Abstract

The telescoping-languages approach achieves high performance from applications encoded as high-level scripts. The core idea is to pre-compile underlying libraries to generate multiple variants optimized for use in different possible contexts including different argument types.

We have previously developed a type inference algorithm that enables type-based specialization. The algorithm infers types over untyped library procedures before actual inputs are known. Type inference is necessary both to determine the minimum number of variants needed to handle all possible uses of the library procedure as well as to statically determine, for each variant, which optimized implementations should be dispatched at each call location.

In this paper, we extend the type inference algorithm to handle user-defined types in an object-oriented scripting language such as Python. Variants will be generated in order to eliminate dynamic dispatch where possible. This will allow for faster runtimes as well as enable more cross-method optimizations. Two problems that arise when handling user-defined types are the lack of a bound on the number of types and the possibility of extensions to the type hierarchy after type inference. We address these problems and describe the type inference solution.

1 Introduction

The telescoping-languages strategy automatically produces optimizing compilers for high-level, domain-specific languages [14]. It pre-compiles the libraries that define the domain-specific operations for the language and generates multiple variants optimized for the different possible uses by future user scripts. The library compiler determines which variants to generate through static analysis and annotations provided by the library writer. The enhanced-language compiler statically re-places calls to the library procedures in the user script with calls to the appropriate library variant. This strategy is shown pictorially in Figure 1.

An important aspect of telescoping languages is that script compilation time remains small. While the libraries may take longer to compile, the cost of that time can be amortized over the lifetime of the library. This is an improvement to existing strategies for library compilation that recompile parts of the library once the calling-context is known, sometimes causing short scripts to take hours to compile. It also has the advantage of achieving similar performance to these strategies.

The telescoping-languages strategy can also aid library developers by allowing them to encode a single version of their libraries in a high-level scripting language. The responsibility of achieving high-performance for the multiple possible uses of the library is then transferred to the compiler.

In previous work, it has been shown that type-based specialization is essential to achieve maximal performance for the libraries [7]. We use a type inference strategy developed for telescoping languages that infers all valid configurations of types over all procedure variables, thus determining which variants are needed to capture all possible uses of the library in terms of types [15]. Traditionally, type inference produces only a single set of possible types for each argument. In order to
determine a minimal set of variants to generate, type inference should result in a set of valid type configurations. Type configurations, or type assignments, give the exact relationship between the types of the variables. This type-inference strategy was demonstrated using MATLAB (TM) as the example prototype language for the libraries, and inferred matrix properties such as size, sparsity pattern, as well as primitive types.

Recently, Python has had growing success among the scientific community. Python is a dynamically-typed, object-oriented language. In order to support object-oriented features in the telescoping compiler, the type-inference algorithm must be extended to handle user-defined types.

By performing type inference in Python, the compiler, in many cases, will be able to eliminate dynamic dispatch. Dynamic dispatch has traditionally caused a degradation in performance. Also, further cross-method optimizations are possible if the result of dispatch is known at compile time.

Our type inference strategy for user-defined types is similar to our previously described strategy for inferring primitive types. However, for primitive types, the type inference engine only needs to handle a bounded number of types. In an object-oriented world, the type lattice can be extended at any point, including after type inference has been performed.

We will first develop the type inference strategy for object-oriented types in Section 2. We will then describe how to handle the problem of extending the type lattice in Section 3. Finally, we will discuss related work and conclude.

2 Algorithm

In this section, we describe a type-inference strategy for the library compiler. We assume that user scripts use only the types defined in the library. We leave extensions to the type hierarchy from user scripts for future work.

In order to infer all possible type configurations, for each method, the type-inference strategy for telescoping languages performs analysis over the whole method at once, rather than iteratively. To achieve this, it first looks at each statement in isolation to determine the constraints on the variables. It then combines the information over the entire procedure to find the type configurations that satisfy all the statement constraints.

To describe the type inference strategy, we will use the Python classes shown in Figure 2 as a running example (taken from [11]). Shape defines a method move.

```python
class Shape:
    origin = (0,0)
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def move(self, delta_x, delta_y):
        self.x += delta_x
        self.y += delta_y
        return self

class Circle(Shape):
    def __init__(self, x=0, y=0, radius=1):
        Shape.__init__(self, x, y)
        self.radius = radius
    def area(self):
        return self.radius * self.radius * 3.14159
```

Figure 2: Example classes written in Python.

Circle is a subtype of Shape and therefore inherits the method move. Circle also defines a method area.

2.1 Type Lattice

In order to perform type inference, we must first construct a lattice of all types used by the libraries. Figure 3 describes the type lattice corresponding to the example shown in Figure 2.

![Type Lattice](image)

Figure 3: Type Lattice for Python with user-defined types.

2.2 Statement Constraints

The return type-jump-function for a method or procedure summarizes the transfer of type information across a procedure. In other words, it describes possible input types and the corresponding output types. The return
type-jump functions for the methods move and area are shown in Figure 4.

Note that the clauses in the return type-jump-function are mutually exclusive. That is, not all the variables can have the same types for any two clauses.

Return type-jump-functions are entered by hand for the primitive operations or for procedures where the source code is unavailable. The return type-jump-functions for the procedures and methods that subsequently call these operations are then inferred from the code as well as library annotations. Every time a method is defined in a new class, the return type-jump-function for that method is updated to reflect the new possible type configurations.

The statement constraints describe the possible types of the variables in terms of ranges on the type lattice. Statement constraints are formed by looking up the entry corresponding to the operation or procedure call in the return-type-jump-function table and replacing the formal argument names with the names of the actual variables involved. Figure 5 gives the statement constraints for move-area using the return type-jump-functions for move and area.

2.3 Solving the Constraints

After the constraints for each statement have been formed, they need to be combined over the whole method or procedure. Once the statement constraints have been formed, our algorithm proceeds in the way described for primitive types in [15]. We provide a brief description of that algorithm here.

First, the statement constraints are combined and represented as a leveled graph. Each clause in each statement constraint corresponds to a node in the graph at the level of the statement number (the statements are numbered according to their order as written in the code). There is an edge from one node to another if the intersection of the ranges is non-empty for all variables involved. No two nodes on the same level will have edges to each other. Note that in Figure 5 there is no edge from node 1b to node 2a.

Once the graph is formed, determining the valid type configurations corresponds to finding n-cliques (complete subgraphs involving n nodes) over the graph, where n is the number of statements. Intuitively, valid type configurations should satisfy one clause from each statement constraint. Therefore, the compiler needs to first determine the sets of nodes, one from each statement constraint, whose clauses do not contradict each other (i.e., each node in the set should have edges to all other nodes in the set). In the example given in Figure 5, the only clique involves nodes 1a and 2a.

Once the cliques have been found, the compiler performs a solving step in which for each clique, the types of the variables in equations are intersected to determine the minimal valid set of types for each variable. What we are left with is the set of valid type configurations. The only valid type configuration for move-area assigns x and y the type Circle and the output type real.

2.4 Complexity

Under practical conditions, the complexity of the algorithm is \( O(l^v n^2) \), where \( n \) is the number of operations and \( l^v \) is the maximum number of nodes in a level. \( v \) is the number of variables in each statement, which is assumed to be small since the number of arguments does not tend to grow with the size of the procedure [8]. \( l \) is the number of types in the lattice. When inferring the primitive types, \( l \) is always constant, which means the actual complexity is \( O(n^2) \). Unfortunately, for user-defined types, this number is unbounded.

Since \( l^v \) represents an upper bound on the number of nodes at a particular level, we can perform optimizations to try to ensure that the number of nodes on a level never reaches these bounds. One optimization we can perform involves using more sophisticated constraints that allow each clause (or node) to encode more information. For example, we could rewrite the return type-jump-function for move as shown in Figure 6. The new return type-jump-function still captures all the information in the original (i.e., the fact that the input and output type must be the same), but it requires one fewer node than the original. In this way, all clauses that describe the same type pattern can be collapsed into a single node.

Therefore, the number of nodes per level corresponds
def move_area(x):
    y = x.move(5,5)  # 1a[(\(\tau^y = \text{Circle}\)) \land (\(\tau^x = \text{Circle}\))] XOR 1b[(\(\tau^y = \text{Shape}\)) \land (\(\tau^x = \text{Shape}\))]
    return x.area()  # 2a[(\(\text{real} \leq \tau^y \leq \text{real}\)) \land (\(\tau^x = \text{Circle}\)])

Figure 5: Example procedure with statement constraints and graph.

to the number of different patterns of the types for a particular procedure, not the number of possible type configurations. This significantly reduces the complexity for our algorithm. Although the number of viable type patterns can still be large, in practice the number of type patterns that apply to a particular method name remains small.

Note that the new constraints will require slight changes in the methods used to determine compatibility as well as to perform the solving phase. Both are straightforward.

2.5 Code Generation

Once the compiler has all of the possible type configurations, it needs to generate variants based on this information. Each type configuration assigns the variable to a particular type variable. The variants generated must cover the ranges for all the type variables for each type configuration. For each variant, the compiler replaces each procedure or method call with a call to the appropriate variant for the types.

Note that the variants to dispatch to are determined by the type of the arguments as well as the type of self.

This straightforward replacement strategy is not able to eliminate all dynamic dispatch. This is primarily due to that fact that a method with the same input types could have different possible output types. We are currently working on optimizations to further reduce the amount of dynamic dispatch.

3 Extending the Lattice

An important feature of object-oriented programming is that the type hierarchy can easily be extended, and by doing so, the types over which methods work are also extended. This poses a problem for the telescoping-languages solution since the type inference algorithm assumes that the return type-jump-functions for the called procedure have been formed before the calling procedure. Adding types after analysis has already been performed on a procedure invalidates the return type-jump-function. Note that for our purposes, types are added only when new libraries that potentially use the existing libraries are added.

class Square(Shape):
    def __init__(self, x=0, y=0, side=1):
        Shape.__init__(self, x, y)
        self.side = side
    def area(self):
        return self.side * self.side

Figure 7: Example type extension.

There are two possible cases to consider. First, if the new types implement methods that have the same type pattern as already-defined methods, all that is required is an extension to the ranges allowed for the type variables in some of the clauses. In this case, the nodes in the graph do not change. However, the edges in the graph could change if extending the ranges makes two nodes that previously contradicted each other compatible. The compiler would need to examine the new edges to see if they allow for more cliques in the graph than were previously possible.

For example, suppose we added a Square class shown in Figure 7. Then the new return type-jump function for area would extend the range of the type variables to include Square.

Second, if the new types define the methods to have different type patterns from the previous definitions of the method, new clauses must be added to the constraint. New nodes will need to be added to the graph corresponding to the new clauses as well as new edges to the rest of the graph.

Obviously, performing type inference again for all methods that are defined in or inherited by a new class as well as all methods that involve the new types indirectly would produce correct results. However, this is more expensive than necessary. One optimization is to store the graph and cliques along with the generated variants. While this could potentially double the storage requirement, it would make updating these constructs to include the new types comparatively efficient.

4 Related Work

Subtyping with union and intersection types express properties similar to those needed in telescoping lan-
guages [20, 17, 4, 9, 22, 21, 18]. However, because the various clauses formed by our system in a statement constraint over the procedures are disjoint, subtyping with intersection types does not apply. The return type-jump-functions are similar to disjoint-union types for procedures, but the relationships between the procedure variables is not expressible using disjoint-union types because disjoint-union types are types of expressions; they are not logical constraints expressing dependencies over multiple program variables.

Soft typing is a strategy for detecting program errors by statically inferring types over programs written in dynamically-typed languages without changing the semantics [5]. In contrast, the primary purpose of our type-inference mechanism is to perform optimization by statically selecting variants. The forms of the type inferred are similar to ours in that union types are used to represent various possibilities during execution. We would be interested in comparing a formulation of soft typing for object-oriented languages to our approach.

There has been a lot of work on statically inferring types for object-oriented languages. We describe the most relevant work here.

Extended static checking [12] has similarities to our type inference since it operates at the procedure level and is partially guided by annotations. However, this system is designed to check for possible errors as opposed to determining type information for specialization purposes. Also, it relies on a full-fledged theorem prover to determine invariants with weak performance guarantees.

Plevyak and Chien developed a type inference strategy, iteration, that uses data-flow information to more precisely infer types [19]. They iteratively determine points in the program where there are imprecisions in the types and focus on these points. We analyze whole-procedures simultaneously to determine exact type configurations.

Agesen developed a cartesian-product algorithm to infer types [1]. The types used in this work have similarities to the types described in this paper, however, they do not express the relationship between the variables of the procedure, but rather operated over expressions to determine types.

The type-based specialization we propose is similar to selective specialization [10]. However, selective specialization uses dynamic profile information to determine which variants to generate.

The result of performing type inference enables transformations such as object inlining [3, 2]. We plan to develop more optimizations based on the information given by the type inference algorithm in future work.

Guyer and Lin have developed an annotation language for guiding optimizations on libraries [13]. An important direction for future work is to determine whether this language could be used to express annotations adequate for our system.

5 Conclusions

This paper extends the type inference algorithm used for library generation in a telescoping compiler to handle user-defined types in languages such as Python. The algorithm infers all possible type configurations in order to determine the minimum number of variants that should be generated to handle all valid types of inputs efficiently as well as the types of the procedure variables in terms of input types and values. The latter information allows the compiler to replace the procedure calls with calls to variants optimized for the given types.

We developed new forms of constraints to better handle the possibility of a large number of user-defined types. We also developed a strategy for updating the type jump-functions after the initial analysis when the type lattice is extended. Both are important to keep the library generation time relatively efficient.

References


