Considering Productivity Effects of Explicit Type Declarations

Michael Coblenz Jonathan Aldrich Brad Myers Joshua Sunshine
Carnegie Mellon University
{mcoblenz, aldrich, bam, sunshine}@cs.cmu.edu

Abstract

Static types may be used both by the language implementation and directly by the user as documentation. Though much existing work focuses primarily on the implications of static types on the semantics of programs, relatively little work considers the impact on usability that static types provide. Though the omission of static type information may decrease program length and thereby improve readability, it may also decrease readability because users must then frequently derive type information manually while reading programs. As type inference becomes more popular in languages that are in widespread use, it is important to consider whether the adoption of type inference may impact productivity of developers.

Keywords type inference, type declarations, programmer productivity

1. Introduction

Recent work described some of the benefits of type declarations in documentation. Sunshine et al. [1] found that users of Plaiddoc, which documents type state, could answer state search questions faster than users of Javadoc. Mayer et al. [2] and Petersen et al. [3] found that static type declarations improved performance on some programming tasks and degraded it on others, generally finding that static types are helpful for understanding documented code and fixing type errors but not for fixing semantic errors. However, a more focused look is needed at the tradeoffs involved specifically in type declarations and their longer-term effects.

In some languages, such as JavaScript, all variables are dynamically typed. Some languages, such as C, require type declarations, but the type declarations may contain very little information (e.g. "void *"). In contrast, other languages, such as SML [4], have static types that are very informative, but programs vary in the extent to which static types are declared due to type inference features in those languages. Type inference is becoming popular among language designers, as shown by its availability in new languages, such as Swift, and recent revisions to existing languages such as C++11, C# 3.0, and Visual Basic. As type inference becomes more popular in languages that are in widespread use, it is important to consider whether and in what cases adoption of type inference may impact the productivity of developers.

Avoiding lengthy type annotations via type inference and via dynamic types has some advantages. Reducing the amount of typing required to enter a program, since the types are either omitted or replaced by a terse keyword such as "auto," may improve programmer productivity both when entering the program and when reading it by avoiding superfluous verbosity. This benefit may be particularly important in languages with parametric polymorphism, in which types can become especially verbose and the code can become repetitious. Type inference may also reduce program maintenance costs by permitting programmers to modify code without updating corresponding type declarations. Particularly in cases of *local* type inference, type inference may improve users' productivity. Robin Milner, the inventor of type inference, wrote: "Although it can be argued convincingly that to demand type specification for declared variables, including the formal parameters of procedures, leads to more intelligible problems, it is also convenient–particularly in on-line programming–to be able to leave out these specifications." [4] Finally, type inference may help programmers notice bugs when they discover that an expression does not have the type they expected. If the type is inferred, the user might notice that the type is unexpected because the inferred type results in a type error. In this case, the type error could occur when the expression of inferred type is used in a place that expects an input of a different type. The user might also notice the error because the compiler gave the user the type name and the user finds it surprising.

The lack of explicit types may alternatively make code more difficult to read and understand, especially in the case of *non-local* type inference, where the information needed to determine an expression's type may be lexically or structurally distant. For example, if the type of a variable is nei-

PLATEAU '14, October 21 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2688204.2688218

ther explicit nor obvious, the user might guess an incorrect type and write code based on that type. Later, when the new code fails to typecheck, the user may be forced to backtrack.

These incorrect beliefs may lead to more frequent bugs in programs written in languages that do not specify types explicitly. Though tools might help by showing type information, the utility of this information depends on whether the tool shows the dynamic type in a specific execution of the code (as in a debugger), or whether it provides a static guarantee regarding the type. But perhaps displaying type information that can be easily inferred sometimes actually makes programs more difficult to understand due to verbosity.

In some languages, users have established conventions that incorporate type information into symbol names. For example, Scheme users are expected to end predicates in the question mark character [5]. Though this convention isn't enforced by the compiler and imposes a burden on programmers, it may be helpful for readers.

The overall research question is: in what situations are explicit static type annotations in programs helpful to programmers, and in what situations are they harmful? By identifying cases in which the presence of explicit types helps or hinders users, we hope to inform future language designs and to design tools to help users use types more effectively.

2. Hypotheses

Suppose a programmer wants to know the type of an expression *e*, where *e* has no type declared. The following hypotheses might be interesting to consider regarding the difficulty of determining the type.

2.1 Locality

In *local* type inference, type information for a given line of code can be derived from only that line and its typing environment. In contrast, *non-local* type inference also requires information from other lines of code. For example, in SML:

```
let val l = List.rev [] in
   42::l
end
```

The type of `l` depends on the next line; until then it is of polymorphic list type. Therefore, a programmer can't determine the type of `l` by reading only its binding.

Hypothesis 1. *Inferring types is easier for users (and therefore less of a usability problem) when the types can be inferred with local type inference.*

2.2 Scope

When a variable has narrow scope, its type may not be particularly important to the rest of the program. As a result, the user may benefit from type inference because the benefits of the terseness may outweigh the benefits of an explicit declaration. A common example in C++: "auto" is sometimes used for loop counters, which frequently have narrow scope:

```
std::vector<int> indices = ...
for (auto it = indices.begin();
      it != indices.end();
      ++it)
{
      ...
}
```

We do not include types of function arguments here, since those are part of the function's type rather than part of the function's body's scope.

Again considering the costs of navigation users bear in dynamically-typed languages [2], narrow scope may improve user productivity in dynamically-typed languages by reducing navigation costs.

Hypothesis 2. *If the scope of a variable is small, type inference is more beneficial than if the scope is large.*

2.3 Function types

Finding the type of a variable without a type declaration requires finding an assignment or binding to that variable, but when functions have undeclared return types, this may significantly increase the burden because users may need to trace an input through a sequence of function calls to determine the type of a given expression. That is, a variable may be bound to the result of a function call, whose implementation may itself call a function, etc. Likewise, if a function or method's arguments have unspecified types, determining what operations can be performed on the arguments may be difficult and require searching for calls to the function or method.

In the example below, determining the type of `something` requires knowing the function types of getCache() and fetch().

```
Cache.fetch = function () {...}
function getCache () {return ...}
var cache = getCache();
var something = cache.fetch(username); // ??
```

However, some functions, particularly anonymous ones, have narrow scope. In these cases, function type declarations may be less useful.

Hypothesis 3. *Programmer productivity is maximized when function types are visible in the function's declaration except for functions of narrow scope.*

2.4 Costs of unavailable type information

When type information is unavailable in systems with complex types, users may need to take several steps in order to determine the types of expressions. Users of SML frequently find that fixing type errors in programs is difficult because the source of the error may be distant from the place referenced by the error message [6]. However, these costs vary according to the language.

Hypothesis 4. *In languages with more expressive type systems (in which types are more informative than in other systems), the costs of unavailable type information are higher.*

2.5 Programmer usage of type inference

By understanding how and when users use type inference, we might hope to better understand how users view the tradeoffs of type inference. In particular, programmers may be using type inference as a tactic to reduce typing and lexically shorten program text. In languages that support type inference, users are saved significantly more typing (and programs are made much less verbose) when type inference is used for complex types than for simple types.

Hypothesis 5. *In languages in which type inference is used relatively rarely, such as in C++11, it is used more often with more complex types than it is with simple types.*

3. Benefits of type inference

In languages that have expressive static type systems and therefore would otherwise require users to insert type declarations manually, type inference may save users significant time when used judiciously. It would be interesting to quantify this benefit and compare it to the above tradeoffs. It has been shown that novices may have trouble using type declarations correctly [7]; perhaps type inference would help them program more successfully in languages that have strong static type systems. However, the existing studies focus on the short-term effects on productivity on small codebases. Less is known about longer-term implications and effects on larger projects.

Parametric polymorphism can lead to longer type names and repetition. This may be a particularly beneficial place for type inference so users can reduce verbosity. From Java example code [8], compare:

```
Map<String, List<String>> myMap =
     new HashMap<String, List<String>>();
```

to:

```
Map<String, List<String>> myMap =
     new HashMap<>();
```

Once more is known about the tradeoffs involved in explicit type declarations, languages and tools should be designed accordingly. IDEs could fill in undeclared types when users would benefit but not when it would only increase verbosity. Some IDEs provide type information when a user hovers over a symbol, but perhaps IDEs could show types automatically if it was known where users needed to know the types. Languages could be designed that support type inference in places where it is useful, and language designers could avoid spending effort supporting type inference in places where it would be detrimental for users to use it.

4. Next Steps

To evaluate the above hypotheses, we should do studies of programmers and of corpora of code from a variety of contexts. A corpus study might help address some of the hypotheses, particularly *programmer usage of type inference*. User studies may be useful, particularly regarding *locality* and *scope*. Novice programmers may have different needs than experts; maintenance tasks may present different problems than original development tasks; large codebases may impose different requirements than small ones.

By identifying cases in which type inference helps or hinders programmers, we can design tools to help users use type inference more effectively. In C++11, for example, users must currently decide whether to use type inference in each declaration. An editor could automatically insert type declarations where helpful and remove them when inference would improve readability and understandability of the code.

5. Acknowledgments

We appreciate the helpful input from Chris Martens. This work was supported in part by the NSA lablet contract #H98230-14-C-0140 and in part by NSF grant IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSA or NSF.

References

[1] J. Sunshine, J. D. Herbsleb, and J. Aldrich. Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming. In Proceedings of ECOOP 2014.

[2] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In Proceedings of OOPSLA, 2012.

[3] P. Petersen, S. Hanenberg, and R. Robbes. An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse. In ICPC 2014, Hyderabad, India, Jun. 2-3, 2014, 212-222. ACM, 2014.

[4] R. Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17:348-375, Aug. 1978.

[5] R. K. Dybvig. The Scheme Programming Language, third edition. The MIT Press. Sept. 2003.

[6] M. Beaven, R. Stansifer. Explaining Type Errors in Polymorphic Languages. In ACM Letters on Programming Languages and Systems, Volume 2 Issue 1-4, Mar.-Dec. 1993, 17-30.

[7] A. Stefik, S. Siebert. An Empirical Investigation into Programming Language Syntax. In ACM Transactions on Computing Education, Volume 13 Issue 4, Nov. 2013, Article No. 19.

[8] Type Inference (The Java™Tutorials > Learning the Java Language > Generics (Updated)). Oracle, 2014. Oct. 1 2014. <http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html>.