Type Oriented Logic Meta Programming for Java

Kris De Volder

kdvolder@vub.ac.be
Vrije Universiteit Brussel
Programming Technology Lab.
Pleinlaan 2, B-1050 Brussels, Belgium

April 29, 1998

Abstract

This paper presents *Type-Oriented Logic Meta Programming.* The idea is based on a representation of programs as sets of logic propositions focusing on the type properties of the base-language program. This representation allows generation of base-level code from compile-time logic meta programs that manipulate code fragments and their type properties. We apply this idea to Java and present TyRuBa, a Type-Oriented Logic Meta Programming system for Java. We illustrate that TyRuBa subsumes existing proposals for adding parametric and bounded parametric polymorphism to Java and surpasses them in expressive power.

Keywords: Java, Code Generation, Logic Programming, Types, Meta Programming.
Word count: 7960

1 Introduction

Current OO languages and systems use type information mainly in a passive way to increase the robustness and readability of code. Types mainly serve as a kind of enforceable documentation. This is useful in itself but there still is unused and largely unexplored potential in a more active usage of type information. By "active" we mean that types are treated as values that can be acted upon by programs. Adding parametric and bounded parametric types in OO languages [8, 14, 1, 13, 16, 12, 4] is definitely a step towards a more active usage of types. A parametric type is like a compile time function that takes some types as arguments and constructs a new type from them. So, in a way, parametric type systems offer a limited functional programming language to manipulate types at compile time. Parametric type systems can also be found in functional languages such as Haskell [9]. *Type classes* in Haskell have proven to be a powerful code structuring tool. Proof of this are the modular interpreter frameworks [11] for which the Haskell type system was the cornerstone. It is no exaggeration that it would have been virtually impossible to construct these frameworks without the type system which allows the code to be structured elegantly around a very well chosen set of abstract types.

In all of the systems mentioned above, strong restrictions are imposed by the "programming language" that is available for manipulating types at compile time. The reason for these restrictions is mainly to ensure termination of the compiler and type-checker. These restrictions can be a great source of frustration because the type language "tastes" like a programming language but often lacks the expressiveness to say what you want. Therefore it is an interesting idea to remove the restrictions, stop worrying about termination and nice theoretical properties for a while and just put in an expressive (Turing equivalent) language instead. This might be a controversial idea since it implies that the compiler and type checker are no longer guaranteed to terminate. This need not be a serious problem though. We consider potential infinite loops to be the result of programming errors in type manipulating meta programs similar to infinite loops in base-level programs caused by for example making a mistake in the end condition of a while loop. Even when you do not agree with this controversial statement, our approach is still useful as an intermediate, exploratory step towards more flexible type systems. Currently the tendency is to go from simple, rigid, not very expressive systems and gradually climb up to more flexible and more sophisticated type systems which allow more and more active usage of types, but without sacrificing the nice properties of the type system. Our approach starts at the other end of the spectrum. Throw away all restrictions and build the most flexible system with the least possible effort. The system will probably be so flexible and ad-hoc that you would not want to use it in a real development environment—too much freedom and too little type checking. We can then try and impose more and more restrictions and add type checking support until the system becomes practically usable again. Presumably both approaches will eventually meet somewhere in the middle.

The rest of this paper is structured as follows. We start in section 2 by giving some examples that illustrate there is a lack of expressiveness in current proposals for parametric polymorphism in Java. Section 3 presents the concept of Type-Oriented Logic Meta Programming, a drastic way to introduce a full fledged logic programming language for manipulating types at compile time. We apply this idea to Java and present our system, TyRuBa, as a concrete example. Section 4 discusses how TyRuBa subsumes existing parametric type proposals for Java and section 5 discusses how TyRuBa circumvents the problems in the examples from section 2. Section 6 gives an example that makes full use of TyRuBa as a Turing equivalent type manipulating meta language. We give some pointers about related work in section 8. Finally we summarize and conclude in section 9 and give some ideas about future research in section 10.

2 Motivating Examples

To give the reader a taste of our "uneasy feeling" that existing parametric type systems lack expressiveness, this section presents some short and intuitive examples that illustrate it. We will come back to these examples in section 5 and discuss how our system can conveniently express what we want for all of them.

2.1 Conditional Implementation

It sometimes happens that the implementation of a certain method or interface on a class invokes methods in one of its type parameters. In most parametric type proposals it is possible to signify this by constraining the type parameter to be a subtype of a type that provides the needed functionality. This is called bounded parametric polymorphism [8, 14, 1, 13]. In all of these we have an all-or-nothing situation. Either the type parameter is a subtype of the required type bound in which case the parametric class can be instantiated, or it is not and the class cannot be instantiated. Often however only part of the implementation of the class depends on the type bound and it would be meaningful to be able to instantiate the parametric class albeit with a smaller interface. An example of this is a parametric array implementation as given in figure 1 in a syntax similar to that of [14] and [13]. Note that type parameters are given between "`<>`" and that type variables start with a "`?`".

The example in figure 1 still deserves some explanation. First note the `Equality` interface. This interface has a type parameter `?This` which is supposed to be filled in with the type of the class upon which it is implemented. Using a type parameter in this way is a programming trick to deal with binary methods and simulate F-Bounded polymorphism [5].

Now let us come back to what we really wanted to illustrate with this example. The `Array` implementation as presented has a type parameter for its elements that is restricted to classes implementing `Equality` in order to be able to implement `Searchable`. However suppose we wish to use an array simply as storage structure somewhere and we do not use the `Searchable` interface. In this case we want to be able to instantiate it with any class as element type even a class not implementing `Equality`. This is impossible because the type language in a way has a too restricted form of `if` construct built into it. It allows imposing and verifying type restrictions but has no way of specifying alternatives when the check fails.

Another example of the same problem is a class whose implementation varies depending on a type parameter. Consider a `Dictionary<?Key,?Val>` class. Typically this imposes the type constraint that `Key` should be `Hashable`. However it makes perfect sense to allow dictionaries whose keys are not hashable but merely provide an `equal` method (i.e. implement `Equality`). In this case we just provide a different implementation which uses a linked list as internal storage rather than a hash-table.

It could be argued that the examples given above can be expressed by splitting up the differing functionality into several classes linked through inheritance. This solution is not satisfactory because it tangles up the class tree and becomes unmanageable with multiple dependencies in a

```
interface Equality<?This> {
  boolean equal(?This e);
}

interface Searchable<?El> {
  boolean contains(?El e);
}

class Array<?El implements Equality<?El> >
implements Searchable<?El>, ...
{
  ?El[] contents;

  /** Construction */
  Array<?El>(int sz) { contents = new ?El[sz]; }

  /** Basic Array functionality */
  ?El at(int i) { return contents[i]; }
  void atPut(int i, ?El e) { contents[i]=e; }
  int length() { return contents.length; }

  /** Searchable Interface */
  boolean contains(?El e) {
    boolean found = false;
    int i = 0;
    while (!found && i<length())
      found = e.equal(at(i++));
    return found;
  }

  /** Other interfaces */
  ...
}
```

Figure 1: A Parametric Array Class

single class. In that case it would require some kind of multiple inheritance or an explosion of subclasses for all possible extensions and variations of the base class.

2.2 Positioning Abstract Code

Abstract code often depends only on interfaces and purposefully ignores the implementation details of a specific class. Hence it should be possible to write abstract code independent of the class hierarchy. In most class based languages this is impossible because method implementations are associated with a specific class. Therefore, abstract code usually ends up in an abstract class which depends on subclasses to fill in the implementation details. Consider the abstract class in figure 2 implementing the `Searchable` interface for collections that provide a way of enumerating their elements. The problem with this abstract class is that it is unclear where to insert it into the class tree. The problem becomes exponentially worse when there are several abstract interface implementations.

```
interface Enumerable<?El> {
  Enumeration<?El> elements();
}
interface Enumeration<?El> {
  boolean hasMoreElements();
  ?El next();
}
abstract class Searchable< ?El implements Equality<?El> >
implements Enumerable<?El>
{ boolean contains(?El e) {
    boolean found = false;
    Enumeration<?El> elems = this.elements();
    while (!found && (elems.hasMoreElements()))
      found = e.equal(elems.next());
    return found;
  }
}
```

Figure 2: Abstract Class Implementation of `Searchable`

A better, more "Type Oriented" flavored solution can be accomplished in some parametric type systems. A *mixin* can be simulated as a parametric class inheriting from one of its parameters[1]. Figure 3 shows the implementation of the searchable interface using this technique. A "mixin class" is like a function which can create a subclass implementing the `Searchable` interface for any class meeting the required type constraint. The solution with a mixin is not always ideal either. Sometimes we want to affect the appropriate classes directly rather than through subclassing. In this case mixin classes will not work. Another problem with mixin classes is that

[1]Some parametric type systems do not allow inheriting from type parameters because of implementation-related restrictions.

the mixin's implementation should sometimes be dependent on the base class to provide a more efficient implementation in particular cases. As an example consider a collection which stores its elements in a hash-table. It would be more efficient to implement the `SearchableMixin` by hashing rather than by enumeration. We could try to accomplish this by implementing another mixin, `HashtableSearchableMixin`, specifically for hash-tables but this would be annoying since the user of our collection library must then be told he should use this other mixin for the specific case of a hash-table. Ideally we would like the mixin `SearchableMixin` to be smart enough to decide for itself which version it should use. Regretfully the functional type language implicitly present in current parametric type systems lacks the needed expressiveness.

```
class SearchableMixin<?Super implements
                        Enumerable<?El implements Equality<?El> >
                     >
extends ?Super
implements Searchable<?El>
{ boolean contains(?El e) { ... } }
```

Figure 3: Mixin implementation of `Searchable`

3 TyRuBa

We will now present TyRuBa. TyRuBa is a concrete system offering Type-Oriented Logic Meta Programming for Java. In the next subsections we explain the underlying principle of TyRuBa: Logic Meta Programming, a drastic way to introduce a full fledged logic programming language for manipulating types at compile time. Subsection 3.1 explains the idea independent of the chosen base language. Subsection 3.2 shows how this idea can be applied to Java.

There are several reasons for choosing Java as the base language. One of them of course is that Java is very popular at the moment. However, this is not the only reason. Java is also fairly simple and fairly well designed. Further, it has a static type system, and last but not least, Java interfaces are natural candidates for describing object types. TyRuBa programs talk mainly about interfaces and classes and about how the former are implemented by the latter.

The meta language for manipulating types at compile time is a Prolog-like [6] logic language. The reason for choosing a logic language is that we believe it offers the right kind of expressiveness for type manipulation. We can give some indications that this true. One such indication is the Haskell Type System[9]. The way type classes can be used in Haskell has a very strong resemblance to writing logic Horn clauses. Haskell's type system is a good guideline as it is one of the most mature, sophisticated, flexible and expressive type systems around. Another indication can be found in the way type theorists formulate the semantics of a type system: as a set of inference

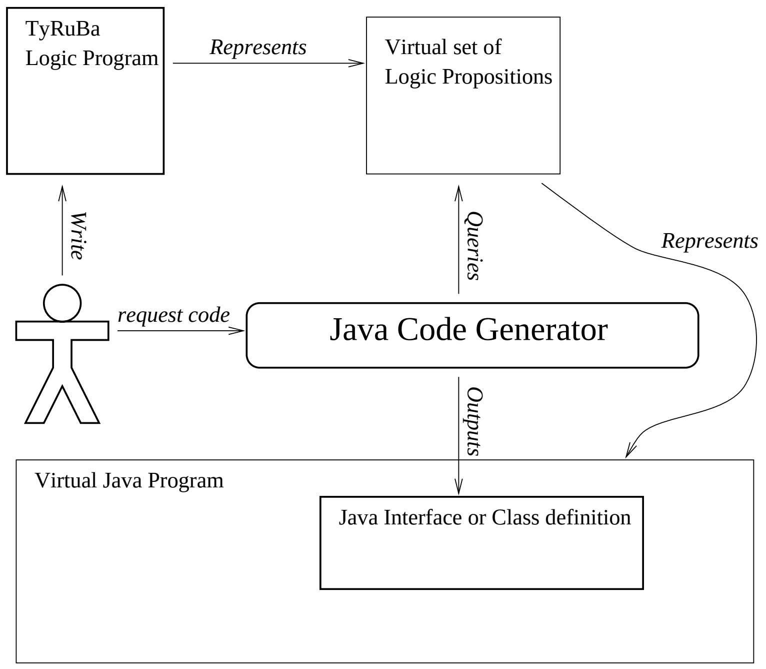

Figure 4: Overview of a Logic Meta System for Java

rules with conditions and conclusions [15].

Essentially TyRuBa programs are logic programs in Prolog-like syntax with the extra possibility to include fragments of Java code as logic terms. Except for syntax analysis, these fragments are mainly treated as text and no type checking is performed on them. This is clearly not what one would put into a real development environment. A real development environment should treat the pieces of Java code as what they are, fully ensuring type correctness. Compare this with the relation between C++ and Pizza [14]. C++ templates[16] can regarded as an ad-hoc, quick and dirty version of parametric polymorphism whereas Pizza is a clean and nicely type checked version of it. For now we are mainly interested in performing experiments and maximizing expressive power. But in the future a more mature, clean variant with nice formal underpinnings and a form of type checking should be developed. Probably this will require compromises with respect to the expressive power of the type-manipulating meta language.

3.1 Logic Meta Programming

The central concept around which everything revolves is a representation scheme which maps a base-language program onto a set of logic propositions which represents it. The possibilities opened up by such a representation are immense. A logic programming language is just an expressive way to denote sets of logic propositions so we can now use the full power of a logic programming language to describe the structure of base-language programs indirectly represented as logic propositions. Figure 4 shows a schematic view of a logic meta programming system. The system depicted here is one where Java is the base language, but other base languages are also possible with a roughly identical schematic layout. Potentially, the logic program describes an infinite set of propositions and hence a potential infinitely large base-language program. Evidently, it's impossible to compile an infinitely large program. Therefore only specific parts of the program will be extracted and compiled as needed. The responsibility for deciding which parts are needed lies with the user outside of the TyRuBa system. In case of Java the "user" explicitly asks for the classes or interfaces he needs. All of the information necessary to generate the source code is present somehow in the logic program. So the code generator formulates some queries to the logic system and finds out what it needs to print out the requested class or interface definition. Note that the term "user" here can be interpreted very broadly. The user might for example be a Java compiler that request the source code for a particular class or interface whenever it encounters a reference to it while compiling another class or interface.

3.2 Type-Oriented Logic Meta Programming for Java

3.2.1 The Mapping

In order to understand TyRuBa, we must know the mapping between Java programs and their representation as sets of logic propositions. As we are interested in types and the dependencies between them, the propositions focus on classes, interfaces and the relations between them. The basic idea of the representation scheme is that a class body is chopped up into pieces providing the implementations for interfaces[2]. Every one of these pieces is put into a proposition of the form:

`implements(`*`className`*`,`*`interfaceName`*`,{...}).`

The remaining code in the class body is put into a proposition:

`class(`*`className`*`,{...}).`

A schematic Java program and the corresponding set of logic propositions are given in figure 5. Note that the propositions signifying relationships of interface extensions and class extensions have been given distinct names (`extends` and `inherits` respectively) to avoid ambiguity.

3.2.2 TyRuBa Syntax and Programs

We can now write logic programs which represent Java programs. The syntax of TyRuBa used in the following examples is a simple variant of the logic language Prolog. Logic variables in TyRuBa

[2]We implicitly assume that interfaces do not overlap. This is not a serious restriction because it is always possible to create an extra interface for the shared part of two overlapping interfaces and add this new interface to their extends clause.

```
interface interface1
extends superInterface1,superInterface2
{ ... }

interface interface2 { ... }

class aClass extends aSuperclass
implements interface1, interface2 {
  /** Some code specific to aClass */
  ...
  /** Some code implementing interface1 */
  ...
  /** Some code implementing interface2 */
  ...
}
```

```
interface(interface1,{...}).
extends(interface1,superInterface1).
extends(interface1,superInterface2).

interface(interface2,{...}).

class(aClass,{/** Some code specific to aClass*/ ...}).
inherits(aClass,aSuperClass).
implements(aClass,interface1,{
  /** Some code implementing interface1 */
  ...}).
implements(aClass,interface2,{
  /** Some code implementing interface2 */
  ...}).
```

Figure 5: A Java Program (on top) and its representation as a set of propositions

```
Program      →  Rule*
Rule         →  Predicate [ ":-" Conjunction ] "."
Conjunction  →  Predicate ( "," Predicate )*
Predicate    →  Identifier
             |  "?"Identifier
             |  Term "(" TermList ")"
Term         →  Identifier
             |  "?"Identifier
             |  Term "<" TermList ">"
             |  "{" JavaWithTerms "}"
TermList     →  Term ( "," Term )*
```

Figure 6: The essence of TyRuBa's syntax

start with a "?" rather than with a capital letter as in Prolog. Thus confusion between Java identifiers starting with capital letters and logic variables is avoided. A TyRuBa logic program may contain fragments of Java code surrounded by "{" and "}". These pieces of code are treated as special compound terms. Any place inside the "{}" where Java syntax allows an identifier to occur one may put a TyRuBa term. The essence of TyRuBa's syntax is given in figure 6. Note that compound terms are written with "<>" instead of "()" to avoid confusion with function or procedure calls in Java.

3.2.3 Generating Java Code

A few technical complications occur in generating Java code from a TyRuBa logic program. One complication is the usage of terms in place of Java identifiers. Terms have a recursive structure (i.e. they may contain other terms) whereas Java identifiers are just flat string like entities. However, it is not too difficult to define a name hashing scheme that associates a valid Java identifier with a term. Currently the hashing scheme does not handle unbound logic variables in the terms. We implicitly assume that no logic variables will appear in the generated Java code and if they do this results in an error. A better solution could be to treat unbound variables as being bound to the type name `Object` since this is the most general type. However this would still leave problems when the variable appears in a position where an interface is expected [3].

Another technical decision involves the handling of conflicting information in the logic database. For example, what to do when multiple definitions of a class are derived from the meta program. One option would be to signal an error. Instead we have chosen to simply take the first class definition returned by the logic evaluator and ignore the rest. The logic system considers rules in the reverse order of how they appear in the source file. So this allows for a kind of overriding of class definitions. For example we can put a more general implementation of a class at the beginning of the file and a more specific, more efficient one near the end. Figure 7 gives an example of a

[3]Java does not provide a most general interface similar to the most general class `Object`

double declaration of a `Dictionary` class. One is a general implementation of `Dictionary` using only the `Equality` interface on its keys whereas a more efficient implementation is chosen when keys are `Hashable`. Since the more efficient implementation appears later in the source code it will take precedence in case both are applicable. Note that this example is the first one which uses real logic rules with a condition constraining their applicability.

```
/** General Dictionary implementation with a Linked list */
class(Dictionary<?Key,?El>,{
LinkedStorage<Association<?Key,?El> > contents;
  ...
})
:- implements(?Key,Equality<?Key>).

/** More efficient implementation with a hash table */
class(Dictionary<?Key,?El>,{
  HashTable<?Key,?El> contents;
  ...
})
:- implements(?Key,Hashable).
```

Figure 7: Two alternative implementations of Dictionary

A conflict also arises when the logic program implies more than one implementation of an interface onto a single class. Once more we chose the solution to simply take the first one returned by the evaluator. This again allows for a kind of overriding when putting more general implementations at the beginning of the file and more specific ones towards the end.

4 Parametric and Bounded Parametric Types

This section gives an example illustrating how the ability to express parametric types is a straightforward result of using logic terms as type names. Reconsider the parametric `Array` class from figure 1. This example can be "translated" into TyRuBa as shown in figure 8.

Such a definition represents not one but several classes: one for each binding of the variables in the class-name. In the given example, that is one `Array<?El>` class for each possible element type. This example also shows how bounded polymorphism is possible by imposing type restrictions on the variables in the condition of the rule defining the class. Note that if the class implements interfaces or extends another class the assertions stating these facts should also be guarded by the same condition.

The `subtype` predicate is not defined by the representational mapping between Java and TyRuBa. It corresponds to the Java subtype relationship between classes and interfaces and is defined in TyRuBa itself by a set of rules displayed in figure 9. These rules are included automatically into the rule base upon initialization.

```
interface(Equality<?This>,{
  boolean equal(?This e);
}).

interface(Searchable<?El>,{
  boolean contains(?El e);
}).

class(Array<?El>,{
  ?El[] contents;

  /** Construction */
  Array<?El>(int sz) { contents = new ?El[sz]; }

  /** Basic Array functionality */
  ?El at(int i) { return contents[i]; }
  void atPut(int i, ?El e) { contents[i]=e; }
  int length() { return contents.length; }
})
:- subtype(?El,Equality<?El>).

implements(Array<?El>,Searchable<?El>,{
  boolean contains(?El e) { ... }
})
:- subtype(?El,Equality<?El>).

implements(Array<?El>,OtherInterface,{ ... })
:- subtype(?El,Equality<?El>).
```

Figure 8: A parametric `Array` class in TyRuBa

```
subtype(?X,?X) :- class(?X,?body).
subtype(?X,?X) :- interface(?X,?body).
subtype1(?X,?Y) :- extends(?X,?Y).
subtype1(?X,?Y) :- inherits(?X,?Y).
subtype1(?X,?Y) :- implements(?X,?Y,?body).
subtype(?X,?Y) :- subtype1(?Z,?Y),subtype(?X,?Z).
```

Figure 9: Default TyRuBa rules implementing the subtype relationship

With respect to the existing proposals for parametric types in Java [14, 1, 13], this emulation of bounded parametric polymorphism resembles most that from Pizza [14] and [1]. The given implementation of `subtype` corresponds to the type restrictions with which type parameters in these two systems can be bounded. Pizza [14] and [1] are very similar to each other but because of its heterogeneous implementation, [1] is somewhat more flexible. The terminology "heterogeneous" and "homogeneous" was introduced in [14]. A heterogeneous implementation reinstantiates the code of a parametric class every time it is used whereas a homogeneous implementation shares the code. In order to be able to share the code between all instantiations of a parametric class more restrictions must be imposed. TyRuBa fits into the "heterogeneous" category since it generates separate Java source code for every instantiation of a class. The system of [1] is heterogeneous but avoids recompilation of a class for every instantiation by delaying instantiating to the moment the compiled Java class file is loaded. Hence instantiation happens at load time rather than at compile time. Because TyRuBa instantiates Java source code at compile time it is not as efficient. However, this also makes TyRuBa somewhat more flexible than [1]. For example, TyRuBa has no problem using primitive types as type parameters and is possible to express all of the examples from [1]. This includes an example with a mixin such as was given in figure 3.

	TyRuBa	[1]	Pizza
Instantiate	Java source	.class file	No instantiation: shared code
Mixin Possible	Yes	Yes	No
Primitive types as parameters	Yes	No	No

Figure 10: Overview of instantiation of parametric classes

5 Motivating Examples Revisited

Because TyRuBa is a real albeit simple programming language to manipulate pieces of Java code together with their type properties, the possibilities are virtually unlimited. You can write logic programs to implement classes, define interfaces, implement interfaces on classes etc. These programs may infer different implementations or interfaces depending on a type parameter. The extra expressive power offered by TyRuBa is more than sufficient to alleviate the problems from section 2 by simple and straightforward usage of logic rules.

5.1 Conditional Implementation

In section 2.1 we discussed that we sometimes are not satisfied with simply putting type constraints on an entire class but want finer control on an interface per interface basis. As an example we presented a parametric `Array` class (figure 1) which implements the `Searchable` interface. In figure 8 we showed how the same `Array` example is expressed in TyRuBa. It is easy to see in this example that in TyRuBa the type constraint is mentioned separately for the base functionality of the class and for each interface implementation. Thus it is easy to control on an interface per interface basis which type constraints are required for its implementation. There is no need to impose the type constraint on the the class as a whole. In a similar way it is possible to provide several rules implementing the same interface but with different type constraints imposed by its condition. It is also possible to do the same with class declarations: declare two alternative rules giving different implementations for a class with different type constraints imposed on the parameters. We already gave an example (figure 7) of this that declares two alternative implementations for a `Dictionary` depending on the type of its keys.

5.2 Positioning Abstract Code

In section 2.2 we discussed that it is not always adequate to put abstract code into abstract classes. The reason is that is not always easy or possible to find a suitable place in the class tree for the abstract class. The problem becomes exponentially worse when several abstract classes are to coexist in the same class library. A nicer solution was using a mixin class but this requires explicit subclassing to add the abstract functionality to the class and makes it difficult to specialize the abstract functionality in specific classes. In TyRuBa it is very natural to write an abstract implementation for an interface completely separate from the class hierarchy and declare by means of an arbitrary logic expression to which classes it should be applied. Figure 11 shows a logic rule that implements the searchable interface on any collection class that implements the `Enumerable` interface and has elements comparable for `Equality`. Note that we can still provide a more efficient implementation for more specific cases as we explained in section 3.2.3.

```
implements(?any, Searchable<?El>, {
  boolean contains(?El e) {
    boolean found = false;
    Enumeration<?El> elems = this.elements();
    while (!found && (elems.hasMoreElements()))
      found = e.equal(elems.next());
    return found;
  }
})
:- implements(?any,Enumerable<?El>),implements(?El, Equality<?El>).
```

Figure 11: An Abstract implementation of `Searchable` in TyRuBa

6 Fully Exploiting Turing Equivalence

TyRuBa is a fully Turing Equivalent meta language for manipulating pieces of Java code together with their type properties. This allows for sophisticated forms of generic programming where code for a certain class or interface can be generated by an arbitrary computation specified in Prolog. This section gives an example illustrating this. The example is a parametric class for a multidimensional array data structure where the dimensionality of the array is a parameter of the class. The example is interesting because it recurses over an integer representing the dimensionality of the array. An iteration like this leads to potential infinite loops when there is no adequate condition that makes it terminate after a finite number of steps. Hence this example could not be expressed in a system which guarantees that "compile time type programs" always terminate.

We start with the most trivial case which ends the recursion, namely an array with dimensionality equal to zero, in figure 12.

```
class(MArray<0,?El>, {
  private ?El contents;
  ?El elementAt() { return contents; };
  void setElementAt(?El el) { contents = el; }
}).
```

Figure 12: A zero-dimensional array

Before giving the recursive class that specifies how an array with n dimensions can be specified in terms of an array of $n-1$ dimensions, let us first have a look at how the specific example of a three-dimensional array can be implemented in terms of a two-dimensional array in figure 13. This will help to understand the recursive TyRuBa code we will give later on.

As can be seen, the implementation of a three-dimensional array in terms of a two dimensional one is pretty much as expected. It stores the array as a Java array of two dimensional arrays. Its constructor and accessor methods have an extra parameter to use for the Java array and pass the rest of the arguments on to the respective two dimensional constructor or method.

In order to express the above in a generic way depending on a variable representing the dimensionality of the array we will need some features of TyRuBa we have not yet explained. First of all, we said that TyRuBa programs may contain Java code between "{}". We silently ignored a few technical problems with this in the parser. Simply putting any kind of Java code in "{}" anywhere in a TyRuBa program is not possible because the parser has no way of guessing what to expect. That is, it cannot know whether to parse the body of a method, an argument list, or a class body etc. Therefore this simple form is only allowed in places where the parser knows what to expect from the name of the predicate or term in which it occurs (e.g. `class`, `interface`,

```
class(MArray<3,?El>, {
  private MArray<2,?El>[] contents;

  MArray<3,?El> (int size3, int size2, int size1) {
     contents = new MArray<2,?El> [size3];
     for (int i=0;i<size3;i++) {
         contents[i] = new MArray<2,?El> (size2,size1);
     }
  }

  ?El elementAt (int index3,int index2,int index1) {
     return contents[index3].at(index2,index1);
  }

  void setElementAt (?El el,int index3,int index2,int index1)
  {
     contents[index3].setElementAt(el, index2, index1);
  }
}
```

Figure 13: A three-dimensional array

`implements`). In other places an explicit prefix must be prepended to the "`{`" to indicate the kind of Java code that will be found inside. Including prolog terms as parts of Java code, is also somewhat more complicated than what we already explained. In places where the parser expects an identifier, a Prolog term may be inserted without any special prefix or indication. Prolog terms may also be inserted at any other arbitrary point in the Java program, by prepending them with a "`@`". What follows should be a prolog term of the form `Java`*`Name`*`(...)` where *`Name`* should be a valid name for a terminal or non terminal in the Java abstract grammar.

We now know enough to understand the remaining (recursive) part of the implementation of arrays which is given in figure 14. The condition of the rule restricts it to be applicable only when the dimensionality of the array is one or greater. This restriction is responsible for the recursion ending when zero is reached. The condition of the rule also computes bindings for the variables `?CFormals` and `?atFormals` by means of an auxiliary predicate `formals`. The `removeTypes` predicate is another auxiliary used to compute a list of actuals to be passed on to the sub-arrays.

7 The Collection Experiment

As an experiment to explore the expressiveness of TyRuBa we have implemented a small collection library based on the Smalltalk-80 collection classes [7]. An important conclusion from the collection experiment is that the granularity of interfaces alone turns out to be too large in many cases. Often we want to be able to reason on a method per method basis. The result of this is that many interfaces with only one method appear in the collection library. This makes the library

```
class(MArray<?Dim,?El>, {
  private MArray<?DDim,?El>[] contents;

  MArray<?Dim,?El> @JavaFormals(?CFormals) {
     contents = new MArray<?DDim,?El>[?CFirst];
     for (int i=0;i<?CFirst;i++) {
         contents[i] = new MArray<?DDim,?El> @JavaActuals(?CRest)
     }
  }

  ?El elementAt @JavaFormals(?atFormals) {
     return contents[?atFirst].at @JavaActuals(?atRest);
  }

  void setElementAt @JavaFormals( [ JavaFormal{ ?El el }
                                  | ?atFormals ] )
  {
     contents[?atFirst].setElementAt @JavaActuals( [el | ?atRest] );
  }
})
:- ?Dim>0, ?DDim=?Dim-1,
   formals(?Dim, int, size, ?CFormals),
   removeTypes(?CFormals,[?CFirst | ?CRest]),
   formals(?Dim, int, index, ?atFormals),
   removeTypes(?atFormals, [?atFirst | ?atRest]).

// Example: formals(2,int,size,
//                 [JavaFormal{int size<2>},JavaFormal{int size<1>} ])
formals(0, ?Type, ?Name, []).
formals(?Dim, ?Type, ?Name,
  [ JavaFormal{?Type ?Name<?Dim>} | ?RestFormals ]
) :- ?Dim>0, ?DDim=?Dim-1,
     formals(?DDim, ?Type, ?Name, ?RestFormals).

// Example: removeTypes([JavaFormal{int size<2>},JavaFormal{int size<1>}],
//                      [               size<2> ,               size<1> ])
removeTypes([],[]).
removeTypes([JavaFormal{?T ?I} | ?RT], [?I | ?R]) :- removeTypes(?RT,?R).
```

Figure 14: A multidimensional array class in TyRuBa

more difficult to understand as there are too many small interfaces with obscure names. The solution to this problem is a refinement of the representational mapping from section 3.2.1 to include information about individual members in classes and interfaces.

Despite of this granularity problem, we think that our experiment shows that Type-Oriented Logic Meta Programming has potential. It was possible to capture more dependencies between types than is described in [7] or then would be possible with parametric types. Especially dependencies between collection types and the elements of the collections. Examples of this are all the interfaces that are implemented only when elements can be compared for equality such as the implementation of the `Searchable` interface which was used as an example in several places in this paper.

Also, because TyRuBa provides an easy and flexible way to write abstract code independent of the class tree (see section 5.2) the level of abstractness of code is raised. It turns out that one can implement a very large portion of the library in terms of a variation of the `Enumerable` interface which captures the most basic and abstract functionality of collections: a way to enumerate their elements. The implementation derived from this interface has a very abstract look and feel. It can be used for inferring implementations of almost all the functionality of both Arrays and LinkedLists for example. We must confess however that due to the abstractness of the code it is often not tuned to the specific performance characteristics of the underlying storage structure and therefore often results in inefficient implementations. For example indexing an array by enumerating its elements until the nth element is reached. These inefficient default implementations can however be overridden by versions with better performance when needed. In this case the abstract implementation can still be regarded as a kind of operational specification or documentation.

8 Related Work

We have already discussed the relationship of TyRuBa with the proposals [14, 1] but we have yet to discuss one other Java parametric types proposal [13]. It offers `where` clauses to constrain type parameters. In the current version of TyRuBa it is not possible to simulate `where` clauses because the granularity of the type information in TyRuBa is too coarse. The only information available for TyRuBa programs are interfaces, classes and the extends and implements relationships between them. In order to simulate where clauses more fine grained information about individual methods must be accessible. It is perfectly feasible to refine the representational mapping from section 3.2.1 in order to include the needed information about individual members. This is a good idea anyway as we mentioned in section 7.

Thorup [17] proposes virtual types as an alternative to parametric types. Bruce [3] discusses that parametric types and virtual types have similar goals, but each solves different kinds of problems better. He therefore proposes a system integrating both. We have not yet investigated in depth how well virtual types can be emulated in TyRuBa. It has been argued [3] that virtual types can be emulated by parametric types but that this leads to very complicated code and is not usable in practice. We feel confident that TyRuBa is powerful enough to conveniently express the examples that work well with virtual types but are hard with parametric types. Typically such examples involve "families" of types that interact. With parametric types it is hard to express a "kinship" relationship between classes. This is responsible for tangling up the code. In TyRuBa we may simply use a proposition stating a kinship between classes as follows `nameOfFamily(className1,className2,...)`. The implementations of the classes in the family may then refer to each other indirectly through a variable that is looked up in the logic database with an appropriate query.

The idea of using logic to describe program structure is not new. This approach was also taken in [10]. This work could be interpreted as a proposal for a formal Type-Oriented Logic Meta Programming system where the propositions representing the base level programs give information about method implementations and their types. It is possible to "assert" a method implementation in a way nearly identical to the way one can assert an implementation of an interface in TyRuBa. In a version of TyRuBa where information about individual methods is made explicit, the resemblance of both systems would be even closer. In [10] the logic meta system is used on a theoretical level and is explicitly hidden from the programmer. In contrast, in our approach exposing the logic foundation of the system towards the programmer is actually a prerequisite. Also, [10] only considers assertions about methods. Classes are more or less treated as atoms. So it is not possible to "assert" implementations of classes. In a way, where our system is too large grained, [10] gives a too fine grained approach. It would thus be meaningful to attempt an integration of both approaches into one system.

9 Conclusion

In this paper we presented the concept Type-Oriented Logic Meta Programming. We also presented TyRuBa, a concrete system for Java. TyRuBa represents a Java program indirectly by means of a database of logic rules and assertions that describe how classes implement interfaces. We illustrated that Type-Oriented Logic Meta Programming subsumes parametric and bounded parametric polymorphism. The emulation of parametric polymorphism in TyRuBa is a straightforward consequence of using logic variables in logic terms representing type names. The kind of constraints that can be used to impose bounds on the type parameters depends on the information that is made explicit in the representation scheme mapping base level programs onto sets of logic propositions. The current version of TyRuBa only makes explicit interfaces classes and the `inherits`, `extends` and `implements` relationships between them. More sophisticated applications would be possible with a representation which also makes information about individual methods and instance variables explicit.

When considering only the issue of parametric types, TyRuBa is more ad-hoc than existing parametric type proposals. TyRuBa does no type checking for its own and simply depends on the Java compiler used to compile the generated code. However, because of its unrestricted (Turing equivalent) language for manipulating pieces of code together with their type properties it far surpasses the expressiveness of mere parametric types. We have given some examples that illustrate how some problems with the expressiveness of existing parametric types proposal are solved easily in TyRuBa. We also presented a more sophisticated example that uses TyRuBa implement a multidimensional array class. This example illustrates the extra expressive power offered by a fully Turing equivalent type manipulation language that allows actually "computing" a class's declaration by means of a compile time meta program.

We briefly reported on the experiment of implementing a small collection library in TyRuBa. This experiment taught us that the approach is promising but that the granularity of interfaces and classes is too coarse. More fine grained information about individual methods and instance variables is needed. This comes down to basing the TyRuBa system on a more fine grained representational mapping than the one presented in section 3.2.1. We don't see any problems in doing this and it could be realized on a reasonably short term basis.

10 Future Work

Currently we have not bothered much with what could be called conflict resolution. Whenever the rule base supplies ambiguous information such as several declarations of one class, or several implementations of one interface onto a single class, we have just taken the most obvious and simple solution: take the first candidate and ignore the rest. This ad-hoc solution can already be used in a meaningful way and even allows for a kind of overriding by depending on the evaluation order of the logic inference engine. However, it would be worthwhile to investigate more structured approaches to conflict resolution. For example a system which allows numerical priorities to be assigned to rules and assertions as in hierarchical constraint solvers [2]. In the context of types it is probably more useful to devise priorities depending on the generality of the types upon which the interface implementation is defined (e.g. depending on the class and interface hierarchy). This means that priorities should be determined symbolically rather than numerically.

Other research tracks could try to narrow the gap between between the drastic ad-hoc implementation of TyRuBa and traditional type checkers with solid theoretical underpinnings. There are mainly two approaches to closing the gap depending which side one starts from. Starting from

TyRuBa we could investigate what part of it can be safely integrated with a type checker that ensures that rules are "type correct". We consider a rule to be type correct in case its condition imposes enough type restrictions to ensure that the pieces of Java code in the conclusion are type correct whenever the rule is applicable. It will probably be impossible to check the more sophisticated usage such as the multi-dimensional array example where part of the Java code is being computed by a program. However it seems probable that simpler cases such as the examples from section 5 could be checked statically before the TyRuBa program is run to produce Java code. It still requires a considerable research effort to determine what kind of restrictions are needed to allow static type checking of rules on the one hand and to devise a type checking algorithm for a mix of logic rules and pieces of base level code on the other hand. If all of this succeeds we could conceive a hybrid system which statically checks TyRuBa rules that confirm to the restrictions but leaves other rules unchecked to allow unlimited expressiveness.

We can also try to close the gap from the other side by examining the typical programming idioms that emerge in TyRuBa programs and suggest statically type checkable language constructs for them that can be integrated into a traditional static type checker for Java. The parametric types proposals [14, 1] are already big steps in this direction. The way some problems with the expressiveness of these systems as illustrated in section 2 is solved by TyRuBa in section 5 can provide valuable clues as to what kind of extensions would be useful for increasing their flexibility to support a more "type-oriented" style of programming.

11 Acknowledgements

I thank all of my colleagues from the Programming Technology Lab for many discussions, brainstorm sessions and suggestions. I especially thank Kim Mens, Tom Mens, Carine Lucas, Patrick Steyaert, Theo D'Hondt, Wolfgang De Meuter, Werner Van Belle, Geert Lathouwers and Tom Tourwé for their meticulous proofreading and constructive criticism.

References

[1] Ole Agesen, Stephen Freund, and John C. Mitchel. Adding type parametrization to java. In *OOPSLA '97 Conference Proceedings*, volume 32(10) of *ACM SIGPLAN Notices*, pages 49–65, 1997.

[2] Alan Borning, Bjorn N. Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[3] Kim B. Bruce. A statically safe alternative to virtual types. Submitted to ECOOP 98.

[4] Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In Walter Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *LNCS*, pages 27–51, Berlin, GER, August 1995. Springer.

[5] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[6] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[7] William R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *OOPSLA '92 Conference Proceedings*, ACM SIGPLAN Notices, pages 1–15, 1992.

[8] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 156–168, 1995.

[9] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[10] John Lamping. Methods as assertions. In Mario Tokoro and Remo Pareschi, editors, *Object-Oriented Programming 8th European Conference, ECOOP '94 Bologna, Italy, Proceedings*, volume 821 of *Lecture Notes in Computer Science*, pages 60–80. Springer-Verlag, New York, N.Y., July 1994.

[11] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343, January 1995.

[12] B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

[13] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for java. In *Proceedings of 24th ACM symposioum on Principles of Programming Languages*, pages 132–145, 1997.

[14] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[15] David A. Schmidt. *The Structure of Typed Programming Languages.* MIT Press, 1994.

[16] Bjarne Stroustrup. Parametrized types for C++. *Computing Systems*, 2(1):55–85, Winter 1989.

[17] Kresten Krab Thorup. Genericity in java with virtual types. In Mehmet Aksit and Satoshi Matsuoka, editors, *Object-Oriented Programming 11th European Conference, ECOOP '97, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer-Verlag, 1997.