be validated, since any **meet** type that is not reflected somewhere in an inheritance clause cannot be the type of any object, and could be rejected by the compiler as useless. However, such types might be introduced in anticipation of classes still to be defined, so it is better to allow them.

> *A clone feature is associated with each partition that contains at least one class that contains a clone front-end feature, and all the clone front-end features in that partition (and only those) refer to that clone feature.*

If this holds, then in the end the right *clone* features will be generated. It remains to show how it can be maintained across every possible disturbing event.

When a new class is introduced, it goes into a partition by itself, with no clone feature.

When a *clone* front-end feature is introduced into a class (not inherited), it does not necessarily cause a new feature to be created, as is the case with other non-inherited features. Rather, it refers to the *clone* feature of whatever partition that class currently lies in. If that partition has no *clone* feature, a new one is made and associated with the partition first.

When validation of a type expression causes two partitions to be merged, if neither partition currently has a *clone* feature, then the resulting partition has no *clone* feature; if one has a *clone* feature, then the result has that feature; and if both have a *clone* feature, then the two features are merged. It is easy to merge two *clone* features: simply append the parameters (except **self**) and feature instances of the first to the second, and change all references to the first to point to the second (or, more practically, add a reference from the first to the second, which is interpreted to mean that all references to the first should be forwarded to the second). The parameter sets must be disjoint, because they originate from creation features in unrelated classes. The bodies must be disjoint, because the sets of classes in the two partitions are disjoint.

All these events are free to occur in any order, except that a class must be placed into the same partition as its parents before a *clone* feature is injected into it. Otherwise, when their partitions are merged later, there will be duplicate parameters. This is trivial to arrange, because the parent types of a class are always validated before any processing of its features begins.

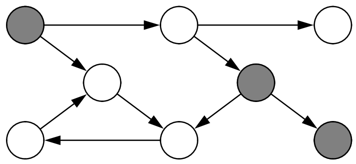
A purely cosmetic issue is what back-end name and type to give to the *clone* feature. A back-end feature contains a pointer back to the front-end feature that originated it, and the back-end name and type are taken from there. In an ordinary feature this pointer points to the base front-end feature. In a *clone* feature it initially points to whatever front-end feature first stimulated the creation of this *clone* feature. However, whenever an uninherited front-end feature is linked to this feature, the pointer is reset to this front-end feature (and the back pointer of its `self` parameter is reset to this front-end feature's **self** parameter). This ensures that in the end the name and type will be of some class that inherits only from *object*. For example, even if the *clone* feature created to service the disjoint set of classes {*list*, *elist*, *nlist*} is originally created for *nlist*, these resets will ensure that it winds up appearing to have been created for *list*. The code generation will follow this hint and print the feature with the others from the *list* class, making everything consistent with the practice for ordinary features.

One issue may be bothering the reader: it is quite possible for two classes in the same partition to contain unrelated creation features with the same name. Each of these will give rise to one parameter of the clone feature of that partition. That is not a problem for the back end, because the parameters in any C function are always assigned distinct names anyway, based on the Nonpareil names but not limited to them. There can be no front-end feature signature or feature body which refers to both of these parameters, since that would imply the existence of a class that inherits both of them, which is prohibited for unrelated features with the same name.

### 4.6. Initialization of predefined object features: a general algorithm

The compiler must either ensure that initialization of predefined object features is carried out in a safe order, or else it must print an error message at compile time explaining why no such order exists. An efficient solution to this problem turns out to require more than simple topological sorting of the call graph. A general version of the problem is defined and solved in this section, then adapted to the particular Nonpareil context in the following section.

An instance of this general problem is a directed graph in which each vertex is either *coloured* or *uncoloured*:



The problem is to order the coloured vertices so that if there is a path from $v$ to $w$, where $v$ and $w$ are coloured vertices, not necessarily distinct, then $w$ appears in the ordering before $v$.

The relevance of this problem should be clear. Roughly speaking, a global order of initialization of predefined object features may be found by letting every feature (predefined or otherwise) be a vertex of this graph, colouring the vertices representing predefined object features, and drawing an edge from $v$ to $w$ whenever $v$'s feature calls $w$'s feature. The full details are given in the next section.

There is a straightforward solution to this problem. Find the strongly connected components of the graph (i.e. partition it into sets of vertices – the components – such that every vertex in any one set is reachable from every vertex in that set). If any component containing a coloured vertex contains more than just that one vertex (even an edge from the vertex to itself), then the graph contains a cycle containing a coloured vertex. Otherwise, a topological sort of the quotient graph (which must always be acyclic) gives a suitable ordering for initialization.

Unfortunately, this algorithm is not a good fit in practice, partly because it requires the full graph to be built, even though most of it is unlikely to be connected to any coloured vertex, and partly because strongly connected component algorithms are messy things (the one known to the author requires reversing all the edges during its second phase, for example). Instead, the following algorithm is used, which, while not guaranteed to run in linear time, is much simpler to implement and likely to be very fast on the highly disconnected graphs expected in practice.

Topological sorting is well known to be implementable by a sequence of depth-first search traversals. Depth-first search requires a boolean flag in each vertex saying whether the search has already visited that vertex, to prevent infinite loops. When there are repeated traversals, a well-known optimization replaces the boolean flag by an integer *visit number*. Each traversal has a visit number which is one larger than the previous traversal's (the first has number 1). Each vertex's visit number is initially 0. When the search reaches a vertex, the vertex is considered to be already visited if its visit number equals the traversal's number. Otherwise it is not visited, so the traversal's visit number is assigned to the vertex, the vertex is visited, and the search continues down its outgoing edges.

The algorithm to be used is based on these ideas, with a few wrinkles. Uncoloured vertices contain a visit number. Coloured vertices contain a *state*, which can be *new* (no search has ever entered this vertex), *active* (a search has entered this vertex and not yet left it for the last time), or *completed* (a search has left this vertex for the last time).

Initially all uncoloured vertices are unvisited and all coloured vertices are new. While a new coloured vertex exists, choose one, change its state to active, get a new traversal visit number larger than all previous visit numbers, and commence a depth-first search of the call graph from the chosen vertex.

When the search reaches an uncoloured vertex, it either recurses or not depending on the visit number, in the usual way. When the search reaches a coloured vertex, the action depends on the state of that vertex.

If the state is new, change it to active, get a new visit number larger than all previous visit numbers, and continue searching recursively in the usual way, only with the larger number. This effectively restarts the search from this vertex, but with previous coloured vertices still active.

If the vertex is active, it lies on a cycle. *Mark* this vertex and return error state 1. Whenever any vertex receives error state 1 as the return value of a recursive call, it prints information identifying itself and immediately returns with error state 1, producing a trace of the cycle of calls that is the problem, until the recursion unwinds back to the marked vertex, whose mark causes it to print its identifying information and return error state 2. Whenever any vertex receives error state 2 as the return value of a recursive call, it immediately returns error state 2, silently unwinding the recursion.

If the vertex is complete, its initialization has already been done. It and everything it calls cannot be a problem, so return success immediately. Success is also returned when a vertex, coloured or uncoloured, completes all its recursive calls successfully.

**Theorem:** The algorithm just described finds an ordering of the coloured vertices, such that if there is a path from $v$ to $w$, where $v$ and $w$ are coloured vertices, not necessarily distinct, then $w$ appears in the ordering before $v$, if and only if such an ordering exists, and returns an error state otherwise. The ordering is that in which coloured vertices are completed.

**Proof:** Termination is guaranteed in the usual way by setting visit numbers that prevent any vertex from being explored twice. When a vertex is made active and a new visit number chosen, the search is restarted, but this cannot continue forever since it only happens at new vertices which are being converted to active.

Consider what happens when the algorithm returns with an error state. This originated at a coloured vertex that was found to be active when the search arrived there. Although the visit numbers change, in most respects the algorithm is a standard depth-first search. In particular, there is always a path from any active vertex to the current spearhead of the search. Hence, arriving at an active coloured vertex proves that that vertex lies on a cycle. The conclusion is that when the algorithm returns an error, it is correct to do so.

Finally, consider what happens when the algorithm returns success. Every coloured vertex must have been completed, since the outer loop guarantees that every new coloured vertex is made active at some point, then when its recursive calls all end (as they must, by termination), the vertex is completed as required. Let $v$ and $w$ be any two coloured vertices such that there is a path $P$ from $v$ to $w$. It is required to prove that $w$ was completed before $v$. The argument breaks into two cases, depending on which of the two vertices was made active first.

Suppose $v$ was made active before $w$. At the moment $v$ was made active, a new visit number was obtained which re-opened the entire graph to the search, including all of $P$. Hence $P$ must have been searched in its entirety, including making $w$ active later than $v$, and completing $w$ before $v$ as required, *unless* the search was terminated early by reaching an active or completed vertex on $P$. An active vertex would cause an error, which is assumed not to have happened. Reaching a completed vertex $x$ would imply that every coloured vertex reachable from $x$ was already completed, including $w$ as required.

In the other case, $w$ was made active before $v$. If $v$ was never reached in the search out of $w$, then $w$ must have been completed before $v$ was even activated. If $v$ was reached before $w$ was completed, then there must be a path from $w$ to $v$ as well as the assumed path from $v$ to $w$ – in other words, a cycle containing $v$ and $w$. The new visit number obtained at $w$ made the whole graph open to the search from $w$, and so after reaching and activating $v$ the search must have continued down some path from $v$ to $w$ and detected the cycle by revisiting the active $w$, contradicting the assumption that the search returned success. This argument, slightly adjusted, also applies to the case $v = w$. $\square$

### 4.7. Initialization of predefined object features: realization

This section explains how the Nonpareil compiler initializes predefined object features in a safe order, based on the general algorithm just described.

For each non-generic non-enumerated class there is a global array which implements the class's *all_predefined* feature. For the *string* class, for example, this array would be called `string_all_predefined`. For each enumerated class there is a global array which implements the class's *all_enumerated* feature, plus a global trie. For the *char* class, for example, these global variables would be called `char_all_enumerated` and `char_enum_trie`.

These are the only global variables in the generated code. This is very convenient, because it means that only these variables need to be initialized, but it does force the backend name of a predefined object to be an array access. For example, the call

*colour*!*red*

causes code such as `colour_all_predefined[6]` to be generated, rather than say `colour_red` – there being no such variable in the generated code. Each predefined object has its own index, chosen by the compiler so that predefined objects which originate in a class (as opposed to those that enter it from child classes) come first.

The generated function that initializes a non-enumerated class's *all_predefined* array begins with a call to `array_create1`, then generates code for initializing every predefined object of the class. Predefined objects of child classes will have already been initialized, so initializing those entries of the *all_predefined* array requires just an assignment from some other *all_predefined* array:

```
ref_all_predefined[20] = ref_book_all_predefined[5]
```

The array initialization is completed by a call to `array_create2`.

Initialization of an enumerated class's *all_enumerated* array is quite trivial, since it just contains pairs of numbers known to the compiler, so it has no dependencies and is done separately, before anything else. Initialization of the class trie is similar to the initialization of *all_predefined*, in that code for creating the records must be generated and pointers to them assigned to array elements. It is much more complex in practice, however, requiring a multi-level array creation with optimizations that ensure efficiency when there are large identical regions caused by code ranges. A long comment in the source code documents this algorithm.

Both initialization algorithms have the property that it is safe to access earlier elements of the array or trie of predefined objects of a class while initializing later ones, and it has proved possible to allow the user to do this in non-enumerated classes. In enumerated classes, 'earlier' would have to be interpreted to mean 'with lower code number', since the predefined object features are sorted by code number before initialization begins (it is not practicable to initialize the trie in any other order), but it is not allowed in any case. *Why not?*

In the general algorithm, vertices represent back-end functions (since what matters is order of initialization in the *generated* code), and coloured vertices represent class initialization functions. The main issue is to work out the dependencies between these vertices.

In general, back-end function *b* is said to depend on back-end function *a* when static analysis shows that a call on *a* might need to have been completed before some call on *b* can complete successfully. A back-end function that represents a predefined object depends on the back-end function that initializes that object. A back-end function with a body depends on all the back-end functions called within that body. As usual, the default values of any parameters are treated as part of the body of the enclosing function, irrespective of what actual values may happen to replace those default values in particular calls.

The back-end functions that initialize *all_enumerated* arrays (`BEFN_ENUM_INIT`), *all_predefined* arrays or class tries (`BEFN_CLASS_INIT`), and the system as a whole (`BEFN_SYSTEM_INIT`), are never called by user code, but are relevant nevertheless because the root of initialization is the unique system initialization function. Initialization of *all_enumerated* arrays depends on nothing, and the unique system initialization function depends on every *all_enumerated* array and class initialization function.

Working out the dependencies between class initialization functions is one of the two aims of the initialization order algorithm (the other being to check initialization within each class). A class initialization function depends on the class initialization functions of all the child classes of its class (since it relies on them to initialize the child objects that it copies into the class's *all* feature), and on all the functions called from within the bodies of the predefined object features of its class, typically including the class's creation function.

Builtin functions (`BEFN_BUILTIN`), downcast variables (`BEFN_DOWNDEF`), and parameters (`BEFN_PARAM`) have no bodies and depend on nothing.

Let definitions (`BEFN_LETDEF`) and invariants (`BEFN_INVARIANT`) depend on all the functions called from within their bodies. The same is true of creation functions (`BEFN_CREATION`), but for them, such calls can occur only within the default values of their parameters, and within the invariants of their classes (including inherited invariants): the rest of a creation function is built-in and depends on nothing.

The dependencies of features (`BEFN_FEATURE`) remain to be described. These vary with the type of feature and whether the feature's class is enumerated or not, as follows.

Creation features (`FEATURE_CREATION`) of non-enumerated classes depend on nothing. The question of whether the objects they access have been created is a different question; once created, there is nothing to prevent their fields being accessed. Creation features of enumerated classes depend on the class trie to map the code number to the record containing the creation features, so depend on the class initialization function. Any *legal_code* (FEATURE_PREDEF_LEGAL_CODE) and *with_code* (FEATURE_PREDEF_WITH_CODE) features also depend on the class trie.

Noncreation features (FEATURE_NONCREATION) depend on all the functions called in all their bodies, including parameter default values. The bodies of *clone* features (FEATURE_NONCREATION_CLONE) are built-in; each makes one call, to a creation function, and depends only on that function.

The dependencies of user-defined predefined object features (FEATURE_PREDEF) vary with the type of class in a perhaps unexpected manner. A user-defined predefined object feature of an enumerated class is implemented by a code number known to the compiler, so it depends on nothing. A user-defined predefined object of a non-enumerated class depends on the class initialization function of its class, which initializes that object. The system-defined *all_predefined* (FEATURE_PREDEF_ALL_PREDEFINED) and class trie (FEATURE_PREDEF_ALL_ENUM_TRIE) predefined object features also depend on that class initialization function.

Finally, the *all_enumerated* features (FEATURE_PREDEF_ALL_ENUMERATED) depend only on the back-end functions that initializes these features. Since those functions depend on nothing and are called at the very start of initialization, in practice the *all_enumerated* features themselves may be treated as depending on nothing.

The problem of finding a safe order of initialization will now be solved in a simplified form which does not permit user-defined predefined objects to refer to predefined objects defined earlier in the same class. This limitation will be removed later.

As described in the Language Specification, Nonpareil does not find a global initialization order for all predefined object features. That could be done quite directly using the general algorithm, but it would lead to a single, enormous, global initialization function, and also interact badly with trie construction. Instead, the predefined object features of each class are initialized together, in their order of appearance, and the compiler must find a safe order of initialization of the classes rather than their features.

If every predefined object feature of some class was given its own vertex, then cycles between these features would be allowed unless the vertices were coloured. But colouring the vertices would license the algorithm to find an arbitrary order of initialization of these features, when in fact they are initialized in a fixed order within their class initialization function. So it would be a mistake to give every predefined object feature its own vertex. Instead, apart from *all_enumerated* which is ignored altogether, a call on a predefined object feature is taken to be a call on the initialization function of its class, which *is* given a vertex (a coloured one). This reflects the fact that the predefined object features of one class are initialized as a unit. All other back-end functions become uncoloured vertices, and edges join the vertices in accordance with the dependencies described earlier.

A run of the general algorithm on this graph will either produce a safe order of initialization of the classes (the order of completion of the vertices corresponding to class initialization functions) or else it will detect a cycle. The graph is not built explicitly, but rather the expressions making up the bodies of the functions involved are traversed. In their manifested state, expressions contain all the needed information about which back-end functions they call.

Using this algorithm, within an enumerated class, access to a creation variable of another predefined object of the same class causes failure. The algorithm will also fail when one predefined object feature of a non-enumerated class calls an earlier one, but this limitation can be removed as follows.

When entering a vertex representing a class initialization function, carry along information indicating which of the features it initializes caused this vertex to be reached. As the vertex's outgoing edges are traversed, by visiting the bodies of all its constituent predefined objects in order of increasing index, remember the index of the most recently completed predefined object. If the incoming index does not exceed this index, and the visit number has not increased, return success rather than the failure that would usually result from re-entering an active coloured vertex. This modification requires coloured vertices to hold a visit number as well as a state.

## 5. Expressions

### 5.1. Relationship to the front end and back end

Nonpareil expressions are quite similar to C expressions, which makes it feasible to represent both kinds by a single data structure, and this Nonpareil does. An incentive for doing it is that expressions hold the bulk of the program, so avoiding the duplication that would result from having front-end and back-end versions of expressions helps with efficiency.

Nevertheless, expressions have distinct front-end and back-end aspects. The front end of the compiler is responsible for *interpreting* an expression, that is, deciding what all the symbols within it refer to. The back end is responsible for code generation.

To enable these responsibilities to be carried out, each expression is linked to the front-end data structures (class views and front-end features) of the compiler, and also to the back-end data structures (classes and features). Interpretation is carried out while traversing the front-end structures, and code generation is carried out while traversing the back-end ones.

In general, there is no simple relationship between the front-end and back-end points of linkage for an expression. The bodies of noncreation features have to be brought together from possibly many different classes in different modules; the default values of creation features have to be brought together into creation functions; and so on. Handling this dual structure is complex and error-prone.

Two principles serve to keep the complexities under control: each expression is linked to the front-end structures at exactly one point, and interpretation occurs at that point; and each expression is linked to the back-end structures at exactly one point, and code generation occurs at that point.

However, there are situations where these two principles seem too rigid. For example, Nonpareil copies the bodies of features when they are shared, as in

**infix** +, *add*(*other*: *int*) := **builtin** *"int_add"*(**self**, *other*)

This copying is carried out very early. It is a simple method of guaranteeing that such definitions will be treated independently, but when the body has an error it will be found and reported more than once, which is clumsy. A better method would be to link this expression to both features.

A multiply linked expression would be manifested only the first time it is touched during a front-end traversal, but code generated each time it is touched during a back-end traversal.

An alternative method of handling a multiply linked expression is to package it in a function, and replace each occurrence of the expression with a call on the function. The mechanism used to uncover hidden parameters of let definitions (i.e. side-effects in the context) may be used to find what parameters such a function needs. This seems to be the best way to handle default values of creation features, each of which is linked to all the creation functions of the classes it is inherited into. Copying these expressions would be expensive in object code.

### 5.2. Small-scale and large-scale expressions

A Nonpareil expression can be printed in either of two C formats, which we call *small-scale* and *large-scale*. Small-scale format uses C expressions; large-scale format uses C statements. In C, statements may contain expressions but expressions may not contain statements, so we have to classify each Nonpareil expression and subexpression as either small-scale or large-scale, such that no subexpression of a small-scale expression is classified as large-scale.

We prefer small-scale format since it requires no variables to be introduced, but large-scale format is necessary for Nonpareil *let* and *case* expressions, since they map to C blocks and C *case* statements, which are not C expressions. So the algorithm, carried out in the first pass, is to mark all leaf nodes as small-scale, and all internal nodes as small-scale except that *let* nodes, *case* nodes, and nodes with a large-scale child are marked large-scale.

### 5.3. Let expressions and anonymous functions

Anonymous functions are converted during parsing into let expressions. For example,

*list*.*map*(**fun**(*x*: *int*) := *x* + 1)

is converted to

**let**
  *f*(*x*: *int*) := *x* + 1
**in**
  *list*.*map*(*f*)
**end**

where the function name *f* is chosen not to clash with any other name in context.

Let definitions without parameters are handled as C declarations within inner blocks. This is easy; the significant problem is what to do with let definitions with one or more parameters.

During preparation for code generation, all let expressions with one or more parameters within each Nonpareil function are found and moved to a separate list, where they can be generated first and so appear in the C code as helper functions just preceding the C version of the original Nonpareil function. Mutual recursion is possible among helper functions, when one occurs within the body of another, so a list of forward declarations is printed just before the functions themselves, unless there are less than two helper functions. Helper functions are declared `static`.

When a function declaration is removed from its context, there is a danger that variables within the body of the function will lose their meaning. For example:

$f(a, b\colon real)\colon real :=$
**let**
$g(c\colon real)\colon real := a + c$
**in**
$g(a) + g(b)$
**end**

Here the occurrence of $a$ in the body of $g$ will lose its meaning when $g$ is moved out of $f$. The solution to this problem is to make the compiler convert this to

$f(a, b\colon real)\colon real :=$
**let**
$g(a\colon real, c\colon real)\colon real := a + c$
**in**
$g(a, a) + g(a, b)$
**end**

Every symbol which is in scope where the function is defined, but not in scope at the point it will be moved to, must be added to the function as an initial *hidden parameter*, and an expression invoking the appropriate value must be added as an actual parameter to each call on the function. Since *let* expressions and anonymous functions may be arbitrarily nested, this transformation may need to be applied to several enclosing functions.

One simple way to decide which extra parameters to add would be to add one for every variable declared between the function and the outer level, including all parameters and other *let* variables, etc. This is obviously very clumsy and wasteful. The Nonpareil compiler has a clever way of deciding, which adds just those parameters that are necessary. It works as follows.

The algorithm for type checking the body of a feature naturally has a parameter holding the current context – the set of symbols that are visible at any point in the function. As the type checking enters *let* expressions and other places where variables are defined, these variables are added to the context; and as it leaves these regions, they are removed from the context.

Now the context is actually organized as a stack of *levels.* At the outermost level are the visible class names; at the next level are the feature names of the current class, and its generic type parameters. When the feature is entered, its own generic and other parameters are added at a third level. New levels are not ordinarily begun when variables are defined within expressions; instead, these variables are added to the current level and removed when the type checking leaves their scope. However, when an anonymous function or *let* definition with parameters is encountered, a new context level is opened with the function *attached* to it.

When retrieving a symbol from the context, if the search passes through a given context level, and the ultimate result is a symbol of a local type (*let* definition without parameters, downcast variable, or parameter, the last possibly **self**), then before returning, the context search carries out the following steps. For each level passed through, outermost first, it notifies the attached function that this symbol has appeared in its scope, and what that symbol refers to. The function thereupon creates a new hidden formal parameter for itself, using the same Nonpareil name (ironically, this is the only name guaranteed not to cause any conflicts) and stores it inside itself along with the actual value. The function then returns the new formal parameter to the context, which inserts it at that level. This ensures that subsequent calls will find the new parameter, so that it is only inserted once even if invoked many times. An inner level function is informed that the symbol refers to the parameter at the next outer level, not to the original outermost definition.

In this way, hidden parameters are handled at virtually no cost, with the main line of code unaware that they are even being inserted; it just receives symbol bindings back from the context as usual, with the hidden parameters inserted as side-effects.

Formal parameters inserted in this way are marked as hidden, and they move to the front when the function's parameters are sorted (sorting is needed anyway to move optional parameters to before the compulsory ones; the canonical sorting order is hidden first, then optional, then self, then other compulsories). When a call is matched against the function, the first step is to insert artificial actual parameters corresponding to each hidden formal parameter, whose values are guaranteed to cause just the parameter name to be printed during code generation, as required.

### 5.4. Case expressions

The type of test expressions in case expressions may be any enumerated type (including *bool* and *char*), *int*, or *string*. These types have widely differing properties, but they also have some common ones. In particular, they are ordered, and *char*, *int*, and *string* have a much wider range of possible values than there are likely to be distinct cases. Nonpareil handles this complex situation with a single overall code generation algorithm, which can call on a repertoire of three different methods: C switches, nested `if` statements, and hash tables.

Some terminology is needed before we come to the actual code generation algorithm. A *case point* is a particular value of the test expression. A *case interval* is an interval, in the mathematical sense, of values of the case expression – a set of values with no gaps. All case points in a given case interval must have the same associated action in the original case expression; this action is an attribute of the case interval.

A case interval is represented by its two endpoints, each of which may be *closed* (meaning that the endpoint value is included in the interval) or *open* (the endpoint value is not included in the interval). Classified another way, a case interval may be a *singleton case interval* (represent a single case point) or a *range case interval* (not a singleton).

A *case partition* is also an interval of values of the type of the test expression, and its endpoints may also be closed or open. It contains a partition of its interval into a sequence of case intervals sorted by increasing test expression value. By a partition, we mean that every case point in the partition's interval of values must lie in exactly one of its case intervals.

To convert a case expression into a case partition, proceed as follows. Let the case interval's interval of values be the complete set of values of the test expression type. Convert each case label or range of the original case expression into a case interval (which will be closed), sort the case intervals into increasing order, merge case intervals whose endpoints are not separated by gaps and which share the same action, and insert case intervals representing the gaps between one case interval and the next (or before the first case interval or after the last), where gaps exist. These 'gap case intervals' will be open intervals; they have the **else** clause for their action.

The *density* of a case partition is the number of case intervals in the partition, divided by the number of case points in the partition. When the test expression type is *string*, the density of every non-trivial case partition is 0, because it has some finite number of case intervals, but an infinite number of case points.

Here is the well-known example of the number of days per month, as a case partition:

| | |
|---|---|
| *jan* | 31 |
| *feb* | **if** *leap_year* **then** 29 **else** 28 **end** |
| *march* | 31 |
| *april* | 30 |
| *may* | 31 |
| *june* | 30 |
| *july, august* | 31 |
| *september* | 30 |
| *october* | 31 |
| *november* | 30 |
| *december* | 31 |

It has 12 case points and 11 case intervals, so its density is 11/12.

The code generation algorithm for case expressions is recursive, and it accepts as input a single non-empty case partition which it is required to generate code for. The initial call receives a case partition covering the entire range of values of the type being tested; recursive calls receive some sub-interval of that interval. The algorithm is as follows.

If there is only one case interval in the given case partition, then generate the code for that case interval's action and stop.

If there is more than one case interval in the given case partition, and the density is at least 0.2 (this rules out *string*), generate a C `switch` statement with one `case` label for each case point. Given the high density this cannot produce an excessive number of `case` labels, by comparison with the original Nonpareil code, even when case ranges are present in that code.

Whenever a `switch` is generated, as each interval is handled the algorithm looks for other intervals with the same action and generates labels for those as well, so that all invocations of a given action are brought together in the switch.

It is common, particularly with switches on *int*, for there to be an initial large case interval and a final large case interval, separated by a dense region. For example, an *int* switch with a case for each integer between 0 and 9 would have a large initial range (the negative integers) and a large final range (the integers above 10). So check to see whether removing the first case interval and/or the final case interval would leave a remainder amenable to either of the cases handled earlier; if so, insert `if` statements separating these intervals from the dense region, and call the algorithm recursively on the branches of these statements. Slightly simpler code is generated when the initial and final case intervals have the same action, and when the dense region separating them consists of a single point.

If the previous cases do not apply (i.e. if there is more than one case interval in the partition, the density is less than 0.2, and there is no central dense region), then proceed as follows.

If there are more than 16 case intervals in the given case partition, and, when one looks through all the range case intervals in the partition, one finds at most one distinct action, then generate a hash table mapping all the singleton case intervals' case points to small positive integers. Store this hash table in a static variable on the spot which is tested each time this branch is entered, and initialized the first time. Use the value returned by the hash table as the test expression in a C `switch` statement; if the hash table fails to find the lookup value, return 0 indicating that the 'one distinct action' is to be performed.[1]

It remains to explain what to do when there is more than one case interval, the density is less than 0.2, and either there are fewer than 16 case intervals or the range case intervals have more than one action. In these cases, find a test which divides the interval into two subintervals with an equal, or almost equal, number of case intervals. Insert an `if` statement containing that test, then recursively generate code for the two case subintervals that are its two branches.

As described, the algorithm may generate the same code several times over, when it is the common action of several case intervals, as 30 and 31 are in the months example earlier. In these cases, however, instead of generating the code again, the algorithm merely generates a `goto` targeted to the first occurrence.

Here is the output for the months of the year example given earlier, with *int* switch type and 31 given as the default value:

```
if( 2 <= x && x <= 11 )
  switch( x )
  {
    case 2:

      return (leap_year ? 29 : 28);

    case 4:
    case 6:
    case 9:
    case 11:

      CASE_BRANCH_6:
      return 30;

    default:

      CASE_BRANCH_7:
      return 31;
  }
else
  goto CASE_BRANCH_7;
```

[1]This part of the algorithm could be omitted, since the following part can handle its case quite well. It has been included so that large **case** expressions selected by strings will be implemented efficiently, but it will handle cases involving sparse integers or characters equally well. Whether this is really needed is not clear yet.

Another important application of case expressions is in extending the list of character properties beyond those defined in the Unicode database and offered by the *char* class, by means of a case expression on the *char* type. The resulting case partition might well have many range case intervals and be of very low density. Here an optimized trie could usefully replace the hash table, since it permits more than one distinct action to be associated with the range case intervals. This is how the builtin properties of the *char* type are implemented, so if the selection task can be carried out by switching on one of those properties rather than on the character code, that will be more efficient.

A central dense region was identified between 2 and 11 inclusive, within which a `switch` statement was utilized. Label `CASE_BRANCH_6` is redundant; the algorithm does not know that all occurrences of that multi-interval case occur within one switch. If an enumerated type were used, the enclosing `if` statement would not be needed and would not be generated.

Since hash tables and C switches terminate the code generation, the only possible hybridization is to have either of these inside some nested `if` statements. A small enumeration type has density 1.0 and will always generate one C switch on the value. A case expression which selects a few random strings, such as `Helvetica`, `Times`, etc., will use nested `if` statements if there are fewer than 8 strings (hence fewer than 16 case intervals, including gap intervals), and a hash table if there are more.

### 5.5. Default values of parameters

It works out best to consider default values of parameters of a function to be a part of the body of that function. Among other advantages, this permits a graceful implementation of redefined default values in redefined class features: they just go along with the redefined body. This is implemented quite literally, by having a special type of expression, `EXPR_FNHEAD`, which holds a set of default values and a subexpression. An expression of this type is inserted as the root node of the body expression of every function.

When a function is called, for each optional parameter the caller has to signal to the call whether to use a value supplied by the caller for that parameter, or the default value. The pure way to do this would be to place a separate Boolean parameter alongside each optional parameter in the C function, signalling this condition. These extra parameters could be gathered together into a bit vector parameter with information about up to 32 optional parameters.

However, the current implementation uses a simpler method, in which a special value, `DFT_PARAM_VAL`, is passed as the value of each parameter where the default value is to be taken. The body of the function begins with one statement of the form

```
if( p1 == DFT_PARAM_VAL )
  p1 = default_val;
```

for each optional parameter, where `default_val` is the expression given as the default value of that parameter. This code is generated, very naturally, by the `EXPR_FNHEAD` expression, which knows not only the default values but also the parameters they apply to.

Default values may refer to parameters appearing earlier in the same function. The code generation scheme handles this quite naturally, provided only that the optional parameters are processed in the order they are declared in the Nonpareil function.

Because `p1` may be a parameter whose Nonpareil type is a generic variable, the same value of `DFT_PARAM_VAL` must be used for all Nonpareil types. The best choice would seem to be `INT_MIN + 1`, cast to whatever C type is required to keep the C compiler happy. This value accordingly must be excised from the legal range of values of all Nonpareil types. This is fine for enumerated types (whose code values are signed quantities that happen to be never negative) and reference types (it is an odd number, so it cannot be the address of a memory word), but it means that one value is lost from the range of integers and reals. The lost real value must be a very negative one, because the floating-point standard requires that the ‘<’ operation on reals be implementable by the ‘<’ operation on signed integers.

## 5.6. Optimizations

Optimizations such as tail recursion elimination and function call inlining are left to the C compiler. Inlining in particular is quite tricky, especially if constant function parameters are to be inlined themselves in the generated code, and since most C compilers (including GCC) do these optimizations, at least at their higher optimization levels, Nonpareil leaves it to them.

There are two exceptions, made basically to simplify the generated code. The first is that a call to a creation feature of a non-enumerated class will be inlined, converting (say)

```
nlist_head(self)
```

to

```
(self)->head
```

The second is that a call to a noncreation feature will be inlined if its body consists entirely of a call on a builtin function with the same parameters in the same order as the enclosing function. For example, instead of generating

```
int_add(int_mul(x, x), int_mul(y, y))
```

the generated code will be merely

```
((x*x) + (y*y))
```

The inlined functions remain available, and are used in curried and unclassed calls.

# 6. Memory

In its quest for speed, Nonpareil caches objects and function calls, and it calculates predefined objects just once, during an initializing run, storing them in a binary file that can be loaded quickly at the start of each ordinary run. This section explains all this in detail.

## 6.1. Memory systems

A *memory system* is a large piece of memory within which objects may be stored. A memory system obtains large *chunks* of memory from the operating system and doles them out in small pieces (*records*) as required. Each new large chunk is twice as large as the previous chunk, starting from about 1MB, so even on a very memory-hungry run there should be less than 10 chunks in a memory system. If a request arrives for a record larger than the current chunk size, that record goes into a fresh chunk of its own.

Each record contains a type tag from which the length of the record can be deduced, and no memory is ever freed, so there is no need for hidden memory management fields of the kind used by *malloc*. Thus there is zero memory allocation overhead per record.

Requiring objects to obtain their memory from a memory system rather than directly from *malloc* speeds up memory allocation, but it is done mainly so that some central authority knows what memory has been allocated, enabling the memory system operations described here.

One chunk in every memory system is devoted to that memory system's *cache* – a large hash table giving access to every record in that memory system. A cache is always the sole occupant of its chunk. When a cache is 80% full it is rehashed into a new, larger chunk. The old chunk is held in reserve by the memory system and re-used next time a new chunk is required, if large enough.

Nonpareil caches both objects and function calls, and at each moment there is a *current object memory system* and a *current function call memory system* into which new objects and function call records are stored. These two memory systems are accessed via global variables.

An object memory system cache is a simple linear probing hash table, with null values denoting empty slots, and non-null values pointing to objects. A function call cache also uses linear probing. Its non-null values point to records containing a length field (the number of parameters), then the parameters (one per word, the first being a pointer to the function that this is a call on), and finally the result value.

Each record, whether object or function call, contains a 32-bit *signature*, which is just the hash value before reduction to the table range via the remainder operation. Storing this signature has a significant memory cost, but it costs next to nothing in time and speeds up retrieval (the key operation), since unequal signatures rule out an object immediately, while equal ones almost always indicate that the full match will succeed.

Actually the signature is stored primarily for a different reason: it greatly simplifies the rehashing operation that occurs when the hash table is enlarged. With a signature available, the new insert position is just the remainder when the signature is divided by the new table size. Without a signature, one must rehash the record, which is slow and surprisingly difficult to get right. Rehashing using full information about the layout of the record requires a huge dynamic dispatch over all concrete record types. Alternatively, rehashing based on (say) just the length of the record requires all unused fields within the record to be carefully cleared when it is created. When creating an object, one then cannot simply hash the parameters of the creation function; instead, the object must be built and hashed, then relinquished if it is found to already exist. This is slow when the object exists already.

### 6.2. Object creation and caching

Whenever an object is to be created, the cache of the current object memory system is consulted. If an equal object is already present in the cache, that object is returned instead of creating a fresh object. In this way the system ensures that two objects are equal if and only if they are the same (lie at the same memory address).

For example, consider the problem of creating an object of the *nlist* class:

**class** *nlist*{*A*} **inherit** *list*{*A*}

*head*: *A*
*tail*: *list*{*A*}

**noncreation**

…

**end**

The C record generated for representing *nlist* objects is

```
struct nlist_rec {
  unsigned int signature;
  unsigned short type_tag;
  unsigned char swizzle_bits;
  unsigned : 8;
  void *head;
  list tail;
};
```

Every object contains at least its own signature and a type tag. The C function generated for creating an *nlist* object is

```
nlist nlist_create(void *head, list tail)
{
  nlist self, other;
  unsigned int sig, i;

  /* calculate hash signature */
  sig = (nlist_tag << 16);
  sig += ((int) head);
  sig += (((int) tail) << 1);

  /* return previously created object if in hash table */
  i = (sig % npsys_obj_cache_len);
  while( npsys_obj_cache[i] )
  {
    if( (npsys_obj_cache[i]->signature == sig) )
    {
      other = ((nlist) npsys_obj_cache[i]);
      if( ((other->type_tag == nlist_tag)
          && (other->head == head)
          && (other->tail == tail)) )
      return other;
    }
    i = ((i + 1) % npsys_obj_cache_len);
  }

  /* create and initialize a new object */
  self = ((nlist) npsys_obj_new(16));
  self->signature = sig;
  self->type_tag = nlist_tag;
  self->head = head;
  self->tail = tail;

  /* insert new creation into hash table and return it */
  npsys_obj_cache_insert(((CACHE_OBJ) self), i);
  return self;
}
```

This function has one parameter for each creation variable. All code excerpts are actual, unmodified Nonpareil compiler output.

The first step is to hash the parameters, producing the signature, held in variable `sig`. If either creation variable had had a default value, the correspondong actual value passed would have first been tested for equality with a special value indicating that the default value was to be used, in which case the value of an expression which evaluates that default value would have been assigned to the parameter. How to handle that expression is a separate topic.

Then, accessed via global variables `npsys_obj_cache` and `npsys_obj_cache_len`, the cache of the current object system is searched for an object equal to this one. If found, that other object is returned. Otherwise, a new record of the appropriate length (16 bytes in this example) is obtained from the current object memory system, initialized, inserted, and returned. The `swizzle_bits` field is not initialized by object creation. It is described in a later subsection.

Array and string creation is somewhat different. The *array* type is built-in, with C struct

```
struct array_rec {
  unsigned int signature;
  unsigned short type_tag;
  unsigned char swizzle_bits;
  unsigned : 8;
  unsigned int length;
  void *elems[1];
};
```

The C struct for the *string* type is the same except for having a specific element type:

```
struct string_rec {
  unsigned int signature;
  unsigned short type_tag;
  unsigned char swizzle_bits;
  unsigned : 8;
  unsigned int length;
  uchar elems[1];
};
```

These definitions are printed by the compiler, but are fixed, not calculated like the one for *nlist*.

There are several functions that create arrays, all builtin. These are required to follow a certain protocol. First they call `array_create1`, which creates a record of the right length for the particular array. Next they assign the elements using any method they choose. Finally they call `array_create2`, which checks whether an array equal to the new one is already in the cache; if no, it returns the new array; if yes, it relinquishes the new array and returns the existing one. For example, the code to initialize a string might look like this:

```
s = string_create1(5);
s->elems[0] = 'H';
s->elems[1] = 'e';
s->elems[2] = 'l';
s->elems[3] = 'l';
s->elems[4] = 'o';
s = string_create2(s);
```

Functions `string_create1` and `string_create2` are clones of `array_create1` and `array_create2`. Like the structs, these functions are fixed, not generated by the compiler. They are kept in a library file which is included with every compile.

A problem here is that the algorithm for filling the array (or string) might involve function calls and object creations, and so a decision to relinquish the array might be taken after many other objects have been allocated, when the array is no longer at the top of the chunk.

The Nonpareil memory system handles this situation very simply. If, at the moment of relinquishment, the array is still at the top of the chunk, then the memory is reclaimed, otherwise it leaks. Such leaks will be very rare, since most array construction operations do not involve the creation of other objects anyway (e.g. literal string construction and array concatenation), and given that the array in question is to be relinquished, any objects that it points to must have already existed at the time the array construction began.

If an array leak occurs during the initialization of predefined object features, the leaked array will be swizzled and saved with the other predefined objects. Then, when they are reloaded, it will be inserted into the cache with them. It is of course equal to at least one other object (the original array that caused it to leak), but the cache reinsertion operation will not notice this, because it is optimized for reloading under the assumption that no two objects are equal. It might seem that this does not matter, since the original array must precede the leaked one in memory, so will be reinserted first and will prevent the leaked array from ever being retrieved. However, a rehash of the cache (which traverses the hash table, not memory) would reorder them when there is an array wraparound between the original and leaked arrays, and then, if a new object creation of an array with this value occured, `array_create2` would return a reference to the leaked array, and a subsequent equality test would return false when it should return true.

The solution adopted to this problem is as follows. Just before relinquishing an array or string `self` within `array_create2` or `string_create2`, assign a value to its first element (if any) that cannot appear in any ordinary array:

```
if( self->length > 0 )
  self->elems[0] = DFT_PARAM_VAL;
```

The value assigned is the one used as the default parameter value. This ensures that a leaked non-empty array will never be confused with any other array, except possibly another leaked array, but that does not matter. This does not solve the problem for empty arrays and strings, but they can never leak, because the protocol permits only one way to create them:

```
a = array_create1(0);
a = array_create2(a);
```

and no code is ever generated in which other code appears between these two calls.

Function call caching is similar to object creation, in that an object is created only if it does not already lie in the cache. However, the object returned is not the created object, but rather one of its fields, which holds the function call result. One cannot unify object caching and function caching, because of this need for returning a field rather than the cached object itself.

### 6.3. Checkpointing and swizzling

After initializing all the predefined objects, a Nonpareil initializing run has to save them in a binary file that can be quickly loaded by a subsequent non-initializing (called *loading*) run. That is, it has to save the current object memory system. The current function call memory system is not saved, nor is the object memory system's cache. This is quite safe, because no object contains a pointer into either type of memory.

When objects are saved to a file, any pointers within them lose their meaning. They must be converted to suitable integers before saving, then converted back again to pointers when the file is reloaded on a later run. These operations will be called *swizzling* and *unswizzling*.

Nonpareil keeps track of the start and end address of each chunk. Corresponding to each byte address in each chunk is a unique integer, obtained by adding the offset of that address within the chunk to the total length of all preceding chunks. To checkpoint object memory, this integer is calculated for each pointer in object memory, by testing which chunk the pointer points into, and overwritten onto the pointer before the chunk is written to disk. When the disk file is read back in it goes directly into a single chunk, and each pointer field is incremented by the base address of that chunk. This reload is quite fast, fortunately, since it is done at the start of each loading run, whereas writing is done only by initializing runs.

The binary file that holds a checkpointed memory system has a very simple format. It begins with a long integer (traditionally called a *magic number*), holding the time when the executable file that wrote this file was compiled. This is checked to ensure that the file is read only by an executable compiled from the same system as the executable that wrote the file. After that comes a single swizzled pointer, pointing into the following chunk to an array of pointers to the *all_enumerated*, *all_predefined*, and trie arrays of the memory system, from which all the other objects are reached. After that comes the memory system itself as a single swizzled chunk.

The name of the file ends in `_be` or `_le` accordingly as it was written by an executable running on a big-endian or little-endian system. When reading, a big-endian executable will only open a binary file whose name ends in `_be`, and a little-endian one will only open a file whose name ends in `_le`. In this way all confusion over endianness is avoided, indeed the same system can be compiled on both big-endian and little-endian machines in a networked environment. All that is required is that an initializing run be performed on one machine of each type.

To swizzle an entire memory system, swizzle each chunk of it. To swizzle a chunk, proceed through the chunk swizzling each record. This requires both knowing how to swizzle one record, and knowing its length so that one can proceed to the next chunk. The swizzling algorithm depends on a function with header

```
int system_swizzle(object self, object other)
```

which handles both: it swizzles `self` and returns the size of `self`'s record, so that the chunk swizzling knows how far to skip along the chunk to the next record. Parameter `other` is concerned with working out which fields to swizzle, as explained below.

To implement `system_swizzle`, one needs a giant dynamic dispatch on the tag field of `self`. To ensure that the C compiler is not overwhelmed, the body of `system_swizzle` contains nested `if` statements that dispatch to a module (this is easily done since the tag fields of types defined in any one module have contiguous values). Within each module is a function which uses a `case` statement to dispatch to swizzle functions for each of that module's types.

Unswizzling is organized in the same way, by proceeding through the chuck calling a `system_unswizzle` function that dispatches to the modules, which then dispatch to the object types. Unswizzling means converting integers back to pointers, but these functions also recalculate object signatures, if required owing to the presence of fields containing pointers, and insert the objects into a new cache.

### 6.4. Knowing what to swizzle

Swizzling requires that the system know which object fields are pointers. Nonpareil's strong typing usually makes this clear, but there is a problem with fields whose type is a variable:

**class** *nlist*{*x*} **inherit** *list*{*x*}

*head*: *x*
*tail*: *list*{*x*}

**end**

In this example, *tail* is always a pointer, but *head* could be a pointer or an immediate value, depending on the instantiation of *x*. The record type tag holds only the uninstantiated type tag value `nlist_tag`; it contains no information about the type of *x*.

One solution to this problem would be to avoid swizzling altogether by ensuring that a saved object file is reloaded into the process address it was stored from. However, the operating system used by the author (Linux) does not offer such a feature, despite possessing a full virtual memory addressing system. The manual entry for the Linux *mmap* function, which comes closest, allows the user to ask for it, but deprecates it and states that the request will not necessarily be granted.

Another solution might be to ensure that the type tag stored in the object record identified an *instantiated type*, that is, a type with particular values for its type variables. Unfortunately, it is not clear how to come up with instantiated type tags. Consider the *map* feature of *nlist*:

*map*{*B*}(*f*: *fun1*{*A*, *B*}): *list*{*B*} := *nlist*(*f*.*apply1*(*head*), *tail*.*map*(*f*))

It creates an object of type *nlist*{*B*}, and it is not clear where an instantiated type tag with this value will come from (it varies from call to call, to begin with).

Nonpareil's solution involves storing a set of *swizzle bits* in each object. There is one bit for each generic type parameter of the class which is also the type of a creation feature. For example, in *nlist* above, there would be one swizzle bit, for *x*; or if *head* had some other type, there would be no swizzle bits. Each swizzle bit is 1 if, in this particular object, the type variable is a pointer type, and 0 if it is not.

Swizzle bits are not inherited. A parent class may have several swizzle bits, while its child has none, and vice versa, as is easily shown by example. But for exactly this reason, the location of the swizzle bits in the object record does not have to be the same from one object to another. So these bits (if needed at all) are allocated *last*, after all other record fields have received their offsets. An upper limit of 32 is placed on the number of swizzle bits in any object – a generous limit – so that the swizzle bits will always fit into a single field. The actual width will be 0, 8, 16, or 32 bits depending on how many swizzle bits there are.

In deciding which swizzle bits to set, Nonpareil relies on the fact that only one checkpoint is needed, at the end of initialization. It runs the initialization twice, in different object memory systems, then compares the two object systems object by object and variable-type field by variable-type field. Where a variable-type field has a different value in each object, those values must be pointers into different memory systems, indicating that that field must be swizzled.[1]

[1]If floating-point operations produce non-deterministic results, that could confuse this method; indeed if these results are used in tests which determine what objects are created, that could even lead to cases where the two object memory systems cannot be compared object by object. I have not been able to find out whether such non-determinism exists in floating-point operations. If it does, it would be necessary to avoid predefined objects containing generic fields whose values depend on the outcome of floating-point operations.

This comparison is part of the swizzling operation; the swizzle bits are set at the same time the fields themselves are swizzled, and only objects destined for saving in a file have their swizzle bits set. The two objects to be compared are passed as parameters `self` and `other` to `system_swizzle` above. Then when the object is loaded again later, known pointers are unswizzled, known non-pointers are left alone, and the appropriate swizzle bits decide the doubtful cases.

## 6.5. Function call caching

As already mentioned, Nonpareil caches function calls. How the hash table is organized was described earlier. This section describes how the cache is accessed, and which kinds of functions are cached.

Each back-end function object contains a Boolean field saying whether or not calls on that function are to be cached. Irrespective of what kind of back-end function it is, if that field is set then during code generation of the body of the function, code is generated at the start of the function (before the preconditions are tested) for hashing the parameters, searching the function call cache, and returning early with an existing result if found. Code is also generated at the end of the function for building a cache object holding the signature, result, and parameters, and inserting it into the cache.

The remaining question is which calls to cache. Provided every *object* is cached, there is no problem with correctness in not caching some function calls.

At the time of writing there were twelve different kinds of back-end function objects. Some of them do not correspond to functions in the C code, and thus caching is irrelevant to them: let definitions without parameters, downcast variables, parameters, creation features, and predefined object features. Creation functions are not cached because object caching does the equivalent for them. Back-end precondition and invariant functions have a `void` result type, not Boolean, so it would be awkward to cache those. However, since they have the same parameters as their enclosing function, and a cache hit on one of them would usually imply a cache hit on the enclosing function, which would cause the precondition or invariant to be bypassed anyway, it would be rare for time to be saved by caching them, and so they aren't. The various initialization functions are not cached, because each of them is called only once anyway. Builtin functions are not cached because, at least so far, their bodies are all too small to benefit from caching, and indeed are usually inlined.

This leaves noncreation features and let definitions with parameters, both of which are cached, and creation feature default value functions, which are currently not cached, on the principle that they are likely to be tiny functions (constants, typically) which would not benefit. It would make sense to heuristically evaluate all these functions to decide whether each is likely to benefit from caching or not, but this is not currently being done.

# 7. Code generation

Code generation is the actual production of C code. C-specific material is concentrated in the `CODEGEN` class, whose various capabilities are the subject of this section.

## 7.1. Assigning C names

Most Nonpareil entities have corresponding C entities which must be assigned C names. The names must be ASCII identifiers and must not conflict with C keywords and other standard names, such as `printf`, `INT_MAX`, and the like. When assigning such names, notice must be taken of C scope regions: most C names are global, but *struct* field names are local to their *struct*, parameter names are local to their function, etc.

A C scope region is modelled by a symbol table containing the C names that have previously been declared present in that region. A table modelling the global C scope is created and initialized with the C keywords and a large number of identifiers gleaned from the standard libraries. Identifiers beginning with '_' are omitted, since they can never conflict with Nonpareil identifiers.

Whenever a Nonpareil symbol requires a C name for itself in some C scope region, it registers its own name with the appropriate C scope object, and receives back a modified form of its own name which is new in that scope, which it saves and makes available to other parties wishing to refer to it. Registration also adds this new name to the C scope. The returned name is the same as the Nonpareil name, except that punctuation characters and non-ASCII characters are swizzled into aphanumeric characters, and a suffix `_1`, `_2` etc. is added to the name if required to make it new in the scope. Common punctuation sequences have predefined conversions: <= is converted to `le`, etc. In practice, virtually all of the C names produced in this way are identical with their Nonpareil antecedents – a great aid to readability.

## 7.2. Assigning C types

Since the front end does a complete type check using rules of inheritance and genericity that are beyond the reach of the C compiler, the system as a whole places no dependence on the C compiler's type checking for its correctness. The generated code may thus be said to be untyped, in the sense that it would not matter if it was itself compiled with no type checking.

Although the generated code is untyped in principle, it is typed in practice. For every Nonpareil type there is a corresponding C type. For variable and meet types, which have no correlative in the C type system, `void *` is used. For class types, a typedef is inserted which defines the class's name (or some approximation of it if necessary) to be either a pointer to a record of its creation features, or (for an enumerated class) either `char`, `short`, or `int` depending on its cardinality. Generic types are assigned a single C type which does not reflect their actual generic parameters. For example, there is one `nlist` C type for all *nlist*{*x*} types.

This mapping is far from perfect, of course. In particular, a validly typed Nonpareil program is not mapped by it into a validly typed C program. Whenever an expression's back-end type differs from the back-end type expected by the context, a type cast is inserted. This handles all kinds of type mis-matches without worrying about classifying them as upcasts, genericity casts, etc. What it does not do is provide a check on the front-end type system, since it assumes that every back-end type mismatch is safe to cover with a cast.

A cast is usually just a C type cast. However, the rules of C forbid casts between `void *` and `float`, so to carry out these casts the generated code calls a function that assigns the value to a union field of one type and retrieves it from an overlapping field of the other type.

### 7.3. C files and the *np* and *np_init* binaries

The Nonpareil compiler produces two binaries: *np_init*, which initializes all the predefined objects, swizzles them, stores them into a checkpoint file, and exits; and *np*, which loads a checkpoint file, unswizzles the objects it contains, and proceeds with an ordinary run. The two binaries share the same magic number, written into the checkpoint file to ensure that the binary that reads it is compatible with the binary that wrote it. The magic number is just the starting time of the run of the Nonpareil compiler that generated those binaries.

The binaries are generated from two kinds of source files: *module files*, which hold code related to particular modules, and *system files*, which hold code related to the system as a whole. System files are of two kinds: *generated system files* and *copied system files*.

The module files consist of three `.c` files generated for each module *m*. These are normally called `m_init.c`, `m_load.c`, and `m.c` (other names will be chosen if these happen to conflict with the names of existing files). Each has its own header file, `m_init.h`, `m_load.h`, or `m.h`, containing `extern` declarations for the global objects in the corresponding `.c` file.

File `m_init.c` contains everything from module *m* specifically concerned with initializing predefined objects, swizzling them, and saving them in a checkpoint file. File `m_load.c` contains everything from module *m* specifically concerned with loading a checkpoint file and unswizzling its contents. File `m.c` contains everything else from module *m*.

The generated system files are `system.c`, `system_init.c`, and `system_load.c`. Each has its own header file as usual. These files contain all the system code that varies from system to system (because it calls module code) and hence must be generated by the compiler.

The copied system files are `npsys.c`, `npsys_init.c`, and `npsys_load.c`. These files contain all the system code that is the same in all systems (memory allocation code, etc.), and so they are copied over from a library directory, not generated by the compiler. Files `npsys_init.c` and `npsys_load.c` both contain a `main` function.

Files `system_init.c`, `npsys_init.c`, and `m_init.c` for each *m*, are called *initialization files*. Files `system_load.c`, `npsys_load.c`, and `m_load.c` for each *m*, are called *loading files*. The rest are *other files*. Compilation begins by compiling every `.c` file into a `.o` file in the usual way. Then binary *np_init* is made by linking all the initialization and other files, and binary *np* is made by linking all the loading and other files.

It would be simpler to have just a single binary, with a command-line flag to say whether to initialize or load. However, there are thousands of predefined objects, and the code for initializing these objects is very large – larger than the checkpoint file itself. Having separate binaries makes *np* a lot smaller; all the initialization code is in *np_init*, which may be thrown away, or left to moulder on disk, after it is run.

### 7.4. Saving C compile time

Although the Nonpareil compiler translates Nonpareil to C reasonably quickly, the C files it generates are quite large. Particularly awkward are the *char* class, with its thousands of character attribute record creations, and the *string* class, with its thousands of predefined strings (most of them originating in the attributes of Unicode characters). File `lang_init.c` contains all of this material. At the time of writing it is 16683 lines long (not too bad considering the *char* class alone is 7340 lines), and takes about 40 seconds to compile on my system.

The compiler has a simple but effective method of avoiding the compilation of this file most of the time, copied from SmallEiffel [1], although doubtless it is used elsewhere too. The method is applied to every `.h` and `.c` file, either generated or copied, as follows.

If there is an old version of the file present, move it to a different name (for `lang_init.c` the different name is `lang_init.c~`). Then generate or copy the file as usual, remembering the non-system files that were `#include`d into the file. This is easy to do for generated files, because there is a specific call on the codegen module for making such an inclusion. For copied files it is necessary to detect `#include` directives in the file while copying it.

Later, when the time comes to compile the file, first compare all the included files for equality with their old versions, then compare the file itself with its old version. If no old version of some file exists, that is equivalent to being not equal. If all files compare equal, then compiling this file will produce exactly what it did last time, so if there is an object file present, just use it without recompiling.

Some included files will be compared with their old versions multiple times if the plan above is followed literally. However the back end maintains a symbol table of the names of files that have already been compared with their old versions, and what the outcome was in each case. The file comparison only occurs if this symbol table does not already contain the answer.

Generated files include a time of generation in an initial comment. The test for file equality ignores any inequality within this initial comment.

Unfortunately, whenever the programmer adds a new string literal anywhere in the source code, it appears as a new predefined object in `lang_init.c`, which must then be recompiled. There does not seem to be any good way around this problem: creating the string literals in place would execute very much more slowly, and per-module initialization of strings would be messy.

## 8. Unresolved issues

The following language and implementation issues were unresolved at the time of release of Version 1.02 of the Nonpareil compiler.

*Redefinition of creation features from optional to compulsory.* It may be safe to allow the redefinition of a creation feature if it is converted from optional to compulsory by the redefinition. Is this safe? Is it useful? Should it be allowed?

*Redefinition of noncreation features into creation features.* Eiffel allows a noncreation feature to be redefined as a creation feature. It is easy to see that this is a kind of shorthand for introducing a new creation feature and redefining the noncreation feature to just return the new creation feature. Presumably this proves that it is safe. Is it useful? Should it be allowed?

*Standard library.* Version 1.02 does not come with a standard library. Various classes that would be part of such a library have been implemented, but only in so far as they make useful tests. I need to review various standard libraries (Haskell, OCaml, Java etc.) and take the time to carefully design a Nonpareil standard library.

*Range types and coercion.* When the compiler matches a call against a signature, it checks that each actual parameter's type is a subtype of its formal parameter's type. This subtype test simultaneously updates constraints on unification variables, and returns a coercion indicating how the actual parameter needs to be coerced. The unresolved question is whether this coercion is calculated too soon. Should we wait until after the whole call is matched and unification variables are converted into range variables?

*Superfluous casts.* Type casts can often be omitted in C, when they are the 'usual arithmetic conversions.' But the compiler tediously inserts them all. They look silly on constants such as

```
(int) (short) 0
```

which records the chain of coercions that the compiler used to convince itself that the *byte* constant 0 could be used in an *int* context. Since these coercions are builtins anyway, these useless coercions can probably be avoided by implementing them as identity functions rather than as casts.

*Unnecessary C variables.* The generated code has some unnecessary C variables. These arise when an expression is large-scale only because of a single point buried within it. Every subexpression on the way to that point gets a variable, but in fact only one variable is needed, for the point itself; calculating that subexpression and storing it in a variable should make everything above it small-scale.

*Copying of shared bodies.* When two features share a body, at present that body is being copied, leading to duplication of C code. It is hard to see how to do it any other way, if the features are to be treated as independent, at least for noncreation features, and it is probably not a significant issue in the overall system, since sharing is not very frequent anyway. But still it would be better avoided.

## References

[1] Dominique Colnet and Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, the GNU Eiffel compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, pages 341–350, 1999.