Università degli Studi di Bologna
DEIS

On the recursive generation of parametric types

Mirko Viroli

September 28, 2000

Draft

DEIS Technical Report no. DEIS-LIA-00-002 *LIA Series no. 42*

On the recursive generation of parametric types

Mirko Viroli

DEIS - Universita' degli Studi di Bologna
Via Rasi e Spinelli 176, 47023 Cesena (FC) - Italy
mviroli@deis.unibo.it

September 28, 2000

Draft

Abstract. For efficiency reasons, the typical implementations of parametric polymorphism in modern languages are built so as to prepare all the information about the instantiations of the parametric types an application use before its actual execution. In some situations however, a program may recursively refer an infinite set of such instantiations, causing the compilation failing to terminate. Most current C++ compilers and proposals for adding parametric polymorphism in Java suffer from this problem. In this paper we introduce a formal framework for reasoning about parametric types, type variables and their dependency. Using this, we analyze in a formal way this non-termination issue and give a necessary and sufficient condition for a program not to cause this inconvenience. By testing this condition future versions of all the compilers may provide for early detection of the problem.

Keywords: *Compilers, Parametric Polymorphism, Program Specification*

Contents

1 Introduction

Parametric polymorphism is a valuable programming mechanism which has been widely used to achieve a further degree of reuse with respect to that allowed by inclusive polymorphism (inheritance). Typical implementations are *ADA* generics [US80], C++ templates [ES90], and *ML* polymorphic functions or data types [Mil85, Ull94]. Even Java, which initially didn't provide it, is due to be extended with this mechanism [S99]. Several proposals for adding genericity to Java, by using parametric types, have in fact been recently published [VN00, SA98, CS98, AFM97, MBL97, OW97]. The two main issues that a compiler for parametric polymorphism should deal with are (i) run-time performance and (ii) the integration of parametric types with the core typing. In order to obtain good results for both these aspects, the typical technique relies on gathering, in structures called type descriptors, all the information about the instantiations of parametric types an application uses. This is usually done before the actual execution, by creating class files at compile-time, as with C++ templates or the *NextGen* translator for Java [CS98]. These implementations in fact, make it possible both to anticipate some management, thus allowing good performance at run-time, and to prepare the information needed to support parametric type at run-time as well, making the integration possible. An important issue that has not been addressed yet however, is whether the set of instantiations of parametric types that an application uses is finite. When this set is infinite these implemantions fail to prepare the type descriptors needed, so for instance a C++ compilation would fail to terminate. A very simple example is that in which a generic class `Cell<T>` has a field of type `Cell<Cell<T>>`. In this case in fact, the compilation of an instantiation of `Cell`, say `Cell<String>`, would require type information about `Cell<Cell<String>>`, which requires that of `Cell<Cell<Cell<String>>>` and so on. Consider one of the most trivial implementations for parametric types: full macro-expansion. When using `Cell<String>` a new class `Cell_String` is compiled, in which all the occurrences of the type variable `T` are substituted with `String`. Thus the compiler would try to compile recursively `Cell_String`, `Cell_Cell_String`, and so on, failing to terminate. Surprisingly this issue has not been addressed yet; so, one of the consequences is that the most widely-used compilers for C++, namely *Visual C++*, *Borland C++* and *GCC*, suffer from this problem. In particular the former two compilers cause the issuing of an *out of memory* error by trying to build as many class representations as possible, while *GCC* intercepts the recursive generation when the parameterization depth overcome the limit set by *ANSI/ISO C++*, which is currently 17. *NextGen* and the heterogenous version of *Pizza* [OW97] have the same drawback. The ideas presented in this paper are part of a project for building a translator for parametric polymorphism in Java by not relying on the macro-expansion technique, called LM [VN00]. In fact in the Java framework, where class files are often loaded through the net from a remote computer, an important issue is that of keeping the size of class files as small as possible. The macro-expansion technique does not give any guarantee to do so, since the size of a library can grow a lot due to the many instantiations of parametric types being used by applications. In the LM translator instead, type descriptor objects are recursively built at load-time, so as to avoid any code expansion. Even though it does not relies on macro-expansion, the LM translator suffer from a particular kind of infinite-generation problem as well: the execution fails to terminate as it attempts to build infinite type descriptors at load-time. In [VN00] we have already mentioned the problem, depicting a solution based on a by-need creation of type descriptors, hence allowing all applications to be executed whether they refer a finite number of types or not. In this paper however, we are interested in analyzing the problem and finding a condition for its detection that would be applicable in general to all the compilers for languages with parametric polymorphism. In particular we give a formal description for this issue, along with a condition which we prove to be sufficient and necessary for a program not to cause this infinite generation, thus allowing early detection. At this time many formal descriptions of parametric polymorphism have been studied. In [BCC98] history and many citations on works about encoding object structures in the framework of λ-calculus are given. The name-based approach is instead studied in [APW99], where a core calculus for an extension of Java with parametric types called GJ is introduced, along with an operational semantics and a type system for it. Our aim is to reason about the instantiations a given parametric type uses, and the dependencies between their type variables. Thus we have introduced a novel formal framework with the key goal of focusing on these aspects. In particular its aim is to obtain a graph representation of the dependencies between the type variables; thus our condition is expressed as the absence of a certain type of cycle in this graph. The reminder of the paper is organized as follows. Section 2 gives useful definitions and properties, which define our mathematical framework. Section 3 introduces the condition and shows the proof of its necessity and sufficiency. Section 4 concludes.

2 Definitions and Properties

For the following discussion we will refer to the simple set of classes shown in Figure 1, which uses a notation for parametric polymorphism in Java that has become a standard [OW98]. Class `A<T>` has a trivial constructor and uses two parametric types, `B<T,T>` and `A<String>`, as types for its fields. Analogously `B<R,S>` uses `B<S,A<R>>` and `A<S>`. Notice that some of the parametric types' instantiations used by a class may depend on the current instantiation of some of its type variable, as for `B<T,T>` in class `A<T>`. The class `String` instead is non generic, and we suppose that it does not use any parametric type. We will use this example to explain the mathematical definitions we give, and to prove that this program is not *generating-safe*,

```
class A<T>{          class B<R,S>{          class String{
    B<T,T> b;           B<S,A<R>> b;           ...
    A<String> a;        A<S> a;                // a non-generic class
    A(){}               B(){}                 ...
}                    }                      }
```

Figure 1. *The case study program*

i.e., its typical compilers cause a recursive generation of type descriptors leading to a non termination of the compilation process. In the following sections for simplicity, we sometimes avoid expressing universal quantification over free variables where this can be inferred by the context in which they are used.

Definition 1. *A* program signature *is defined as a couple* $\mathcal{Q} = (\mathcal{G}, \|.\|)$ *in which* $\mathcal{G}$ *is a finite set of names of generic class, and* $\|.\| : \mathcal{G} \mapsto \mathbb{N}$ *is a function mapping generic classes to their number of type parameters. When this number is zero the class is said to be* simple*.*

A program signature defines, so to speak, the structure of a program. For instance considering our sample program we have:

$$\mathcal{Q} = (\{A, B, String\}, \{A \mapsto 1, B \mapsto 2, String \mapsto 0\})$$

In order to introduce the notion of referenced type, which will be useful to determine the condition for early detection, we first define the notion of instantiation of a parametric type.

Definition 2. *Given a program signature* $\mathcal{Q} = (\mathcal{G}, \|.\|)$*, an* instantiation dependent on n variables in $\mathcal{Q}$ *is an element of the following set:*

$$\mathcal{I}_n^{\mathcal{Q}} = \{g\langle x_1, .., x_{\|g\|}\rangle : g \in \mathcal{G}, \quad x_j \in \{\#1, .., \#n\} \cup \mathcal{I}_n^{\mathcal{Q}}\}$$

Each x_j, which is said to be the j^{th} *type variable* of the instantiation, can be bound to one of the n type variables of the scope ($\in \{\#1, .., \#n\}$), or to another instantiation ($\in \mathcal{I}_n^{\mathcal{Q}}$). In general we will use the notion of instantiation dependent on n variables as a way of representing instantiation for parametric types that depend on the current binding of some other type variables. For example, in the scope of the generic class `A<T>` we use the type `A<String>`, which can be directly represented with the instantiation $A\langle String\rangle$. In the scope of `B<R,S>` the type `B<S,A<R>>` can be represented by $B\langle\#2, A\langle\#1\rangle\rangle$. For the sake of simplicity, in the rest of the paper we will refer to a unique program signature $\mathcal{Q}$, so we will simply use the symbol $\mathcal{I}_n$ to denote an instantiation dependent on n variables. Notice that the relation $\mathcal{I}_n \subset \mathcal{I}_{n+1}$

holds for each $n \in \mathbb{N}$. We introduce the symbols $\mathcal{I} := \mathcal{I}_0$ (set of *full-instantiations*) and $\mathcal{I}^+ = \bigcup_n \mathcal{I}_n \setminus \mathcal{I}$ (set of *free-instantiations*), along with the shorthand $\#\mathbb{N}$ for $\{\#n : n \in \mathbb{N}\}$. To access an instantiation's name and an instantiation's type variable we will use the following notations:

$$\underline{g\langle x_1, .., x_{\|g\|}\rangle} := g \qquad g\langle x_1, .., x_{\|g\|}\rangle|_j := x_j$$

In order to check if an instantiation dependent on n type variables actually uses one of them, or has a given full instantiation within one of its type variables, we use the symbols $\preceq$ and $\prec$ defined in the following way:

$$\forall i \in \mathcal{I}_n, \forall x \in \{\#1, .., \#n\} \cup \mathcal{I} \qquad \begin{array}{l} x \preceq i \Leftrightarrow \exists k\ (x = i|_k \vee x \preceq i|_k) \\ x \prec i \Leftrightarrow x \preceq i \wedge \nexists k\ i|_k = x \end{array}$$

So we have for example $\#1 \preceq A\langle\#1\rangle$, $\#1 \not\prec A\langle\#1\rangle$, $\#1 \prec A\langle A\langle\#1\rangle\rangle$ and finally $A\langle String\rangle \preceq A\langle A\langle String\rangle\rangle$. Notice that if $x \prec i$ then i is of the kind $\underline{i}\langle .., g\langle .., x, ..\rangle, ..\rangle$ for some $g \in \mathcal{G}$. For any set X we write $|X| < \infty$ to state that X has a finite number of elements, and $|X| = \infty$ in the opposite case. We will deal with parametric types used within the definition of generic classes as expressed in the following definition:

Definition 3. *We define a* program *a couple* $\mathcal{P} = (\mathcal{Q}, \widehat{.})$ *with* $\mathcal{Q} = (\mathcal{G}, \|.\|)$ *a program signature as usual, and* $\widehat{.}$ *a function, called* using function, *mapping generic classes to finite sets of instantiations, so that* $\widehat{g} \subset \mathcal{I}_{\|g\|}$, $|\widehat{g}| < \infty$ *and* $\|g\| = 0 \Rightarrow \widehat{g} = \{\}$

The function $\widehat{.}$ of a program maps a generic class to all the instantiations it uses. In our case we have:

$\widehat{A} = \{A\langle String\rangle, B\langle\#1,\#1\rangle\}$ (`class A<T>{..A<String>..B<T,T>..}`)

$\widehat{B} = \{B\langle\#2, A\langle\#1\rangle\rangle, A\langle\#2\rangle\}$ (`class B<R,S>{..B<S,A<R>>..A<S>..}`)

$\widehat{String} = \{\}$ (`class String{ ........ }`)

The operation of *parametric instantiation*, taking an instantiation and some actual types, and performing the binding of the type variables thus obtaining a full-instantiation, can be realized with the following operator:

$$g\langle x_1, .., x_{\|g\|}\rangle \bowtie [y_1, .., y_m] = g\langle z_1, .., z_n\rangle,\ z_j = \begin{cases} y_\lambda & \text{if } x_j = \#\lambda \\ x_j & \text{if } x_j \in \mathcal{I} \\ x_j \bowtie [y_1, .., y_m] & \texttt{otherwise} \end{cases}$$

For instance, the idea that when binding R to $String$ and S to $A\langle String\rangle$ the type `B<R,S>` becomes instantiated to `B<String,A<String>>` can be expressed by the equation:

$$B\langle\#1,\#2\rangle \bowtie [String, A\langle String\rangle] = B\langle String, A\langle String\rangle\rangle$$

Definition 4. *Given a full-instantiation $i \in \mathcal{I}$ we call the set of* full-instantiations directly referenced by i*, the set:*

$$dir(i) := \{j \bowtie [i|_1, .., i|_{\|\underline{i}\|}] : j \in \widehat{\underline{i}}\}$$

We have for example:

$$dir(A\langle String\rangle) = \{A\langle String\rangle, B\langle String, String\rangle\}$$

By applying the transitive closure to the *directly-referenced* relation, we obtain the *referenced* relation as follows:

Definition 5. *Given a full instantiation $i \in \mathcal{I}$ we call the set of* full-instantiations referenced by i *the following set:*

$$ref(i) := \{i\} \cup \bigsqcup_{j \in dir(i)} ref(j)$$

The goal of this paper then is to give a necessary and sufficient condition for a given program $\mathcal{P}$ not to have full-instantiations with an infinite number of referenced types.

Definition 6. *A given program $\mathcal{P} = (\mathcal{Q}, \widehat{\cdot})$ is said to be* generating-safe *if and only if:*

$$\forall i \in \mathcal{I}^Q : |ref(i)| < \infty$$

To introduce a condition for this property to hold, we need to build a graph structure describing the dependencies between the type variables of different classes.

Definition 7. *Given a program $\mathcal{P}$ we call the* type variable dependencies graph associated to $\mathcal{P}$ *the triplet $\mathcal{M}^{\mathcal{P}} = (\mathcal{H}, \mathcal{A}, \mathcal{A}^{\diamond})$ with $\mathcal{A}, \mathcal{A}^{\diamond} \subset \mathcal{H} \times \mathcal{H}$ so that:*

$$\mathcal{H} = \{(g/m) : g \in \mathcal{G}, 1 \leq m \leq \|g\|\}$$
$$\mathcal{A} = \{(g'/m', g''/m'') : \exists g''\langle x_1, .., x_{\|g''\|}\rangle \in \widehat{g'},\ \#m' = x_{m''}\}$$
$$\mathcal{A}^{\diamond} = \{(g'/m', g''/m'') : \exists g''\langle x_1, .., x_{\|g''\|}\rangle \in \widehat{g'},\ \#m' \prec x_{m''}\}$$

Again we will simply write $\mathcal{M}$ instead of $\mathcal{M}^{\mathcal{P}}$. $\mathcal{A}$ is said to be the set of *simple arcs*, while $\mathcal{A}^{\diamond}$ is said to be the set of *nesting arcs*. In general each free-instantiation used within a generic class introduces one or more arcs in the graph. For instance the type variable dependencies graph of our program is the one of Figure 2. Within class `A<T>`, the type `A<String>` introduces no arcs, while `B<T,T>` introduces the simple arcs $(A/1, B/1)$ and $(A/1, B/2)$, `B<S,A<R>>` in `B<R,S>` introduces the simple arc $(B/2, B/1)$ and the nesting arc $(B/1, B/2)$, while `A<S>` introduces the simple

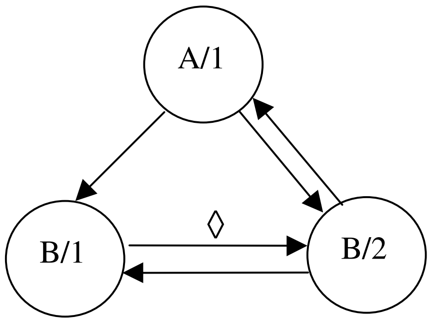

Figure 2. *The type variable dependencies graph of the sample program*

arc $(B/2, A/1)$. Let $a = (g/m, h/n) \in \mathcal{A}$ and $a^\diamond = (g^\diamond/m^\diamond, h^\diamond/n^\diamond) \in \mathcal{A}^\diamond$, we define the two following sets (which are not void due to the definition of $\mathcal{A}$ and $\mathcal{A}^\diamond$):

$$\mathcal{T}_a := \{j \in \widehat{g} : \quad \underline{j} = h \wedge \#m = j|_n\} = \{h\langle i_1, .., i_{n-1}, \#m, i_{n+1}, .., i_{\|h\|}\rangle \in \widehat{g} : i_\lambda \in \mathcal{I}\}$$

$$\begin{aligned} \mathcal{T}^\diamond_{a^\diamond} := \{j \in \widehat{g^\diamond} : \quad \underline{j} = h^\diamond \wedge \#m^\diamond \prec j|_{n^\diamond}\} = \\ = \{h^\diamond\langle i_1, .., i_{n-1}, x\langle ..\#m^\diamond ..\rangle, i_{n+1}, .., i_{\|h^\diamond\|}\rangle \in \widehat{g} : i_\lambda \in \mathcal{I} \wedge x \in \mathcal{G}\} \end{aligned}$$

Informally $\mathcal{T}_a$ contains the free-instantiations used by g responsible for the simple arc a to be inserted in the graph, and $\mathcal{T}^\diamond_a$ those responsible for the nesting arc $a^\diamond$ to be in the graph. From the definition of $\mathcal{T}_a$ and $\mathcal{T}^\diamond_a$ the following useful properties hold:

$$\text{given} \quad i \in \mathcal{I},\ \underline{i} = g,\ j \in \mathcal{T}_a, \quad \text{we have} \quad i' := j \bowtie [i|_1, .., i|_{\|\underline{i}\|}] \Rightarrow i|_m = i'|_n \qquad (1)$$

$$\text{given} \quad i \in \mathcal{I},\ \underline{i} = g^\diamond,\ j \in \mathcal{T}^\diamond_{a^\diamond}, \quad \text{we have} \quad i' := j \bowtie [i|_1, .., i|_{\|\underline{i}\|}] \Rightarrow i|_{m^\diamond} \prec i'|_{n^\diamond} \qquad (2)$$

Roughly speaking, these properties highlight in which way the instantiation of a type variable may flow from a full-instantiation i to an instantation i' obtained by applying the bind operator to an element in $\mathcal{T}_a$. In particular the type variable instantiation is exactly copied in the case of simple arcs and it is inserted within a type variable of i' in the case of nesting arcs. Now, let $i \in \mathcal{I}$ and $j \in \mathit{dir}(i)$, we define the set $\mathcal{J}^i_j$ (which is non-void due to the definition of $\widehat{.}$ and $\mathit{dir}(.)$) as:

$$\mathcal{J}^i_j := \{t \in \widehat{\underline{i}} : t \bowtie [i|_1, .., i|_{\|\underline{i}\|}] = j\}$$

$\mathcal{J}^i_j$ contains those instantiations used by $\underline{i}$ (in the sense of using function $\widehat{.}$) responsible for j being an instantiation directly referenced by i. The following properties hold:

$$t \in \mathcal{J}^i_j,\ t|_k \in \mathcal{I} \Rightarrow j|_k = t|_k \tag{3}$$

$$t \in \mathcal{J}^i_j,\ t|_k = m \Rightarrow j|_k = i|_m \tag{4}$$

$$t \in \mathcal{J}^i_j,\ t|_k \in \mathcal{I}^+ \Rightarrow (m \prec t|_k \Leftrightarrow \exists x \in \mathcal{G}\ j|_k = x\langle .., i|_m, ..\rangle) \tag{5}$$

They show how a type variable flows from an instantiation i to another j depending on the instantiations in $\mathcal{J}^i_j$. For the following discussion it will be helpful to introduce the notion of cycles of a graph $cl(F, E)$, with F set of nodes and $E \subset F \times F$ set of oriented edges, as:

$$cl(F, E) = \{(f_1, .., f_n) \in F^n : n > 1, ((f_1, f_2), (f_2, f_3), .., (f_n, f_1)) \in E^n\}$$

Now we are ready for theorem and proof.

3 The theorem

Theorem 1. *Given a program* $\mathcal{P} = ((\mathcal{G}, \|.\|), \widehat{.})$*, let* $\mathcal{M} = (\mathcal{H}, \mathcal{A}, \mathcal{A}^\diamond)$ *be its type variable dependencies graph, then* $\mathcal{P}$ *is safe-generating if and only if* $\mathcal{M}$ *has no cycles with nesting arcs:*

$$\forall i \in \mathcal{I}, (|ref(i)| < \infty) \Leftrightarrow cl(\mathcal{H}, \mathcal{A}) = cl(\mathcal{H}, \mathcal{A} \cup \mathcal{A}^\diamond) = cl(\mathcal{H}, \mathcal{A} \setminus \mathcal{A}^\diamond)$$

Proof. Notice that the expression in the right side of the equivalence asserts that adding or dropping nesting arcs from the graph would not change its set of cycles, which is equivalent to assert that there are no cycles with nesting arcs.

Sufficiency. Let $(g_1/m_1, .., g_n/m_n) \in cl(H, A \cup \mathcal{A}^\diamond)$ be a cycle, and let the element $(g_l/m_l, g_{l+1}/m_{l+1}) \in \mathcal{A}^\diamond$ be the only nesting arc of this cycle. We can choose a sequence $t_1, .., t_n$, so that:

$$t_k \in \begin{cases} \mathcal{T}^\diamond_{g_k/m_k, g_{k+1}/m_{k+1}} & \texttt{if } k = l \\ \mathcal{T}_{g_k/m_k, g_1/m_1} & \texttt{if } k = n \\ \mathcal{T}_{g_k/m_k, g_{k+1}/m_{k+1}} & \texttt{otherwise} \end{cases}$$

Given an initial instantiation i_1 so that $\underline{i_1} = g_1$, by recursively applying the bind operator in the following way:

$$i_{j+kn+1} := t_j \bowtie [i_{j+kn}|_1, .., i_{j+kn}|_{|g_j|}] \quad \forall k \geq 0, \forall j : 1 \leq j \leq n$$

we obtain an infinite sequence in $\mathcal{I}$ so that $i_{k+1} \in dir(i_k)$ for all $k > 0$. For Properties 1 and 2 we have:

$$\begin{aligned} i_1|_{m_1} = i_2|_{m_2} = .. = i_l|_{m_l} \quad &\prec \quad i_{l+1}|_{m_{l+1}} = i_{l+2}|_{m_{l+2}} = .. = i_n|_{m_n} = \\ = i_{n+1}|_{m_1} = i_{n+2}|_{m_2} = .. = i_{n+l}|_{m_l} \quad &\prec \quad i_{n+l+1}|_{m_{l+1}} = i_{n+l+2}|_{m_{l+2}} = .. = i_{2n}|_{m_{2n}} = \\ = i_{2n+1}|_{m_1} = i_{2n+2}|_{m_2} = .. = i_{2n+l}|_{m_l} \quad &\prec \quad i_{2n+l+1}|_{m_{l+1}} = i_{2n+l+2}|_{m_{l+2}} = .. = i_{3n}|_{m_{2n}} = .. \end{aligned}$$

and hence:

$$i_1|_{m_1} \prec i_{n+1}|_{m_1} \prec i_{2n+1}|_{m_1} \prec .. \prec i_{kn+1}|_{m_1} \prec ..$$

It follows that the subsequence $(i_{kn+1})_{k\geq 0}$ has infinite different elements (their m_1^{th} type variable always being different), so the set $\{i_k : k \geq 1\}$ has infinite elements. As for each k we have $i_k \in \mathit{ref}(i_1)$, then $|\mathit{ref}(i_1)| = \infty$, hence the contradiction. The sufficiency is proved. We omit the proof in the case that a cycle has more than one nesting arc, which is at this point quite strightforward.

Necessity. Suppose the dependencies graph has no cycles with nesting arcs. Consider an infinite sequence $(i_k)_{k>0}$ in which $i_{k+1} \in \mathit{dir}(i_k)$ for all $k > 0$, and a sequence $(t_k)_{k>0}$ so that $t_k \in \mathcal{J}_{i_{k+1}}^{i_k}$ for all $k > 0$. The set of all the instantiations of type variables in the sequence, Y, which is $\{i_k|_j : k > 0, 1 \leq j \leq \|\underline{i_k}\|\}$ (we write $Y = \{i_k|_j\}$ for the sake of brevity) is the union of the three following sets:

$$Y_1 = \{i_1|_j\} \qquad Y_2 = \{i_k|_j : k > 1 \wedge t_{k-1}|_j \in \mathcal{I}\}$$
$$Y_3 = \{i_k|_j : k > 1 \wedge t_{k-1}|_j \in \mathcal{I}^+ \cup \#\mathbb{N}\}$$

Y_1 is clearly finite. Y_2 is finite too, since due to Property 3 we also have $Y_2 = \{t_k|_j \in \mathcal{I}\}$, and $\cup\{t \in \widehat{g}, g \in \mathcal{G}\}$ being clearly finite, it follows that Y_2 is finite too. Now let $i_k|_j$ be any element in Y_3. We build a sequence $(q_\lambda)_{\lambda\geq 0}$, with $q_0 = j$ and $1 \leq q_\lambda \leq \|\underline{i_{k-\lambda}}\|$, which identifies the elements $i_{k-\lambda}|_{q_\lambda} \in Y$, in the following way:

$$q_{\lambda+1} = \begin{cases} \epsilon & \text{if } t_{k-\lambda-1}|_{q_\lambda} = \#\epsilon \\ \text{any element in } \{m : q_\lambda \prec t_{k-\lambda-2}|_m\} & \text{otherwise} \end{cases} \tag{6}$$

Notice that the set $\{m : q_\lambda \prec t_{k-\lambda-2}|_m\}$ is always not empty due to Property 5. Then we define the sequences $(r_\lambda)_{\lambda\geq 0}$ in $\mathcal{I}$, and $(a_\lambda)_{\lambda\geq 1}$ in $\mathcal{A} \cup \mathcal{A}^\diamond$ with:

$$r_\lambda := i_{k-\lambda}|_{q_\lambda} \qquad a_{\lambda+1} := (\underline{i_{k-\lambda-1}}/q_{\lambda-1}, \underline{i_{k-\lambda}}/q_\lambda)$$

We stop the recursive process when we find a $\lambda = \gamma$ so that $r_\gamma \in Y_1 \cup Y_2$. Notice that such a λ exists, in fact when $\lambda = k-1$ we have $r_\lambda \in Y_1$. The sequence $(a_\lambda)_{1\leq\lambda\leq\gamma}$ identifies a path in the dependencies graph, with simple arcs when $t_{k-\lambda-1}|_{q_\lambda} \in \#\mathbb{N}$ (former case in Equation 6) and nesting arcs otherwise. For hypothesis the graph has no cycle with nesting arcs, thus the latter case, where $t_{k-\lambda-1}|_j \in \mathcal{I}^+$, may occur at most $|\mathcal{A}^\diamond|$ times. But due to Property 1 and Property 2, only in the latter case the relation $r_{\lambda+1} \neq r_\lambda$ holds, hence the sequence r_λ may have at most $|\mathcal{A}^\diamond|$ different elements. Since each element in Y_3 belongs to a sequence of instantiations starting from an element in $Y_1 \cup Y_2$ which has at most $|\mathcal{A}^\diamond|$ elements, we have $|Y_3| \leq |Y_1 \cup Y_2| * |\mathcal{A}^\diamond|$, thus Y_3 is finite too. So Y, which is the set of all the instantiations of the type variables of the sequence, is finite; $\mathcal{G}$ is also finite, so the sequence has a finite set of elements. Now consider a tree of full-instantiations with root $i_1 \in \mathcal{I}$, and in which each node s has all its directly referenced types ($dir(s)$) as sons, omitting those that are anchestors of s, i.e. those included in the path from i_1 to s. This tree has finite depth, since each path starting from the root towards the sons determines a sequence such as the one described above, or possibly a subsequence of it. Moreover this tree contains all the instantiations referenced by i_1. However the set of nodes of the tree is finite, since each node has a number of sons which is at most $max|\widehat{g}|_{g \in \mathcal{G}}$, and its depth is finite . Such a property holds for each $i_1 \in \mathcal{I}$, so $\forall i \in \mathcal{I} : |ref(i)| < \infty$. $\square$

Using the result given by this theorem we infer that our sample program is not generating-safe, in fact it has the cycle $(A/1, B/1, B/2, A/1^*)$. So, starting from an instantiation $i_1 = A\langle W\rangle$ (where W is any full-instantiation, say it is $String$) we may write the sequence:

$$(i_k) = (A\langle W\rangle, B\langle W, W\rangle, B\langle W, A\langle W\rangle\rangle, A\langle A\langle W\rangle\rangle,$$
$$B\langle A\langle W\rangle, A\langle W\rangle\rangle, B\langle A\langle W\rangle, A\langle A\langle W\rangle\rangle\rangle, A\langle A\langle A\langle W\rangle\rangle\rangle, ..) =$$
$$= (A\langle W\rangle, .., A\langle A\langle W\rangle\rangle, .., A\langle A\langle A\langle W\rangle\rangle\rangle, ..)$$

This sequence has infinite different elements, and all are instantiations referenced by i_1.

4 Conclusions

In this paper we addressed an issue which has not been dealt with either by previous papers or by existing commercial products: most compilers for languages with parametric polymorphism are built so that the compilation process fails to terminate when recursive and nested referentiation of parametric types occurs. The condition that we proved to be sufficient and necessary for the detection of the problem can be tested during a preliminary step of checking, allowing early detection. In trivial implementations the source for all the generic classes a program uses can be examined, so as to check for the presence of dependency cycles with nesting arcs. When this might be too expensive, a more feasible solution would be to keep a description file containing the dependencies graph, which would be updated each time a new class is compiled, or an older one is modified. The same result can be applied to those environments, such as the LM translator for Java, where instantiation-binding is resolved at load-time. There the condition for detecting recursive generation can be tested during class loading, for instance by keeping the description file in the memory and updating it each time a new class is loaded. In general, our condition seems to be applicable with minimum impact on most compilation environments, so as to prevent the wasting of computational resources caused by the recursive generation.

References

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding parametrized type to Java. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 215-230, 1997.

[APW99] Atsushi Igarashi, Benjamin Pierce, Philip Wadler. Featherweight Java, a minimal core calculus for Java and GJ. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 132-146, 1999.

[BCC98] Kim B. Bruce, Luca Cardelli, and Benjamin C.Pierce. Comparing object encodings. *Information and Comptation 155(1-2)*, 108-133 (1999)

[CS98] Corky Cartwright and Guy Steele. Compatible genericity with run-time types for the Java programming language. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 201-215, 1998.

[DGLM95] M.Day, R.gruber, B.Liskov, and A.C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 156-158, 1995.

[ES90] M.Ellis and B. Stroustroup. The Annotated C++ Reference Manual. Addison-Wesley, 1990.

[MBL97] Andrew C. Meyers, Joseph A. Bank, and Barbara Liskov, Parameterized types for Java. *Symposium on Principles of Programming Languages*, pages 132-145, ACM, 1997.

[Mil85] R. Milner. The Standard ML core language. Polymorphism, 2(2), 1985, 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[OW97] Martin Odersky and Philip Wadler, Pizza into Java: Translating theory into practice. *Symposium on Principles of Programming Languages*, pages 146-159, ACM, 1997.

[OW98] Martin Odersky, Philip Wadler, Gilad Bracha, and David Stoutamire, Making the future safe for the past: Adding Genericity to the Java Programming language. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 183-200, 1998.

[S99] Guy L. Steel, Growing a Language. *Journal of Higher-Order and Symbolic Computation*, 12(3), pages 221-236, Kluwer 1999.

[SA98] Johe H. Solorzano and Suad Alagic, Parametric Polymorphism for Java: A Reflective Solution, *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 216-225, 1998.

[Ull94] J.D. Ullman. Elements of ML programming. Prentice Hall, 1994.

[US80] US Dept. of Defence. Reference Manual for the Ada Programming Language. GPO 008-000-00354-8, 1980.

[VN00] Mirko Viroli, Antonio Natali, Parametric Polymorphism in Java: An Approach to Translation Based on Reflective Features. To appear in *Conference on Object-Orientd Programmin, Systems, Languages and Applications*, 2000.