Theorems for Free!

by Philip Wadler

Kevin Kappelmann

May 27, 2021

Technical University of Munich

Type Systems and Polymorphism

Sigh, Just Apply It Twice!

```haskell
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

What is the result of . . .

Sigh, Just Apply It Twice!

```
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

What is the result of . . .

```
appTwice (*2) 1 = ?
```

Sigh, Just Apply It Twice!

```
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

What is the result of . . .

```
appTwice (*2) 1 = 4
```

Sigh, Just Apply It Twice!

```
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

What is the result of . . .

```
appTwice (*2) 1 = 2
appTwice (*2) "bogus" = ?
```

Sigh, Just Apply It Twice!

```haskell
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

What is the result of . . .

```haskell
appTwice (*2) 1 = 2
appTwice (*2) "bogus" = ...
```

```
• Couldn't match expected type ‘Int’ with
  actual type ‘[Char]’
• In the second argument of ‘appTwice’,
  namely ‘"bogus"’
  In the expression: appTwice (*2) "bogus"...
```

Sigh, Just Apply It Twice!

```haskell
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

What is the result of . . .

```haskell
appTwice (*2) 1 = 2
appTwice (++"l") "haske" = ?
```

Sigh, Just Apply It Twice!

```haskell
appTwice :: (Int -> Int) -> Int -> Int
appTwice f x = f (f x)
```

What is the result of ...

```haskell
appTwice (*2) 1 = 2
appTwice (++"l") "haske" = ...
```

```
• Couldn't match type ‘[Char]’ with ‘Int’
  Expected type: Int -> Int
  Actual type: [Char] -> [Char]
• In the first argument of ‘appTwice’,
  namely ‘(++"l")’
  In the expression: appTwice (++"l") "haske"...
```

Polymorphism to the Rescue!

Haskell knows *parametric polymorphism*: we can abstract over types by using type variables.

Polymorphism to the Rescue!

Haskell knows *parametric polymorphism*: we can abstract over types by using type variables.

```haskell
appTwice :: (a -> a) -> a -> a
appTwice f x = f (f x)
```

Polymorphism to the Rescue!

Haskell knows *parametric polymorphism*: we can abstract over types by using type variables.

```
appTwice :: (a -> a) -> a -> a
appTwice f x = f (f x)
```

And we are happy:

```
appTwice (*2) 1 = 4
appTwice (++"l") "haske" = "haskell"
```

Polymorphism to the Rescue!

Haskell knows *parametric polymorphism*: we can abstract over types by using type variables.

```haskell
appTwice :: (a -> a) -> a -> a
appTwice f x = f (f x)
```

And we are happy:

```haskell
appTwice (*2) 1 = 4
appTwice (++"l") "haske" = "haskell"
```

End of the story?

Polymorphism to the Rescue!

Haskell knows *parametric polymorphism*: we can abstract over types by using type variables.

```
appTwice :: (a -> a) -> a -> a
appTwice f x = f (f x)
```

And we are happy:

```
appTwice (*2) 1 = 4
appTwice (++"l") "haske" = "haskell"
```

End of the story? Of course not!

Polymorphism comes with another twist!

Black Magic

I show you a term's type but not its definition. You tell me the possible results: [1]

[1] Let us forget about `undefined` for a moment.

Black Magic

I show you a term's type but not its definition. You tell me the possible results: [1]

```haskell
g :: (a -> a) -> a -> a
-- def. of g hidden

g (*2) 1 =? 1
g (*2) 1 =? 2
g (*2) 1 =? 3
g (*2) 1 =? 16
g (*2) 1 =? 42
```

[1] Let us forget about `undefined` for a moment.

Black Magic

I show you a term's type but not its definition. You tell me the possible results: [1]

```
g :: (a -> a) -> a -> a
-- def. of g hidden
```

```
g (*2) 1 =? 1
g (*2) 1 =? 2
g (*2) 1 /= 3
g (*2) 1 =? 16
g (*2) 1 /= 42
```

[1] Let us forget about `undefined` for a moment.

Black Magic

I show you a term's type but not its definition. You tell me the possible results: [1]

```haskell
g :: (a -> a) -> a -> a
-- def. of g hidden
```

```
g (*2) 1 =? 1
g (*2) 1 =? 2
g (*2) 1 /= 3
g (*2) 1 =? 16
g (*2) 1 /= 42
```

What if I told you that `g (++"I") "" = "IIII"`?

[1] Let us forget about `undefined` for a moment.

Black Magic

I show you a term's type but not its definition. You tell me the possible results: [1]

```
g :: (a -> a) -> a -> a
-- def. of g hidden

g (*2) 1 /= 1
g (*2) 1 /= 2
g (*2) 1 /= 3
g (*2) 1 = 16
g (*2) 1 /= 42
```

What if I told you that `g (++"I") "" = "IIII"`?

[1] Let us forget about `undefined` for a moment.

Black Magic

I show you a term's type but not its definition. You tell me the possible results: [1]

```
g :: (a -> a) -> a -> a
g f x = f (f (f (f x)))


g (*2) 1 /= 1
g (*2) 1 /= 2
g (*2) 1 /= 3
g (*2) 1 = 16
g (*2) 1 /= 42
```

What if I told you that `g (++"I") "" = "IIII"`?

[1] Let us forget about `undefined` for a moment.

Polymorphic functions are defined once and for all for any type and as such must work uniformly on values of any type.

Polymorphic functions are defined once and for all for any type and as such must work uniformly on values of any type.

From the type of a polymorphic function, we can derive a theorem that it satisfies.

Polymorphic types provide us *theorems for free*.

Technical Development

System F

The polymorphic lambda calculus aka System F:

System F

The polymorphic lambda calculus aka System F:

Types:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau$$

System F

The polymorphic lambda calculus aka System F:

Types:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau$$

Terms:

$$t ::= x \mid \lambda x : \tau.\, t \mid t\, t \mid \Lambda \alpha.\, t \mid t\, [\tau]$$

System F

The polymorphic lambda calculus aka System F:

Types:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau$$

Terms:

$$t ::= x \mid \lambda x : \tau.\, t \mid t\, t \mid \Lambda \alpha.\, t \mid t\, [\tau]$$

Values:

$$v ::= x \mid \lambda x : \tau.\, t \mid \Lambda \alpha.\, t$$

System F

The polymorphic lambda calculus aka System F:

Types:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau$$

Terms:

$$t ::= x \mid \lambda x : \tau.\, t \mid t\, t \mid \Lambda \alpha.\, t \mid t\, [\tau]$$

Values:

$$v ::= x \mid \lambda x : \tau.\, t \mid \Lambda \alpha.\, t$$

Here is our `appTwice` function:

$$\Lambda \alpha.\, \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, f\, (f\, x) : \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

System F

The polymorphic lambda calculus aka System F:

Types:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau$$

Terms:

$$t ::= x \mid \lambda x : \tau.\, t \mid t\, t \mid \Lambda \alpha.\, t \mid t\, [\tau]$$

Values:

$$v ::= x \mid \lambda x : \tau.\, t \mid \Lambda \alpha.\, t$$

Here is our `appTwice` function:

$$\Lambda \alpha.\, \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, f\, (f\, x) : \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

We can call it like this: `appTwice [Int] (+1) 0`

Types as Relations

It is natural to interpret a type τ as a set $[\![\tau]\!]$ containing all values of type τ, e.g. $[\![\texttt{Bool}]\!] = \{\texttt{True},\texttt{False}\}$ in Haskell.

Types as Relations

~~It is natural to interpret a type τ as a set $[\![\tau]\!]$ containing all values of type τ, e.g. $[\![\texttt{Bool}]\!] = \{\texttt{True}, \texttt{False}\}$ in Haskell.~~

New slogan: types relate terms and related terms lead to related results.

Types as Relations: Examples

- Base types are interpreted as their identity relation, e.g. $[\![\texttt{Bool}]\!] = \{(\texttt{True},\texttt{True}), (\texttt{False},\texttt{False})\}$.

Types as Relations: Examples

- Base types are interpreted as their identity relation, e.g. $[\![\texttt{Bool}]\!] = \{(\texttt{True},\texttt{True}),(\texttt{False},\texttt{False})\}$.
- Two pairs are related if their components are related, i.e.

$$\big((t_1, t_2),(t'_1, t'_2)\big) \in [\![(\tau_1, \tau_2)]\!] \iff (t_1, t'_1) \in [\![\tau_1]\!] \land (t_2, t'_2) \in [\![\tau_2]\!].$$

Types as Relations: Examples

- Base types are interpreted as their identity relation, e.g. $[\![\texttt{Bool}]\!] = \{(\texttt{True},\texttt{True}), (\texttt{False},\texttt{False})\}$.
- Two pairs are related if their components are related, i.e.

$$\big((t_1, t_2), (t_1', t_2')\big) \in [\![(\tau_1, \tau_2)]\!] \iff (t_1, t_1') \in [\![\tau_1]\!] \wedge (t_2, t_2') \in [\![\tau_2]\!].$$

- Two lists are related if they have the same length and their elements are related, i.e.

$$\begin{gathered}\big([t_1, \ldots, t_n], [t_1', \ldots, t_{n'}']\big) \in [\![\mathsf{List}\ \tau]\!] \\ \iff n = n' \wedge \forall i \in \{1, \ldots, n\}.\ (t_i, t_i') \in [\![\tau]\!].\end{gathered}$$

Types as Relations: Examples

- Base types are interpreted as their identity relation, e.g. $[\![\texttt{Bool}]\!] = \{(\texttt{True}, \texttt{True}), (\texttt{False}, \texttt{False})\}$.
- Two pairs are related if their components are related, i.e.

$$\big((t_1, t_2), (t_1', t_2')\big) \in [\![(\tau_1, \tau_2)]\!] \iff (t_1, t_1') \in [\![\tau_1]\!] \land (t_2, t_2') \in [\![\tau_2]\!].$$

- Two lists are related if they have the same length and their elements are related, i.e.

$$\begin{aligned} &\big([t_1, \ldots, t_n], [t_1', \ldots, t_{n'}']\big) \in [\![\mathsf{List}\ \tau]\!] \\ \iff &n = n' \land \forall i \in \{1, \ldots, n\}.\ (t_i, t_i') \in [\![\tau]\!]. \end{aligned}$$

- Two functions are related if they map related arguments to related results, i.e.

$$(f, f') \in [\![\tau_1 \to \tau_2]\!] \iff \forall (t, t') \in [\![\tau_1]\!].\ \big(f\,t, f'\,t'\big) \in [\![\tau_2]\!].$$

Recipe for Logical Relations

A *logical relation* $R\llbracket\tau\rrbracket$ is an inductive family of relations indexed by types.

Recipe for Logical Relations

A *logical relation* $R[\![\tau]\!]$ is an inductive family of relations indexed by types.

If we want to prove a property P for the terms of our language, we construct $R[\![\tau]\!]$ in a way such that:

1. If $R[\![\tau]\!](t_1, \ldots, t_n)$ then

Recipe for Logical Relations

A *logical relation* $R[\![\tau]\!]$ is an inductive family of relations indexed by types.

If we want to prove a property P for the terms of our language, we construct $R[\![\tau]\!]$ in a way such that:

1. If $R[\![\tau]\!](t_1, \ldots, t_n)$ then
   1.1 $\vdash t_i : \tau$ for $1 \leq i \leq n$ (we write $\mathrm{wt}_\tau(t_1, \ldots, t_n)$)

Recipe for Logical Relations

A *logical relation* $R[\![\tau]\!]$ is an inductive family of relations indexed by types.

If we want to prove a property P for the terms of our language, we construct $R[\![\tau]\!]$ in a way such that:

1. If $R[\![\tau]\!](t_1, \ldots, t_n)$ then
    1.1 $\vdash t_i : \tau$ for $1 \leq i \leq n$ (we write $\mathrm{wt}_\tau(t_1, \ldots, t_n)$)
    1.2 $P(t_1, \ldots, t_n)$

Recipe for Logical Relations

A *logical relation* $R[\![\tau]\!]$ is an inductive family of relations indexed by types.

If we want to prove a property P for the terms of our language, we construct $R[\![\tau]\!]$ in a way such that:

1. If $R[\![\tau]\!](t_1, \ldots, t_n)$ then
   1.1 $\vdash t_i : \tau$ for $1 \leq i \leq n$ (we write $\mathrm{wt}_\tau(t_1, \ldots, t_n)$)
   1.2 $P(t_1, \ldots, t_n)$
2. The conditions of the relation are preserved by eliminating forms.

Our Logical Relation

We split our logical relation into two parts:

1. $\mathcal{V}[\![\tau]\!]$ on values
2. $\mathcal{E}[\![\tau]\!]$ on general terms

Our Logical Relation

We split our logical relation into two parts:

1. $\mathcal{V}[\![\tau]\!]$ on values
2. $\mathcal{E}[\![\tau]\!]$ on general terms

$$\mathcal{E}[\![\tau]\!] \coloneqq \left\{(t_1, t_2) \mid \mathsf{wt}_\tau(t_1, t_2) \wedge (t_1{\downarrow}, t_2{\downarrow}) \in \mathcal{V}[\![\tau]\!]\right\}$$

Relating Functions

If we had a `Bool` base type, we would first define:

$$\mathcal{V}[\![\texttt{Bool}]\!] := \{(\texttt{True},\texttt{True}), (\texttt{False},\texttt{False})\}$$

Relating Functions

If we had a `Bool` base type, we would first define:

$$\mathcal{V}[\![\texttt{Bool}]\!] := \{(\texttt{True},\texttt{True}), (\texttt{False},\texttt{False})\}$$

And then continue with function types:

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] := \Big\{(\lambda x : \tau_1.\ t_1, \lambda x : \tau_1.\ t_2) \mid \qquad \Big\}$$

Relating Functions

If we had a `Bool` base type, we would first define:

$$\mathcal{V}[\![\texttt{Bool}]\!] := \{(\texttt{True},\texttt{True}), (\texttt{False},\texttt{False})\}$$

And then continue with function types:

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] := \Big\{(\lambda x : \tau_1.\ t_1, \lambda x : \tau_1.\ t_2) \mid$$
$$\mathsf{wt}_{\tau_1 \to \tau_2}(\lambda x : \tau_1.\ t_1, \lambda x : \tau_1.\ t_2)$$
$$\Big\}$$

Relating Functions

If we had a `Bool` base type, we would first define:

$$\mathcal{V}[\![\texttt{Bool}]\!] := \{(\texttt{True},\texttt{True}), (\texttt{False},\texttt{False})\}$$

And then continue with function types:

$$\begin{aligned}\mathcal{V}[\![\tau_1 \to \tau_2]\!] := \Big\{(\lambda x : \tau_1.\ t_1, \lambda x : \tau_1.\ t_2) \mid \\ \mathrm{wt}_{\tau_1 \to \tau_2}(\lambda x : \tau_1.\ t_1, \lambda x : \tau_1.\ t_2) \\ \land \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1]\!].\ \big(t_1[v_1/x], t_2[v_2/x]\big) \in \mathcal{E}[\![\tau_2]\!]\Big\}\end{aligned}$$

Relating Type Abstractions: First Try

$$\mathcal{V}[\![\forall\alpha.\ \tau]\!] := \Big\{(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \mid \qquad \Big\}$$

Relating Type Abstractions: First Try

$$\mathcal{V}[\![\forall\alpha.\,\tau]\!] := \Big\{(\Lambda\alpha.\,t_1, \Lambda\alpha.\,t_2) \mid$$

$$\mathsf{wt}_{\forall\alpha.\,\tau}(\Lambda\alpha.\,t_1, \Lambda\alpha.\,t_2)$$

$$\Big\}$$

Relating Type Abstractions: First Try

$$
\begin{aligned}
\mathcal{V}[\![\forall\alpha.\ \tau]\!] := \Big\{ &(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \mid \\
&\quad \mathsf{wt}_{\forall\alpha.\ \tau}(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \\
&\quad \wedge \forall\tau_1, \tau_2.\ \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau[?/\alpha]]\!] \Big\}
\end{aligned}
$$

Relating Type Abstractions: First Try

$$\mathcal{V}[\![\forall\alpha.\ \tau]\!] := \Big\{(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \mid \\ \mathsf{wt}_{\forall\alpha.\ \tau}(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \\ \wedge \forall\tau_1, \tau_2.\ \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau[?/\alpha]]\!]\Big\}$$

Under which type should $t_1[\tau_1/\alpha]$ and $t_2[\tau_2/\alpha]$ be related under?

Relating Type Abstractions: Second Try

Trick: keep track of the chosen types in a substitution ρ

Relating Type Abstractions: Second Try

Trick: keep track of the chosen types in a substitution ρ

$$\begin{aligned}\mathcal{V}[\![\forall\alpha.\ \tau]\!]_{\rho} := \Big\{&(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \mid \\ &\mathrm{wt}_{\forall\alpha.\ \rho(\tau)}(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \\ &\wedge \forall\tau_1, \tau_2.\ \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha\mapsto(\tau_1,\tau_2)]}\Big\},\end{aligned}$$

Relating Variables: First Try

$$\mathcal{V}[\![\alpha]\!]_\rho := \Big\{(v_1, v_2) \mid \qquad\qquad \Big\}$$

Relating Variables: First Try

$$\mathcal{V}[\![\alpha]\!]_\rho := \Big\{(v_1, v_2) \mid$$

$$\mathsf{wt}_{\rho(\alpha)}(v_1, v_2)$$

$$\Big\}$$

Relating Variables: First Try

$$\mathcal{V}[\![\alpha]\!]_\rho := \Big\{(v_1, v_2) \mid \mathsf{wt}_{\rho(\alpha)}(v_1, v_2) \wedge \, ? \Big\}$$

How should we relate values of possibly different types?

Relating Type Abstractions: Final Version

Idea: whenever we pick two types τ_1, τ_2, we also pick a relation on τ_1 and τ_2.

Relating Type Abstractions: Final Version

Idea: whenever we pick two types τ_1, τ_2, we also pick a relation on τ_1 and τ_2.

$$
\begin{aligned}
\mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho \coloneqq \Big\{(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \mid\ & \\
& \mathsf{wt}_{\forall\alpha.\ \rho(\tau)}(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \\
& \wedge \forall \tau_1, \tau_2, R \in \mathcal{R}(\tau_1, \tau_2). \\
& \quad \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha \mapsto (\tau_1, \tau_2, R)]}\Big\},
\end{aligned}
$$

Relating Type Abstractions: Final Version

Idea: whenever we pick two types τ_1, τ_2, we also pick a relation on τ_1 and τ_2.

$$
\begin{aligned}
\mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho := \Big\{(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \mid\ & \\
& \mathrm{wt}_{\forall\alpha.\ \rho(\tau)}(\Lambda\alpha.\ t_1, \Lambda\alpha.\ t_2) \\
& \wedge \forall\tau_1, \tau_2, R \in \mathcal{R}(\tau_1, \tau_2). \\
& \quad \big(t_1[\tau_1/\alpha], t_2[\tau_2/\alpha]\big) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha\mapsto(\tau_1,\tau_2,R)]}\Big\},
\end{aligned}
$$

where R is just a relation of closed, well-typed values:

$$
\mathcal{R}(\tau_1, \tau_2) := \big\{R \in \mathcal{P}(\mathit{Val} \times \mathit{Val}) \mid \forall(v_1, v_2) \in R.\ \mathrm{wt}_{\tau_1}(v_1) \wedge \mathrm{wt}_{\tau_2}(v_2)\big\}.
$$

Final Updates

$$\mathcal{V}[\![\alpha]\!]_\rho := \left\{(v_1, v_2) \in R \mid \mathsf{wt}_{\rho(\alpha)}(v_1, v_2) \wedge \rho(\alpha) = (\tau_1, \tau_2, R)\right\}$$

Final Updates

$$\mathcal{V}[\![\alpha]\!]_\rho := \big\{(v_1, v_2) \in R \mid \mathsf{wt}_{\rho(\alpha)}(v_1, v_2) \wedge \rho(\alpha) = (\tau_1, \tau_2, R)\big\}$$

$$\begin{aligned}\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho := \Big\{&\big(\lambda x : \rho(\tau_1).\ t_1, \lambda x : \rho(\tau_1).\ t_2\big) \mid \\ &\mathsf{wt}_{\rho(\tau_1)\to\rho(\tau_2)}\big(\lambda x : \rho(\tau_1).\ t_1, \lambda x : \rho(\tau_1).\ t_2\big) \\ &\wedge \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho.\ \big(t_1[v_1/x], t_2[v_2/x]\big) \in \mathcal{E}[\![\tau_2]\!]_\rho\Big\}\end{aligned}$$

$$\mathcal{E}[\![\tau]\!]_\rho := \big\{(t_1, t_2) \mid \mathsf{wt}_{\rho(\tau)}(t_1, t_2) \wedge (t_1\!\downarrow, t_2\!\downarrow) \in \mathcal{V}[\![\tau]\!]_\rho\big\}$$

Parametricity Theorem

Theorem (Parametricity Theorem)

If $\vdash t : \tau$ *then* $(t, t) \in \mathcal{E}[\![\tau]\!]_\emptyset$.

In other words: every closed, well-typed term is related to itself.

Examples of Free Theorems

Rearranging Lists

Assume we are given a value $t : \forall\alpha.\, \text{List}\ \alpha \to \text{List}\ \alpha$.

Rearranging Lists

Assume we are given a value $t : \forall\alpha.\, \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$.

We show that

$$\forall\alpha, \alpha', (f : \alpha \to \alpha'), (xs : \mathsf{List}\,\alpha).\ \mathsf{map}\, f\, (t\,[\alpha]\, xs) =^{*} t\,[\alpha']\,(\mathsf{map}\, f\, xs)$$

Rearranging Lists

Assume we are given a value $t : \forall\alpha.\, \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$.

We show that

$$\forall\alpha, \alpha', (f : \alpha \to \alpha'), (xs : \mathsf{List}\,\alpha).\ \mathsf{map}\, f\, (t\,[\alpha]\, xs) =^{*} t\,[\alpha']\, (\mathsf{map}\, f\, xs)$$

Pick any $\tau, \tau', f : \tau \to \tau'$.

Rearranging Lists

Assume we are given a value $t : \forall\alpha.\, \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$.

We show that

$$\forall\alpha, \alpha', (f : \alpha \to \alpha'), (xs : \mathsf{List}\,\alpha).\ \mathsf{map}\,f\,(t\,[\alpha]\,xs) =^* t\,[\alpha']\,(\mathsf{map}\,f\,xs)$$

Pick any $\tau, \tau', f : \tau \to \tau'$. By the Parametricity Theorem, for any R and $(xs, xs') \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_{[\alpha\mapsto(\tau,\tau',R)]}$, we have $(t\,[\tau]\,xs, t\,[\tau']\,xs') \in \mathcal{E}[\![\mathsf{List}\ \alpha]\!]_{[\alpha\mapsto(\tau,\tau',R)]}$.

Rearranging Lists

Assume we are given a value $t : \forall\alpha.\, \text{List}\ \alpha \to \text{List}\ \alpha$.

We show that

$$\forall\alpha, \alpha', (f : \alpha \to \alpha'), (xs : \text{List}\,\alpha).\ \text{map}\,f\,(t\,[\alpha]\,xs) =^* t\,[\alpha']\,(\text{map}\,f\,xs)$$

Pick any $\tau, \tau', f : \tau \to \tau'$. By the Parametricity Theorem, for any R and $(xs, xs') \in \mathcal{V}[\![\text{List}\ \alpha]\!]_{[\alpha\mapsto(\tau,\tau',R)]}$, we have $(t\,[\tau]\,xs, t\,[\tau']\,xs') \in \mathcal{E}[\![\text{List}\ \alpha]\!]_{[\alpha\mapsto(\tau,\tau',R)]}$.

If we specialise $R = \left\{(v, (f\,v){\downarrow}) \mid \text{wt}_\tau(v)\right\}$, the property $(xs, xs') \in \mathcal{V}[\![\text{List}\ \alpha]\!]_{[\alpha\mapsto(\tau,\tau',R)]}$ translates to $xs' =^* \text{map}\,f\,xs$.

Rearranging Lists

Assume we are given a value $t : \forall\alpha.\, \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$.

We show that

$$\forall \alpha, \alpha', (f : \alpha \to \alpha'), (xs : \mathsf{List}\,\alpha).\ \mathsf{map}\, f\, (t\,[\alpha]\, xs) =^{*} t\,[\alpha']\,(\mathsf{map}\, f\, xs)$$

Pick any $\tau, \tau', f : \tau \to \tau'$. By the Parametricity Theorem, for any R and $(xs, xs') \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$, we have $(t\,[\tau]\, xs, t\,[\tau']\, xs') \in \mathcal{E}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$.

If we specialise $R = \left\{(v, (f\, v){\downarrow}) \mid \mathsf{wt}_{\tau}(v)\right\}$, the property $(xs, xs') \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$ translates to $xs' =^{*} \mathsf{map}\, f\, xs$.

Similarly, $\left(t\,[\tau]\, xs, t\,[\tau']\, xs'\right) \in \mathcal{E}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$ translates to $t\,[\tau']\, xs' =^{*} \mathsf{map}\, f\, (t\,[\tau]\, xs)$.

Rearranging Lists

Assume we are given a value $t : \forall\alpha.\, \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha$.

We show that

$$\forall \alpha, \alpha', (f : \alpha \to \alpha'), (xs : \mathsf{List}\,\alpha).\ \mathsf{map}\,f\,(t\,[\alpha]\,xs) =^* t\,[\alpha']\,(\mathsf{map}\,f\,xs)$$

Pick any $\tau, \tau', f : \tau \to \tau'$. By the Parametricity Theorem, for any R and $(xs, xs') \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$, we have $(t\,[\tau]\,xs, t\,[\tau']\,xs') \in \mathcal{E}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$.

If we specialise $R = \left\{(v, (f\,v)\!\downarrow) \mid \mathsf{wt}_\tau(v)\right\}$, the property $(xs, xs') \in \mathcal{V}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$ translates to $xs' =^* \mathsf{map}\,f\,xs$.

Similarly, $\left(t\,[\tau]\,xs, t\,[\tau']\,xs'\right) \in \mathcal{E}[\![\mathsf{List}\ \alpha]\!]_{[\alpha \mapsto (\tau, \tau', R)]}$ translates to $t\,[\tau']\,xs' =^* \mathsf{map}\,f\,(t\,[\tau]\,xs)$.

Putting it together: $t\,[\tau']\,(\mathsf{map}\,f\,xs) =^* \mathsf{map}\,f\,(t\,[\tau]\,xs)$.

Negative Results

Note that in Haskell `undefined :: forall a. a`. Can we define such a term in System F?

Negative Results

Note that in Haskell `undefined :: forall a. a`. Can we define such a term in System F?

Assume there is a value $u : \forall \alpha .\, \alpha$. Pick any τ, τ' and set $R = \emptyset$.

Negative Results

Note that in Haskell `undefined :: forall a. a`. Can we define such a term in System F?

Assume there is a value $u : \forall\alpha.\,\alpha$. Pick any τ, τ' and set $R = \emptyset$. Then by the Parametricity Theorem, $\big(u\,[\tau], u[\tau']\big) \in R = \emptyset$, which is impossible.

Going Beyond System F

Further Applications and Extensions

1. Representation independence of abstract data types

Further Applications and Extensions

1. Representation independence of abstract data types
2. Free Theorems for non-total extensions with general recursion

Further Applications and Extensions

1. Representation independence of abstract data types
2. Free Theorems for non-total extensions with general recursion
3. Free Theorems for recursive data types

Further Applications and Extensions

1. Representation independence of abstract data types
2. Free Theorems for non-total extensions with general recursion
3. Free Theorems for recursive data types
4. Free Theorems for type constructors and type classes

Further Applications and Extensions

1. Representation independence of abstract data types
2. Free Theorems for non-total extensions with general recursion
3. Free Theorems for recursive data types
4. Free Theorems for type constructors and type classes
5. Free Theorems for pure type systems (and hence in particular for dependent type systems)

Further Applications and Extensions

1. Representation independence of abstract data types
2. Free Theorems for non-total extensions with general recursion
3. Free Theorems for recursive data types
4. Free Theorems for type constructors and type classes
5. Free Theorems for pure type systems (and hence in particular for dependent type systems)
6. Free Theorems for gradually typed systems

Further Applications and Extensions

1. Representation independence of abstract data types
2. Free Theorems for non-total extensions with general recursion
3. Free Theorems for recursive data types
4. Free Theorems for type constructors and type classes
5. Free Theorems for pure type systems (and hence in particular for dependent type systems)
6. Free Theorems for gradually typed systems
7. Free Theorems in interactive theorem provers (Isabelle: transfer)

How useful are the [free] theorems so generated? Only time and experience will tell…

— Wadler

How useful are the [free] theorems so generated? Only time and experience will tell...

— Wadler

It kicked off much fruitful research and the results can indeed be very useful in formal verification.

— Me

Any questions?

WADLER'S

THEOREM COMPANY

ISSUED TO	INVOICE NO.	DATE ISSUED
Audience	2021-2201	27/05/2021

DESCRIPTION	QTY	PRICE	TOTAL
(∀α. α) is uninhabited	1	£0	£0
g (*2) 1 /= 3	1	£0	£0
map f (t xs) = t (map f xs)	1	£0	£0
map f (concat xss) = concat (map (map f) xss)	1	£0	£0

ACCOUNT NAME

Philip Wadler

IBAN

GB XXXX XX X XXXX

TOTAL AMOUNT	£0
TAX	20%
AMOUNT DUE	**£0**

Natural Numbers

Assume we have a base type $\mathbb{N}$ with $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Natural Numbers

Assume we have a base type $\mathbb{N}$ with $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Assume we are given some value $t : \forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$, a type τ, and values $s : \tau \to \tau$ and $z : \tau$. What is the result of $t\,[\tau]\,s\,z$?

Natural Numbers

Assume we have a base type $\mathbb{N}$ with succ : $\mathbb{N} \to \mathbb{N}$.

Assume we are given some value $t : \forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$, a type τ, and values $s : \tau \to \tau$ and $z : \tau$. What is the result of $t\,[\tau]\,s\,z$?

Set $R \coloneqq \left\{(n, (s^n\,z){\downarrow}) \mid n \in \mathbb{N}\right\}$ and note that $(\mathsf{succ}, s) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$:

Natural Numbers

Assume we have a base type $\mathbb{N}$ with $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Assume we are given some value $t : \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$, a type τ, and values $s : \tau \to \tau$ and $z : \tau$. What is the result of $t\,[\tau]\,s\,z$?

Set $R := \big\{(n, (s^n\,z){\downarrow}) \mid n \in \mathbb{N}\big\}$ and note that $(\mathsf{succ}, s) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$:
If $(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$ then $(v_1, v_2) = \big(n, (s^n\,z){\downarrow}\big)$ for some n. Thus $(\mathsf{succ}\,v_1, s\,v_2) =^* \big(n+1, (s^{n+1}\,z){\downarrow}\big) \in R$.

Natural Numbers

Assume we have a base type $\mathbb{N}$ with $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Assume we are given some value $t : \forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$, a type τ, and values $s : \tau \to \tau$ and $z : \tau$. What is the result of $t\,[\tau]\,s\,z$?

Set $R := \big\{(n, (s^n\,z)\!\downarrow) \mid n \in \mathbb{N}\big\}$ and note that $(\mathsf{succ}, s) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{[\alpha \mapsto (\mathbb{N},\tau,R)]}$:
If $(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]_{[\alpha \mapsto (\mathbb{N},\tau,R)]}$ then $(v_1, v_2) = \big(n, (s^n\,z)\!\downarrow\big)$ for some n. Thus $(\mathsf{succ}\,v_1, s\,v_2) =^* \big(n+1, (s^{n+1}\,z)\!\downarrow\big) \in R$.

Hence, $\big(t\,[\mathbb{N}]\,\mathsf{succ}\,0, t\,[\tau]\,s\,z\big) \in \mathcal{E}[\![\alpha]\!]_{[\alpha \mapsto (\mathbb{N},\tau,R)]}$ by the Parametricity Theorem.

Natural Numbers

Assume we have a base type $\mathbb{N}$ with $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Assume we are given some value $t : \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$, a type τ, and values $s : \tau \to \tau$ and $z : \tau$. What is the result of $t\,[\tau]\,s\,z$?

Set $R := \big\{(n, (s^n\,z){\downarrow}) \mid n \in \mathbb{N}\big\}$ and note that $(\mathsf{succ}, s) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$:
If $(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$ then $(v_1, v_2) = \big(n, (s^n\,z){\downarrow}\big)$ for some n. Thus $(\mathsf{succ}\,v_1, s\,v_2) =^* \big(n + 1, (s^{n+1}\,z){\downarrow}\big) \in R$.

Hence, $\big(t\,[\mathbb{N}]\,\mathsf{succ}\,0, t\,[\tau]\,s\,z\big) \in \mathcal{E}[\![\alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$ by the Parametricity Theorem. So there is $v' : \mathbb{N}$ and $v : \tau$ such that $t\,[\mathbb{N}]\,\mathsf{succ}\,0 \to^* v'$ and $t\,[\tau]\,s\,z \to^* v$ and $(v', v) \in R$.

Natural Numbers

Assume we have a base type $\mathbb{N}$ with succ : $\mathbb{N} \to \mathbb{N}$.

Assume we are given some value $t : \forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$, a type τ, and values $s : \tau \to \tau$ and $z : \tau$. What is the result of $t\,[\tau]\,s\,z$?

Set $R := \{(n, (s^n\,z){\downarrow}) \mid n \in \mathbb{N}\}$ and note that $(\mathrm{succ}, s) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$:
If $(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$ then $(v_1, v_2) = \big(n, (s^n\,z){\downarrow}\big)$ for some n. Thus $(\mathrm{succ}\,v_1, s\,v_2) =^* \big(n+1, (s^{n+1}\,z){\downarrow}\big) \in R$.

Hence, $\big(t\,[\mathbb{N}]\,\mathrm{succ}\,0, t\,[\tau]\,s\,z\big) \in \mathcal{E}[\![\alpha]\!]_{[\alpha \mapsto (\mathbb{N}, \tau, R)]}$ by the Parametricity Theorem. So there is $v' : \mathbb{N}$ and $v : \tau$ such that $t\,[\mathbb{N}]\,\mathrm{succ}\,0 \to^* v'$ and $t\,[\tau]\,s\,z \to^* v$ and $(v', v) \in R$. Hence $v =^* s^n\,z$ where n is determined by $t\,[\mathbb{N}]\,\mathrm{succ}\,0 \to^* v' =^* n$.

Negative Results II

We cannot define a polymorphic equality function $\mathsf{eq} : \forall \alpha.\, \alpha \to \alpha \to \mathsf{Bool}$:

We would get $\mathsf{eq}\,[\tau]\,v_1\,v_2 =^* \mathsf{eq}\,[\tau']\,(f\,v_1)\,(f\,v_2)$ for any $f : \tau \to \tau', v_1 : \tau, v_2 : \tau$.