Type Models: Toward Unifying Concepts for Language Interoperability*

Arturo J. Sánchez Ruíz
Centro de Ingeniería de Software y Sistemas (ISYS)
Facultad de Ciencias. Universidad Central de Venezuela.
Apartado 47642, Caracas 1041–A, VENEZUELA.

E–mail: *arsanche@conicit.ve*, *asanchez@anubis.ciens.ucv.ve*

Abstract

The term language interoperability refers to the ability of one language to use software resources (perhaps previously) written in another language. It is therefore important to deal with the issues introduced by the fact that the languages in question have, in general, type systems of different nature. This problem arises in applications such as distributed object interoperability, multi-language/multi-paradigm programming and the re-engineering of legacy code for ulterior reuse. This paper presents the concept of *type models* with the goal of providing a common platform on which apparently different type systems can coexist in harmony. We also present a taxonomy of type models, and examples of type models associated with various programming languages which implement different paradigms.

1 Introduction

The role of a type model is to provide an abstract platform to give meaning to the type construction process in a programming language. Let us consider a particular programming language L. The *type system* of L is the set of syntactic rules for defining types, whereas its *type model* assigns meaning to syntactically valid constructions. The type model should therefore use a battery of well understood mathematical concepts. As we shall see, the concepts we will use are: algebras, functions, sets, etc. (cf. Fig. 1).

Since language interoperability implies dealing with heterogeneous type systems, we must concentrate on concepts instead of becoming distracted by details associated with syntax and/or the run–time representation of objects. For example, when we say "multi–dimensional array" we immediately visualize some sort of array of slots which can hold objects (cf. Fig. 2). This picture is independent of the language in question. There are, however, differences between multi–dimensional arrays in actual languages with respect both to how they are defined/declared and how they are stored at run–time. With the aid of type models we want to formalize (among other things) the idea behind the phrase: "arrays are essentially the same in any programming language." The essence is precisely the concept behind arrays. So, using type models the phrase above can be generalized to: "the type construction process is essentially the same for a wide family of programming languages."

*This work was partially supported by *Consejo de Desarrollo Científico y Humanístico de la Universidad Central de Venezuela*. The author wishes to thank Dr. Ephraim Glinert for his insight and feedback while this work was in progress.

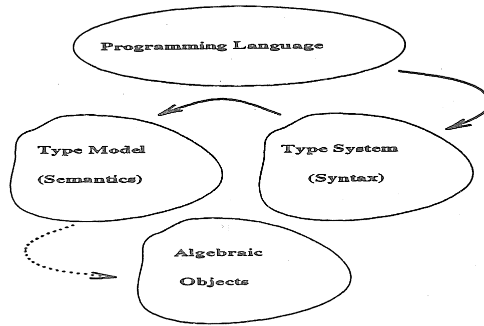

Figure 1: The type system and type model of a programming language.

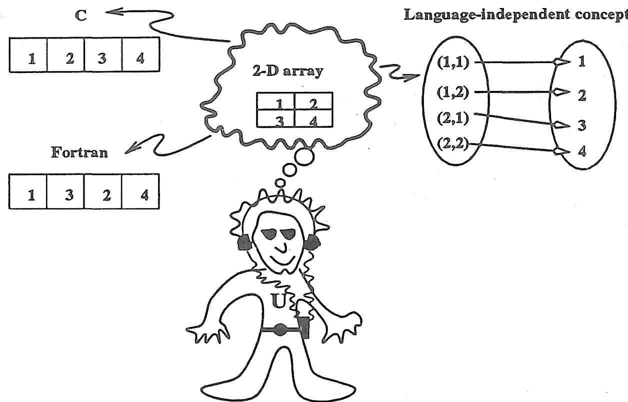

Figure 2: Using abstraction to define a 2-D array.

The concept of type models perceives the type construction process in an inductive way. There are atomic types (which we call *primitive domains*) and operators (which we call *type builders*) used to build new types from existing ones. When building new types, we usually need labels to name things with. So, a type model can be seen as a type–building machine, as illustrated in Fig. 3.

The structure of this paper is as follows. In Section 2 we formally define type models and give a taxonomy which shows seemingly different languages from a common perspective. In Section 3 we show examples of type models associated with the programming languages C, C++ and Eiffel. In Section 4 previous work in the field is reviewed. Finally, Section 5 presents a summary and suggests directions for future work.

2 Type Models

A *type model* is a triple $\mathcal{M} = (\mathcal{P}, \mathcal{B}, \mathcal{L})$, where $\mathcal{P}$ is a family of *primitive domains*, $\mathcal{B}$ is a family of *type builders* and $\mathcal{L}$ is an *alphabet*. As mentioned previously, a type model can be seen as a type–

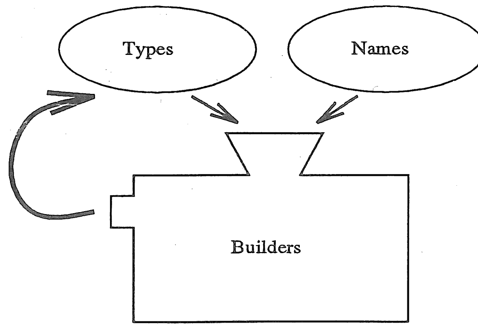

Figure 3: The type model as a type–building machine.

building machine (cf. Fig. 3). In light of this formal definition, to build a type under $\mathcal{M}$ we take an element B of $\mathcal{B}$, existing types $T_1, \cdots, T_k$, and a string w formed with elements of $\mathcal{L}$ and then apply $B(T_1, \cdots, T_k, w)$.[†] So, no matter how complicated the structure of an object of a given type is, it will always be formed by some aggregation of objects taken from domains in $\mathcal{P}$. Each particular programming language will have its own set of primitive domains, type builders and labels, but all of them will build their types in the same way.

The elements of $\mathcal{P}$ are not mere sets but rather *heterogeneous type algebras*[‡] [5] D_i such that we have $D_i = (V_{D_i}, O_{D_i})$, where V_{D_i} is a set of atomic values and $O_{D_i} = \{o^j_{D_i} : V^{m_j}_{D_i} \to V'_{D_i}\}$ is a set of operations acting on V_{D_i} where $m_j \geq 1$ denotes the m_j–fold cartesian product and $V'_{D_i} \in \mathcal{P}$.

To define the elements of $\mathcal{B}$ we need to first define the "pool" from which types are taken to form new types (which in turn go into this pool). Informally, we denote by $\mathcal{T}(\mathcal{M})$ the collection of all types generated by $\mathcal{M}$. We would expect the members of $\mathcal{P}$ to belong to $\mathcal{T}(\mathcal{M})$ and, since they are algebras, we would expect all members of $\mathcal{T}(\mathcal{M})$ to be algebras as well. So, for now, we will conceive of $\mathcal{T}(\mathcal{M})$ as the set of all algebras generated by $\mathcal{M}$ via its builders.

We need some notation before we can define the members of $\mathcal{B}$. Let X be a set. Then X^* is the set of all sequences of elements of X whose length is greater than or equal to zero. If $w \in X^*$ then $|w|$ denotes its length and $X^+ = \{w \in X^* \mid |w| \geq 1\}$. If u, v are two members of X^* then $u \cdot v$ denotes the concatenation of u and v. The *empty sequence* ϵ whose length is zero belongs to X^*.

Now, let B be a member of $\mathcal{B}$. Then

$$B : \mathcal{T}(\mathcal{M})^+ \times \mathcal{L}^* \to \mathcal{T}(\mathcal{M}).$$

Informally, a builder takes one or more types and some labels, and produces another type. More precisely, a builder takes at least one algebra, and one string, and produces another algebra. The *domain dom*(B) of a builder B is the subset of $\mathcal{T}(\mathcal{M})^+ \times \mathcal{L}^*$ for which B is defined.

[†]Assuming that $(T_1, \cdots, T_k, w)$ is in the domain of B.

[‡]From now on, we will just use the term algebra.

We shall give examples below of type models associated with well–known programming languages. However, we find it useful to give a simple example at this point. Let us return to the type builder *array* for multi–dimensional arrays discussed at the beginning of this section. This builder takes a type T_0 (the slot type) and labels which identify each dimension, to produce an array whose values are defined to be total functions (i.e., totally defined functions) which associate some index sets with values of T_0. We underline the word values to stress the point that both T_0 and what the builder *array* produces are algebras. Thus, to completely define the effect of the builder *array* we should also indicate the set of operations associated with the algebra. So, let $\mathcal{L}$ be the alphabet from which the labels are created and let us denote by $\langle n_1, \cdots, n_d \rangle$ a string of symbols taken from $\mathcal{L}$. Then:

$$array\ (T_0, \langle n_1, \cdots, n_d \rangle) = \{f \mid f : I = I_1 \times \cdots \times I_d \rightarrow T_0 \ \&\ f\ is\ total\}$$

where $I_j = \{1, \cdots, n_j\}$, $1 \leq j \leq d$, $1 \leq n_j \leq k$, and k is the maximun dimension allowed for these arrays. In other words, we model array values as total functions which associate tuples $(i_1, \cdots, i_d) \in I_1 \times \ldots \times I_d$ with values of T_0. Of course, not all sequences of types and strings define an array under the builder *array*, therefore we say that a pair[§] $(T, w) \in \mathcal{T}(\mathcal{M})^+ \times \mathcal{L}^*$ belongs to $dom(array)$ if and only if $|T| = 1$ and $1 \leq |w| \leq D$ where D is the maximun number of dimensions allowed. This algebra comes equipped with two operations:

- Function evaluation: given a function f and a tuple of indices $(i_1, \cdots, i_d) \in I$, then $f(i_1, \cdots, i_d)$ is the evaluation of f at $(i_1, \cdots, i_d)$.
- Function modification: given f, $(i_1, \cdots, i_d) \in I$ and $v \in T_0$ then the expression $f(i_1, \cdots, i_d) = v$ denotes a function $\bar{f}$ such that:

$$\bar{f}(x) = \begin{cases} f(x) & \text{if } x \neq (i_1, \cdots, i_d) \\ v & \text{otherwise} \end{cases}$$

We close this section by mentioning that we can also consider in $\mathcal{T}(\mathcal{M})$ the relation "*is defined in terms of*" which would give $\mathcal{T}(\mathcal{M})$ a richer structure known as a *category* [8]. This may not have a direct practical implication but helps to formally characterize $\mathcal{T}(\mathcal{M})$ mathematically. Details can be found in [14].

2.1 A Taxonomy of Type Models

The formalism discussed above indicates that we can get different classes of type models by varying $\mathcal{P}$ and $\mathcal{B}$. Let us concentrate on the case where $\mathcal{P}$ is fixed, which leads us to three alternatives:

- $\mathcal{B}$ is empty. The only types we have at hand are those in $\mathcal{P}$.
- $\mathcal{B}$ is closed. The type system allows the construction of new types, but to construct them we are limited to the power afforded by the type builders in $\mathcal{B}$. This class includes:

[§] From now on, a pair (T, w) will denote a member of $\mathcal{T}(\mathcal{M})^+ \times \mathcal{L}^*$ unless we say otherwise.

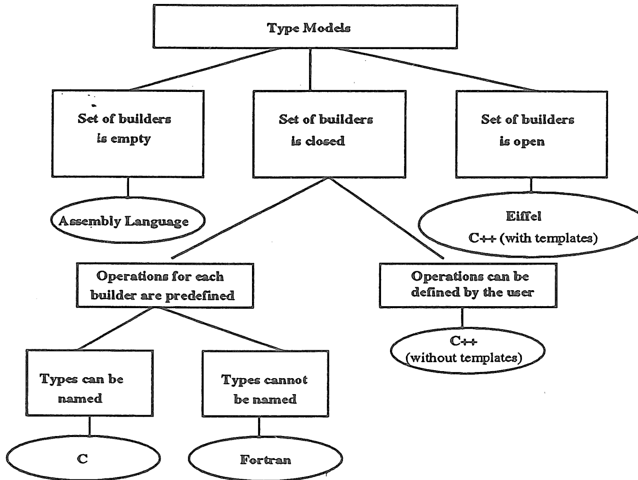

Figure 4: A taxonomy of type models.

- Standard operations. Each type builder comes equipped with a set of standard operations to manipulate objects of the given type. These are a minimal set of operations which can be used to build more complicated ones by mimicking the way structured types are created. This class has two subclasses:
  * Types can be named: it is possible to associate a type with a name which can be recalled afterwards, so recursive definitions are possible.
  * All types are anonymous: we can't name type definitions.
- Non-standard operations: the operations associated with a type builder can be defined by the programmer.

• $\mathcal{B}$ is open. The type system allows the creation of meta type builders, i.e., builders which produce builders (e.g., generic types).

This taxonomy, along with examples of languages having the corresponding type model, is depicted in Fig. 4.

3 Examples of Type Models

In this section our goal is to show how the type building process in seemingly different programming languages can be viewed from a common perspective afforded by the concept of type models. Therefore, we do not attempt by any means to give a complete and exhaustive semantics for all possible type builders associated with the languages in question.

3.1 C

Consider the type model $\mathcal{M}_C = (\mathcal{P}_C, \mathcal{B}_C, \mathcal{L}_C)$, where:

- $\mathcal{P}_C = \{\texttt{int}, \texttt{float}, \texttt{char}\}$
- $\mathcal{B}_C = \{\texttt{arr}, \texttt{fun}, \texttt{struct}, \texttt{union}, \star\}$
- $\mathcal{L}_C = \texttt{Num} \uplus (\texttt{Let} \cup \texttt{Char})^* \uplus \texttt{Addr}$ where $\texttt{Num} = \{1, \cdots, k\}$ with k defined as in the previous example, $\texttt{Let} = \{a, \cdots, z, A, \cdots, Z\}$, $\texttt{Char} = \{\textit{special characters}\}$, Addr is an interval of natural numbers used to model run-time addresses, and the operator $\uplus$ is defined so that $x \in X \uplus Y$ if and only if $x \in X$ or $x \in Y$ but not both.

For the sake of brevity, we only show the definition of builder struct. Let us then consider this builder and its domain, which are defined as:

$$\texttt{struct}(\langle T_1, \cdots, T_k \rangle, \langle l_1, \cdots, l_k \rangle) = \prod_{i=1}^{k}(\{l_i\} \times T_i)$$
$$\mathit{dom}(\texttt{struct}) = \{(T, w) \mid 1 \leq |T| = |w| \leq F, w \in (\texttt{Let} \cup \texttt{Char})^{**}\}$$

where $\prod_{i=1}^{k} A_i = A_1 \times \cdots \times A_k$, and F is the maximun number of fields a C struct can have. Thus, a value of a C struct with $k \geq 1$ fields named $l_1, \cdots, l_k$ is a k–tuple R so that each element of R is a pair (l_i, v_i) where each l_i is a string of elements of $(\texttt{Let} \cup \texttt{Char})^*$ and $v_i \in T_i$ for $1 \leq i \leq k$. This algebra has the following standard operations:

- struct *selector*: given a struct $s = ((l_i, v_i), \cdots, (l_k, v_k))$ and a selector l_i, then $s.l_i = v_i \in T_i, 1 \leq i \leq k$.
- struct *modifier*: given a struct s, a selector l_i and a value $v \in T_i$, the expression $s.l_i = v$ denotes a struct value s' such that:

$$s'.l_j = \begin{cases} s.l_j & \text{if } i \neq j \\ v & \text{otherwise, } 1 \leq j \leq k. \end{cases}$$

3.2 C++ Without Templates

From the point of view of object building, C++ [17] can be considered as an extension to C in the sense that its type system allows the definition of classes. The C++ type model is $\mathcal{M}_{C++} = (\mathcal{P}_{C++}, \mathcal{B}_{C++}, \mathcal{L}_{C++})$, where:

- $\mathcal{P}_{C++} = \{\texttt{char}, \texttt{int}, \texttt{short int}, \texttt{long int}, \texttt{float}, \texttt{double}\}$
- $\mathcal{B}_{C++} = \mathcal{C} \cup \mathcal{D}$ with $\mathcal{C} = \{\star, \&, \texttt{arr}, \texttt{fun}, \texttt{struct}, \texttt{union}\}$ and $\mathcal{D} = \{\text{class}, \text{deriv}, \text{mderiv}\}$[¶]

[¶] deriv models inheritance and mderiv models multiple inheritance.

- $\mathcal{L}_{C++} = \mathcal{L}_C$

Type builders in C++ are of two kinds, (type) *constructors* which are essentially the counterparts of C builders and (type) *definers* which build algebras in a less restrictive way because their operations can be defined explicitly. What follows is the description of these definers.

To begin, let us consider the definer **class**. The idea behind this definer is to build an algebra from the types used to represent the underlying abstract data type and its operations. We introduce the following terminology:

- $RT_1, \cdots, RT_k$ are the types of private data members, and $R = \langle RT_1, \cdots, RT_k \rangle$.
- $rf_1, \cdots, rf_t$ denote the private member functions, and $r = \langle rf_1, \cdots, rf_t \rangle$.
- $PT_1, \cdots, PT_m$ are the types of public data members, and $P = \langle PT_1, \cdots, PT_m \rangle$.
- $pf_1, \cdots, pf_s$ denote the public member functions, and $p = \langle pf_1, \cdots, pf_s \rangle$.
- $TC = RT_1 \times \cdots \times RT_k \times PT_1 \times \cdots \times PT_m = R \times P$ is the class representation.

We can now define:

$$\mathbf{class}(R \cdot P, r \cdot p) = C = (V_C, O_C)$$
$$dom(\mathbf{class}) = \{(T, w) \mid T = R \cdot P, w = r \cdot p, |T| \geq 1, |w| \geq 1\}$$

where:

$$V_C = \{TC\} \bigcup_{f \in r \cdot p} (\{CD_f\} \cup AG_f)$$

$$O_C = \{f \mid f \in r \cdot p\}$$

Where the expression $f \in r \cdot p$ means "f appears in the sequence $r \cdot p$", CD_f denotes the type f returns, and AG_f denotes the set containing the types of each argument of f. Thus, the set of domains V_C of this algebra is the union of the class representation and all those types which appear in the prototype of each operation. The set of operations is simply the union of all involved operations.

Consider now the case of **deriv** used to model *inheritance*. Intuitively, we have a base type T_B to which we add new members to get a new type. According to [4] (cf. page 195):

> *"The derived class inherits the properties of its base classes, including its data members and member functions. In addition, the derived class can override virtual functions of its bases and declare additional data members, functions and so on."*

We will show how to define **deriv** when the (public) member functions of T_B are virtual (recall that a member function of T_B is virtual if and only if it can be redefined by an heir of T_B). To this end, we introduce additional notation in the same spirit of definer **class** above:

- R_B, R_D are the types of private data members associated with the base type and derived type, respectively.
- P_B, P_D are the types of public data members associated with the base type and derived type, respectively.
- r_B, r_D denote the private member functions associated with the base type and derived type, respectively.
- p_B, p_D denote the public member functions associated with the base type and derived type, respectively.
- $TC_B = R_B \times P_B, TC_D = R_D \times P_D$ are the representations of the base type and derived type, respectively.

Using these definitions, the definer **deriv** and its domain are given by:

$$\begin{gathered}\mathbf{deriv}(\langle BT\rangle \cdot R_D \cdot P_D, r_D \cdot p_D) = C = (V_C, O_C)\\ dom(\mathbf{deriv}) = \{(T,w) \mid T = \langle BT\rangle \cdot R_D \cdot P_D, w = r_D \cdot p_D\}\end{gathered}$$

where:

$$BT = \mathbf{class}(R_B \cdot P_B, r_B \cdot p_B)$$

$$V_C = \{TC_B \times TC_D\} \bigcup_{f\ \in\ r\cdot p} (\{CD_f\} \cup AG_f)$$

$$O_C = \{f \mid f\ \in\ r \cdot p\},\ r = r_B \cdot r_D,\ p = p_D \oplus p_B$$

The operator $\oplus$ selects all elements in p_D and those of p_B which are not overridden by elements of p_D. Let h be a function in both p_D and p_B. The h in p_D overrides that in p_B if and only if their corresponding CD_h and AG_h agree, that is to say, if they have the same arguments and return type. In summary, the definer **deriv** takes a base class, new types and member functions to create another class whose values are the values of the base class, possibly extended with new values, and whose operations are the collection of all private functions and a combination of public member functions given by the overriding operator $\oplus$.

Finally, the definer **mderiv** (which models multiple inheritance) can be defined along the lines of **deriv** by using an iterative approach, as follows.

$$\begin{gathered}\mathbf{mderiv}(\langle BT_1, BT_2, \cdots, BT_k\rangle \cdot R_D \cdot P_D, r_D \cdot p_D) = C = (V_C, O_C)\\ dom(\mathbf{mderiv}) = \{(T,w) \mid T = \langle BT_1, BT_2, \cdots, BT_k\rangle \cdot R_D \cdot P_D,\\ w = r_D \cdot p_D, k \geq 1,\\ \textit{all elements of } \langle BT_1, BT_2, \cdots, BT_k\rangle \textit{ are different}\}\end{gathered}$$

where:

$$BT_i = \mathbf{class}(R_{B_i} \cdot P_{B_i}, r_{B_i} \cdot p_{B_i}), \text{ for } 1 \leq i \leq k$$

$$V_C = \{TC_{B_1} \times \cdots \times TC_{B_k} \times TC_D\} \bigcup_{f \in r \cdot p} (\{CD_f\} \cup AG_f)$$

$$O_C = \{f \mid f \in r \cdot p\}, r = r_{B_1} \cdots r_{B_1} \cdot r_D, p = p_D \oplus (p_{B_1} \cdots p_{B_k})$$

3.3 Eiffel

In Eiffel [9] an object is either of (primitive) simple type or it belongs to a class which is either predefined (taken from Eiffel's class library), as is the case of Array[T], or defined by the programmer. Eiffel supports multiple inheritance and generic types. As we shall see, the latter can be modeled as functions which generate type builders from a given tuple of types (parameters).

Let us consider Eiffel's type model $\mathcal{M}_{Eiffel} = (\mathcal{P}_{Eiffel}, \mathcal{B}_{Eiffel}, \mathcal{L}_{Eiffel})$, where:

- $\mathcal{P}_{Eiffel} = \{\texttt{INTEGER}, \texttt{REAL}, \texttt{CHARACTER}, \texttt{BOOLEAN}\}$
- $\mathcal{B}_{Eiffel} = \mathcal{L} \cup \mathcal{D} \cup \mathcal{G}$ where $\mathcal{L} = \{\text{Array}[\,], \cdots\}$, $\mathcal{D} = \{\mathbf{class}, \mathbf{deriv}, \mathbf{mderiv}\}$ and $\mathcal{G} = \{\mathbf{gclass}\}$
- $\mathcal{L}_{Eiffel} = \mathcal{L}_C$

The builders **class**, **deriv**, **mderiv** behave essentially in the same way as their C++ counterparts. In Eiffel's terminology [9], a `class` contains a `feature` clause which introduces its features. These can be *attributes* and *routines*. An attribute is the equivalent of a C++ data member, while a routine is the equivalent of a C++ member function. In Eiffel, the visibility of features associated with a class is controlled by the `export` clause. All features under it are visible to any client of the class. On the other hand, any feature not listed as an export is not visible. Thus, the `export` clause is equivalent to the C++ `public` clause.

We can model Eiffel classes exactly in the same way we did with C++ classes. Let $A = \langle A_1, \cdots, A_k \rangle$ be the types of attributes and let $r = \langle r_1, \cdots, r_t \rangle$ be the routines. Then

$$\mathbf{class}(A, r) = C(V_C, O_C)$$
$$dom(\mathbf{class}) = \{(T, w) \mid |T| \geq 1, |w| \geq 1\}$$

where:

$$V_C = \{A_1 \times \cdots \times A_k\} \bigcup_{f \in r} (\{CD_f\} \cup AG_f)$$

$$O_C = \{f \mid f \in r\}$$

Let us now turn our attention to the definition of the builder **gclass** to model Eiffel's parameterized `class` construction. In this case, both attributes and routines are a function of the parameters **T** which is a tuple of types:

$$A[\mathrm{T}] = \langle A_1[\mathrm{T}], \cdots, A_k[\mathrm{T}] \rangle$$
$$r[\mathrm{T}] = \langle r_1[\mathrm{T}], \cdots, r_t[\mathrm{T}] \rangle$$

Thus, we can define:

$$\mathbf{gclass}(\mathrm{T} \cdot A[\mathrm{T}], r[\mathrm{T}]) = \mathbf{class}(A[\mathrm{T}], r[\mathrm{T}])$$
$$dom(\mathbf{gclass}) = \{(T, w) \mid T = \mathrm{T} \cdot A[\mathrm{T}], w = r[\mathrm{T}], |\mathrm{T}| \geq 1, \forall \mathrm{T} : (A[\mathrm{T}], r[\mathrm{T}]) \in dom(\mathbf{class})\}$$

We see that **gclass** can be viewed either as a function that takes a tuple of parameters to build a class (type), or as a builder that takes a function (which transforms tuples of types into tuples of types), and a function (which transforms tuples of types into tuples of names), to produce a type. In other words, **gclass** can be interpreted either as:

$$\mathbf{gclass}\colon \mathcal{T}(\mathcal{M})^+ \rightarrow (\mathcal{T}(\mathcal{M})^+ \times \mathcal{L}^* \rightarrow \mathcal{T}(\mathcal{M}))$$

or as:

$$\mathbf{gclass}\colon (\mathcal{T}(\mathcal{M})^+ \rightarrow \mathcal{T}(\mathcal{M})^+) \times (\mathcal{T}(\mathcal{M})^+ \rightarrow \mathcal{L}^*) \rightarrow \mathcal{T}(\mathcal{M})$$

4 Related Work

From a practical point of view, the need of a concept that unifies the way types are created in different programming languages arises naturally when dealing with the interoperability problem. Intuitively, this is supported by the fact that to achieve interoperability among n languages we only need to build n "mappings" (via such unifying concept) instead of the $O(n^2)$ language–to–language mappings which would otherwise be required. Our contribution aims at the formal definition of such a concept, namely type models.

In distributed object interoperability, the unifying concept is supported by *interface definition languages* or IDLs [19], and the trend seems to be to define a standard IDL. A good example in this direction is OMG's CORBA (Common Object Request Broker Agent) [10]. By using the concept of type models, CORBA can be considered as a common type system associated with the underlying common type model. It has been suggested that CORBA lacks support for overloading and pointer manipulation [13]. We believe that, by using the concept of type models, type systems like CORBA could be improved to better represent a wider family of languages.

In multi–language programming, we want to write a program as a coherent collage of parts written in different languages. Once again, we need a common perspective from which language heterogeneity can be tackled. Examples of common worlds that have been proposed in this field are UTS [6], Polylith [11] and SLI [18]. Once again, each of the (so called) universal type systems these approaches use can be considered as common type systems associated with the corresponding common type models. Previous work related to the last approach [12] acknowledges the need of a concept "above" type systems, but they don't give a formal definition for it. In [14] we have used type models to define a common type model (CTM) and its associated common type system (CTS) to

approach multi–language programming via code reuse. The concept helped us to clearly define the elements to be present in CTS to achieve language interoperability. We have also used these ideas to re–engineer legacy code for ulterior reuse [7] by applying a reverse object–oriented approach dubbed ROOM [16], and to prove that two of ROOM's properties are undecidable [15].

From the point of view of defining formal frameworks to understand types, the classical paper by Danforth and Tomlinson [3] surveys a wide variety of formalisms proposed with this goal in mind. However, none of these approaches seem to try to explain the coexistence of different type systems on a common platform. We believe type models is a step in this direction. However, the discussion is no yet closed as indicated by [1].

5 Summary and Directions for Future Work

We have presented the concept of type models with the goal of providing a common platform on which different type systems can coexist in harmony. This concept has a fundamental importance in language interoperability because we evidently need to deal with the complexity introduced by language heterogeneity. We have given examples to illustrate the concept, and a taxonomy which contains languages that implement different paradigms.

We plan to continue working on the application of type models in the areas mentioned above. We would also like to introduce the notion of correctness by means of which we could certify that the proposed type model captures all the concepts it should. An initial approach would be to try to prove correctness with respect to an operational semantics for the languages in question, along the lines of [2]. Other issues to be considered are late binding, dynamic type checking and type inference.

References

[1] A. Black and J. Palsberg. Foundations of Object–Oriented Languages. *ACM SIGPLAN Notices* 29(3):3–11. 1994.

[2] W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. *In Proc. of OOPSLA'89*, pp. 433–443. 1989.

[3] S. Danforth and Ch. Tomlinson. Type Theories and Object–Oriented Programming. *ACM Computing Surveys* 20(1):29–72. 1988.

[4] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. New York, 1990. Addison–Wesley.

[5] J.V. Guttag and J.J. Horning. The Algebraic Specification of Abstract Data Types. Acta Informatica 10:27-52. 1978.

[6] R.L. Hayes. *UTS: A Type System for Facilitating Data Communication*. PhD Thesis, Dept. of Computer Science, University of Arizona, Tucson (Technical Report 89–16), 1989.

[7] E. Karlsson. *Software Reuse: A Holistic Approach*. Chichester, 1995. John Wiley & Sons Ltd.

[8] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. New York, 1986. Springer–Verlag.

[9] B. Meyer. *Object Oriented Software Construction*. Englewood Cliffs, 1988. Prentice–Hall.

[10] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. OMG Document Number 91.12.1, 1991.

[11] J.M. Purtilo. The POLYLITH Software Bus. *ACM Trans. on Programming Languages and Systems* 16(1):151-174. 1994.

[12] W.R. Rosenblatt, J.C. Wileden and A.L. Wolf. OROS: Toward a Type Model for Software Development Environments. *In of Proc. OOPSLA'89*, pp. 297-304. 1989. ACM Press.

[13] M. Roy and A. Ewald. Distributed Object Interoperability. *Object Magazine* 5(1):18-20, March–April 1995.

[14] A.J. Sánchez–Ruíz. *On Automatic Approaches to Multi–Language Programming via Code Reusability*. PhD Dissertation, Computer Science Department, Rensselaer Polytechnic Institute, 1995.

[15] A.J. Sánchez–Ruíz. Two Undecidable Problems in Multi–Language Reusability Under ROOM. *In Proc. of* PANEL'95, pp. 1163–1174. Canela (Brasil), August 1995.

[16] A.J. Sánchez–Ruíz and E. P. Glinert. ROOM: A Reverse Object–Oriented Approach to Multi–Language Reusability. Technical Report 95–17, Computer Science Department, Rensselaer Polytechnic Institute, November 1995.

[17] B. Stroustrup. *The C++ Programming Language*. New York, 1987. Addison–Wesley.

[18] J.C. Wileden, A.L. Wolf, W.R. Rosenblatt and P.L. Tarr. Specification Level Interoperability. *CACM* 34(5):72-87. 1991.

[19] J. Wing (editor). Proceedings of the Workshop on Interface Definition Languages. *ACM SIGPLAN Notices* 29(8). 1994.