Today's Lecture Notes for cs173

sk and dbtucker

October 13, 2000

Type Soundness

What can we say about programs that are well-typed? We want a guarantee that the type of a program, as computed by the type checker, will be the type of the result when the interpreter evaluates the program. We call this property *type soundness*:

> **Type Soundness (version #1):** $\forall p \in$ Programs, if the type of p is τ, then p will evaluate to a value of type τ.

What do we mean when we say a value has type τ? A type is a syntactic, static entity, so it's meaningless to talk about the "type" of a value—a value is after all just some sort of sequence of bits on hardware (in most cases). Before we type checked the program, we agreed on some relation between values and types; for example, `1` has type `num`, and `true` has type `bool`. We should explicitly refer to this compact when defining our property:

> **Type Soundness (version #2):** $\forall p \in$ Programs, if the type of p is τ, then p will evaluate to a value which by our prior compact has type τ.

Type soundness relates a type checker and an evaluator. A type system is said to be *sound* with respect to an evaluator if the type soundness property holds.

Not all languages have sound type systems. For example, C's type system is not sound with respect to its evaluator.

Other type errors

So is our type system sound with respect to our interpreter? What happens if we try to evaluate `(first (empty))`? This expression should cause a type error, since it cannot be given any meaningful value. However, according to our type rules:

```
(first (empty)) : num
```

since

```
first : nlist -> num
empty : nlist
```

So in order for type soundness (version #2) to hold, `(first (empty))` must evaluate to a number.

What can we do when we interpret `(first (empty))`? Here are some possibilities:

- return an error value, such as -1. This is a bad idea because it will cause hard-to-catch bugs. For example, `((first (empty)) + 42)` will evaluate to the number `41`, which doesn't make any sense. Also, if you test for emptiness using `(equal? (first (L)) -1)`, the program will act correctly unless you have a list containing the number `-1`.
- go into an infinite loop. This is not a viable solution for a programming language, but we'll discuss later why we might suggest this.
- throw an exception. This the right thing to do, since it explicitly declares this expression to be an error condition, and notifies the programmer of the error.

We'll define our interpreter so that it throws an exception when it evaluates `(first (empty))`. Now we have to rewrite our claim:

> **Type Soundness (version #3):** $\forall p \in$ Programs, if the type of p is τ, then p will either evaluate to a value, which by our prior compact has type τ, or raise an exception.

Recursion

What happens if we add the `rec` construct to our typed language? For example, how do we type check the following expression?

```
(rec length
       (fun (L : nlist)
         (if (empty? L)
             0
             (1 + (length (rest L)))))
  (length (cons 1 (cons 2 empty))))
```

As before, we need to specify the type of the argument to the function. But we still can't type-check the body of the function, since it calls itself, and we don't know its return type. Therefore, we also need to specify the return type. We'll annotate `length` with its type:

```
(rec length : (nlist -> num)
       (fun (L : nlist)
         (if (empty? L)
             0
             (1 + (length (rest L)))))
  (length (cons 1 (cons 2 empty))))
```

Rather than specify the type twice, we'll revise the syntax of `rec` so that it has four parts—the function name, the function argument, the function body, and the expression body. Now we'll only specify the type with the function name:

```
(rec length : (nlist -> num)
     L
     (if (empty? L)
         0
         (1 + (length (rest L)))))
  (length (cons 1 (cons 2 empty))))
```

Now we can type check `rec` expressions. First, we need to check that that the function body has the declared type. Second, we define the type of the expression to be the type of the expression body. The rule is:

$$\frac{\Gamma[\mathtt{v} \leftarrow (q_1 \rightarrow q_2)][\mathtt{a} \leftarrow q_1] \vdash \mathtt{f} : q_2 \quad \Gamma[\mathtt{v} \leftarrow (q_1 \rightarrow q_2)] \vdash \mathtt{b} : t}{\Gamma \vdash (\mathtt{rec}\ \ \mathtt{v} : (q_1 \rightarrow q_2)\ \ \mathtt{a}\ \ \mathtt{f}\ \ \mathtt{b}) : t}$$

Note that we've extended Γ with v *twice*, once for the function body and once for the expression's body.

By adding the `rec` expressions and a type rule for them, we've now added expressiveness to our typed language. Consider the `length` example, except with the call to `rest` removed:

```
(rec length : (nlist -> num)
       (fun (L)
         (if (empty? L)
             0
             (1 + (length L))))
  (length (cons 1 (cons 2 empty))))
```

This expression is still well-typed, but when the interpreter tries to evaluate it, the evaluation will not terminate. Version #3 of the type soundness property has to be rewritten to take into account programs that diverge:

> **Type Soundness (version #4):** $\forall p \in$ Programs, if the type of p is τ, then p will, if it terminates, evaluate to a value which by our prior compact has type τ, or raise an exception.

In our original type system, without the `rec` construct, every computation terminated. We gave an informal argument which said that every function application "uses up" an arrow, and since types have to be finite, we can't have

an infinite number of function applications. So how does the type rule for `rec` allow infinite computations? We noted above that we extend Γ with the type of v twice. The second extension, for the `rec` expression's body, is what allows a recursive use to begin. The first extension, to the function body, is in a scope where the `rec`-bound identifier is bound. (This is different from the scoping rule for `let`.) This allows recursion, even infinite recursion, to proceed.

Now we can revisit an earlier question—what to do with the expression `(first (empty))`. Remember that we said it might make sense to define the evaluation of this expression to be an infinite loop. The reason is that we can assign the expression type `num`, and the type soundness property (version #4) still holds because the evaluation of the expression doesn't terminate.

We don't use this strategy in practice, because it makes programs very difficult to debug. The only way to tell if an error has occurred is if your program appears to be in an infinite loop. Furthermore, you can't distinguish between different kinds of errors.

Logic

The idea of type soundness comes from the study of logic. We encourage the interested reader to consult *A Mathematical Introduction to Logic*, by Herbert Enderton (1972).