Typechecking, Unification, and Substitution

CS 440: Programming Languages and Translators, Fall 2020

Introduction

• The problem of how to do typechecking for a language with parametric polymorphism was first solved in Standard ML, a cousin of OCaml. The same basic algorithm is used in OCaml and Haskell.

• To solve the typechecking problem, we have to take two parametric types, ask if they're compatible (are they "unifiable"?), and come up with a type that combines the two types (using textual substitution).

Checking Polymorphic Types Using Instantiation and Unification

• Without worrying about what expressions look like, let's look at a system of (syntactically) simple polymorphic types. The most basic types are type variables (α, β, ...) and monomorphic base types like int, char, ....

• To construct new types from old, we can use type operations: For simplicity, let's use:
  • $t_1 \times t_2$, for ordered pairs
  • $t_1 \mid t_2$, for either-or/alternation types
  • $t_1 \rightarrow t_2$, for the functions from $t_1$ to $t_2$.

• Notation: $t_1$ and $t_2$ above aren't type variables; they're names for types. They can stand for type variables, base types, or constructed types. E.g., just saying $t_1 \times t_2$ allows types like int × char, α × int, and $\alpha \times (\alpha \rightarrow \alpha)$ (an ordered pair where the first value is of some type and the second value is a function from that type to itself).

• Definition: Instantiation is the process of substituting a type for a type variable. The type being introduced can be mono- or polymorphic (or a combination). E.g., If we instantiate $\alpha$ in $(\alpha \rightarrow \beta)$ to be int × β, we get $(\text{int} \times \beta \rightarrow \beta)$. If we could instantiate $\alpha$ to $\text{int}$ and $\beta$ to $\text{char}$, then $(\alpha \rightarrow \beta)$ becomes $(\text{int} \rightarrow \text{char})$.

• If a type variable appears more than once, we have to substitute the same type throughout. E.g., $(\alpha \rightarrow \alpha)$ can be instantiated to $\text{int} \rightarrow \text{int}$ and $\text{char} \rightarrow \text{char}$, but not $(\text{int} \rightarrow \text{char})$.

• The typechecking problem: Here is the most basic typechecking problem I know:
  • If type $t_1$ is the type some expression $e$ seems to have and type $t_2$ is the type we expect $e$ to have, then the typechecking question is “Are $t_1$ and $t_2$ compatible?” (The technical name is unifiable.)
  • For polymorphic typechecking, the question translates to “Can we get $t_1$ and $t_2$ to be the same type by instantiating their type variables?”
Case 1: If $t_1$ and $t_2$ are both monomorphic types, then the question turns into “Are $t_1$ and $t_2$ syntactically the same?” E.g., as in `int` and `int` but not `int` and `char`.

Case 2: If $t_1$ and $t_2$ are both built using operators, then it has to be the same operator and the corresponding pairs of component types have to be unifiable. (Note this clause makes the definition of “unifiable” recursive.)

- E.g., to unify the types `int × char` and `$u_1 \times u_2$`, we verify that they both use `×` and then try to unify their left components, `int` and $u_1$, and their right components, `char` and $u_2$.
- (From the previous rule, we’d need $u_1$ to be `int` and $u_2$ to be `char`.)

Case 3: If one of the types, say $t_1$, is a type variable $\alpha$, then typechecking requires instantiating $\alpha$ to $t_2$.

- E.g., unifying $\alpha$ and, say, `char × \delta` makes us instantiate $\alpha$ to `char × \delta`.
- If $t_2$ is a type variable, then we instantiate it to $t_1$.
- This gets done even if both $t_1$ and $t_2$ are type variables (say $\alpha$ and $\beta$), in which case $\alpha$ and $\beta$ have to be instantiated to each other.
  - At some point, we have to make sure we don’t bounce around infinitely from “$\alpha$ is instantiated to $\beta$ which is instantiated to $\alpha$, which ....”

- Note cases 1 and 2 apply to the monomorphic parts of polymorphic types, so the monomorphic case is just the polymorphic case with zero type variables.

**Uses of Unification**

- Unification is used in more than typechecking. For example, it’s critical for executing programs in the language Prolog. (We’ll see this when we study Prolog.)
  - E.g., in Prolog, unifying `likesDogs(fred)` and `likesDogs(X)` and `likesCats(X)` requires $X$ be `fred`, who likes both cats and dogs.
  - In general, unification requires trying to match some syntactic constructs that involve variables, constants, and operations.
    - For types, unification works with type variables, monomorphic types, and the type operators.
    - Unification can also be done on terms that evaluate to values (such as arithmetic terms).
      - E.g., if $X$ and $Y$ are variables for unification, then $X + 3$ and $2 + Y$ unify if we use $2$ for $X$ and $3$ for $Y$. The function call $f(X, Y)$ unifies with $f(2, 3)$, again using $2$ for $X$ and $3$ for $Y$.
        (In these examples, the operators are $+$ and $f$ respectively.)
    - And unification can be done on logical propositional formulas. E.g., $X \land Y > Z$ unifies with $Z = 6 \land 8 > 6$ if we use $Z = 6$ for $X$, $8$ for $Y$, and $6$ for $Z$. (Note the two uses of $Z$ have to be consistent; if we tried unifying $Z = 2$ and $Z = 6$, we’d fail because we can’t unify the constants $2$ and $6$.)
    - Let’s just use one set of notation for all the different areas in which we might do unification.
• **Notation:** Numerals (like 17) and lower-case identifiers are constants; upper-case NAMES like X are variables. Lower case also gets used for function/relation/operator names, e.g., plus(2, 2) and divides(2, 6), but we'll also use infix symbols e.g., 2 + 2. We'll use c, d, ... and x, y, ... to stand for constants; X, Y, ... stand for variables; f, g, ... stand for operators; and s, t, ... refer to terms.

**Substitution**

• To talk formally about unification, first we need to look at substitution, which is the syntactic operation of replacing variables by terms.
  • Instantiating a type variable to a type expression is an example of a substitution.
  • Replacing a parameter variable by an expression when we use referential transparency is an example of substitution. E.g., in Haskell with \( f \ x \ y = x \ (x \ y) \), we get \( f \ \sqrt{16} = \sqrt{\sqrt{16}} \).

• **Notation:** \( s[X \mapsto t] \) is pronounced “s with X replaced by t” or “s with t for X” or “s where X goes to t”, and it means the result of substituting term t for every occurrence of the variable X within the term s. We say that \( X \mapsto t \) is a substitution binding of X to t. We also say that X is bound to t (and that t is bound to X).
  • We can generalize to perform a bunch of substitutions simultaneously, as in \( s[X_1 \mapsto t_1, X_2 \mapsto t_2] \) (the X’s must be unique, but the t’s can use the X’s).
  • E.g., \( (X * Y) [X \mapsto 12, Y \mapsto X + 3] \) is \( 12 * (X + 3) \). (The parentheses are inserted to avoid writing \( 12 * X + 3 \), which doesn’t preserve the original structure of \( X * Y \).)
  • We can also iterate substitutions left-to-right. E.g., \( (X * Y)[X \mapsto Y-5][Y \mapsto 7] \) is \( ((Y-5) * Y)[Y \mapsto 7] \), which is \( (7-5)*7 \).

• **Notation:** \( \sigma \) and \( \tau \) refer to substitutions (single, multiple parallel, or iterated). We write them in postfix.
  • E.g., \( (X * Y) \ \sigma \) where \( \sigma \) is \( [X \mapsto 12, Y \mapsto X + 3] \), or \( (X * Y) \ \tau \) where \( \tau \) is \( [X \mapsto Y-5][Y \mapsto 7] \).
  • Substitution is written as having very high priority: \( (X * Y) \ \sigma \) has us applying \( \sigma \) to both X and Y, but \( X * Y \ \sigma \) means \( X * (Y \ \sigma) \).

• **Notation: Textual** (a.k.a. syntactic) equality and inequality are written \( = \) and \( \neq \). Textual equality means equality as text, so \( 2 + 2 \) evaluates to \( 4 \) but \( 2 + 2 \) is not \( = 4 \). On the other hand, \( (X+X)[X \mapsto 2] = 2+2 \) because substitution is a syntactic operation.
  • The empty substitution is written \( \emptyset \); it never does anything: \( t \ \emptyset = t \) for all t.

**Carrying out a Substitution**

• **Definition:** (Substitution). Let \( t \) be a term and \( \sigma \) be the substitution \([X_1 \mapsto t_1, X_2 \mapsto t_2, ..., X_n \mapsto t_n]\), then definition of \( t \ \sigma \) is by induction on the structure of terms:
• \( c \sigma = c \), where \( c \) is a constant. E.g., \( 17 \sigma = 17 \) (for all \( \sigma \)). This also holds for named constants like \( \text{int} \); i.e., \( \text{int} \sigma = \text{int} \).

• \( Y \sigma = (t_k) \) if \( Y = \text{some} \ X_k \), otherwise \( Y \sigma = Y \). In the result, the parentheses around \( t_k \) can be omitted if they're redundant\(^1\). Example: \((X*Y)[X \mapsto x-z] = (x-z)*Y \) but \((X-Y)[X \mapsto x-z] = (x-z)-Y = x-z-Y \).

• \((f(t_1, ..., t_n)) \sigma = (f \sigma)(t_1 \sigma, ..., t_n \sigma)\). Example: \(f(X, y)[X \mapsto 17] = f(17, y)\).
  • Infix operators are included in this definition, so \((X+y)[X \mapsto 17] = 17+y\).

• Depending on the application, we might omit \( f \) from the substitution and just use \( f \(t_1 \sigma, ..., t_n \sigma)\). It depends on whether we want to be able to substitute functions or not.

• This definition of substitution is fairly restricted. In general, people look at substituting into logical predicates, which gets complicated because of having quantified variables.
  • \((\exists x \ x = y)[x \mapsto t] = (\exists x \ x = y)\) because the quantified \( x \) is considered to be different from the \( x \) in \( x \mapsto t \).

\(^{1}\) We ignore redundant parentheses when checking for \( = \) or \( \neq \), so \((1*2)+3 = 1*2+3\).