Towards Subtyped Program Generation in F#

Baris Aktemur*
Özyeğin University
Istanbul, Turkey
baris.aktemur@ozyegin.edu.tr

ABSTRACT

Program Generation is the technique of combining code fragments to construct a program. In this work we report on our progress to extend F# with program generation constructs. Our prototype implementation uses a translation that allows simulating program generators by regular programs. The translation enables fast implementation and experimentation. We state how a further extension with subtyping can be integrated by benefiting from the translation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, interpreters*

General Terms

Languages

Keywords

Meta-programming, program generation, F#, subtyping

1. INTRODUCTION

Program Generation (PG) is the technique of putting code fragments together to construct a program. PG systems use a quotation syntax to denote code fragments and an antiquote syntax to splice fragments into other fragments – similar to quasiquotations (i.e. macros) in Lisp. PG systems typically provide an *eval* operator to execute generated programs at runtime. Because an extra generation stage is introduced, PG is also called *staged* programming.

A major problem in PG is to make sure that generators do not produce ill-typed programs. This is a difficult problem because the program source in PG is not statically available; it is formed at runtime. In this ongoing work, our goal is to integrate a PG type system with *subtyping* into F#.

*Supported by Microsoft Research Software Engineering Innovation Foundation (SEIF) Award, 2010.

TOPI' 11, May 28, 2011, Waikiki, Honolulu, HI, USA

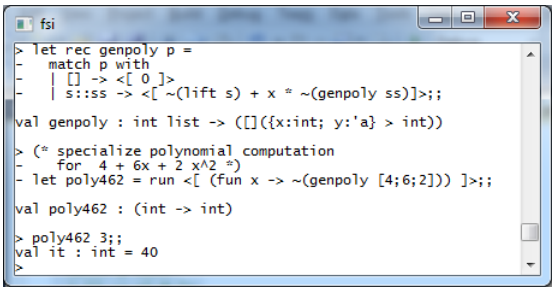

Figure 1: Specialization of polynomial calculation.

We use `<[...]>` syntax as code quotation, `~(...)` as antiquotation. The `run` operator evaluates quoted fragments. The `lift` operator converts values to quoted code. Generating a specialized polynomial calculation function for a particular polynomial is shown in Figure 1, where a polynomial is represented as a list that contains the coefficients.

In our prototype implementation we extended the F# parser in the usual way. For evaluation of fragments we used a translation that allows sound simulation of staged semantics using regular operational semantics [1]. This way, we reuse existing features of the F# interpreter for fast prototyping and add PG constructs incrementally, each time testing a small step. We use the translation to also type-check fragments in the style of λ^{open}_{poly} [2], where a quoted expression is typed as $\Box(\Gamma \triangleright A)$, meaning "the fragment will result in a value of type A if used in an environment Γ." λ^{open}_{poly} uses rho variables for polymorphism of fragments. Because there is no rho polymorphism in F#, we currently assume environments contain two variables –`x` and `y`– as seen in Figure 1 in the type of `genpoly`. To improve expressiveness of the type system, we are enriching it with subtyping. In this ongoing work, again with the help of the translation, we are leveraging subtyping for regular programs to the staged domain. Pottier provides the type system needed for this purpose [3].

2. REFERENCES

[1] W. Choi, B. Aktemur, K. Yi, and M. Tatsuta. Static analysis for multi-staged programs via unstaging translation. *POPL 2011*: 81–92.

[2] I.-S. Kim, K. Yi, and C. Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. *POPL 2006*: 257–268.

[3] F. Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.