An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types

Nicolás Passerini
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
UTN – F.R. Buenos Aires
npasserini@gmail.com

Pablo Tesone
Universidad Nacional del Oeste
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
tesonep@gmail.com

Stephane Ducasse
RMoD Project-Team, Inria
Lille–Nord Europe / Université de Lille
stephane.ducasse@inria.fr

Abstract

Dynamically typed languages promote flexibility and agile programming. Still, their lack of type information hampers program understanding and limits the possibilities of programming tools such as automatic refactorings, automated testing framework, and program navigation. In this paper we present an extensible constraint-based type inference algorithm for object-oriented dynamic languages, focused on providing type information which is useful for programming tools. The algorithm is able to infer types for small industrial-like programs, including advanced features like blocks and generic types. Although it is still an early version, its highly extensible and configurable structure make our solution a useful test bench for further investigation.

Categories and Subject Descriptors F-3.3 [*Logics and meanings of programs*]: Studies of Program Constructs

General Terms type inference, dynamic languages, concrete types, abstract types

Keywords dynamic languages, type inference, ide, tools, automatic refactorings, type annotations, Smalltalk, Pharo.

1. Introduction

Dynamically-typed languages have many advantages, such as a strong flexibility, reduced development time and code size [Tra09]. Still their lack of type information hampers program understanding and limits the possibilities of programming tools such as an automated testing framework [Ahm13, GKS05], program navigation and refactorings [Opd92] or smart suggestions [RL13]. In addition type information improves programmer understanding by showing method parameter type, or the messages that can safely be sent to a variable. Such properties are important when a programmer should join in an existing team with a large code base to maintain. The same difficulties appear when learning how to use a new API. Types provide an important conceptual framework for the programmer, useful for program design, maintenance and understanding [Bra04].

There are many different approaches (such as type inferencing [Suz81, GJ90, Unt12, HPHf11, OPSb92] or gradual typing [ACF+13]) to provide static type information to dynamic languages. However, industrial programming environments for object-oriented dynamically-typed languages take little or no advantage of these ideas. One of the causes for this is that frequently these approaches have somewhat limited applicability on industrial programming, because they work on a limited version of a general purpose programming language [Gar01, SS04] or even define their own language specifically for that purpose [PS91, Hen94, OPSb92]. With Gradual Typing, other approaches propose to modify the language semantics and to add type annotations to the language itself [Gra89, ST07, WB10]. In addition many solutions are partial [Suz81, PS91] and do not cover complex part of the language such as closures. Finally there are approaches which are successful for program optimisation and delivery, but its performance would be unacceptable for interactive uses such as automatic refactorings and code completion [Age96].

The programming environment chosen for this development is Pharo[1]. Pharo is an Open Source programming language and environment. It is inspired in Smalltalk and it is used in many industrial applications.

[1] http://pharo.org/

The objective of this work is to create a *pluggable type system* [Bra04] which can improve a programming environment by feeding the programming tools with type information. Our main contributions are:

- a prototype implementation of an algorithm which computes type information for an existing Pharo program, supporting some of its most complex features, like block closure and generic data types,
- a strategy for combining both *concrete* and *abstract types*,
- a modular, configurable and extensible algorithm which provides a framework for future analysis and, for example, tune it alternatively for better performance or precision.

The rest of this paper is structured as follows. In Section 2 we present the problems arising from the lack of type information in dynamic languages. Section 3 proposes an inference algorithm to get this information from an existing code base without modifying it, also explaining how the different type variables and constraints are built and how they interact. Section 4 explains how we address some advanced features: blocks and generic data types. In Section 5 we show a small example of the type information that can be obtained using our algorithm on a small but non-trivial program. Section 6 analyses the pros and cons of our solution, while Section 7 compares our proposal with other type inference alternatives. Finally, we summarize our contributions in Section 8, along with some possible lines of further work derived from this initial ideas.

2. The Challenge of Lack of Type Information

Type information can be useful to improve programming environments (IDEs). For example, an automatic method rename tool could use type information to select more precisely the message sends that should be updated when a method is renamed. Also type information can be used to show the programmer a list of messages that can be sent to a variable or what kind of objects a method expects to receive as parameter. Moreover, code navigation can be improved for example if the IDE can provide a more acurate set of the possible receivers of a message.

In several languages, the programmer is forced to provide type information, for example associating each variable with a *type annotation*. In these languages, static type information is available and IDEs can take advantage of it. *Dynamically-typed languages* such as Pharo, Javascript or Ruby allow the programmer to avoid type annotations. This simplifies prototyping and modelling, albeit sometimes limiting the power of some programming tools.

A *type inferer* is a tool that automatically computes types for some part of a program which lacks type annotations. Statically typed languages such as Haskell or ML have a large experience with type inference [Hud89]. However in those languages the type system is integrated with the language, *i.e.,* we are unable to run a program with a type error. It is a much more difficult task to apply type inference to a program written with no notion of typing, which is the case in object-oriented dynamically typed languages.

Dynamic object-oriented languages pose several difficult challenges to type inference. *Subtype polymorphism* allows a variable to hold objects of different types (and not related types). For this reason, when a message is sent to an object, it is not simple to see which method is going to be executed. Also, for container objects such as collections it is not sufficient to know its type (*e.g.,* Set or OrderedCollection), instead we need to have information about the type of its elements. This capability of an object to be instantiated using different types for its instance variables is known as *generic data types*. [CW85]

The purpose of this work is to build a type-inferer system that can be practically used to improve programming tools, such as automatic refactorings, code navigation and smart suggestions. To achieve this objective, a type-system should comply to the following:

- It should *work on existing real-world programs*, *i.e.,* we may not restrict the use of the language or create an ad-hoc language that fits the needs of our type-system. Also, we should avoid requiring the programmer to add type annotations to its program in order to use our tools.
- The type-inferer should be *responsive*, *i.e.,* it should be able to run while a user is coding, for example each time he compiles a method. An execution time of even a few seconds is already unacceptable. To fulfill this goal, we think that a type system should be able to work *incrementally*, *i.e.,* when a method is modified, rebuild type information only for that method and re-use all the information computed for the rest of the system.

On the other hand, since our goal is restricted to feed tools with type information, we do not intend to detect type errors.

A usual purpose of a type system is to detect programming errors [Mil78]. Though, dynamic language programs frequently make use of programming idioms that are be very hard or even impossible to type-check, as those described in the work of Allende *et al.,* [ACF+13], such as the use of the same local variable to hold non related objects, or any use of meta programming techniques. Having to avoid those idioms to conform to a static type system, would cut off a significant portion of the sense of a dynamic language. Therefore, dynamically typed language programmers prefer other tools to detect programming errors, such as unit testing [GNDc04, Eck03, Mar03].

For these reasons, our solution does not intend to detect errors in the program under analysis. Instead, we assume it correct and just try to infer type information to help the

programmer understand the program or modify it with more confidence.

Dynamically-typed languages do not provide type information for their core libraries. Since such libraries are just programs expressed in the same language, it is possible to infer their types with the same tools used to analyse individual programs. Only primitives, which tend to be scarce in most dynamic languages, require special inferencing handling because they often represent the connection to external world such as C libraries or plugins. However, because of their complexity and generality, the analysis of these libraries often will consume most of the time of a type-inferer. Since application programmers seldom modify core libraries, we choose to consider them also as primitives and feed the type-inferer with their types [Gar01].

Type systems can be characterized from concrete to abstract [Age96]. *Abstract types* specify an object's interface, while *concrete types* describe its implementation. Abstract types can be modelled as a set of messages an object should understands. Concrete types are usually modelled as sets of concrete classes. Most type-inference systems on object-oriented dynamic languages focus on concrete types, because they are useful for some purposes such as program optimisation and delivery. Also they are simpler to understand to programmers without a heavy background on type systems. However, concrete types cannot express the type of a method parameter whithout knowing all of its clients (*closed-world assumption*). A somewhat novel feature of our approach is to combine concrete and abstract type information.

3. Basic Algorithm

Our solution implements a type inference algorithm based in constraints generation which combines concrete and abstract types. It associates each expression (in our implementation each AST[2] node) with a *type variable*, it analyses the code gathering constraints for each type variable and associates each type variable with possible types so that all constraints are meet.

The constraint solving algorithm is divided into independent *tasks* which are related through a workflow. The algorithm is *extensible* and *configurable*, new tasks can be created and the workflow can be changed. In this way, our solution provides both a useful tool for type inference research and a configurable framework adaptable to different purposes.

The algorithm is *iterative*, *i.e.,* each task is executed multiple times and each time it may produce new information, based on the information obtained by other tasks or by a previous execution of the same task. After a task is executed, the workflow decides which task to execute next, depending on if the previous task was successful in obtaining more type information or not.

[2] Abstract Syntax Tree

Also the solution is capable of handling generic data types and inferring type information on independent parts of the code, allowing the use of unbound parameters in the root analysed method; in other words it can infer type information in a program without a main method. This is achieved by the combined use of concrete and abstract types. These two sources of information provides a better understanding of the analysed programs.

The input of our solution is one or more initial methods, as it was said, these methods can have unbound parameters. The algorithm will automatically select which other methods have to be analysed. The answer of our algorithm is a set of restrictions gathered into type variables, which are associated to each expression in the program. The restrictions specify the possible types of the values of the expressions, using both concrete and abstract types. We name the set of known classes that comply with those restrictions the *result* of a type variable.

Our solution allows one to manually specify the types of some methods, thus avoiding to infer their type information. In fact a few of those *type specifications* are necessary for the algorithm to work. This is the case for

- Virtual machine primitives
- Methods of generic classes, such as Collections (*cf.* Section 4.2)

However, type specifications are also useful to set boundaries to type inference. For example, an application developer could take advantage of having type specifications for the language core and other libraries he uses, as this would shorten inference time. Also type specifications can improve precision in cases where the algorithm cannot infer the most precise type.

3.1 Type Variables and Constraints

As it was said previously, each expression has a type variable associated with it. For example, for the expression: `z := x addTo: y`, the following type variables are generated: t_x, t_y, t_z and $t_{(x\ \mathsf{addTo:}\ y)}$. For each variable in the code, we generate a single type variable which is shared among all occurences of the variable. Pharo has several types of variables including class variables, instance variables, local variables, method parameters and block parameters. Also a type variable is created for the return type of each method.

The different type variables are related by constraints. There are only two types of constraints: *subtype* constraints and *receive message* constraints.

3.1.1 Subtype constraints.

A subtype constraint "t_x is subtype of t_y", written $t_x \prec t_y$ establishes that

1. If we know that `C` is a valid result for the subtype t_x, then `C` should be also a valid result for the supertype t_y.

2. If we know that the result of the supertype t_x has to understand message #m1, then the result of the subtype t_y also has to understand #m1

Therefore, these constraints can be used to propagate type information. Each time we obtain some information about a type variable, we can use it to deduce something about its subtypes or supertypes.

It is worth mentioning that subtyping is a relationship between type variables and it does not have a direct relationship to inheritance. Subtype constraints are generated in the following cases:

- **Assignment:** a := b implies that $t_b \prec t_a$.
- **Return:** If we find the expression ^a in the method #m of class C, we can deduce $t_a \prec t_{(C \gg \#m.\uparrow)}$, where $t_{(C \gg \#m.\uparrow)}$ is the type variable associated to the return type of the method.
- **Parameter passing** From the expression obj m: a and for each class C that is a valid result for t_{obj} we can deduce that $t_a \prec t_{(C \gg \#m:.1)}$, where $t_{(C \gg \#m:.1)}$ is the type variable associated to the first formal parameter of the method #m: in the class C

The first two cases occur during the analysis of the AST (*cf.* Section 3.3), while the last one is a bit more complex and therefore it can only derived by the constraint solving tasks (*cf.* Section 3.3).

3.1.2 Receive message constraint

A receive message constraint establishes that any valid result of a type variable has to understand the specified message. Moreover, it establishes relationships to other type variables which will be related to the arguments and the result type of the messages. For example, from the expression x addTo: y we deduce:

t_x *should understand* #addTo: *with type* $t_y \rightarrow t_{(x\ addTo:\ y)}$

This means not only restricting the possible results for t_x but also associating the variable t_y to the argument of the message and the variable $t_{(x\ addTo:\ y)}$ to its return type. This associations are not useful at this point, but they will become relevant once we assign t_x to a concrete type (or many) and hence #addTo: to a specific method (or many).

3.2 Type Information

All the type information collected by the algorithm is related to a type variable. For each type variable (and hence for each AST Node) we collect:

- The set with all the type variables which are direct *subtypes* of this variable.
- The set with all the type variables which are direct *supertypes* of this variable.
- The set of *messages* that the types assigned to this type variable must understand *msgs(t_x)*.
- The *minimal set of concrete types* that must fit in the type variable: *min(t_x)*.
- The *maximal set of concrete types* that could fit in the type variable: *max(t_x)*.

Subtypes and supertypes express relationships between type variables and are created by the constraint generation task (*cf.* Section 3.1). They will allow to propagate type information between type variables during constraint solving. For example if we know that $t_x \prec t_y$ and t_x can be of type SmallInteger, then t_y must also be able to hold SmallInteger's. Following the relationship in the opposite way, if t_y must understand #addTo:, then its subtype t_x must also understand it.

The set of messages is also created by the constraint generation task. Also, each message send is related to other type variables representing the arguments and the return value of this message send.

The minimal set of concrete types is a set of Pharo classes. A class C is included in *min(t_x)* each time the algorithm can find evidence that an instance of C can be the result of evaluating the AST Node related to t_x. Examples of evidence are:

- *Literal values.* For example from the expression x := 37 follows that SmallInteger $\in$ *min(t_x)*.
- *Primitive return types.* For example from x := OrderedCollection basicNew follows that OrderedCollection $\in$ *min(t_x)*.
- *Propagation due to subtype/supertype relationships.* For example, from this sequence of assignments: x := 37. y := x, we know that SmallInteger $\in$ *min(t_x)* (from the first assignment) and $t_x \prec t_y$ (from the second one). Then we can deduce that SmallInteger $\in t_y$

The maximal set of concrete types of a type variable t_x is the set of all Pharo classes that could possibly be the result of evaluating the AST node related to t_x. If a class C is not included in *max(t_x)*, it is because we have evidence that the related AST node could never be evaluated to an instance of C without producing an error. There are three ways of compute *max(t_x)*:

- *Primitive parameter types.* Primitives usually can accept only a limited number of classes as parameters, or even a single one.
- *Message sends.* If *msgs(t_x)* = {#add:, #remove: } then *max(t_x)* is the set of all classes that understand both #add: and #remove:.
- *Propagation due to subtype/supertype relationships*. For example, $t_y \prec t_x$ and *max(t_x)* = {A, B }, then *max(t_y)* $\subseteq$ {A, B }.

In our algorithm the minimal set starts empty and is enlarged as the algorithm finds new evidence. On the other

hand, the maximal set starts containing all the classes in the image[3] and it is reduced as the algorithm progresses. During type inference always holds that $min(t_x) \subseteq max(t_x)$. Since $min(t_x)$ can only grow and $max(t_x)$ can only shrink, the final type computed will be between those two limits.

3.3 Tasks

Our algorithm is divided in 6 independent tasks:

- Generate type variables and constraints for a method.
- Propagate minimal set of concrete types.
- Link a message-send with a method.
- Propagate maximal types.
- Compute max concrete types from message sends.
- Infer a minimal type from a singleton maximal type.

The following sections explain each of these tasks.

Constraint generation. This task has the responsibility of creating the type variables for a method. For each method that has to be analysed, it can proceed in two ways. If there is a type specification for that method, it generates only type variables for its formal parameters and return type, with fixed types. If a type is not present (which is the general case), the task walks through the AST of the method generating a type variable for each expression in the method. In this case, while traversing the AST, it also collects the subtype relationships between type variables, and each message sent to an expression is translated intro a constraint applied to the associated type variable, as described in Section 3.1.

This is the starting point of the algorithm and the only mandatory task. Since the input to the algorithm is a set of methods, the first execution of this will compute type variables and constraints for each of these input methods. As other tasks are executed, the algorithm will discover other methods that have to be analysed. These methods will not be analysed immediately but enqueued. The workflow has the resposibility of deciding when to re-execute this task.

Propagate Minimal Set of Concrete Types. The main objective of this task is the propagation of the minimal types detected in previous tasks. For each pair of type variables t_x and t_y such that $t_x \prec t_y$, we have to update $min(t_y)$ so that:

$$min(t_y) := min(t_y) \cup min(t_x)$$

i.e., the minimal type set of the supertype t_y is enlarged to include the minimal type set of the subtype t_x.

For efficiency reasons, the algorithm keeps track of which concrete types have already been propagated. By doing so, in each execution of the task we also are able to tell if new information has been discovered or not, which is necessary for the workflow to decide which task to run next.

[3] This is only from a theoretical poin of view. For efficiency reasons, in the actual implementation the set of all classes is never computed.

Link a message sent with a method. The main objective of this task is to link the type variables associated to a message-send with the type information associated to a specific method. To be able to do this, we have to infer the possible method that could be executed as response to the message send. Those methods are looked up in the classes from the minimal type set of the receiver. Thus, for each message-send `x m: y` and for each $C \in min(t_x)$, we look up the type variables for the formal parameters and return type of the method C≫#m and create the following constraints:

$$t_y \prec t_{(C\gg\#m.1)} \qquad\qquad t_{(C\gg\#m.\uparrow)} \prec t_{(x\ m:\ y)}$$

The use of type variables allows us to create these subtype relationships without having actual type information for the method C≫#m. If the type information is not available, we only create the type variables and enqueue the method for being processed by the constraint generation task (*cf.* Section 3.3).

As in the previous task, for each message-send, the algorithm keeps track of the concrete types of the receiver for which we already generated constraints, avoiding to process the same method twice for the same message-send and allowing the task to inform the workflow if it has make some progress.

Propagate Minimal Set of Concrete Types. This task propagates the information of the maximal types set is propagated through in a similar way as explained in Section 3.3. Still, there are two big differences in the operation of both tasks.

First, the nature of the maximal types set mandates to propagate the information in the opposite direction, *i.e.,* from supertypes to subtypes. If $t_x \prec t_y$ this means that the final result of t_x has to be included in the final result of t_y. Since $max(t_x)$ is an upper limit of the final result of t_x, it also works as un upper limit t_y.

Second, as we said before, the initial maximal types set contains all the classes in the image and it shrinks as we obtain more information. Therefore, the propagation process leaves only the concrete types present in both the type variable and its supertype:

$$max(t_x) := max(t_x) \cap max(t_y)$$

Compute max concrete types from message sends. In this task the information of the maximal concrete types are filtered using the messages sent to the type variable. Only the concrete types which implement all the messages are kept in the maximal set.

Infer a minimal type from a singleton maximal type. This is the only task in the current implementation which combines information from the minimal and maximal sets. When we find out that the maximal types set of a type variable has only one concrete type ($max(t_x) = \{C\}$), then the minimal types set should contain exactly the same type.

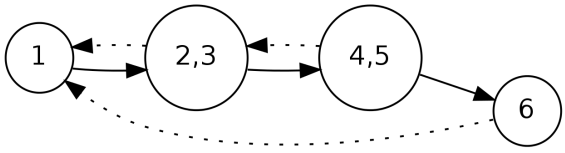
$$min(t_x) := max(t_x)$$

3.4 Coordination of tasks

The tasks of the algorithm are coordinated by a workflow. The workflow is organized in *run levels*, each run level has one or more tasks to execute. The workflow executes the run levels in a sequential way, deciding which run level to execute next depending on if the previous run level has produced new information or not.

The workflow is fully configurable, allowing to create new tasks, remove some of the current tasks and change how the tasks are organized run levels. Having a configurable workflow allows to use our algorithm as a testing bench for different configurations, comparing their performance, accuracy and fitness for different purposes. Also, different configurations could be useful for different purposes. For example, a smart suggestions tool requires a very fast type inferer, even at the cost of loosing precision, while a type inferer used for compiler optimization can be allowed to run for hours but has to be extremely precise because a wrong inference would make the program fail at runtime.

So far we have been working with single workflow configuration, which proved to be successful in typing some small but not trivial programs (*cf.* Section 5). This configuration includes all the 6 tasks described in Section 3.3, organized as described in Figure 1. In the figure each node represents a run level and the numbers in the nodes represent tasks. Each node has two outcoming links. The dotted link shows the path to be followed by the workflow in the case that the run level is successful in producing new type information. The other one is the path to follow if the run level does not find new information.



1. Generate constraints type variables for a new method.
2. Propagate minimal types.
3. Link a message send with a method.
4. Propagate maximal types.
5. Compute max concrete types from message sends.
6. Infer a minimal type from a singleton maximal type.

Figure 1. A possible workflow configuration

4. Advanced Features

One of the distinctive feature of our solution is the capability of handling blocks (also known as closures) and generic data types. Generic data types are specially useful for typing collection objects.

4.1 Blocks

A block in our type system is represented by a special type variable, which acts as a *composite* [GHJV95] type variable. Its component type variables are: the type variable associated to the return value of the block, and each of the type variables related to the parameters of the block, if the block has any.

Block type variables can be introduced either by the presence of a block literal in the AST or by a type specification. Also, when a block type variable gets involved in a subtype constraint, the other type variable is also converted to a block type variable if necessary.

When a subtype relationship is established between two block type variables, the subtyping relationship is propagated to the component variables. For example, in the following use of a block:

```
x := [ :a | a msg ] value: y.
```

In this example, some of the generated type variables are: $t_{([\ :a\ |\ a\ msg\ ])}$, $t_{([\ :a\ |\ a\ msg\ ].\uparrow)}$, t_a, t_x, t_y. And the following constraints are generated between them:

- $t_y \prec t_a$.
- $t_{([\ :a\ |\ a\ msg\ ].\uparrow)} \prec t_x$
- #msg $\in$ *msgs*(t_a)

4.2 Generic Data Types

Our type system can handle generic data types, which are also represented by special type variables. *Generic type variables* have a subsidiary type variable. For example, if the type variable t_x is infered to be an OrderedCollection (which is a generic data type), a subsidiary variable $t_{(x.\alpha)}$ is created. The subsidiary type variable will be used to compute the type of the elements of the collection.

Generic data types are introduced by type specifications, the current algorithm does no attempt to infer that a class requires to be handled as a generic data type. Still, once introduced, generic data types can be propagated through subtype constraints. As with blocks, each time a generic type variable gets involved in a subtype constraint, the related variable is converted to a generic one, if necessary. Block type variables cannot be converted into generic data type variables or viceversa. If such situation arises, we consider it an error in the program and inform it to the user. Also, a type variable's minimal type set cannot contain both collections and other non-generic concrete types.

In our system, generic data types are *invariant*, *i.e.,* if $t_x \prec t_y$, then both subsidiary variables have to be equal: $t_{(x.\alpha)} = t_{(y.\alpha)}$. This resembles the type system of several statically-typed object-oriented languages with generic data types, such as Java [BOSW98]. Generic data types are always at class level, and not at method level.

The following example shows the constraints generated in the presence of generic data types:

```
a := OrderedCollection new.
a add:x.
y := a any.
```

Some of the generated type variables are t_{a}, $t_{(\mathsf{a}.\alpha)}$, t_{x} and t_{y}. And the interesting constraints generated are:

- OrderedCollection $\in min(t_{\mathsf{a}})$.
- $t_{\mathsf{x}} \prec t_{(\mathsf{a}.\alpha)}$ because the type of the parameter of #add: is a subtype of the elements in the collection.
- $t_{(\mathsf{a}.\alpha)} \prec t_{\mathsf{y}}$ because the return type of #any:[4] is a subtype of the type of y.

4.3 Mixing Collections and Blocks

Since in Pharo, iterators are fully integrated in the language, collections are often combined with blocks. This combination provides some of the most challenging situations for a type inferer, but also very important to handle. Since the use of these combination is so frequent, a type inferer that is not able to handle it will be of little use for industrial programs. The following example shows the constraints that are generated in these situations:

```
| col r |
col := OrderedCollection new.
col add: anObject.
r := col sum: [ :e | e msg ].
```

In order to infer types for this piece of code, the algorithm will make use of the type specification associated to the methods #add: and #sum: of class OrderedCollection. To specify a generic data type we use a symbol instead of a concrete type, for example α, which will be associated to the subsidiary variable of the container type. So, the methods are specified as follows:

- #add: receives α[5].
- #sum: receives a block with type $\alpha\rightarrow$ SmallInteger and returns SmallInteger.

With this information, the algorithm can infer the information in Table 1.

Type Variable	Maximal Set	Minimal Set	Messages
t_{col}		OrderedCollection	#add: #sum:
t_{anObject}			#msg
$t_{(\mathsf{e\ mesg})}$		SmallInteger	
$t_{(\mathsf{col}.\alpha)}$			#msg
t_{r}		SmallInteger	

Table 1. Example combining collections and blocks

5. Example

In this section we will show a small example of the execution of the inference algorithm. This example, even being an small one, it is a clear example of real industrial code. It is an implementation of a Composite Pattern [GHJV95].

[4] Which returns any element of the collection

[5] The return type is not of interest

```
Task >> initialize
    subtasks := OrderedCollection new.

Task >> addSubtask: anObject
    subtasks add: anObject

Task >> complexity: anObject
    complexity := anObject

Task >> ownCost
    ^ complexity cost: self

Task >> totalCost
    ^ self ownCost +
    (subtasks sum: [ :subtask | subtask totalCost ])
```

Figure 2. Code for the Task class.

Type Variable	Maximal Set	Minimal Set	Messages
t_{subtasks}		OrderedCollection	#add:
$t_{\mathsf{subtasks}.\alpha}$	Task	Task[1]	#totalCost
$t_{(\mathsf{Task}\gg\#\mathsf{totalCost}.\uparrow)}$	SmallInteger[3]		
$t_{\mathsf{complexity}}$	MediumComplexity, SmallComplexity, HighComplexity[2]		#cost:
$t_{(\mathsf{self\ ownCost})}$	SmallInteger		#+

Notes:

1. Task is the only implementor of #totalCost
2. The implementors of #cost:.
3. It is the result of adding two SmallInteger's.

Table 2. Results of the Task class

The figure 2 presents the code of the Task class, from a task management system. An object of this class is responsible for the calculation of its own total cost. The cost of a task depends on two things: his own cost and the sum of the total costs of his subtasks. The own cost of a task is calculated by another object that understands the message #cost: with a task as a parameter, implementing an abstract Strategy pattern [GHJV95]. There are three classes which implement #cost:, MediumComplexity, SmallComplexity and HighComplexity. The task has two instance variables, #subtasks and #complexity. Table 2 shows the most relevant information resulting from analysing the Task class.

It is remarkable that the algorithm detects the type of the elements inside *subtasks*. Once established that t_{subtasks} is an OrderedCollection, we are able to take advantage of the type specification provided to OrderedCollection$\gg$#sum:. Therefore, the block has to be of type $\alpha \rightarrow SmallInteger$, where α is the type of the elements of the collection and is associ-

ated to the argument of the block (t_{subtask}). Given that t_{subtask} has to understand #totalCost and the only class implementing this method is Task, we deduce that $min(t_{\text{subtasks}.\alpha})$ = {Task}.

6. Discussion

One of the biggest challenges about infering types for object-oriented dynamic languages is scaling to handle big programs. Our solution has been only tested in small programs and an exhaustive analysis of its performance using more benchmarks and different kinds of programs is still a pending task. However, we are working on four different strategies to obtain acceptable execution times.

First, the possibility of specifying fixed types for core classes and other reusable libraries, reduces the number of methods to analyse and hence the number of type variables and constraints to solve.

Second, our implementation allows for different configurations, creating new tasks or even changing the existent ones. These capabilities can be exploded to regulate the relationship between precision and speed, increment the amount of information generated and even design custom tasks for a specific program.

Third, we are working on an *incremental* version of the algorithm, *i.e.,* which is able to handle method changes without running the whole type inference process from scratch. Since our intention of this work is to provide information for interactive tools, our aim is to listen to announcements regarding the method updates, discard only the type information related to the old version of the method, and keep the rest of the type information as input to the next execution of the algorithm.

Finally, one frequent problem of constraint solving type inferer is the amount of type variables that they generate, both because of their size in memory and execution time (since algorithms tend to grow exponentially on the amount of type variables). The incremental version of the algorithm will also cope with this problem, since after analysing a method we will be able to keep only the type variables in the *method interface* (*i.e.,* formal parameters and return type) discarding all other type variables which can be considered *internal* to the method.

Other approaches have proposed to use *type annotations* in order to have static type information. Type specifications show several advantages over type annotations. Since type specifications are not part of the code, the type system and the tools do not affect the structure of the code or the way a programmer works with it. In this way, our solution is oriented by the idea of *pluggable types* [Bra04], which proposes that the type information and the type system are not an integral part of the language, but only optional tools.

Moreover, most of the type specifications are not necessary for the algorithm to work. Instead, they are only a useful tool to speed up the inference process, by avoiding to analyse code that we are not intending to change. Is the programmer intends to change a portion of the core library, he could simply remove type specifications for that portion. Also, some type specifications can be the result of a previous execution of the type inferer, *i.e.,* we run the inference process on a reusable library and save the inferred types to be used as input when inferring types for a client of this library.

Our solution lacks of a formal model. The aim of this work is to provide a tool for industrial use, that can deal with real programs written in industrial languages and that is integrated in industrial programming environments. Therefore, our focus is to provide *useful* type information fast enough to be used in interactive tools. This objective can not be fulfilled if, in order to get a formal proof of soundness, we would cut off some of the most interesting parts of the language. We share this approach with renaming tool of Unterholzner [Unt12].

Our solution is able to handle generic data types but our approach can be limiting in some situations, we are analysing the idea of adding more support to Polymorphic Types: adding support to polymorphic messages, and detection of generic types developed by the programmer.

7. Related Work

To our knowledge, there are few type inference approaches combining abstract and concrete type information. In this regard, Graver [Gra89] proposed a type checking solution, based in an open world assumption. The main difference with our work, is that Graver's solution requires type specifications for class-, instance- and global variables, while in our solution this information is inferred.

Martin Unterholzner's work [Unt12] shares with ours the objective of using type inference to improve programming tools. In his case type information is used to aid method rename refactorings. To achieve his goal, he uses symbolic evaluation of the AST generated from the source code. His solution starts in a closed world context and then opens it to gather more information. His solution is focused in providing information for the renaming tool he is presenting, generating only the type information which is necessary for this specific purpose. Therefore it is not clear the applicability for use with other objectives.

Spoon and Shivers [SS04] have proposed an algorithm called DDP which prunes subgoals by giving solutions that are trivially true, reducing computation time at the cost of reducing precision. Their solution put emphasis in the performance and the scalability of the solution, but not in the precision. Our solution works also with different goals and tasks, but it does not use pruning. On the other hand our solution provides a way of inferring the type of the variable from the messages sent to it; in other words, we combine the information produced by the concrete types and the information from the abstract types.

Haupt *et al.,* [HPHf11] have proposed to gather type information by observing application code while it is running. They claim to provide more fine-grained results than type inference approaches. In this work the harvested types are objects' classes. The crucial difference with our work is that our solution is based in a static analysis, while the solution of Haupt *et al.,* gather all the information from running the tests. This approach needs to have running tests, and all the inference is based only in these tested cases.

Wang and Smith [WS01] propose an extension to CPA [Age95] which provides a way of handling generic data types. Their solution is focused in the checking of downcasts and it is highly coupled to Java Language.

Pluquet *et al.,* proposed a solution named Fast Type Reconstruction [PMW09], which is based in the static analysis of methods, but their solution only takes into account a small amount of expressions: assignation to literals, arithmetic expressions, boolean expressions and instance creations. This approach allows speeding up the solution but loses precision. Our solution propagates the return type of any message sent; providing more information about types.

Oxhoj *et al.,* [OPSb92] present a solution that can handle generic data types in the collection classes by using type variables for all the expressions and solving their relationships. However, their solution produces a high duplication of type variables, classes and methods as any generic type is duplicated for each use. For example, if there is a List of Booleans and a List of Integers; the algorithms creates two subclasses of List, one for each class of the elements.

8. Conclusion

In this work we proposed a practical solution for the lack of type information in dynamic languages, which focuses in providing useful information to be used in programming tools such as automatic refactoring, program understanding, program navigation and smart suggestions.

The solution provides an implementation for a real language in industrial level environment, which allows to build tools on top of it. The solution can be used as an external tool, without affecting the normal work of the programmer. Also, these ideas are applicable to other industrial level dynamic languages.

Regarding performance, we have promising results infering types for small programs and a testing bench for developing a responsive solution able to work on bigger programs. Our solution lacks of a formal proof of soundness, but we do not consider it a big drawback, given our objective.

One of the next steps in our research work is to add a global analysis of the type variables. For example, if the algorithm finds a local or instance variable with only one assignment, we could establish a new constraint, stronger than the subtype constraint that we are using now.

Also we intend to incorporate new tasks based on heuristics. This tasks provide potential answer with a lesser percentage of precision. Moreover, heuristic tasks could make use of fuzzy logic [ABCD65]. Heuristic tasks will be used in cases when the more precise tasks are not succesfull, adding them in a new run level. Still, other sequences of execution can be explored, changing the way the tasks are combined into different run levels.

Our solution can be used as a framework for developing different type systems. These type systems can work like plug-ins in a IDE. In this way the programmer can choose the set of tools to use in a particular problem. Also, our approach can be combined with other unrelated pluggable type systems, providing complementary information about the same program. As a result, the way different type systems and implementation cooperate and behave is an interesting approach to study.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219, the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS (no. 295261), Universidad Nacional de Quilmes through project PUNQ 1242/13 and Universidad Nacional San Martín's through project PRI01/13.

References

[ABCD65] Peter Fisher A, Charles Arnot B, Richard Wadsworth C, and Jane Wellens D. Fuzzy sets. *Information and Control*, pages 338–353, 1965.

[ACF+13] Esteban Allende, Oscar Callau, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual Typing for Smalltalk. *Science of Computer Programming*, August 2013.

[Age95] Ole Agesen. The cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, August 1995. Springer-Verlag.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph.D. thesis, Stanford University, December 1996.

[Ahm13] Mian Asbat Ahmad. *New Strategies for Automated Random Testing*. PhD thesis, University of York, March 2013.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 183–200. ACM Press, 1998.

[Bra04] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Eck03] Bruce Eckel. Strong Typing vs. Strong Testing, 2003. http://www.mindview.net/WebLog/log-0025.

[Gar01] Francisco Garau. *Inferencia de tipos concretos en squeak*. Master's thesis, Universidad de Buenos Aires, Buenos Aires, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for smalltalk. In *In Seventeenth Symposium on Principles of Programming Languages*, pages 136–150. ACM Press, 1990.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation (PLDI'05)*, pages 213–223, New York, NY, USA, 2005. ACM.

[GNDc04] Markus Gaelli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests. In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[Gra89] Justin Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. Ph.D. thesis, University of Illinois at Urbana-Champaign, August 1989.

[Hen94] Andreas V. Hense. *Polymorphic type inference for object-oriented programming languages*. Pirrot, 1994.

[HPHf11] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. Type harvesting: A practical approach to obtaining typing information in dynamic programming languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1282–1289, New York, NY, USA, 2011. ACM.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Mar03] Robert C. Martin. Are dynamic languages going to replace static languages?, 2003. http://www.artima.com/weblogs/viewpost.jsp?thread=4639.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[OPSb92] Nicholas Oxhoj, Jens Palsberg, and Michael Schwartzbach. Making type inference practical. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 329–349, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[PMW09] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 146–161, November 1991.

[RL13] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 2013. to appear.

[SS04] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 151–175. Springer Verlag, 2007.

[Suz81] Norihisa Suzuki. Inferring types in smalltalk. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 187–199, New York, NY, USA, 1981. ACM Press.

[Tra09] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[Unt12] Martin Unterholzner. Refactoring support for Smalltalk using static type inference. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '12, pages 1:1–1:18, New York, NY, USA, 2012. ACM.

[WB10] Allen Wirfs-Brock. A prototype mirrors-based refection system for javascript. https://github.com/ allenwb/jsmirrors, 2010.

[WS01] Tiejun Wang and Scott 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.