Combining Logical
Assertions, Inheritance,
Relations and Entities

The μCLAIRE kernel

Version 4.0 (DRAFT)

Yves Caseau

Table of Contents

0. INTRODUCTION

This document describes the Go µCLAIRE Kernel, in the same spirit that the previous "microCLAIRE" document described the C++ implementation. The Go Kernel replaces the C++ Kernel, which has lasted more than 20 years, and which was itself inheriting from previous LAURE Kernels (LISP then C). Thus, one can think of this µCLAIRE Kernel as a Go package which is a toolbox for implementing a LISP interpreter. As Go provides its own memory management and garbage collection, this Kernel is significantly simpler than the C++ version.

More precisely, the content of the Go Kernel can be summarized with the following list of capabilities:

- Generic entity management (a common type that represent the statement "everything is an entity"). These are called EID in Go (vs OID in C++).
  EID management is a key feature of the Go Kernel, which is much more efficient that the native Interface() provided by Go.
- Reflective access to objects (dynamic read/write), and reflective method calls (using function calls). Here, because Go is stricter that C, the Kernel code is longer and more cumbersome than the C++ equivalent in CLAIRE 3.5. However, this is the only way to work along Go memory management. The benefit is a set of reflective capabilities which is much faster than what is natively provided by Go.
- Exception Handling : a large part of the complexity of the implementation comes for exception handling which is not supported by Go and relied here on EIDs. Notice that the real burden in on the CLAIRE compiler that generates more complicated code to handle the possibility of an exception at any moment.
- Implementation of the classical CLAIRE data structures: list, array, sets, maps, using the underlying Go equivalent data structure. This kernel includes the redefinition of basic capabilities such as sets and file I/O, for performance reasons.

This note is organized as follows. The next section describes the key principles of the Go implementation, that is, how CLAIRE entities (objects, functions, primitive) are represented in the Kernel. The second section is a short presentation of the 6 files that make the Go kernel. The last section illustrates how CLAIRE 4.0 generates Go code and make use of the Kernel objects and functions.

The current implementation of the µCLAIRE Kernel (version 4.0) is made of approximately 8000 lines of Go code. The µCLAIRE Library is the combination of this Kernel and the compiled Core package, that is the result of compiling the Core module (2400 lines of CLAIRE code that generate 10000 lines of Go) into Go. This document is a first DRAFT, it is intended to help any developer who want to join in maintaining CLAIRE 4.0. All comments are welcome. This first version tries to stay "*as simple as possible but not simpler*".

Conventions for this document:

- <n> represents the Claire object with name n (e.g., <class>)
- If n is a named object or a class, [N] represents the capitalization of the name:
  - Replace first char by capital letter (class -> Class)
  - Replace _ by subsequent capitalization (made_of -> MadeOf)
- #<m> is the function name that CLAIRE generate for method m.
  It is usually simple : foo@c -> foo_c, but it may become more complex with polymorphic definitions (when there are many restrictions of foo define on class c)

1. PRINCIPLES FOR GO IMPLEMENTATION

1.1 Modules

CLAIRE code is organized into files, which are grouped into modules. A module is defined as a namespace and a list of instructions (content of the file). CLAIRE uses go **packages** to group instructions and to provide a compiled name space.

- Any CLAIRE module m with associated files (m.made_of not empty) is represented by a package m
- Modules with no files are intermediary nodes in the module hierarchy and are not represented in Go
- The CLAIRE compiler generates three kinds of Go items with names : variables, classes and functions.

CLAIRE is more sophisticated than Go: a named object with name that is defined in module M may be placed in module N to benefit from visibility rules. A named object N/x:M (object with name N/x defined in module M) is represented with a Go variable defined in package M:

```
Var C_N_x , if N is not <claire>, and var C_x
```

The capital C is a both a CLAIRE mark and a way to ensure Go visibility (must start with a capital letter)

A CLAIRE class <c> is represented two ways in Go:

- A Go class (see next section)
- A Claire object (meta-description) which is accessible through a Go variable:
  var ClaireC *Claire[C] for Kernel classes, and var ClaireC *[M][C] if c is defined in m

Last, CLAIRE uses go functions and go methods. CLAIRE uses a go method – because it makes the Go code more readable – whenever possible:

- When the CLAIRE method is defined in the same modules as the CLAIRE domain class (Go constraint)
- When the CLAIRE method is "simple enough" in terms of polymorphism, that is all other arguments have the same type. This is a big difference with Go, as soon as one uses polymorphism seriously, CLAIRE must rely on go functions.

A method foo(x:c) defined in module m with be represented with the following Go definition in package m:

```
• func (x:*Claire[C]) Foo()     when  a method pattern may be used
• func #<foo@c>(..)                     otherwise
```

When any of these three (named objects, classes or functions) are used, Go name scoping rule apply:

- when using an item from the same package, or when using the Kernel package which is always imported with "import (. Kernel)", no prefixing is necessary
- otherwise, when using an item from module m, prefixing is necessary: m.C_x, m.NClass, m.Foo(), etc.

A module <m>, like a class, is a named object in CLAIRE. It is represented in Go with two things:

- The associated package m
- The CLAIRE object <m> which is made accessible with the "it" variable defined for each package : m.it is a variable with range *ClaireModule that contains <m>. Notice that since <claire> is a module that is not represented with a package, C_claire is the go variable that contains <claire> (root of the module tree)

This document is mostly about the Kernel module, which is written in Go, and made of the 6 files that are described in Section 2. Other modules are generated by the Claire compiler, as (briefly) explained in section 3.

1.2 Classes

Go does not really provide classes with OOP capabilities, but rather structures with single inheritance implemented with embedding. This is what is used by CLAIRE: each CLAIRE class becomes a structure in Go, each slot becomes a field of the structure, and the single inheritance is represented with embedding. If we have a CLAIRE class X that inherits from Y with some slots: a:int, b:list, c:X, d:set<Y>, the Go representation will be:

```
type ClaireX struct {
    ClaireY
    a int
    b *ClaireList
    c *ClaireX
        d *ClaireSet
}
```

As mentioned earlier, a CLAIRE class is represented by two things in the Go implementation:

- The previously mentioned "go type", with a name go_class(c) = Claire|[M][C].
- A *ClaireClass object, contained in a variable C_c (kernel) or C_m_c (when c is defined in m)

A class object is defined with a statement : C_x = MakeClass("name",<Y>,<m>)

1.3 Methods

Methods are restrictions of properties, defined with a lambda-expression in CLAIRE, and represented with a Go function. Properties are named objects, created with MakeProperty (cf. ClReflect.go).

Because CLAIRE is both interpreted and compiled, each method has a dual representation:

- A native representation, that is a function or a method, that is used in compiled code. There is no other access that knowing the name of the method/function when writing or generating code. A native representation means that all arguments use the native associated Go representation.
- An EID reflective function, that is used with dynamic calls by the interpreter. An EID function means that all arguments are passed as EIDs. A reflective function means that the Go function (E_f_c pattern) is also represented by a CLAIRE object (*ClaireFunction) that can be manipulated (to perform the dynamic calls).

Let us start with the native representation for method foo(x:c1;y:c2). The simple case is when we can use a go "method" on the class c1. As expressed earlier, it requires

- That foo(…) is defined in the same package as c1
- That all restrictions of foo have the same "codomain" = the list of types of other arguments (here the codomain is C2).

When this is the case (a "simple method") the Go native representation for foo is

```
func (x *ClaireC1) Foo( y *ClaireC2)   <range>  { … }
```

Otherwise, for a « complex method", we use a Go function as the native representation :

```
func  F_foo_C1( x *ClaireC1, y *ClaireC2)   <range>  { … }
```

The name of the go function, which we note #<foo>, is usually F_<property>_<class>, but may be more complex for polymorphic method (see the CLAIRE compiler's documentation).

Let us now turn to the EID representation of the method. We always generate a function (never a go method) with a name that starts with E_ (vs F_) :

```
func  E_foo_C1( x EID, y EID)   EID  { … }
```

Notice that most of the time (99%) these functions are generated automatically to call their native counterparts with the proper native arguments, so that there is no code duplication. In some very rate case the EID function is written separately for performance (allocation) optimization. There are some examples in the Claire Kernel, for instance in ClBag.go

1.4 Primitive entities

As mentioned, a key principle in Claire is that everything is an entity that belongs to a class. We use the word "sort" to designate a "kind of entity" with a shared implementation principle. There are 4 sorts in the Go implementation, plus EID that represents the implementation pattern to handle an entity whose sort is not known at compile time:

- CLAIRE objects are represented with "pointers to struct". The structure is obviously the one associated to the class as explained previously. The slot "isa" is the first slot of any Claire class, so each object can access its class through o.isa. There are two kinds of objects: the regular objects whose class is a subset of <object> and the primitive objects (subset of <primitive>) that are defined and described in the Kernel
- Claire integers are represented natively as Go "int" (64 bits signed integers).
- Claire floats are represented natively as Go "float64"
- In addition to these 4 native representations that covers everything in Claire, the Go Kernel proposes EID, a struct with two parts:
    - Type EID { PRT *ClaireAny, VAL uint64}

The Go Kernel provides the implementation of the following primitive entities as CLAIRE objects:

- Strings: a string is a an object that contains a Go string (as a Value field), and the reflective access to the class <string>. The indirection is needed because Go slides are immutable and CLAIRE slides are not.
- Functions: EID functions are first-class citizens of the Go Kernel. Because Go is strict with arity, the class ClaireFunction comes in many flavors : ClaireFunction<i> for EID functions for arity i. Notice that only the EID functions have a reflective equivalent in CLAIRE : a ClaireFunction<i> has a field "call" that contains a Go function of arity I, a field "name" that contains a string, and a field "arity" that contains i.
- Ports: ClairePort offers the implementation of a UNIX FILE using the standard functions of Go.

Let us now address the multiple representations (native / object / EID) for each sort of entity:

1. Objects are represented with an EID whose PTR value is the native object (a pointer) and whose VAL field contains 0. By construction, any object (of type *ClaireC) is also a *ClaireAny (because of Go embedded structure inheritance). So for objects there only two kinds of representation : native (a pointer) or dynamic (EID).
2. Integers are represented with an EID whose PTR value is C__INT (a special marker) and whose VAL is the integer. Integers, when manipulated by the interpreter, are represented with primitive objects: the class <integer> has an equivalent ClaireInteger Go class to produce objects that represent integers. A ClaireInteger is a pointer to a struct with two fields: the "isa" class and the "value" which contains the integer. This means that integers exist in 3 forms: native (int), object (the integer is contained in an object wrapper of type *ClaireInteger) and dynamic (EID).
3. The story is the same for floats : there exists a ClaireFloat structure class (that inherits from ClaireAny) use to represent a float as an object.
4. Last, characters also have an object representation, ClaireChar, where the Value field contains a Fo rune.

Booleans are Claire objects, with two constant values, CTRUE and CFALSE of Go type *ClaireBoolean.

Exceptions are also Claire objects, with classes and slots as with any other Claire object. The only difference is that in the EID representation, the VAL is set to 1 (to accelerate error handling).

1.5 Objects

At this point, we already know that CLAIRE objects are represented in Go with pointers to a struct element from the associated Go class.

However, inheritance is only partial in Go. If A is a subclass of B and x an object from A represented with a *ClaireA, then access to a slot from B (x.s) or to a method (x.foo()) will work, but function calls (foo(x)) will not. This is why CLAIRE requires a casting function for be defined for each class C :

```
func To<C>(x *ClaireAny) Claire<C> {  return (*Claire<C>)(unsafe.Pointer(x)) }}
```

These functions are defined in ClKernel.go for the Go Kernel. For other objects, they are produced automatically by the compiler. To illustrate method inheritance, the reverse operation (casting from a specific class to the generic any class) is done with:

```
func (x *ClaireAny) Id() *ClaireAny {return x}
```

this function is used as a casting macro everywhere in the code, when a *ClaireAny is expected.

Instantiation (creating a new object) is left to Go new(…) operator so that Go performs memory management. To be able to create, access and modify objects dynamically, the CLAIRE Kernel uses the following principles:

- Everything is a pointer to a struct
- All structs have 64bits fields, of 3 kinds : int, float64 or *ClaireAny
- Claire has a series of dummy classes : ClaireDummy<i> (i in 1 to 6) to create and manipulate objects from size 1 to 30. This will expanded in the future to cover the case of "longer" objects with more slots.
- Creation and Updates are only performed with Go safe functions (cumbersome, but GC-friendly).

Global variables in CLAIRE are represented with package variables in Go.

1.6 Bags

Bags are a key feature of CLAIRE that is implemented in this Go kernel with the different kinds:

- Lists are ordered and mutable in Claire. They are represented with ClaireList go objects. For performance reasons, the Kernel implements three types of ClaireList: ClaireListObject, ClaireListInteger and ClaireListFloat, who respectively contains objects (any kind), integers (and only integers) and floats. First, note that this separation into 3 is unknown to CLAIRE, it is an "implementation feature" of the Go kernel that does not exist with the C++ implementation. Second, notice that, since everything is an object, ClaireListObject is always the default choice, especially when the type of the list is not known.
- Arrays are fixed-size lists and are implemented with ClaireList in the Go Kernel
- Similarly, tuples (immutable lists of fixed size) are also implemented with ClaireList
- Sets are implemented with ClaireSet. After trying unsuccessfully the "map" Go structure, this Kernel version is similar to the C++ Kernel and uses sorted list as the underlying implementation principle. This is not totally satisfying as reflected in the performance benchmarks (Java does a much better job). Because we rely implicitly on lists, we find in the Kernel three specialized subclasses : ClaireSetObject, ClaireSetInteger and ClaireSetFloat.

Each bag has a "isa" slot (its class), an "of" slot for typed bags, and a Values field. For list, the Values field contains a go slice, of type *ClaireAny[], int[] or float64[] based on the kind of list. To improve performance, each list has a "srange" field that contains <object>, <integer> or <float> to reflexively know whether the list is a *ClaireListObject, *ClaireListInteger or a *ClaireListFloat.

Creation of a bag follows the same pattern:

bag<t>(a1, a2, … an) in CLAIRE becomes Make<bag>(t,a1,…an)
for more complex expression, the bag may be created in successive steps:

```
v := t.Empty<bag>()
v.addFast(a1) …
```

Access to a bag is either generic using the .At(i int) and .PutAt(i int, x *ClaireAny) methods, or it can use the native access using Values when the sort is known (.Values[i]). In the Go code, indexation follows Go convention : the index i ranges from 0 to n-1 (where n is the length of the bag). This is different from CLAIRE that adopts the mathematical range (1 to n). Hence the Kernel code, as well as the code produced by the compiler, includes a “-1” shift when translating from Claire to Go.

Iteration is based on indexed access :

```
for k := 0; k < s.Length(); k++ { x := s.At(k) ….}
```

Bags are implemented mostly in the ClBag.go file that will be described in the next section.

2. GO KERNEL

2.1 ClKernel.go

As the name suggests, this is the core file of the Kernel. It is also the only one that uses the “unsafe” Go package (so that the rest of the Go code is rather clean). This file contains the variable definition of all the objects which are instantiated in ClReflect.go. Most importantly it contains the Go description of the CLAIRE taxonomy

This file is divided into five parts:

(1) **EID**. This part contains the definition of the EID struct and the most used methods. For instance, OBJ, INT, FLOAT and CHAR are the value extraction “macros” (there are no macros in Go but the compiler works well).
There are two reverse functions to move from EID to Object (generic) representation:
`func ANY(x EID) *ClaireAny {}  &   func  (x *ClaireAny) ToEID() EID {}`

(2) **Classes**. This part the Go definition of all the structures that are associated to CLAIRE classes, starting at the root with ClaireAny (associated with <any>). Each structure is defined (with the list of slots : name & range), together with a constructor for primitive classes and the casting method “ToC” that was described earlier.

(3) **Kernel variables**. This is where we find the Go variable associated to each named object from the Kernel.

(4) **Dummy Classes**. This part contains dummy classes and their associated methods for dynamically creating / reading and updating a CLAIRE object.

(5) **Unsafe utility functions**. This last part contains a few low-level high-performance functions that use “unsafe” casting to optimize performance. These include IsIn (class inclusion, the heart of object membership), Equal (deep equality), Of and findRestrictions (used for dynamic calls) plus all the functions used for dynamic calls (funcall and stack_apply)

2.2 ClReflect.go

This file contains the reflective definition of the Kernel classes and objects. This matches closely clKernel : the first file is the Go description for the compiler, the second is the reflective description for the CLAIRE interpreter.
This file is divided into five parts :

(1) **Reflective definition of classes**. This is the CLAIRE “bootstrap”, where we define the first objects from which other are later constructed. It starts with “bootCore” where the first handful of objects is created by hand because of circularity. The “boostrap” is the Go version of code that used to be defined in CLAIRE in earlier version. It contains the class inheritance tree.

(2) **Property methods**. This part contains the functions that are used to create and manage properties, and their restrictions : slots and methods. The core methods are MakeProperty, AddSlot and AddMethod.

(3) **Class and Type methods**. This is where we find MakeClass() (and its variants) that are used to create the classes in Part1. We also find the methods to create a new module (MakeModule) and to instantiate a class (create a new instance from a class). The two preferred patterns used by the CLAIRE compiler are (respectively for <object> and <thing>) :

```
new(ClaireC).Is(C_c)            // creates a new instance of c
new(ClaireC).IsNamed(C_c,MakeSymbol(…))      // creates a new instance of c
```

(4) **Reflective description of classes (slots and methods)**. This part contains the description of each class listed in Part1, as far as their slots are concerned. This is where the c.AddSlot(p *ClaireProperty, range *ClaireType, default *ClaireAny) method is used. The rest is the reflective definition of the CLAIRE methods from the Kernel. That is, the bootMethod() function creates all the *ClaireMethod objects, while the actual functions will be found in the rest of the files.

(5) **Read & Write**. This last part contains the read and write functions: how to read the value of a slot and how to write a new value. This shows an example where the EID functions are optimized to avoid unnecessary allocations.

2.3 ClEnv.go

This file contains the "CLAIRE environment", that is mostly about stack management and exception handling. It is also organized into five parts :

(1) **System Object (ClEnv)**. ClEnv is a large static objects that regroup system/environment values, such as the evaluation stack, the world management stacks, debug stack, environment variables, etc. We also find in this first part the few low-level functions necessary for the CLAIRE evaluator (which is mostly defined in CLAIRE in the Core module). String buffer management, time management, shell functions are other examples of "system functions" found here.

(2) **Exception handling**. This part contains the few functions which are necessary for handling error:

- Cerror() for creating a Kernel error.
- Close() which ends the instantiation of an error object and produces an EID with the error code 1
- ErrorIn(x EID) bool {} which is THE macro to see if x is the result of an exception

(3) **World Management**. This is the heart of "defeasible update" management (what we call worlds in CLAIRE). World management is organized with multiple stacks to keep track of any "defeasible" update so that we can backtrack to a previous state (a "world"). This is the core mechanism for constraint programming and combinatorial optimization through search trees (see CLAIRE documentation). The Go Kernel uses a ClRes object that combines all the necessary stacks (similar to C++ implementation). All the defeasible update method share the "store" prefix: store_list, store_object, … while the world management methods are world_push(), world_pop() and world_remove().

(4) **Dictionaries**. This Go implementation of the CLAIRE Kernel leverages Go maps to provide with first-class-citizen dictionaries (called map_set in CLAIRE 4).

(5) **Reader functions**. This last part contains the functions that are used for the read part of the `print(eval(read()))` of CLAIRE's top level. As mentioned in the introduction, the CLAIRE kernel contains the lego pieces to build an interpreter. Although most of the CLAIRE reader is defined (recursively) in CLAIRE in the Reader module, the low-level part (read a number, read an identifier, read a string) are define here. The primitives (read a char) are defined in ClString.go (ClairePort methods).

2.4 ClBag.go

This file contains the implementation of lists and sets in CLAIRE. The concept that regroups lists and sets is called bag (cf. 1.6). <u>This file is divided into four parts</u> :

(1) **List Objects**. This part contains all the methods that are used to create a new list, and all the methods that are proposed in ClReflect.go to manipulate lists. Although a ClaireList is just a wrapper around a Go slice, the existence of three kinds (object, integer, float) yields code that is pretty long (because CLAIRE inherits from LISP, it comes with a long list of methods for lists). It is important to notice that each list l has two fields:

- `l.of  with range *ClaireType, which the parametric type of the list`
- `l.Srange with range *ClaireClass, which is the sort of the list content : integer, float or any`

Here we need to remember that lists in CLAIRE comes in three kinds: immutable (of = empty), mutable and not typed (of = any) and typed (of = t means that we have a list<t>).

(2) **Sets**. This part contains the CLAIRE 4 implementation of sets, based on sorted lists. As for list we have 3 sorts (integer and floats, with the regular ordering, and any with the ordering implied by SKEY(x *ClaireAny) which is a kind of hash function).

(3) **Tuples**. Tuples are represented as *ClaireList, but their "isa" slot contains C_tuple instead of C_list. The code is mostly restricted to instantiation and conversion methods.

(4) **Arrays**. This is quite similar : arrays use the *ClaireList representation; so they are simply lists with fixed sizes, without append or deletion methods.

2.5 ClString.go

This file contains the string and symbol library for CLAIRE. It contains the usual string library functions (using the standard "Rune" Go representation) and the more specialized symbol functions that implement the concept of namespaces.
It is divided into five parts :

(1) **Characters**. Characters are based on Go rune. A ClaireChar is simply a wrapper around a Rune. However, to avoid allocation costs of creating too many ClaireChar, they are pre-allocated and placed in ClEnv.ascii table. The most specific function is `c_princ(r  rune)` that embodies the conversion CLAIRE -> Go : special characters have an equivalent that is used when CLAIRE idents are translated into Go ident (for instance, + becomes _plus).

(2) **Strings**. This part contains the definition of functions for creating and handling strings.

(3) **Symbols**. This part contains the code that is necessary to manage symbols. A symbol is a Claire object (*ClaireObject) that represents an identifier: a name, a module to which the symbol belongs (each module is a namespace), and a "defined" module to keep track where the symbol was created. Note that all fields from ClaireSymbol are private, they can be accessed from Claire only through the API methods.

(4) **Module definition**. This part contains the management of ClaireModule objects and the functions that handle namespaces. Each ClaireModule, in addition to its CLAIRE slots, had a private field called "table" which is a map[string] that represents the namespace (name:string -> symbol).

(5) **Port objects**. This last part contains the definition of ClairePort objects. A port is a "enum-kind" struct that can represent four kinds of ports (it contains all the necessary fields, some of which are used and some of which are not, depending on the value of p.status):

- File buffer for reading (using a string buffer to read faster), associated to a UNIX file. The status is 0.
- File buffer for writing (status = 1), associate to a UNIX file. The most famous example is stdout (stored in ClEnv.Cout).
- String buffer output (status = 2): the port is a string buffer into which we can add characters. We get the resulting string when we close the port.
- String buffer input (status = 3), used to read from a string
- File buffer for reading on char at a time without buffering, such as stdin (stored in ClEnv.Cin). This is represented with status = 4.

This fifth part contains the methods to create the port (F_fopen_string), start reading/writing and then close the port. The most important methods are the GetChar (getc in CLAIRE) and PutChar (putc in CLAIRE). Because of the utf8 encoding of runes, the code is much more complex than the C++ code of CLAIRE3.5.

2.6 ClUtil.go

This file contains both additional utility functions that complete the Go Kernel and the minimal subset of type methods. This is an important difference with CLAIRE 3.5 (C++) : Go does not accept circularity in module import, hence Kernel must be a self-contained package. This has led to include the minimal description of the type system and the associated methods into the Go Kernel. This also explains why the Go kernel for CLAIRE is as long as the C++ Kernel even though it is much simpler (no memory management). This file is divided into four parts:

(1) **Type system**. This part contains the definition of the key methods for the type system. Recall that the type taxonomy (set of classes : classes, unions, intervals, parameter selection) is described in ClKernel/ClReflect. The most important methods are:

```
t.Contains(x *ClaireAny)  => x belongs to t ?
Included(y *ClaireType)   => x <= y
t.Member(),               => type t2 such that (x % y) & y:t => x % t2
At(p *ClaireProperty),    => x:t => p(x) % t.At(p)
Union (y *ClaireType)
```

(2) **Utility functions**. This part includes debug and printing functions).

(3) **Integer functions**. Here we import integer arithmetic functions from Go to CLAIRE. This is rather straightforward since the native representation of CLAIRE integers are int.

(4) **Float functions**. Same thing for floats (float64 in Go).

3. GO CODE GENERATION

The purpose of the CLAIRE kernel is two-folds: to support the implementation of the interpreter and to provide the "lego blocks" for the compiler's code generation. This third section gives a short overview of the Go fragments that are produced by the compiler for CLAIRE expressions and CLAIRE statements. This section is "Work in Progress" and will expand in the future. The goal is to help understanding the Go code generated by the compiler.

3.1 Expressions

The most central question is how CLAIRE calls (like p(x,y)) are compiled. Statically-typed calls (when the compiler can find which method should be used) are compiled into method/function calls:

```
F_p_c(x,y)   or x.P(y)
```

In the other situation, we use the call method – defined in the Core module - that implements dynamic binding (finding which method of p should be used at run-time):

```
Core.F_CALL(p,ARGS(x,y))
```

Note that ARGS is a listargs-method that is found in ClReflect.go that pushes all its arguments on the eval stack.

Reading a slot value ("call slot") is straightforward (x.s) unless we have to check for the possibility of an unknown value. In that case, CLAIRE uses x.s.KNOWN(C_s) which transforms CNULL (representation of unknown) into an exception.

Lists and Sets are created with the previously mentioned constructors:

```
MakeList(ToType(C_string.Id()), MakeString("a"))
MakeConstantSet(MakeInteger(1).Id(),MakeInteger(2).Id())
MakeTuple(C_class.Id(), C_type.Id())
```

Access to lists (or arrays, tuple, sets) is made either with the generic properties (when the underlying sort "srange" is uknown) or by reading the "Values" field. ClKernel.go contains three "access methods with casting" to read this Value field for each of the possible "srange":

```
func (x *ClaireList) ValuesO() []*ClaireAny
func (x *ClaireList) ValuesI() []int
func (x *ClaireList) ValuesF() []float64
```

Tables in CLAIRE4 are of three kinds: (1) indexed by integers with a finite range (one-dimensional arrays), (2) indexed with two integers with finite range (two-dimensional arrays) and (3) relations between any CLAIRE entities that are implemented with dictionaries. The slot "param" tells which kind of table we are using while the slot "graph" is, accordingly, a one-dimension list, a two-dimension list or a dictionary.

The CLAIRE compiler produced Boolean expressions of two kinds:

- Native bool Go code that is used in test conditions (If).
  Notice that Go does not hold a construct for conditional expressions (the famous ((x == 1) ? 1 : 2), hence the code is more cumbersome. If the values are really simple (no side effects), CLAIRE kernel implements IfThenElse((x == 1), 1,2).
- CLAIRE object code (that produces a *ClaireBoolean) otherwise

An important difference between Go and the previous target languages is that assignment are no longer values (as in x := (y := 12)). All assignments must be considered as statement.

The Go kernel defines the following useful constants:

- `CNIL and CEMPTY, to represent an empty list and the empty set`
- `CNULL : the representation of <unknown> as a *ClaireAny`
- `EVOID : the EID associated to CNULL`
- `CTRUE and CFALSE, the two Boolean values.`

3.2 Statements

Statements in CLAIRE have a direct equivalent in the Go language which makes code translation easy. Conditional statements are a perfect example:

```
if (bool-expression(test)) {
    statement(arg)
    }else {
    statement(other)
    }
```

The same thing applied to Do (block of statements in CLAIRE) or While/Until. What makes the generated code more complex to read is the management of exception handling (cf. next section).

Iterations – which may be very sophisticated in CLAIRE - are mostly managed by macroexpansion until we reach the simpler form (iteration of an interval or a bag) which is a direct use of Go for statement :

```
For k := start; k <= end; k++ {…}
For k := 0; k < l.Length(); k++ { v = l.At(k) …}
```

CLAIRE supports "breaking from an iteration" (with return(x)) which is implemented in Go with "break" statement.

3.3 Exceptions

The most important feature that CLAIRE is adding to Go is exception handling.

The first major difference between CLAIRE4 and CLAIRE3 - because of exception handling – is that if an exception may be raised in the body of a method, then the generated Go function has range EID, so that the result may be tested:

```
v := x.P(y)
    if ErrorIn(v) {result = v
    } else { ….
```

The try/catch statements in CLAIRE are compiled into the following patterns:

```
{h_index := ClEnv.Index     // store the position of the evaluation stack top
  h_base := ClEnv.Base      // base is the position of the bottom of the current stack slice
  result = F_p_c(x,y)            // whichever statement is being tried …
  if ErrorIn(result) && ToType(C_error.Id()).Contains(ANY(result)) {
          ClEnv.Index =
          ClEnv.Base = h_base    // return stack to its previous state
          result = F_foo_c()     // whichever expression is the catch
}}
```

3.4 Worlds (defeasible updates)

The management of "worlds" is based on the following principles:

(1) CLAIRE uses a set of seven group of 3 stacks, for each of the six categories : integer slots (slotInt), float slots (slotFloat), object slot (slotObj), list of integers (listInt), list of floats (listFloat) and list of object (listObject) plus dictionaries which have only one sort (dictObj)
(2) For each group, there are three stacks to store : (a) the thing that is being modifed ("Rec" suffix), the index (slot index or position in the list, with "Index" suffix) and the value ("Val" suffix) which is either an int, a float or a *ClaireAny
(3) Each defeasible update x.s := y or l[i] := y records the receiver (x or l), the index and the previous value. The stacks will be used to restore the previous values when we backtrack
(4) Stacks are managed with slices that are called "worlds". When we create a world, we add an empty slice, when we return to a previous world, we restore all the values contained in the associated slice. Hence a world is represented with a set of pointers (integer) that define the slices of the 6 groups. As for the evaluation stack, we use "*base" to represent the bottom of the slice and "*Index" to represent the top of the slice (which is the top of the stack). For instance flBase is the base pointer for the 3 float list stacks (the prefix are i,f,o for object stacks, od for dictionaries, and il,fl and ol for list stacks).

The stacks and the pointers are all regrouped in the ClRes structure (of type ClaireResource), defined in ClEnv.go. The maximum size of the stacks is a parameter that is controlled with the -s option (which control the MAXSIZE global variable of the Go kernel).

INDEX