Incremental Type Inference for Software Engineering

Jonathan Aldrich
University of Washington
jonal@cs.washington.edu

Abstract

Software engineering focused type inference can enhance programmer productivity in statically typed object-oriented languages. Type inference is a system of automatically inferring the argument and return types of a function. It provides considerable programming convenience, because the programmer can realize the benefits of a statically typed language without manually entering the type annotations. We study the problem of type inference in object-oriented languages and propose an incremental, programmer-aided approach. Code is added one method at a time and missing types are inferred if possible. We present a specification and algorithm for inferring simple object-oriented types in this kind of incremental development environment.

1. Introduction

Type inference was popularized by the ML language [Milner et al., 1990]. ML is made up of functions and simple data types such as records, tuples, and lists. Every function in ML has a well-defined type that includes the required types of its arguments and the type of the result of the function. Because ML is strongly typed and statically typed, each function must be type-checked to ensure that it calls other functions with arguments of the correct type. Static typing is desirable because it ensures that there will be no run time type errors.

1.1. Motivation

ML types can be rather complex. It has been shown [Mairson, 1990] that the description of the most general type of a function can be exponential in the size of the function definition. In practice, inferred types are not exponentially long but may be more obscure than one might like. Much of this complexity comes from the power of the ML typing system—it supports a parameterized type (a list), record and tuple types, polymorphic types, and closures. This type complexity allows programmers to use powerful language constructs like higher-order functions in a statically type-safe way. The desire to use high-level language constructs while preserving a static type-safe guarantee drives programmers to ask for ever-more powerful type systems. However, specifying a type that may be almost as long and complicated as the function it describes can be impractical.

1.2. Type Inference in ML

In ML, it is possible for the compiler to analyze the body of a function and infer the type of that function. For example, a function that squares each of its arguments and returns the sum of the squares clearly takes several arguments of type int and yields an int in return (see figure 1).

```
- fun sumsqrs (x, y) = x*x + y*y;

val sumsqrs = fn : int * int -> int
```

Figure 1. An example of type inference in ML.

A more powerful aspect of ML's type system is illustrated in figure 2. The append function takes two lists that holds elements of some type 'a, and returns another list of the same type which is the two lists' concatenation. ML infers that append can take any kind of list, but that the two lists and the list returned must contain the same type of elements. Thus it infers the type ('a list * 'a list) -> 'a list, indicating that append is a polymorphic function that can take several different kinds of arguments.

```
- fun append(nil, l) = l
  |  append(head:tail, l)
        = head :: append(tail, l);

val append = fn :
     ('a list * 'a list) -> 'a list
```

Figure 2. A polymorphic function.

If the compiler can automatically determine types, programming effort is saved because the programmer

need not enter the type annotations or even analyze exactly what the type should be. This eliminates a potential source of bugs: a function is type-safe, but the programmer has annotated it with the wrong type. Furthermore, the compiler can often infer a more general type than the programmer might have annotated a function with, making that function potentially more extensible. Because of these benefits of type inference, it would be nice to provide type inference in object-oriented programming languages.

1.3. Object Oriented Type Systems

Types in a statically typed object-oriented language fulfill four principal roles. The primary role is to statically typecheck a program to guarantee that no type errors will occur at runtime. Another role is to provide a framework for extensibility within that static type safety guarantee. We can extend an object-oriented program by providing a new implementation for any type in the program. Third, types provide a valuable kind of machine-checkable documentation. They specify what kind of object a method operates on, giving important clues to the behavior of that method. Finally, type declarations can improve the efficiency of code, because the compiler can perform many optimizations of dynamic message sends if it can statically limit the receiver of the message to a small set of implementations.

1.4. A Software Engineering Focus

Our focus is on the software engineering roles of types, including program safety, future extensibility, specification and documentation. Other researchers have made considerable progress automatically inferring types for safety and code optimization. This work often hides the extensibility and documentation benefits of strongly typed languages. Therefore, we propose that code be developed with a type system that supports software engineering. When the code is compiled, a separate type inference pass focused on optimizing code can achieve the performance benefits that other researchers have observed [e.g. Grove 1995].

We see type inference as a tool to enhance programmer productivity when working with existing object-oriented type systems. It is crucial that we do not sacrifice the benefits of a well-designed type system just to make type inference easier. This restriction makes type inference more difficult; but if type inference is an optional productivity-enhancing tool, our algorithms need not be perfect. Following the designers of CLP(R) [Jaffar et al., 1992], we believe it is better to provide an approximate solution to the type inference problem than to provide a perfect solution to an easier-to-solve approximation of the type inference problem. We break this rule by excluding F-bounded polymorphism and parameterized types only because of the time pressure to complete this project.

Thus we are willing to design our inference algorithms to work correctly in the common case. In more difficult situations, type inference may require some aid from the programmer in the form of type annotations. In this way type inference is able to support the programmer without interfering with the important software engineering abstractions supported by a powerful type system. If it can be shown that type inference usually deduces the correct type in practice, it will be successful in lightening the burden of static types. A beneficial side effect may be raising programmers' tolerance for more powerful type systems, since the additional power will come at a lower cost due to the convenience of type inference.

1.5. Environment Issues

We believe that type inference is most useful in an incremental development environment such as the Smalltalk or ML environment. In ML, functions are typed in one by one, and as each function is compiled, its type is inferred and reported to the programmer. The programmer can then check the inferred type to make sure it describes the intended type. The inferred type may be incorrect either because a more restricted type was desired or because the inferred type showed some programming error. In either case, the programmer can edit the function (perhaps adding optional type annotations) before continuing on to the next piece of code. The Smalltalk development environment (which does not provide type inference) shows that support for incremental development can be of great value in object-oriented programming languages as well as functional languages (see figure 3).

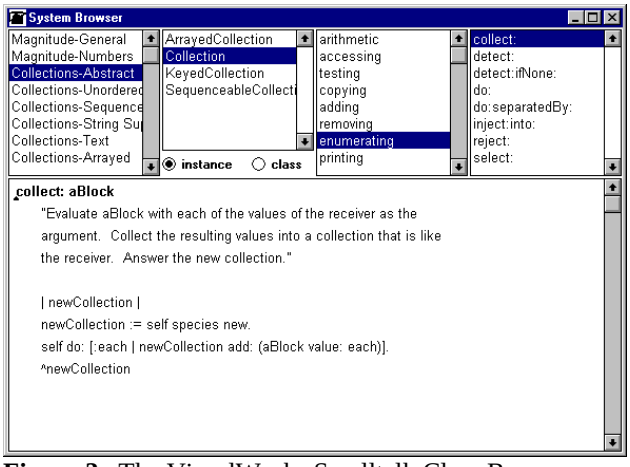

Figure 3. The VisualWorks Smalltalk Class Browser.

Therefore, we propose that an incremental type inference system be added to an interactive, incremental development environment for object-oriented programs. A graphical browser such as the Smalltalk browser (figure 3) provides a window for editing class, method, and object definitions. Methods are added to the system one at a time, and as each method is compiled the development environment infers any types that are left out by the programmer. Just as in ML, the programmer could then examine the inferred types and update the code to reflect his understanding of the proper types. Ideally, the inferred types would correspond to the programmer's intentions (as is usually the case in ML) and so the programmer gains the software-engineering benefits of having static types in the code without much of the usual effort required. The programmer specifies only type definitions, the types that each class conforms to, and the types of global, class, and instance variables. The types of local variables, methods, and closures can all be inferred, eliminating up to 90% of all type annotations [Graver, 1989].

2. Type Systems

Many different type systems have been proposed for object-oriented programming systems. This section discusses the software engineering tradeoffs inherent in many of these choices, motivating the type system used in our model.

2.1. Structural Subtyping

Cardelli [Cardelli, 1984] proposed a simple model of structural subtyping in an object-oriented language with multiple inheritance. In structural subtyping, type a subtypes type b iff type a includes all of the method interfaces in type b. This system is convenient because any set of methods defines a type, and any class that implements those methods automatically subtypes from that type. Several object-oriented type inference projects have used structural subtyping because it makes type inference easier and provides a well-defined solution (a *principal type*) to the type inference problem.

Unfortunately, structural subtyping is far from ideal from a software engineering perspective. Abstract types are useful because they express a programmatic concept that is otherwise missing from the text of a program. An abstract type stands for not only the method signatures in that type but also the behavioral specification present only in the programmer's head and in code documentation. Structural subtyping loses this semantic value by equating types with sets of signatures. Furthermore, a structural type is often difficult to understand because it may include many signatures, rather than using a single meaningful name to encapsulate a set of methods and their intended behavior.

By separating a signature from the abstract type it is part of, structural subtyping can cause subtle bugs. Although the types Bag and Set both have a method add, the methods mean very different things. Adding an element to a Bag always increases the number of elements, while adding an element to a Set may not increase its size if the element is already present! A method that operates on any object with an add method (using structural subtyping) could be incorrect if it assumes that add always increases the size of the collection.

Because of these problems with structural subtyping, we model types as abstract specifications of methods and their behavior. Two types may be distinct even if their interfaces are identical, because their different names correspond to different programming concepts.

2.2. Sets of Classes

In some systems, a type is an arbitrary set of classes. Inferring the type of a function is thus equivalent to discovering which classes are passed to the function and which classes it returns. Sets of classes are easy to compute, and are convenient for optimization purposes.

Unfortunately, sets of classes are poorly suited to supporting the software-engineering role of types. Although class analysis can determine that a program is type-safe, the type indicated by a set of classes is often not a meaningful abstraction to the programmer. Furthermore, the set of classes that is passed to a particular function changes from one program to another. This is because the algorithms are intended to analyze which classes are actually passed to the function in the execution of a particular program, not which classes could be passed in the most general case. If new classes are added to the system, it is not immediately clear where they can be safely used, because the types inferred in the previous analysis do not include the new classes. These types are thus unable to inform the programmer of which classes can be extended to increase functionality without breaking type safety.

2.3. Union and Intersection Types

Union and intersection types are kinds of structural types that combine the interfaces of two types. The union of two types A and B is the most general subtype of A and B; every type that inherits from both A and B also implicitly inherit from the union type. An intersection type is the most specific supertype of A and B; every type that is a supertype of both A and B is implicitly a supertype of their intersection.

While these types can add a limited amount of power to the typing system, they can be very confusing and difficult to understand. A type inference algorithm that includes union and intersection types is likely to infer them often, because they can often allow a more general type to be inferred.

```
type A; type B; type C; type D;

signature foo(A,B):A;
signature foo(C,D):C;

method bar(arg1, arg2) {
  return foo(arg1, arg2);
}
```

Figure 4. A method with two ambiguous types.

Figure 4 shows a method bar that could be given a more general union type. Without union types method bar could be assigned type (A * B) -> A or (C * D) -> C. A more general type is ((A * B) -> A) | (C * D) -> C)) where | indicates a union type. Unfortunately, this type is rather complicated. A programmer looking at a method bar with a union type may need to expend considerable effort just to understand what the type means. Thus a compiler that often infers union types for methods may produce type documentation that is unintelligible to the programmer.

The union type we could assign to bar is not only confusing, it is likely to be incorrect. If there is no relationship between the pair A,B and the pair C,D, then the foo signatures are probably unrelated except by their name. They may refer to completely different operations, or on the same operation applied to unrelated domains. In this case the programmer's intention in writing bar probably applies to only one of the foo signatures.

Because union and intersection types are confusing and error-prone, we limit types to the abstractions developed by programmers.

2.4. Principal Types

ML's system infers principal types [Palsberg, 1996], which means that an inferred type generalizes all possible types for a function. For example, type ('a list * 'a list) -> 'a list is a principal type for the append function discussed earlier; it includes all other possible types such as (int list * int list) -> int list. A principal type property is desirable because it means there is a unique best answer to the type inference problem.

Our type system does not allow the inference of principal types. This is because we do not allow union and intersection types or other forms of structural subtyping. In the example shown in figure 4 above, there are two possible types for method bar, and without knowing the programmer's intention in the foo method it is impossible to determine which is correct.

In cases like this (which are likely to be uncommon) we guess a possible type assignment using heuristics, and let the programmer correct the guess by adding type annotations if necessary. A reasonable alternative is to present the programmer with a list of choices whenever a type could be ambiguous. This solution follows the pattern of approximating the ideal type inference solution rather than forcing the programmer to use an inferior structural type system.

2.5. A New Inference System

We present a specification, algorithm, and implementation of a simple incremental type inference system suitable for software engineering in an incremental development environment like the one suggested above. Our type inference system is powerful enough to infer types for a simple object-oriented programming language like Java. As far as our type inference system is concerned, types may be associated with classes (as in Java), or they may be a separate entity (as in Cecil). We include closures, multiple inheritance and overloading in our system, but leave advanced typing features such as polymorphic, parameterized, and F-bounded types [Canning et al., 1989] to future research.

While these advanced type systems are necessary for many real applications, they complicate the inference problem considerably. Many real-world programming languages, including Java, omit features like F-bounded polymorphism to simplify their programming model. The scope of this research does not allow us to explore inference in more powerful object-oriented type systems, but we hope to tackle these problems in the future.

3. Related Work

The presence of subtyping makes type inference more complicated, and researchers have had difficulty making inference work as well in object-oriented systems as it works in functional programming systems. [Agesen, 1995] provides a summary of work in the field prior to 1995. Here I provide a description of those projects most relevant to this proposal. Although I have attempt to included many object-oriented type inference projects that support the software-engineering aspects of type inference, this list is almost certainly incomplete.

3.1. Early Work

Alan Borning and Dan Ingalls were among the first to add type inference to an object-oriented language [Borning and Ingalls, 1982]. They created a type

declaration and inference system for Smalltalk-80. Although their primary goal was to allow the static typechecking of Smalltalk code, their system included type annotations for methods and thus yields the software engineering benefits of a static type system. Their system was not as powerful as the one we present here, because it infers only the types of local variables—the programmer must still declare the type of each method. As the first work in the field, their type system was later discovered to be unsound. However, the inference system was successful in lessening the burden of static types on the programmer.

3.2. Inference of Sets of Classes

Much of the type inference literature focuses on inferring the most specific types of a function. This research focuses on proving that an untyped program is typable or on providing optimization opportunities by narrowing the set of classes a variable can hold. As discussed above, inferring sets of classes is undesirable from a software engineering perspective.

This class of type inference literature may still be relevant as a source of algorithms that could be modified to support software-engineering types. Jens Palsberg and Michael Schwartzbach have done a lot of work in this area; one algorithm they developed with Nicholas Oxhøj is described in [Oxhøj et al., 1992]. Ole Ageson extended their work to produce a type inference system for Self [Agesen, 1995]. David Grove and the Cecil group applied these methods successfully in an optimizing compiler for Cecil [Grove, 1995].

3.3. Is Inference Feasible?

Cardelli and Wegner developed a strongly typed language called Fun [Cardelli and Wegner, 1985], integrating type inference with subtyping and a powerful type system called bounded quantification. In 1992 Benjamin Pierce proved that their type inference system was undecidable [Pierce, 1992] and there was speculation that type inference would be impossible in any reasonably powerful object-oriented type system. However, Vorobyov showed that a more general type inference system was decidable [Vorobyov, 1994], indicating that inference of powerful type systems is not necessarily infeasible.

More recently, the Hopkins Object Group has experimented with recursively constrained types [Efrig et al., 1995], which are even more general than F-bounded types. Their inference algorithm is capable of verifying the type safety of a very large class of dynamically typed programs. However, the recursively constrained types it infers are very difficult to understand, making them less useful for software engineering purposes. Furthermore, typechecking is done by translating an object-oriented program to a procedural language and then verifying that the translated program can be typed, meaning that the types inferred for the procedural language may have little relation to the original program.

3.4. Typed Smalltalk

The Typed Smalltalk project, headed by Ralph Johnson and Justin Graver, designed and implemented a static type specification and inference system for Smalltalk [Graver, 1989]. Their inference algorithm is intended to automatically infer types for non-primitive methods, local variables, blocks, and literals. In their survey this includes over 95% of all Smalltalk methods and 90% of variables. The type system focuses on guaranteeing type safety and improving performance, but a secondary goal is the documentation of method interfaces. As in our proposal, programmers may supply types to supplement those inferred by the system.

Typed Smalltalk includes parameterized types, sets of types, block or closure types, a *selftype*, and some minor variants of structural types. While this type system offers significant power (it's more powerful than the type system in our initial inference model, described below) it falls short of the F-bounded types necessary to typecheck many object-oriented systems. Also, in Typed Smalltalk the goal of supporting optimization conflicted with the software engineering goals, since optimization requires specific types and extensibility demands general ones. For this reason, we suggest that type inference focused on extensibility be kept entirely separate from type inference used to optimize program execution.

3.5. Recent Related Work

O'small [Hense and Smolka, 1993] is a minimal object-oriented language that includes inference of principal types, similar to that of ML. It appears they use record subtyping, which isn't as expressive as abstract typing from a software-engineering perspective.

Objective Caml [Rémy and Vouillon, 1998] is an object-oriented extension to ML. It includes principal types, structural subtyping, multiple inheritance and parameterized types. Polymorphic types are allowed for functions (as in ML), but not for methods of objects. The language has some nice object features, but its object features seem somewhat low-level, almost like those provided by C++. Caml's type inference system is based on HM(X), a theoretical framework described in [Läufer and Odersky, 1994]. Two other proposals to add objects to ML, one by Dominic Duggan [Duggan, 1995] and the other named Object ML [Reppy and Riecke, 1996], provide weaker kinds of type inference.

Local Type Inference [Pierce and Turner, 1997] is an attempt to provide partial type inference support for new languages such as Pizza [Odersky and Wadler, 1997] that include powerful type systems and higher-order functions. Their system infers the types of local variables and anonymous blocks, while still requiring the programmer to provide types for top-level function definitions. They seem to be successful at achieving this goal—in fact, they are able to use algorithms that rely only on information propagated from adjacent nodes in the syntax tree, ensuring that inference is simple, quick and decidable. It would be nice, however, if their system could be extended to infer the types of top-level functions as well as local variables and blocks.

3.6. How our Research Differs

All of the related work we studied uses some type of structural subtyping, ignores the documentation role of types, or does not allow inferring the types of top-level methods. Believing that abstract types are an important documentation tool and that inferring method types represents a significant productivity gain, we differentiate our research by focusing on these issues.

4. A Formal Problem Specification

A *type* may be a *simple type* (an identifier) or *a closure type*, defined as (t1 * t2 * t3 * … * tn-1) -> tn where t1..tn are other types. We say that t1..tn-1 are the *argument types* of the closure type, and tn is the *return type* of the closure type. We define a special simple type **undefined**, not permitted in closure types. Let *T* be the set of all simple types.

Let *subtypes*(t1, t2) be a reflexive transitive relation on *T* that models subtyping. We have the usual contravariant rule: for closure types t1 and t2 we have subtypes(t1, t2) iff subtypes(returntype(t1), returntype(t2)) and for all arguments i subtypes(argument(t2,i), argument(t1,i)). We insist that the subtype relation define a partial order on *T*.

Let a *signature* be a tuple (name, type) where name is an identifier and type is a closure type. Let *S* be the set of all signatures in a program. We require that for all pairs (s1, s2) of signatures in *S* if any argument type of s1 is a subtype of the corresponding argument type in s2 then the return type of s1 is a subtype of the return type of s2. This requirement guarantees that when calling a method we only need to consider signatures with argument types more general than the types of the actual parameters at the call site.

Let a *variable* be a tuple (name, type) where name is a unique identifier. We assume that variables are renamed to eliminate scope hiding. Let *V* be the set of all variables in the program.

Let a *method call* be a tuple (name, list of variables). We define the relation *matches*(call, sig) to be true iff the method call and the signature have the same name and the type of each variable in the method call is a subtype of the corresponding argument type in the signature. Note that in the case of invoking a closure, name may identify a variable, not a method name. There is a special method call of the form (**closure**, list of variables and return type). This form returns a closure with argument types corresponding to the types of the first few variables in the list and a return type corresponding to the last type in the list. Since this model is not concerned with computation, only type inference, the code in the closure is not important.

Let a *statement* be a tuple (variable, method call) or the special form (**return**, variable).

Let a *method* be a tuple (name, list of variables, return type, list of statements). A set of methods *M* is *mutually recursive* if for each statement in each method in *M* there exists either a signature in S with the right name and number of arguments or such a method in *M*, and no strict subset of *M* fulfills this condition.

Given the above definitions and a set of mutually recursive methods *M*, assign a type to every variable in *V* that appears in *M* and currently has type **undefined** such that the following conditions hold:

- We add a signature for each method in *M* to *S* such that the condition on *S* continues to hold and the signature is a closure type constructed from the types of the arguments and return value of the method.
- For each statement stmt in each method in *M* there exists at least one signature sig in *S* such that matches(stmt.call, sig)
- For each statement stmt in each method in *M* for each sig in *S* such that matches(stmt.call, sig) we have subtypes(sig.returntype, statement.variable. type).
- All variables in return statements have types that are a subtype of the return type of the corresponding method.
- No other assignment exists that fulfills the above conditions and assigns a supertype to one argument of one method in *M* and assigns a supertype or equal type to all other arguments.
- No other assignment exists that fulfills the above conditions and assigns a subtype to the return type of one method in *M* and a subtype or equal type to all other return types.

4.1. A Simple Example

Figure 5 is a definition of the factorial method in some object-oriented language that matches our model: it provides subtyping, type inference, and closures.

```
method factorial(n)
  if (n = 1)
    then [return 1]
    else [return n * factorial(n-1);]
```

Figure 5. The factorial function.

To translate this definition to our formal model, we need to convert it into a series of simple statements. Since the statement order does not matter in our model, we can break down the syntax of factorial in any way we like. In this case, we reduce the code from inside out, following the usual applicative evaluation order.

```
factorial(n)
  t1 = equals(n,1)
  return 1
  t2 = closure(1)
  t3 = minus(n,1)
  t4 = factorial(t3)
  t5 = times(n,t4)
  return t5
  t6 = closure(t5)
  t7 = ifthenelse(t1,t2,t6)
```

Figure 6. Factorial in our model

Figure 6 shows factorial written in the format specified by our formal model. First, we assign the result of (n=1) to a new temporary variable t1. For simplicity, we will treat the constant 1 as a variable of type int; clearly this is equivalent to a new, constant variable c1 with the value 1. The inside of the closure (return 1) is encoded as a single statement. Then we must create a closure (to be passed to the if statement) that takes 0 arguments and returns the type of variable 1. This statement will be of the form t2 = **closure**(1). Next we will create a temporary t3 to hold the result of n-1. We apply factorial to this temporary to produce a new result t4. The multiplication with n leads to another result t5, which is returned from the second closure in the if statement in the form t6=**closure**(t5). Finally, we apply the ifthenelse function to t1, t2, and t6 and then return the result of the ifthenelse application.

```
boolean equals(object, object)
number minus(number, number)
number times(number, number)

void ifthenelse(boolean,
       closure(void), closure(void))
```

Figure 7. Signatures needed by factorial.

To infer a type for factorial, there must be other methods in the system including equals, minus, times, and ifthenelse. Figure 7 shows possible signatures for these other methods.

Of course, there are likely to be other types (such as int) and other signatures for these functions. Probably equals is defined for every type, and minus and times have signatures just for int and float types as well as the more general number signature.

In general, factorial could have many types, including int -> int, number -> number, and int -> number. Of these, our formal definition suggests that number -> number is the best type, because number is a more general argument type than int. This matches well with the programmer's intuition that factorial can sensibly be applied to any kind of number, yielding another number in return.

5. A Constraint Based Algorithm

The requirements listed in the formal model above suggest that type inference can be posed as an optimization problem under certain type constraints. Generating all possible type assignments that fulfill the proper type constraints and then choosing one with general argument types and a specific return type satisfies our model. The constraints we need are easy to encode in a logic programming language, and so this constitutes our initial implementation of the algorithm.

5.1. Simple Constraints for Type Inference

For each signature we can create a simple constraint that ensures that the actual parameters in a method call have a type that is a subtype of the corresponding parameter type in the signature. Several signatures with the same name correspond to alternate constraints, one of which must be true for the method call to be well-typed. The signatures also constrain the variable receiving the result of the method call to be a supertype of the result type of the signature. Figure 8 shows two constraints for the times signature. One signature constraints its arguments m and n to be subtypes of int and its result r to be a supertype of int. The second signature is more general, accepting two arguments that subtype number and putting the result in a supertype of number. When solving the constraints for a method call to times, either signature constraint may be chosen to satisfy the type system requirements.

```
times(n,m,r) = subtype(n,int)
      and subtype(m, int)
      and subtype(int,r)
times(n,m,r) = subtype(n,number)
      and subtype(m, number)
      and subtype(number, r)
```

Figure 8. Constraints representing times signatures.

For each statement in each method, we set up a constraint between the argument types and return types to ensure that they match some signature with the appropriate name. If we will be multiplying two numbers n and m and putting the result into r, then we have a constraint times(n,m,r) that can be satisfied by either of the constraints in figure 8.

For statements that create closures and assign them to variables, we must create a special constraint that the result variable is a supertype of the closure type created. For any recursive call, we insist that the actual parameters must subtype the formal parameter types, and the return variable must be a supertype of the formal return type. Finally, for each return statement, we require that the return type of the corresponding method is a supertype of the type of the variable being returned.

5.2. A Set of Constraints for Factorial

Figure 9 shows how the definition of factorial maps into constraints on the types of variables in factorial.

```
factorial(n,r)
  equals(n,int,t1)
  subtype(int,r)
  subtype(closure(int), t2)
  minus(n,int,t3)
  subtype(t3,n)
  subtype(r,t4)
  times(n,t4,t5)
  subtype(t5,r)
  subtype(closure(t5), t6)
  ifthenelse(t1,t2,t6,t7)
```

Figure 9. Constraints for the factorial function.

In the method calls (for example, equals(n,int,t1)) the first few parameters are arguments and the last parameter is the return value. For the recursive call to factorial with temporary t3 and result variable t4, we require that t3 is a subtype of n and that r is a subtype of t4.

The return 1 statement is encoded as subtypes(int,r) to indicate that the formal return type r of the method must be a subtype of int for this statement to be legal. Immediately afterward, the t2 = [return 1] statement is encoded as subtypes(closure(int), t2) to indicate that the creation of the closure is only legal if t2 is a supertype of closures returning ints. Finally, the if-then-else statement is encoded as a simple method call that takes three arguments.

5.3. A Translation into Prolog

It is very straightforward to translate the above constraints into Prolog, and Appendix A lists a prolog program that determines an ideal type for the factorial function. The file declares the type hierarchy declaring an *inherits* constraint between each type and the types it inherits from. Next, it defines the *subtype* constraint as the reflexive transitive closure of the *inherits* relation. That is, each type subtypes itself, any type it inherits from, and any type further up the inheritance hierarchy. The *subtype* relation is extended to handle closure types using the contravariant rule.

Each signature is encoded into its own constraint as described in section 5.1, and a constraint definition for factorial is created almost identical to the specification given in figure 9. The end of the prolog file contains some simple code to extract the type with most general arguments and (given equivalent arguments) the most specific return type.

The answer(A) constraint is set up so that A will be the best type assignment of factorial. When queried, the prolog system computes for a few tenths of a second and then happily responds with A = pair([number], number). This indicates the number -> number type that we inferred earlier to be the best possible type of factorial according to our formal model. Appendix B shows a sample execution of the inference program for factorial.

5.4. An Analysis of our Implementation

The Prolog implementation has the virtue of being very simple. Since our constraints essentially correspond to three-address code, writing a translation from a high-level object-oriented language to the Prolog construction given here requires little more than a simple parser. Because we derive our constraints directly from the formal model, it is clear that our implementation is correct (given a Prolog interpreter that can solve the constraints!). Our example did not include mutually recursive functions, but the our technique can be easily extended to include this problem.

Unfortunately, this implementation may be very inefficient. In a poor implementation of Prolog, an exponential number of possible type combinations could be created and before the illegal ones are eliminated. This explosion in the size of the search space may lead to unacceptably long running times. If we programmed the constraint solver ourselves, we could use our knowledge

of the problem space to avoid generating type combinations that are clearly impossible.

Because many if not all Prolog interpreters use recursive descent techniques to solve constraints, it is also possible that the Prolog interpreter may never terminate. For example, the factorial example does not converge to a solution in our Prolog interpreter if we change the order of the constraints. One problem is that any underconstrained type variable may have an infinite number of possible values, since it could be a closure type with arbitrarily many parameters. In practice, no type variable can have a closure type with more parameters than the maximum size of closures created in the program or present in the signatures. This is clear because if no closure of the proper size is ever created or present in a signature, we can neither use nor assign anything to a variable with that closure type. Thus while the type search space in any real situation is finite, the Prolog interpreter may nevertheless try to search an infinite space.

6. Future Work

The constraints outlined above can probably be solved more efficiently using a solver written with the problem space in mind. Thus it would be beneficial to write an optimized type inference constraint solver in some imperative language to improve the shortcomings of the Prolog implementation.

Since the number of possible type combinations may be exponential in the number of type variables in the worst case (and thus exponential in the length of the input function) the running time of any exhaustive search algorithm will be long in the worst case. Thus, a practical system will have to give up after a certain period of unsuccessful searches. In this case, the best solution found at that point could be returned, or the programmer could be asked to add type annotations to that method. Practical use will determine how often this case occurs—if it is sufficiently infrequent, our system will be successful in lightening the burden that static types impose upon the programmer.

If the running time of the exponential algorithm is often too slow to be useful, we could approximate the solution by using heuristics to guess which type combinations are most likely to be correct solutions to the formal type inference problem. It is likely that heuristics could expand the number of functions whose type can be practically inferred.

A central element of this paper's thesis is that type inference for software engineering is most helpful when it is assimilated into an incremental programming environment like the Smalltalk browser. One relevant piece of future work is to integrate our type inference algorithm into a graphical incremental programming environment for a real language such as Java.

Although type inference is useful even for the (relatively weak) type system we have proposed, it would be most effective in the advanced type systems being developed today. These systems include features like parameterized types and F-bounded polymorphism, which are known to increase the complexity of type inference. The most important area of future work is to develop algorithms that can infer these more powerful types.

7. Conclusions

We have discussed how a type inference system could be useful for software engineering purposes, and suggested that type inference should be integrated into an incremental development environment like the Smalltalk browser. A formal model of a simple type inference system leads to a constraint-based algorithm with a straightforward implementation in Prolog. Future work in type inference systems for software engineering includes improving our algorithm, integrating it into a development environment, and extending it to handle more advanced type systems.

Acknowledgements

Special thanks to Alan Borning for his suggestions and help in ferreting out related work. Craig Chambers also helped to focus and direct this project. Thanks to Vasily Litvinov, Todd Millstein, and other members of the Cecil team who provided constructive criticism. Patrick Logan and Elliot Silver suggested pointers to related work via the Internet. This graduate student is supported by the Department of Defense through a National Defense Scholarship and Education Grant.

References

[Ageson, 1995] Ole Agesen, *Concrete Type Inference: Delivering Object-Oriented Applications*, Ph.D. thesis, Stanford University, December 1995.

[Borning and Ingalls, 1982] Alan H. Borning and Dan H. Ingalls. *A Type Declaration and Inference System for Smalltalk*, Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, pp. 133-141, Albuquerque, New Mexico, January 1982.

[Cardelli, 1984] Luca Cardelli. *A Semantics of Multiple Inheritance*, Proceedings of the Conference on Semantics of Data Types, France, 1984.

[Cardelli and Wegner, 1985] Luca Cardelli and Peter Wegner. *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, 17(4), December 1985.

[Canning et al., 1989] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. *F-Bounded Polymorphism for Object-Oriented Programming*, Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, pp. 273-280, September 1989.

[Duggan, 1995] Dominic Duggan. *Polymorphic Methods With Self Types for ML-like Languages*, Technical report CS-95-03, University of Waterloo, 1995.

[Eifrig et al., 1995] J. Eifrig, S. Smith, V. Trifonov. *Sound Polymorphic Type Inference for Objects*, OOPSLA'95 Object-Oriented Programming Systems, Languages and Applications, pp. 169-184, 1995.

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

[Grove, 1995] David Grove. *The Impact of Interprocedural Class Inference on Optimization*, Proceedings of CASCON'95 Centre for Advanced Studies Conference, pp. 195-203, Toronto, Ontario, Canada, November 1995.

[Hense and Smolka, 1993] A.V. Hense and G. Smolka. *Principal Types for Object-Oriented Languages*, Technischer Bericht Nr. A 02/93, Universität des Saarlandes, Im Stadtwald 15, 6600 Saabrücken 11, Germany, June 1993.

[Jaffar et al., 1992] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H.C. Yap. *The CLP(R) Language and System*, ACM Transactions on Programming Languages and Systems, Vol. 14, No. 3, July 1992.

[Läufer and Odersky, 1994] Konstantin Läufer and Martin Odersky. *Polymorphic Type Inference and Abstract Data Types*, Transactions of Programming Languages and Systems, 1994.

[Mairson, 1990] Harry G. Mairson. *Decidability of ML Typing is Complete for Deterministic Exponential Time*, 17th Symposium on Principles of Programming Languages, ACM Press, January 1990.

[Milner et al., 1990] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*, The MIT Press, 1990.

[Odersky and Wadler, 1997] Martin Odersky and Philip Wadler. *Pizza into Java: Translating Theory into Practice*, Principles of Programming Languages (POPL), 1997.

[Oxhøj et al., 1992] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. *Making Type Inference Practical*, in ECOOP'92, Sixth European Conference on Object-Oriented Programming, pp. 329-349, Utrecht, The Netherlands, June 1992.

[Palsberg, 1996] Jens Palsberg. *Type inference for objects*, ACM Computing Surveys, 28(2):358-359, June 1996.

[Pierce, 1992] Benjamin C. Pierce. *Bounded Quantification is Undecidable*, in 19th ACM Symposium on Principles of Programming Languages, pp. 305-315, 1992.

[Pierce, 1997] Benjamin C. Pierce and David N. Turner. *Local Type Inference*, Indiana University CSCI Technical Report #493, October 1997.

[Rémy and Vouillon, 1998] Didier Rémy and Jérôme Vouillon. *Objective ML: An Effective Object-Oriented Extension to ML*, to appear in Theory And Practice of Objects Systems, 1998.

[Reppy and Riecke, 1996] John H. Reppy and Jon G. Riecke. *Classes in Object ML*, presented at the FOOL'3 workshop, July 1996.

[Vorobyov, 1994] S. Vorobyov. *Fsub: Bounded quantification is NOT essentially undecidable*, Technical Report CRIN—94—R—018, Centre de Recherche en Informatique de Nancy, January 1994.

Appendix A: A Prolog Program for Type Inference of Factorial

```
      /* The simple type declarations */
inherits(void,[]).
inherits(object,[void]).
inherits(boolean,[object]).
inherits(number,[object]).
inherits(ord,[object]).
inherits(int,[ord,number]).
inherits(float,[number]).

      /* A membership test, or "pick a member" constraint */
partof(A, [A | _]).
partof(A, [_ | XS]) :- partof(A,XS).

      /* Subtyping for simple types--transitive closure of inherits */
subtype(A,A).
subtype(A,B) :- inherits(A,C), partof(D,C), subtype(D,B), not B=closure(_,_).

      /* Subtyping for closure types */
subtype(closure(L1,R1), closure(L2,R2)) :-
      subtype(R1,R2), subtype_list(L2,L1).

      /* Tests that each type in A is a subtype of that in B */
subtype_list([],[]).
subtype_list([A | R1], [B | R2]) :-
      subtype(A,B), subtype_list(R1,R2).

      /* The functions that can be called, with their type declarations. */
minus(A,B,R) :- subtype(A,number), subtype(B,number), subtype(number,R).
minus(A,B,R) :- subtype(A,float), subtype(B,float), subtype(float,R).
minus(A,B,R) :- subtype(A,int), subtype(B,int), subtype(int,R).
times(A,B,R) :- subtype(A,number), subtype(B,number), subtype(number,R).
times(A,B,R) :- subtype(A,int), subtype(B,int), subtype(int,R).
times(A,B,R) :- subtype(A,float), subtype(B,float), subtype(float,R).
equals(A,B,R) :- subtype(A,object), subtype(B,object), subtype(boolean,R).
ifthenelse(B,T,E,R) :- subtype(B,boolean),
      subtype(T,closure([],R)), subtype(E,closure([],R)).

      /* A strictly more general test */
more_general(A,B) :- subtype(B,A), not A=B.

      /* Ensures that the first list is more general than the other: it
       * must be strictly more general in one position and at least equivalent
       * in all the others. */
more_general_list([A | R1],[B | R2]) :-
      more_general(A,B), subtype_list(R1,R2).
more_general_list([A | R1],[B | R2]) :-
      subtype(B,A), more_general_list(R1,R2).
```

```
      /* The function whose type we are trying to deduce.
       *
       * factorial(n) = if (n=1) then 1 else n*factorial(n-1)
       * N = type of n
       * R = result type
       * n=1 => equals(N,int,T1)
       * n-1 => minus(N,int,T2)
       * factorial(n-1) => subtype(T2,N), subtype(T3,R)
       * n*factorial(n-1) => times(N,T3,T5)
       * if (n=1) then 1 else n*factorial(n-1)
       *     => ifthenelse(T1,closure([],int),closure([],T5),R)
       */
factorial(N,R) :-
      equals(N,int,T1),
      subtype(int,R),
      subtype(closure([],int), T2),
      minus(N,int,T3),
      subtype(T3,N),
      subtype(R,T4),
      times(N,T4,T5),
      subtype(T5,R),
      subtype(closure([],T5), T6),
      ifthenelse(T1,T2,T6,T7).

      /* All possible types for the factorial function */
possible(L) :- setof(pair([N],R),factorial(N,R),L).

      /* True iff the first assignment is strictly better than the second:
       * either one of its arguments is more general and the others are
       * more general or the same, or the argument types are the same and
       * the return type is more specific. */
better_assignment(pair(L1,_), pair(L2,_)) :-
      more_general_list(L1,L2).
better_assignment(pair(L,R1), pair(L,R2)) :-
      more_general(R2,R1).

      /* Picks the best type from a list of possible types */
best_assignment([A, B | R], Best) :-
      not better_assignment(B,A), best_assignment([A | R], Best).
best_assignment([A, B | R], Best) :-
      better_assignment(B,A), best_assignment([B | R], Best).
best_assignment([Best], Best).

      /* Finds all possible types and chooses the best one */
answer(A) :-
      possible(L),
      best_assignment(L,A).
```

Appendix B: Sample Execution of Type Inference of Factorial

```
sanjuan% pl
Welcome to SWI-Prolog (Version 2.9.5)
Copyright (c) 1993-1997 University of Amsterdam.  All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- consult(factorial).
[WARNING: (/homes/iws/jonal/classes/cs505/project/factorial.pl:50)
      Singleton variables: T7]
factorial compiled, 0.00 sec, 12,160 bytes.

Yes
2 ?- answer(A).

A = pair([number], number)

Yes
3 ?- [halt]
sanjuan%
```