Improved Program Synthesis Through the Use of Packages

[DREU Final Report]

Santiago Gonzalez
Colorado School of Mines
Golden, Colorado
sgonzale@mines.edu

Armando Solar-Lezama
Massachusetts Institute of Technology
Cambridge, Massachusetts
asolar@csail.mit.edu

ABSTRACT

This is the final report of the my research experience at the Massachusetts Institute of Technology (MIT) with Dr. Armando Solar-Lezama under the DREU program. Described herein is a system designed to improve the synthesis-based Sketch programming language (Solar-Lezama, et. al.). We propose the use of a new programming construct, the package, in which behavioral subtyping between packages in an inheritance tree allows for substitutions between simple, and complex variants of a package. Such substitution allows for simpler synthesis, due to a simpler, equivalent implementation, while maintaining performance in the end program. We describe how packages can be easily applied to data structures and various types of libraries.

1. RESEARCH PROBLEM

A common hurdle in the field of constraint-based synthesis involves reasoning about large code bases efficiently. Often, efficient implementations of certain constructs, such as data structures, are more complex and difficult to reason about, for both the synthesizer and the programmer, than equivalent, less-efficient implementations. That is, we want the synthesizer to be capable of reasoning about a complex implementation by representing it in a simpler form that facilitates reasoning.

Consider that a programmer has a relatively large and complex program which uses sets (i.e. unordered collections) in several places and is written for a synthesis based compiler. Sets can typically be implemented in a variety of different ways, each suited to different types of problems. In some cases, the programmer knows that a tree-based set implementation would be ideal. However, in other cases, a hash table based implementation would be best, but the programmer is unsure which set implementation would be optimal, so he (erroneously) chooses the tree-based implementation. Upon compilation, the synthesizer takes a very long time to reason about the program, due to the complexity of the tree-based set. A system which could use a simpler, equivalent set implementation for reasoning while still using the efficient implementation, which could potentially be inferred from analysis, for the compiled program would be very beneficial.

2. BACKGROUND

The Computer Aided Programming group at the Massachusetts Institute of Technology (MIT) has been working on a C-like, constraint-based synthesis programming language, named Sketch for several years. Sketch relies on a SAT solver and a program synthesizer based on a counterexample-guided inductive synthesis (CEGIS) algorithm. CEGIS converges on a correct solution by iteratively generating potential implementations which are then run though a verifier, a better potential candidate can then be produced from counterexample traces upon failure. [10]

Sketch has demonstrated the applicability of constraint-based synthesis to a variety of programming problems, including automated grading for programming assignments [9], machine learning problems such as recommendations within social media [2], and optimization of database-backed applications which use an object-relational mapping layer [3]. Sketch's versatility and flexibility arise from its easy to understand programming model (i.e. the programmer does not need to be well versed in verification proofs), its ability to run on nearly any system due to its cross-platform Java and C++ codebase, and its syntactic similarity to C.

3. APPROACH

We introduce a solution to the problem of reasoning about complex structures by extending the Sketch programming language to include a new programming construct called the package, which has the ability to define subtype relations between related packages. Such subtype relations allow the synthesizer to use an alternative package for analysis which has a simpler, equivalent implementation, while still using the more efficient package for the compiled code. Packages can have hierarchical relationships with each other. As such, packages have similar behavior to conventional classes in object-oriented programming languages, such as Java, with several key differences.

3.1 Package Semantics

While packages have the concept of inheritance and polymorphism, they are not instantiated like classes. Packages serve to contain code, and as such, packages may include a variety of functions and C-like structs, which *can* be instantiated.

The availability of a function or struct (i.e. whether it is private to the package or public to the entire program) within a package is determined by wether all child packages implement that function or struct. For example, assume there are packages A and B, such that $B <: A$, and A contains a function f_1, while B contains a function f_2; f_1 would be private to its package unless B also implemented while f_2 is public since B has no children. Note, that child packages can introduce new functionality to a package, much like classes.

Additionally, a package's structs can not be instantiated from outside of a package, necessitating the use of factory functions, and such a struct's fields can only be accessed from within that struct's containing package. This enforces the use of setter and getter functions to access a struct's data, allowing different, replaceable struct implementations. These semantics essentially simplify and allow the use of structs as data objects whose exact implementations are unknown by the programmer. Furthermore, the synthesizer has the assurance that instances of a struct can be replaced with other package's implementations, provided that the packages vend equivalent behavior.

Packages are best used to represent collections of code with very similar behaviors and differing implementations. Essentially, the Liskov Substitution Principle (LSP) [6] is enforced for both parent-child and child-parent package relationships.

Packages with no parent package and no children packages, termed utility packages, are essentially simple containers for code, much like namespaces and packages in C++ and Java, respectively. Utility packages allow full access to structs' fields since it is impossible for ambiguities and type conflicts to arise, due to the lack of descendants.

3.2 Package Verification

In order to be able to replace a package with one of its descendants, both packages must be verified to be functionally equivalent for a given program. We accomplish such verification through the use of a most general client that aims to generalize the potential usage instances of a package. The most general client is constructed by leveraging nondeterministic unknowns in SKETCH to iterate over all possible function call sequences and meaningful parameters within a closed number of iterations. Primitive types that are returned from functions can simply be compared for equality. Since the fields of a struct can and do vary between two packages, returned structs (including those passed by reference) are stored in a collection. Structs which are parameters to a function are then nondeterministically selected from this collection. Primitive type function parameters are selected from a nondeterministic input array. Preliminary test results have shown that the most general client is able to find all functional discontinuities between two packages in very few iterations.

Further work will involve refining the most general client for a specific program. There may be cases where two packages are functionally equivalent only for the functions used by the program, but not for all functions. As a contrived example, consider that a programmer has a program with packages A and B, such that $B <: A$, and both A and B contain a function f_1 that performs an operation on a number. Assume that B has a more efficient, and complex, implementation that works with all number inputs, while A's implementation only works with even number inputs. In the most general sense, A and B are not equivalent since they fail verification with odd number inputs. However, if the program only uses B with even numbers, the two packages can be considered equivalent for the program.

4. RESULTS AND CONTRIBUTIONS

We have fully defined and integrated the type semantics for the package system into the SKETCH compiler. The package system's type rules have been formalized in standard notation as described in [1]. Package abstract syntax tree (AST) nodes have been extended from simple code containers to support inheritance, public function and struct interfaces, and the parser grammar has been extended to support the Java-like syntax for declaring parent packages. Furthermore, the package system's type-checking has been implemented as a new compiler pass using a visitor pattern as opposed to being integrated into the preexisting type-checking compiler pass.

Due to the slightly unconventional nature of packages, several of the type rules have unique formulations. For example, the type-checking rule for determining the ambiguity of a function call avoids strange, unintuitive situations where the removal of an unused package from the program changes the package from which function is used in an unspecified function call. Such scenarios can arise due to the nature of child packages which allows them to modify a parent package's function's visibility. These would bring about subtle, difficult to debug issues that result in the program's logic changing, hence eliciting the more conservative rule.

We are also successfully able to generate a most general client for a given pair of packages to verify if they are functionally equivalent. During compilation, we have created a compiler pass that generates the most general client by adding an AST for a test harness function to each child package.

Continuing efforts will go towards modifying the synthesizer to reason with simpler implementations and extensive testing, most likely with string libraries and other data structures. Future work could potentially include modifications to the compiler in order to allow it to deduce which package implementation is most efficient for a given usage instance.

5. RELATED WORK

There has been significant work by researchers at ETH Zürich on using the Eiffel programming language with contracts [4] as an alternative approach to this problem; namely, in the field of program verification. [8] The design by contract programming model is a way to achieve modularity and local reasoning by having the programmer write very detailed contracts. Specifically, model-based contracts which support the verification of software modules. [7] Contracts are invariant specifications which cover pre-conditions, known as obligations, and post-conditions, known as benefits. Proponents for design by contract programming argue that rigorous functionality specifications help to bring successful, reusable software to fruition. [5]

6. REFERENCES

[1] L. Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, Mar. 1996.

[2] A. Cheung, A. Solar-Lezama, and S. Madden. Using program synthesis for social recommendations. *CIKM*, 2012.

[3] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. *PLDI*, 2013.

[4] Eiffel Software. *Building bug-free O-O software: An Introduction to Design by Contract.*

[5] J.-M. Jazequel and B. Meyer. Design by contract: the lessons of ariane. *Computer*, 30(1):129–130, Jan 1997.

[6] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *TOPLAS*, 1994.

[7] N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. *VSTTE*, 2010.

[8] N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer. Flexible invariants through semantic collaboration. *International Symposium on Formal Methods*, 19, 2014.

[9] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated semantic grading of programs. *PLDI*, 2013.

[10] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. *PLDI*, 2008.