Free Theorems and “Real” Languages

Janis Voigtländer

Technische Universität Dresden

April 24th, 2009

Free Theorems and Applications

As we have seen, types:

- constrain the behaviour of programs

Free Theorems and Applications

As we have seen, types:

- constrain the behaviour of programs
- thus lead to interesting insights about programs

Free Theorems and Applications

As we have seen, types:

- constrain the behaviour of programs
- thus lead to interesting insights about programs
- combine well with algebraic techniques, equational reasoning

Free Theorems and Applications

As we have seen, types:

- constrain the behaviour of programs
- thus lead to interesting insights about programs
- combine well with algebraic techniques, equational reasoning

Application areas include:

- efficiency-improving program transformations
- more specific domains

Free Theorems and Applications

As we have seen, types:

- constrain the behaviour of programs
- thus lead to interesting insights about programs
- combine well with algebraic techniques, equational reasoning

Application areas include:

- efficiency-improving program transformations
- more specific domains

But:

- We could ask for more (expressive) type features.

Free Theorems and Applications

As we have seen, types:

- constrain the behaviour of programs
- thus lead to interesting insights about programs
- combine well with algebraic techniques, equational reasoning

Application areas include:

- efficiency-improving program transformations
- more specific domains

But:

- We could ask for more (expressive) type features.
- We have not been considering a full programming language.

Example Feature: Type Classes [Wadler & Blott 1989]

We used that for every

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)$$

for arbitrary f and l.

Example Feature: Type Classes [Wadler & Blott 1989]

We used that for every

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)$$

for arbitrary f and l.

What about

$$\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \quad ?$$

Example Feature: Type Classes [Wadler & Blott 1989]

We used that for every

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)$$

for arbitrary f and l.

What about

$$\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \quad ?$$

The above free theorem fails!

Consider, e.g., $\texttt{get} = \texttt{nub}$, $f = \texttt{const}\ 1$, and $l = [1, 2]$.

Why `map` f (`get` l) = `get` (`map` f l), Intuitively

- `get` $:: [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.
- The lists (`map` f l) and l always have equal length.
- `get` always chooses "the same" elements from (`map` f l) for output as it does from l, except that in the former case it outputs their images under f.
- (`get` (`map` f l)) is equivalent to (`map` f (`get` l)).
- That is what was claimed!

Why `map` f (`get` l) $=$ `get` (`map` f l), Intuitively

- `get` :: $\text{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \text{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \text{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

$\lightning$ Not true! Also possible: check elements of l for equality.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \text{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

$\lightning$ Not true! Also possible: check elements of l for equality.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \mathrm{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

$\lightning$ Not true! Also possible: check elements of l for equality.

- The lists ($\texttt{map}\ f\ l$) and l always have equal length.

But equality checks on corresponding elements are not always guaranteed to have the same outcome!

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \mathrm{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

$\lightning$ Not true! Also possible: check elements of l for equality.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.

But equality checks on corresponding elements are not always guaranteed to have the same outcome! They are, if f is “injective”.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

↯ Not true! Also possible: check elements of l for equality.

- The lists ($\texttt{map}\ f\ l$) and l always have equal length.

But equality checks on corresponding elements are not always guaranteed to have the same outcome! They are, if f is "injective".

- Then, $\texttt{get}$ always chooses "the same" elements from ($\texttt{map}\ f\ l$) for output as it does from l,

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

$\lightning$ Not true! Also possible: check elements of l for equality.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.

But equality checks on corresponding elements are not always guaranteed to have the same outcome! They are, if f is "injective".

- Then, $\texttt{get}$ always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as it does from l, except that in the former case it outputs their images under f.

Why $\mathtt{map}\ f\ (\mathtt{get}\ l) = \mathtt{get}\ (\mathtt{map}\ f\ l)$, Intuitively

- $\mathtt{get} :: \mathrm{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

↯ Not true! Also possible: check elements of l for equality.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.

But equality checks on corresponding elements are not always guaranteed to have the same outcome! They are, if f is "injective".

- Then, $\mathtt{get}$ always chooses "the same" elements from $(\mathtt{map}\ f\ l)$ for output as it does from l, except that in the former case it outputs their images under f.
- $(\mathtt{get}\ (\mathtt{map}\ f\ l))$ is equivalent to $(\mathtt{map}\ f\ (\mathtt{get}\ l))$.

Why $\texttt{map}\ f\ (\texttt{get}\ l) = \texttt{get}\ (\texttt{map}\ f\ l)$, Intuitively

- $\texttt{get} :: \text{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

$\lightning$ Not true! Also possible: check elements of l for equality.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.

But equality checks on corresponding elements are not always guaranteed to have the same outcome! They are, if f is "injective".

- Then, $\texttt{get}$ always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as it does from l, except that in the former case it outputs their images under f.
- $(\texttt{get}\ (\texttt{map}\ f\ l))$ is equivalent to $(\texttt{map}\ f\ (\texttt{get}\ l))$.
- This gives a revised free theorem.

More Formally: Dictionary Translation

Every

$$\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$$

can be seen as a

$$\texttt{get}' :: (\alpha \to \alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

More Formally: Dictionary Translation

Every

$$\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$$

can be seen as a

$$\texttt{get}' :: (\alpha \to \alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]\,,$$

where for every type τ that is an instance of Eq,

$$\texttt{get}_\tau \quad = \quad \texttt{get}'_\tau\ (==)_\tau$$

More Formally: Dictionary Translation

Every

$$\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$$

can be seen as a

$$\texttt{get}' :: (\alpha \to \alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]\,,$$

where for every type τ that is an instance of Eq,

$$\texttt{get}_\tau \quad = \quad \texttt{get}'_\tau\ (\texttt{==})_\tau$$

The free theorem for $\texttt{get}'$ is that

$$\texttt{map}\ f\ (\texttt{get}'\ p\ l) \quad = \quad \texttt{get}'\ q\ (\texttt{map}\ f\ l)$$

provided that for every x and y, $p\ x\ y = q\ (f\ x)\ (f\ y)$.

More Formally: Dictionary Translation

Every

$$\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$$

can be seen as a

$$\texttt{get}' :: (\alpha \rightarrow \alpha \rightarrow \mathsf{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]\,,$$

where for every type τ that is an instance of Eq,

$$\texttt{get}_\tau \quad = \quad \texttt{get}'_\tau\ (==)_\tau$$

The free theorem for $\texttt{get}'$ is that

$$\texttt{map}\ f\ (\texttt{get}'\ p\ l) \quad = \quad \texttt{get}'\ q\ (\texttt{map}\ f\ l)$$

provided that for every x and y, $p\ x\ y = q\ (f\ x)\ (f\ y)$.

This means that

$$\texttt{map}\ f\ (\texttt{get}'\ (==)\ l) \quad = \quad \texttt{get}'\ (==)\ (\texttt{map}\ f\ l)$$

provided that for every x and y, $x == y$ iff $(f\ x) == (f\ y)$.

More Formally: Dictionary Translation

Every

$$\texttt{get} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$$

can be seen as a

$$\texttt{get}' :: (\alpha \rightarrow \alpha \rightarrow \mathsf{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]\,,$$

where for every type τ that is an instance of Eq,

$$\texttt{get}_\tau \quad = \quad \texttt{get}'_\tau\ (\texttt{==})_\tau$$

The free theorem for $\texttt{get}'$ is that

$$\texttt{map}\ f\ (\texttt{get}'\ p\ l) \quad = \quad \texttt{get}'\ q\ (\texttt{map}\ f\ l)$$

provided that for every x and y, $p\ x\ y = q\ (f\ x)\ (f\ y)$.

This means that

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)$$

provided that for every x and y, $x \texttt{==} y$ iff $(f\ x) \texttt{==} (f\ y)$.

Another Feature: General Recursion

We claimed that for every

$$\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

we have

$$\mathtt{g}\ p\ (\mathtt{map}\ f\ l) \quad = \quad \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$$

for arbitrary p, f, and l.

Another Feature: General Recursion

We claimed that for every

$$\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

we have

$$\mathtt{g}\ p\ (\mathtt{map}\ f\ l) \quad = \quad \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$$

for arbitrary p, f, and l.

What about

$$\begin{array}{l} \mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\ \mathtt{g}\ p\ l = [\mathtt{head}\ (\mathtt{g}\ p\ l)] \end{array} \qquad ?$$

Another Feature: General Recursion

We claimed that for every

$$\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

we have

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

for arbitrary p, f, and l.

What about

$$\begin{array}{l} \texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\ \texttt{g}\ p\ l = [\texttt{head}\ (\texttt{g}\ p\ l)] \end{array} \qquad ?$$

The above free theorem fails!

Consider, e.g., $p = \texttt{id}$, $f = \texttt{const}\ \mathsf{True}$, and $l = [\,]$.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\texttt{g}$ with p always chooses “the same” elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.
- $(\texttt{g}\ p\ (\texttt{map}\ f\ l))$ is equivalent to $(\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l))$.
- That is what was claimed!

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- $\mathtt{g} :: (\alpha \rightarrow \mathsf{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

$\lightning$ Not true! Also possible: $\perp$.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

- The output list can only contain elements from the input list l.

$\lightning$ Not true! Also possible: $\perp$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

- The output list can only contain elements from the input list l.

$\lightning$ Not true! Also possible: $\bot$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\perp$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

↯ Not true! Also possible: checking outcome of p on $\perp$.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\perp$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

↯ Not true! Also possible: checking outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\bot$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

↯ Not true! Also possible: checking outcome of p on $\bot$.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\bot$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

↯ Not true! Also possible: checking outcome of p on $\bot$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\bot$ has the same outcome as applying $(p \circ f)$ to $\bot$, provided f is strict ($f\ \bot = \bot$).

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- $\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

- The output list can only contain elements from the input list l.

↯ Not true! Also possible: $\perp$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

↯ Not true! Also possible: checking outcome of p on $\perp$.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\mathtt{g}$ with p always chooses "the same" elements from $(\mathtt{map}\ f\ l)$ for output as does $\mathtt{g}$ with $(p \circ f)$ from l,

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- The output list can only contain elements from the input list l.

$\lightning$ Not true! Also possible: $\perp$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

$\lightning$ Not true! Also possible: checking outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- The output list can only contain elements from the input list l.

$\lightning$ Not true! Also possible: $\perp$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

$\lightning$ Not true! Also possible: checking outcome of p on $\perp$.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f \perp = \perp)$.

- $\mathtt{g}$ with p always chooses "the same" elements from $(\mathtt{map}\ f\ l)$ for output as does $\mathtt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\perp$.

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

$\lightning$ Not true! Also possible: $\bot$.

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

$\lightning$ Not true! Also possible: checking outcome of p on $\bot$.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\bot$ has the same outcome as applying $(p \circ f)$ to $\bot$, provided f is strict $(f\ \bot = \bot)$.

- $\mathtt{g}$ with p always chooses "the same" elements from $(\mathtt{map}\ f\ l)$ for output as does $\mathtt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\bot$.

- $(\mathtt{g}\ p\ (\mathtt{map}\ f\ l))$ is equivalent to $(\mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l))$,

Why $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$, Intuitively

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

$\lightning$ Not true! Also possible: checking outcome of p on $\perp$.

- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\perp$ has the same outcome as applying $(p \circ f)$ to $\perp$, provided f is strict $(f\ \perp = \perp)$.

- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\perp$.

- $(\texttt{g}\ p\ (\texttt{map}\ f\ l))$ is equivalent to $(\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l))$, if f is strict.

Why $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$, Intuitively

- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

$\lightning$ Not true! Also possible: checking outcome of p on $\bot$.

- The lists $(\mathtt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\mathtt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Applying p to $\bot$ has the same outcome as applying $(p \circ f)$ to $\bot$, provided f is strict $(f\ \bot = \bot)$.

- $\mathtt{g}$ with p always chooses "the same" elements from $(\mathtt{map}\ f\ l)$ for output as does $\mathtt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f.

But they may also choose, at the same positions, to output $\bot$.

- $(\mathtt{g}\ p\ (\mathtt{map}\ f\ l))$ is equivalent to $(\mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l))$, if f is strict.

Recall: The Polymorphic Lambda Calculus

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

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

$$\Gamma, x : \tau \vdash x : \tau \qquad [\![x]\!]_{\theta,\sigma} = \sigma(x)$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2} \qquad [\![\lambda x : \tau_1.t]\!]_{\theta,\sigma}\ a = [\![t]\!]_{\theta,\sigma[x \mapsto a]}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2} \qquad [\![t\ u]\!]_{\theta,\sigma} = [\![t]\!]_{\theta,\sigma}\ [\![u]\!]_{\theta,\sigma}$$

$$\frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau} \qquad [\![\Lambda\alpha.t]\!]_{\theta,\sigma}\ S = [\![t]\!]_{\theta[\alpha \mapsto S],\sigma}$$

$$\frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash (t\ \tau') : \tau[\tau'/\alpha]} \qquad [\![t\ \tau']\!]_{\theta,\sigma} = [\![t]\!]_{\theta,\sigma}\ [\![\tau']\!]_{\theta}$$

Adding General Recursion

Terms: $t := \quad \cdots \mid \textbf{fix } t$

Adding General Recursion

Terms: $t := \ \cdots \mid \textbf{fix } t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix } t) : \tau}$$

Adding General Recursion

Terms: $t := \;\; \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders now, and:

$$[\![\textbf{fix}\ t]\!]_{\theta,\sigma} = \bigsqcup_{i \geq 0} ([\![t]\!]^{i}_{\theta,\sigma}\ \bot).$$

Adding General Recursion

Terms: $t := \quad \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders now, and:

$$[\![\textbf{fix}\ t]\!]_{\theta,\sigma} = \bigsqcup_{i \geq 0} ([\![t]\!]^{i}_{\theta,\sigma}\ \bot).$$

And what about the parametricity theorem?

Adding General Recursion

Terms: $t := \quad \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders now, and:

$$[\![\textbf{fix}\ t]\!]_{\theta,\sigma} = \bigsqcup_{i \geq 0} ([\![t]\!]^i_{\theta,\sigma}\ \bot).$$

And what about the parametricity theorem?

The relevant inductive case is:

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

Adding General Recursion

Terms: $t := \ \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders now, and:

$$[\![\textbf{fix}\ t]\!]_{\theta,\sigma} = \bigsqcup_{i \geq 0} ([\![t]\!]^{i}_{\theta,\sigma}\ \bot).$$

And what about the parametricity theorem?

The relevant inductive case is:

$$\frac{\Gamma \vdash t : \tau \to \tau}{([\![\textbf{fix}\ t]\!]_{\theta_1,\sigma_1}, [\![\textbf{fix}\ t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}}$$

Adding General Recursion

Terms: $t := \ \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders now, and:

$$[\![\textbf{fix}\ t]\!]_{\theta,\sigma} = \bigsqcup_{i \geq 0} ([\![t]\!]^{i}_{\theta,\sigma}\ \bot).$$

And what about the parametricity theorem?

The relevant inductive case is:

$$\frac{([\![t]\!]_{\theta_1,\sigma_1}, [\![t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau\to\tau,\rho}}{([\![\textbf{fix}\ t]\!]_{\theta_1,\sigma_1}, [\![\textbf{fix}\ t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}}$$

Adding General Recursion

Terms: $t := \ \cdots \mid \textbf{fix}\ t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders now, and:

$$[\![\textbf{fix}\ t]\!]_{\theta,\sigma} = \bigsqcup_{i \geq 0} ([\![t]\!]^i_{\theta,\sigma}\ \bot).$$

And what about the parametricity theorem?

The relevant inductive case is:

$$\frac{\forall (a_1, a_2) \in \Delta_{\tau,\rho}.\ ([\![t]\!]_{\theta_1,\sigma_1}\ a_1, [\![t]\!]_{\theta_2,\sigma_2}\ a_2) \in \Delta_{\tau,\rho}}{([\![\textbf{fix}\ t]\!]_{\theta_1,\sigma_1}, [\![\textbf{fix}\ t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}}$$

Adding General Recursion

Terms: $t := \;\cdots \mid \textbf{fix } t$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix } t) : \tau}$$

To provide semantics, types are interpreted as pointed complete partial orders now, and:

$$[\![\textbf{fix } t]\!]_{\theta,\sigma} = \bigsqcup_{i \geq 0} ([\![t]\!]^{i}_{\theta,\sigma} \perp).$$

And what about the parametricity theorem?

The relevant inductive case is:

$$\frac{\forall (a_1, a_2) \in \Delta_{\tau,\rho}.\ ([\![t]\!]_{\theta_1,\sigma_1}\ a_1, [\![t]\!]_{\theta_2,\sigma_2}\ a_2) \in \Delta_{\tau,\rho}}{([\![\textbf{fix } t]\!]_{\theta_1,\sigma_1}, [\![\textbf{fix } t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}}$$

The parametricity theorem still holds, provided all relations are strict and continuous.

Automatic Generation of Free Theorems

At `http://linux.tcs.inf.tu-dresden.de/~voigt/ft`:

This tool allows to generate free theorems for sublanguages of Haskell as described here.

The source code of the underlying library and a shell-based application using it is available here and here.

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]" or simply "filter":

```
g :: (a -> Bool) -> [a] -> [a]
```

Please choose a sublanguage of Haskell:

- ( ) no bottoms (hence no general recursion and no selective strictness)
- (•) general recursion but no selective strictness
- ( ) general recursion and selective strictness

Please choose a theorem style (without effect in the sublanguage with no bottoms):

- (•) equational
- ( ) inequational

[Generate]

Adding Selective Strictness

Terms: $t := \quad \cdots \mid \textbf{seq}\ t\ t$

Adding Selective Strictness

Terms: $t := \ \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Adding Selective Strictness

Terms: $t := \ \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Semantics:

$$[\![\textbf{seq}\ t_1\ t_2]\!]_{\theta,\sigma} = \begin{cases} \bot & \text{if } [\![t_1]\!]_{\theta,\sigma} = \bot \\ [\![t_2]\!]_{\theta,\sigma} & \text{if } [\![t_1]\!]_{\theta,\sigma} \neq \bot. \end{cases}$$

Adding Selective Strictness

Terms: $t := \; \cdots \mid \textbf{seq}\ t\ t$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\textbf{seq}\ t_1\ t_2) : \tau_2}$$

Semantics:

$$[\![\textbf{seq}\ t_1\ t_2]\!]_{\theta,\sigma} = \begin{cases} \bot & \text{if } [\![t_1]\!]_{\theta,\sigma} = \bot \\ [\![t_2]\!]_{\theta,\sigma} & \text{if } [\![t_1]\!]_{\theta,\sigma} \neq \bot. \end{cases}$$

The parametricity theorem is jeopardised again!

Without **seq**, $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements and on $\perp$.
- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- Applying p to $\perp$ has the same outcome as applying $(p \circ f)$, provided f is strict.
- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that in the former case it outputs their images under f, and they may also choose, at the same positions, to output $\perp$.
- $(\texttt{g}\ p\ (\texttt{map}\ f\ l)) = (\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l))$, if f is strict.

With **seq**, $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$?

- $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.

With **seq**, $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$?

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.

With **seq**, $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$?

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.

With **seq**, $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$?

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements and on $\perp$.

With **seq**, $\texttt{g}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$?

- $\texttt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements and on $\perp$.

$\lightning$ Not true! Also possible:

- checking elements from l for being $\perp$
- checking p for being $\perp$

With **seq**, $\mathtt{g}\ p\ (\mathtt{map}\ f\ l) = \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$?

- $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly.
- The output list can only contain elements from the input list l and $\perp$.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements and on $\perp$.

$\lightning$ Not true! Also possible:

  - checking elements from l for being $\perp$
  - checking p for being $\perp$

. . . ???

Revising Free Theorems

[Wadler 1989] : for every $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

- if f strict.

Revising Free Theorems

[Wadler 1989] : for every $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

- if f strict.

[Johann & V. 2004] : in presence of **seq**, if additionally:

- $p \neq \bot$,
- f total $(\forall x \neq \bot.\ f\ x \neq \bot)$.

Revising Free Theorems

[Wadler 1989] : for every $\mathtt{g} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$,

$$\mathtt{g}\ p\ (\mathtt{map}\ f\ l) \;=\; \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$$

- if f strict.

[Johann & V. 2004] : in presence of **seq**, if additionally:

- $p \neq \bot$,
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

[Johann & V. 2009] : take finite failures into account

[Stenger & V. 2009] : take imprecise error semantics into account

Automatic Generation of Counterexamples

The ideal scenario:

- I give the system a type, say $g :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$.

Automatic Generation of Counterexamples

The ideal scenario:

- I give the system a type, say $\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$.
- The system gives me the free theorem. Here:

  for strict f, $\quad \texttt{g}\ p\ (\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$

Automatic Generation of Counterexamples

The ideal scenario:

- I give the system a type, say $\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$.
- The system gives me the free theorem. Here:

  for strict f, $\quad \mathtt{g}\ p\ (\mathtt{map}\ f\ l) \;=\; \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$
- I ask: why must f be strict? What if it were not?

Automatic Generation of Counterexamples

The ideal scenario:

- I give the system a type, say $\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$.
- The system gives me the free theorem. Here:

  for strict f, $\quad \mathtt{g}\ p\ (\mathtt{map}\ f\ l) \;=\; \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$

- I ask: why must f be strict? What if it were not?
- The system gives me concrete $\mathtt{g}$, p, l, and (nonstrict) f that refute the thus naivified free theorem.

Idea 1: First Capture *Non*-Counterexamples

Replace

$$\frac{\Gamma \vdash t : \tau \rightarrow \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau}$$

by

$$\frac{\Gamma \vdash \tau \in \text{Pointed} \qquad \Gamma \vdash t : \tau \rightarrow \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau}$$

Idea 1: First Capture *Non*-Counterexamples

Replace

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau}$$

by

$$\frac{\Gamma \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\mathbf{fix}\ t) : \tau},$$

where

$$\frac{\alpha^* \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Pointed}} \qquad \frac{\Gamma \vdash \tau_2 \in \mathsf{Pointed}}{\Gamma \vdash \tau_1 \to \tau_2 \in \mathsf{Pointed}}$$

$$\Gamma \vdash \mathsf{Bool} \in \mathsf{Pointed} \qquad \Gamma \vdash [\tau] \in \mathsf{Pointed}$$

Idea 1: First Capture *Non*-Counterexamples

Replace

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau}$$

by

$$\frac{\Gamma \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t : \tau \to \tau}{\Gamma \vdash (\textbf{fix}\ t) : \tau},$$

where

$$\frac{\alpha^* \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Pointed}} \qquad \frac{\Gamma \vdash \tau_2 \in \mathsf{Pointed}}{\Gamma \vdash \tau_1 \to \tau_2 \in \mathsf{Pointed}}$$

$$\Gamma \vdash \mathsf{Bool} \in \mathsf{Pointed} \qquad \Gamma \vdash [\tau] \in \mathsf{Pointed}$$

Gain: Relations interpreting non-Pointed types need not be strict anymore, but parametricity theorem still holds! [Launchbury & Paterson 1996]

Idea 2: Search for Terms in the Difference Set

For the example, search for a $\mathtt{g}$ such that

$$\alpha^* \vdash \mathtt{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathtt{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

Idea 2: Search for Terms in the Difference Set

For the example, search for a $\mathtt{g}$ such that

$$\alpha^* \vdash \mathtt{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathtt{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash (\mathbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Idea 2: Search for Terms in the Difference Set

For the example, search for a g such that

$$\alpha^* \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash (\textbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Problem: For term search, not all rules are "syntax-directed".

Idea 2: Search for Terms in the Difference Set

For the example, search for a g such that

$$\alpha^* \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash (\mathbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Problem: For term search, not all rules are "syntax-directed".

Particularly:

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2}$$

Idea 2: Search for Terms in the Difference Set

For the example, search for a g such that

$$\alpha^* \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash (\mathbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Problem: For term search, not all rules are "syntax-directed".

Particularly:

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2}$$

Idea 2: Search for Terms in the Difference Set

For the example, search for a g such that

$$\alpha^* \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

but not

$$\alpha \vdash \mathrm{g} : (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

Natural first rule:

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma \Vdash (\mathbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

Problem: For term search, not all rules are "syntax-directed".

Particularly:

$$\frac{\Gamma \vdash \qquad \tau_1 \to \tau_2 \qquad \Gamma \vdash \qquad \tau_1}{\Gamma \vdash \qquad \tau_2}$$

Idea 3: Use the Curry/Howard-Isomorphism

- [Dyckhoff 1992] gives a proof search procedure for intuitionistic propositional logic.

Idea 3: Use the Curry/Howard-Isomorphism

- [Dyckhoff 1992] gives a proof search procedure for intuitionistic propositional logic.

- It has been turned into a **fix**-free term generator for polymorphic types (Djinn, by L. Augustsson).

Idea 3: Use the Curry/Howard-Isomorphism

- [Dyckhoff 1992] gives a proof search procedure for intuitionistic propositional logic.
- It has been turned into a **fix**-free term generator for polymorphic types (Djinn, by L. Augustsson).
- We mix it with our rule

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma \Vdash (\textbf{fix}\ (\lambda x : \tau . x)) : \tau}$$

and perform further adaptations . . .

An Example

The Free Theorem

The theorem generated for functions of the type

```
f :: (a -> Int) -> Int
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
 forall p :: t1 -> Int.
  forall q :: t2 -> Int.
   (forall x :: t1. p x = q (g x)) ==> (f p = f q)
```

The Counterexample

By disregarding the strictness condition on `g` the theorem becomes wrong. The term

```
f = (\x1 -> (x1 _|_))
```

is a counterexample.

By setting `t1 = t2 = ... = ()` and

```
g = const ()
```

the following would be a consequence of the thus "naivified" free theorem:

```
(f p) = (f q)

where

p        = (\x1 -> 0)
q        = (\x1 -> (case x1 of {() -> 0}))
```

But this is wrong since with the above `f` it reduces to:

```
0 = _|_
```

Another Example

The Free Theorem

The theorem generated for functions of the type

```
f :: [a] -> Int
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
 forall x :: [t1]. f x = f (map g x)
```

The Counterexample

Disregarding the strictness condition on `g` the algorithm found no counterexample.

Another Example

The Free Theorem

The theorem generated for functions of the type

```
f :: [a] -> Int
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
 forall x :: [t1]. f x = f (map g x)
```

The Counterexample

Disregarding the strictness condition on `g` the algorithm found no counterexample.

Future work:

- investigate soundness and completeness more formally

Another Example

The Free Theorem

The theorem generated for functions of the type

```
f :: [a] -> Int
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
 forall x :: [t1]. f x = f (map g x)
```

The Counterexample

Disregarding the strictness condition on `g` the algorithm found no counterexample.

Future work:

- investigate soundness and completeness more formally
- study counterexample generation in the presence of selective strictness, finite failures, ...

Some Interesting Further Reading

- Program transformations based on free theorems:
  [Gill et al. 1993], . . . , [Svenningsson 2002], . . . ,
  [Pardo et al. 2009]

Some Interesting Further Reading

- Program transformations based on free theorems: [Gill et al. 1993], . . . , [Svenningsson 2002], . . . , [Pardo et al. 2009]
- Parametricity in operational semantics: [Pitts 2000], [Johann 2002], . . .

Some Interesting Further Reading

- Program transformations based on free theorems:
  [Gill et al. 1993], . . . , [Svenningsson 2002], . . . ,
  [Pardo et al. 2009]

- Parametricity in operational semantics:
  [Pitts 2000], [Johann 2002], . . .

- Parametricity for strict languages ("ML, not Haskell"):
  [Pitts 2005], [Ahmed 2006], . . . , [Ahmed et al. 2009]

Some Interesting Further Reading

- Program transformations based on free theorems:
  [Gill et al. 1993], . . . , [Svenningsson 2002], . . . ,
  [Pardo et al. 2009]

- Parametricity in operational semantics:
  [Pitts 2000], [Johann 2002], . . .

- Parametricity for strict languages ("ML, not Haskell"):
  [Pitts 2005], [Ahmed 2006], . . . , [Ahmed et al. 2009]

- Parametricity and dynamic typing:
  [Washburn & Weirich 2005], [Matthews & Ahmed 2008], . . .

Some Interesting Further Reading

- Program transformations based on free theorems:
  [Gill et al. 1993], . . . , [Svenningsson 2002], . . . ,
  [Pardo et al. 2009]
- Parametricity in operational semantics:
  [Pitts 2000], [Johann 2002], . . .
- Parametricity for strict languages ("ML, not Haskell"):
  [Pitts 2005], [Ahmed 2006], . . . , [Ahmed et al. 2009]
- Parametricity and dynamic typing:
  [Washburn & Weirich 2005], [Matthews & Ahmed 2008], . . .
- Parametricity and computational effects:
  [Møgelberg & Simpson 2007]

References I

A.J. Ahmed, D. Dreyer, and A. Rossberg.
State-dependent representation independence.
In *Principles of Programming Languages, Proceedings*, pages 340–353. ACM Press, 2009.

A.J. Ahmed.
Step-indexed syntactic logical relations for recursive and quantified types.
In *European Symposium on Programming, Proceedings*, volume 3924 of *LNCS*, pages 69–83. Springer-Verlag, 2006.

R. Dyckhoff.
Contraction-free sequent calculi for intuitionistic logic.
Journal of Symbolic Logic, 57(3):795–807, 1992.

References II

A. Gill, J. Launchbury, and S.L. Peyton Jones.
A short cut to deforestation.
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.

P. Johann.
A generalization of short-cut fusion and its correctness proof.
Higher-Order and Symbolic Computation, 15(4):273–300, 2002.

P. Johann and J. Voigtländer.
Free theorems in the presence of seq.
In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

References III

P. Johann and J. Voigtländer.
A family of syntactic logical relations for the semantics of Haskell-like languages.
Information and Computation, 207(2):341–368, 2009.

J. Launchbury and R. Paterson.
Parametricity and unboxing with unpointed types.
In *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996.

J. Matthews and A.J. Ahmed.
Parametric polymorphism through run-time sealing or, theorems for low, low prices!
In *European Symposium on Programming, Proceedings*, volume 4960 of *LNCS*, pages 16–31. Springer-Verlag, 2008.

References IV

R.E. Møgelberg and A.K. Simpson.
Relational parametricity for computational effects.
In *Logic in Computer Science, Proceedings*, pages 346–355. IEEE Computer Society, 2007.

A. Pardo, J.P. Fernandes, and J. Saraiva.
Shortcut fusion rules for the derivation of circular and higher-order monadic programs.
In *Partial Evaluation and Program Manipulation, Proceedings*, pages 81–90. ACM Press, 2009.

A.M. Pitts.
Parametric polymorphism and operational equivalence.
Mathematical Structures in Computer Science, 10(3):321–359, 2000.

References V

- A.M. Pitts.
  Typed operational reasoning.
  In B.C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 245–289. MIT Press, 2005.
- F. Stenger and J. Voigtländer.
  Parametricity for Haskell with imprecise error semantics.
  In *Typed Lambda Calculi and Applications, Proceedings*, LNCS. Springer-Verlag, 2009.
- J. Svenningsson.
  Shortcut fusion for accumulating parameters & zip-like functions.
  In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM Press, 2002.

References VI

- P. Wadler.
  Theorems for free!
  In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
- P. Wadler and S. Blott.
  How to make ad-hoc polymorphism less ad hoc.
  In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press, 1989.
- G. Washburn and S. Weirich.
  Generalizing parametricity using information-flow.
  In *Logic in Computer Science, Proceedings*, pages 62–71. IEEE Computer Society, 2005.