Typechecking, Unification, and Substitution

CS 440: Programming Languages and Translators, Spring 2020

Introduction

- How does typechecking work in a language with parametric polymorphism like Haskell? This problem was first solved in Standard ML. ML is an older functional language; the design of Haskell uses a lot of ML, including an extension of the type and typechecking algorithm ML uses.

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`, $\alpha \times$ `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 α in ($\alpha \rightarrow \beta$) to be `int` × β, we get (`int` × $\beta \rightarrow \beta$). If we could instantiate α to `int` and β to `char` then ($\alpha \rightarrow \beta$) becomes (`int` → `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 `int` → `int` and `char` → `char`). but not (`int` → `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 t₂ 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 α, then typechecking requires instantiating α to t_2.
    - E.g., unifying α and, say, `char` × δ makes us instantiate α to `char` × δ.
    - 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 α and β), in which case α and β have to be instantiated to each other.
        - At some point, we have to make sure we don't bounce around infinitely from "α is instantiated to β which is instantiated to α, 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 s, 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., $X \wedge Y > Z$ unifies with $Z =$ `6` ∧ `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 $X \wedge Y > Z$ and $Z =$ `2` ∧ `8` > `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 and Y stand for variables, f and g stand for operators; 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" 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., $(X_1 * X_2)\ [X_1 \mapsto \texttt{12}, X_2 \mapsto X_1\texttt{+3}]$ is $\texttt{12*}(X_1\texttt{+3})$. (The parentheses are inserted to avoid writing $\texttt{12*}X_1 \texttt{ + 3}$, which doesn't preserve the original structure of $X_1 * X_2$.)
  - We can also iterate substitutions left-to-right. E.g., $(X_1 * X_2)[X_1 \mapsto X_2\texttt{-5}][X_2 \mapsto \texttt{7}]$ is $((X_2\texttt{-5}) * X_2)\ [X_2 \mapsto \texttt{7}]$, which is `(7-5)*7`.

- **Notation:** σ and τ refer to substitutions (single, multiple parallel, or iterated). We write them in postfix.
  - E.g., $(X_1 * X_2)\ \sigma$ where σ is $[X_1 \mapsto \texttt{12}, X_2 \mapsto X_1\texttt{+3}]$, or $(X_1 * X_2)\ \tau$ where τ is $[X_1 \mapsto X_2\texttt{-5}][X_2 \mapsto \texttt{7}]$.
  - Substitution is written as having very high priority: $(X_1 * X_2)\ \sigma$ has us applying σ to both X_1 and X_2, but $X_1 * X_2\ \sigma$ means $X_1 * (X_2\ \sigma)$.

- **Notation**: Textual (a.k.a. syntactic) equality and inequality are written $\equiv$ and $\not\equiv$. Textual equality means equality as text, so `2+2` evaluates to `4` but $\texttt{2+2} \not\equiv \texttt{4}$. On the other hand, $(X\texttt{+}X)[X \mapsto \texttt{2}] \equiv \texttt{2+2}$ because substitution is a syntactic operation.

- The empty substitution is written $\varnothing$; it never does anything: $t\ \varnothing \equiv t$ for all t.

Carrying out a Substitution

- **Definition:** (Substitution). Let t be a term and σ 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:
  - $c\ \sigma \equiv c$, where c is a constant. E.g., $\texttt{17}\ \sigma \equiv \texttt{17}$ (for all σ). This also holds for named constants like `int`; i..e, $\texttt{int}\ \sigma \equiv \texttt{int}$.
  - $Y\ \sigma \equiv (t_k)$ if $Y \equiv$ some X_k , otherwise $Y\ \sigma \equiv Y$. In the result, the parentheses around t_k can be omitted if they're redundant[1] . **Example**: $(X\texttt{*}Y)[X \mapsto \texttt{x-z}] \equiv (\texttt{x-z})*Y$ but $(X\texttt{-}Y)[X \mapsto \texttt{x-z}] \equiv (\texttt{x-z})\texttt{-}Y \equiv \texttt{x-z-}Y$.
  - $(f(t_1, \ldots, t_n))\ \sigma \equiv (f\,\sigma)(t_1\ \sigma, \ldots, t_n\ \sigma)$. Example: $\texttt{f}(X, \texttt{y})\ [X \mapsto \texttt{17}] \equiv \texttt{f(17, y)}$.
    - Infix operators are included in this definition, so $(X\texttt{+y})[X \mapsto \texttt{17}] \equiv \texttt{17+y}$.
  - Depending on the application, we might omit f from the substitution and just use $f\,(t_1\ \sigma, \ldots, t_n\ \sigma)$. 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.
  - (E.g., $(\exists\,\texttt{x}\,.\,\texttt{x} = \texttt{y})[\texttt{x} \mapsto t] \equiv (\exists\,\texttt{x}\,.\,\texttt{x} = \texttt{y})$ because the quantified `x` is considered to be different from the `x` in $\texttt{x} \mapsto t$.)

[1] We ignore redundant parentheses when checking for $\equiv$ or $\not\equiv$, so $\texttt{(1*2)+3} \equiv \texttt{1*2+3}$.

Activity Questions for Lecture 15

Lecture 15.1: Typechecking

1. What are types, monomorphic types, type variables, polymorphic types, and type operators?
2. What is instantiation? For each of the following: Can we do this instantiation? If so, what's the result? If not, why?
   a. α to be `int` → `int` in $\alpha \to \beta$?
   b. `int` to be α in `int` × `float`?
   c. α to be `char` for the first α and `String` to be the second α in $\alpha \to \alpha$?
   d. β to be $\alpha \times \alpha$ in $\alpha \to \beta$?
   e. α to be γ and β to be $\gamma \to$ `int` in $\alpha \to (\alpha \to \beta) \to \beta$?
3. What is the basic typechecking problem for monomorphic types? For polymorphic types? Does one problem subsume the other?
4. Are the following pairs of types compatible? If so, what are the instantiations?
   a. $(\alpha, \beta \times \alpha)$ and (`float`, (`int` → `int`) × `float`) ?
   b. $\alpha \to (\alpha \to \beta) \to \alpha \times \beta$ and `int` → (`int` → `char`) → (`double` × `char`)
   c. $(\alpha, \beta \times \alpha)$ and $(\alpha \times \delta, (\delta \to \delta) \times \gamma)$. (Hint: The simplest solution uses only δ as the result of instantiating the two types.)
5. Name three areas in which unification has been used.

Lecture 15.2: Substitution

1. How is $s[X \mapsto t]$ pronounced, and what does it mean? What kinds of things do s, X, and t stand for here? What is a *variable*? What is a *substitution binding*?
2. What does $t\ \sigma$ mean? What is the precedence of σ here? Is there a difference between $t_1 + t_2\ \sigma$ and $(t_1 + t_2)\ \sigma$ and $t_1 + (t_2)\ \sigma$?
3. What is the difference between $[X \mapsto s, Y \mapsto t]$ and $[X \mapsto s][Y \mapsto t]$? Are they different from $[Y \mapsto t, X \mapsto s]$ and $[Y \mapsto t][X \mapsto s]$ respectively? For each: Can X and Y match? Can X and/or Y occur in s and/or t? What happens if neither appear in s or t?
4. What do $\equiv$ and $\not\equiv$ mean? How are they different from $=$ and $\neq$?
5. What is the result of the following substitutions?
   a. (`a*X` + `f(X+Y*X)`)[`X` ↦ `Z+b`] ?
   b. (`X` + `Y` > `Z`)[`X` ↦ `X*Y`][`Y` ↦ `Z^2`]
   c. (`X` + `Y` > `Z`)[`Y` ↦ `Z^2`][`X` ↦ `X*Y`].
   d. (`X` + `Y` > `Z`)[`Y` ↦ `Z^2`][`X` ↦ `Z`]
   e. (`X` + `Y` > `Z`)[`X` ↦ `Z`][`Y` ↦ `Z^2`]

Solutions to Activity Questions for Lecture 15

Lecture 15.1: Typechecking

1. Types are syntactic items that stand for a collection of values. Monomorphic types are `int` like type constants (they always stand for the same thing). Type variables are types that can stand for other types. Polymorphic types contain type variables.
2. Instantiation is the replacing of a type for a type variable. (The type can be monomorphic or polymorphic.)
   a. Instantiation of α to be `int` $\rightarrow$ `int` in $\alpha \rightarrow \beta$ works and yields (`int` $\rightarrow$ `int`) $\rightarrow \beta$. (The parentheses are necessary.)
   b. Instantiation of `int` to be α can't be done: We can only replace type variables by types.
   c. Instantiation α to be `char` for the first α and `String` to be the second α in $\alpha \rightarrow \alpha$ can't be done because all occurrences of α have to be instantiated to the same type.
   d. Instantiation of β to be $\alpha \times \alpha$ in $\alpha \rightarrow \beta$ works and yields $\alpha \rightarrow \alpha \times \alpha$
   e. Instantiation α to be γ and β to be $\gamma \rightarrow$ `int` in $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$ works and yield

      $\gamma \rightarrow (\gamma \rightarrow (\gamma \rightarrow \texttt{int})) \rightarrow (\gamma \rightarrow \texttt{int})$.

      Note some of the parentheses are optional: The minimal parenthesization is

      $\gamma \rightarrow (\gamma \rightarrow \gamma \rightarrow \texttt{int}) \rightarrow \gamma \rightarrow \texttt{int}$
3. For monomorphic types, the basic typechecking problem is “Are these two types syntactically the same?” For polymorphic types, the problem is “Can these two types be made into the same type by instantiating their type variables?”. The monomorphic parts of two polymorphic types have to be syntactically the same, so the polymorphic case subsumes the monomorphic case.
4. (Typechecking problems)
   a. $(\alpha, \beta \times \alpha)$ and (`float`, (`int` $\rightarrow$ `int`) $\times$ `float`) are compatible; the instantiations are $\alpha \equiv$ float and $\beta \equiv$ `int` $\rightarrow$ `int`.
   b. $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \times \beta$ and `int` $\rightarrow$ (`int` $\rightarrow$ `char`) $\rightarrow$ (`double` $\times$ `char`) aren't compatible; we can start off with $\alpha \equiv$ `int` and $\beta \equiv$ `char`, but then the final `double` $\times$ `char` needs to be `int` $\times$ `char`.
   c. $(\alpha, \beta \times \alpha)$ and $(\beta \times \delta, (\delta \rightarrow \delta) \times \gamma)$ are compatible; we need $\alpha \equiv \beta \times \delta$, $\beta \equiv \delta \rightarrow \delta$, and $\gamma \equiv \alpha$, but we can simplify the result if we expand this to $\alpha \equiv (\delta \rightarrow \delta) \times \delta$, $\beta \equiv \delta \rightarrow \delta$, and $\gamma \equiv (\delta \rightarrow \delta) \times \delta$. These take $(\alpha, \beta \times \alpha)$ and $(\beta \times \delta, (\delta \rightarrow \delta) \times \gamma)$ both to $((\delta \rightarrow \delta) \times \delta, (\delta \rightarrow \delta) \times ((\delta \rightarrow \delta) \times \delta))$.
5. Unification can be done on types, arithmetic expressions (and other expressions that evaluate to values), and logical propositions, to name three areas.

Lecture 15.2: Substitution

1. $s\,[X \mapsto t]$ is pronounced various ways such as “s with X replaced by t”. s and t stand for terms (i.e., expressions), and X stands for a variable, which is a symbol that we can replace by a term. (As opposed to named or manifest constants like `pi` or `17`.) A substitution binding $X \mapsto t$ is a substitution action.

2. Substitutions are written in postfix, so $t\ \sigma$ means the result of applying substitution σ to t. Substitution has high precedence, so $t_1 + t_2\ \sigma$ and $t_1 + (t_2)\ \sigma$ mean $t_1 + (t_2\ \sigma)$. (I.e., we apply σ to t_2 but not t_1). In $(t_1 + t_2)\ \sigma$, we're applying σ to the whole of $t_1 + t_2$.

3. $[X \mapsto s, Y \mapsto t]$ means parallel (simultaneous) application of two substitutions. $[X \mapsto s][Y \mapsto t]$ means two substitutions iterated one after the other. $[X \mapsto s, Y \mapsto t]$ and $[Y \mapsto t, X \mapsto s]$ are the same; $[X \mapsto s][Y \mapsto t]$ and $[Y \mapsto t][X \mapsto s]$ apply the two substitutions in opposite orders, so they can differ in effect. In $[X \mapsto s, Y \mapsto t]$, X and Y can't match; in $[X \mapsto s][Y \mapsto t]$ they can. For both parallel and iterated substitution, X and Y can appear in s and t. If neither X nor Y appear in s or t, then $[X \mapsto s, Y \mapsto t]$ and $[X \mapsto s][Y \mapsto t]$ and $[Y \mapsto t][X \mapsto s]$ all have the same effect.

4. $\equiv$ means syntactic equality, i.e., textual equality; $=$ means "evaluates to the same value". E.g., `2+2` $=$ `4`, but `2+2` $\not\equiv$ `4`.

5. (Calculate substitutions)

   a. (`a*X` + `f(X+Y*X)`)[`X` $\mapsto$ `Z+b`] is `a*(Z+b)` + `f((Z+b)+Y*(Z+b))`. The parentheses around `Z+b` before `+Y` are optional, since addition is right-associative. The others are required.

   b. (`X + Y > Z`)[`X` $\mapsto$ `X*Y`] [`Y` $\mapsto$ `Z^2`] is (`X*Y + Y > Z`) [`Y` $\mapsto$ `Z^2`] is `X*Z^2 + Z^2 > Z`

   c. (`X + Y > Z`)[`Y` $\mapsto$ `Z^2`][`X` $\mapsto$ `X*Y`] is (`X + Z^2 > Z`)[`X` $\mapsto$ `X*Y`] is `X+Y + Z^2 > Z`

   d. (`X + Y > Z`)[`Y` $\mapsto$ `Z^2`][`X` $\mapsto$ `Z`] is (`X + Z^2 > Z`)[`X` $\mapsto$ `Z`] is `Z + Z^2 > Z`.

   e. (`X + Y > Z`)[`X` $\mapsto$ `Z`][`Y` $\mapsto$ `Z^2`] is (`Z + Y > Z`)[`Y` $\mapsto$ `Z^2`] is `Z + Z^2 > Z`.