From Typesafe to Curry-Howard

Three Constraints That Give
Dependently Typed Languages
the Power to Prove
by Preventing Paradoxes

Adrian King
11 February 2019

Motivation:
Type Systems and What We Get out of Them

Classifying Type Systems by the Guarantees They Provide

- C-class: type annotations are just suggestions. You can cast anything to anything. Memory errors are possible.
- Java-class: (non-array) type annotations are enforced, but arbitrary typecasts are allowed. Runtime type errors are possible, but not memory errors.
- ML-class: runtime type errors are impossible (Milner's “well-typed programs cannot go wrong”), but termination is not guaranteed.
- Agda-class: termination is guaranteed (we might say “well-typed programs always go right”). The Curry-Howard correspondence is valid.

Curry and Howard and
the Fine Print

- The Curry-Howard correspondence says that, in a programming language, types can be interpreted as propositions and a term with a given type as a proof of the corresponding proposition.

- When I say the Curry-Howard correspondence is *valid* in a given language, I mean all terms are logically valid proofs of their types—it is impossible to prove a contradiction with such proofs.
- It is also possible to say that Curry-Howard holds of languages where a proof term is augmented by a separate, manual proof that the proof term is valid. But I won't say that.

How Nontermination Breaks Curry-Howard

- Consider this Idris function:

```
partial
ohNo: a -> Void
ohNo x = ohNo x
```

- Void is a type with no inhabitants. It is logically equivalent to falsehood.

- If the typechecker accepts a top-level expression of type Void:

```
partial
argh: Void = ohNo 0
```

then it is letting us prove something false, which means the underlying logic is inconsistent.

Dependent Types in an ML-Class Language

- Cayenne-class: full support for dependent types. Type safety in the ML sense, but because evaluation does not always terminate and evaluation may occur during typechecking, typechecking is not decidable.

Some Cayenne

Cayenne example from Wikipedia:

```
PrintfType :: String -> #
PrintfType (Nil)          = String
PrintfType ('%':('d':cs)) = Int    -> PrintfType cs
PrintfType ('%':('s':cs)) = String -> PrintfType cs
PrintfType ('%':( _ :cs)) =           PrintfType cs
PrintfType ( _ :cs)       =           PrintfType cs

aux :: (fmt::String) -> String -> PrintfType fmt
aux  (Nil)          out = out
aux  ('%':('d':cs)) out = \ (i::Int)    -> aux  cs (out ++ show i)
aux  ('%':('s':cs)) out = \ (s::String) -> aux  cs (out ++ s)
aux  ('%':( c :cs)) out =                  aux  cs (out ++ c : Nil)
aux  (c:cs)         out =                  aux  cs (out ++ c : Nil)

printf :: (fmt::String) -> PrintfType fmt
printf fmt = aux fmt Nil
```

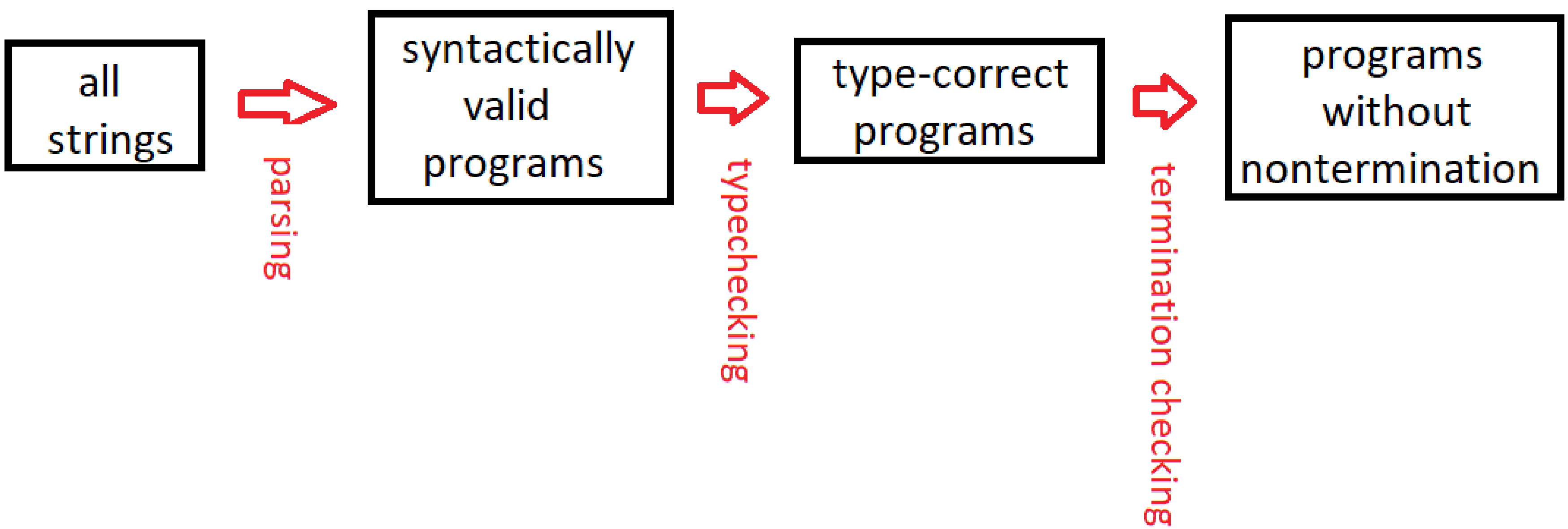
Cayenne Is an Outlier

- Cayenne occupies an unusual place in the programming language design space.
- It offers type safety, but Cayenne terms are not valid proofs under the Curry-Howard correspondence.
- I think most programming language designers, having gone to all the trouble to implement dependent types, figure they might as well go ahead and offer valid proofs as well as type safety.
- ML-class languages with termination checks (so that Curry-Howard is valid) are also uncommon.
- Cayenne is no longer supported.

Three Constraints

Compilation As a Series of Filters

- Consider compilation as a sequence of filters:



- Each filter can only *reduce* the set of programs accepted.
- Cayenne stops at the third box, Agda at the last.
- Termination checking in dependently typed languages has three components.

Code, Data, Type

- It is convenient to think of the features of dependently typed languages as being grouped into three categories: code, data, type.
  - Code is what can appear in a function.
  - Data is what can be declared in an inductive datatype.
  - Type is what can appear in a type annotation.
- Each of the three termination constraints corresponds (roughly) to one of the above categories.

Constraint #1 (Code): Structural Recursion

Recursing on Pieces

- The structural recursion constraint requires that when a function calls itself, it can only do so by passing to the recursive call a proper substructure of one its arguments.
- You can obtain such a substructure by pattern matching (in Coq, Idris, or Agda).

```
-- good: decreasing on the argument
total
len: List a -> Nat
len [] = 0
len (x :: xs) = S (len xs)

-- bad: increasing on the argument
partial
notLen: List Nat -> Nat
notLen xs = notLen (1 :: xs)

-- also bad: not decreasing on the
-- argument
partial
ohNo: a -> Void
ohNo x = ohNo x
```

Nested Primitive Recursion

- Agda and Idris (at least) can recognize a slightly more complex case than the one from the previous slide, one where multiple arguments decrease in lexicographic order.
- In ack below, either the first argument decreases, or it stays the same and the second argument decreases.

```
total
ack: Nat -> Nat -> Nat
ack Z n = S n
ack (S m) Z = ack m 1
ack (S m) (S n) = ack m (ack (S m) n)
```

Structural Recursion Is Often Too Conservative

- Many algorithms terminate for reasons that are obvious to a human reader, but invisible to the structural recursion checkers in available languages.

```
partial
mrgSort: (Ord a) => List a -> List a
mrgSort [] = []
mrgSort [x] = [x]
mrgSort xs =
  mrg (mrgSort firstHalf) (mrgSort secondHalf)
  where
    halfLen: Nat
    halfLen = (length xs) `div` 2
    firstHalf = take halfLen xs
    secondHalf = drop halfLen xs
```

Constraint #2 (Data):
Strict Positivity

Unbounded Recursion That Looks *Structurally* OK

- Consider this attempt to encode the untyped lambda calculus, adapted from Adam Chlipala's *Certified Programming with Dependent Types*:

```
data Term: Type where
  App: Term -> Term -> Term
  Abs: (Term -> Term) -> Term

partial
uhOh: Term -> Term
uhOh (Abs f) = f (Abs f)
uhOh t = t
```

- Note that uhOh *doesn't* call itself.
- But what happens when you do uhOh (Abs uhOh)?

Limits on a Datatype's Occurrence in Its Own Definition

- The strict positivity constraint says that a datatype cannot appear on the left side of an arrow in the type of any of its constructors' arguments.

```
data Term: Type where
  App: Term -> Term -> Term
  Abs: (Term -> Term) -> Term
```

- As the previous slide showed, violating the constraint lets you sneak in an unbounded recursion.

Constraint #3 (Type): Type Stratification

Russell and Girard (a Pair o' Docs)

- Bertrand Russell's famous paradox about the set of all sets that don't contain themselves has an analog that can be expressed in a programming language, which was discovered by Jean-Yves Girard.
- Girard's paradox appears in any type system where Type (the type of types) is an element of itself.

A Paradox in Idris

- Because of Idris bug #3194 (type stratification not enforced across module boundaries), this actually compiles:

```
%default total

data Tree: Type where
  Sup: (a: Type) -> (f: a -> Tree) -> Tree

A: Tree -> Type
A (Sup a _) = a

F: (t: Tree) -> A t -> Tree
F (Sup _ f) = f

Normal: Tree -> Type
Normal t =
  (y: A t ** (F t y = Sup (A t) (F t))) -> Void

NT: Type
NT = (t: Tree ** Normal t)
```

```
P: NT -> Tree
P (x ** _) = x

R: Tree
R = Sup NT P

Lemma: Normal R
Lemma ((y1 ** y2) ** z) =
  y2 (
    replace
      {P =
        (\y3 =>
          (y: A y3 ** F y3 y =
            Sup (A y3) (F y3)))}
      (sym z) ((y1 ** y2) ** z))

Russel: Void
Russel = Lemma ((R ** Lemma) ** Refl)
```

Layered Universes

- The usual solution to Russell's/Girard's paradox is divide the type of types into separate, stratified types (**universe levels**), where each level cannot contain itself or higher levels, but does contain the next lower level.

  Type i : Type (i + 1)

- Some programming languages (Coq, Idris) let you write Type for any level of the type hierarchy, but they don't really allow Type to be a member of itself. Instead, they figure out the appropriate level constraints automatically.

Type Stratification Rules (Approximately)

- The type level of a function type $A \rightarrow B$ is at least the maximum of the levels of A and B.

- The type level of an inductive datatype is:
  - at least the level of it largest parameter;
  - greater than the largest level of its indices;
  - and greater than the largest level of its constructors' parameters.

- But note that the exact details for the inductive datatype rules vary from one language to another.

Agda Makes Stratification Explicit

- Agda forces you to keep track of type levels explicitly:

```
data Tree : Set lone where
  Sup : (a : Set) → (f : a → Tree) → Tree

A : Tree → Set
A (Sup a _) = a

F : (t : Tree) → A t → Tree
F (Sup _ f) = f

Normal : Tree → Set lone
Normal t =
  ¬ (Σ (A t) (λ y → F t y ≡ Sup (A t) (F t)))
```

```
NT : Set lone
NT = Σ Tree (λ t → Normal t)

P : NT → Tree
P (x , _) = x

R : Tree
R = Sup NT P

{- Agda objects to the use of NT in the last
line above:

Set₁ != Set
when checking that the expression NT has
type Set -}
```

- The Agda approach is more work. Does it let you write code Coq or Idris wouldn't? I don't know.

Type Stratification Is Also about Termination

- The type stratification constraint is subtler than the other two, and violations are less localized.
- But its purpose, like that of the other constraints, is to prevent nontermination.

So why is termination such a big deal?

Theory and Terminology

Soundness and Completeness

- We want compilers to accept only programs that obey the proper rules, that is, we want them to be **sound**.
- A compiler that accepted *every* program that obeyed the rules would be **complete**.
- Real-world compilers are usually sound (we hope), but not usually complete.
- In languages complicated enough to be useful, typechecking would be undecidable if we tried to go for both soundness and completeness.

Soundness is Conservative

- Consider:

```
good: Nat
good =
  if 0 == 1 then 0 else 42

bad: Nat
bad =
  if 0 == 1 then "oops"
  else 42
```

- If you ran the code without checking it first, both good and bad would return Nats. A complete typechecker would accept both. But the real Idris typechecker applies the more conservative rule that both branches of an if must have the same type.

Typing Rules and Judgments

- Basic typing rules for a Cayenne-class language:

$$\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S}$$

$$\frac{\Gamma ; x{:}S \vdash e : T \quad \Gamma \vdash (x{:}S) \rightarrow T : \text{Type}}{\Gamma \vdash \lambda x{:}S.e : (x{:}S) \rightarrow T}$$

$$\frac{\Gamma \vdash f : (x{:}S) \rightarrow T \quad \Gamma \vdash s : S}{\Gamma \vdash f\, s : T[s/x]}$$

$$\frac{\Gamma ; x{:}S \vdash T : \text{Type} \quad \Gamma \vdash S : \text{Type}}{\Gamma \vdash (x{:}S) \rightarrow T : \text{Type}}$$

$$\frac{}{\Gamma \vdash \text{Type} : \text{Type}}$$

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash f : T \quad e \Downarrow v \quad f \Downarrow v}{\Gamma \vdash e \equiv f : T}$$

$$\frac{\Gamma \vdash x : S \quad \Gamma \vdash S \equiv T : \text{Type}}{\Gamma \vdash x : T}$$

Evaluation Rules

- **Evaluation** (aka **reduction**) is expressed in the **small-step** style. For example, a rule for function application:

$$\frac{\text{value } v}{(\lambda x{:}S.e)\ v \Rightarrow e[v/x]}$$

- *v* above represents a **value**, something that cannot be evaluated further.

- **Normalization** (**big-step** evaluation) consists of small steps repeated until you get a value.

$$\frac{\text{value } v}{v \Downarrow v}$$

$$\frac{v \Rightarrow v' \quad v' \Downarrow v''}{v \Downarrow v''}$$

Progress and Preservation

- **Progress** is a programming language property that says every **well-typed** term either is a value or can take a small evaluation step.
- It is easy to ensure progress if your language is well-defined and you're writing an interpreter for it in a language that checks that pattern matching is complete.

- **(Type) preservation** says that after each small evaluation step, a well-typed term has the same type as before the step.
- Preservation is usually more interesting and harder to prove than progress.
- Both properties are necessary if your language is to be well-behaved.

Logical Falsehood

- In a dependently typed language, logical falsehood can be represented as a proposition in one of two ways:

- A datatype with no data constructors (like Void in Idris):

```
data Void where
```

- A function type that says every type is inhabited:

```
(a: Type) → a
```

- If a programming language admits a well-typed term of either type above, that shows that the logic underlying the language is inconsistent, so the Curry-Howard correspondence does not hold.

With Preservation, All Proofs of Falsehood Involve Nontermination

- Suppose a language has a *value* representing falsehood. Then it is either:
  - headed by a constructor of a datatype with no constructors, which is a contradiction; or
  - a function that can be applied to any type to produce a value of that type. Take the type to be a datatype with no constructors, and arrive at the previous case.
- Given nontermination and preservation, every well-typed term normalizes to a value of its type. So there can be no *term* representing falsehood; it would reduce to a value that would give a contradiction.

… which means that fixing nontermination in a Cayennne-class language gives you an Agda-class language.

What about
Other Type Systems?

Alternatives to the Three Constraints

- ML, Haskell: types are not terms (so you don't need universe levels).
- Martin-Löf: raw induction principles (so structural recursion is automatic).
- Coq: `Prop` universe (where, with some restrictions, you don't worry about type levels).

Believe Me?

Fake News?

- I've made some assumptions. They might be wrong.

- When I talk about languages, I mostly assume away nonterminating primitives (like exceptions) or unsafe features.

- Does Cayenne have type preservation? I expect so, but I haven't found a proof.
- Do I understand all the details of type stratification? No.
- Are the three constraints sufficient to eliminate all unbound recursion? I think people think so.