Comp 411
Principles of Programming Languages
Lecture 23
Unification

Corky Cartwright

March 29, 2021

An Important Technical Omission

There is a fundamental difference between the simplification of arithmetic expressions taught in grammar school and syntactic semantics taught in this course, which has been glossed over up to this point in the course.

Higher-order vs First-order Expressions (Functional notation)

Expressions are constructed from constants, variables, and operators (which can be converted to constants at the cost of eliminating infix notation).

Conventional arithmetic expressions are *first-order*: functions and operators must be constants (not variables or expressions!). In most functional programming languages (but not mainstream languages including Java!), expressions are *higher-order*: variables can stand for functions (and in some cases operators). Note that the distinction between *first-order* and *higher-order* notation is independent of (orthogonal to) *functional* vs *imperative* meaning. In conventional expression notation, higher-order applications can be hard to read because the function at the head of application can be an expression as in **`(S(K))(K)`**. Functional languages typically use a different syntax in which functions are generally curried and application parentheses are omitted, enabling us to write **`SKK`** instead of **`(S(K))(K)`**.

Computer hardware is *first-order*: the functions (machine instructions) are constants. To encode higher-order expressions, we must use indirection (addresses of function representations, *e.g.*, closures or subroutines) to represent functions (operations) that are not constants.

Functional Languages Can Be First-Order

First-order functional languages are uncommon because this prevents functions from being treated as conventional data values; a function cannot be bound to a conventional variable. Algol-like languages support procedure parameters that are functions but these parameters are not conventional variables. They can only be passed as arguments to other functions; they cannot be used as conventional values, *e.g.*, returned as results of computations.

Syntactic semantics (explicit substitution) can be restricted to first-order languages in which functions are not conventional (sometimes called *first-class*) values. In such a first-order language, we still need a notion of let-binding, but it is not an abbreviation for the application of a λ-expression to argument expressions. In such languages, **`let`** is simply a mechanism for defining scoped (local) constants. In a syntactic semantics, this local constant corresponds to rewrite rule that expands invocations of a locally defined function into instantiations of the body (a restricted form of β–reduction). This form of syntactic semantics does *not* replace function symbols by their corresponding bindings (λ-abstractions) as in Jam. In Jam, the mistake in hand evaluation of forgetting to replace a non-constant function name (which is a variable) by its binding and immediately performing the next step (reducing an argument or performing a β–reduction) corresponds to a first-order view of **`let`**.

My early research on first-order programming logic, where recursively-defined functions are interpreted as definitions extending a first-order logical theory of the data domain is based on this perspective. See https://www.cs.rice.edu/~javaplt/411/19-spring/Readings/RecPrograms.pdf.

First-Order Is Important!

Logicians have deeply investigated the distinction between first-order and higher-order notation. Higher-order logic is not "better" than first-order logic. In fact, there are good reasons to prefer first-order logic (which is "complete") over higher-order logics (which are inherently "incomplete"). For an explanation of this distinction, you should take a rigorous course in mathematical logic. At Rice, only Comp 409 qualifies, but most mathematics departments at major universities offer such a course.

Encoding Higher-Order Expressions as First-Order Expressions

In the abstract syntax for Jam, all Jam expressions have first-order representations even though Jam expressions are higher-order. This encoding relies on a very simple trick: higher order expressions can be encoded as first-order expressions where the application of computed (non-constant) functions is expressed using an explicit *apply* operation that takes the computed function as a value (called the *rator* or *head* of the application). In higher order expressions, variables can be bound to functions and function applications can return functions. Functions are values like numbers or lists. If we introduce an explicit operation to apply functions, then the apply operation is a constant and functions are data values that behave like functions when they are applied. Our abstract syntax for Jam includes several different apply operations (to reduce the number of constant functions) but only one of them involves applying function values (closures).

Unification

The unification problem is very easy to state. Given two expressions **M** and **N**, a *unifier* of **M** and **N** is a substitution **ρ** (an environment binding variables to expressions exactly as in the n-ary generalization of β–reduction) such that **ρ M** = **ρ N**. **M**, **N**, and **ρ** are restricted to expressions constructed from a given finite set of function symbols of specified arity. Note that unification is essence solves the symbolic equation **M** = **N** ignoring the semantics of the constant symbols appearing in **M** and **N**.

Unification is much cleaner (and in my view more elegant) when the expression language is first-order, implying that function symbols must be constants. This restricted version of unification is the standard case in mathematical logic and the only one that we will consider. When the expressions are first-order, there is always most general unifier if any unifier exists. A unifier **ρ** is most general if any other unifier is a substitution instance of **ρ**.

It is straightforward to devise a recursive matching algorithm that is exponential. The computationally complex case arises when unifying two applications of an n-ary function symbol **f**. (Applications of two different function symbols are not unifiable!) The substitutions derived by matching preceding arguments must be applied to subsequent arguments. It is easy to write such code in a functional language like Scheme but it is very slow. Linear algorithms exist although the most practical algorithms are slightly worse than linear (like Union-Find with path compression). See the reference on unification (an article from Computing Surveys) for more details.

Type Reconstruction

The handout on type inference for Polymorphic Jam introduces two different type inference systems for Jam. The simpler one, which we call Typed Jam performs type reconstruction using a type system based on the simply typed λ-calculus. The better one, which we call Polymorphic Jam performs type reconstruction using a type system based on Milner's implicit polymorphism with the polymorphic `let` rule instead of the simply typed `let` rule. Note that both rules support recursive `let` which is almost always the right choice in practice.

To perform type reconstruction for a Typed/Polymorphic Jam program, construct the inference tree corresponding to the program where each binding occurrence of a variable is labeled with a distinct (fresh) type variable. Only one such tree exists because there is one typing rule for each Jam construct, but each rule in the tree imposes constraints on the type variables. To reconstruct the types, we solve this set of equations using first-order unification.