Type systems questions and answers

COMP 105

24 October 2018

Contents

<table>
<thead>
<tr>
<th>Kind overview</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kinds</td>
<td>1</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>2</td>
</tr>
<tr>
<td>More pleasant forms of polymorphism</td>
<td>3</td>
</tr>
<tr>
<td>The interpreter</td>
<td>3</td>
</tr>
<tr>
<td>Arrays</td>
<td>3</td>
</tr>
<tr>
<td>The unit type</td>
<td>3</td>
</tr>
<tr>
<td>Pragmatics</td>
<td>4</td>
</tr>
<tr>
<td>Learning</td>
<td>4</td>
</tr>
<tr>
<td>Unanswerable</td>
<td>4</td>
</tr>
</tbody>
</table>

Kind overview

Several of you asked questions about kinds. To address them, I’ll start with an overview. Here’s the problem that kinds are meant to solve: the kind system is meant to give you, the programmer, the power to create a new type constructor that takes type parameters. Good examples include the list type constructor, an array type constructor, a dictionary type constructor, and other things like that.

- In a monomorphic language, there’s no such thing as a type parameter. So you can’t define a polymorphic type that expects a type parameter. You have to settle for whatever polymorphic things the compiler gives you. For example, in C you get polymorphic pointers and arrays, but you don’t get polymorphic lists or dictionaries.

Monomorphism is not a tragedy. You can still create user-defined types like `exp` and `value`. You just can’t define types that take type parameters.

(In C++, you can code polymorphic types with templates, but a template is its own thing and is different from a type. Compile-time analysis involving C++ templates is insanely complicated.)

- In a civilized polymorphic language, you can create whatever types you want—and they can take type parameters. If you want arrays, you can define arrays that take type parameters.

But by giving you this ability, we make it harder for the type checker to answer the question, “what’s a good type?” In particular, if you claim that a parameter to a function has a given type, how does the type checker know if your claim is any good? How does it know that `exp` is a good type for a parameter, but `(exp bool)` isn’t? How does it know that `(array int)` is a good type for a parameter, but `array` isn’t? This is the problem that the kind system solves.

The same problem shows up in the types from your ML homework: how does the compiler know that a type like `nat * nat -> nat` is OK, and yet another type like `env * env -> env` is not OK? How does it know that `value ref env` is OK, but `value ref nat` is not OK?

In practice, here’s what the kind system does for you: every time you write down the type of a formal parameter, the kind system makes sure the type is meaningful. And in Typed μScheme, it’s all implemented for you: to make sure a type is good, you just call function `asType` in the interpreter.

Kinds

*If lambdas need kind * for arguments, can you pass functions to a lambda in tuscheme?*

Yes. A function type has kind `*`. (The `arrow constructor`, by itself without any argument or result types, has a different kind.)

*What is in the kind environment? Names or types or both?*

Names. More specifically, the name of each type constructor (and its kind).

*Are kind environments to types as expressions are to values? In that the evaluation of a kind is a type.*

No. Kind environments are to types as type environments are to expressions:

- The type environment `Γ` contains the context needed to ensure that the user’s expressions are good.

- The kind environment `Δ` contains the context needed to ensure that the user’s type expressions (as in, the type claimed for a formal parameter) are good.

Finally, kinds and types are never “evaluated.” Only expressions and definitions are evaluated.
Why are kinds either * or some combo of * => *? Does that exhaust all possibilities?

Those are only the common cases: constructors that are types (like int and bool) or constructors that make types from other types (like list and option). There are other possibilities, but they are exotic. You could Google “higher-kindled polymorphism,” but the information you’ll find is not always helpful.

Does it make sense to allow the programmer to extend the kinds available? Similarly to standard ML where the programmers can add new data types.

Yes. I believe the latest Haskell allows the programmer to define new kinds. But I don’t swear to it. If you are interested in this stuff, there is a great paper by Tim Sheard called “Languages of the Future.”

Are there any languages which use a more complicated kind-layer?

Yes. Haskell. Also Tim Sheard’s Omega, which is described in his paper “Languages of the Future.” There are doubtless others that I’m not aware of.

How do we extend the language using kinds? How is it done in code?

You add a new type constructor to the kinding environment, and you give it a kind. You would add a line to the code in chunk 427c.

How do you add a type to the kind environment? Where do you apply the CONAPP constructor?

To add a type to the kind environment, you would add a line to the code in chunk 427c. You would apply the CONAPP constructor anywhere you wanted to build a new type using a non-nullary type constructor, as in the listtype function in chunk 455c.

How would one practically leverage kinds in code? I.e., why is this so useful?

Leverage for programmers is indirect. Kinds are really a tool for the language designer or the compiler writer, and the main benefit is to provide a language that simultaneously supports (a) parametric polymorphism and (b) user-defined types.

Can you explain how kinds are kept track of in Impcore/uScheme?

Impcore doesn’t use kinds. Typed uScheme tracks them in an environment labeled Δ, which is initialized with a collection of primitive type constructors, and which is extended by every type-lambda.

What’s the difference between a kind and a type?

Kinds are simpler.

How do I understand kinds better?

Write code in Typed uScheme that triggers error messages because the kinds aren’t right.

Typed uScheme does not need to define new datatype and syntax; why?

A new type constructor, with the right kinds, and accompanied by primitive functions, is sufficient.

Why is a pair * x * => * when List and Array are * => *? Are they seen more as functions compared to pair?

To build a list type or array type, you need only one type parameter (the type of an element). To build a pair type, you need two type parameters (the type of the left half and the type of the right half.) Type constructors do resemble functions that operate on types—they resemble value constructors even more. But all three have equal “function-ness;” the only difference is in the number of arguments that are expected.

Polymorphism

What is the nuance behind tylambdas not being nestable?

They are nestable; you just can’t “capture” existing type variables. Here’s a forbidden expression:

\[
\text{(type-lambda ['a]}
\text{  (lambda ([x : 'a])}
\text{    (type-lambda ['a]}
\text{      (lambda [y : 'a]
\text{        (if #f x y))))}}
\]

This function, if we allowed it, would allow us to pretend that any two values, of arbitrary types, actually have the same types. Subverting the type system. The nuance: When we accept \(\text{(type-lambda ['a] \ldots)}\), we’re saying, “You can decide later what ‘a is, and it can be anything you want.” But once we have an x of type ‘a, it’s not safe for ‘a to be anything we want: it’s safe only if ‘a is the type of x.

What are the pitfalls with typechecking type-lambda?

The only real pitfall is the nesting issue above. Or you might forget to extend the environment with the new type variables.

How much would need to be added to typed uScheme to avoid @ notation?

Joe Wells showed that the full problem is undecidable. But we can implement an important special case by adding a constraint solver. You will do this next week.

Does the type in the judgment of TYLAMBDA extend an environment (I don’t think it does)? If not, what’s the point of those extra brackets?
The typing judgment above the line extends the kind environment, $\Delta$. It adds all the type parameters so that in the body of the $\text{TYLAMBDA}$ (and only there), they are accepted as good types.

**What does $\tau_1[\alpha_1 \mapsto \tau]$ mean?**

“Type $\tau_1$ with type $\tau$ substituted for every free occurrence of $\alpha_1$.”

**What does Theorem 6.1 in the book mean?**

It says that two polymorphic types are equivalent unless we can find some particular type $\tau$ which, when substituted for the bound type variable, produces new types that are not equivalent. For example, $\forall \alpha. \alpha$ and $\forall \beta. \beta$ are equivalent because if we substitute any $\tau$, we get $\tau$ on both sides. By contrast, $\forall \alpha. \alpha \rightarrow \text{int}$ and $\forall \alpha. \text{int} \rightarrow \alpha$ are not equivalent, because if we substitute $\text{bool}$ on both sides, we get $\text{int} \rightarrow \text{bool}$ and $\text{bool} \rightarrow \text{int}$, which are definitely different.

**Is it possible to have a polymorphic type with restrictions?**

Yes. The most common restriction would be something like “only types that support equality,” or “only types that support arithmetic.” Something like “$\text{int}$ or $\text{bool}$ but not array.”

**What is the purpose of allowing polymorphic functions if you must specify the type anyways?**

Doesn’t this detract from the value of polymorphism? Yes. The most common restriction would be something like “only types that support equality,” or “only types that support arithmetic.” Something like “$\text{int}$ or $\text{bool}$ but not array” is actually quite sophisticated, and I don’t remember what buzzword goes with it. But for starters you could look up “intersection types” and “union types.”

**More pleasant forms of polymorphism**

*If type-checking is delayed, would polymorphic functions be easier to implement? Or dynamic instantiation/substitution?*

If you delay type checking, you lose the benefits. Also, the implementation of polymorphic functions is not so bad—any cost you pay as part of a function’s definition is paid only once. It’s having to instantiate every use of a polymorphic function that is so painful. We’ll see that the ML type system can instantiate these functions automatically.

**Is there some shorthand for the $[@'() \text{bool}]$ notation?**

Alas, no.

**The interpreter**

*Can you explicitly explain what abstract syntax in the interpreter applies to which judgment form? I.e., what applies to types vs kinds vs expressions vs other?*

I’m hoping that the tables on pages 406 and 441 answer this question.

**In comparison $f = \text{binaryOp} (\text{BOOLV } o f)$, what is the last part doing?**

Function $f$ is an ML-function that returns an ML value of type $\text{bool}$. $\text{BOOLV}$ is the value constructor that is applied to an ML value of type $\text{bool}$ to produce a $\mu$-Scheme value. (The ML type of $\text{BOOLV}$ is $\text{bool } \rightarrow \text{value}$.) The composition gives me a function, based on ML function $f$, which returns a $\mu$-Scheme value. Since I am trying to use ML functions to implement $\mu$-Scheme primitives, this back-and-forth is appropriate.

**Arrays**

*Will arrays we add have constant time indexing?*

Yes. They are built on ML primitive arrays, which in turn are built on machine pointers and pointer arithmetic.

**The unit type**

*What is a unit?*

*Could you explain “unit” a little more?*
In a typed language, every function has to have a result type. But some functions are evaluated only for side effects; at run time, they don’t return anything. Such a function still needs a result type, and in the ML family of languages, that result type is called unit. (In C, C++, and Java, that result type is called void.)

I still don’t understand why the function you showed ended with an empty `begin`

I wanted an expression of type `unit`, and I was in too much of a hurry to see if I would get one by calling a function.

Pragmatics

What is the practical difference between types and values (i.e., int vs NUM)

Types happen only at compile time and have no run-time overhead. Applying a value constructor like `NUM` has a small constant cost.

Does the type system add to the run-time of a program? If not, how can it do so?

No. The type system’s work is finished before code generation. Type soundness guarantees that the machine code finds the bits that it is expecting to find.

How does the formation, construction introduction, elimination analysis of a type (or a data structure?) inform code we may write, and is it at the typechecking, extension, or implementation level?

Different answers depending on if it’s a type or a data structure:

- Formation/introduction/elimination frames your understanding of what features a language has and what you can do with them. For example, looking at pointer types, C has an elimination form for pointer types that lets you do arithmetic, but it’s unsafe. Modula-3 has no elimination form for pointer arithmetic, and it’s safe. Rust and other more advanced languages have sophisticated elimination forms for pointer arithmetic, with sophisticated pointer types, with the goal of doing pointer arithmetic safely.

As another example, if you want to understand “ownership” and “borrowing” in Rust, your first questions should be questions about the formation, introduction, and elimination of types related to ownership.

- If you’re designing a data structure, almost certainly an abstraction, then the type analysis relates closely to the classification of operations on page 109 of my book:
  - Creators correspond to introduction forms.
  - Observers and mutators correspond to elimination forms.

If your abstraction defines an “introduction” operation with no corresponding elimination, that’s cause for concern—you would ask why the abstraction lets you make things you can’t look at.

Any improvement from non-typed?

Yes. Eliminated all bugs of the form “wrong number of arguments,” “unbound name,” “variable not defined,” “car applied to non-list,” “arithmetic on non-integer.”

Learning

What was the hardest thing for YOU to wrap your head around when learning this?

I was self-taught on all this stuff, so I didn’t learn it systematically. Here were a few rough spots:

- Understanding the nuances of the type-`lambda` rule, and the reason for the side conditions on free type variables, was hard.
- Capture-avoiding substitution is crazy hard to get right. (Even world experts get it wrong.)
- It took me a long time to grasp the idea that “types classify terms.” I was stuck with an idea that “types are properties of values.”
- I struggled for eons because I couldn’t find the concrete syntax for a type constructor that takes two or more type parameters. Typed μScheme doesn’t have this problem.

Unanswerable

So far, makes sense! I think I maybe don’t yet understand enough to have questions.

We’ll always have Piazza.

I just want to see forall implementation in depth to understand that.

I’m not sure what is asked here.