

Type Checking & Unification

COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University
Semester 2, 2011

Primitive Values

- The value $(\lambda a b. b)$ has an ambiguous interpretation. It might be a Bool, Integer, or general function.
- We can extend the λ -calculus with primitive values and types, so that the interpretation of Booleans and Integers is no longer ambiguous.

`true, false :: Bool`

`0, 1, 2, 3... :: Int`

- Now `false` cannot be mistaken for `0` or *vice-versa*.

Types

- With Church encoding, the representations of `c0` and `false` are α -equivalent (ie, differ only in the names of variables).

$$c_0 \stackrel{\text{def}}{=} \lambda s z. z$$
$$\text{false} \stackrel{\text{def}}{=} \lambda x y. y$$

- If a term reduces to $(\lambda a b. b)$ how will we know whether it should be taken a Boolean, an Integer, or some other type of object?
- Notice how hard it is to define the word “type” without using that word, or a synonym in the definition – this hints at how fundamental the idea is.

More Primitive Values

Unfortunately, as our primitive values are no longer represented by lambda-terms, we cannot use the Church versions of the functions ‘succ’, ‘plus’, ‘and’, ‘or’, ‘not’ ... etc

We must also add these functions to our new system as primitives.

`succ :: Int → Int`

`plus :: Int → Int → Int`

`isZero :: Int → Bool`

`not :: Bool → Bool`

These type signatures become axioms in our type system.

The Typing Relation

$$\Gamma \vdash e :: \tau$$

“With type environment Γ , the expression e has type τ ”

The *type environment* is a set which contains the types of all variables which are free in the expression. (Γ = “gamma”, τ = “tau”)

The type of the expression depends on the types we assign to these free variables, for example:

$$x : \text{Bool} \vdash x :: \text{Bool}$$

vs

$$x : \text{Int} \vdash x :: \text{Int}$$

Application

When evaluating a function application, the function and argument must have appropriate types.

If they don't, then the evaluation gets *stuck* before we reach normal form.

$$\text{isZero} :: \text{Int} \rightarrow \text{Bool}$$

$\text{isZero } 0$	$\rightarrow \text{true}$	OK
isZero true	$\rightarrow ???$	STUCK
isZero (succ 5)	$\rightarrow \text{isZero } 6 \rightarrow \text{false}$	OK
0 isZero	$\rightarrow ???$	STUCK
isZero (isZero)	$\rightarrow ???$	STUCK

The Application Rule

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}}$$

- The application rule expresses the typing constraints we wish to place on the two expressions e_1 and e_2 .
- As we're applying e_1 to e_2 , the expression e_1 must evaluate to a function.
- We've named the argument type τ_{11} . The function must accept an argument of this type.
- Applying the function will generate a new value as the result. We've named the type of this result τ_{12} .

Type Checking

We check the type of an expression by *proving* that it has some type.

The proofs are usually presented as Gentzen style *proof trees*, whose structure follows the syntax of the expression being checked.

$$\frac{\frac{\frac{}{0 \vdash \text{succ} :: \text{Int} \rightarrow \text{Int}}{0 \vdash \text{succ } 5 :: \text{Int}} \text{(App)}}{0 \vdash \text{isZero} :: \text{Int} \rightarrow \text{Bool}} \quad \frac{}{0 \vdash 5 :: \text{Int}} \text{(App)}}{0 \vdash \text{isZero (succ 5)} :: \text{Bool}} \text{(App)}$$

In this proof each statement is either an axiom, or has been constructed with the Application rule (App).

An Alternate Presentation

We could also lay out the proofs in a flat style, giving each statement and its justifications in turn.

1. $\emptyset \vdash 5 :: \text{Int}$ (Axiom)
2. $\emptyset \vdash \text{succ} :: \text{Int} \rightarrow \text{Int}$ (Axiom)
3. $\emptyset \vdash \text{succ } 5 :: \text{Int}$ (App 2 1)
4. $\emptyset \vdash \text{isZero} :: \text{Int} \rightarrow \text{Bool}$ (Axiom)
5. $\emptyset \vdash \text{isZero} (\text{succ } 5) :: \text{Bool}$ (App 4 3)

This style is easier to write down, but makes it harder to see the proof's natural branching structure.

Type Annotations

What is the type of the following expression?

$$\lambda x. 5$$

Without knowing the type of the argument (and without polymorphism), there is no way to tell. It could be $\text{Int} \rightarrow \text{Int}$, $\text{Bool} \rightarrow \text{Int}$ or some other type.

We will add a *type annotation* to the parameter to make its type explicit.

$$\lambda(x : \text{Bool}). 5$$

Type Errors

If there is a *type error* in the expression then we will not be able to construct a valid proof tree.

The following proof is not valid because function and argument types of (succ true) do not match. The application rule does not apply.

$$\emptyset \vdash \text{isZero} :: \text{Int} \rightarrow \text{Bool} \quad \frac{\frac{\emptyset \vdash \text{succ} :: \text{Int} \rightarrow \text{Int} \quad \emptyset \vdash \text{true} :: \text{Bool}}{\emptyset \vdash \text{succ true} :: \text{Int}} \text{ (App)}}{\emptyset \vdash \text{isZero} (\text{succ true}) :: \text{Bool}} \text{ (App)}$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}}$$

The Abstraction Rule

$$\frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda(x : \tau_1). e_2 :: \tau_1 \rightarrow \tau_2}$$

- A λ -abstraction defines a function, so it has a function type.
- The annotation on the parameter gives the type of argument this function can be applied to.
- The parameter variable x is likely to be free in the function body e_2 , so we must *extend* the type environment with its type.

Example

$$\begin{array}{c}
 \frac{x : \text{Int} \vdash \text{succ} :: \text{Int} \rightarrow \text{Int} \quad \frac{x : \text{Int} \in \{x : \text{Int}\}}{x : \text{Int} \vdash x :: \text{Int}} \text{(Var)}}{x : \text{Int} \vdash \text{succ } x :: \text{Int}} \text{(App)} \\
 \frac{x : \text{Int} \vdash \text{isZero} :: \text{Int} \rightarrow \text{Bool} \quad \frac{x : \text{Int} \vdash \text{succ } x :: \text{Int}}{x : \text{Int} \vdash \text{isZero} (\text{succ } x) :: \text{Bool}} \text{(Abs)}}{\emptyset \vdash \lambda(x : \text{Int}). \text{isZero} (\text{succ } x) :: \text{Int} \rightarrow \text{Bool}} \text{(Abs)}
 \end{array}$$

Type Inference

To apply the typing rules, we need type annotations on the parameters of all λ -abstractions (and **let**-bindings) in our program.

These annotations are tedious to write.

Luckily, we can *infer* them automatically from an un-annotated program, and type check it at the same time.

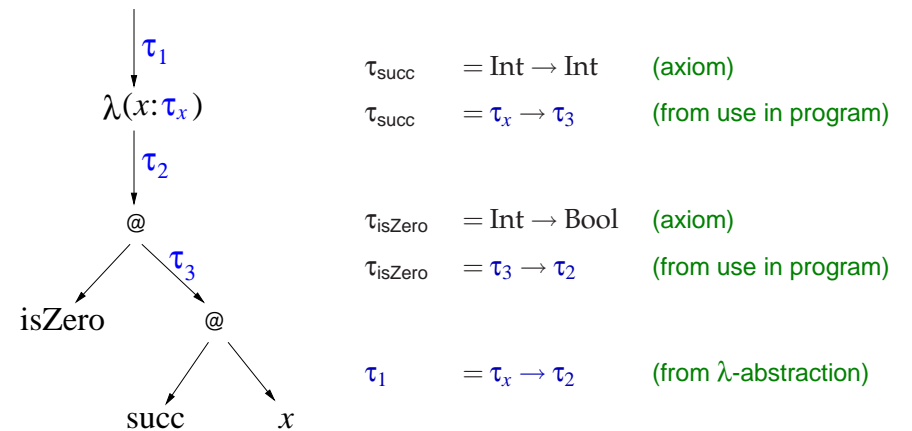
Step 1: Add fresh type variables to λ -abstractions in the program.

$$\begin{array}{l}
 \lambda x. \text{isZero} (\text{succ } x) \\
 \Rightarrow \lambda(x : \tau_x). \text{isZero} (\text{succ } x)
 \end{array}$$

Summary of rules

$$\begin{array}{c}
 \frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau} \text{(Var)} \\
 \\
 \frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda(x : \tau_1). e_2 :: \tau_1 \rightarrow \tau_2} \text{(Abs)} \\
 \\
 \frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}} \text{(App)}
 \end{array}$$

Step 2: Extract type constraints from this annotated program.



Constraint Generation

$$\begin{array}{c} \tau_x \\ | \\ x \end{array}$$
 (Var) If x is primitive, eg isZero
 $\tau_x = \dots$ known type for $x \dots$

$$\begin{array}{c} \tau_3 \\ | \\ @ \\ / \quad \backslash \\ \tau_1 \quad \tau_2 \end{array}$$
 (App)
 $\tau_1 = \tau_2 \rightarrow \tau_3$

$$\begin{array}{c} \tau_1 \\ | \\ \lambda(x: \tau_x) \\ | \\ \tau_2 \end{array}$$
 (Abs)
 $\tau_1 = \tau_x \rightarrow \tau_2$

First-Order Unification

Given a language of *variables* and *constants* (ie, known quantities — they may be functions), to instantiate variables to make expressions equal.

Example: unify types

$$\alpha \rightarrow (\alpha, \gamma)$$

$$(\beta, \gamma) \rightarrow (\delta, \text{Bool})$$

Requires unifying (1): α with (β, γ) and (2): (α, γ) with (δ, Bool) .

Unify pair (1): set $\alpha = (\beta, \gamma)$

(NOTE — we could instead unify pair (2) first — try it!!)

(We unify pair (1) and pair (2) *together*:
ie, what happens to α in the first also applies to the second).

Thus pair (2) now is: unify $((\beta, \gamma), \gamma)$ with (δ, Bool)

Step 3: Solve constraints.

1. $\tau_{\text{succ}} = \text{Int} \rightarrow \text{Int}$
2. $\tau_{\text{succ}} = \tau_x \rightarrow \tau_3$
3. $\tau_{\text{isZero}} = \text{Int} \rightarrow \text{Bool}$
4. $\tau_{\text{isZero}} = \tau_3 \rightarrow \tau_2$
5. $\tau_1 = \tau_x \rightarrow \tau_2$
6. $\tau_x = \text{Int}$ (Unify 1 2)
7. $\tau_3 = \text{Int}$ (Unify 1 2)
8. $\tau_2 = \text{Bool}$ (Unify 3 4)
9. $\tau_1 = \text{Int} \rightarrow \text{Bool}$ (Substitute 5 6 8)

Step 4: Substitute into original program.

Unification — example, continued

Requires unifying (β, γ) with δ and γ with Bool .

Set $\delta = (\beta, \gamma)$, and set $\gamma = \text{Bool}$

Having set $\gamma = \text{Bool}$, this affects the substitutions for α and δ , thus we have:

$$\alpha = (\beta, \text{Bool}) \quad \delta = (\beta, \text{Bool}) \quad \gamma = \text{Bool}$$

and the original type expressions

$$\alpha \rightarrow (\alpha, \gamma)$$

$$(\beta, \gamma) \rightarrow (\delta, \text{Bool})$$

both become

$$(\beta, \text{Bool}) \rightarrow ((\beta, \text{Bool}), \text{Bool})$$

The Most General Unifier

The algorithm(s) give a *most general unifier* of τ_1 and τ_2 :

If

- our algorithm gives a substitution function θ and unified result τ (thus $\theta(\tau_i) = \tau$), and
- there is another substitution function ϕ which makes τ_1 and τ_2 equal, say $\phi(\tau_i) = \sigma$,

then

- σ can be got by further substitution from τ , that is
- there exists a substitution function ψ such that $\phi = \psi \circ \theta$, and $\sigma = \psi(\tau)$

The Unification Algorithm

To unify $f(e_1, e_2)$ with $f'(e'_1, e'_2)$:

- if $f \neq f'$, unification FAILS
- otherwise unify e_1 with e'_1 , and e_2 with e'_2 *together* (ie substitutions of variables in one also apply in the other)
- this rule applies for any number of arguments *including 0*

To unify e (any expression) with x (variable) (but if $x = e$, nothing to do):

- set $x = e$
- apply this substitution to expressions not yet unified, and to rhs of substitutions already obtained
- this is NOT allowed if x is contained in e (fails “OCCURS” check); in this case, unification FAILS

The Most General Unifier — diagrammatically

Where the mgu is $\tau_1 \xrightarrow{\theta} \tau$ and we also have $\tau_1 \xrightarrow{\phi} \sigma$
 $\tau_2 \xrightarrow{\theta} \tau$ $\tau_2 \xrightarrow{\phi} \sigma$

then we can express ϕ as $\tau_1 \xrightarrow{\theta} \tau \xrightarrow{\psi} \sigma$
 $\tau_2 \xrightarrow{\theta} \tau \xrightarrow{\psi} \sigma$

This is why

- type inference in Haskell, and similar languages, works so well
- the algorithm allows non-determinism: to unify several pairs of expressions, can choose which pair you attack first, and get same result
- if the algorithm reaches “FAIL” no need to backtrack; can’t unify

First-Order Unification only

- The *function symbols* in the language must be *constants*, not variables
- if f and x are variables, then plenty of substitutions to get $f x = c$, none more general than the rest
- with type expressions like $\alpha \rightarrow (\alpha, \gamma)$
 $(\beta, \gamma) \rightarrow (\delta, \text{Bool})$
 - the *variables* are pure types $\alpha, \beta, \gamma, \delta$
 - the “function symbols” $_ \rightarrow _$ (binary), $(_, _)$ (binary), and Bool (nullary) are constants (ie, fixed quantities)
- so this *type* language is first-order, although the *term* language of Haskell is higher-order
- Prolog has a first-order *term* language, and an execution model based on unification

Soundness = Progress + Preservation

“Well typed programs don’t *go wrong*” – Milner

or

“If an expression is well typed, then its evaluation does not get stuck”

Progress:

If e is a well typed expression
 then (e is in normal form or e can be evaluated further).

Preservation:

If (e has type τ and e evaluates to a new expression e')
 then e' has type τ

Completeness

“If the evaluation of an expression does not get stuck, then it is well typed.”

The evaluation of the diverging expression $(\lambda x. x x)$ $(\lambda z. z z)$ does not get stuck. Can we work out its type?

$$\frac{\frac{\text{WRONG}}{x : \tau_x \vdash x :: \tau_x \rightarrow \tau_2} \quad \frac{x : \tau_x \in \{x : \tau_x\}}{x : \tau_x \vdash x :: \tau_x} \text{(Var)}}{x : \tau_x \vdash x x :: \tau_2} \text{(App)}}{\frac{x : \tau_x \vdash x x :: \tau_2}{\emptyset \vdash (\lambda(x : \tau_x). x x) :: \tau_x \rightarrow \tau_2} \text{(Abs)} \quad \frac{\vdots}{\emptyset \vdash (\lambda(z : \tau_z). z z) :: \tau_x} \text{(App)}}{\emptyset \vdash (\lambda(x : \tau_x). x x) (\lambda(z : \tau_z). z z) :: \tau_2} \text{(App)}$$

No!

To assign a type to $(x x)$ with this system we would need to solve the type constraint: $\tau_x = \tau_x \rightarrow \tau_2$

$$\begin{aligned} \tau_x &= \tau_x \rightarrow \tau_2 \\ &\equiv \tau_x = (\tau_x \rightarrow \tau_2) \rightarrow \tau_2 \\ &\equiv \tau_x = ((\tau_x \rightarrow \tau_2) \rightarrow \tau_2) \rightarrow \tau_2 \\ &\equiv \tau_x = (((\tau_x \rightarrow \tau_2) \rightarrow \tau_2) \rightarrow \tau_2) \rightarrow \tau_2 \\ &\equiv \tau_x = (((\dots \rightarrow \tau_2) \rightarrow \tau_2) \rightarrow \tau_2) \rightarrow \tau_2 \end{aligned}$$

Our system has no way of representing infinite/graphical types like this one.

We cannot assign a type to all expressions which do not get stuck during evaluation.

This typing system is incomplete.

The Big Picture

- Static typing is fun and profitable!
- It’s better to know if your program will crash sooner rather than later!
- There is an ongoing (endless?) quest to find more complete and expressive type systems.
- There is a fine balance between completeness and the ability to do full type inference.
- More complete type systems often sacrifice the ability to perform full type inference, requiring (some) type annotations to be added to the program.
- The type system for Haskell98 is sound, and supports full inference.