Mocha: Local Type Inference for Java

Chris Male and David J. Pearce

Computer Science Group,
Victoria University of Wellington, NZ
{malechri,djp}@mcs.vuw.ac.nz

Abstract. Java's static type system protects against certain runtime errors, but is often unwieldy and repetitive. Type inference gives the benefits of strong typing, without the burden of traditional static typing. We consider the local type inference problem, where fields and parameter/return types are given, but local variable types are inferred. We have designed and implemented Mocha, a Java language extension providing type inference, and we formalise a subset of it here. Mocha differs from other languages using type inference, such as Scala, C# 3.0 and OCaml, since variables can have different types at different program points.

1 Introduction

Static type information in Java provides strong protection against certain kinds of runtime error. As it stands, the static type information required is often unwieldy and repetitive. Type Inference is a well-known technique for inferring the static type of a variable based on its actual usage. Type inference gives the benefits of strong typing, whilst alleviating some of the burdens found with traditional static typing. We consider the restricted *local type inference* problem, where fields and parameter/return types are given, and the types of local variables are inferred. Several existing languages, most notably Scala [33], C #3.0 [4] and OCaml [32], employ type inference. Our approach represents a departure from such systems as we allow variables *to have different types at different program points within a method*.

We have designed and implemented Mocha, an extension to Java supported type inference. Consider the following simple method written in Mocha:

```
List<String> f(String x) {
  var y;
  if(...) { y = new ArrayList<String>(); }
  else {  y = new LinkedList<String>(); }
  return y;
}
```

Here variable `y` is declared `var`, indicating its type should be inferred. In fact, `y` has different inferred types at different points — `ArrayList<String>` on the true branch, `LinkedList<String>` on the false branch and `AbstractList<String>` after the conditional (which is the nearest super class of both and implements `List<String>`). The latter being the most specific type that Mocha can infer for `y` after the conditional. In languages like Scala, C #3.0 and OCaml, such code cannot be written since variables must have a single fixed type for the life of the method.

Allowing variables to have different types at different program points also means they can be *retyped* as a result of `instanceof` expressions. For example:

```
void f(Number x) { if(x instanceof Integer) { ... } }
```

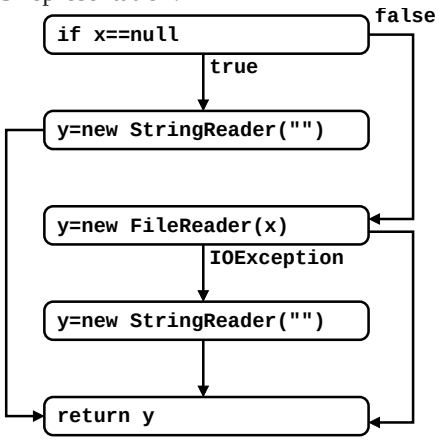
Here, Mocha automatically retypes variable `x` to `Integer` inside the true-branch of the conditional. Again, languages like Scala, C #3.0 and OCaml cannot support this. The contributions of this paper are:

1. We present Mocha, an extension of Java supporting Type Inference.
2. We formally describe the type inference algorithm for a subset of Java, including non-generic types and exceptions.

2 Overview

The syntax and subtyping rules for the subset of Mocha being considered are given in Figure 1. The reasons for having three subtype operators will become apparent in §2.1. We formalise our type inference algorithm as a dataflow analysis operating on a method's control-flow graph (CFG). This simplifies the handling of CFG join points, especially those arising from `try`/`catch` blocks. In the CFG, each node is a simple statement and edges may be labelled with true/false. Our CFG representation is unstructured, meaning high-level statements such as `for` loops and `try`/`catch` blocks are represented with conditional/unconditional branches. Likewise, expressions cannot have side-effects. We assume the program code has been first broken down into this CFG representation, as is commonly done within a compiler. While we do not detail this translation step, it should be relatively easy to see how it is done. The following illustrates a simple Mocha program and its CFG representation:

```
Reader f(String x) {
 var y;
 try {
   if(x == null) {
     y = new StringReader("");
   } else {
     y = new FileReader(x);
   }
 } catch(IOException e) {
   y = new StringReader("");
 }
 return y;
}
```



Here, we can see the main features of our CFG representation. In particular, exceptional control-flow is represented by *exceptional edges* (similar to those of [7, 22, 17]) which flow from statements to handlers.

2.1 Most Specific Type

A key issue lies in determining the most specific type for a variable at control-flow join points. Considering the example from above, three separate control-flow paths reach

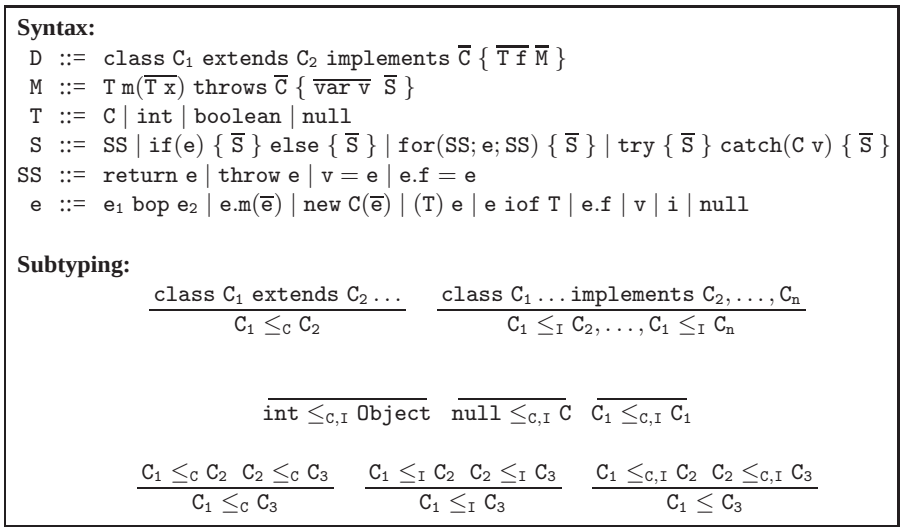


Fig. 1. The syntax and subtyping rules for the subset of Mocha being considered. Note that `iof` is short-hand for `instanceof`; a special type is required to represent the type of `null`; and, the notation $\leq_{C,I}$ is short-hand for $\leq_C$ or $\leq_I$. Thus, there are three subtype operators defined here: $\leq_C$ (for classes); $\leq_I$ (for interfaces); $\leq$ (for both, i.e. the usual subtyping operator).

the `return` statement. The most suitable type to give variable `y` at this point is clearly `Reader`, since it is a superclass of both `StreamReader` and `FileReader`. However, this is not the only option, as both `Cloneable` and `Serializable` are also implemented by `StringReader` and `FileReader`. The question then, is how to choose the appropriate type for `y` at this point.

The general approach to dealing with this kind of problem is to compute the *least upper bound* of types from incoming control-flow paths. However, as is apparent from our example, Java's subtype relation does not form a join semi-lattice, meaning there is not necessarily a unique least upper bound of any two types [19]. One solution is to ignore interfaces altogether (which, in fact, the Java bytecode verifier does [19]). This approach is not suitable here, however, since it would essentially mean that interfaces could not be used at all. Therefore, we adopt a hybrid approach which prioritises classes over interfaces. We define a *join operator* on types as follows:

$$\mathtt{T}_1 \sqcup \mathtt{T}_2 = \begin{cases} \mathtt{T}_1 & \text{if } \mathtt{T}_2 \leq \mathtt{T}_1 \\ \mathtt{T}_2 & \text{if } \mathtt{T}_1 \leq \mathtt{T}_2 \wedge \mathtt{T}_2 \not\leq \mathtt{T}_1 \\ \mathtt{T}_3 & \text{if } \mathtt{T}_3 = \mathtt{T}_1 \sqcup_{\mathtt{C}} \mathtt{T}_2 \text{ exists and } \mathtt{T}_1 \not\leq \mathtt{T}_2 \wedge \mathtt{T}_2 \not\leq \mathtt{T}_1 \\ \mathtt{T}_4 & \text{otherwise, where } \mathtt{T}_4 \in \mathtt{T}_1 \sqcup_{\mathtt{I}} \mathtt{T}_2 \end{cases}$$

Here, $\mathtt{T}_1 \sqcup_{\mathtt{C}} \mathtt{T}_2$ determines the (unique) least upper bound of $\mathtt{T}_1$ and $\mathtt{T}_2$, according to the class subtype relation (i.e. $\leq_{\mathtt{C}}$). In contrast, $\mathtt{T}_1 \sqcup_{\mathtt{I}} \mathtt{T}_2$ computes the *set* of minimal upper bounds of $\mathtt{T}_1$ and $\mathtt{T}_2$ according to the interface subtype relation (i.e. $\leq_{\mathtt{I}}$). And, since the latter may not return a unique element, we arbitrarily select from the set re-

turned[1]. Clearly, this approach to joining types cannot guarantee to always select the best type. In fact, it's easy to find situations *where it is provably impossible to select an appropriate type*, such as the following:

```
interface X { void f(); }
interface Y { void g(); }
class A implements X,Y {}
class B implements X,Y {}

void f(A a, B b) {
  var x;
  if(...) { x = a; } else { x = b; }
  x.f(); x.g();
}
```

In principle, this code is safe, since `x` is always both an `X` and a `Y` and, hence, must implement both `f()` and `g()`. However, our join operator is forced to pick either `X` *or* `Y` at the control-flow join. This causes a type error as neither interface has both methods. While this limitation may seem problematic, our experiences with using several real-world examples suggest it is not in practice. Examples such as the above appear to be more of a theoretical, rather than practical, interest. In any case, the programmer can always override the inferred type with an explicit cast.

Extending our type inference system with *intersection types* (e.g. [26]) should provide an elegant solution to this problem. With this approach, we would allow types of the form $\mathtt{T}_1 \wedge \mathtt{T}_2$ to represent values which are both instances of $\mathtt{T}_1$ *and* $\mathtt{T}_2$. However, we have not implemented this yet, but plan to do so in the future.

3 Type Inference Algorithm

Our type inference algorithm infers the type of local variables at each point within a method. Unlike traditional type systems, where a single typing environment Γ is used for the whole method, our algorithm maintains a typing environment Γ at each distinct program point. Furthermore, the algorithm cannot type the whole method in a single pass; rather, it may require several iterations to obtain the correct typing. The following piece of code illustrates the main issues for the algorithm:

```
Number f(Integer x) { ... }
Number f(Number x) throws IOException { ... }
Number g() {
  var y,i;
  y = new Integer(0);
  try { for(i=0; i<10; i=i+1) { y = f(y); }
  } catch(IOException e) { y = new Integer(1); }
  return y;
}
```

[1] In practice, we can further prioritise against picking certain interfaces, such as `Cloneable` and `Serialisable` which are rarely needed.

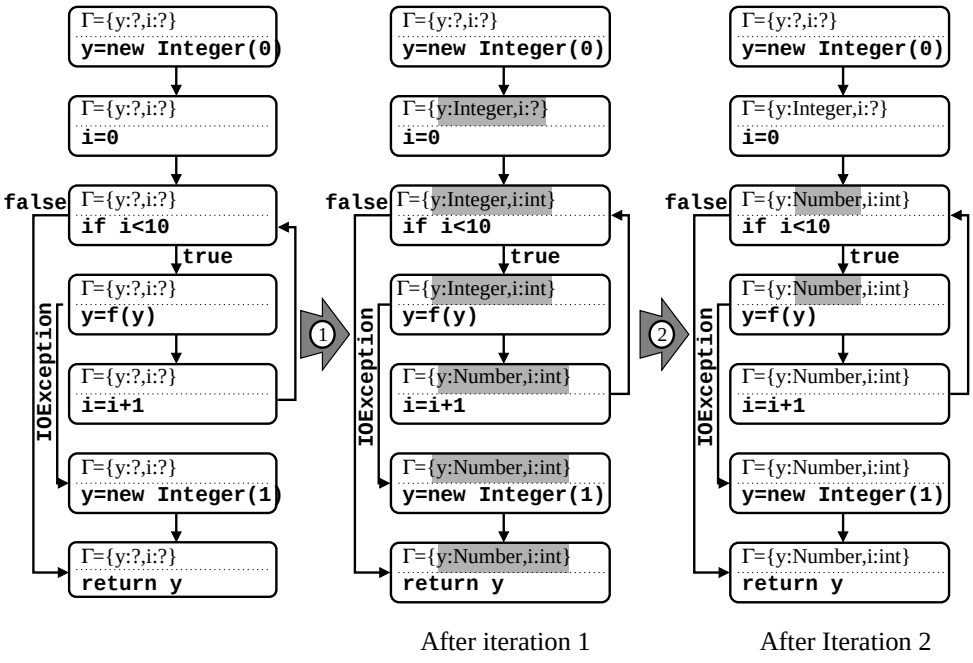


Fig. 2. Illustrating our type inference algorithm operating on a simple example. The typing environment immediately before each statement is given. The grey boxes highlight the changes between each iteration.

Figure 2 illustrates our algorithm performing type inference on the control-flow graph of this method. Initially, the type of each variable is marked as unknown. Then, the algorithm visits each node inferring types based upon the expressions they are assigned (iteration 1). At this point, the type information is not yet complete and the algorithm must iterate again to propagate the type for `y` along the back-edge of the loop (thus, reaching a fix-point). Observe that, when $\texttt{y} = \texttt{f}(\texttt{y})$ is first visited, the type of `y` was `Integer` and, hence, the resolved type for `f` was `Integer` $\rightarrow$ `Number`; but, when $\texttt{y} = \texttt{f}(\texttt{y})$ is revisited, `y` has type `Number` and, hence, the resolved type for `f` is `Number` $\rightarrow$ `Number throws IOException`. Thus, it is apparent that method resolution is entwined with the type inference algorithm.

Figure 2 also illustrates the role exceptional edges play in the algorithm. The only statement within the original `catch` block which could potentially throw an `IOException` is connected via an exceptional edge to the handler. Observe that, in fact, this edge is redundant — that is, its statement cannot actually throw an `IOException` since the method called doesn't declare this in its `throws` clause. At CFG construction time, this was not known — it is only *after* type inference that it becomes known. One may wonder whether or not we could eliminate such redundant exceptional edges. To do this, our type inference algorithm would need to be able to add exceptional edges as it proceeds (and, in fact, remove them). This significantly increases the algorithm's com-

plexity and, for simplicity, we ignore the issue of eliminating redundant exceptional edges in this paper.

3.1 Expressions and Statements

Typing an expression in Mocha, given an appropriate type environment, is identical to that of normal Java. The rules for our subset of Mocha are given in Figure 3. Dealing with statements, however, is rather different. We use the transition rules given in Figure 4 to describe the effect of a statement on its typing environment. These give the typing environment after a statement in terms of the typing environment before it, whilst accounting for the statement's effect. The expression $\Gamma[\texttt{x} \mapsto \texttt{T}]$ yields the typing environment Γ, except with variable `x` given type `T`. The two most interesting rules are `S-ASSIGN1` and `S-IF2`. The former infers the type of a variable at a particular program point based on the type of the expression it is assigned; the latter retypes a variable as a result of an `instanceof` test. Note that, while the obvious use of `instanceof` is for downcasting, upcasting is also useful when a type has multiple super types.

3.2 Typing Equations

The final piece of the jigsaw are the *typing* or *dataflow equations*. These specify how the typing environments entering each statement are actually computed. Before we can present the typing equations, we must first indicate how two typing environments are *joined*. That is, when two typing environments from different paths come together at a join point, what results? To do this, we define a join operator on typing environments as follows:

Definition 1. *Let Γ_1, Γ_2 be type environments. Then, $\Gamma_1 \sqcup \Gamma_2 = \{x \mapsto T_1 \sqcup T_2 \mid x \mapsto T_1 \in \Gamma_1 \wedge x \mapsto T_2 \in \Gamma_2\}$.*

Observe that this uses the join operator on types defined in §2.1. Using this, the typing equations can now be defined as follows:

Definition 2. *Let $G = (V, E)$ be a control flow graph for method M, where V is the set of nodes, and E the set of edges. Then the typing equations for each point y in M are given by $T_M(y) = \bigsqcup_{x \xrightarrow{l} y \in E} f(S_y, T_M(x), l)$.*

Here, f is the *transfer function* defined by the statement transition rules of Figure 4. Furthermore, S_y is the statement at the node y, and the edge label l separates the true and false edges coming from conditional statements.

Mocha solves the typing equations using a standard worklist algorithm, similar to that used in the Java bytecode verifier (see e.g. [31] for more on this).

3.3 Soundness

Although we have not yet proved soundness of our system, we believe it will be amenable to standard techniques. We now sketch the basic idea. Essentially, we say that our system is sound if every program that can be typed will not get *stuck* (e.g. attempting to call

Expression Typing:

$$\frac{\{\mathtt{x} \mapsto \mathtt{T}\} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \text{ [T-VAR]} \qquad \frac{\mathtt{i} \in \{\ldots, -1, 0, 1, 2, 3, \ldots\}}{\Gamma \vdash \mathtt{i} : \mathtt{int}} \text{ [T-INT]}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}, \mathtt{e}_2 : \mathtt{T} \quad \mathtt{bop} \in \{==, !=, <=, \ldots\}}{\Gamma \vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2 : \mathtt{boolean}} \text{ [T-BOP1]} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{int}, \mathtt{e}_2 : \mathtt{int} \quad \mathtt{bop} \in \{+, -, /, *, \ldots\}}{\Gamma \vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2 : \mathtt{int}} \text{ [T-BOP2]}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}_1 \quad \mathtt{T}_1 \leq \mathtt{T}_2 \vee \mathtt{T}_2 \leq \mathtt{T}_1}{\Gamma \vdash (\mathtt{T}_2)\ \mathtt{e} : \mathtt{T}_2} \text{ [T-CAST]} \qquad \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \mathbf{field}(\mathtt{f}, \mathtt{T}_0) = \mathtt{T}_1}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{T}_1} \text{ [T-FIELD]}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \Gamma \vdash \mathtt{e}_1, \ldots, \mathtt{e}_n : \mathtt{T}_1, \ldots, \mathtt{T}_n \quad \mathbf{method}(\mathtt{T}_0, \mathtt{m}, \mathtt{T}_1, \ldots, \mathtt{T}_n) = (\mathtt{T}_1, \ldots, \mathtt{T}_n) \rightarrow \mathtt{T}_r}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{T}_r} \text{ [T-IVK]}$$

$$\frac{\Gamma \vdash \mathtt{e}_1, \ldots, \mathtt{e}_n : \mathtt{T}_1, \ldots, \mathtt{T}_n \quad \mathbf{constructor}(\mathtt{C}, \mathtt{T}_1, \ldots, \mathtt{T}_n)}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{C}} \text{ [T-NEW]} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}_0 \quad \mathtt{T}_0 \leq \mathtt{T}_1 \vee \mathtt{T}_1 \leq \mathtt{T}_0}{\Gamma \vdash \mathtt{e}\ \mathtt{iof}\ \mathtt{T}_1 : \mathtt{boolean}} \text{ [T-IOF]}$$

Fig. 3. Typing rules for the expressions of our language. Note, $\mathbf{method}(\mathtt{T}, \mathtt{m}, \mathtt{T}_0, \ldots, \mathtt{T}_n)$ determines the type of method $\mathtt{m}$ on receiver $\mathtt{T}$ with arguments $\mathtt{T}_0, \ldots, \mathtt{T}_n$. This requires traversing T's type hierarchy looking for the most applicable method [13]. Likewise, $\mathbf{field}(\mathtt{T}, \mathtt{f})$ returns the type of field $\mathtt{f}$ of receiver $\mathtt{T}$, again by traversing the class hierarchy as necessary. Finally, $\mathbf{constructor}(\mathtt{C}, \mathtt{T}_1, \ldots, \mathtt{T}_n)$ checks whether such a constructor exists for class $\mathtt{C}$.

Statement Transitions:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\mathtt{v} = \mathtt{e} : \Gamma \longrightarrow \Gamma[\mathtt{v} \mapsto \mathtt{T}]} \text{ [S-ASSIGN1]} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \mathtt{e}_2 : \mathtt{T}_2 \quad \mathbf{field}(\mathtt{f}, \mathtt{T}_1) = \mathtt{T}_3 \quad \mathtt{T}_2 \leq \mathtt{T}_3}{\mathtt{e}_1.\mathtt{f} = \mathtt{e}_2 : \Gamma \longrightarrow \Gamma} \text{ [S-ASSIGN2]}$$

$$\frac{\mathbf{method}(\mathtt{T}_0, \mathtt{this}_m, \mathtt{this}_{p1}, \ldots, \mathtt{this}_{pn}) = (\ldots) \rightarrow \mathtt{T}_0 \quad \Gamma \vdash \mathtt{e} : \mathtt{T}_1 \quad \mathtt{T}_1 \leq \mathtt{T}_0}{\mathtt{return}\ \mathtt{e} : \Gamma \longrightarrow \bot} \text{ [S-RETURN]}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \mathtt{T} \leq \mathtt{Throwable}}{\mathtt{throw}\ \mathtt{e} : \Gamma \longrightarrow \bot} \text{ [S-THROW]}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{boolean}}{\mathtt{if}\ \mathtt{e} : \Gamma \longrightarrow \Gamma} \text{ [S-IF1]} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}_0 \quad \mathtt{T}_0 \leq \mathtt{T}_1 \vee \mathtt{T}_1 \leq \mathtt{T}_0}{\mathtt{if}\ \mathtt{v}\ \mathtt{iof}\ \mathtt{T} : \Gamma \xrightarrow{\mathtt{true}} \Gamma[\mathtt{v} \mapsto \mathtt{T}]} \text{ [S-IF2]}$$

Fig. 4. Transitions rules for statements. Note, the variables $\mathtt{this}_m, \mathtt{this}_{p1}, \ldots, \mathtt{this}_{pn}$ give the name and parameter types of the enclosing method. Also, the S-IF2 rule applies for the transition along the true branch, and we assume it takes precedence over S-IF1 when it applies. Finally, $\bot$ is the empty typing environment.

a method m on an object which has no method m). Observe that stuckness is different from throwing runtime errors (e.g. `NullPointerExceptions`); that is, well-typed programs may still throw such errors, as they can in normal Java. We also require variables are defined before used and assume the normal Java rules for checking this.

Theorem 1. *Given the control-flow graph for a method M, the typing equations of Definition 2 always generate a program point specific typing that is sound.*

4 Discussion

Field Retyping. An interesting problem arises when retyping fields in Java. Consider the following:

```
class X {
  Number f;
  void g() { if(f instanceof Integer) { ... } }
}
```

The issue is whether or not we can safely retype field `f` inside the true branch. This seems sensible, but can cause problems in the presence of concurrency if `f` is reassigned by some separate thread after the `instanceof` test. Of course, code which assumes `f` is an `instanceof Integer` when such reassignments may occur has undefined semantics under the Java memory model. A similar situation can occur when the true branch calls another method that may potentially update field `f`. To deal with these situations, we currently require the programmer manually apply the *field-load fix*, giving something like the following:

```
class X {
  Number f;
  void g() {
    Number _f = f;
    if(_f instanceof Integer) { ... }
} }
```

Here, `f` is shadowed with a local variable which is implicitly thread local. Thus, one can now be certain that `_f` is indeed an `instanceof Integer` in the true branch. An interesting question is whether or not our system should be applying this transformation automatically. Certainly, it seems that it would be helpful in situations such as this.

Equality. Another interesting situation that can arise is illustrated in the following:

```
interface A { ... }
interface B { ... }
class C implements A,B { ... }
void f(A a, B b) {
  if(a == b) { ... }
}
```

In this example it would be possible to retype both `a` and `b` to have type `C` in the true branch. Currently, we do not support this, although it would be interesting to see whether such behaviour was useful in practice.

5 Related Work

The first, and most widely used type inference system was developed by Hindley [15] and later independently by Milner [23]. Since then, numerous systems have been developed for object-oriented languages (e.g. [25, 24, 18, 2, 28, 8, 34, 3]). These, almost exclusively, assume the original program is completely untyped and employ set constraints (see [1, 14]) as the mechanism for inferring types. As such, they address a somewhat different problem to that studied here. To perform type inference, such systems generate constraints from the program text, formulate them as a directed graph and solve them using an algorithm similar to transitive closure. When the entire program is untyped, type inference must proceed across method calls (known as *interprocedural analysis*) and this necessitates knowledge of the program's call graph (in the case of languages with dynamic dispatch, this must be approximated). Typically, a constraint graph representing the entire program is held in memory at once, making these approaches somewhat unsuited to separate compilation [25].

Bierman *et al.* formalise the type inference mechanism to be included in C# 3.0, the latest version of the C# language [4]. This uses a very different technique from us, known as *bidirectional type checking*, which was first developed for System F by Pierce and Turner [27]. This approach is suitable for C# 3.0 which does not permit variables to have different types at different program points.

Of more direct relevance to this work are type inference systems based on *dataflow analysis*. The Java bytecode verifier is a well-known example [20]. Since locals and stack locations are untyped in Java Bytecode, it must infer their types to ensure type safety. The verifier uses a similar algorithm to that presented here, although with some notable differences. In particular, method types are encoded into the `invoke` bytecodes, so method resolution is separated from type inference. Also, to deal with the Java class hierarchy not forming a join semi-lattice, the bytecode verifier ignores interfaces altogether, instead relying on runtime checks to catch type errors (see e.g. [19]). However, several works on formalising the bytecode verifier have chosen to resolve this issue with intersection types instead (see e.g. [12, 30]).

The JACK tool for bytecode verification of `@NonNull` types has some relation to our approach [21]. This extends the bytecode verifier with an extra level of indirection called *type aliasing*. This technique tracks local and stack locations which are known to aliases, thus enabling the verification of `@NonNull` types. The algorithm used is otherwise identical to that of the bytecode verifier and, hence, has the same differences from that studied here. An interesting observation from this and other work on `@NonNull` types (e.g. [29, 10, 9, 6]), is that our type inference algorithm would be valuable in this setting as well. This is because it would allow statements such as "`if(x!=null) { ... }`" to retype `x` as `@NonNull` in the conditional body.

The work of Gagnon *et al.* presents a technique for converting Java Bytecode into an intermediate representation [11]. Key to this is the ability to infer static types for the locals and stack locations used in the bytecode. Their problem is rather different from ours, since it is about inferring a single static type for each variable. Since variables are untyped in Java bytecode, this is not always possible as a variable can — and often does — have different types at different points; in such situations, they split variables as necessary into two or more variables, with each having a different static type.

Finally, we are aware of few works which attempt to extend the Java language with intersection types. The most relevant is that of Büchi and Weck who introduce *compound types* in to Java to overcome limitations caused by a lack of multiple inheritance [5]. Another interesting work is that of Igarashi Nagira, who introduce *union types* into Java [16]. These represent values which may be one *or* other of the possibilities; this differs from intersection types which represent values that are instances of *all* the possibilities.

6 Conclusion

In this paper, we have formalised a subset of Mocha, an extension to Java supporting type inference that we have designed and implemented. This differs from other languages that use type inference, since it permits a variable to have different types at different points within a method. This facilitates a more flexible style of programming that is more natural to existing Java programmers. One of the key challenges in our system is determining what type a variable should have at control-flow join points. Currently, we employ a simple mechanism that attempts to make a sensible choice. While we have found this works well in practice, it is not guaranteed to work in all cases. In such situations, the programmer can still employ a cast to obtain the necessary type. In the future, we would like to explore the use of intersection types to alleviate this problem, and also to extend our system to Java generics.

Acknowledgements. This work was supported by a masters scholarship from Victoria University of Wellington.

References

1. A. Aiken and E. L. Wimmers. Solving systems of set constraints. In *Proc. symposium on Logic in Computer Science*, pages 329–340. IEEE Computer Society Press, 1992.
2. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proc. conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, 1993.
3. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *Proc. ECOOP*, pages 428–452, 2005.
4. G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *Proc. OOPSLA*, pages 479–498, 2007.
5. M. Büchi and W. Weck. Compound types for java. In *Proc. OOPSLA*, pages 362–373, 1998.
6. P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proc. ECOOP*, pages 227–247. Springer, 2007.
7. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proc. PASTE*, pages 21–31, 1999.
8. J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA*, pages 169–184. ACM Press, 1995.
9. T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.
10. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, pages 302–312. ACM Press, 2003.

11. E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Proc. SAS*, volume 1824, pages 199–219. Springer, 2000.
12. A. Goldberg. A specification of java loading and bytecode verification. In *Proc. CCS*, pages 49–58, 1998.
13. J. Gosling, G. S. B. Joy, and G. Bracha. *The Java Language Specification, 3rd Edition*. Prentice Hall, 2005.
14. N. Heintze. Set-based analysis of ML programs. In *Proc. conference on Lisp and Functional Programming*, pages 306–317. ACM Press, 1994.
15. J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transations of the American Mathematical Society*, 146:29–60, 1969.
16. A. Igarashi and H. Nagira. Union types for object-oriented programming. In *Proc. SAC*, pages 1435–1441, 2006.
17. J. Jorgensen. Improving the precision and correctness of exception analysis in SOOT. Technical report, McGill University, Canada, 2003.
18. S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. conference on LISP and Functional Programming*, pages 193–204. ACM Press, 1992.
19. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.
20. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.
21. C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @nonnull types. In *Proc. Compiler Construction*, pages 229–244, 2008.
22. J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proc. Compiler Construction*, pages 153–184, 2002.
23. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
24. N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *Proc. ECOOP*, pages 329–349. Springer, 1992.
25. J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA*, pages 146–161. ACM Press, 1991.
26. B. C. Pierce. Intersection types and bounded polymorphism. *Math. Structures in Comp. Sci.*, 7(2):129–193, 1997.
27. B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.
28. J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. OOPSLA*, pages 324–340. ACM Press, 1994.
29. P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. Compiler Construction*, pages 334–554, 2001.
30. C. Pusch. Proving the soundness of a java bytecode verifier specification in isabelle/hol. In *Proc. TACAS*, pages 89–103, 1999.
31. Z. Qian. Standard fixpoint iteration for java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.
32. D. Remy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.
33. The scala programming language. http://lamp.epfl.ch/scala/.
34. T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *Proc. ECOOP*, pages 99–117. Springer, 2001.