Dynamic Type Inference to Support Object-Oriented Reengineering in Smalltalk

Pascal Rapicault*,
Mireille Blay-Fornarino*, Stéphane Ducasse+, Anne-Marie Dery*
* I3S University of Nice-Sophia Antipolis*
+ Software Composition Group, Universität Bern†

Abstract

Type information is a crucial information to support object-oriented reengineering. In a dynamically typed language like Smalltalk standard static type inference is a complex and heavily computational task. In this paper, we report how we use message passing control and compiler extension to support dynamic inference type in Smalltalk.

1 Object-Oriented Reengineering Goals and Needs

In the context of the european esprit project Famoos, the SCG team is focusing especially on application model capture, (i.e. supporting the documentation and the understanding of an application) and problem detection, (i.e. supporting detection of possible problems in regards to the requirements). The Rainbow team at University of Nice-Sophia Antipolis was developing a dynamic type inference systems for Smalltalk. So both teams collaborated having quite similar goals: capture the model underlying an operational Smalltalk program to support the developer to modify or rewrite it. In this context, type information of method arguments or instance variables is absolutely necessary in the following tasks:

- To compute for metrics based on coupling among classes,
- To check class coupling (Law of Demeter),
- To group classes according to their relationships,
- To support extraction of design model entities like association and aggregation,
- To detect patterns and class properties such as abstract classes, mixins, constructors, builders, and so on.

Some commercial tools support lightweight static type inference by looking for instance variable assigments in the initialize method. Such an approach alone is not really convincing because the initialize method can be implemented using a composed method [Bec97], resulting in no type information.

Static type inference techniques have been developed for dynamically typed languages like Self and Smalltalk [BI82, Suz81, PC94, Age96, Joh86]. However, those techniques are complex to implement and require a lot of computation power. Type inference techniques-for compiler implementation must provide strong optimizations and distinguish between

*Bat. Essi 650 Route des Colles. B.P. 145. 06903 Sophia-Antipolis Cedex, France (blay, rapicaul, dery)@essi.fr, http://www.essi.fr/∼blay/
†Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, 3012 Berne, Switzerland (ducasse)@iam.unibe.ch, http://www.iam.unibe.ch/∼(ducasse)/

primitive types like small integer, integer and float. In our context, we are not really interested to know if an instance variable is a small integer or an integer, the main type values are the other domain classes defined in the reengineered application. Finally, in the context of reengineering, we do not believe in "click and doit" fully automatized systems that reengineer object-oriented applications like the refactoring approach of Moore [Moo96], even if such an approach could be suitable in the context of application deployement. We strongly believe in systems that help and support the domain experts in their work but also sollicite their knowledge about the application.

We have chosen to look at the execution of the programs and to dynamically deduce types. The hypothesis of this work are then the coverage of tests is "complete" and the program is operational. Supposing these conditions are verified, we deduce for the interesting classes and their metaclasses, the type of their variables and the signature of the methods. We also determine the type of the temporal variables manipulated in the methods associated to the classes we want to type. Moreover, the programmer can control the deduced type by using the offered dynamic information (number of class, list of type values taken by variables and parameters).

Having the presented goals in mind, we experimented with a ligthweight dynamic type inference in Smalltalk based on variable access and message passing control. We extended the Method Wrapper and sligthly changed the Smalltalk compiler. After a short presentation of the used algorithm, we report here this experiment.

Algorithm

Let us note T the type hierarchy and the partial order relation $\subset$.

- Let us note $\emptyset$[1] the common subtype.

  $\forall \tau \in T, \emptyset \subset \tau$

- We define the function $\uparrow$ that computes the smallest common super class of two classes :

  $\uparrow: T \times T \rightarrow T$

  $t1, t2 \rightarrow a, t1 \subset a, t2 \subset a, \neg \exists b \in T, b \subset a, t1 \subset b, t2 \subset b.$

- We define the function *infer* that deduces according to the current type, initially $\emptyset$, the encounter type and the set of encountered types, the new current type and the new set of encountered types.

  Infer : $T \times T \times H* \rightarrow T \times H*$

  - $\emptyset, t, h \rightarrow t, h \cup \{t\}$
  - $t', t, h \rightarrow t, h \cup \{t\}$ *where* $t' \subset t$
  - $t', t, h \rightarrow t', h \cup \{t\}$ *where* $t \subset t'$
  - $t', t, h \rightarrow (t \uparrow t'), h \cup \{t\}$ *where* $\neg(t \subset t'), \neg(t' \subset t)$

This algorithm allows one to infer the type of a variable. Applied sequentially on all the parameters of a method, we obtain its signature.

[1] $\emptyset$ corresponds in Smalltalk to UndefinedObject. In this particular case we don't use the Smalltalk class hierarchy.

2 Method Wrappers and Compiler Extension

Wrapping methods is one of the most suitable techniques to control message passing at the level of the class [BJRF98, Duc98]. We created a new subclass of the MethodWrapper class. The new class TypingMethodWrapper maintains a reference to the associated structure storing type method information. We then specialized the method valueWithReceiver:arguments: so that we stored the type after each invocation of the wrapped method.

To instrument the code, we extended in a simple manner the method wrappers [BJRF98] and modified the Smalltalk compiler [Riv96]. We implemented various structures to store types of class elements: methods, instance variables, class variables. Such structures are responsible for storing and managing type information queries.

Typing Variable: a First Attempt. To know the type of variable, we subclassed the class Parser by the class TypingParser and redefined the assigment:startingAt: method. We also specialized the Compiler class method preferredParserClass on a new subclass TypingCompiler whose instances are used to recompile methods being typed.

```
TypingParser>>assignment: var startingAt: start
 "variable <left arrow> (':=') expression => AssignmentNode"
 " !!!! Change the parse tree.
   The ValueNode is now a MessageNode that computes the value,
   examines and stores its type."

 | leftArrow param |
 leftArrow := token = Character leftArrow.     "left arrow"
 self scanToken. "skip the left arrow or :="
 self expression ifFalse: [^self expected: 'Expression'].
 param := Array
           with: (LiteralNode new value: var)
           with: (LiteralNode new
                       value: TypingParser currentSelector)
           with: (LiteralNode new
                       value: TypingParser theClass).
 parseNode := AssignmentNode new
                     variable: var
                     value: (MessageNode new
                                  receiver: parseNode
                                  selector: #var:inMethod:in:
                                  arguments: param)
                      leftArrow: leftArrow.
 parseNode sourcePosition: (start to: self endOfLastToken)
```

Then we defined on Object the method var:inMethod:in: that is invoked when a variable is affected and store the value of the variable.

```
Object>>var: varName inMethod: aSel in: aClass
   DTTypeStructureRegister new
        registerVarType: varName name
        inMethod: aSel
        in: aClass
        value: self class
```

This first attempt was positive with regards to the time invested and the results obtained. However, we would like to see if we extended the Smalltalk compiler using the right hot spots. We want to have a deeper understanding of the Smalltalk compiler to evaluate the impacts of our changes [HJ95b, Riv96].

3 Thoughts and Evaluation

The assumption that the coverage is complete is an important factor for computing the type and for analysising the results. This assumption differs in both teams. In the Rainbow

team, that originally developed the algorithm, the assumption about the existence of a complete coverage was strong. This is reflected also in the presented algorithm as discussed later. The SCG team took the opposite view about the completeness of the test. Principally because in the context of application understanding, no one is sure that running the application will simply cover all the cases. So SCG team developed another algorithm not presented here.

The following situation illustrates the main difference. If a method is defined in a class A and is only invoked via inheritance on instances of one of its subclasses, each algorithm gives a different result: if there are no calls done via other subclasses or via the class B itself the Rainbow algorithm returns a type in terms of the subclass B, because they choose dynamic precision and assume strong complete coverage. In the same situation, the SCG algorithm returns as type the class A, the class in which the method is currently defined. This was needed because the goal was to avoid introducing relationships between classes and subclasses.
Besides knowing instance variable and method types, applying dynamic type inference helps to find other information like:

1. Some methods are never called. If we assume that the coverage is complete, we can deducethat the method is abstract or hooked. In the same way that the Coverage Tools based on Method Wrapper [BJRF98], pointing this out helps the developer to decide if a method is obsolete.

2. As we explained above depending on the algorithm, a deduced type value could be a subclass of the class defining the method itself. In the context of a complete coverage, if no other methods of the same class exist having as type for receiver or parameters, we can deduce that this class is abstract. Note that under the hypothesis that the coverage is complete, inferring as type of a method a subclass of the class in which the method is defined shows a design problem where class is bound with its subclasses.

3. Sometimes, comparing the signature of the calls allows one to detect a method needing multi-discrimination. In such a case, it's interesting to detect it and to propose at model level several methods capturing better the reality.

4. When the return value of a class method is an instance of the class, we probably have found a "constructor". If the return value are only instances of some subclasses, the class is abstract.

5. Finally, often the deduced type are classes of classes. This information is essential to capture the model. But more studies should be done to really take advantages of such an information.

4 Related and Future Work

As we already discussed, part of this work relies on the completeness of the test coverage. The synergy between these two approaches is important and using the tests developed for covering a program leads to better results.

[RB98] describes how they use both dynamic type inference via method wrappers and active values [HJ95a]. Their approach is similar to ours, so we definitely plan to compare both approaches. Moreover we want to knowto which degree our approach can be extended to take into account their work on the determination of class relationship cardinality.

We also want to analyse memory consumption and the scalability of the approach to large applications.

As this work is in its early stages, we have not yet exploited all the advantages of this approach. We yet argue that it allows some detections which are not allowed by static

inference. We are working on the determination of other features of the model underlying a reengineered application by a finer analysis of the obtained results.

References

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Sun Microsystems Laboratories, 1996.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997. ISBN: 0-13-476904-X.

[BI82] Alan Borning and Dan Ingalls. A type declaration and inference system for smalltalk. In *ACM Symposium on Principles of Programming Languages*, 1982.

[BJRF98] John Brant, Ralph E. Johnson, Donald Roberts, and Brian Foote. Wrappers to the rescue. In *To appear in Proceedings of ECOOP'98*, LNCS, page ??? Springer-Verlag, 1998.

[Duc98] Stéphane Ducasse. Evaluating Message Passing Control Techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 1998. To appear near the end of the year.

[HJ95a] Bob Hinkle and Ralph Johnson. The active life is the life for me. *Smalltalk Report*, 5(7):14–21, may 1995.

[HJ95b] Bob Hinkle and Ralph Johnson. Parametrized compiler: Making code reusable. *Smalltalk Report*, 4(9):13–18, jul-aug 1995.

[Joh86] R.E. Johnson. Type-checking Smalltalk. In *Proceedings OOPSLA'86*, pages 315–321. ACM, sep 1986.

[Moo96] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of OOPSLA'96*, pages 235–250. ACM, 1996.

[PC94] J. Plevyak and A. Chien. A precise concrete type inference for object-oriented languages. In *Proceedings of OOSPLA'94*, pages 324–340, 1994.

[RB98] Don Roberts and John Brant. "good enough" analysis for refactoring. In *ECOOP'98 International Workshop on Object-Oriented Reengineering*, 1998.

[Riv96] Fred Rivard. Smalltalk : a Reflective Language. In *Proceedings of REFLECTION'96*, pages 21–38, April 1996.

[Suz81] N Suzuki. Inferring types in smalltalk. In *8th Annual Symposium on Principles of Programming Languages*, pages 187–199, 1981.