Specification Framework for Data Aggregates

This work appears in the proceedings of "Journées Françaises des Langages Applicatifs", Journal Bigre 69 of July 1990.

Christian Queinnec*
ICSLATeam†
Internet: queinnec@poly.polytechnique.fr
Laboratoire d'Informatique de l'École Polytechnique
92128 PALAISEAU Cedex — France

Keywords: Data Representation, Type Specification, Reflective Types, Garbage Collector Synthesis.

Abstract

The representation of data aggregates is fundamentally made of concatenation and/or repetition of smaller representations. These structurations may be arbitrarily composed to form complex aggregates of primitive representations such as characters, integers or pointers. Knowing its structuration allows the interpretation of a sequence of contiguous bits as an instance of a type represented by the given structuration. This paper presents a formalization of data representations, it identifies and analyses the features which describe primitive representations as well as representation structurers which allow to compose data representations. These features can be very naturally expressed in terms of objects, classes and methods.

Our model makes structurations fully explicit: some computations can be performed on them. We will show how to derive a general data inspector and a garbage collector from them. We also present an extension of the subclass concept based on concatenation of representations. All these capabilities participate to the definition of a low level machinery for a powerful and generalized memory management.

Primitive data are fixed sized sequences of bits that the computer can handle primitively. Naturals, integers, floats, characters are examples of such primitive data. Capabilities of these data merely depend on computers which impose their size, their set of associated operations and their performances. References i.e. pointers are also primitive data. They have a size and associated operations like dereferenciation and pointer arithmetics. Primitive data have a limited range but are atomically operated.

Necessary is the need to handle larger data: vectors, arrays are possible *structurations*, especially for numerical computations. Traditionally, instances of such structured types are represented by the concatenation of their component representations. References are sometimes inserted to represent composed subparts, for instance an "`array [10][5] of integer`" can be a vector of 50 integers or a vector of 10 references towards vectors of 5 integers.

Strings of characters are more versatile since several instances of the same `string` type may have different lengths. Two major *representations* exist for strings: à la Pascal, with the number of components as prefix or, à la C, with a null character as suffix. The first representation limits strings to a maximum length whilst the second representation excludes the use of the null character. Like arrays, strings are repetitions except that their length is found in the instance rather than in the type.

Records are physically represented by the concatenation of the representations of their fields. Fields are not restricted to be homogeneous but languages often impose them to have fixed lengths in order to compute, at compile-time, all the offsets needed to access the various fields of the record. Once again references may be inserted in place of subparts with varying length.

*This work has been partly funded by Greco de Programmation
†Interprétation, Compilation et Sémantique des Langages Applicatifs

So far, primitive data can be gathered by concatenation or repetition to form more complex data aggregates. Supplementary informations can appear in the exact representation of these data to allow their undestanding i.e. lengths of repetition, special markers or padding bytes. Representations are thus either primitive or composed thanks to *representation structurers*.

Types stand for sets of objects having a common behaviour. Usually instances of a type share a common representation which is deduced from the nature of the type: that is why types and representations are often confused though behaviours and physical representation must not. *Type descriptors* are the run-time support of types in dynamically typed languages like Lisp [Steele 84] or Smalltalk [Goldberg 83]. Type descriptors often contain informations depending on types such as the set of methods which can be applied on their instances. Type descriptors also provide all the parameters needed to read, write or inquire instances of the associated type.

This paper presents an uniform model describing data representations. The model is fully orthogonal and extensible, has a well defined semantics and will be proved to be worthwile for implementors who have to design memory managers in full details and, probably also, to handle data they had not formerly designed. The model is currently used to generate the memory manager of an experimental Lisp system. Any useful data abstraction (including stack frames) is given a type descriptor defining its structure. From these type descriptors are derived data inspectors as well as garbage collector methods. Other data, related to the operating system or useful libraries, are also described and managed by our system.

The first section presents the usual representation structurers such as concatenation, fixed or variable repetition. The second section contains the equations defining the accesses to instances whilst the third section deals with the allocation of such instances. The link between the model and object oriented programming [Briot 87] is commented upon in section four. Section five presents some new kinds of representation structurers and their associated equations. Eventually, various applications will be presented such as — type coding for transmission or persistency, — generalized data inspection, — garbage collector synthesis and — inheritance by similitude.

1 Basic Structurations

Let us provide some convenient notations for structurations. Let $\tau, \tau_1, \ldots\ \tau_i$ be types. Basic types corresponding to primitive data will be noted `Int`, `Char`, `Ref` ...

A representation structurer is an operator that combines representations. Three basic representation structurers are recognized: concatenation, fixed repetition and variable repetition.

- **Concatenation** The representations of the subparts are concatenated and form the representation of the whole part. For example

  ```
  record R is
    C: Char ;
    I: Int ;
    end record
  ```

  An instance of record `R` may correspond to $\boxed{\texttt{c}}\boxed{\;\texttt{i}\;}$.

  Concatenation of types $\tau_1, \tau_2 \ldots \tau_n$ will be noted $\tau_1 \times \tau_2 \times \ldots \times \tau_n$. Therefore the record `R` type is $\texttt{Char} \times \texttt{Int}$.

- **Fixed Repetition** A fixed number of subpart representations is concatenated and form the representation of the whole part. For example

  ```
  array 1..4 of Char
  ```

  One instance is $\boxed{\texttt{c}}\boxed{\texttt{h}}\boxed{\texttt{a}}\boxed{\texttt{r}}$. The number of repetition does not appear in the instance since it is always the same and can be easily derived from the type descriptor.

  A fixed repetition of type τ will be noted τ^n. The "`array 1..4 of Char`" type is then noted $\texttt{Char}^4$.

- **Variable Repetition** A variable number of subparts is concatenated together with supplementary informations that indicates the exact number of subparts. Variable repetition will be, for the moment, the Pascal version i.e. the representation of the repetition will be prefixed by the number of its components. This number is coded as an integer. A string of three characters thus corresponds to $\boxed{\texttt{3}}\boxed{\boxed{\texttt{F}}\boxed{\texttt{o}}\boxed{\texttt{o}}}$.

A variable repetition of type τ will be noted $\tau^\star$.

It is simple to imagine arbitrary combinations of these different structurations.
For example, $\boxed{\boxed{1}\boxed{\texttt{T}}}\boxed{\boxed{3}\boxed{\texttt{b}}\boxed{\texttt{a}}\boxed{\texttt{r}}}$ is an instance of "**`array 1..2 of string of Char`**", more compactly noted as $(\texttt{Char}^\star)^2$. The structure of the instance has been outlined above but is no more than its flat counterpart: `01 54 03 62 61 72`. To imagine that such objects can exist is easy but more difficult is to explicit how they can be read or written or even allocated. These behaviours can be formally derived from their structurations.

2 Access

Instances will be noted ϵ. Offsets within instances will be noted ι, offsets are zero-based and are measured from the beginning of the instance. Offsets may always be counted in bits but in some case a bigger unit (byte, word or anything else) may be more appropriate. A part of an instance ϵ, located at offset ι within it, will be identified by the couple (ϵ, ι). Subparts of (ϵ, ι) will be numbered from zero and indexed by the variables j or l.

To be able to access instances of arbitrary structurations made of concatenation and fixed or variable repetition, the following questions must be answered:

- $\mathcal{S}$(ize) – What is the size of a subpart ?
- $\mathcal{N}$(umber) – What is the number of subparts ?
- $\mathcal{D}$(isplace) – What is the offset of a given subpart within the instance ?

When a part (ϵ, ι) is associated to a type τ then $\mathcal{N}(\epsilon, \iota, \tau)$ is the number of subparts of the part beginning at offset ι within instance ϵ according to type τ. $\mathcal{S}(\epsilon, \iota, \tau)$ is the size of the part beginning at offset ι within instance ϵ according to type τ. $\mathcal{D}(\epsilon, \iota, \tau, j)$ is the offset of the j^{th} subpart of part (ϵ, ι) according to type τ; τ must be of course compatible with j i.e. admit at least j subparts.

Besides functions $\mathcal{N}$, $\mathcal{S}$ and $\mathcal{D}$, we will also use $\mathcal{R}$(ead) and $\mathcal{W}$(rite) to read or write fields of instances. $\mathcal{R}$ and $\mathcal{W}$ are limited to basic types only. $\mathcal{R}(\epsilon, \iota, aBasicType)$ decodes the bits beginning at offset ι within instance ϵ and returns an instance of $aBasicType$. $\mathcal{W}(\epsilon, \iota, aBasicType, anInstanceOfBasicType)$ writes the bit representation of $anInstanceOfBasicType$ into the bit-slice corresponding to $aBasicType$ and beginning at offset ι within instance ϵ. $\mathcal{R}$ and $\mathcal{W}$ are the ultimate coercers in the sense that they are the only functions which know exactly what are primitive data and how they are coded. They translate slices of bits into values and back. In the dynamically typed language Lisp, a character is a first class value which may be represented by $\boxed{\texttt{Char-tag}}\boxed{\texttt{0}}\boxed{\texttt{0}}\boxed{\texttt{ascii-code}}$ whilst this same character is only archived as a byte within a string. To extract a byte from a string creates a first-class character while writing a character in a string coerces it back to a byte.

The general equations for $\mathcal{N}$ are rather simple:

$$\boxed{\begin{array}{l}\mathcal{N}(\epsilon, \iota, \tau_1 \times \tau_2 \times \ldots \times \tau_n) = n \\ \mathcal{N}(\epsilon, \iota, \tau^n) = n \\ \mathcal{N}(\epsilon, \iota, \tau^\star) = \mathcal{R}(\epsilon, \iota, \texttt{Int}) \\ \mathcal{N}(\epsilon, \iota, aBasicType) = \perp\end{array}}$$

The number of components of a n-ary cartesian product or a n-ary fixed repetion is naturally n. The number of components of a variable repetition is the integer which prefixes the repetition; this integer is located at the current offset. Eventually since a basic type has no components, it is an error to apply $\mathcal{N}$ on it. This error is noted $\perp$.

The equations for $\mathcal{S}$ and $\mathcal{D}$ are mutually recursive:

$$\boxed{\begin{array}{l}
\mathcal{S}(\epsilon, \iota, aBasicType) = aBasicType.\mathtt{size} \\
\mathcal{S}(\epsilon, \iota, \tau_1 \times \ldots \times \tau_n) = \sum_{j=1}^{n} \mathcal{S}(\epsilon, \mathcal{D}(\epsilon, \iota, \tau_1 \times \ldots \times \tau_n, j-1), \tau_j) \\
\mathcal{S}(\epsilon, \iota, \tau^n) = \sum_{j=0}^{n-1} \mathcal{S}(\epsilon, \mathcal{D}(\epsilon, \iota, \tau^n, j), \tau) \\
\mathcal{S}(\epsilon, \iota, \tau^\star) = \mathtt{Int.size} + \sum_{j=0}^{\mathcal{N}(\epsilon,\iota,\tau^\star)-1} \mathcal{S}(\epsilon, \mathcal{D}(\epsilon, \iota, \tau^\star, j), \tau)
\end{array}}$$

The size of a basic type is a known constant named $aBasicType.\mathtt{size}$. The three last equations are basically the same. The size of a composed part is the sum of the sizes of its subparts plus the size of any supplementary information needed in the instance to manage it.

$$\boxed{\begin{array}{llll}
\mathcal{D}(\epsilon, \iota, aBasicType, j) = \perp & & & \\
\mathcal{D}(\epsilon, \iota, \tau_1 \times \ldots \times \tau_n, j) = & \textbf{if}\ \ j = 0 & \textbf{then} & \iota \\
 & \textbf{elseif}\ \ j < n & \textbf{then} & \iota' + \mathcal{S}(\epsilon, \iota', \tau_j) \\
 & & \textbf{where} & \iota' = \mathcal{D}(\epsilon, \iota, \tau_1 \times \ldots \times \tau_n, j-1) \\
 & \textbf{else}\ \ \perp & & \\
\mathcal{D}(\epsilon, \iota, \tau^n, j) = & \textbf{if}\ \ j = 0 & \textbf{then} & \iota \\
 & \textbf{elseif}\ \ j < n & \textbf{then} & \iota' + \mathcal{S}(\epsilon, \iota', \tau) \\
 & & \textbf{where} & \iota' = \mathcal{D}(\epsilon, \iota, \tau^n, j-1) \\
 & \textbf{else}\ \ \perp & & \\
\mathcal{D}(\epsilon, \iota, \tau^\star, j) = & \textbf{if}\ \ j = 0 & \textbf{then} & \iota + \mathtt{Int.size} \\
 & \textbf{elseif}\ \ j < \mathcal{N}(\epsilon, \iota, \tau^\star) & \textbf{then} & \iota' + \mathcal{S}(\epsilon, \iota', \tau) \\
 & & \textbf{where} & \iota' = \mathcal{D}(\epsilon, \iota, \tau^\star, j-1) \\
 & \textbf{else}\ \ \perp & &
\end{array}}$$

Since basic types have no subparts, it is an error to compute offsets within them. It is also an error to compute with out of range indexes. The three last equations for $\mathcal{D}$ are similar: the offset for the j^{th} subpart is the offset for the $(j-1)^{th}$ subpart plus the size of the j^{th} subpart. These expressions can naturally be simplified if subparts have all a same size: incremental sums can be turned into products and will thus be computed more quickly.

An Example

Given an instance and its type (or, more generally, any type into which one wants to cast the instance), the above equations allow to extract every subparts of the original instance. Consider, for example, the following program excerpt:

```
type τ is (Char*)² ;
var ϵ: τ ;
ϵ := ["T","bar"] ;
ϵ[1][j] := 'g'
```

The latter assignment can be translated into

$$\mathcal{W}(\epsilon, \mathcal{D}(\epsilon, \mathcal{D}(\epsilon, 0, (\mathtt{Char}^\star)^2, 1), (\mathtt{Char}^\star), j), \mathtt{Char}, \mathtt{'g'})$$

which, once simplified, becomes

$$\mathcal{W}(\epsilon, \mathcal{R}(\epsilon, 0, \mathtt{Int}) + 2 * \mathtt{Int.size} + j * \mathtt{Char.size}, \mathtt{Char}, \mathtt{'g'})$$

This expression is rather precise since it mentions all the necessary readings that must be performed to compute the correct offset for the final writing.

These equations can be simply obtained from the general equations by partial evaluation. The full power of partial evaluation [Bjørner, Ershov & Jones 88] is not completely used since only systematic unfoldings need to be performed. The resulting expression can be compiled as it is, but usual techniques such as common subexpressions elimination are still useful.

The interest of partial evaluation compared to a static computation performed at compile-time is its generality. No matter how are defined the equations for $\mathcal{N}$, $\mathcal{S}$ or $\mathcal{D}$ (and we will see in a following section how to define new representation structurers), all computations that solely depend on data constants or types will be evaluated at compile-time and eliminated from run-time code. Compilers cannot offer an unrestricted structuration power without a complete calculus on these structurations. Functional programming and partial evaluation provide an universal basis for such a calculus and are simple, well studied and sufficient models for that.

3 Allocation

The next important operation to explain is allocation. Allocation of fixed sized instances is simple and well understood but our model also allows to define complex objects: for example, instances of $(\texttt{Char}^\star)^\star$. This type precisely corresponds to the "STR#" resource of the Macintosh[1] toolbox [1]. In order to allocate such varying objects, the allocator needs some informations grouped in a so-called *allocation parameter*.

To allocate a string of characters, $\texttt{Char}^\star$, requires a natural number which imposes the length of the string. To allocate an instance of $(\texttt{Char}^\star)^2$ requires two natural numbers which impose the lengths of the two strings. Therefore and as may be seen in figure 1, the allocation of a string of string of characters naturally requires a string of integers.

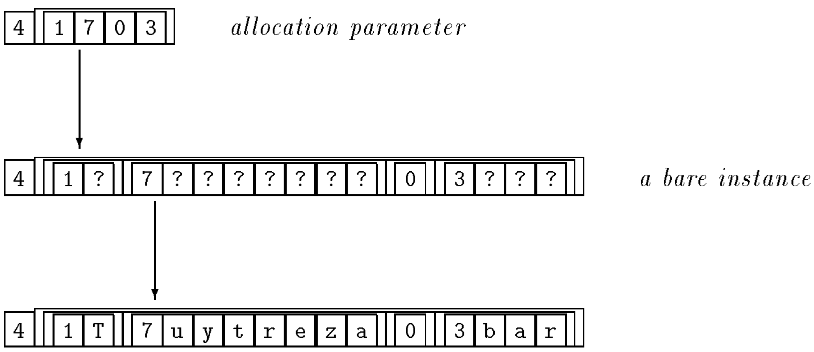

Figure 1: An allocation of a string of string of characters

We presented in [Queinnec & Cointe 88] the star calculus which derives from a type, the type of the associated allocation parameter. Although elegant, the star calculus involves too many types and makes inconvenient the allocation process and its description. We will therefore adopt the simpler convention that all allocation parameters are just sequence of numbers. If the language supports n-ary functions, allocation functions will take as many integer arguments as necessary; for other languages, sequences of numbers will be coded as a variable repetition of numbers: $\texttt{Int}^\star$.

The allocation process of highly structured data will be explained with only two functions.

- $\mathcal{A}$(llocate) — returns the size of the instance to be allocated according to the allocation parameter,
- $\mathcal{I}$(nitialize) — sets up the skeleton of the instance freshly allocated according to the allocation parameter.

[1] Macintosh is a trademark of Apple Computer Inc.

$\mathcal{A}$ takes the allocation parameter and computes the size of the instance to be allocated. This size is then submitted to the memory manager which returns an uninitialized area of memory compatible with the given size. This area has to be structured according to its type. For example, if one wants to allocate an instance of $(\texttt{Char}^\star)^\star$ and gives 4, 1, 7, 0 and 3 as allocation parameters then one will get the bare instance of figure 1. $\mathcal{A}$ returned the overall size: 16[2] whilst $\mathcal{I}$ wrote at offsets 0, 1, 3, 11 and 12 the different lengths of internal repetitions. The resulting bare instance is uninitialized from the user's point of view since all its readable fields do not contain anything sensible but its internal structure is correct.

The $\mathcal{A}$ and $\mathcal{I}$ functions are sequential processes which are defined with a functional style i.e. with continuations [Stoy 77]. $\mathcal{A}(\tau, n^\star, \kappa)$ takes a type τ, a sequence of natural numbers $n^\star$ which are the allocation parameters and a continuation κ. When $\mathcal{A}$ terminates its computation, it applies κ on the numbers which were not consumed during its computation. An example is given later. Continuations are λ-terms, looking like $\lambda(n^\star).body$, whose value is the computed size. Continuations make explicit the sequentiality of the various parts of the computation and can be easily adapted into formal semantics such as denotational semantics.

$\mathcal{I}(n^\star, \epsilon, \iota, \tau, \kappa)$ takes a sequence of natural numbers (the allocation parameters), the uninitialized instance ϵ, the current offset ι within the instance, the type τ of the current subpart (ϵ, ι) and a continuation κ. This continuation is a λ-term, looking like $\lambda(\epsilon, \iota, n^\star).body$, where ι is the offset beyond which the instance ϵ is not yet initialized and $n^\star$ is the sequence of allocation parameters not yet consumed. The value of the continuation is the instance itself properly initialized.

The equations for $\mathcal{A}$ are

$$\boxed{\begin{array}{ll}
\mathcal{A}(aBasicType, n^\star, \kappa) = aBasicType\texttt{.size} + \kappa(n^\star) & \\
\mathcal{A}(\tau_1 \times \tau_2 \times \ldots \times \tau_n, n^\star, \kappa) = \mathcal{A}(\tau_1, n^\star, \lambda(n'^\star).\mathcal{A}(\tau_2 \times \ldots \times \tau_n, n'^\star, \kappa)) & \\
\mathcal{A}(\tau^n, n^\star, \kappa) = \quad \textbf{if} \;\; n = 0 \;\textbf{then}\;\; \kappa(n^\star) & \\
\qquad\qquad\qquad\quad\;\; \textbf{else} \;\; \mathcal{A}(\tau, n^\star, \lambda(n'^\star).\mathcal{A}(\tau^{n-1}, n'^\star, \kappa) & \\
\mathcal{A}(\tau^\star, < j, n^\star >, \kappa) = \mathcal{A}(\tau^j, n^\star, \kappa) + \texttt{Int.size} & \\
\kappa_{init} = \lambda(n^\star).0 \qquad\qquad\qquad \text{/* \textit{ignore remaining allocation parameters} */} &
\end{array}}$$

The notation $< j, n^\star >$ represents a non empty sequence of natural numbers which head is j and which tail is $n^\star$. The initial continuation κ_{init} does not check if superfluous numbers are given. It is of course possible to devise another κ_{init} which checks that and otherwise raises an error.

Let show how is computed the size of our previous example. This computation is tedious but illustrates the use of continuations.

$$\begin{array}{l}
\mathcal{A}((\texttt{Char}^\star)^\star, < 4, 1, 7, 0, 3 >, \kappa_{init}) \\
= \texttt{Int.size} + \mathcal{A}((\texttt{Char}^\star)^4, < 1, 7, 0, 3 >, \kappa_{init}) \\
= \texttt{Int.size} + \mathcal{A}(\texttt{Char}^\star, < 1, 7, 0, 3 >, \lambda(n^\star).\mathcal{A}((\texttt{Char}^\star)^3, n^\star, \kappa_{init})) \\
= 2 * \texttt{Int.size} + \mathcal{A}(\texttt{Char}, < 7, 0, 3 >, \lambda(n^\star).\mathcal{A}((\texttt{Char}^\star)^3, n^\star, \kappa_{init})) \\
= 2 * \texttt{Int.size} + \texttt{Char.size} + \mathcal{A}((\texttt{Char}^\star)^3, < 7, 0, 3 >, \kappa_{init}) \\
= 3 * \texttt{Int.size} + \texttt{Char.size} + \mathcal{A}(\texttt{Char}^7, < 0, 3 >, \lambda(n^\star).\mathcal{A}((\texttt{Char}^\star)^2, n^\star, \kappa_{init})) \\
= 3 * \texttt{Int.size} + 8 * \texttt{Char.size} + \mathcal{A}((\texttt{Char}^\star)^2, < 0, 3 >, \kappa_{init}) \\
= 3 * \texttt{Int.size} + 8 * \texttt{Char.size} + \mathcal{A}(\texttt{Char}^\star, < 0, 3 >, \lambda(n^\star).\mathcal{A}(\texttt{Char}^\star, n^\star, \kappa_{init})) \\
= 4 * \texttt{Int.size} + 8 * \texttt{Char.size} + \mathcal{A}(\texttt{Char}^0, < 3 >, \lambda(n^\star).\mathcal{A}(\texttt{Char}^\star, n^\star, \kappa_{init})) \\
= 4 * \texttt{Int.size} + 8 * \texttt{Char.size} + \mathcal{A}(\texttt{Char}^\star, < 3 >, \kappa_{init})) \\
= 5 * \texttt{Int.size} + 8 * \texttt{Char.size} + \mathcal{A}(\texttt{Char}^3, <>, \kappa_{init})) \\
= 5 * \texttt{Int.size} + 11 * \texttt{Char.size} + \kappa_{init}(<>) \\
= 5 * \texttt{Int.size} + 11 * \texttt{Char.size}
\end{array}$$

If `Int.size` and `Char.size` are both equal to 1 then the result is 16. The allocation parameters are one by one consumed by each $\tau^\star$ repetition allocation, the remaining parameters are given to the continuation which needs them to continue to compute the size of the rest of the structuration.

[2] We consider here that $\texttt{Int.size} = \texttt{Char.size} = 1$.

The equations for $\mathcal{I}$ are

$$\boxed{\begin{array}{l}
\mathcal{I}(n^\star, \epsilon, \iota, aBasicType, \kappa) = \kappa(\epsilon, \iota + aBasicType.\mathtt{size}, n^\star) \\
\mathcal{I}(n^\star, \epsilon, \iota, \tau_1 \times \tau_2 \times \ldots \times \tau_n, \kappa) = \mathcal{I}(n^\star, \epsilon, \iota, \tau_1, \lambda(\epsilon, \iota', n'^\star).\mathcal{I}(n'^\star, \epsilon, \iota', \tau_2 \times \ldots \times \tau_n)) \\
\mathcal{I}(n^\star, \epsilon, \iota, \tau^n, \kappa) = \mathcal{I}(n^\star, \epsilon, \iota, \tau, \lambda(\epsilon, \iota', n'^\star).\mathcal{I}(n'^\star, \epsilon, \iota', \tau^{n-1})) \\
\mathcal{I}(< j, n^\star >, \epsilon, \iota, \tau^\star, \kappa) = \ \textbf{let} \ \ \alpha = \mathcal{W}(\epsilon, \iota, \mathtt{Int}, j) \ \textbf{in} \ \ \mathcal{I}(n^\star, \epsilon, \iota + \mathtt{Int.size}, \tau^j, \kappa) \\
\kappa_{init} = \lambda(\epsilon, \iota, n^\star).\epsilon
\end{array}}$$

The `let` form appearing in the equation defining the initialization of a $\tau^\star$ instance expresses that the value of $\mathcal{W}$ is useless. The simplest is probably to think of the whole `let` expression as a sequence of instructions: write first the length j at current offset then initialize the rest of the instance.

These equations conclude the description of all instances which have a structuration arbitrarily composed of concatenation and fixed or variable repetition.

4 Types and Objects

As may be seen in the previous equations, only three properties characterize basic types:

their size: $aBasicType$`.size`
how to read them: $\mathcal{R}(\epsilon, \iota, aBasicType)$
how to write them: $\mathcal{W}(\epsilon, \iota, aBasicType, anInstanceOfBasicType)$

On the other hand composed types belong to three possible sets depending on how they are defined i.e. by concatenation, fixed repetition or variable repetition. They all have specialized behaviours under $\mathcal{N}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{A}$ and $\mathcal{I}$ functions.

A class taxonomy as illustrated by Object Oriented Languages can well explain these facts.

- A type is an object and belongs to a class.
- The `BasicType` class imposes one slot to hold a size along with a read and write methods.
- The `CartesianProduct` class (the set of types $\tau_1 \times \ldots \times \tau_n$) offers $\mathcal{N}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{A}$ and $\mathcal{I}$ methods and as many slots as there are components in the cartesian product: τ_1, τ_2 ...and τ_n.
- The `CartesianPower` class (the set of types τ^n) offers $\mathcal{N}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{A}$ and $\mathcal{I}$ methods and two slots: one for the power: n and one for the repeated type: τ.
- The `KleeneStar` class (the set of types $\tau^\star$) offers $\mathcal{N}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{A}$ and $\mathcal{I}$ methods and one slot for the repeated type τ.

Type structurers are supported by the `ComposedType` virtual class which imposes the virtual methods $\mathcal{N}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{A}$ and $\mathcal{I}$. Any subclass of `ComposedType` must implement these methods and therefore is a legal representation structurer. Figure 2 graphically shows the inheritance and instanciation graph of these classes.

Since types are now objects, they themselves have a type. It appears in [Queinnec & Cointe 88] that a reflective view is possible and avoids infinite regression since `CartesianProduct` class is the ultimate metaclass.

Suppose we want to define the `Pair` type as in Lisp. A pair is an object which has two pointers referring to its left and right sons. The `Pair` type can be defined as $\mathtt{Ref}^2$ i.e. `Pair` is an instance of `CartesianPower` with a *repeatedType* slot referring to the basic type `Ref`, and a *repetitionFactor* slot containing the number 2. Reading or writing the left or the right son of a pair may be inferred from the general properties of the types belonging to the `CartesianPower` family. The `car` and `rplacd` functions of Lisp which, respectively, returns the left son of a pair or updates the rigth son of a pair, are then

$\mathtt{car} = \lambda(\epsilon).\mathcal{R}(\epsilon, 0, \mathtt{Ref})$
$\mathtt{rplacd} = \lambda(\epsilon, \epsilon').\mathcal{W}(\epsilon, \mathtt{Ref.size}, \mathtt{Ref}, \epsilon')$

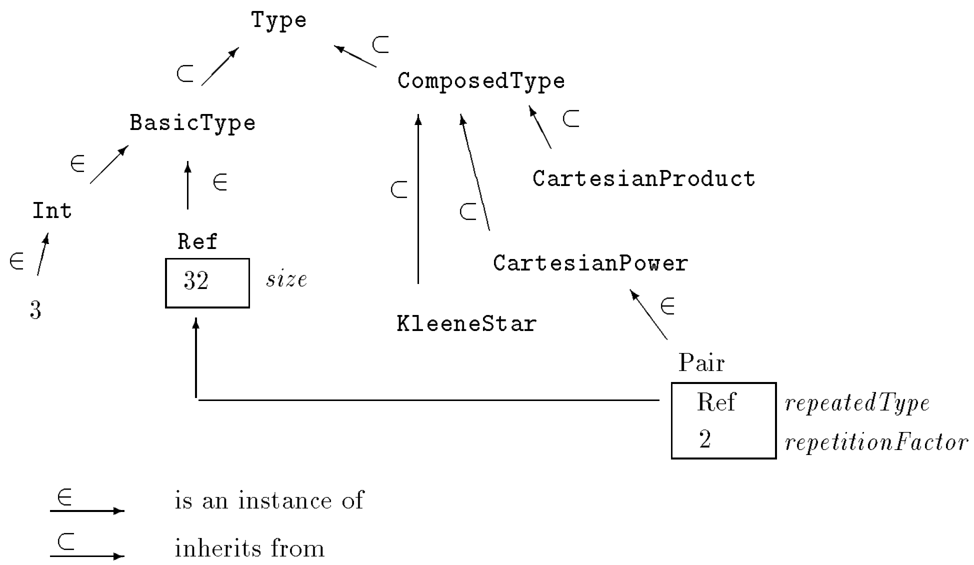

Figure 2: Types Net

It is worthwile to note that the behaviour of a pair is determined by its type or to adopt the terminology of Object Oriented Languages, by its class. But its representation is only determined by its metaclass (i.e. the class of its class). This is normal since the message "`new`", which returns a fresh instance of a class, is sent to this class and is answered by a method held by the metaclass. In brief, classes support behaviours whilst metaclasses support representation.

5 Variants

Structurations are defined via the five methods they must provide. Let us give another illustration namely: *variants*. A variant record as in Pascal may appear under two different forms: tagged or untagged variants. In the tagged version, it is possible to know from the instance in which mode it is. This distinction cannot be made in the untagged case which appears to be a kind of EQUIVALENCE as in Fortran or an `union` structure as in the C language [Kernighan and Ritchie 78]. The untagged version is naturally provided by our model when interpreting an instance with a type different from its natural type. Tagged variant prefixes a type information in front of the contents of its fields. Instead of prefixing the type, it is also possible to only prefix its index in the list of possible types for the variant. It is the solution we define hereafter.

We will note $\tau_1 \vee \tau_2 \vee \ldots \vee \tau_n$ a variant type among $\tau_1, \tau_2 \ldots\ \tau_n$. The equations are

$$
\boxed{
\begin{array}{ll}
\mathcal{S}(\epsilon, \iota, \tau_1 \vee \tau_2 \vee \ldots \vee \tau_n) = & \mathtt{Int.size} + \mathcal{S}(\epsilon, \iota + \mathtt{Int.size}, \tau_l) \\
 & \quad \textbf{where} \quad l = \mathcal{R}(\epsilon, \iota, \mathtt{Int}) \\
\mathcal{N}(\epsilon, \iota, \tau_1 \vee \tau_2 \vee \ldots \vee \tau_n) = & \mathcal{N}(\epsilon, \iota + \mathtt{Int.size}, \tau_l) \\
 & \quad \textbf{where} \quad l = \mathcal{R}(\epsilon, \iota, \mathtt{Int}) \\
\mathcal{D}(\epsilon, \iota, \tau_1 \vee \tau_2 \vee \ldots \vee \tau_n, j) = & \mathcal{D}(\epsilon, \iota + \mathtt{Int.size}, \tau_l, j) \\
 & \quad \textbf{where} \quad l = \mathcal{R}(\epsilon, \iota, \mathtt{Int}) \\
\mathcal{A}(< l, n^\star >, \tau_1 \vee \tau_2 \vee \ldots \vee \tau_n, \kappa) = & \textbf{if} \quad 0 \leq l < n-1 \\
 & \textbf{then} \quad \mathcal{A}(n^\star, \tau_l, \mathtt{Int.size} + \kappa(n^\star)) \\
 & \textbf{else} \quad \bot \\
\mathcal{I}(\epsilon, \iota, < l, n^\star >, \tau_1 \vee \tau_2 \vee \ldots \vee \tau_n, \kappa) = & \textbf{let} \quad \alpha = \mathcal{W}(\epsilon, \iota, \mathtt{Int}, l) \\
 & \textbf{in} \quad \mathcal{I}(\epsilon, \iota + \mathtt{Int.size}, \tau_l, n^\star)
\end{array}
}
$$

A careful reader may have recognized that in the encoding of the variant, we did not impose all alternatives of the variant to have a same length: the variant is packed. The used variant is coded by a natural number. Smaller number representations may be used if variants are restricted to have less alternatives than may be coded in an integer.

With variants, concatenation and variable repetition, are offered the equivalent of the programmatic `if-then-else`, `begin-end` and `while-do` features.

Let us give another example of a new representation structurer: *à la C variable repetition*. Like for other representation structurers, we will only give the five methods which completely define it. The equations are:

$$
\begin{array}{l}
\mathcal{S}(\epsilon,\iota,\tau^\star) = \ \textbf{if}\ \ \mathcal{R}(\epsilon,\iota,\textit{type-of}(\textit{end-mark})) = \textit{end-mark}\\
\qquad\qquad\quad \textbf{then}\ \ \textit{type-of}(\textit{end-mark})\texttt{.size}\\
\qquad\qquad\quad \textbf{else}\ \ \mathcal{S}(\epsilon,\iota+\mathcal{S}(\epsilon,\iota,\tau),\tau^\star)+\mathcal{S}(\epsilon,\iota,\tau)\\
\mathcal{N}(\epsilon,\iota,\tau^\star) = \ \textbf{if}\ \ \mathcal{R}(\epsilon,\iota,\textit{type-of}(\textit{end-mark})) = \textit{end-mark}\\
\qquad\qquad\quad \textbf{then}\ \ 0\\
\qquad\qquad\quad \textbf{else}\ \ \mathcal{N}(\epsilon,\iota+\mathcal{S}(\epsilon,\iota,\tau),\tau^\star)\\
\mathcal{D}(\epsilon,\iota,\tau^\star,j) = \ \textbf{if}\ \ j=0\ \textbf{then}\ \ \iota\\
\qquad\qquad\quad \textbf{elseif}\ \ \mathcal{R}(\epsilon,\iota,\textit{type-of}(\textit{end-mark})) = \textit{end-mark}\\
\qquad\qquad\quad \textbf{then}\ \ \perp\\
\qquad\qquad\quad \textbf{else}\ \ \mathcal{D}(\epsilon,\iota+\mathcal{S}(\epsilon,\iota,\tau),\tau^\star,j-1)\\
\mathcal{A}(<l,n^\star>,\tau^\star) = \ \texttt{Char.size}+\mathcal{A}(n^\star,\tau^l)\\
\mathcal{I}(\epsilon,\iota,\tau^\star,<l,n^\star>,\kappa) = \mathcal{I}(\epsilon,\iota,\tau^l,\lambda(\epsilon,\iota',n'^\star).\ \ \textbf{let}\ \ \alpha=\mathcal{W}(\epsilon,\iota',\textit{type-of(end-mark)},l)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \kappa(n'^\star))\\
\\
\qquad\qquad\qquad\qquad \textit{end-mark} = \texttt{'\textbackslash 0'}\\
\qquad\qquad\qquad\qquad \textit{type-of}(\textit{end-mark}) = \texttt{Char}
\end{array}
$$

Like C strings with embedded null characters, à la C repetitions offer the same kind of ambiguity with respect to the $\mathcal{N}$, $\mathcal{S}$ and $\mathcal{D}$ functions. This fact can be alleviated if the mark ending the repetition is lengthened and complexified. À la C repetition is therefore a whole family of representation structurers parameterized by the choosen end-mark. Whatever this end-mark may be, the above equations remain and fully describe how to handle à la C repetitions. Figure 3 shows the place of these new classes.

From the object point of view, à la C repetition is a metaclass which instances are representation structurers with one slot holding the the end-mark.

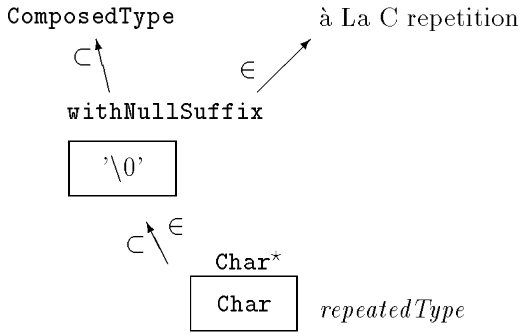

Figure 3: à la C repetitions

6 Applications

Some applications are described hereafter which show the interest to have explicit data structurations.

6.1 Transmission

The XDR protocol [XDR] showed that having explicit representations of types simplifies their exchange through various media. When a type is described with the XDR type specification language, an instance of that type can be converted into a sequence of four byte words. That sequence may then be exchanged and can be rebuilt at the other end thanks to the type description. XDR is tightly coupled with the C language but ignores pointers. Our model is independent from any language and allows to describe a richer world than XDR. Moreover since types are objects thus instances, types may be exchanged as normal instances. The receptor needs only to know the standard structurations to decode these types and therefore is able to exchange any instance of them. This capability is also useful for systems with persistent objects where their structuration must be remembered.

6.2 Data Inspection

An inspector is a tool which allows to navigate through data, following pointers, showing fields and overall structure of networked data. Given an instance, an inspector looks for an appropriate method to display that instance.

An inspector can be easily defined in our model with as many methods as there are basic types (in order to know how to display them) plus one method on the `ComposedType` class. This lattest method just says that inspection of a composed part of an instance is the inspection of all its subparts. The number of components, their type (and therefore their inspection method), their location within the instance are results obtained from the standard functions $\mathcal{N}$, $\mathcal{S}$ and $\mathcal{D}$.

Of course it is possible to define better inspection methods for some structurations or for some precise types, but this is by no means necessary to allow full inspection of any instances.

Besides inspection of small instances, one may also think of inspecting the whole memory of a computer. The whole memory may be assigned a type which is the concatenation of, say, the interrupt vector, segments of system codes and data, virtual memory table, process segments, bitmaps ... The whole type may be difficult to elaborate but, once established, the whole memory can be inspected: a great improvement for the analysis of post-mortem dumps.

6.3 Garbage Collector Synthesis

More and more languages offer a garbage collector (GC) for the management of their heap. As experimented for many years in Lisp, the design and the implementation of a performant GC is a huge task which requires as much efforts as a compiler writing [Cohen 81, Lang & Dupont 87].

All GCs have to mark and to sweep or copy useful data. Suppose we are in the context of a mark and sweep GC. When marking an useful data, all the instances it refers must be also marked. When marking all useful data is achieved, the heap is then linearly swept. Every instance is considered and recycled if useless. This M(ark) and S(weep) GC is defined by the following equations. It uses some functions left unspecified (and depending on the implementation) to physically mark, unmark and test the mark of an instance. It also uses some pointer arithmetics. Approriate castings are made between instances and addresses.

$$\begin{array}{l}
\mathsf{M}(\epsilon,\iota,\mathtt{Ref}) = \mathsf{M}(\mathcal{R}(\epsilon,\iota,\mathtt{Ref}), 0, \mathit{type\text{-}of}(\mathcal{R}(\epsilon,\iota,\mathtt{Ref}))) \\
\mathsf{M}(\epsilon,\iota,\mathit{otherBasicType}) = \mathit{nothing} \\
\mathsf{M}(\epsilon,\iota,\tau_1\times\tau_2\times\ldots\times\tau_n) = \mathsf{M}(\epsilon,\iota,\tau_1)\,;\mathsf{M}(\epsilon,\mathcal{D}(\epsilon,\iota,\tau_1\times\tau_2\times\ldots\times\tau_n,1),\tau_2\times\ldots\times\tau_n) \\
\mathsf{M}(\epsilon,\iota,\tau^n) = \mathsf{M}(\epsilon,\iota,\tau)\,;\mathsf{M}(\epsilon,\mathcal{D}(\epsilon,\iota,\tau^n,1),\tau^{n-1}) \\
\mathsf{M}(\epsilon,\iota,\tau^\star) = \mathsf{M}(\epsilon,\iota+\mathtt{Int.size},\tau^{\mathcal{N}(\epsilon,\iota,\tau^\star)}) \\
\\
\mathsf{S}(\epsilon,\tau) = \quad \textbf{if}\ \ \mathit{is\text{-}marked?}(\epsilon,\tau)\ \textbf{then}\ \ \mathit{unmark}(\epsilon,\tau)\ \textbf{else}\ \ \mathit{recycle}(\epsilon,\tau)\,; \\
\qquad\qquad\quad \textbf{let}\ \ \alpha = \mathit{instance\text{-}at}(\mathit{address\text{-}of}(\epsilon)+\mathcal{S}(\epsilon,0,\tau)) \\
\qquad\qquad\quad \textbf{in}\ \ \mathsf{S}(\alpha,\mathit{type\text{-}of}(\alpha))
\end{array}$$

The semi-colon which appears in several equations expresses the sequentiality. The M method recursively marks instances. M is dispatched on every composed subparts of an instance and is especially handled on a reference; it does nothing on other terminal subparts of instances such as integers or characters. The S method handles the instance and then skips it in order to inspect the next instance lying in the heap. To skip an instance requires to know its size which is just what computes $\mathcal{S}$.

From these equations can be synthesized the appropriate (and efficient) mark and sweep methods for a whole system. These technique was also used and experimented in [Spir 88] for Le-Lisp[3] [Chailloux 89].

6.4 Inheritance by Similitude

Inheritance in Object Oriented Languages may take two forms: slot inheritance or method inheritance. The latter ensures that a method on a class is accessible from all its subclasses. The former that slots defined on a class also are implicit slots of its subclasses. The usual convention is that the representation of an instance of a subclass is the concatenation of the representation of an instance of the original class followed by the additional slot representations defined at the subclass level. This representation allows offsets of common fixed sized slots to be the same between instances of the class and of its subclasses. Therefore methods on the class do not need to be recompiled when defining subclasses.

If cartesian product is the means to concatenate additional slots then cartesian product may, more generally, be seen as a subclass constructor. Therefore

$$\forall\tau_1,\tau_2,\ldots\tau_n, \qquad \tau_1\times\tau_2\times\ldots\times\tau_n \quad \text{is a subclass of} \quad \tau_1.$$

Even if it cannot be ensured in our model that offsets for accessing subcomponents of τ_1 are fixed, they are similarly computed for τ_1 and $\tau_1\times\tau_2\times\ldots\times\tau_n$. Thus methods may be shared between these two types and do not need to be recompiled. This property does not depend on the complexity of τ_1. For example, $(\mathtt{Char}^\star)\times\mathtt{FontIndex}$ is a subclass of $(\mathtt{Char}^\star)$ and any method dealing with a string of characters can similarly act on instances of $(\mathtt{Char}^\star)\times\mathtt{FontIndex}$.

This property extends the usual convention of Object Oriented Languages where instances are identified to records which fields must have a fixed size. But one may also think that if τ_1 is a subclass of τ_2 then $\tau_1^\star$ may be a subclass of $\tau_2^\star$. All methods dealing with instances of $\tau_2^\star$ have to cope with the varying length of them, they can therefore handle instances of $\tau_1^\star$ since these are also variable repetitions. This property may be expensive to compute since $\tau_1^\star$ might not offer a fast indexed access to its components although $\tau_2^\star$ may do; one has thus better recompile the method on $\tau_1^\star$ rather than sharing it with $\tau_2^\star$ which would be inefficient on $\tau_2^\star$.

One can also think that a type might have several different representations concurrently used. For instance `string` can be a *logical type* which instances may be à la Pascal or à la C repetitions i.e. may have two different *physical types*. All these new features are currently under experimentation.

7 Conclusions

We presented the general equations specifying access and allocation of arbitrary data aggregates made of basic types and structured by concatenation or fixed or variable repetition. These structurations may be

[3] Le-Lisp is a trademark of INRIA.

composed as deeply as wished. Five methods with simple semantics were sufficient to define them. These equations were explained in the context of generic functions indicating what features basic types (`Int`, `Ref` ...) and type structurers (concatenation, repetition ...) must have. It is straightforward to add new basic types or new type structurers if they provide the required features.

To have such explicited types allows to perform some computations on them:

- exchange of types and of their instances,
- remembrance of types in persistent object systems,
- post-mortem dump analysis or live inspection of data,
- synthesis of garbage collector.

This uniform view of types eases the construction of compilers or run-time libraries. This work eventually offers a systematic tool, based on functional calculus and partial evaluation, to describe data aggregates and to synthesize all the low level code that manage them.

References

[1] Apple Computer Inc. *Inside Macintosh.*

[Bjørner, Ershov & Jones 88] Dines Bjørner, Andrei P. Ershov, Neil D. Jones (eds.), *Partial Evaluation and Mixed Computation*, GL. Avernæs, Denmark, North-Holland, 1988.

[Boehm 86] Hans-Juergen Boehm, Alan Demers, *Implementing Russell*, Proceedings of the SIGPLAN'86 Symposium on Compiler construction, Palo Alto (CA), June 25-27, 1986, SIGPLAN Notices Vol 21, # 7, July 1986.

[Briot 87] Jean-Pierre Briot, Pierre Cointe, *A Uniform Model for Object-Oriented Languages using the Class Abstraction*, IJCAI'87, pp 40–43, 1987.

[Cardelli 85] Luca Cardelli, Peter Wegner, *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol 17, # 4, December 1985.

[Chailloux 89] Jérôme Chailloux, Le-Lisp– *Manuel de Référence*, INRIA.

[Cohen 81] Jacques Cohen, *Garbage Collection of Linked Data Structures*, Computing Surveys, Volume 13, Number 3, September 1981, pp 341 – 367.

[Goldberg 83] Adele Goldberg and David Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, 1983.

[Kernighan and Ritchie 78] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice-Hall Software Series, 1978.

[Lang & Dupont 87] Bernard Lang, Francis Dupont, *Incremental Incrementally Compacting Garbage Collection*, SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, Saint Paul, MA, pp 253 – 263.

[Queinnec & Cointe 88] Christian Queinnec, Pierre Cointe, *An open-ended Data Representation Model for* EuLisp, 1988 ACM Conference on Lisp and Functional Programming, pp 298–308, Snowbird, Utah.

[Spir 88] Eric Spir, *Étude d'un glaneur de cellules adaptatif et configurable*, Thèse d'Université Paris VI, 1988.

[Steele 84] Guy L. Steele Jr., *Common Lisp, the Language*, Digital Press, Burlington MA, 1984.

[Stoy 77] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

[XDR] *eXternal Data Representation Standard: Protocol Specification*, RFC 1014, ARPA Network Information Center.