Fast Type Reconstruction for Dynamically Typed Programming Languages

Frédéric Pluquet Antoine Marot

Université Libre de Bruxelles
Computer Science Department
Faculty of Sciences
fpluquet@ulb.ac.be - amarot@ulb.ac.be

Roel Wuyts

IMEC, Leuven and Katholieke Universiteit Leuven
roel.wuyts@imec.be

Abstract

Type inference and type reconstruction derive static types for program elements that have no static type associated with them. They have a wide range of usage, such as helping to eliminate the burden of manually specifying types, verifying whether a program is type-safe, helping to produce more optimized code or helping to understand programs. While type inference and type reconstruction is an active field of research, most existing techniques are interested foremost in the precision of their approaches, at the expense of execution speed. As a result, existing approaches are not suited to give direct feedback in development environments, where interactivity dictates very fast approaches. This paper presents a type reconstruction algorithm for variables that is extremely fast (in the order of milliseconds) and reasonably precise (75 percent). The system is implemented as a byte-code evaluator in several Smalltalk environments, and its execution speed and precision are validated on a number of concrete case studies.

Categories and Subject Descriptors D.2 [*Software Engineering*]: D.2.3 Coding Tools and Techniques

General Terms Performance, Experimentation, Algorithms, Human Factors

Keywords Type Reconstruction, Type Inference, Dynamic Programming Languages, Development Environments

1. Introduction

Type inference and type reconstruction are both processes of finding types for a program within a given type system [3]. The difference between the two lies in the amount of information they have available. *Type reconstruction* deals with programs where no type information is available, while *type inference* deals with programs that contain types in certain (otherwise ambiguous) places. Note that both of these differ from *type checking*, where the goal is to verify whether the given static types in a program are correct, not to find types.

Type inferencers exist for several statically typed languages, for example for Haskell [15], ML [9], OCaml [4], Scala [12], or Java [16]. They employ sophisticated algorithms to infer types for programs that contain at least some static type annotations to resolve ambiguities or help developers.

One obvious application of type reconstruction is for dynamically typed object-oriented programming languages, such as Smalltalk, Python, Perl or Ruby. After all, the lack of static types has been perceived as one of the drawbacks of dynamic languages, since it can inhibit program understanding, limits the usability of refactorings, and means that certain optimizations cannot be done. Research in this area can be divided in two broad groups. One group of algorithms does *whole program analysis*, constructing the types for every possible construct in the entire program. The problem with such approaches is that they do not scale very well [13]. Another group of algorithms is demand-driven: it will only look for the type for one specific construct. Demand-driven approaches scale better, since they do not necessarily have to treat the whole program. Moreover, they have a second advantage which is that they can be chosen to be either more precise or faster [13], by simply running longer and traversing more of the program.

Most work on type inferencing tries to be as exact as possible in its result, at the expense of execution speed. This makes sense when the results are used for program optimization or in the context of refactorings. But the situation is different when the results are for better program understanding. In that case, the feedback needs to be integrated in a development environment, requiring speeds in the order of milliseconds. The goal in this context is therefore to trade precision for execution speed, and almost none of the algorithms we looked at did this. A notable exception, Chuck [13], is a demand-driven approach implemented in Smalltalk that uses subgoal pruning. However, when set to be very fast, we found its precision to be too low to be really useful.

This paper presents an algorithm that reconstructs the types of variables (instance variables, temporary variables, arguments and return of method) in the order of 3,5 milliseconds, while still offering a precision usable for program understanding. This performance is achieved by two means. First of all, when we want to find the type for a variable, we only use information found in the methods of the class where that variable is defined, without even taking subclasses into account. Hence we severely restrict the part of the program we need to look at, much more so than in any other approach that we know of. Secondly, we use a number of heuristics to improve the precision.

We are fully aware that this approach might be perceived as 'naive', 'dreadfully simple' or even 'heresy' by some, yet this was intentional. For program understanding purposes in an interactive environment we are happy with the precision of the system, given its performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DLS'09, October 26, 2009, Orlando, Florida, USA.

Copyright © 2009 ACM 978-1-60558-769-1/09/10…$10.00

The approach has been implemented and is freely available for different Smalltalk environments, more specifically for *Cincom VisualWorks*[1] (since 2005), *Squeak*[2] (since 2006) and *Squeak/Pharo*[3] (since 2009). This allowed us to test the heuristics proposed for execution speed and accuracy on real-world case studies. We found that the precision is typically around 75 percent, with execution speeds in the order of 3.5 milliseconds. We were therefore able to integrate the approach in the development environment for showing the types of variables in a special pane, for ameliorating code completion, and for providing type-related feedback.

The rest of the paper is structured as follows. Section 2 gives a high-level overview of our approach. Section 3 has a detailed look at how *interface types* are extracted by taking into account the messages that are being sent to a variable. Section 4 shows how to extract *assignment types* by determining the type of expressions in assignments. Section 5 discusses how to merge interface types and assignment types into a final set of types. Section 6 shows how to apply our technique on other kinds of variables. Section 7 shows how relations between variables are used to extract more information. Section 8 shows the important parts of the implementation. Section 9 validates our approach on a number of concrete cases. Section 10 discusses some of the results found in the validation. Section 11 shows a number of type-oriented tools that we or others have implemented based on these results. Section 12 discusses related work, while Section 13 concludes.

2. Overview

The approach outlined in this paper reconstructs the types of variables, i.e., instance variables (attributes) of classes, temporary variables, arguments and return types of methods. It does not look into class variables or global variables because conceptually they are very close to the other variables but would require more implementation work in the tool.

For explaining the algorithm we first show the basic version of reconstructing the type of a single instance variable of a class, without taking advantage of relations between variables through the call graph. Extending this scheme for other variables is later on explained in Section 6, and how to finally take advantage of relations between variables is explained in Sec. 7.

Our type-reconstruction algorithm can be decomposed into three phases:

1. **interface type extraction** This phase reconstructs the types according to the messages that are being sent to the variable in the class where it is defined. This is done in two steps: first of all the set of messages sent to the variable is collected. Secondly we look through the system and find all types that understand this set of selectors. The output of this phase are the *interface types*.
2. **assignment type extraction** This phase reconstructs the types by looking at assignments made to the variable in the class where it is defined. It collects all right-hand sides of assignment expressions involving the variable, and applies a series of heuristics to find the type results of these expressions. These are then collected in the *assignment types*.
3. **merging**: This phase takes the interface types and the assignment types as input, and merges them into the final type results for the variable.

The following sections explain these phases in detail. Throughout the explanation we use a running example in Smalltalk, that is deliberately kept small and easy. The source of the toy example is shown in Figure 1. It consists of a class `DateWrapper` that has one instance variable (`date`), and five methods: `initialize` (assigns a default date), `date` (getter method), `date:` (setter method), `printOn:` (provides a textual representation of the wrapper) and `<` (a comparison method).

```
Class DateWrapper
   superclass: Object
   instance variables: 'date'.

DateWrapper>>initialize
      " I Initialize my date with the current date."
   date := Date now.

DateWrapper>>date
      "I return my date"
   ^date

DateWrapper>>date: aDate
      "I set my date"
   date := aDate

DateWrapper>>printOn: aStream
      " I print a textual description on the argument aStream"
   aStream
      nextPutAll: 'Wrapped date: ';
      nextPutAll: date weekday;
      space;
      print: date.

DateWrapper>> < aDateWrapper
      "I compare my date with the date of my argument, aDateWrapper"
   ^self date < aDateWrapper date
```

Figure 1. Source code of the DateWrapper class, used as example throughout the paper.

3. Interface Type Extraction

This phase reconstructs the types according to the messages that are being sent to the variable. This is done in two steps: first of all the set of messages sent to the variable is collected. Secondly we look through the system and find all classes that understand this set of selectors. The output of this phase are the *interface types*.

3.1 Constructing the Interface

To find the interface for a variable (all the messages sent to that variable), we simply traverse all the methods of the class where the variable is defined and collect all the messages sent to that variable. We call this set the set of *direct sends* to that variable. We also construct a second set, namely the set of messages that are being sent to a getter method for that variable. This we call the *indirect sends* for that variable. The union of the *direct sends* and the *indirect sends* gives us the interface for the variable.

Let's apply this on our example. Since in the body of the `initialize`, `date` and `date:` methods, no messages are being sent to the `date` instance variable, these methods will not contribute to the interface of the `date` variable. However, in the body of the `printOn:` method we can see that a message `weekday` is being sent to `date`. Therefore this augments the set of direct sends. Finally, in the method `<`, we can see that the message `<` is sent to `date`'s getter method[4], augmenting the set of indirect sends. The interface for

[1] `http://www.cincomsmalltalk.com/`

[2] `http://www.squeak.org/`

[3] `http://www.pharo-project.org/home`

[4] In Smalltalk arithmetic operators and comparison operators are just message sends.

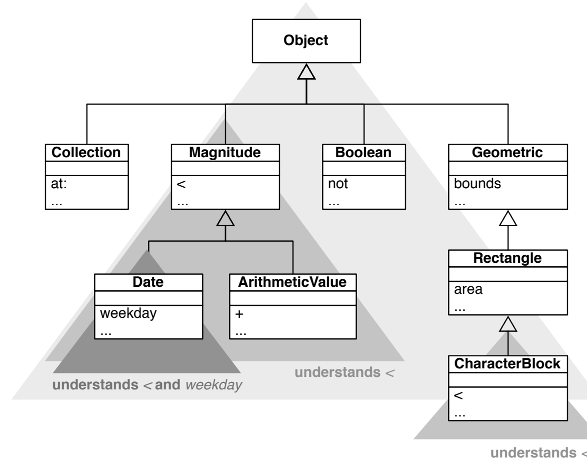

Figure 2. Finding the highest possible classes in the class hierarchy that implement the selectors `<` and `weekday`

variable `date`, after looking at all the methods, is therefore a set consisting of two elements, `<` and `weekday`.

3.2 Calculating the Interface Types

This step calculates a set of types that understand a given set of selectors. Since the selectors were those that were being sent to a variable, the types will be possible types for that variable.

Exactly how the set of types is calculated depends on the programming language at hand.

In the case of Smalltalk, types and classes align. Hence we can rephrase our problem, and calculate a set of classes that understand a given set of selectors. Given that the Smalltalk classes form a tree, with the class Object at its root, we devised an efficient iterative algorithm for doing this. The algorithm iterates over the selectors in the interface. At each iteration it finds those classes in the class hierarchy that understand all the selectors that have been processed so far, *and* that are the highest possible in the hierarchy.

More specifically, for the first selector, the algorithm will start at Object, and do a breadth-first traversal of the class hierarchy, stopping and remembering classes that implement the selector. The system then proceeds with the second selector and, for each class that implemented the first selector, finds the subclasses that implement this second selector. The result is the set of highest possible classes that implement both selectors. This is then repeated for the rest of the interface. The end result of the traversal is therefore the set of highest possible classes in the hierarchy that understand the given set of selectors.

We can illustrate this with our `date` example, and by using Figure 2 to visualize the traversal through a very small portion of the Smalltalk class hierarchy. Remember that the interface for the `date` variable was a set with two selectors, `<` and `weekday`. Therefore the algorithm takes the first element of this set, `<`. It tests whether class `Object` understands this selector, but this is not the case. It therefore looks through all direct subclasses of class `Object`, and finds class `Magnitude` that implements the required selector. Therefore it will not further enumerate the `Magnitude` hierarchy. However, it proceeds to enumerate the subclasses of the other three classes (`Collection`, `Boolean` and `Geometric`), looking for more highest possible classes that implement `<`. It eventually finds one more, in class `CharacterBlock`. To recapitulate, starting from Object we have found two classes that implement `<`. For the second selector that needs to be understood, we start from these two classes, since we need to find the highest possible classes that understand selectors `<` and `weekday`. Proceeding as before, we find one class, `Date`.

Note that it is quite possible that the classes found have subclasses themselves. This is not a problem: all of these subclasses are possible types, since subclassing induces subtyping in Smalltalk (and, in fact, most object-oriented languages).

4. Assignment Type Extraction

As indicated by the name, the assignment types are found by investigating right-hand sides in assignments to the variable of which we want to reconstruct the type, again by looking only at the methods in the class where variables is defined. The regular way of handling the right-hand side expressions would be to investigate the expressions, type them, and add type constraints on the left-hand side with these results. While this is more precise than what we do next, it also means that we might end up analyzing a big part of the system because this is a recursive process.

Therefore we use heuristics in order not to have to recurse into the actual expressions, and only use local (to the class of the method we are analyzing). These heuristics will analyze the right-hand side expression for particular patterns of which we can infer the results. Before we list the heuristics we used, let's have a look at a general assignment expression. We consider two kinds: either a direct assignment, or a call of a setter method. In Smalltalk notation, this gives:

- direct assignment: `X := Y` : assign the result of the expression Y to variable X.
- indirect assignment: `self x: Y` : calls a mutator with as argument the result of evaluating expression Y.

Regarding the type extraction we are interested in the type of expression Y, for both cases. To find this type, we experimented with different sets of heuristics. The concrete ones for the Smalltalk environment *Squeak* are shown in Table 1.

- Literal value: when a literal value is being used, the type of this literal is used as result.
- Instance creation: when instances are created of a certain class, that class is used as the (concrete) type.
- Boolean expressions: when comparison operators are being used between objects, the result is a Boolean type.

Expression	Conditions	Type
Literal	-	Literal type
Class msg:args	message is in instance creation protocol	Class
X msg Y	msg ϵ {= == < > <= >= =}	Boolean
X msg Y	msg ϵ {/ + - * abs negated reciprocal // quo: rem: \\ ceiling floor rounded roundTo: truncated truncateTo:}	Number

Table 1. Heuristics to extract types from right-hand-side expressions in *Squeak* assignments.

- Arithmetic expressions: when arithmetic expressions are being used, the type result is of an arithmetic type. Which particular arithmetic type depends on the arithmetic expressions: some can be specific to integers, for example, while other ones can be generally applicable.

Note that these heuristics need to be expressed for the language at hand. Table 1 shows how this can be done for *Squeak/Pharo* Smalltalk. Note that the heuristics used are quite general and only need slight adaptation when used in the *Cincom VisualWorks* or even other programming languages.

Note also that these remain heuristics. For instance, if one defines a binary plus method on a non numerical object the heuristic will add an arithmetic type to the assignment types, which is clearly incorrect. As mentioned before, doing more analysis of the receiver could improve the precision, but would be costly.

We can now apply the assignment type extraction on our example. We see that there is a (direct) assignment expression in method `initialize`, with `date` as left-hand side. The right-hand side is an expression that creates an instance of class `Date`. We therefore remember `Date` as possible type. Further on we can see another assignment in the setter method for `date:`. However, from that right-hand side we do not extract any information, since we remain local. Hence the assignment types for instance variable `date` of class `DateWrapper` will be a set containing the single type `Date`.

5. Merging the Types

Once we have extracted the interface types and the assignment types, their results need to be merged. This is not a trivial step: the types found in the set of interface types can be subtypes of the types found in the set of assignment types, or vice versa.

We also have to deal with *data polymorphism*, which means that a variable can be assigned values of different types. This is already taken into account in a number of approaches, such as in DCPA [16] to which we come back in Sec. 12. In dynamically typed languages it is not uncommon for variables to hold values of different unrelated types, so the merging has to take care not to remove these possibilities. Put in other words: it is quite possible that the reconstruction results in a set of unrelated types, and that the result is valid (while it would not be in most type systems, that expect a type relation between these reconstructed types).

There is one other difference between interface types and assignment types that needs to be taken into account before we show how both can be merged. The interface types that are reconstructed are *abstract types*: they indicate that at runtime the variable is allowed to be any subtype of that type. In our `DateWrapper` example, as far as the interface types are concerned, the `date` variable can hold any object as long as it understands `weekday`. These could be instances of the class `Date`, or might be subclasses of class `Date` (since subclassing induces subtyping). The assignment types on the other hand are *concrete* types: at runtime there will be a moment in the execution where the variable holds an instance of the class corresponding to the type.

Take for example the pieces of Smalltalk code shown in Figure 3. They all show an assignment expression and a message send.

`x := 1.`	`x := Magnitude new.`	`x := 1.`
`x + y.`	`x weekday.`	`x at: 5.`
(1)	(2)	(3)

Figure 3. Three pieces of code to illustrate merge difficulties.

As we have seen before, the assignment expression will be used to calculate the set of assignment types, while the message sends will be used to calculate the set of interface types. We can have the following four possibilities, the first three of which correspond with their respective number in Figure 3:

1. **assignment type subtype of interface type**: In this scenario, an interface type is reconstructed, an abstract type for which several concrete (sub)types are plausible at runtime. The assignment type is found to be one of these types: it is a subtype of the interface type. In the left column of Figure 3 we see that the interface type will be `Number`, while the assignment type is `SmallInteger`, a subclass of `Number`. Note that this scenario is the most frequently occurring.
2. **interface type subtype of assignment type**: It can occur that the assignment type does not lie within the hierarchy defined by the interface type, but is a supertype of that hierarchy. In that case the concrete assignment type is more general than the abstract interface type. This is illustrated in the middle pane of Figure 3: the extracted type (which would be `Date`, since it implements a method `weekday`) is subtype of the assignment type (`Magnitude`).
3. **assignment and interface type unrelated**: Another possibility is related to the previous one, but here the assignment type and the interface type are completely unrelated. This is a technique sometimes used in dynamic languages. For example, one can hold a collection of objects and, when there is only one object in the collection, decide to directly hold that object instead of a collection with only one element. When doing type reconstruction this can easily lead to two different types being extracted: one a collection type, and a second, completely different type. The right side of Figure 3 shows how an assignment type could be `SmallInteger`, while the interface type would be `Collection`, two completely separated types.
4. **assignment type same as the interface type**: Both can be the same, in which case there is no need to merge.

When merging the sets of interface and assignment types, these cases need to be taken into account. We therefore tried a number of different merging approaches:

- with the *AbstractMerger* the result of the first case in the example will be the interface type, since it is the most abstract.
- the *ConcreteMerger* will favor the assignment type for the first case in the example. The motivation for this choice is that it is very likely that the concrete type will indeed be the one assigned, given that it corresponds to the active type. However,

```
Class TextBlock
    superclass: Object
    instance variables: 'myChar'.

TextBlock>>areaOf: aRect
    "I return the area of my argument"
    ^aRect area

TextBlock>>defaultSquaredArea
    "I return the squared area of a new CharacterBlock"
    ^(self areaOf: CharacterBlock new) squared

TextBlock>>newArea
    "I return the area of a call to putNewChar"
    ^self areaOf: self putNewChar

TextBlock>>putNewChar
    "I return myChar, possibly replacing it with a new CharacterBlock
      instance"
    | old |
    old := myChar.
    old bounds ifNil: [myChar := CharacterBlock new].
    ^old
```

Figure 4. Source code of the TextBlock class

this is a heuristic, and some valid types might be omitted because of it.

- the *AssignmentsFirstMerger* is a refinement of the *ConcreteMerger* and therefore also favors the assignment types over the interface types. However when there are assignment types and all interface types are subtypes of the assignment types, the result will be the assignment types (and the found interface types are not used). It only considers the interface types when there is no assignment information, or when there are interface type that are not related to the assignment types.

In the experiments we show the results for the *ConcreteMerger* and *AssignmentsFirstMerger*. We omitted the results for the *AbstractMerger* due to space reasons and because they are less useful for program understanding because their precision is lower.

6. Beyond instance variables

As mentioned in Sec. 2 we decided to first explain the basic working of our algorithm focussing on instance variables. In this section we show how the same principles apply to other kinds of variables: temporary variables, arguments and returns of methods.

The extraction of the interface types and assignment types of temporary variables is the same as what is explained for instance variables, except that we stay in the method containing this temporary variable.

Fig. 4 shows some Smalltalk code with more interesting relations between variables than the example used previously. The implementation does not implement anything particularly useful. It defines a class `TextBlock`, subclass of class `Object` and with one instance variable, `myChar`. It has four methods:

- `areaOf:` returns the area of its argument,
- `defaultSquaredArea` squares the result of calling `areaOf` with a newly created CharacterBlock object,
- `newArea` passes the result of calling `putNewChar` as argument in the call to `areaOf:`,
- `putNewChar` uses a temporary value to store instance variable `myChar` and does a conditional assignment to `myChar`.

For type reconstruction we consider an argument of a method as a read only temporary variable (no assignment to this argument is possible within the method body) without lack of generality[5]. Even though there can be no direct assignments to an argument in the body of the method we can use calls to the method to find assignment types. More specifically we consider all self sends to the method defining the argument we want to type, from methods in the same class. These self sends contain expressions we can use to extract assignments types from, in the same way as using right-hand expressions in direct assignments.

Suppose we want to type the argument `aRect` of method `areaOf:` in the source code of Fig. 4. Method `areaOf:` is called through a self-send in method `defaultSquaredArea`. The argument expression in that message can be analyzed as explained in Sec. 4 for right-hand side expressions, and will yield an assignment type `CharacterBlock`. We add this type in the assignment types set of *aRect*.

The return type of a method only has interface types. Return types are found while a message is sent directly to the result of sending another message. For instance, the message `squared` is sent to the result of `areaOf:` in method `defaultSquaredArea`. The value returned by the method `areaOf:` must therefore understand the message `squared`. We add this message to its set of *direct sends*.

7. Relations between variables

Finding the interface and assignment types is done without exploiting relations between variables. This section shows how to discover relations between the variables through the call graph, and use this to augment the interface constructed for variables as well as the assignments. This remains cheap because we still restrict ourselves to *self sends* to methods only defined in our class.

We say that two variables are related when one variable is assigned to the second one. This relation has an important property: two related variables will share the same interface types and the same assignment types. The property also implies that this relation is transitive. If a is assigned to b and b assigned to c, the three variables a, b and c share the same interface and assignment types.

Fig. 5 shows the successive results of processing the methods in order to extract the variable interface and finding the assignment types for the variables in the code shown in Fig. 4.

Each grey-filled rectangle shows the interface and assignment types found after analyzing a method (unused variables are not shown). The interface of each variable (i.e., all messages sent to this variable) is collected in the case labelled "I", under the variable name. For instance, the collected interface of the argument *aRect* after the analysis of `areaOf` contains the selector #area. The assignment types of each variable are shown in the case labelled "A". For instance, after the analysis of the first two methods, the argument *aRect* has *CharacterBlock* as assignment type. Finally a relation between two variables is shown by a double arrow. During the analysis of `newArea` we found that the return of `putNewChar` is related to the argument *aRect* (because it is passed as argument).

The bottom grey rectangle shows the complete information found after analyzing all methods of the class. Because the relations are transitive we can merge the interfaces and assignment types of related variables (Fig. 6). They will therefore have the same reconstructed types.

[5] For languages where arguments are assignable we would consider them to be exactly like temporary variables.

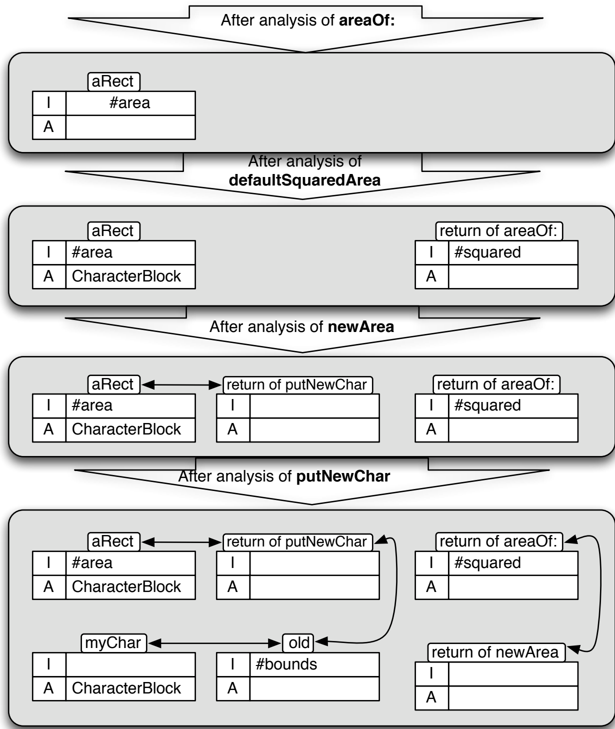

Figure 5. Variable relation information collected for the code shown in Fig. 4.

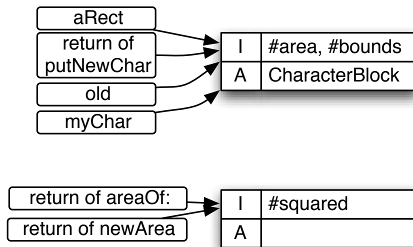

Figure 6. Merge all interface and assignment types for transitive relations in Fig. 5.

8. Implementation

The approach presented in the previous section can be implemented in different ways. One possibility is to use the abstract syntax tree, and using it to collect all the information needed. This is one solution we have implemented as an example in previous work on logic meta programming [7]. It has the advantage that it is not very hard to implement the approach and makes it easy to experiment with various heuristics. Applying the approach outlined in the paper to many languages for which it is easy to construct an abstract syntax tree is relatively straightforward. The disadvantage, however, is that the code needs to be parsed before it can be traversed, which hinders performance.

To maximize performance, we decided to implement the approach as a bytecode interpreter. Since in Smalltalk the bytecodes of all methods live in memory, the actual type reconstruction can immediately start. This means that parsing is not needed, and no extra memory is needed to store the parse tree. The price we pay for this is ease-of-implementation. While walking a parse tree is relatively straightforward, writing it as an efficient bytecode interpreter is more challenging. We constantly manipulate a stack in order to push and pop intermediate typing results, and there is not a lot of context information to work with: when we have the bytecodes of one method, we cannot easily fetch information about other methods of the same class where that method is residing in. When walking a parse tree, the data structure provides this information in a much more accessible way.

The implementation does one pass over the methods of a class, and in this one pass constructs the interface and assignment types for all variables declared in that class, exploiting the relations between variables.

As a side note it can be interesting to know that the implementation works for several Smalltalk environments (*Cincom VisualWorks*, *Squeak* and *Squeak/Pharo*). The implementations are nearly identical, with only minor differences in the bytecode interpretation[6]

9. Validation

Our approach claims to be more efficient than other approaches (with respect to execution speed and scalability), while still providing a precision which is acceptable for program understanding purposes. To validate this claim, we have done experiments on three applications for the *Squeak/Pharo* Smalltalk distribution. The first one is *Dr.Geo II*[7], a tool to interactively create geometric figures with respect to their geometric constraints. The second one is a web application framework named *Seaside*[8]. The last one is *Smallwiki*[9], a fully object-oriented wiki implementation. We used these applications because they are real applications that are practically used and not trivial, because they have extensive unit test suites, and because they are available for anyone that would like to validate our results.

9.1 Execution speed and scalability

We benchmarked our system on a complete *Squeak/Pharo* image and on the three applications described above. Table 2 shows information about the number of classes and the number of variables (attributes, temporary variables, arguments and return types of methods) defined by these classes. The following columns represents the time in milliseconds to compute the types with both mergers. For each of them 2 times are displayed: `tt` the total time, `tpv` the average time per variable.

We remark that both mergers show very similar execution times. Doing the type reconstruction for one variable takes on average 3,5 milliseconds.

9.2 Precision

This is a notoriously difficult notion to measure in dynamically typed languages because applications carry no types at all and we therefore would have to manually check the results of each reconstructed type to see if it is correct. Because this is very hard and time consuming to do for the non-trivial applications that we want to show we tackled the problem differently.

We decided to monitor the execution of the applications we chose, running (elaborate) unit test suites as well as running demo applications and interacting with the applications in general. We recorded every different type stored in variables (including argu-

[6] Most importantly, *Squeak*, *Cincom VisualWorks* and *Squeak/Pharo* have different byte codes for block closures.

[7] `http://wiki.laptop.org/go/DrGeo`

[8] `http://www.seaside.st/`

[9] `http://smallwiki.unibe.ch/smallwiki/`

Application	# classes	# variables	ConcreteMerger		AssignmentsFirstMerger	
			tt	tpv	tt	tpv
All system classes	2830	112247	355209	3,16	354835	3,16
DrGeoII	80	2478	8472	3,42	8456	3,41
Seaside	124	3437	14864	4,32	14282	4,16
SmallWiki	100	2140	6703	3,13	6439	3,04
Average	-	-	-	3.50	-	3.44

Table 2. Type reconstruction time in milliseconds (tt: total time, tpv: time per variable)

ments and return types) of classes of the applications. The (concrete) type information retrieved this way may be incomplete but we believe that if the execution of the application covers enough different scenarios the recorded types should be near completeness and enough to validate the results of our static type reconstruction.

Before we delve into these results we first define our notion of precision.

9.2.1 Notion of correctness

Before defining what we consider as a *correct*, *partially correct* or *incorrect* result for the reconstructed types we need to define some properties.

Let R_i be the set of reconstructed types for a variable i and C_i the set of dynamically recorded types of the same variable.

Definition R_i *covers* C_i iff for each $c \in C_i$ there is a $r \in R_i$ such that r is equal to c or a supertype of c.

Definition R_i *is focused on* C_i iff for each $r \in R_i$ there is a $c \in C_i$ such that r is equal to c or a supertype of c.

Definition R_i *is too general for* C_i iff `Object` $\in R_i$ and `Object` is not the smallest common supertype (with respect to the type hierarchy) of the elements of C_i.

We remind that by construction our type reconstruction ensures that if `Object` is in the reconstructed types, it is actually the only type in that set.

We have now the three properties we need to classify experimental results as *correct*, *partially correct* or *incorrect*.

Definition R_i *is correct for* C_i iff R_i covers C_i, is focused on C_i and is not too general for C_i.

Definition R_i *is partially correct for* C_i iff R_i covers C_i, is not too general for C_i but is not focused on C_i.

Definition R_i *is incorrect for* C_i iff R_i does not cover C_i.

Definition *There is no enough information* iff R_i is too general for C_i.

9.2.2 Experimental Results

The percentage of (partially) correct and incorrect reconstructed types is given in Table 3. Two of the mergers explained in Sec. 5 were used in order to compare them. We see that in around 75% of the cases, the system finds correct types with both mergers (possibly including incorrect ones for partially correct results). The difference between the mergers lies in the proportion of partially correct results which is higher with the assignments-first merger. This is perfectly understandable since the concrete merger give the priority on the interface types found but will still include results from assignment types. Its results will thus include all the classes (with some unrelated to the actual types) sharing the same interface parts while the assignments-first merger will typically have less classes (the ones assigned to the variable).

We say that a reconstructed type is *optimal* if it is the smallest type (with respect to the type hierarchy) compatible with the recorded types. It is interesting to see in Table 3 that in around 80% of the cases the reconstructed types (in correct or partially correct results) are optimal. This is positive because it means that when correct results are found they are not too abstract.

The fourth entry of Table 3 shows the percentage of cases in which the type reconstruction did not find enough information and returned `Object`. It is about 25% of cases.

The percentage of incorrect reconstructed types is very small (about 2 percent). We discuss these cases in the next section.

10. Discussion

Doing detailed analysis of our experimental results of the Seaside application revealed two interesting reasons for the cases of incorrectness of our type reconstructor.

Looking through the incorrect results found by our type reconstructor we stumbled on the class `WAAnchorTag`, and more particularly its instance variable `url`. The reconstructed type for this variable was `WAUrl`. However the concrete types found during test runs and interacting with the program showed that this variable indeed stored `WAUrl` objects, but also `ByteString` objects (which have no relation to `WAUrl` objects)! Following our correctness rules this case is therefore incorrect. Manually inspecting the implementation in detail reveals that this is actually a bug in Seaside because the code clearly assumes `WAUrl` objects.

Another case we found has to do with reflection. Our type reconstruction has no problems with reflection: reflective expressions will simply not be taken into account while regular usage of the variable will still result in type reconstruction. In one case, for example, a reconstructed type was found but the concrete types revealed that there was another type that was also used. Investigating why this did not show up in the reconstructed type we noted that the code first checked whether a message was understood before actually sending it. Therefore the resulting implementation is correct, and it shows that we find an approximative result.

11. Using the Type Information

Several tools benefit from the types that can be reconstructed from the bytecodes, and we implemented some examples in *Cincom VisualWorks*. First of all we integrated an information pane directly in the Smalltalk development browser, which is shown in Fig. 7. It shows, for each instance variable of a class, the extracted types. The type information can be selected and explained: this shows for an instance variable the interface types and the assignment types, helping to gain a better insight in the code and the reconstructed type.

Application	Kind of variables	Concrete Merger Correct	Part.Correct	Incorrect	No enough information	Optimal
DrGeoII	Attributes	30,89	37,40	0,81	30,89	80,95
	Other	59,12	20,09	1,85	18,94	83,67
	All	52,88	23,92	1,62	21,58	83,14
Seaside	Attributes	37,16	35,78	2,29	24,77	80,50
	Other	56,30	13,94	2,13	27,64	79,93
	All	53,49	17,14	2,15	27,22	70,63
Smallwiki	Attributes	45,38	31,13	3,77	19,81	65,43
	Other	54,19	17,41	1,18	27,23	80,80
	All	53,10	19,08	1,49	26,32	78,82

Application	Kind of variables	AssignmentFirst Merger Correct	Part.Correct	Incorrect	No enough information	Optimal
DrGeoII	Attributes	47,97	19,51	1,63	30,89	83,14
	Other	61,89	17,09	2,08	18,94	85,09
	All	58,81	17,63	1,98	21,58	84,00
Seaside	Attributes	53,67	19,27	2,29	24,77	80,50
	Other	58,90	11,02	1,81	28,27	80,18
	All	58,13	12,23	1,88	27,76	80,23
Smallwiki	Attributes	66,04	10,38	3,77	19,81	65,43
	Other	59,03	11,65	0,92	28,40	81,11
	All	59,89	11,49	1,26	27,36	79,07

Table 3. Precision experiment results.

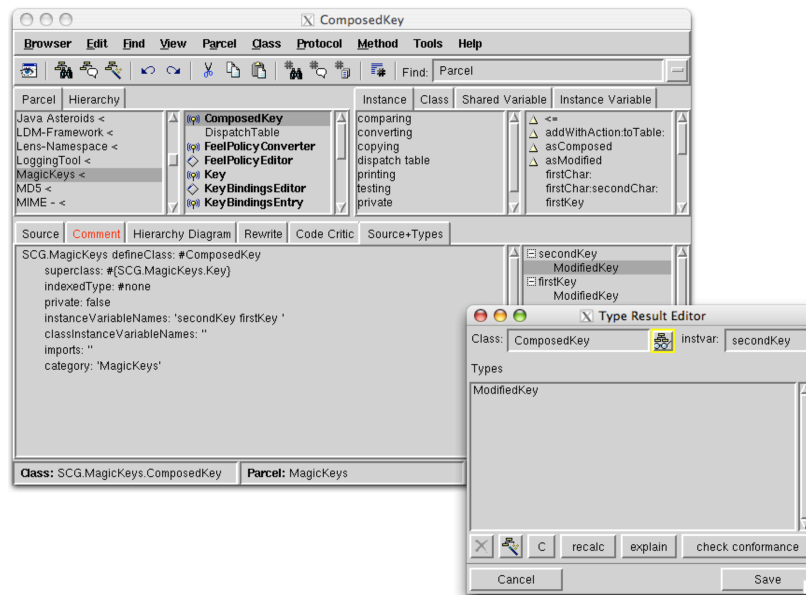

Figure 7. Smalltalk SystemBrowser extended with a type information pane and the type explanation dialog for a selected type.

A second tool we made, shown in Fig. 8 uses the types to warn about certain type errors. After a method is being accepted into a class[10], a type check is launched that compares the type of each instance variable of the class before the method was added with types extracted after the method was added. When type conflicts occur, a window that shows type violations is being updated. Using the same explanation tools as described in previous paragraphs, the user can get an idea about the conflict and resolve it.

Note that there is an important difference with type errors returned in statically typed languages: in our case the program is compiled and installed. A type error in a statically typed language will not allow the program to be compiled at all. We intentionally decided to generate only a warning to the user (we could as easily have opted for an approach to raise an error that would have prevented the method to be installed). We chose to generate the warning to not hinder the development process, in accordance with other Smalltalk tools.

[10] Smalltalk uses incremental compilation, and therefore this check happens immediately after a method is being compiled. In environments that do not support incremental compilation, the checks can be done when a class is compiled.

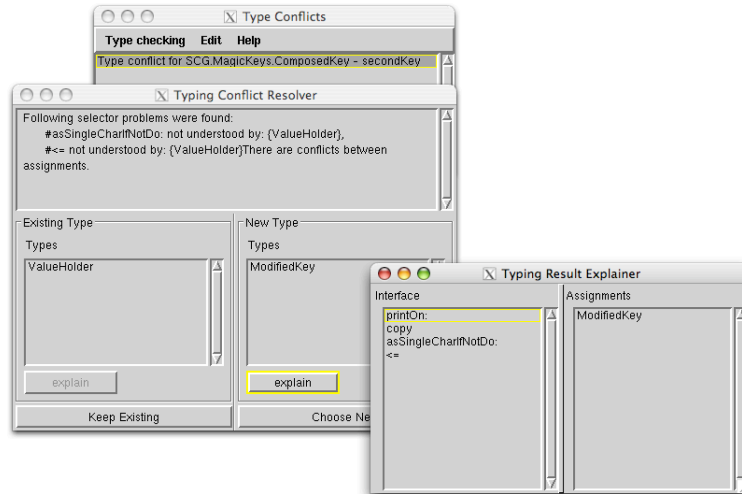

Figure 8. Type error warning application. Clicking a type problem brings up a dialog box that can be used to define a new static type and see the reason for the warning.

A third tool that has been implemented is a code completion tool (called *eCompletion*[11]). Without type reconstruction the completion can only use a global symbol table containing all the class names, method names, etc.. Hence, to reuse the example we introduced before, when one would ask to complete the name of a method starting with `'in'` being sent to the `date` instance variable, one would get a huge list. Using the reconstructed type *eCompletion* can limit this list to the set of messages understood by the `Date` class that start with `'in'`. Robbes *et al.* [10] also used our type reconstruction system in order to improve the performance of a completion tool based on the program history.

12. Related Work

First of all we again stress that our main goal was to have a very fast approach, with reasonable precision. In general this makes our reconstructor very different from most other type inference or reconstruction approaches that focus on precision. The reason is the difference in application. While we use our approach in development environments, and therefore embrace a limited precision on-demand approach, other approaches are part of compilation or optimization approaches and can spend some more time to gain in precision.

In the rest of the section we first have a look at type inferencing approaches in general. Then we look at a number of tools and approaches specifically for dynamically typed languages.

12.1 General approaches

One of the basic techniques for type inferencing object-oriented programs is the Cartesian Product Algorithm approach (CPA) [1], an extension of the Palsberg and Schwartzbach type inferencer [8]. The local constraints in CPA correspond more or less to the interface and assignment types. However, CPA goes further and also looks at non-local constraints to gain in precision. While it deals fairly well with parametric polymorphism, it is not very good at handling data polymorphism [16].

Extensions of CPA exist, such as DCPA [16], that is more precise yet has execution times comparable to CPA. The execution times lie in the order of tens of seconds for bigger case studies, not milliseconds. Moreover, because our approach is on-demand it will always take in the order of 3,5 milliseconds for a variable, while these approaches will take much more time for the first variable (or when code has changed and it needs to be re-run).

12.2 Dynamic Language Tools

There are different versions of Strongtalk that are relevant for the research in this paper. First of all there is a version of Strongtalk, which we will call Strongtalk 1, with a well-known paper published in Ecoop [2]. Strongtalk 1 is primarily a type-checking approach that is downwards compatible with Smalltalk and therefore allows to run untyped Smalltalk code. Static types can then be added at will and are then taken into account for type checking. Strongtalk 1 does not do actual type inference.

A start-up company reimplemented Strongtalk 1, resulting in what we will call Strongtalk 2, which was later on released as open-source project[12]. Strongtalk 2 contains a strong, static type system for Smalltalk that is both optional and incremental. We wanted to run the same experiments we did in *Squeak/Pharo* in Strongtalk 2 in order to directly compare the results but have to port the applications to use the Strongtalk 2's different class libraries. Alternatively we could port our approach to Strongtalk, which would not be too hard if we implementation it on the AST instead of on the bytecodes. We could then reconstruct the types of code for which we know the types (but not using that information), and compare the results with the types. We are continuing the investigation.

Starkiller [11] is a type inferencer and compiler for Python, with a type inference algorithm based on CPA as explained in the previous paragraph. Starkiller showed that with an implementation of Fibonacci it was able to significantly outperform other compilers and optimizers for Python, due to its type inference. Execution times for the compiler or inferencer are not given, and it is unclear how it would fare on other (more complex) benchmarks. Because the code was never released we were not able to experiment with it ourselves. Due to the fact that it is based on CPA we assume it inherits the same advantages and disadvantages that we discussed previously.

[11] by Ruben Bakker (http://uncomplex.net/ecompletion/)

[12] `http://www.strongtalk.org/index.html`

Cincom VisualWorks uses bytecode analysis to infer the types of instance variables and use this information when generating a default comment for a class. This is related to what we do, but their approach is simpler because it is limited to instance variables, it only takes messages sent to instance variables into account and their implementation is about a factor of 5 slower.

Chuck is a type inferencer [14, 13] that uses *subgoal pruning*, an approach that, like ours, also trades precision for scalability. Chuck first computes an initial set of tables before type inference is done, which can take quite some time. Note that the tables are kept up-to-date incrementally, and therefore only need to be calculated once. We are investigating how our approach could be used to substantially speed up their table calculation, because it is similar to what we do.

Diamondback Ruby (or DRuby) [5] is a tool that aims to integrate static typing into Ruby. It uses a constraint-based approach. From the experiments it seems to be quite fast on smaller Ruby programs (about 2 seconds on programs of 100 to 500 lines of code on a machine comparable to ours but with more memory), but we process such programs in the order of milliseconds. The largest program tested, of about 900 lines of code, takes 36 seconds but this discrepancy is not explained and it is unclear whether this is due to scaling issues or some particularly hard to analyze code. Given the fact that this system is available we plan to apply DRuby on bigger case studies to investigate this in more detail.

13. Conclusion

The problem we tackled with this paper is how to reconstruct types in dynamically typed languages to ease program understanding. Because we want to be able to integrate the results in interactive development tools we needed a fast approach and looked for a balance between execution speed and precision. Therefore we developed a type reconstruction approach that relies on extracting interface types by looking at the messages sent to a variable only in the context of the class where the variable is used, and merging these results with types found by heuristics working on right-hand side assignment expressions. The contribution is not of a theoretical nature, but nevertheless shows that taking a very pragmatic and limited, and hence fast, approach is enough to give usable results for a number of applications of which we showed some. Our current approach is very fast. It is actually so fast that we plan to extend it with some more advanced reasoning. As long as we remain with an execution speed in the neighborhood of tens of milliseconds the interactivity will not be harmed and we can try to increase precision. We are also investigating whether a variant of our approach that just uses heuristics that do not introduce false positives cannot be used as first (filtering) pass for classical type reconstruction techniques. Last but not least we plan to use our technique on other dynamically typed languages like Ruby and Python, because we strongly believe that this is easy to achieve and might benefit developers in these languages as well. We are even considering trying this on statically typed code and directly compare the execution speeds and precision with for example the DCPA results [16].

Acknowledgments

We thank the users for their valuable feedback on RoelTyper, the tool that implements these ideas and that has been freely available for *Cincom VisualWorks*, *Squeak* and *Squeak/Pharo* for many years. We also thank Gilad Bracha, Stéphane Ducasse, and Wim Lybaert for their comments and discussions on early drafts of this paper.

References

[1] O. Agesen. The cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, Aug. 1995. Springer-Verlag.

[2] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, Oct. 1993.

[3] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[4] Chailloux. *Développement d'applications avec Objective CAML*. O'Reilly, 2000.

[5] M. Furr, J. hoon (David) An, J. S. Foster, and M. Hicks. Static type inference for ruby. To appear in OOPS track, SAC 2009, 2009.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[7] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.

[8] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 146–161, Nov. 1991.

[9] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[10] R. Robbes and M. Lanza. How program history can improve code completion. In *ASE: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 317–326, 2008.

[11] M. Salib. Starkiller: A static type inferencer and compiler for python. Master's thesis, Massachusetts Institute of Technology, may 2004.

[12] The scala programming language. http://lamp.epfl.ch/scala/.

[13] S. A. Spoon and O. Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[14] S. A. Spoon and O. Shivers. Dynamic data polyvariance using source-tagged classes. In R. Wuyts, editor, *Proceedings of the Dynamic Languages Symposium'05*, pages 35–48. ACM Digital Library, 2005.

[15] S. Thompson. *Haskell: The Craft of Functional Programming (2nd edition)*. Addison Wesley, Reading, Mass., 1999.

[16] T. Wang and S. F. Smith. Precise constraint-based type inference for java. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings ECOOP '01*, volume 2072 of *LNCS*, pages 99–118, Budapest, Hungary, June 2001. Springer-Verlag.