Typechecking Functions

Kyle Dewey

1 Introduction

All modern programming languages have some concept of a bit of code that can be called on demand. These bits go by a lot of names, each with their own subtly different meanings, including functions, methods, subroutines, and procedures. In this handout, we'll go over how to typecheck functions.

Specifically, we will cover two kinds of functions, which are not mutually-exclusive:

- First-order functions
- Higher-order functions

First-order functions generally exist at the toplevel (or near the toplevel) of a program, and are required to have a name. All functions in C are first-order functions. In contrast, higher-order functions generally do not have an associated name, and these can be created anywhere. Higher-order functions allow us to treat functions as data: whole functions can be passed as parameters, returned from other functions, or saved in a data structure. We assume the reader already has some familiarity with higher-order functions.

The rest of this handout covers how to handle these two kinds of functions.

2 Handling First-Order Functions

We'll first cover how to handle first-order functions, and then add in higher-order functions.

2.1 Syntax

We will first introduce first-order functions. As part of this, we need to modify our syntax, shown below. For convenience, we'll introduce the notation $\vec{a}$, which means we have zero or more a's; this has the same meaning as a^* in standard EBNF, but it avoids some ambiguities.

$$x \in \mathit{Variable} \qquad i \in \mathit{Integer} \qquad fn \in \mathit{FunctionName}$$

$$\begin{aligned}
\tau \in \mathit{Type} &::= \textsf{int} \mid \textsf{bool} \\
e \in \mathit{Exp} &::= x \mid i \mid \textsf{true} \mid \textsf{false} \mid e_1 \mathbin{\&\&} e_2 \mid e_1 + e_2 \mid e_1 < e_2 \mid fn(\vec{e}) \\
s \in \mathit{Stmt} &::= \textsf{let}\ x : \tau = e \mid x = e \\
f \in \mathit{Function} &::= \tau_1\ fn(\overrightarrow{\tau_2\ x})\{\vec{s}\ e\} \\
p \in \mathit{Program} &::= \vec{f}
\end{aligned}$$

Compared to some prior changes, this one is fairly dramatic. Notably:

- We now have function names (fn) as a primitive thing.
- We now have first-order functions (f). First order functions have a return type (τ_1). These take zero or more parameters, each with their own type ($\overrightarrow{\tau_2\ x}$). The body of the function consistes of zero or more statements ($\vec{s}$), followed by a single expression (e). The intention here is that a function returns whatever e evaluates to. By forcing functions to end with an expression, we syntactically ensure that functions always have a return. If we instead made a `return` statement, we'd need to ensure that the function contains a `return` statement; this is doable, and real languages do this, but it's extra relatively uninteresting work.

- We now have function calls, which take zero or more parameters $fn(\vec{e})$.
- Programs are now defined as a series of zero or more first-order functions. The intention is that one of these functions (traditionally named `main`) serves as the entry point of the program.

2.2 Type Rules

As before, we will need a type environment mapping variables to types:

$$\Gamma \in \mathit{TypeEnv} = \mathit{Variable} \to \mathit{Type}$$

However, this time around, the type environment alone won't be enough. Consider function calls. When calling a function, we need to know a few things:

- Whether or not the function is defined (its a type error to call a nonexistant function)
- The number of parameters the function takes (if we pass too many or too few, this should be a type error)
- The types of the parameters the function takes (if we pass arguments of the wrong type, this should be a type error)
- The return type of the function (if we pass the right number of parameters of the right types, then we need to know what we get back)

It's most convenient to gather all this information into a single data structure before we execute any typechecking rules. This data structure is commonly referred to as a *symbol table*, which is a fancy term for a data structure that holds program information which can be derived directly from the abstract syntax tree. For our purposes, the following definition suffices:

$$st \in \mathit{SymbolTable} = \mathit{FunctionName} \to \overrightarrow{\mathit{Type}} \times \mathit{Type}$$

The above definition says that a symbol table maps function names to tuples (specifically 2-tuples: pairs). The first element of the pair is a list of types, which encode the parameter types to the function. The second element of the pair is a type, which encodes the return type of the given function. The symbol table can be derived using the following Python-like pseudocode:

```
symbol_table = dict()
for f in functions:
  if f.name in symbol_table:
    raise TypeError("duplicate function name")
  param_names = set([entry.name for entry in f.parameters])
  if len(param_names) != len(f.parameters):
    raise TypeError("duplicate parameter name")
  dict[f.name] = (f.return_type, f.parameters)
```

The above code disallows two functions to have the same name, and similarly disallows two formal parameters of the same function to have the same name. Other than this, it basically just summarizes all the function parameter and return types.

With the symbol table in mind, we can define our typing rules. Throughout the duration of typechecking, the symbol table (st) never changes. As such, we will treat st as if it's a global variable. Pedantically, st should be threaded through all the rules, but because it never changes, this just adds a lot of noise to the rules without giving any real benefit. Without further ado, we define the rules below.

2.2.1 Rules for Expressions

The original rules for expressions are unchanged, though LESS-THAN has been renamed LT for space. We now have a rule handling first-order function calls (FO-CALL), which makes use of st.

$$\frac{}{\Gamma \vdash i : \textsf{int}} \text{ (INTEGER)} \qquad \frac{}{\Gamma \vdash \textsf{true} : \textsf{bool}} \text{ (TRUE)} \qquad \frac{}{\Gamma \vdash \textsf{false} : \textsf{bool}} \text{ (FALSE)}$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \quad \Gamma \vdash e_2 : \textsf{bool}}{\Gamma \vdash e_1 \;\&\&\; e_2 : \textsf{bool}} \text{ (AND)} \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 + e_2 : \textsf{int}} \text{ (PLUS)} \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 < e_2 : \textsf{bool}} \text{ (LT)}$$

$$\frac{x \in \texttt{dom}(\Gamma) \quad \tau = \Gamma[x]}{\Gamma \vdash x : \tau} \text{ (VAR)} \qquad \frac{\begin{array}{c} fn \in \texttt{dom}(st) \quad (\tau_1, \vec{\tau_2}) = st[fn] \\ |\vec{e}| = |\vec{\tau_2}| \quad \overrightarrow{\Gamma \vdash e : \tau_2} \end{array}}{\Gamma \vdash fn(\vec{e}) : \tau_1} \text{ (FO-CALL)}$$

FO-CALL above a lot of work, specifically:

- It checks that the called function (fn) was defined ($fn \in \texttt{dom}(st)$).
- It looks up the expected return type and parameter types of the function ($(\tau_1, \vec{\tau_2}) = st[fn]$).
- It checks that the number of actual parameters matches up with the number of expected parameters ($|\vec{e}| = |\vec{\tau_2}|$). Note that $|a|$ denotes the cardinality of a here.
- It checks that the types of the actual parameters ($\vec{e}$) match up with the types of the expected parameters ($\vec{\tau_2}$).

If any of the above checks fail, the typechecking of the call will fail.

2.2.2 Rules for Statements

The rules for statements are unchanged, and have been duplicated below for convenience. $\vdash_s$ denotes typing rules for statements.

$$\frac{\Gamma \vdash_s e : \tau}{\Gamma \vdash_s \textsf{let } x : \tau = e : \Gamma[x \mapsto \tau]} \text{ (LET)} \qquad \frac{x \in \texttt{dom}(\Gamma) \quad \tau = \Gamma[x] \quad \Gamma \vdash_s e : \tau}{\Gamma \vdash_s x = e : \Gamma} \text{ (ASSIGN)}$$

2.2.3 Rules for Functions

We now need a typing rule that handles functions. Since our functions are all defined at the toplevel, and because we have no global variables in this language, our typing rules for functions do not need to take a type environment Γ; Γ will always be empty initially, as no variables can possibly be in scope. For similar reasons, it doesn't make sense to return a type environment or a type; functions live in the symbol table (st), which exists independently of thiese rules. We define the typing rules below:

$$\frac{\Gamma_1 = \overrightarrow{[x \mapsto \tau_2]} \quad \overrightarrow{\Gamma_1 \vdash_s s} : \Gamma_2 \quad \Gamma_2 \vdash e : \tau_1}{\tau_1 \; fn(\overrightarrow{\tau_2 \; x})\{\vec{s} \; e\}} \text{ (FUNCTION)}$$

The above rule says:

- Construct a type environment (Γ_1) from the parameters to the function. Each parameter name (x) becomes a key, and the parameter's corresponding type (τ_2) becomes a value.
- Using Γ_1 as an initial type environment, typecheck the statements ($\vec{s}$), yielding a new type environment (Γ_2).
- Typecheck the expression (e) underneath Γ_2. The type of e should match up with the return type the user specified (τ_1)

If we apply FUNCTION to each function in the program, then we will end up typechecking the whole program. If any application of FUNCTION fails, then there is a type error in the program.