Available online at www.sciencedirect.com
ScienceDirect

Computer Languages, Systems & Structures 34 (2008) 1 – 17

COMPUTER LANGUAGES, SYSTEMS & STRUCTURES
www.elsevier.com/locate/cl

# Mechanisms for improved covariant type-checking

Kevin Cleereman[a,*], Michelle Cheatham[a], Krishnaprasad Thirunarayan[b]

[a]Air Force Research Laboratory, Information Directorate, Collaborative Technology Branch, WPAFB, OH 45433-7334, USA
[b]Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435, USA

Received 23 September 2005; accepted 27 March 2006

**Abstract**

Covariant types are a powerful language feature for improving type-safety. However, covariant types complicate type-checking when combined with polymorphism. We propose two new language features that can improve type-checking in a language with polymorphism and covariant typing, and also have the potential to improve efficiency as well.
Published by Elsevier Ltd.

*Keywords:* Type checking; Covariance; Object-oriented programming languages

## 1. Introduction

Frequently when designing a system, it is desirable for a method that is being overridden in a subclass to specialize the arguments given in the superclass. For example, in the class Employee we have the method process(File f), but in the Employee subclasses Engineer and HR (Human Resources) we would like to specialize the method arguments to process(Design d) and process(Personnel p), where Design and Personnel are both subclasses of File. The goal of such a system design is to handle all Employees in a uniform way while still guaranteeing type-safety. For instance, an Administrator class may contain a method process_All(List⟨File⟩), which causes each Employee to process all Files of the applicable type while ignoring inapplicable File objects. This situation occurs frequently enough in practice that several design patterns (e.g., the Visitor pattern) have been created to enable this system design.

Design patterns can be created to address shortfalls in programming languages [1], and this case is no exception. Most common programming languages will not support the above design because they lack covariant types. Covariant type relations [2] permit subclasses to specialize the formal parameters of an inherited method, e.g., specializing the process(File f) method from the Employee class into the process(Design d) method in the Engineer class, where Design is a subclass of File. However, covariant types complicate type-checking, and so many languages enforce non-variant type relations (such as C++ [3] and Java [4]), while statically typed languages that permit covariant type relations (such as Eiffel [5,6]) either restrict polymorphism or else require potentially costly whole program analyses to guarantee system validity. But without covariant types, the process_All method is not type-safe—there is nothing in the language's type system that will prevent the Engineer.process method from being called with a Personnel argument. The following

* Corresponding author. Tel.: +1 937 904 9090.
*E-mail address:* kevin.cleereman@wpafb.af.mil (K. Cleereman).

1477-8424/$ - see front matter Published by Elsevier Ltd.
doi:10.1016/j.cl.2006.03.001

section will discuss various approaches to solving this problem and their shortfalls. Next we will introduce an OnlyWith operator that will address many of the shortfalls of the standard solutions to the problem. We then propose a new solution to this problem that is intended to both enable this system design at the programming language level and to also improve speed and memory efficiency. Experimental results from a Java simulation follow, and we end with conclusions and ideas for future research.

```
Abstract Class File { }

Abstract Class Employee {
  void process(File f) { ... }
}

Class Design extends File { ... }

Class Personnel extends File { ... }

Class Engineer extends Employee {
  void process(Design d) { ... }
}

Class HR extends Employee {
  void process(Personnel p) { ... }
}

Class Administrator {
  void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
    foreach(emp in e) {
      foreach(file in f) {
        emp.process(file);
      }
    }
  }
}
```

## 2. Standard approaches

Assertions can be added to insure that the process methods in the Employee subclasses are only executed on File objects of the appropriate subclass. The Administrator.process_All method would then need to be rewritten as shown to catch and ignore any failed assertions. The problem with this solution is that the method signatures are now inaccurate—the client of an HR object expects the process method to accept any File object (as the signature attests), not just a Personnel object (as the assertion attests). Thus, this technique relies on the author of the Administrator class being aware of the fact that the authors of Employee subclasses may write unsafe pseudo-covariant methods—the type-checker will in general not be able to flag possible type violations without requiring an exception handler for every assertion or downcast.

This technique also suffers from slow execution speed. It relies on the language's exception-handling mechanism, along with all of the necessary support machinery the mechanism entails. This cost can be somewhat alleviated in languages that include reflective mechanisms (e.g., Java) or conditional downcasts (e.g., Eiffel) by reducing the number of failed downcasts, but then we necessarily forfeit the generality of the exception-handler by requiring the inclusion of a switch statement to test all legal downcasts.

```
Class Engineer extends Employee {
  void process(File f) {
    assert(f.instanceOf(Design)); ...
  }
}
```

```
Class HR extends Employee {
  void process(File f) {
    assert(f.instanceOf(Personnel)); ...
  }
}

Class Administrator {
  void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
    foreach(emp in e) {
      foreach(file in f) {
        try {
          emp.process(file);
        } catch (TypeException e) { }
      }
    }
  }
}
```

Alternatively, we can remove the process(File f) method from the Employee class. However, by removing the abstraction from the superclass, we have made it significantly more difficult for the Administrator to process all files. The author of the Administrator class must now be aware of all dependencies between the File and Employee methods. Rather than using a single exception handler for all possible type combinations, we must now include a new exception handler for each new type combination. For example, if the access privileges of the Engineer class were modified to permit the reading of additional file types, then the process_All method would need to be rewritten to reflect this. Likewise, if a new Employee subclass were added to the system, then the new class's access privileges would need to be coded in the process_All method. In summary, modularity is adversely affected in this approach.

```
Class Administrator {
  void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
    foreach(emp in e) {
      foreach(file in f) {
        try {
          Engineer eng := (Engineer) emp;
          Design des := (Design) file;
          eng.process(des);
        } catch (ClassCastException e) { }
        try {
          HR hr := (HR) emp;
          Personnel per := (Personnel) file;
          hr.process(per);
        } catch (ClassCastException e) { }
      }
    }
  }
}
```

It is possible in object-oriented languages with reflective mechanisms to design a system in which individual classes can be modified without necessitating a rewrite of the Administrator class. For example, we can define a set of Class_Pair objects that form a set of all legal method calls.

```
Class_Pair {
  Class ⟨Class⟩ first;
  Class ⟨Class⟩ second;
```

```
    Class_Pair next = null;

    Class_Pair(Class ⟨Class⟩ x, Class ⟨Class⟩ y) {
      first := x;
      second := y;
    }
  }

  Class Static_Process_Pairs {
    Static Class_Pair head = null;

    Static add(Class_Pair x) {
      x.next := head;
      head := x;
    }
  }
```

Now the Administrator class need not be rewritten when the class hierarchy or access privileges change in the Employee and File classes.

```
  Class Administrator {
    void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
      foreach(emp in e) {
        foreach(file in f) {
          Class_Pair pair := Static_Process_Pairs head;
          while(pair / = null) {
            try {
              ⟨pair.first⟩ empx := (pair.first) emp;
              ⟨pair.second⟩ filex := (pair.second) file;
              empx.process(filex);
              pair := null;
            } catch (ClassCastException e) {pair := pair.next;}
          }
        }
      }
    }
  }
```

The author of an Employee or File class registers all legal first.process(second) method calls in the Static_Process_Pairs class, and the Administrator then loops through the legal type combinations until a legal combination is found (assuming there is a legal combination).

Unfortunately, this technique suffers from two major drawbacks. The first drawback is in the efficiency. The process_All method with the simple try-catch block runs in $O(E * F)$ time, where $E$ is the number of all Employee objects and $F$ is the number of all File objects. However, the process_All method using the Class_Pair technique runs in $O(E * F * T)$ time, where $T$ is the number of valid type combinations in the Static_Process_Pairs list. Further note that all but the final invocation of the inner (while (pair / = null)) loop will require the additional overhead of the exception handling mechanism, although this is only necessarily the case for a language that does not support reflection or conditional downcasts. The second drawback is that the technique is still not statically type-safe. An author of an Employee or File subclass will need to register the subclass in the Static_Process_Pairs object, but there is nothing in the typing rules in the inherited Employee or File classes that will generate a compile-time error if a subclass author fails to adhere to the methodology by neglecting to register all legal pair combinations and/or by registering illegal pair combinations. This is an attempt to correct a language deficiency with methodology, with the result being that program correctness rests not on the type-checker but on the source code documentation and programmer discipline.

Covariant subtyping alone (such as that permitted in Eiffel) does not completely solve the problem. It *would* permit the original formulation of the class hierarchy:

```
Abstract Class File { ... }

Abstract Class Employee {
  void process(File f) { ... }
}

Class Design extends File { ... }

Class Personnel extends File { ... }

Class Engineer extends Employee {
  void process(Design d) { ... }
}

Class HR extends Employee {
  void process(Personnel p) { ... }
}
```

However, we do not gain as much as we would have hoped for. Although the methods are now type-safe, we have impaired our ability to make polymorphic method calls. Our original formulation of the Administrator class can no longer satisfy a flow-insensitive type-checker, because the File arguments being passed to the Employee.process methods do not conform to the arguments expected by the Engineer.process and HR.process methods.

```
Class Administrator {
   void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
      foreach (emp in e) {
         foreach (file in f) {
            try {
               emp.process (file);
            } catch (TypeException e) { }
         }
      }
   }
}
```

The system cannot verify that all calls to emp.process(file) satisfy the typing properties of the polymorphic method, and so the process_All method is rejected. Instead, we must resort again to trying all possible type combinations.

```
Class Administrator {
   void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
      foreach(emp in e) {
         foreach(file in f) {
            Engineer eng ? = emp;
            HR hr ? = emp;
            Design des ? = file;
            Personnel per ? = file;
            if(eng / = null and des / = null) {
               eng.process(des);
            }
            else if(hr / = null and per / = null) {
               hr.process(per);
            }
         }
      }
   }
}
```

The inability to polymorphically call the Employee.process(File) method means that the process_All method needs to be redefined every time an Employee or File subclass is added or modified. We have achieved static type-safety, but have not necessarily made the programmer's job any easier by doing so.

## 3. New approaches

### *3.1.* OnlyWith *downcasts*

By introducing an OnlyWith operator into the language, we can benefit from the simplicity of the try-catch block while eliminating the overhead of the exception handler. An OnlyWith operator specializes the parameter(s) of an inherited method by executing the method body as normal for any calls whose arguments conform to the formal parameters, and ignoring any calls whose arguments fail to conform to the formal parameters. This is in contrast to a covariant type relation that statically rejects any method calls whose arguments may not conform to the method parameters.

Implementing the OnlyWith operator for procedures is straightforward. If a method argument conforms to the OnlyWith formal parameter then the method body is executed, otherwise the method acts as a nop.

```
Class Engineer extends Employee {
  void process(File OnlyWith Design d) { ... }
}
Class HR extends Employee {
  void process(File OnlyWith Personnel p) { ... }
}
Class Administrator {
  void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
    foreach(emp in e) {
      foreach(file in f) {
        emp.process(file);
      }
    }
  }
}
```

The process_All method is type-safe by virtue of the fact that the Engineer.process and HR.process method signatures now conform to the Employee.process method signature. This is functionally equivalent to performing a conditional assignment within the method bodies:

```
Class Engineer extends Employee {
  void process(File f) {Design d ?= f; ... }
}
Class HR extends Employee {
  void process(File f) {Personnel p ?= f; ... }
}
```

However, the OnlyWith keyword allows the subclass to advertise the covariant type relationships in the Engineer.process and HR.process method signatures, whereas a programmer would only otherwise be aware of the conditional assignment from the method documentation. By exposing the downcast in the method signature the client of an Engineer or HR class is given additional flexibility in handling the process method, e.g., the client can use a type-sensitive iterator to improve performance (see below), or can throw an exception if a conditional assignment is inappropriate in the particular calling context.

The potential complication introduced by the OnlyWith keyword is with methods that return a value, in which case replacing the method body with a nop will not suffice. We can correct this problem by always returning a null value or else by requiring the programmer to specify a default value to be returned if the argument does not conform to the OnlyWith parameter.

Returning a null value is the simplest solution for reference types, but will likely require a major language redefinition in order to be applicable to primitive/expanded types. Floating point primitives could return a NAN and character primitives could return the null character, but integers and booleans do not have natural null representations. We could steal a bit sequence from the set of integers (say, 0xFFFF) and define it as the null integer, but without hardware support for this scheme we would then have to use software for such things as overflow detection. Booleans are even more problematic, because if we were to add a null flag to each boolean variable then we would now require two bits for a boolean whereas without null booleans we would only need one bit.

It is therefore more straightforward to provide the programmer with a means of defining a default return value for cases when it is inappropriate to return the natural null representation or for cases when there is no natural null representation. If no default return value is specified then either a static type exception is raised or else a run-time exception is thrown when the method is called with arguments that do not conform to the parameters, depending on the language specification.

```
Abstract Class Employee {
  boolean process(File f) { ... ; return true}
}

Class Engineer extends Employee {
  boolean ⟨default: false⟩ process(File OnlyWith Design d) {
    ... ; return true }
}

Class HR extends Employee {
  boolean ⟨default: false⟩ process(File OnlyWith Personnel p) {
    ... ; return true }
}
```

This is functionally equivalent to the use of conditional assignment:

```
Class Engineer extends Employee {
  boolean process(File f) {
    Design d ? = f
    if(d  = null) { return false }
    else { ... ; return true }
  }
}

Class HR extends Employee {
  boolean process(File f) {
    Personnel p ? = f
    if(p  = null) { return false }
    else { ... ; return true }
  }
}
```

The advantages of the use of OnlyWith over the use of conditional assignment remain the same: the OnlyWith method makes the default return value explicit in the method signature, whereas the conditional assignment method hides the default return value in the method body and/or documentation.

An alternative to requiring the programmer to specify a default return value is to always return the language-defined default initialization for the value (e.g., 0 and 0.0, respectively for integers and reals) unless the programmer defines a default or specifies that a run-time exception ought to be thrown if the method is called with arguments that do not conform to the parameters.

```
Class Engineer extends Employee {
  boolean ⟨default: throw new RunTimeException( )⟩
```

```
        process(File OnlyWith Design d) {
            ... ; return true }
}

Class HR extends Employee {
    boolean ⟨default: throw new RunTimeException( )⟩
        process(File OnlyWith Personnel p) {
            ... ; return true }
}
```

The power of the OnlyWith operator can be further enhanced by permitting boolean type expressions in the argument, e.g.:

```
Class Engineer extends Employee {
    void process(File OnlyWith (Design OR Spec) f) { ... }
}
```

The Engineer.process method call would then execute on arguments that conformed to either Design or Spec (both being subclasses of File), and would otherwise perform a nop. This is functionally equivalent to the method call using conditional downcasts or reflective mechanisms to insure that the arguments conforms to the (implicit, when using the techniques of conditional downcasts or reflection) method parameters.

```
Class Engineer extends Employee {
    void process(File f) {
        Design d ? = f;
        Spec s ? = f;
        if(d / = null or s / = null) {
            ...
        }
    }
}
```

*3.2. Type-sensitive iterators*

We can facilitate the use of covariant typing relations with iterators that eliminate the need to explicitly downcast objects. Consider a straightforward generic List iterator that would require downcasting in the examples given above.

```
Class List ⟨G⟩ {
    private Node head := null;

    private Class Node {
        ⟨G⟩ obj;
        Node next;
        Node(⟨G⟩ o) {
            obj := o;
        }
    }

    public void add(⟨G⟩ o) {
        Node n := new Node(o);
        n.next := head;
        head := n;
    }

    public Iterator getIterator( ) {
```

```
    return new Iterator( );
  }

  private Class Iterator {
    Node current;
    Iterator( ) {
      current := head;
    }
    public boolean hasNext( ) {
      return current / = null;
    }
    public ⟨G⟩ getNext( ) {
      return current.obj;
    }
    public void next( ) {
      current := current.next;
    }
  }
}
```

As we have seen, given lists of Employees and Files the Administrator must downcast the return values of the iterator, and/or the Administrator must be prepared to handle type violations. However, this need not be the case if we rewrite our List so as to be type-sensitive, using generic methods (an extension on generic classes). We will use the ⟨X⟩⟨⟨X1⟩⟨X2⟩⟩ notation to assert that X1 and X2 are subtypes of X.

```
Class List ⟨G⟩ ⟨⟨G1⟩ ⟨G2⟩⟩ {
  private Node⟨G⟩ head⟨G⟩ := null;
  private Node⟨G1⟩ head⟨G1⟩ := null;
  private Node⟨G2⟩ head⟨G2⟩ := null;

  private Class Node⟨G⟩ {
    Node⟨G⟩ next⟨G⟩;
  }

  private Class Node⟨G1⟩ extends Node⟨G⟩ {
    ⟨G1⟩ obj;
    Node⟨G1⟩ next⟨G1⟩;
    Node⟨G1⟩(⟨G1⟩ o) {
      obj := o;
    }
  }

  private Class Node⟨G2⟩ extends Node⟨G⟩ {
    ⟨G2⟩ obj;
    Node⟨G2⟩ next⟨G2⟩;
    Node⟨G2⟩(⟨G2⟩ o) {
      obj := o;
    }
  }

  public void add(⟨G1⟩ o) {
    Node⟨G1⟩ n := new Node(o);
    n.next⟨G⟩ := head⟨G⟩;
    head⟨G⟩ := n;
```

```
      n.next⟨G1⟩ := head⟨G1⟩;
      head⟨G1⟩ := n;
   }

   public void add(⟨G2⟩ o) {
      Node⟨G2⟩ n := new Node(o);
      n.next⟨G⟩ := head⟨G⟩;
      head⟨G⟩ := n;
      n.next⟨G2⟩ := head⟨G2⟩;
      head⟨G2⟩ := n;
   }

   public Iterator⟨G⟩ getIterator⟨G⟩( ) {
      return new Iterator(⟨G⟩);
   }

   public Iterator⟨G1⟩ getIterator⟨G1⟩( ) {
      return new Iterator(⟨G1⟩);
   }

   public Iterator⟨G2⟩ getIterator⟨G2⟩( ) {
      return new Iterator(⟨G2⟩);
   }

   private Class Iterator⟨X⟩ {
      Node⟨X⟩ current;
      Iterator(⟨X⟩) {
         current := head⟨X⟩;
      }
      public boolean hasNext( ) {
         return current / = null;
      }
      public ⟨X⟩ getNext( ) {
         return current.obj;
      }
      public void next( ) {
         current := current.next;
      }
   }
}
```

Note that, rather than speeding up computation at the cost of space by declaring multiple references in each Node (in this case, one for the superclass iterator, one for the subclass iterator), we could instead achieve the same effect by having the specialized ⟨G1⟩ or ⟨G2⟩ iterators search through the list of ⟨G⟩ objects until a ⟨G1⟩ or ⟨G2⟩ object is located, rather than maintain references to both the next ⟨G⟩ object and the next ⟨G1⟩ or ⟨G2⟩ object.

Now rather than declaring a List⟨File⟩ or List⟨Employee⟩ container that would require downcasts or exception handling mechanisms, we can instead declare a List⟨File⟩⟨⟨Design⟩⟨Personnel⟩⟩ or List⟨Employee⟩⟨⟨Engineer⟩⟨HR⟩⟩ container. Then the Administrator can select the appropriate iterator from the List container to circumvent the need to downcast objects in the Administrator object.

```
Class Administrator {
   void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
      Iterator⟨Engineer⟩ eng := e.getIterator⟨Engineer⟩( );
      foreach(emp in eng) {
         Iterator⟨Design⟩ des := f.getIterator⟨Design⟩( );
      foreach(file in des) {
        emp.process(file);
      }
    }
    Iterator⟨HR⟩ hr := e.getIterator⟨HR⟩( );
    foreach(emp in hr) {
      Iterator⟨Personnel⟩ per := f.getIterator⟨Personnel⟩( );
      foreach(file in per) {
        emp.process(file);
      }
    }
  }
}
```

We are still required to rewrite the Administrator class whenever there is a change in access controls for the Employee or File subclasses, but now we can rely on the local type-checking mechanism to detect errors in the process_All method rather than on a global system validity check. In addition, we can now instruct the type-checking mechanism to issue warnings if the nested iterator (in this case, the File iterator) is a proper subtype of the class that would be accepted by the nesting iterator, flagging many potential errors that would not have been detected without using covariant type relations. For example, if a Supervisor object capable of processing all File objects was instead provided an iterator that was only capable of processing all Design objects, then the type-checking mechanism would be able to provide a warning to the code author. (It would only be appropriate to provide a warning, not an error, in this circumstance; the code author is certainly permitted to provide the Supervisor with an iterator that does not encompass all objects that the Supervisor is allowed to process.)

### *3.3. Type parameters and type iterators*

The method for incorporating flexible covariant typing with polymorphism discussed in the previous section still requires us to rewrite the Administrator class to reflect changes in the Engineer, HR, Design, and Personnel classes. Fortunately, the compiler is now able to locally type-check the Administrator class, so there is no longer a risk that a change to an Employee or File subclass may cause a runtime type mismatch or failed cast in the process_All method, nor is there a need for a global system validity check. But there is no reason why we ought to limit ourselves to simply type-checking the Administrator class—if we extend the language to allow us to extract type parameters from formal method arguments, then we can produce an Administrator class that does *not* need to be rewritten to reflect changes in the Employee and File subclasses.

```
Class Administrator {
  void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
    Iterator⟨Engineer⟩ eng := e.getIterator⟨Engineer⟩( );
    foreach(emp in eng) {
      Type ⟨G⟩ := emp.process(⟨G⟩);
      Iterator⟨G⟩ gen := f.getIterator⟨G⟩( );
      foreach(file in gen) {
        emp.process(file);
      }
    }
    Iterator⟨HR⟩ hr := e.getIterator⟨HR⟩( );
    foreach(emp in hr) {
      Type ⟨G⟩ := emp.process(⟨G⟩);
      Iterator⟨G⟩ gen := f.getIterator⟨G⟩( );
      foreach(file in gen) {
        emp.process(file);
      }
    }
  }
}
```

The Type ⟨G⟩ := Employee.process(⟨G⟩) statements indicate that ⟨G⟩ takes the type of the formal parameter to Employee.process(⟨G⟩). For example, the method call Type ⟨G⟩ := Engineer.process(⟨G⟩) would evaluate to Type ⟨G⟩ := Design, and likewise the method call Type ⟨G⟩ := HR.process(⟨G⟩) would evaluate to Type ⟨G⟩ := Personnel. The type-checking mechanism need only verify that every formal argument Employee.process(⟨G⟩) referenced in the foreach loops have corresponding iterators FileList.getIterator(⟨G⟩), i.e., in the case that there is a potential call to Engineer.process(⟨G⟩) (as in the Iterator⟨Engineer⟩ loop) then the type-checker would throw an error if there is not a corresponding Iterator⟨Design⟩. The compiler may then implement this loop by statically eliminating the Type ⟨G⟩ definition and replacing all instances of ⟨G⟩ with instances of the actual formal argument to Employee.process( ) - Type arguments are not intended to be evaluated with reflective mechanisms at runtime (although this remains a possibility in languages with support for reflection), but are instead only used by the type-checker and compiler. If (as in this case) all iterators reference leaf classes (that is, classes that do not have any subclasses), then the compiler will also be able to replace the virtual call to Employee.process(File) with a static call to ⟨A⟩.process(⟨B⟩) where ⟨A⟩ is the Employee type and ⟨B⟩ is the File type. This in turn will facilitate inlining and other optimizations that would not have been possible without the use of a type-sensitive iterator.

One final step remains in making the Administrator class completely independent of the Employee and File classes. As written, the Administrator must still address each part of the Employee taxonomy with its own foreach loop, for example, adding a Technician subclass that inherits from Employee would necessitate the addition of a foreach(emp in Iterator⟨Technician⟩) { ...} loop. This maintenance shouldn't be necessary—the inheritance hierarchy will either form a tree or a DAG (for class hierarchies using single inheritance or multiple inheritance, respectively), and each of these data structures is sufficiently well understood that the compiler will be able to automatically generate iterators through the inheritance hierarchy. In addition to improving ease of maintenance, the introduction of type iterators can also result in significant reduction in code size—the process_All method below requires the same number of instructions regardless of how many Employee and File subclasses are in the inheritance hierarchy.

```
Class Administrator {
  void process_All (List ⟨File⟩ f, List ⟨Employee⟩ e) {
    TypeIterator itr = ⟨Employee⟩.getSetIterator(process(⟨File⟩));
    foreach(set in itr) {
      Iterator⟨set⟩ empItr := e.getIterator⟨set⟩( );
      foreach(emp in empItr) {
        Type ⟨G⟩ := emp.process(⟨G⟩);
        Iterator⟨G⟩ gen := f.getIterator⟨G⟩( );
        foreach(file in gen) {
          emp.process(file);
        }
      }
    }
  }
}
```

The ⟨Employee⟩.getSetIterator(process(⟨File⟩)) method call returns an iterator through the type hierarchy designed to satisfy every Employee method's process(File) call exactly once. For a true taxonomic inheritance tree in which all superclasses are abstract, this iterator trivially returns only the leaf classes, in this case the Engineer and HR classes. The situation becomes more complicated when non-trivial taxonomies and/or multiple inheritance are introduced. Consider the following inheritance hierarchy:

```
Abstract Class File { ... }

Abstract Class Employee {
  void process(File f) { ... }
}

Class Design extends File { ... }
Class Personnel extends File { ... }

Class Engineer extends Employee {
  void process(Design d) { ... }
}

Class HR extends Employee {
  void process(Personnel p) { ... }
}

Class Manager extends HR, Engineer {
  void process(File f) { ... }
}
```

To satisfy each Employee's process method exactly once, the compiler can simply ignore the Manager node. The union of all Design and Personnel objects equals the set of File objects, and so the Manager.process(File) method will be satisfied by simply evaluating both its HR.process(Personnel) method call and its Engineer.process(Design) call in the course of iterating through the HR and Engineer lists. In the case that this fails to satisfy the hierarchy, for example if there were an additional File type:

```
Class Accounting extends File{ ... }
```

then the compiler could either iterate through the Engineer and HR lists and ignore the Managers in them and then iterate through the Manager list and process all File objects, or the compiler could iterate through the Engineer and HR lists without ignoring Managers and then iterate through the Managers list and process all Accounting objects. In ambiguous cases such as this the compiler may request assistance from the author of the Accounting or Manager class for selecting the best strategy (although the compiler may always fall back on a trivial but possibly sub-optimal iteration strategy), but the author of the Administrator class need not be concerned with the matter.

## *3.4. Automated genericity*

The final obstacle to the safe and unimpeded use of covariant typing relations is in the complexity of the iterator. By using type parameters and type iterators we no longer need to modify the Administrator class to reflect changes in the Employee and File type hierarchies, but we are still required to rewrite the List⟨G⟩ class every time the hierarchy changes to incorporate new type-sensitive iterators into the container. The container may also become unwieldy over the course of a project's lifetime as methods become obsolete, for example, if a type-sensitive iterator were no longer in use due to changes in the method arguments and/or inheritance hierarchy. Performance may also begin to degrade as expected object statistics change: for example, a type-sensitive iterator for class A1 inheriting from A, where A1 objects are initially expected to compose a large percentage of the A objects, may favor space efficiency over speed by iterating through the entire list of A objects and ignoring those objects that do not conform to A1; however, over the course of the project's lifetime the set of A1 objects may only compose a marginal percentage of the set of A objects, and the performance of the type-sensitive A1 iterator will suffer as a result.

We argue that the advantages presented by using type-sensitive iterators in addition to the potential difficulty of maintaining type-sensitive iterators justifies the inclusion of automated genericity as a special language construct.

```
Class List ⟨G⟩ (⟨X⟩ ➔ ⟨G⟩) {
  private Field⟨Node⟨G⟩⟩ head⟨X⟩;

  private Class Node⟨X⟩ extends Node⟨G⟩ {
    head.addField(Node⟨X⟩);
    Node⟨X⟩ next⟨X⟩;
    Node(⟨X⟩ o) {
      obj := o;
    }
  }
  public void add(⟨X⟩ o) {
    Node⟨X⟩ n := new Node⟨X⟩(o);
    TypeIterator itr = ⟨X⟩.getSuperTypeIterator(head);
    foreach(Y in itr) {
      n.next⟨Y⟩ := head⟨Y⟩;
      head⟨Y⟩ := n;
    }
  }

  public Iterator⟨X⟩ getIterator⟨X⟩( ) {
    return new Iterator⟨X⟩( );
  }

  private Class Iterator⟨X⟩ {
    Node⟨X⟩ current;
    Iterator(⟨X⟩) {
      current := head⟨X⟩;
    }

    public boolean hasNext( ) {
      return current / = null;
    }
    public ⟨X⟩ getNext( ) {
      return current.obj;
    }
    public void next( ) {
      current := current.next;
    }
  }
}
```

The (⟨X⟩ ➔ ⟨G⟩) parameter indicates that type X is equal to or a subtype of type G. Thus, the list is designed to fulfill a request for any type-sensitive iterator whose generic type is G or a subtype of G - it is up to the compiler to determine which type-sensitive iterators may be required. In this case the compiler favors speed over memory efficiency by generating a separate Node⟨X⟩ class for each potential argument to getIterator⟨X⟩( ), as well as a separate Node⟨X⟩ head field and Node⟨X⟩.next⟨X⟩ link. An alternative formulation could favor memory efficiency over speed by using a single Node⟨G⟩ class and filtering the results using reflection.

```
Class List ⟨G⟩ (⟨X⟩ ➔ ⟨G⟩) {
  private Node⟨G⟩ head;

  private Class Node⟨G⟩ {
    Node⟨G⟩ next⟨G⟩;
    Node(⟨G⟩ o) {
      obj := o;
    }
  }

  public void add(⟨G⟩ o) {
    Node⟨G⟩ n := new Node⟨G⟩(o);
    n.next⟨G⟩ := head;
    head := n;
  }
  public Iterator⟨X⟩ getIterator⟨X⟩( ) {
    return new Iterator⟨X⟩( );
  }

  private Class Iterator⟨X⟩ {
    Node⟨X⟩ current;
    Iterator(⟨X⟩) {
      Node⟨G⟩ test := head;
      current := null;
      while(current = null and test.hasNext( )) {
        current ? = test.getNext( );
        test.next( );
      }
    }
    public boolean hasNext( ) {
      return current / = null;
    }
    public ⟨X⟩ getNext( ) {
      return current.obj;
    }
    public void next( ) {
      Node⟨G⟩ test := current.next( );
      current ? = null;
      while(current = null and test.hasNext( )) {
        current ? = test.getNext( );
        test.next( );
      }
    }
  }
}
```

### 3.5. *Experimental setup and results*

We wrote a test program in Java to simulate the three basic mechanisms discussed in this paper: downcasting inside of a try-catch block, conditional downcasting (as with Eiffel's conditional assignment ? =, or using an OnlyWith construct), and type-sensitive iteration. The program generates $N$ elements of type A and $N$ elements of type B, and then uses a double foreach loop to process all elements process(A ai, B bj). The A and B elements are split into $S$ subtypes {A1, A2, …AS}, {B1, B2, …BS}, where the typing relation is defined for Ai.process(Bi), that is, there is a one-to-one mapping between subtypes.

Conditional downcasts were simulated with Java's reflection mechanisms. All calls to the process(A ai, B bj) method were made inside of an if (isValid(ai.getClass( ), bj.getClass( )) block, where isValid returned true if the method call would not generate a run-time exception from a failed class cast or assertion failure. Thus, the conditional downcast simulation never invoked the exception handler. (We used a constant-time lookup table rather than a linear-time boolean expression to implement the isValid method. This implementation will not be effective in a many-to-many mapping, however.)

We simulated type-sensitive iteration by removing the process method from the superclass, so that every A subclass could contain a process(Ai ai, Bi bi) method (instead of the process(A ai, B bi) method). The lists linked each X element to the next X element, and also linked each Xi element to the Xi element (for X being A for the list of A elements and X being B for the list of B elements). Thus, the implementation favored speed over memory efficiency.

We ran our tests using jre1.5.0_04 running on the Eclipse platform, on an Intel Pentium 4 with 16 kB of L1 cache, 1 MB of L2 cache, and 1 GB of main memory. We averaged our results from 20 runs of the Try-Catch block tests and from 100 runs of the conditional downcast and type-sensitive iterator tests.

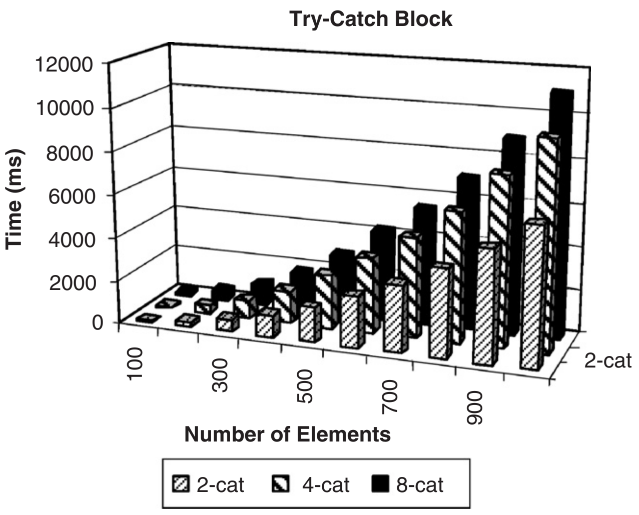

Fig. 1. Averaged results of the Try-Catch block tests.

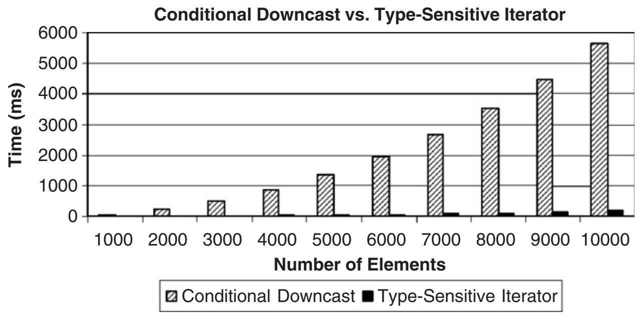

Fig. 2. Averaged results of the Conditional Downcast and Type-Sensitive Iterator tests.

On the tests using a Try-Catch block to catch and ignore type errors we used either two (2-cat), four (4-cat), or eight (8-cat) subtypes with element counts ranging from 100 to 1000 (Fig. 1). The exception handler introduced a two–three order of magnitude slowdown over the conditional downcast and type-sensitive iterator programs. The 8-cat tests threw more exceptions than the 2-cat tests, and therefore introduced more overhead from the exception handler.

We timed the runs of the program using a conditional downcast or a type-sensitive iterator with element counts ranging from 1000 to 10 000. There was negligible deviation in performance from the 2-cat, 4-cat, and 8-cat runs, and so we only show the results of the 8-cat runs (Fig. 2). The conditional downcast implementation required less than 100 ms for 1000 elements, compared to the more than 10 000 ms required in the Try-Catch block implementation for 1000 elements. The type-sensitive iterator implementation required fewer than 200 ms for 10 000 elements, compared to the more than 5000 ms required in the conditional downcast implementation for 10 000 elements. Had the type-sensitive iterator favored memory efficiency over speed then the timing results would have been equivalent.

The Try-Catch block implementation is the easiest to maintain, but incurs by far the most overhead. The conditional downcast implementation is much more efficient, but requires language support for reflection and/or conditional assignment, and makes code much more difficult to maintain as all method call sites must be modified when a method signature or inherited method signature changes. However, if conditional downcasts are implemented by OnlyWith operators then the code becomes as easy to maintain as the Try-Catch block implementation. Alternatively, conditional downcasts can be implemented with the code that is functionally equivalent to the code with OnlyWith operators, but this in turn hides the method's type information inside of the method or documentation whereas the OnlyWith operator keeps the type information in the method signature. The type-sensitive iterator implementation is the most efficient, but is not as practical as the Try-Catch block or conditional downcast implementations without language support for type parameters, type iterators, and automated genericity.

## 4. Conclusions and future research

Type-safety justifies the inclusion of a covariant type mechanism into a language. Ease of type-checking and ease of implementation justifies the introduction of an OnlyWith operator to facilitate covariant type relations. Inclusion of type-sensitive iterators requires some additional compiler support, but can produce significant improvements in speedup and memory efficiency.

The examples of type-sensitive iterators using automated genericity given in this paper do not address the issue of using profiler data to tailor an iterator to either improve speed or memory efficiency as appropriate—as given, the iterators either favor speed or memory efficiency, and do not provide opportunities for the compiler to assist in selecting one over the other. To address this issue (and the more general issue of enabling the use of profiler data to tailor other data structures, e.g., in selecting structures with equivalent interfaces but different run-time properties like a LinkedList, ArrayList, or RandomAccessList) we would like to study language mechanisms for deferred expressions (e.g., "instantiating" an abstract List object, leaving the selection of a concrete implementation for later in the development or compilation process).

We are also interested in developing a compact intermediate language that can serve as an appropriate target for a language that enables covariant type relations. Even if type-sensitive iterators are not enabled at the language level then they may still prove useful in reducing global system validity checks to local class validity checks, thus improving the efficiency of verification. In addition, we would like to incorporate automated genericity into an intermediate language to examine the potential benefits of generating self-modifying code using a just-intime compiler.

## References

[1] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Boston, MA: Addison-Wesley Professional; 1995.
[2] Castagna, G. Covariance and contravariance: conflict without cause. ACM Transactions on Programming Languages and Systems 1995; 431–47.
[3] Stroustrup B. The C + + programming language. Boston, MA: Addison-Wesley; 1997.
[4] Gosling J. et al. The java language specification. Boston: Addison-Wesley; 2000.
[5] Meyer B. Object-oriented software construction. Upper Saddle River, NJ: Prentice-Hall; 1997.
[6] Meyer B. Eiffel: the language. New York: Prentice-Hall; 1992.