Outrageously Expressive Types:
Dependent Types
Refinement Types
Gradual Typing
and Hybrid Type Checking

and Why You (Mostly) Don't Have Them Yet

Adrian King
24 March 2014

Where's My Flying Car?

- Why don't mainstream languages have gradual typing, dependent types, refinement types, and hybrid type checking?
- Theory is more complex than parametric polymorphism
- In the general case, typechecking is undecidable
- Concise syntax yet to be discovered

Part I: Dependent Types

In the dependently typed lambda calculus:

- Types are terms. That is, type expressions evaluate to first-class values, just like any other expression.

- Types can depend on other terms, including terms that are not types.
- These features make dependent typing radically more expressive than simpler, more familiar type systems.

Everyone's Favorite Dependent Type Example: Vectors

Consider this function signature on a vector of a specified length:

```
zipExact (a: Type) (n: Int) (Vec a n) (Vec a n): Vec (a,a) n
```

- Means you combine two vectors of the *same* length (**n**) into a vector of **n** pairs (unlike typical **zip**, which discards extras)
- Note type parameter **a** has the same syntax as non-type parameter **n**—because all parameters are *just terms*

Dependently Typed Function Rules

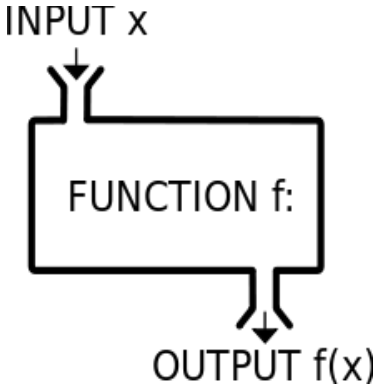
Dependent function type:

$$\forall (x:\tau).\,\tau'$$

or, equivalently:

$$(x:\tau)\to\tau'$$

where **x** may appear in the term **τ′**



Keep in mind that a dependent function type is itself a type:

$$\frac{\Gamma\vdash\tau:\star \qquad \Gamma, x:\tau\vdash\tau':\star}{\Gamma\vdash(\forall(x:\tau).\,\tau'):\star}$$

- So a function type may contain a nontrivial function of its argument:

$$(x:\tau)\to(f\ x)$$

Evaluation during Typechecking: Runtime at Compiletime

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \Downarrow \tau \quad \tau_2 \Downarrow \tau}{\Gamma \vdash e : \tau_2}$$

If, in the environment **Γ**, ***e***'s type evaluates completely to an equivalent type, then ***e*** is of that type, too

- This formulation of the type equivalence rule for dependent types means you must evaluate type expressions while typechecking
- Typechecking is usually thought of as a compiletime thing, and evaluation as a runtime thing
- Can't just bolt on dependent types to languages whose semantics depend on separation between compiletime and runtime

Part II: Refinement Types

A refinement type (or subset type) includes a proposition **p**:

$$\{ x : \tau \mid p \}$$

where **p** may refer to **x**, and **x** belongs to the type if **p** holds

Refinement types let you say anything allowed by the language for propositions. This can be quite expressive, if verbose:

$$\begin{aligned} \text{sqrt: } & \{ x\text{: Double} \mid x \geq 0.0 \} \rightarrow \\ & \{ r\text{: Double} \mid \text{if } x < 0.999 \text{ then } r \geq x \\ & \quad \text{else if } x > 1.001 \text{ then } r < x \\ & \quad \text{else } 0.999 < r < 1.001 \} \end{aligned}$$

Expressiveness of Refinement Types

- Refinement types let you refer to characteristics not explicitly declared as part of a base type
- length below is not a parameter to the List constructor (contrast with Vec in dependent type example earlier):

```
zipExact
    (a: Type) (aa: List a)
    { aaa: List a | length aa = length aaa }:
        { result: List (a,a) | length result = length aa }
```

- Put all your tests into your types?

OMG, Subtyping

- Dependent types don't need subtyping to be useful, but...
- Refinement types would be stupid without subtyping
- If **A <: B** and **x: A**, then **x: B**
- Refinement type subtyping is defined by implication of propositions:

$$\{ x : \tau \mid p \} <: \{ x : \tau \mid q \}$$

iff

$$\forall x \epsilon \tau,\ p \rightarrow q.$$

Refinement Subtyping

$$\{ x: \text{Int} \mid \text{positive } x \wedge \text{odd } x \}$$

is a subtype of

$\{ x: \text{Int} \mid \text{positive } x \}$ and $\{ x: \text{Int} \mid \text{odd } x \}$

which are subtypes of

$$\{ x: \text{Int} \mid \text{positive } x \vee \text{odd } x \}$$

- Given **f { n: Int | odd n }: Int**, should allow **f (2 * n + 1)** for any **Int n** (as a rule, infer the most specific type possible)
- Requires typechecker to prove all sorts of interesting properties about integers
- You can't build a system that can prove every proposition!

Saying Too Much

- Refinement types let you express a lot, and smart compilers can infer a lot

- Would you want to see a type like the one on the right in an error message?

- In their 1991 paper that introduced the term "refinement type", Freeman and Pfenning infer this type for addition on bitstrings:

```
?e → ?e → ?e ^
?e → stdpos → stdpos ^
?e → std → std ^
?e → bitstr → bitstr ^
stdpos→ ?e → stdpos ^
stdpos→ stdpos → stdpos ^
stdpos→ std → stdpos ^
stdpos→ bitstr → bitstr ^
std → ?e → std ^
std → stdpos → stdpos ^
std → std → std ^
std → bitstr → bitstr ^
bitstr→ ?e → bitstr ^
bitstr→ stdpos → bitstr ^
bitstr→ std → bitstr ^
bitstr→ bitstr → bitstr
```

Blackbox vs. Whitebox

- When typechecking a function **f**, a compiler for a simpler type system looks only at the type signature for functions invoked by **f**—the other functions are black boxes
- When a system like Coq tries to prove something like **forall x: Nat, f x = g x**, it uses the implementations of **f** and **g**—the functions are white boxes
- Functions invoked during typechecking are effectively white boxes to the typechecker

- What color box is this?

```
odd:
  (n: Nat) →
    { r: Bool |
      r = (n % 2 = 1) }

odd n := n % 2 = 1
```

Booleans vs. Propositions

- There is a conceptual difference between boolean expressions (that compute a value) and propositions (that can be proven or disproven)
- Boolean expressions are blackbox, propositions whitebox

- In most implementations of refinement types, propositions (which appear in refinements) are syntactically similar to boolean expressions (unlike, say, in Coq)
- In some formulations of propositions, ordinary boolean expressions may be supplemented by quantification and implication

Typechecking Perils

- Fancy type systems can fail in confusing ways
- Compilers have to know their limits and describe what happened when they've failed in terms programmers understand
- Evaluation during typechecking in dependent types might not terminate, or might throw an exception
- What to do if unable to prove an obviously true implication in refinement subtype checking?

Soundness vs. Completeness

- A *sound* type system disallows all illegal programs (programs that might have type errors at runtime)
- A *complete* type system allows all legal programs (programs that might not have type errors at runtime)
- For simpler type systems, we err on the side of soundness
- But for more complicated type systems, that disallows too many obviously valid programs

Aside: Building Refinement Types from Dependent Types

- If you add a notion of subtyping to dependent type rules, you can build refinement types directly:

$$\text{refine } (T\text{: Type}) \ (\text{pred: } T \to \text{Bool})\text{: Type} = (x\text{: } T) \to (\text{if pred } x \text{ then } T \text{ else } \bot)$$

(where $\bot$ is the uninhabited bottom type)

- Whether it is better to do this or make refinement types primitive depends on whether the compiler is smart enough to make sense of this definition, and give useful error messages for refinement type errors

First Interlude: or, Despair Is a Viable Option, But Not the Only One

Mathematicians Are from Vulcan, Engineers Are from Qo'noS

- Type annotations tell the compiler what I want to prove
- Programming without proving is pointless
- Runtime type errors are illogical

- Type annotations tell other engineers what I mean
- Proofs are a lot of work. Sometimes they can wait
- The type system should let me fake it through the demo

Deciding about Undecidability

- Allow only types simple enough that the compiler can check them automatically
- Not so bad? SMT solvers, etc.

- Rely on the programmer to prove anything the compiler can't
- Can your average programmer prove theorems?

- Perform runtime checks of everything that can't be proven at compiletime
- Is a type system still sound if it can throw type errors?

Part III: Contracts and the Alternate Universe of Runtime Type Checking

- Eiffel (1992): compiletime OO typechecking supplemented by contracts
- Contracts are *part of the type* of functions
- Assert statements are also runtime checks, but *not* part of a function type

- Conceptually, contracts look just like refinement types:

$$\{ x : \tau \mid p \}$$

  but **p** is always a boolean expression to evaluate at runtime, not a proposition
- Function contracts are covariant in return type and contravariant in parameter type
- Eiffel actually got function contract type variance right!

Baseless Contracts and Runtime-Only Typechecking

- A system like Eiffel's uses compiletime checking for the base type **τ** in **{ x : τ | p }**
- If you have a runtime test for membership in every type, you can eliminate **τ**:

$$\{ x : \tau \mid p \} \Rightarrow x \mid (x \in \tau \wedge p)$$

(where our new syntax for types is just **x | p**)

Assuming for now that contracts are mandatory. Programs look just like statically typed ones!

- Alternatively, leave the base type in the contract type syntax, but treat $\{ x : \tau \mid p \}$ as $\{ x : \top \mid x \in \tau \wedge p \}$
- Looks like static refinement types—but no typechecking before runtime

From Contracts to Coercions

- You can imagine that a baseless contract type **C** looks like a boolean function:

$$\mathsf{isC}: \top \to \mathsf{Bool}$$

- For execution, convert to a function **toC** that coerces any argument to have type **C**:

```
toC (a: ⊤) =
  if isC a then a
  else error
```

- Now, given:

$$f: A \to B$$

we can compile **f** to the untyped lambda calculus as:

$$\mathsf{toB} \circ f \circ \mathsf{toA}$$

Signs of Trouble

∀

- Previous slides blithely assume you can construct a coercion function for any type, including a function type
- But a function type **f: A → B** corresponds to this contract predicate:

$$\forall\ a : \top, \forall\ b: \top, \mathsf{isA}\ a \rightarrow \mathsf{isB}\ (f\ a)$$

- **∀**-quantified predicates cannot normally be implemented by a simple runtime check

Coercing Wraps Function Types

- For a function type **C → D**, we can't always come up with a runtime test **isCToD**
- But we *can* create a coercion **toCToD** that coerces to type **C → D**

- **toCToD** *wraps* its argument:

$$\text{toCToD f} = \text{toD} \circ \text{f} \circ \text{toC}$$

- Recurse if **C** or **D** is also a function type
- Error reporting is *deferred* to the point where wrapped **f** is fed an argument that isn't an **C**, or returns something that isn't a **D**

Layers upon Layers

- Hey, didn't we just say that **f**: **A** → **B** is compiled as **toB** ∘ **f** ∘ **toA**?
- And now you're telling me that coercing **f** to type **C** → **D** wraps it in **toD** ∘ **...** ∘ **toC**?

  - So the result of the coercion is **toD** ∘ **toB** ∘ **f** ∘ **toA** ∘ **toC**?
  - Yup. This is not a very smart way to compile, but it does check all type annotations
  - Consecutive coercions like **toA** ∘ **toC** can be thought of as a cast *from* **C** *to* **A**
  - Alternate notation: $\langle A \Leftarrow C \rangle$

Part IV: Gradual Typing

- People love dynamically typed languages
- Until they hate them
- How to move a large body of code without type declarations to something with declared types?
- Solution: add types selectively
- (Near) synonyms: soft typing, optional static typing

Top, Bottom, Dynamic Types

- Names: ⊤, Top, Any, Anything
- Supertype of every type
- Inhabited by every value
- All types are tagged variants!

- Names: ⊥, Bottom, Nothing
- Subtype of every type
- Uninhabited
- Some functions can't be called, or never return!

- Names: Dyn, Dynamic, Any
- Inhabited by every value
- Compatible with every type
- All types are tagged!

Confusing Compatibility

```
∀T,U : Type,
 f : T → U
 x : Dyn
 (f x) is legal.
```

Looks like
`Dyn <: T`!

```
∀T,U : Type,
 f : Dyn → U
 x : T
 (f x) is legal.
```

Looks like
`T <: Dyn`!

- For conversion purposes, the dynamic type acts like a subtype of every type (like the bottom type) and also like a supertype of every type (like the top type). But it cannot be a synonym of both top and bottom (at least in a language that has any values)!
- Dynamic is its own thing, outside the regular static type lattice

Runtime Cast at Dynamic-Static Boundary

- If a value **v** is declared to be of dynamic type, we have no information (locally) about its static type.
- To convert **v** from dynamic type to some static type **T**, perform a runtime check that **v** is indeed of type **T**.
- To convert from static to dynamic, no runtime check is necessary (like converting from any **T** to top type).
- The conversion from dynamic type to **T** is the same as the coercion function **toT** from the top type to **T**, as shown earlier

Problems with Naïve Coercion-Based Runtime Typechecking

- There are problems with the treatment of runtime checking of function types as described so far
- Error reporting for function coercions is delayed until we can perform a non-function coercion, and so appears to come from the wrong place
- Passing functions around can result in deep layers of wrapping
- Fortunately, there is a way to address each of these problems

Second Interlude; or, You Know You Want It

Real Programmers Use Fancy Types

- A recent paper (*Contracts in Practice*, Estler et al.) examined use in actual projects of contract types in 3 OO languages that allow contracts (Eiffel, C# with Code Contracts, and Java with JML)
- Programmers annotated about 1/3 of the eligible places with contracts
- People seem inclined to use a high level of expressiveness in the real world

Part V: The Blame Calculus

- Function coercions described earlier throw exceptions where function type errors are *detected*. But the type error may have *occurred* much earlier, so the error message is confusing

```
f (g: A → B) =
  … do a bunch of stuff
  … call g at the end
h: C → D = … stuff
f h
```

where **C** is not a supertype of **A**, or **D** is not a subtype of **B**

- Logically, error occurs at call to **f h** (function type **C → D** is not a subtype of **A → B**)
- But report comes from call to **g** in **f** (**f** applies **g** to **A**-not-**C**, or gets back **D**-not-**B**, so a cast fails)

Blame Labels

- The blame calculus keeps track of both source and target types in a coercion, along with a blame label that points to the original argument of the coercion

$$\langle (T \Leftarrow S)^{p} \rangle\, t$$

- This is a term **t** cast from type **S** to type **T**; **p** is the label that refers to **t**
- Blame labels have a positive or negative polarity. Positive labels blame their original designees; negative labels (written $-\mathbf{p}$) blame the environment where the cast is executed
- Double negatives cancel:

$$--\mathrm{p} = \mathrm{p}$$

Function Type Casts

- A function application **(f x)**, where **f** is cast from type **C → D** to type **A → B**, reduces like this:

$$\left(\langle A \to B \Leftarrow C \to D \rangle^{p} f\right) x \quad \Rightarrow$$
$$\langle B \Leftarrow D \rangle^{p} \left(f \left(\langle C \Leftarrow A \rangle^{-p} x\right)\right)$$

- If **x** is also a function application and **C** and **A** are function types, recurse (this could negate the negated label **p**, so that the resulting label blames **f**)

Part VI: Space-Efficient Coercions

- Because function coercions wrap function, passing dynamically typechecked functions around a lot yields large coercions
- For example, in a gradually typed language, passing a function between two mutually recursive functions that call each other in tail position, where one function parameter is statically typed and one dynamically, winds up with wrappers nested to the depth of the recursion
- Keeping this type of overwrapping under control requires composing and normalizing adjacent coercions

Threesomes

- Any sequence of casts can be reduced to a single cast involving three types
- You might think you could combine consecutive casts this way:

$$\langle T \Leftarrow U \rangle \langle U \Leftarrow S \rangle \;\Rightarrow\; \langle T \Leftarrow S \rangle$$

but that doesn't work for something like:

$$\langle Dyn \Leftarrow Bool \rangle \langle Bool \Leftarrow Dyn \rangle 42$$

which must fail because **42** isn't a **Bool**

- Instead, you must keep track of the greatest lower bound of intermediate types, and check against that as well:

$$\langle T \Leftarrow U \rangle \langle V \Leftarrow S \rangle \;\Rightarrow\; \langle T \Leftarrow_{U \cap V} S \rangle$$

Strange Intersection

- The type intersection operator ∩ from the previous slide is not quite the normal type intersection operator

- ∩ means intersection with respect to *naïve function subtyping*
- Normal function subtyping is covariant in return type and contravariant in parameter type, but naïve subtyping is covariant in both
- So threesomes have this function type intersection rule:

$$(T \to U) \cap (V \to W) = (V \cap T) \to (U \cap W)$$

Part the Last: Hybrid Typechecking

- “Static typechecking where possible, dynamic where necessary”
- That is, attempt to perform static typechecking over entire program
- If typechecker cannot prove a given type claim either correct or incorrect, generate a dynamic check for the claim

- What is not to love?

Hybrid Meets Gradual

- Hybrid typechecking seems ideal for gradually typed languages
- Casts from type dynamic can be treated as unprovable type claims

- Or get more ambitious:
  - Treat dynamically typed code or code without type annotations as statically typed, where static types are to be *inferred*
  - And perform usual static analysis
- But useful static types cannot always be inferred for complex type systems, so fall back to coercions

Sage

- Ambitious hybrid-typechecked language
- Experimental; developed at UC Santa Cruz until around 2007-2008

- General dependent and refinement types (unrestricted function type terms and unrestricted predicates)
- Static and dynamic typing
- Type inference
- Uses external theorem prover (Simplify)

Sage Learns from Its Failures

- One of the coolest ideas in Sage: when a runtime cast from one type to another fails, Sage adds the failing value to a counterexample database
- Uses the database in static typechecking during future compilations
- Also remembers past compilations that generated the same cast, and warns about them
- What if that database could be distributed?

Surprisingly Affordable

- Experience with Sage suggests that most of the type annotations programmers make can be checked at compiletime
- In the Sage benchmarks, only about 0.5% of type annotations resulted in runtime checks

What If You Could Have It All?

- What's missing from Sage?
- (Aside from actual support)
- Access to the theorem prover, with a proof assistant

So that if the typechecker can't prove something, you can provide a handwritten proof (if you're sufficiently hardcore)

- Also, a way to insist on compiletime rather than runtime checking for a given type claim

Customers Who Viewed This Item Also Viewed

- I'm not aware of a language with full-blown hybrid typechecking besides Sage
- But other languages have some of the features mentioned here

- Dependent types (with proof assistant): Coq, Agda, Idris
- Refinement types (statically verified): Liquid Haskell
- Contracts (enforced at runtime): Eiffel, C# with Code Contracts, Java with JML
- Gradual typing: TypeScript, Dart
- This list is incomplete and probably wrong