Recursive Types and Type Soundness

sk and dbtucker

2002-11-06

1 Recursive Types

1.1 Declaring Recursive Types

We saw in the previous lecture how rec was necessary to write recursive programs. But what about defining recursive types? Recursive types are present all over computer science: even basic data structures like lists and trees are recursive (since the rest of a list is also a list, and each sub-tree is itself a tree).

Suppose we try to type the program

\[
\{\text{rec} \ {\text{length}} : \ ??? \\
\quad \{\text{fun} \ {l : \ ???} : \ \text{number} \\
\qquad \{\text{if} \ {\text{empty?} \ l} \\
\qquad \qquad 0 \\
\qquad \quad (+ \ 1 \ {\text{length} \ {\text{rest} \ l}}))
\}
\}
\]

(We’ve taken generous liberties in this program, assuming the existence of ordinary conditionals and lists of numbers.) What should we write in place of the question marks?

Let’s consider the type of \(l\). What kind of value can be an argument to \(l\)? Clearly a numeric cons, because that’s the argument supplied in the first invocation of \(\text{length}\). But eventually, a numeric empty is passed to \(l\) also. This means \(l\) needs to have two types: (numeric) cons and empty.

In languages like ML (and Java), procedures do not consume arguments of more than one distinct type. Instead, they force programmers to define a new type that encompasses all the possible arguments. This is precisely what a datatype definition, of the kind we have been writing in Scheme, permits us to do. So let’s try to write down such a datatype in a hypothetical extension to our (typed) implemented language:

\[
\{\text{datatype numList} \\
\quad \{[\text{numEmpty}] \\
\quad \quad \{\text{numCons} \ {\text{fst} : \ \text{number}} \\
\quad \quad \quad \{\text{rst} : \ ???})
\}
\}
\]

We assume that a datatype declaration introduces a collection of variants, followed by an actual body that uses the datatype. What type annotation should we place on \(\text{rst}\)? This should be precisely the new type we are introducing, namely \text{numList}.

A datatype declaration therefore enables us to do a few distinct things all in one notation:

1. Give names to new types.
2. Introduce conditionally-defined types (variants).
3. Permit recursive definitions.
If these are truly distinct, we should consider whether there are more primitive operators that we may provide, so a programmer may assemble a datatype if that’s what they need, but they could possibly use the primitives to assemble other types also.

But how distinct are these three operations, really? Giving a type a new name would be only so useful if the type were simple (for instance, creating the name \texttt{bool} as an alias for \texttt{boolean} may be convenient, but it’s certainly not conceptually earth-shattering), so this capability is most useful when the name is assigned to a complex type. Recursion needs a name to use for declaring self-references, so it depends on the ability to introduce a new name. Finally, well-founded recursion depends on having both recursive and non-recursive cases, meaning the recursive type must be defined as a collection of variants (of which at least one is not self-referential). So the three capabilities coalesce very nicely.

1.2 Judgments for Recursive Types

Let's consider another example of a recursive type: a family tree.

```plaintext
{datatype FamilyTree
  {[unknown]
    [person {name : string}
      {mother : FamilyTree}
      {father : FamilyTree}
    ]
  ...
}
```

This data definition allows us to describe as much of the genealogy as we know, and terminate the construction when we reach an unknown person. What type declarations ensue from this definition?

```plaintext
unknown : → FamilyTree

person : string × FamilyTree × FamilyTree → FamilyTree
```

This doesn’t yet give us a way of distinguishing between the two variants, and of selecting the fields in each variant. In Scheme, we use \texttt{cases} to perform both of these operations. A corresponding case dispatcher for the above datatype might look like

```plaintext
{FamilyTree-cases v
  {[unknown] ...]
  {[person n m f] ...])
```

Its pieces would be typed as follows:

\[
\Gamma \vdash v : FamilyTree \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau
\]
\[
\Gamma \vdash \{FamilyTree-cases v \{[unknown] e_1 \} \{[person n m f] e_2 \} : \tau
\]

In other words, to type the entire \texttt{cases} statement to type \(\tau\), we first ensure that the value being dispatched is of the right type. Then we must make sure each branch of the switch returns a \(\tau\).\(^1\) We can ensure that by checking each of the bodies in the right type environment. Because \texttt{unknown} has no fields, its \texttt{cases} branch binds no variables, so we check \(e_1\) in \(\Gamma\). In the branch for \texttt{person}, however, we bind three variables, so we must check the type of \(e_2\) in a suitably extended \(\Gamma\).

Though the judgment above is for a very specific type declaration, the general principle should be clear from what we’ve written. Effectively, the type checker introduces a new type rule for each typed \texttt{cases} statement based on the type declaration at the time it sees the declaration. Writing the judgment above in terms of subscripted parameters is tedious but easy.

Given the type rules above, consider the following program:

\(^1\)Based on the preceding discussion, if the two cases needed to return different types of values, how would you address this need in a language that enforced the type judgment above?
{datatype FamilyTree
  {[unknown]
   [person (name : string)
     {mother : FamilyTree}
     {father : FamilyTree}]}

  {person "Mitochondrial Eve" (unknown) (unknown)}
}

What is the type of the expression in the body of the datatype declaration? It’s FamilyTree. But when the value escapes from the body of the declaration, how can we access it any longer? (We assume that the type checker renames types consistently, so FamilyTree in one scope is different from FamilyTree in another scope—just because the names are the same, the types should not conflate automatically.) It basically becomes an opaque type that is no longer usable. This is not very useful at all.\footnote{Actually … that’s not quite true. You could use this to define the essence of a module or object system. These are called existential types. But we won’t study them further in this course.}

At any rate, the type checker permitted a program that is quite useless, and we might want to prevent this. Therefore, we could place the restriction that the type defined in the datatype (in this case, FamilyTree) should be different from the type of the expression body $\tau$. This prevents programmers from inadvertently returning values that nobody else can use.

Obviously, this restriction doesn’t reach far enough. Returning a vector of FamilyTree values avoids the restriction above, but the effect is the same: no part of the program outside the scope of the datatype can use these values. So we may want a more stringent restriction: the type being different should not appear free in $\tau$.

This restriction may be overreaching, however. For instance, a programmer might define a new type, and return a package (a vector, say) consisting of two values: an instance of the new type, and a procedure that accesses the instances. For instance,

{datatype FamilyTree
  {[unknown]
   [person (name : string)
     {mother : FamilyTree}
     {father : FamilyTree}]}

  with {unknown-person : FamilyTree (unknown)
    {vector
      {person "Mitochondrial Eve"
        unknown-person
        unknown-person}
    {fun (v : FamilyTree) : string
      {FamilyTree-cases v
       [[{unknown}   {error ...}]
        [{person n m f} n]}}}}}

In this vector, the first value is an instance of FamilyTree, while the second value is a procedure of type $\text{FamilyTree} \rightarrow \text{string}$.

Other values, such as unknown-person, are safely hidden from access. If we lift the restriction of the previous paragraph, this becomes a legal pair of values to return from an expression. Notice that the pair in effect form an object: you can’t look into it, the only way to access it is with the “public” procedure. Indeed, this kind of type definition sees use in defining object systems.

That said, we still don’t have a clear description of what restriction to affix on the type judgment for datatypes. Modern programming languages address this quandary by affixing no restriction at all. Instead, they force all type declarations to be at the “top” level. Consequently, no type name is ever unbound, so the issues of this section do not arise.

1.3 Space for Datatype Variant Tags

One of the benefits programmers incur from using datatypes—beyond the error checking—is slightly better space consumption. (Note: “better space consumption” = “using less space”.) Whereas previously we needed tags that
indicate both the type and the variant (as when we wrote 'num-empty and 'num-cons), we now need to store only the
variant. Why? Because the type checker statically ensures that we won’t pass the wrong kind of value to procedures!
Therefore, the run-time system needs to use only as many bits as are necessary to distinguish between all the variants
of at type, rather than all datatypes. Since the number of variants is usually quite small, of the order of 3-4, the number
of bits necessary for the tags is usually small also.

We are now taking a big risk, however. In the liberal tagging regime, where we use both type and variant tags, we
can be sure a program will never execute on the wrong kind of data. But if we switch to a more conservative tagging
regime—where we don’t store type tags also—we run a huge risk. If we perform an operation on a value of the wrong
type, we may completely destroy our data. For instance, suppose we can somehow pass a NumList to a procedure
expecting a FamilyTree. If the FamilyTree-cases operation looks only at the variant bits, it could end up
accessing a numCons as if it were a person. But a numCons has only two fields; when the program accesses the
third field of this variant, it is essentially getting junk values. Therefore, we have to be very careful performing these
kinds of optimizations. How can we be sure they are safe?

2 Formalizing the Type Abstraction

We would like some kind of guarantee that the program will fail to exhibit certain errors (or, to put it more positively,
that it will execute without certain errors, such as inadvertently accessing a NumList as if it were a FamilyTree). In
particular, we would like to know that the type system did indeed abstract over values: that running the type checker
correctly predicted (up to the limits of the abstraction) what the program would do. We call this property of a type
system type soundness:

For all programs \( p \), if the type of \( p \) is \( \tau \), then \( p \) will evaluate to a value that has type \( \tau \).

Note that the statement of type soundness connects types with execution. This tells the user that the type system is not
some airy abstraction: what it predicts has bearing on “practice”, namely program execution.

We have to be somewhat careful about how we define type soundness. For instance, we say above (emphasis added) “\( p \) will evaluate to a value such that . . .”. But what if the program doesn’t terminate? So we must recast this
statement to say

For all programs \( p \), if the type of \( p \) is \( \tau \) and \( p \) evaluates to \( v \), then \( v : \tau \).

Actually, this isn’t quite true either. What if the program executes an expression like (first empty)? There are a few
options open to the language designer:

- Return a value such as \(-1\). We hope you cringe at this idea! It means a program that fails to check return values
everywhere will produce nonsensical results.

- Diverge, i.e., go into an infinite loop. This approach is used by theoreticians (study the statement of type
soundness carefully and you can see why), but as software engineers we should soundly reject this.

- Raise an exception. This is the preferred modern solution.

Raising exceptions means the program does not terminate with a value, nor does it not terminate. So we have to refine
this statement still further:

For all programs \( p \), if the type of \( p \) is \( \tau \), \( p \) will, if it terminates, either evaluate to a value \( v \) such that \( v : \tau \),
or raise one of a well-defined set of exceptions.

The exceptions are a bit of a cop-out, because we can move arbitrarily many errors into that space. In Scheme, for
instance, the trivial type checker rejects no programs, and all errors fall under the exceptions. In contrast, researchers
work on very sophisticated languages where some traditional exceptions (such as violating array bounds) become type
errors. This last phrase of the type soundness statement gives lots of room for type system design.

As software engineers, we should care deeply about type soundness. To paraphrase Robin Milner, who first proved
a modern language’s soundness (specifically, ML),

---

The term “soundness” comes from logic, where it has a very specific meaning. Probably related is the British usage, found in writings of
authors like P.G. Wodehouse, of referring to a person as “a sound egg”.

We could write this more explicitly as: ‘For all programs \( p \), if the type checker assigns \( p \) the type \( \tau \), and the semantics say that \( p \) evaluates to a
value \( v \), then the type checker will also assign \( v \) the type \( \tau \).’
Well-typed programs do not go wrong.

This is partially rhetorical, but there is an underlying truth here: a program that passes the type checker (and is thus “well-typed”) absolutely cannot exhibit certain classes of mistakes. (The choice of the word “wrong” is definitely an exaggeration, since it suggests that types ensure “correctness”. But we should not let this verbal quibble overshadow Milner’s achievement.)

Why is type soundness not obvious? Consider the following simple program (the details of the numbers aren’t relevant):

\{if0 \{+ 1 2\} \\
  \{\{fun \{x : number\} : number \{+ 1 x\}\} 3\} \\
  \{\{fun \{x : number\} : number \{+ 1 \{+ 2 x\}\}\} 1\}\}

During execution, the program will explore only one branch of the conditional:

\[
\begin{align*}
1,\emptyset & \Rightarrow 1 \\
2,\emptyset & \Rightarrow 2 \\
\ldots & \\
\{+ 1 2\},\emptyset & \Rightarrow 3 \\
\{\{fun \ldots\} 1\},\emptyset & \Rightarrow 4 \\
\{if0 \{+ 1 2\} \{\{fun \ldots\} 3\} \{\{fun \ldots\} 1\}\},\emptyset & \Rightarrow 4
\end{align*}
\]

but the type checker must explore both:

\[
\begin{align*}
\emptyset-1&:number \\
\emptyset-2&:number \\
\ldots & \\
\emptyset-{\{+ 1 2\}}&:number \\
\emptyset-{\{fun \ldots\}}&:number \\
\emptyset-{\{fun \ldots\}}&:number \\
\emptyset-{if0 \{+ 1 2\} \{fun \ldots\} 3 \{fun \ldots\} 1\}}&:number
\end{align*}
\]

Furthermore, even for each expression, the proof trees in the semantics and the type world will be quite different. As a result, it is far from obvious that the two systems will have any relationship in their answers. This is why a theorem is not only necessary, but sometimes also difficult to prove.

Type soundness is, then, really a claim that the type system and run-time system (as represented by the semantics) are in sync. The type system erects certain abstractions, and the theorem states that the run-time system mirrors those abstractions. Most modern languages, like ML and Java, have this flavor.

In contrast, C and C++ do not have sound type systems. That is, the type system may define certain abstractions, but the run-time system does not honor and protect these. (In C++ it sort of does for object types, but not for types inherited from C.) This is a particularly insidious kind of language, because the static type system lulls the programmer into thinking it will detect certain kinds of errors, but fails to deliver on that promise during execution.

Actually, the reality of C is much more complex: C has two different type systems. There is one type system (with types such as int, double and even function types) at the level of the program, and a different type system, defined solely by lengths of bitstrings, at the level of execution. This is a kind of “bait-and-switch” operation on the part of the language. As a result, it isn’t even meaningful to talk about soundness for C, because the static types and dynamic type representations simply don’t agree. Instead, the C run-time system simply interprets bit sequences according to specified static types. (Procedures like printf are notorious at this: if you ask to print using the specifier %s, printf will simply print a sequence of characters until it hits a null-terminator: never mind that the value you were pointing to was actually a double! This is of course why C is very powerful at low-level programming tasks, but how often do you actually need such power?)

To summarize all this, we introduce the notion of type safety:

Type safety is the property that no primitive operation is ever applied to values of the wrong type.

By primitive operation we mean not only addition and so forth, but also procedure application. In short, a safe language honors the abstraction boundaries it erects. Since abstractions are crucial for designing and maintaining large systems, safety is a key software engineering attribute in a language. (Even most C++ libraries are safe, but the problem is you have to be sure nobody in some legacy C library isn’t performing unsafe operations.) Using this concept, we can construct the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Statically Checked</th>
<th>Not Statically Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe</td>
<td>ML, Java</td>
<td>C, C++</td>
</tr>
<tr>
<td>Unsafe</td>
<td>Scheme</td>
<td>Assembly</td>
</tr>
</tbody>
</table>
The important thing to remember is, due to the Halting Problem, some checks simply can never be performed statically; something must always be deferred to execution time. The trade-off in type design is to minimize the time and space consumed by these objects during execution (and, for that matter, how many guarantees a type system can tractably give a user)—in particular, in shuffling where the set of checked operations lies between static and dynamic checking.

So what is “strong typing”? As best as we can tell, this is a meaningless phrase, and people often use it in a nonsensical fashion. To some it seems to mean “The language has a type checker”. To others it means “The language is sound” (that is, the type checker and run-time system are related). To most, it seems to just mean, “A language like Pascal, C or Java, related in a way I can’t quite make precise”. For amusement at cocktail parties, when someone mentions the phrase “strongly typed”, ask them to define it and catch the errors and inconsistencies. And please, don’t use the term yourself unless you want to sound poorly-trained and ignorant. Use the terminology of this course instead.