Two Undecidable Problems
In Multi-Language Reusability
Under ROOM*

Arturo J. Sánchez-Ruíz
Centro de Ingeniería de Software y Sistemas (ISYS)
Facultad de Ciencias. Universidad Central de Venezuela
Apartado 47642, Caracas 10141-A, VENEZUELA
E-mail: arturo@cs.rpi.edu, arsanche@conicit.ve

Abstract
ROOM is our (reverse) object-oriented approach to multi-language reusability which consists of wrapping a given reusable software resource by automatically computing a set of augmenting functions (constructors, equality test functions, copy procedures, selectors and modifiers) to transform it into a wrapper class. In this context, the term multi-language reusability means that once the resource is wrapped it can be used from a programming language which is not necessarily the one the resource was originally written in. A resource is complete if and only if it has all the constructors needed to build any possible object (which belongs to the wrapper class). A constructor is minimal with respect to some operation \( f \), if and only if it generates those and only those values in the domain of \( f \). In this paper we prove that the problems of checking (1) whether a resource is complete, and (2) whether a constructor is minimal, are undecidable. We also point out the practical implications these results have in the context of multi-language reusability.

1 Introduction and Overview of ROOM

Software reusability, whether of one's own programs or those written by others, is a main objective of modern software engineering. Since we want to use existing software resources, we cannot impose strong restrictions on the way things are created. Ideally, we should have no say regarding the choice of programming language used to code a (sub)program, neither should we be able to dictate the machine on which that (sub)program executes efficiently. Therefore, heterogeneity arises naturally when we want to put reusability into practice. The term multi-language programming via code reusability (or multi-language reusability, for short) denotes the process of producing software whose various modules exploit the capabilities afforded by language heterogeneity by reusing existing software resources (that is collections of data structures/classes and procedures/functions) as black boxes.

*The work reported here comprises part of the author's dissertation which was partially supported by Consejo de Desarrollo Científico y Humanístico de la UCV and by the Computer Science Department of Rensselaer Polytechnic Institute while the author was a doctoral student. The author wishes to thank his adviser, professor Ephraim P. Glinert, and professor Robert McNaughton for their suggestions to improve early versions of this paper.
ROOM is our reverse object-oriented approach to multi-language reusability which uses the philosophy behind the object-oriented paradigm but follows a reverse path from the representation defined by the resource to a class with this representation. We visualize this class as wrapping the resource, and so call it the wrapper class. In this context, the automatic nature of ROOM consists of automatically computing the wrapper class, which consists of:

- The original data structures and operations acting on them.
- A set of augmenting functions which complete the set of operations needed to reuse the resource. These augmenting functions can be classified into three groups:
  - Those which are always generated and visible to the resource user: constructors (to build any possible class object), equality test functions (to test whether to objects are equal), copy procedures (to copy the value of an object onto another), and assignment.
  - Those that can be generated on demand: selector functions (to select components from structured objects), and modifier procedures (to selectively modify components of structured objects). These, too, are visible to the resource user.
  - Those generated when needed: p-constructors (to create a pointer to an object from a given object), which are not visible to the user.

To specify resources in a language-independent manner we use our CTS, which stands for Common Type System. With CTS one can specify atomic types (integers, reals, etc), concrete types (data structures created with arrays, records, unions, pointers, etc), and abstract types (classes which could use inheritance and genericity). An example of a CTS specification can be seen in Fig. 1, whose first two lines indicate that the resource is implemented in Pascal (language: declaration) and that its implementation can be found in files matrix.h and matrix.p (where: declaration). According to the specification, matrix is an abstract CTS type represented as a record with two fields each of which is an array of pointers to node, which is a concrete CTS type. The interface part of matrix, that is the set of operations acting on matrix, is not shown. This specification is used by ROOM to produce the wrapper class associated with the resource. The semantics of CTS along with details about ROOM can be found in [1].

In this paper we present two undecidable problems associated with the automatic computation of wrapper classes by ROOM. In Section 2 we define the problem of checking whether a resource is complete and show its undecidability proof. In Section 3 we present the problem of checking whether a constructor is minimal (with respect to some operation $f$) and its undecidability proof. Finally, in Section 4, we present our conclusions.

2 Completeness Issues

Although all augmenting functions are important from a practical point of view, we found that once we have an algorithm for building the constructors then the algorithms for building the remaining augmenting functions follow rather easily [1] which leads us to concentrate on
language: Pascal
where: matrix
const MAX = 10;
type vector =
representation: [1..MAX, real2]
interface: (...) eod;
type matrix =
representation:
{row: vec.node; column: vec.node}
interface: (...) eod;
type vec.node = [1..MAX, #node];
type node = {element: real2; row: int; col: int;
next_in_row: #node; next_in_column: #node}

Figure 1: The CTS specification of a sparse matrix.

\[
t ::= \begin{array}{l}
\text{atomic} \mid \text{concrete} \mid \text{abstract} \\
\text{concrete} ::= \text{array}(I, T) \mid \text{pointer}(I, T) \mid \text{record}(f_1 : T_1, \ldots, f_n : T_n) \\
\text{union}(g = u_1(f_1 : T_1, \ldots, f_n : T_n), \ldots, g = u_k(f_1 : T_1, \ldots, f_n : T_n)) \\
\text{abstract} ::= \text{abs}(R)
\end{array}
\]

Figure 2: Abstract syntax of a CTS type definition.

constructors only. We therefore define a resource as complete if and only if it has all the constructors needed to build any possible object value. Intuitively, this means that for each possible value associated with the resource there is an assignment to the formal arguments of the constructor(s) which, when invoked, will produce the value in question.

Formally, let \( t \) be a CTS type definition, whose abstract syntax is depicted in Fig. 2.\(^1\) We denote by \( \mathcal{V}(t) \) the set of possible values associated with \( t \), which can be inductively defined as in Fig. 3. If \( t \) is an atomic CTS type definition, we assume \( \mathcal{V}(t) \) to be defined.

In [1] we show algorithms for building a set of constructors from the CTS specification of a given resource, so as to make it complete (by wrapping it). Therefore, the completeness question is: \textit{How do we know whether a given resource is already complete?}

Without loss of generality, suppose we are given a resource with a function which is the potential constructor. That is to say, we assume that the given resource needs only one constructor, and the alleged one is given. This could be the case for a C++ class, for instance. We need to check whether the alleged constructor can, in fact, be used to construct any possible object value.

Since we have defined the set of possible values associated with any CTS type definition \( t \)

\(^1\) \( I \) denotes a range set in the case of array, and a range of valid run–time addresses in the case of pointer. Also, \( f \) denotes field names, \( g \) denotes the name of a tag (for tagged unions), \( v \) denotes a tag value, and \( R \) denotes a CTS type which is the representation of the abstract type in question. Any other detail such as name of ancestors and/or parameter name(s) is irrelevant in this context.
as $V(t)$, the problem is equivalent to determining whether the given function computes all the values in $V(t)$. We want to prove this problem undecidable.

A standard technique used to prove a given decision problem undecidable consists in reducing another decision problem $Q$, already known to be undecidable, to the problem at hand [5]. The question then is which problem $Q$ do we choose to prove the problem at hand undecidable? Fig. 4 depicts the sequence of reductions we shall show.

The simple language $L_1$ whose three statements are reproduced in Table 1 (where the symbol $\epsilon$ denotes the empty string of $A^*$) is the one presented in [3] to write programs which compute string-valued functions of $m$ variables on $A^*$, where $A = \{1\}$ and $A^*$ is the set of all strings that can be formed with elements of $A$. In [3] a string-valued function of $m$ variables is defined to be partially computable if it is computable in $L_1$, or equivalently if it is computed by a program written in $L_1$. We can now define the problem $P1$ associated with $L_1$ as:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Interpretation in $L_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V \rightarrow 1V$</td>
<td>Put 1 to the left of $V$.</td>
</tr>
<tr>
<td>$V \rightarrow V^-$</td>
<td>Delete the final symbol of $V$. If $V = \epsilon$, leave it unchanged.</td>
</tr>
<tr>
<td>IF $V \neq \epsilon$ GO TO $L$</td>
<td>If the value of $V$ is not $\epsilon$, execute the first instruction labeled $L$; else execute next instruction.</td>
</tr>
</tbody>
</table>

Table 1: The statements of $L_1$. 

![Diagram](image-url)
There are two reasons why we choose this problem as being the base for our undecidability proof. The first is that we can show $P_1$ undecidable by using a powerful result of computability theory known as Rice’s Theorem [3].\footnote{Informally, this theorem states that if we consider any predicate over the set of partially computable which is non-trivial (meaning that the predicate is not a constant function), then the decision problem associated with this predicate (i.e. given a partially computable function, does the function satisfy the predicate?) is not decidable.} Let us recall that the Church–Turing thesis [4] states that a function is partially computable if and only if there exists an algorithm that computes it. Therefore, by invoking the Church–Turing thesis the undecidability of $P_1$ implies that there is no algorithm to check whether a given program $P$ in $L_1$ generates all values of $A^*$. Likewise, once we show the reductions, we will be able to conclude that there is no algorithm to check whether a given resource is complete.

The second reason for choosing this $P_1$ is that we can easily reduce $P_1$ to $P_2$, a simplified version of the completeness problem defined as:

$P_2$: Given a program $P'$ in $L_{\text{list}}$, check whether $P'$ generates all values in $V(\text{list}).$

Just as $L_1$ deals with string-valued functions, we shall define $L_{\text{list}}$ in such a way that it deals with CTS objects of type list, defined in Fig. 5. The idea is that $L_{\text{list}}$ will simulate $L_1$, in the sense that if a program $P$ in $L_1$ computes the function $\Phi_P(x)$ then we can effectively build a program $P'$ in $L_{\text{list}}$ that will compute the function $\Phi_{P'}(\omega(x)) = \omega(\Phi_P(x))$ where $\omega$ is a function that uniquely represents any string of $A^*$ as a CTS list. We must therefore show that such a function exists. The statements of $L_{\text{list}}$, which are quite similar to those of $L_1$, are shown in Table 2.

Having defined problems $P_1$ and $P_2$, we are ready to define problem $P_3$ as:

$P_3$: Given a program $Q$ in $L_{\text{CTS}}$ and a CTS type expression $t$, check whether $Q$ generates all values in $V(t).$

Since a program in $L_{\text{CTS}}$ must be able to manipulate any CTS object, it should have an elaborate repertoire of statements to manipulate CTS primitive objects, arrays, records, pointers, and unions. At this point it is not necessary to give a precise definition of this repertoire, it is enough to think of it as similar to that of any well known high level programming language which supports these types. The fundamental point is that we are assuming $L_{\text{list}}$ is a strict restriction...
of $L_{CTS}$. That is to say, any $L_{list}$ program is an $L_{CTS}$ program whose variables can be of type list only. Therefore, any $L_{list}$ program is an $L_{CTS}$ program.\footnote{We could relax this last condition and require that $L_{list}$ could be \textit{simulated} in $L_{CTS}$, which means we can write subroutines in $L_{CTS}$ for each of the statements of $L_{list}$ in table 2. We prefer the former approach because is simpler.}

### 2.1 One-to-One Correspondence Between $A^*$ and $V$(list)

We begin by showing a structural property of $V$(list) which inductively characterizes the values of $V$(list) and will therefore be useful for defining the one-to-one correspondence between $A^*$ and $V$(list).

**Lemma 2.1** Any value $l$ of $V$(list) is characterized by:

1. either $l = \text{NULL}$, or
2. $l \neq \text{NULL}$ and $l\text{.symbol} = 1$; $l\text{.next} = l'$, where $l' \in V$(list).

**Proof.** By the definition of $V$ (cf. Fig. 3) we have:

$$V(\text{node}) = \{(\text{symbol}, 1, \text{next}, l') \mid l' \in V(\text{+node})\}$$

$$V(\text{+node}) = V(\text{list}) = \{\text{NULL}\} \cup \{(a, b) \mid a \in I, b \in V(\text{node})\}$$

Therefore, we can write:

$$V(\text{list}) = \{\text{NULL}\} \cup \{(a, \text{symbol}, 1, \text{next}, l') \mid a \in I, l' \in V(\text{list})\}$$

and the lemma follows $\Box$

We can now define the function $\omega : A^* \rightarrow V(\text{list})$ which will map each string of $A^*$ to a list of $L_{list}$. Let $x$ be any member of $A^*$, then either $x = \epsilon$ (the empty string) or there is a string $s \in A^*$ such that $x = 1s$. Therefore, we can define $\omega$ as follows:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Interpretation in $L_{list}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre.Insert($V$)</td>
<td>Insert 1 at the beginning of list $V$.</td>
</tr>
<tr>
<td>Post.Eliminate($V$)</td>
<td>Delete the final symbol of list $V$. If $V = \text{NULL}$, leave it unchanged.</td>
</tr>
<tr>
<td>IF $V \neq \text{NULL}$ GO TO $L$</td>
<td>If the value of $V$ is not $\text{NULL}$, execute the first instruction labeled $L$; else execute the next instruction.</td>
</tr>
</tbody>
</table>

Table 2: The statements of $L_{list}$.
The result we want to prove in this section is the

**Theorem 2.1** The function \( \omega \) defined above is one-to-one.

**Proof.** For the injectivity, proceed by structural induction on the elements of \( A^* \). For the surjectivity, proceed by structural induction on the elements of \( \mathcal{V}(\text{list}) \). In both cases the results follow rather easily \( \square \)

### 2.2 \( \Pi_1 \) is Undecidable

Our next step is to prove that there is no algorithm to check, in general, whether a given program in \( L_1 \) generates all values of \( A^* \).

**Theorem 2.2** Problem \( \Pi_1 \) defined above is undecidable.

**Proof.** Let us consider the set:

\[
R_\Gamma = \{ t | \text{range}(\Phi_t) = A^* \}
\]

There are values of \( t \), namely \( t' \) and \( t'' \), such that \( \Phi_{t'} \in R_\Gamma \), and \( \Phi_{t''} \not\in R_\Gamma \). These are the Gödel numbers [3] of programs \( P' \) and \( P'' \) in \( L_1 \), described in Figs. 6 and 7. Recall that each program in \( L_1 \) has a unique Gödel number associated with it (which can be effectively computed from the program), and that each Gödel number uniquely defines a program-in-\( L_1 \) (which can be effectively built from this number). Therefore, by Rice's Theorem, \( R_\Gamma \) is not recursive (i.e. there is no algorithm to compute its characteristic function) and hence problem \( \Pi_1 \) is undecidable \( \square \)

---

\[ \text{In fact, } R_\Gamma \text{ is not even recursively enumerable, as shown in } [6] \text{ (cf. Corollary 1, page 192). This means there is no algorithm that runs forever and prints all the elements of } R_\Gamma. \]
2.3 Reducing P1 to P2

To prove that P1 is reducible to P2 we must show how to construct, for a given program P in L1, an equivalent program P' in List. The equivalence is understood in the sense induced by the function ω defined above, that is to say programs P and P' are equivalent if and only if:

∀x ∈ (A*)m : P'(ω(x)) = ω(P(x))

where (A*)m = A* × m times × A* and ω(x) is understood as ω(x₁, · · ·, xₘ) = (ω(x₁), · · ·, ω(xₘ)).

Lemma 2.2 For each program P in L1 it is possible to build an equivalent program P' in List.

Proof. To construct P' from P we apply the following procedure. For each instruction (l, I) of P, where l is a label and I is an instruction of L₁, generate an instruction (l', I') of Listₙ, such that:

1. l' = l
2. If I = V ← 1V then I' = Prec.Insert(V)
3. If I = V ← V^- then I' = Post.Eliminate(V)
4. If I = IF V ENDS σ GO TO L then I' = IF V ≠ NULL GO TO L

Let us assume that if a variable V that appears in P is initialized to some value x, then the same variable is initialized to ω(x) in P'. On the other hand, if the variable V that appears in P is not initialized, then we initialize the same variable to NULL in P'. To prove the equivalence between P and P' we must introduce the concept of a state of a program [3], which is a list of equations of the form V = m where V is a variable in the program and m is the value of the variable in that state. In the case of P, m ∈ A*, whereas in the case of P', m ∈ List. We assume that all variables which appear in a state have a unique value defined in it. Also, for a program with n instructions, we define a snapshot [3] as a pair (s, i) where s is a state and i is a number such that 1 ≤ i ≤ n + 1.

The interpretation of a snapshot (s, i) is that the current state is s and the i-th statement is about to be executed (by convention, i = n + 1 means that no more statements are to be executed). If J is a statement in a given program, then the effect of J on a snapshot (s, i) is another snapshot (s', i') obtained from (s, i) according to the semantics of J; we denote this effect as (s, i) →ₖ (s', i'). To show that P and P' are equivalent we must then show that, if I is a statement of P such that

(s, i) →ₖ (s', i')

then the corresponding statement I' (according to the procedure above) must be such that:
\[(\omega(s), i) \xrightarrow{(I')} (\omega(s'), i')\]

where (for a state \(s\) of \(P\)) \(\omega(s)\) is defined to be the list of equations \(V = \omega(m)\) (in \(P'\)) obtained from the list of equations \(V = m\) (in \(P\)). From the definition of \(I, I'\) and \(\omega\) these equivalences follow rather easily. We conclude that \(P\) and \(P'\) built in this manner are equivalent \(\Box\)

We are now ready to prove

**Theorem 2.3** Problem \(P_1\) can be reduced to problem \(P_2\), hence \(P_2\) is undecidable.

**Proof.** Let \(t\) be the Gödel number of a program \(P\) in \(L_1\) that computes \(\Phi_1\). From \(t\) we recover the program \(P\). We then apply the procedure described in the previous lemma to get a program \(P'\) which, by the previous lemma, computes \(\Phi'_t(\omega(x)) = \omega(\Phi_t(x))\) for some \(t'\). Therefore, \(\text{range}(\Phi_t) = A^*\) if and only if \(\text{range}(\Phi'_t) = \omega(\text{range}(\Phi_t)) = \omega(A^*) = V(\text{list})\) \(\Box\)

2.4 Reducing \(P_2\) to \(P_3\)

To close the cycle, suppose we had a decision procedure for \(P_3\). We will show how we could use such a procedure to resolve \(P_2\).

**Theorem 2.4** Problem \(P_2\) can be reduced to \(P_3\), hence \(P_3\) is undecidable.

**Proof.** Suppose we have a procedure such that given a program \(P\) in \(L_{\text{CTS}}\) and a \(\text{cts}\) type expression \(t\) we can check whether \(P\) generates all values of \(V(t)\) or not. To decide \(P_2\) we simply set \(P = P'\) and \(t = \text{list}\), where \(P'\) is the given program in \(L_{\text{list}}\) \(\Box\)

We conclude this section with the

**Theorem 2.5** There is no algorithm to check, in general, whether a given resource is already complete or not \(\Box\)

3 On Minimal Sets of Constructors

In general, the set of possible values associated with a data structure is somehow broader than what is really needed, in the sense that if we are using a data structure \(D\) to represent some abstract set of objects \(S\) then clearly every member \(s\) of \(S\) will have a representation \(d\) in \(D\), but not all members \(d\) of \(D\) necessarily represent some object \(s\) of \(S\). As a simple example, consider the typical data structure used to represent binary search trees, consisting of a cell to hold the information at a node and two pointers to the node's descendants. There certainly are possible values of cells which do not represent binary search trees, as illustrated in Fig. 8!

Intuitively, the set of valid values associated with a \(\text{cts}\) type definition is a subset of \(V(t)\) which is application-dependent. To make this more precise let us assume that, without loss of generality, we are given a resource with only one operation that manipulates it. Consider, for
instance, binary search trees represented as cells, with the operation print to print the tree in some order. Formally, the set of valid values associated with a CTS type expression \( t \) (with respect to an operation \( f \)) is:

\[
\text{valid}(t) = \{ x \in V(t) \mid f(x) \}
\]

Thus, the valid values are the set of values for which the function computed by \( f \) is defined. The "binary search tree" in Fig. 8 is invalid because any attempt, say, to print it will get trapped in an infinite loop.

Our approach to completing a resource is to compute a constructor for each type present in it. As an example, let us consider again the resource described in Fig. 1. To complete this resource four constructors would be needed, namely one constructor for each type: node, vec_node, matrix and vecor. However, any clever programmer would build only one constructor! This constructor (let us call it Insert, for instance) would take a matrix, a couple of índices, and a real value and would insert it appropriately in the data structure, as shown in Fig. 9. Note that this constructor does not build the set of possible values for type matrix. Given a real matrix \( M = \{m_{ij}\} \), the corresponding value \( m \) of type matrix is such that, for all \( m_{ij} \neq 0 \):

a. \( m_{ij} \) belongs to the list \( m.\text{row}[i] \).

b. \( m_{ij} \) belongs to list \( m.\text{column}[j] \).

c. The node \((m_{ij}, i, j, \alpha, \beta)\) appears exactly once in the data structure, where \( \alpha \) and \( \beta \) are the values of type p-node that get defined once \( m_{ij} \) is inserted in the data structure.

Our intuitive idea of a minimal constructor says that it should generate only valid values. Therefore, a constructor \( P \) is minimal (with respect to an operation \( f \)) if and only if:

\(\text{Insert (m: matrix; row: integer; column: integer; value: real) }\{
\begin{array}{l}
n = \text{node (value, row, column, NULL, NULL)}; \\
\text{Insert n at the beginning of list m.row[row]}; \\
\text{Insert n at the beginning of list m.column[column]};
\end{array}\}\)
range(\Phi_P) = valid(t).

or, equivalently, if and only if:

range(\Phi_P) = dom(\Phi_f).

Thus, for a general resource, the minimal set of constructors is the one formed by the minimal constructor associated with each abstract type. Thus, it makes sense to consider the problem:

**P2**: Given a cts resource with a constructor P and an operation f, is P minimal with respect to f? Equivalently, given programs P and P' in LCT, is range(\Phi_P) = dom(\Phi_{P'})?

We would like to prove that P3' is undecidable by using the reduction technique illustrated in the previous section. To this end, let us define the problems:

**P1**: Given P, P' in L1, is range(\Phi_P) = dom(\Phi_{P'})?

**P2**: Given P, P' in list, is range(\Phi_P) = dom(\Phi_{P'})?

Since every program in L1 is effectively transformable to an equivalent program in list and since every list program is an L.rs program, the reductions of P1' to P2' and of P2' to P3' follow immediately.

**Theorem 3.1** Problem P1' can be reduced to problem P2' which, in turn, can be reduced to P3' □

We should now prove

**Theorem 3.2** P1' is undecidable.

**Proof.** Let us assume we have a decision procedure D for P1'. Thus, for arbitrary programs P, P' in L1, D(P, P') decides the predicate \( R(P, P') = [\text{range}(\Phi_P) = \text{dom}(\Phi_{P'})] \). Let P be the program in Fig. 6 and Q be an arbitrary program. Since range(\Phi_P) = A*, D(P, Q) decides the predicate \( T(Q) = [\text{dom}(\Phi_Q) = A^*] \). Now, this predicate is not trivial (as shown in Theorem 2.2) and therefore undecidable by Rice's Theorem. We conclude that D doesn’t exist □

We conclude this section with

**Theorem 3.3** There is no algorithm to check, in general, whether a given constructor is minimal with respect to a given operation □
4 Conclusions

We have shown two undecidability results associated with the automatic computation of wrapper classes by ROOM to achieve multi-language reusability. The undecidability of the completeness problem tells us that when we are given a resource to be reused, we have to compute all constructors to make it complete. Therefore, we cannot check whether this computation can be skipped or not. However, this result is not as pessimistic as it might look, because in [1] we show algorithms to complete a given resource by wrapping it. On the other hand, the undecidability of the minimality problem tells us that we cannot expect ROOM to build sophisticated constructors such as those that only generate valid values. Thus, ROOM's approach of building all constructors for all major types (those which do not depend on other types) seems to be the safest.

References