Semantics of Advanced Data Types

Patricia Johann
Appalachian State University

June 16, 2021

Course Outline

Lecture 1: Syntax and semantics of ADTs and nested types $\checkmark$

Lecture 2: Syntax and semantics of GADTs $\checkmark$

$$\boxed{\text{ADTs}} \xhookrightarrow[\text{generalized by}]{\text{syntactically}} \boxed{\text{nested types}} \xhookrightarrow[\text{generalized by}]{\text{syntactically}} \boxed{\text{GADTs}}$$

Lecture 3: Parametricity for ADTs and nested types

Lecture 4: Parametricity for GADTs

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

Lecture 3:
Parametricity for ADTs and Nested Types

- Question: Can advanced data types be included in languages that otherwise support parametricity?
- Relational parametricity was introduced by Reynolds to model type uniformity, or representation independence, in functional languages.
- Reynolds developed parametricity for System F. It has now been developed for many extensions of System F.
- Parametricity formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations.
- It requires that every polymorphic program preserves all relations between pairs of types at which it is instantiated.
- Wadler popularized Reynolds' parametricity as "theorems for free" — "for free" because it can deduce properties of programs from just their types, with no knowledge whatsoever of the text of the programs involved.
- Wadler only considered lists (and, implicitly, other ADTs). And most of the free theorems he gives for them in his paper are actually consequences of naturality rather than parametricity.
- There was no reason to distinguish since, for ADTs, they coincide!
- We recently gave a parametric model for nested types. Again, there is no reason to distinguish between consequences of naturality and of parametricity more generally.

The Polymorphism Balance Sheet

+ Polymorphic functions can be instantiated to any types whatsoever, so they are very general.

- Polymorphic functions must “work for” (i.e., be instantiable to) all types, so they cannot perform type-specific operations. So, in another sense, polymorphic functions are not very general at all.

+++ But this means that we can tell a lot about polymorphic functions just by knowing their types...

... without knowing anything about their definitions!

The Polymorphism Balance Sheet

+ Polymorphic functions can be instantiated to any types whatsoever, so they are very general.

- Polymorphic functions must "work for" (i.e., be instantiable to) all types, so they cannot perform type-specific operations. So, in another sense, polymorphic functions are not very general at all.

+++ But this means that we can tell a lot about polymorphic functions just by knowing their types...

... without knowing anything about their definitions!

The Polymorphism Balance Sheet

+ Polymorphic functions can be instantiated to any types whatsoever, so they are very general.

- Polymorphic functions must "work for" (i.e., be instantiable to) all types, so they cannot perform type-specific operations. So, in another sense, polymorphic functions are not very general at all.

+++ But this means that we can tell a lot about polymorphic functions just by knowing their types...

... without knowing anything about their definitions!

The Polymorphism Balance Sheet

$+$ Polymorphic functions can be instantiated to any types whatsoever, so they are very general.

- Polymorphic functions must "work for" (i.e., be instantiable to) all types, so they cannot perform type-specific operations. So, in another sense, polymorphic functions are not very general at all.

$+++$ But this means that we can tell a lot about polymorphic functions just by knowing their types...

... without knowing anything about their definitions!

Parametricity Informally

- Consider

$$\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$$

- If filter is the “real” filter function on lists, then it satisfies

$$\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs}) \;=\; \mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$$

  for $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and $\mathsf{p} : \mathsf{B} \to \mathsf{Bool}$.
- But even without knowing if filter is the “real” filter function on lists, if it has the type given it will satisfy the same theorem!

Parametricity Informally

- Consider

$$\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$$

- If filter is the "real" filter function on lists, then it satisfies

$$\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs}) \;=\; \mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$$

  for $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and $\mathsf{p} : \mathsf{B} \to \mathsf{Bool}$.

- But even without knowing if filter is the "real" filter function on lists, if it has the type given it will satisfy the same theorem!

Parametricity Informally

- Consider
$$\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List\,A} \to \mathsf{List\,A}$$

- If filter is the "real" filter function on lists, then it satisfies
$$\mathsf{filter\,p\,(map\,f\,xs)} \;=\; \mathsf{map\,f\,(filter\,(p \circ f)\,xs)}$$
for $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and $\mathsf{p} : \mathsf{B} \to \mathsf{Bool}$.
- But even without knowing if filter is the "real" filter function on lists, if it has the type given it will satisfy the same theorem!

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List\,A} \to \mathsf{List\,A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists map f xs and xs always have the same length.
- Applying p to an element of map f xs always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- filter p always chooses "the same" elements from map f xs for output as filter $(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map\,f\,(filter\,(p \circ f)\,xs)}$ must be equal to $\mathsf{filter\,p\,(map\,f\,xs)}$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ and xs always have the same length.
- Applying p to an element of $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- $\mathsf{filter}\,\mathsf{p}$ always chooses "the same" elements from $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ for output as $\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$ must be equal to $\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs})$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists map f xs and xs always have the same length.
- Applying p to an element of map f xs always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- filter p always chooses "the same" elements from map f xs for output as filter $(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So map f (filter $(\mathsf{p} \circ \mathsf{f})$ xs) must be equal to filter p (map f xs).
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists map f xs and xs always have the same length.
- Applying p to an element of map f xs always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- filter p always chooses "the same" elements from map f xs for output as filter $(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$ must be equal to $\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs})$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ and xs always have the same length.
- Applying p to an element of $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- $\mathsf{filter}\,\mathsf{p}$ always chooses "the same" elements from $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ for output as $\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$ must be equal to $\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs})$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists map f xs and xs always have the same length.
- Applying p to an element of map f xs always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- filter p always chooses "the same" elements from map f xs for output as filter $(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$ must be equal to $\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs})$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists map f xs and xs always have the same length.
- Applying p to an element of map f xs always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- filter p always chooses "the same" elements from map f xs for output as filter $(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So map f (filter $(\mathsf{p} \circ \mathsf{f})$ xs) must be equal to filter p (map f xs).
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ and xs always have the same length.
- Applying p to an element of $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- $\mathsf{filter}\,\mathsf{p}$ always chooses "the same" elements from $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ for output as $\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$ must be equal to $\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs})$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists map f xs and xs always have the same length.
- Applying p to an element of map f xs always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- filter p always chooses "the same" elements from map f xs for output as filter $(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$ must be equal to $\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs})$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Free Theorem for filter's Type, Informally

- $\mathsf{filter} : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$ must work uniformly on all instantiations of A.
- The output list can only contain elements from the input list xs.
- Which elements, in which order, and with which multiplicity can only be decided based on xs and the input predicate p.
- This can be decided based only on the length of xs and on the results of applying p to the elements of xs.
- The lists $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ and xs always have the same length.
- Applying p to an element of $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ always has the same outcome as applying $\mathsf{p} \circ \mathsf{f}$ to the corresponding element of xs.
- $\mathsf{filter}\,\mathsf{p}$ always chooses "the same" elements from $\mathsf{map}\,\mathsf{f}\,\mathsf{xs}$ for output as $\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})$ chooses from xs, except that f must still be applied to each of them to get the same results.
- So $\mathsf{map}\,\mathsf{f}\,(\mathsf{filter}\,(\mathsf{p} \circ \mathsf{f})\,\mathsf{xs})$ must be equal to $\mathsf{filter}\,\mathsf{p}\,(\mathsf{map}\,\mathsf{f}\,\mathsf{xs})$.
- Note that this free theorem does not just follow from the fact that List can be interpreted as the fixpoint of a higher-order functor.
- That is, this free theorem is not a consequence of naturality.

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathrm{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathrm{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathrm{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathrm{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathrm{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\, B : \mathrm{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathrm{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathrm{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.

- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathsf{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathsf{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathrm{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathrm{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.

- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{A}$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B} \to \mathsf{B}$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathsf{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathsf{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathrm{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathrm{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{A : \mathsf{Set}\} \to A \to A$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{A\,B : \mathsf{Set}\} \to (A \to B \to B) \to B \to B$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Some Questions

- How can we formalize this intuition? (By constructing parametric models for calculi in which ADTs have functorial semantics.)
- What other kinds of free theorems do such parametric models give?

  - Type inhabitation results — e.g., in System F, the only inhabitants of the type $\forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{A}$ are the polymorphic identity function and bottom.

  - Enforcement of abstraction barriers — e.g., ensuring that classes really are abstract, in the sense that a client cannot distinguish different implementations of an interface.

  - Enforcement of program invariants — e.g., invariants ensuring privacy, security, correct compilation, ...

  - Proving correctness of program transformations — e.g., the free theorem for the type $\forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B} \to \mathsf{B}$ gives a transformation that can turn the (standard) quadratic reverse function into a linear one.
- Can we construct a parametric model in which nested types have functorial semantics? (Yes. See "Parametricity for Primitive Nested Types".)
- Can we construct a parametric model in which GADTs have functorial semantics? (No, at least not a traditional parametric semantics. Stay tuned for Lecture 4.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type T[A] with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type T[A] also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term t(A, x) : T[A] with one free term variable x : F[A] is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of T[A] and t(A, x) in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0\, A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type T[A] with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type T[A] also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term t(A, x) : T[A] with one free term variable x : F[A] is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of T[A] and t(A, x) in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0\, A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type T[A] with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type T[A] also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term t(A, x) : T[A] with one free term variable x : F[A] is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of T[A] and t(A, x) in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0\, A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type T[A] with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type T[A] also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term t(A, x) : T[A] with one free term variable x : F[A] is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of T[A] and t(A, x) in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0 A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type T[A] with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type T[A] also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term $\mathsf{t(A,x) : T[A]}$ with one free term variable $\mathsf{x : F[A]}$ is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of T[A] and $\mathsf{t(A,x)}$ in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0 A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type T[A] with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type T[A] also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term t(A, x) : T[A] with one free term variable x : F[A] is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of T[A] and t(A, x) in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0 A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type T[A] with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type T[A] also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term t(A, x) : T[A] with one free term variable x : F[A] is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of T[A] and t(A, x) in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0\, A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

Formalizing Parametricity

- To formalize parametricity, we give two interpretations for the calculus of interest. In particular, we give two interpretations for every type in the calculus — including any ADTs, nested types, and GADTs it supports.
- Every type $\mathsf{T}[\mathsf{A}]$ with one free type variable A is given a set interpretation T_0 taking sets to sets as we have already discussed.
- Every such type $\mathsf{T}[\mathsf{A}]$ also has a relational interpretation T_1 taking relations $R : Rel(A, B)$ to relations $T_1\, R : Rel(T_0\, A, T_0\, B)$.
- Each term $\mathsf{t}(\mathsf{A}, \mathsf{x}) : \mathsf{T}[\mathsf{A}]$ with one free term variable $\mathsf{x} : \mathsf{F}[\mathsf{A}]$ is given a set interpretation as a map t_0 associating to each set A a morphism $t_0\, A : F_0\, A \to T_0\, A$ in Set.
- These interpretations are given inductively on the structures of $\mathsf{T}[\mathsf{A}]$ and $\mathsf{t}(\mathsf{A}, \mathsf{x})$ in such way that they imply two fundamental theorems:

  - An Identity Extension Lemma, which states that $T_1\, Eq_A = Eq_{T_0\, A}$.

  - An Abstraction Theorem, which states that, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism of relations from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.
- Similar theorems are required for types and terms with any number of free type and term variables. (In particular, if t is closed, then $t_0\, A : T_0\, A$.)

The Category Rel

- A relation (A, B, R), or $R : Rel(A, B)$, is given by
  - $A : Set$ (domain) and $B : Set$ (codomain)
  - $R \subseteq A \times B$ (so R relates a and b if $(a, b) \in R$)
- A morphism of relations $(f, g) : (A_1, B_1, R) \to (A_2, B_2, S)$ is given by:
  - $f : A_1 \to A_2$
  - $g : B_1 \to B_2$

  such that if $(a, b) \in R$ then $(f\, a,\ g\, b) \in S$.
- The identity morphism on (A, B, R) is (id_A, id_B).
- Composition of morphisms in Rel is given by componentwise composition in Set.
- We can interpret the type constructors $\top$, $\bot$, sums, and products in Rel whenever we can interpret them in Set.
- We can also take fixpoints in Rel whenever we can take them in Set.

The Category Rel

- A relation (A, B, R), or $R : Rel(A, B)$, is given by
  - $A : Set$ (domain) and $B : Set$ (codomain)
  - $R \subseteq A \times B$ (so R relates a and b if $(a, b) \in R$)
- A morphism of relations $(f, g) : (A_1, B_1, R) \to (A_2, B_2, S)$ is given by:
  - $f : A_1 \to A_2$
  - $g : B_1 \to B_2$

  such that if $(a, b) \in R$ then $(f\,a,\ g\,b) \in S$.
- The identity morphism on (A, B, R) is (id_A, id_B).
- Composition of morphisms in Rel is given by componentwise composition in Set.
- We can interpret the type constructors $\top$, $\bot$, sums, and products in Rel whenever we can interpret them in Set.
- We can also take fixpoints in Rel whenever we can take them in Set.

The Category Rel

- A relation (A, B, R), or $R : Rel(A, B)$, is given by
  - $A : Set$ (domain) and $B : Set$ (codomain)
  - $R \subseteq A \times B$ (so R relates a and b if $(a, b) \in R$)
- A morphism of relations $(f, g) : (A_1, B_1, R) \to (A_2, B_2, S)$ is given by:
  - $f : A_1 \to A_2$
  - $g : B_1 \to B_2$

  such that if $(a, b) \in R$ then $(f\,a,\ g\,b) \in S$.
- The identity morphism on (A, B, R) is (id_A, id_B).
- Composition of morphisms in Rel is given by componentwise composition in Set.
- We can interpret the type constructors $\top$, $\bot$, sums, and products in Rel whenever we can interpret them in Set.
- We can also take fixpoints in Rel whenever we can take them in Set.

The Category Rel

- A relation (A, B, R), or $R : Rel(A, B)$, is given by
  - $A : Set$ (domain) and $B : Set$ (codomain)
  - $R \subseteq A \times B$ (so R relates a and b if $(a, b) \in R$)
- A morphism of relations $(f, g) : (A_1, B_1, R) \to (A_2, B_2, S)$ is given by:
  - $f : A_1 \to A_2$
  - $g : B_1 \to B_2$

  such that if $(a, b) \in R$ then $(f\, a,\ g\, b) \in S$.
- The identity morphism on (A, B, R) is (id_A, id_B).
- Composition of morphisms in Rel is given by componentwise composition in Set.
- We can interpret the type constructors $\top$, $\bot$, sums, and products in Rel whenever we can interpret them in Set.
- We can also take fixpoints in Rel whenever we can take them in Set.

The Category Rel

- A relation (A, B, R), or $R : Rel(A, B)$, is given by
  - $A : Set$ (domain) and $B : Set$ (codomain)
  - $R \subseteq A \times B$ (so R relates a and b if $(a, b) \in R$)
- A morphism of relations $(f, g) : (A_1, B_1, R) \to (A_2, B_2, S)$ is given by:
  - $f : A_1 \to A_2$
  - $g : B_1 \to B_2$

  such that if $(a, b) \in R$ then $(f\, a,\ g\, b) \in S$.
- The identity morphism on (A, B, R) is (id_A, id_B).
- Composition of morphisms in Rel is given by componentwise composition in Set.
- We can interpret the type constructors $\top$, $\bot$, sums, and products in Rel whenever we can interpret them in Set.
- We can also take fixpoints in Rel whenever we can take them in Set.

Interpreting Type Constructors

- The interpretations of $\bot$, $\top$, sums, and products all preserve relatedness, as does the fixpoint operator.
- The initial object of Rel is $(\emptyset, \emptyset, \emptyset)$, where the third component is the empty relation on the empty set.
- The terminal object of Rel is $(1, 1, 1 \times 1)$, where 1 is the terminal object of Set, i.e., is "the" singleton set.
- The sum $(A_1, B_1, R_1) + (A_2, B_2, R_2)$ is $(A_1 + A_2, B_1 + B_2, R)$, where

$$R = \{(inl\, a_1, inl\, b_1) \mid (a_1, b_1) \in R_1\} \cup \{(inr\, a_2, inr\, b_2) \mid (a_2, b_2) \in R_2\}$$

- The product $(A_1, B_1, R_1) \times (A_2, B_2, R_2)$ is $(A_1 \times A_2, B_1 \times B_2, R)$, where

$$R = \{((a_1, a_2), (b_1, b_2)) \mid (a_1, b_1) \in R_1 \text{ and } (a_2, b_2) \in R_2\}$$

- The relational interpretation of each operation type constructor transforms its argument relations $\overline{R : Rel(A, B)}$ into a relation that relates elements obtained by applying the set interpretation of the operation to elements in $\overline{A}$ to elements obtained by applying the set interpretation of the operation to $\overline{B}$.

Interpreting Type Constructors

- The interpretations of $\bot$, $\top$, sums, and products all preserve relatedness, as does the fixpoint operator.
- The initial object of Rel is $(\emptyset, \emptyset, \emptyset)$, where the third component is the empty relation on the empty set.
- The terminal object of Rel is $(1, 1, 1 \times 1)$, where 1 is the terminal object of Set, i.e., is "the" singleton set.
- The sum $(A_1, B_1, R_1) + (A_2, B_2, R_2)$ is $(A_1 + A_2, B_1 + B_2, R)$, where

$$R = \{(inl\, a_1, inl\, b_1) \mid (a_1, b_1) \in R_1\} \cup \{(inr\, a_2, inr\, b_2) \mid (a_2, b_2) \in R_2\}$$

- The product $(A_1, B_1, R_1) \times (A_2, B_2, R_2)$ is $(A_1 \times A_2, B_1 \times B_2, R)$, where

$$R = \{((a_1, a_2), (b_1, b_2)) \mid (a_1, b_1) \in R_1 \text{ and } (a_2, b_2) \in R_2\}$$

- The relational interpretation of each operation type constructor transforms its argument relations $\overline{R : Rel(A, B)}$ into a relation that relates elements obtained by applying the set interpretation of the operation to elements in $\overline{A}$ to elements obtained by applying the set interpretation of the operation to $\overline{B}$.

Interpreting Type Constructors

- The interpretations of $\bot$, $\top$, sums, and products all preserve relatedness, as does the fixpoint operator.
- The initial object of Rel is $(\emptyset, \emptyset, \emptyset)$, where the third component is the empty relation on the empty set.
- The terminal object of Rel is $(1, 1, 1 \times 1)$, where 1 is the terminal object of Set, i.e., is “the” singleton set.
- The sum $(A_1, B_1, R_1) + (A_2, B_2, R_2)$ is $(A_1 + A_2, B_1 + B_2, R)$, where

$$R = \{(inl\, a_1, inl\, b_1) \mid (a_1, b_1) \in R_1\} \cup \{(inr\, a_2, inr\, b_2) \mid (a_2, b_2) \in R_2\}$$

- The product $(A_1, B_1, R_1) \times (A_2, B_2, R_2)$ is $(A_1 \times A_2, B_1 \times B_2, R)$, where

$$R = \{((a_1, a_2), (b_1, b_2)) \mid (a_1, b_1) \in R_1 \text{ and } (a_2, b_2) \in R_2\}$$

- The relational interpretation of each operation type constructor transforms its argument relations $\overline{R : Rel(A, B)}$ into a relation that relates elements obtained by applying the set interpretation of the operation to elements in $\overline{A}$ to elements obtained by applying the set interpretation of the operation to $\overline{B}$.

Interpreting Type Constructors

- The interpretations of $\bot$, $\top$, sums, and products all preserve relatedness, as does the fixpoint operator.
- The initial object of Rel is $(\emptyset, \emptyset, \emptyset)$, where the third component is the empty relation on the empty set.
- The terminal object of Rel is $(1, 1, 1 \times 1)$, where 1 is the terminal object of Set, i.e., is "the" singleton set.
- The sum $(A_1, B_1, R_1) + (A_2, B_2, R_2)$ is $(A_1 + A_2, B_1 + B_2, R)$, where

$$R = \{(inl\, a_1, inl\, b_1) \mid (a_1, b_1) \in R_1\} \cup \{(inr\, a_2, inr\, b_2) \mid (a_2, b_2) \in R_2\}$$

- The product $(A_1, B_1, R_1) \times (A_2, B_2, R_2)$ is $(A_1 \times A_2, B_1 \times B_2, R)$, where

$$R = \{((a_1, a_2), (b_1, b_2)) \mid (a_1, b_1) \in R_1 \text{ and } (a_2, b_2) \in R_2\}$$

- The relational interpretation of each operation type constructor transforms its argument relations $\overline{R : Rel(A, B)}$ into a relation that relates elements obtained by applying the set interpretation of the operation to elements in $\overline{A}$ to elements obtained by applying the set interpretation of the operation to $\overline{B}$.

Interpreting Type Constructors

- The interpretations of $\bot$, $\top$, sums, and products all preserve relatedness, as does the fixpoint operator.
- The initial object of Rel is $(\emptyset, \emptyset, \emptyset)$, where the third component is the empty relation on the empty set.
- The terminal object of Rel is $(1, 1, 1 \times 1)$, where 1 is the terminal object of Set, i.e., is "the" singleton set.
- The sum $(A_1, B_1, R_1) + (A_2, B_2, R_2)$ is $(A_1 + A_2, B_1 + B_2, R)$, where

$$R = \{(inl\, a_1, inl\, b_1) \mid (a_1, b_1) \in R_1\} \cup \{(inr\, a_2, inr\, b_2) \mid (a_2, b_2) \in R_2\}$$

- The product $(A_1, B_1, R_1) \times (A_2, B_2, R_2)$ is $(A_1 \times A_2, B_1 \times B_2, R)$, where

$$R = \{((a_1, a_2), (b_1, b_2)) \mid (a_1, b_1) \in R_1 \text{ and } (a_2, b_2) \in R_2\}$$

- The relational interpretation of each operation type constructor transforms its argument relations $\overline{R : Rel(A, B)}$ into a relation that relates elements obtained by applying the set interpretation of the operation to elements in $\overline{A}$ to elements obtained by applying the set interpretation of the operation to $\overline{B}$.

Interpreting Type Constructors

- The interpretations of $\perp$, $\top$, sums, and products all preserve relatedness, as does the fixpoint operator.
- The initial object of Rel is $(\emptyset, \emptyset, \emptyset)$, where the third component is the empty relation on the empty set.
- The terminal object of Rel is $(1, 1, 1 \times 1)$, where 1 is the terminal object of Set, i.e., is "the" singleton set.
- The sum $(A_1, B_1, R_1) + (A_2, B_2, R_2)$ is $(A_1 + A_2, B_1 + B_2, R)$, where

$$R = \{(inl\, a_1, inl\, b_1) \mid (a_1, b_1) \in R_1\} \cup \{(inr\, a_2, inr\, b_2) \mid (a_2, b_2) \in R_2\}$$

- The product $(A_1, B_1, R_1) \times (A_2, B_2, R_2)$ is $(A_1 \times A_2, B_1 \times B_2, R)$, where

$$R = \{((a_1, a_2), (b_1, b_2)) \mid (a_1, b_1) \in R_1 \text{ and } (a_2, b_2) \in R_2\}$$

- The relational interpretation of each operation type constructor transforms its argument relations $\overline{R : Rel(A, B)}$ into a relation that relates elements obtained by applying the set interpretation of the operation to elements in $\overline{A}$ to elements obtained by applying the set interpretation of the operation to $\overline{B}$.

Relation Transformers

- A k-ary relation transformer is a triple $F = (F^1, F^2, F^*)$ where

  - F^1 and F^2 are ω-cocontinuous functors from Set^k to Set and F^* is a ω-cocontinuous functor from Rel^k to Rel.

  - if $R_1 : Rel(A_1, B_1), ..., R_k : Rel(A_k, B_k)$ then $F^*\overline{R} : Rel(F^1\overline{A}, F^2\overline{B})$.

  - if $(\alpha_1, \beta_1) : R_1 \to S_1, ..., (\alpha_k, \beta_k) : R_k \to S_k$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$.

- Define $F\overline{R} = F^*\overline{R}$ and $F\overline{(\alpha, \beta)} = F^*\overline{(\alpha, \beta)}$.

- The category RT_k of k-ary relation transformers is given by

  - objects: k-ary relation transformers

  - morphisms: $\delta : (G^1, G^2, G^*) \to (F^1, F^2, F^*)$ in RT_k is a pair of natural transformations $(\delta^1 : G^1 \to F^1, \delta^2 : G^2 \to F^2)$ such that for all $\overline{R : Rel(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}} x, \delta^2_{\overline{B}} y) \in F^*\overline{R}$.

  - identities and composition are inherited from the category of functors on Set.

Relation Transformers

- A k-ary relation transformer is a triple $F = (F^1, F^2, F^*)$ where

  - F^1 and F^2 are ω-cocontinuous functors from Set^k to Set and F^* is a ω-cocontinuous functor from Rel^k to Rel.

  - if $R_1 : Rel(A_1, B_1), ..., R_k : Rel(A_k, B_k)$ then $F^*\overline{R} : Rel(F^1\overline{A}, F^2\overline{B})$.

  - if $(\alpha_1, \beta_1) : R_1 \to S_1, ..., (\alpha_k, \beta_k) : R_k \to S_k$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$.

- Define $F\overline{R} = F^*\overline{R}$ and $F\overline{(\alpha, \beta)} = F^*\overline{(\alpha, \beta)}$.

- The category RT_k of k-ary relation transformers is given by

  - objects: k-ary relation transformers

  - morphisms: $\delta : (G^1, G^2, G^*) \to (F^1, F^2, F^*)$ in RT_k is a pair of natural transformations $(\delta^1 : G^1 \to F^1, \delta^2 : G^2 \to F^2)$ such that for all $\overline{R : Rel(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}} x, \delta^2_{\overline{B}} y) \in F^*\overline{R}$.

  - identities and composition are inherited from the category of functors on Set.

Relation Transformers

- A k-ary relation transformer is a triple $F = (F^1, F^2, F^*)$ where

  - F^1 and F^2 are ω-cocontinuous functors from Set^k to Set and F^* is a ω-cocontinuous functor from Rel^k to Rel.

  - if $R_1 : Rel(A_1, B_1), ..., R_k : Rel(A_k, B_k)$ then $F^*\overline{R} : Rel(F^1\overline{A}, F^2\overline{B})$.

  - if $(\alpha_1, \beta_1) : R_1 \to S_1, ..., (\alpha_k, \beta_k) : R_k \to S_k$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$.

- Define $F\overline{R} = F^*\overline{R}$ and $F\overline{(\alpha, \beta)} = F^*\overline{(\alpha, \beta)}$.

- The category RT_k of k-ary relation transformers is given by

  - objects: k-ary relation transformers

  - morphisms: $\delta : (G^1, G^2, G^*) \to (F^1, F^2, F^*)$ in RT_k is a pair of natural transformations $(\delta^1 : G^1 \to F^1, \delta^2 : G^2 \to F^2)$ such that for all $\overline{R : Rel(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}} x, \delta^2_{\overline{B}} y) \in F^*\overline{R}$.

  - identities and composition are inherited from the category of functors on Set.

Relation Transformers

- A k-ary relation transformer is a triple $F = (F^1, F^2, F^*)$ where

  - F^1 and F^2 are ω-cocontinuous functors from Set^k to Set and F^* is a ω-cocontinuous functor from Rel^k to Rel.

  - if $R_1 : Rel(A_1, B_1), ..., R_k : Rel(A_k, B_k)$ then $F^*\overline{R} : Rel(F^1\overline{A}, F^2\overline{B})$.

  - if $(\alpha_1, \beta_1) : R_1 \to S_1, ..., (\alpha_k, \beta_k) : R_k \to S_k$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$.

- Define $F\overline{R} = F^*\overline{R}$ and $F\overline{(\alpha, \beta)} = F^*\overline{(\alpha, \beta)}$.

- The category RT_k of k-ary relation transformers is given by

  - objects: k-ary relation transformers

  - morphisms: $\delta : (G^1, G^2, G^*) \to (F^1, F^2, F^*)$ in RT_k is a pair of natural transformations $(\delta^1 : G^1 \to F^1, \delta^2 : G^2 \to F^2)$ such that for all $\overline{R : Rel(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}} x, \delta^2_{\overline{B}} y) \in F^*\overline{R}$.

  - identities and composition are inherited from the category of functors on Set.

Relation Transformers

- A k-ary relation transformer is a triple $F = (F^1, F^2, F^*)$ where

  - F^1 and F^2 are ω-cocontinuous functors from Set^k to Set and F^* is a ω-cocontinuous functor from Rel^k to Rel.

  - if $R_1 : Rel(A_1, B_1), ..., R_k : Rel(A_k, B_k)$ then $F^*\overline{R} : Rel(F^1\overline{A}, F^2\overline{B})$.

  - if $(\alpha_1, \beta_1) : R_1 \to S_1, ..., (\alpha_k, \beta_k) : R_k \to S_k$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$.

- Define $F\overline{R} = F^*\overline{R}$ and $F\overline{(\alpha, \beta)} = F^*\overline{(\alpha, \beta)}$.

- The category RT_k of k-ary relation transformers is given by

  - objects: k-ary relation transformers

  - morphisms: $\delta : (G^1, G^2, G^*) \to (F^1, F^2, F^*)$ in RT_k is a pair of natural transformations $(\delta^1 : G^1 \to F^1, \delta^2 : G^2 \to F^2)$ such that for all $\overline{R : Rel(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}} x, \delta^2_{\overline{B}} y) \in F^*\overline{R}$.

  - identities and composition are inherited from the category of functors on Set.

Relation Transformers

- A k-ary relation transformer is a triple $F = (F^1, F^2, F^*)$ where

  - F^1 and F^2 are ω-cocontinuous functors from Set^k to Set and F^* is a ω-cocontinuous functor from Rel^k to Rel.

  - if $R_1 : Rel(A_1, B_1), ..., R_k : Rel(A_k, B_k)$ then $F^*\overline{R} : Rel(F^1\overline{A}, F^2\overline{B})$.

  - if $(\alpha_1, \beta_1) : R_1 \to S_1, ..., (\alpha_k, \beta_k) : R_k \to S_k$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$.

- Define $F\overline{R} = F^*\overline{R}$ and $F\overline{(\alpha, \beta)} = F^*\overline{(\alpha, \beta)}$.

- The category RT_k of k-ary relation transformers is given by

  - objects: k-ary relation transformers

  - morphisms: $\delta : (G^1, G^2, G^*) \to (F^1, F^2, F^*)$ in RT_k is a pair of natural transformations $(\delta^1 : G^1 \to F^1, \delta^2 : G^2 \to F^2)$ such that for all $\overline{R : Rel(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}}x, \delta^2_{\overline{B}}y) \in F^*\overline{R}$.

  - identities and composition are inherited from the category of functors on Set.

Relation Transformers

- A k-ary relation transformer is a triple $F = (F^1, F^2, F^*)$ where

  - F^1 and F^2 are ω-cocontinuous functors from Set^k to Set and F^* is a ω-cocontinuous functor from Rel^k to Rel.

  - if $R_1 : Rel(A_1, B_1), ..., R_k : Rel(A_k, B_k)$ then $F^*\overline{R} : Rel(F^1\overline{A}, F^2\overline{B})$.

  - if $(\alpha_1, \beta_1) : R_1 \to S_1, ..., (\alpha_k, \beta_k) : R_k \to S_k$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$.

- Define $F\overline{R} = F^*\overline{R}$ and $F\overline{(\alpha, \beta)} = F^*\overline{(\alpha, \beta)}$.

- The category RT_k of k-ary relation transformers is given by

  - objects: k-ary relation transformers

  - morphisms: $\delta : (G^1, G^2, G^*) \to (F^1, F^2, F^*)$ in RT_k is a pair of natural transformations $(\delta^1 : G^1 \to F^1, \delta^2 : G^2 \to F^2)$ such that for all $\overline{R : Rel(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}} x, \delta^2_{\overline{B}} y) \in F^*\overline{R}$.

  - identities and composition are inherited from the category of functors on Set.

Functors on Categories of Relation Transformers

- A functor H on RT_k is a triple $H = (H^1, H^2, H^*)$ where
  - H^1 and H^2 are functors from $[Set^k, Set]$ to $[Set^k, Set]$.
  - H^* is a functor from RT_k to $[Rel^k, Rel]$.
  - For all $\overline{R : Rel(A, B)}$,

$$\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}} \text{ and } \pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$$

  - The action of H on objects is given by

$$H\,(F^1, F^2, F^*) = (H^1F^1,\ H^2F^2,\ H^*(F^1, F^2, F^*))$$

  - The action of H on morphisms is given by

$$H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$$

  - many coherence conditions hold.
- If $H = (H^1, H^2, H^*)$ is a functor on RT_k then

$$\mu H = \varinjlim_{n \in N}(H^n\,K_0) = (\mu H^1, \mu H^2, \varinjlim_{n \in N}(H^n\,K_0)^*)$$

so the fixpoint operator also has an intepretation as a relation transformer.

Functors on Categories of Relation Transformers

- A functor H on RT_k is a triple $H = (H^1, H^2, H^*)$ where
  - H^1 and H^2 are functors from $[Set^k, Set]$ to $[Set^k, Set]$.
  - H^* is a functor from RT_k to $[Rel^k, Rel]$.
  - For all $\overline{R : Rel(A, B)}$,

$$\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}} \text{ and } \pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$$

  - The action of H on objects is given by

$$H\,(F^1, F^2, F^*) = (H^1F^1,\ H^2F^2,\ H^*(F^1, F^2, F^*))$$

  - The action of H on morphisms is given by

$$H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$$

  - many coherence conditions hold.
- If $H = (H^1, H^2, H^*)$ is a functor on RT_k then

$$\mu H = \varinjlim_{n \in N}(H^n\,K_0) = (\mu H^1, \mu H^2, \varinjlim_{n \in N}(H^n\,K_0)^*)$$

so the fixpoint operator also has an intepretation as a relation transformer.

Functors on Categories of Relation Transformers

- A functor H on RT_k is a triple $H = (H^1, H^2, H^*)$ where
  - H^1 and H^2 are functors from $[Set^k, Set]$ to $[Set^k, Set]$.
  - H^* is a functor from RT_k to $[Rel^k, Rel]$.
  - For all $\overline{R : Rel(A, B)}$,

    $\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}}$ and $\pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$

  - The action of H on objects is given by

    $$H\,(F^1, F^2, F^*) = (H^1 F^1,\ H^2 F^2,\ H^*(F^1, F^2, F^*))$$

  - The action of H on morphisms is given by

    $$H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$$

  - many coherence conditions hold.
- If $H = (H^1, H^2, H^*)$ is a functor on RT_k then

  $$\mu H = \varinjlim_{n \in N}(H^n\, K_0) = (\mu H^1, \mu H^2, \varinjlim_{n \in N}(H^n\, K_0)^*)$$

  so the fixpoint operator also has an intepretation as a relation transformer.

Functors on Categories of Relation Transformers

- A functor H on RT_k is a triple $H = (H^1, H^2, H^*)$ where
  - H^1 and H^2 are functors from $[Set^k, Set]$ to $[Set^k, Set]$.
  - H^* is a functor from RT_k to $[Rel^k, Rel]$.
  - For all $\overline{R : Rel(A, B)}$,

$$\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}} \text{ and } \pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$$

  - The action of H on objects is given by

$$H\,(F^1, F^2, F^*) = (H^1F^1,\ H^2F^2,\ H^*(F^1, F^2, F^*))$$

  - The action of H on morphisms is given by

$$H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$$

  - many coherence conditions hold.
- If $H = (H^1, H^2, H^*)$ is a functor on RT_k then

$$\mu H = \varinjlim_{n \in N}(H^n\,K_0) = (\mu H^1, \mu H^2, \varinjlim_{n \in N}(H^n\,K_0)^*)$$

so the fixpoint operator also has an intepretation as a relation transformer.

Functors on Categories of Relation Transformers

- A functor H on RT_k is a triple $H = (H^1, H^2, H^*)$ where
  - H^1 and H^2 are functors from $[Set^k, Set]$ to $[Set^k, Set]$.
  - H^* is a functor from RT_k to $[Rel^k, Rel]$.
  - For all $\overline{R : Rel(A, B)}$,

$$\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}} \text{ and } \pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$$

  - The action of H on objects is given by

$$H\,(F^1, F^2, F^*) = (H^1F^1,\; H^2F^2,\; H^*(F^1, F^2, F^*))$$

  - The action of H on morphisms is given by

$$H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$$

  - many coherence conditions hold.
- If $H = (H^1, H^2, H^*)$ is a functor on RT_k then

$$\mu H = \varinjlim_{n \in N}(H^n\,K_0) = (\mu H^1, \mu H^2, \varinjlim_{n \in N}(H^n\,K_0)^*)$$

so the fixpoint operator also has an intepretation as a relation transformer.

Functors on Categories of Relation Transformers

- A functor H on RT_k is a triple $H = (H^1, H^2, H^*)$ where
  - H^1 and H^2 are functors from $[Set^k, Set]$ to $[Set^k, Set]$.
  - H^* is a functor from RT_k to $[Rel^k, Rel]$.
  - For all $\overline{R : Rel(A, B)}$,

$$\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}} \text{ and } \pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$$

  - The action of H on objects is given by

$$H\,(F^1, F^2, F^*) = (H^1 F^1,\ H^2 F^2,\ H^*(F^1, F^2, F^*))$$

  - The action of H on morphisms is given by

$$H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$$

  - many coherence conditions hold.
- If $H = (H^1, H^2, H^*)$ is a functor on RT_k then

$$\mu H = \varinjlim_{n \in N}(H^n\, K_0) = (\mu H^1, \mu H^2, \varinjlim_{n \in N}(H^n\, K_0)^*)$$

so the fixpoint operator also has an intepretation as a relation transformer.

Functors on Categories of Relation Transformers

- A functor H on RT_k is a triple $H = (H^1, H^2, H^*)$ where
  - H^1 and H^2 are functors from $[Set^k, Set]$ to $[Set^k, Set]$.
  - H^* is a functor from RT_k to $[Rel^k, Rel]$.
  - For all $\overline{R : Rel(A, B)}$,

$$\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}} \text{ and } \pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$$

  - The action of H on objects is given by

$$H\,(F^1, F^2, F^*) = (H^1F^1,\ H^2F^2,\ H^*(F^1, F^2, F^*))$$

  - The action of H on morphisms is given by

$$H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$$

  - many coherence conditions hold.
- If $H = (H^1, H^2, H^*)$ is a functor on RT_k then

$$\mu H = \varinjlim_{n \in N}(H^n\, K_0) = (\mu H^1, \mu H^2, \varinjlim_{n \in N}(H^n\, K_0)^*)$$

so the fixpoint operator also has an intepretation as a relation transformer.

Relational Interpretations of ADTs and Nested Types

- If D is a type constructor for an ADT or nested type, then the action of the relational interpretation D_1 of D on relations $\overline{R : Rel(A, B)}$ interpreting its free type variables is the relation $D_1\,\overline{R} : Rel(D_0\,\overline{A}, D_0\,\overline{B})$ where $d \in D_0\,\overline{A}$ and $d' \in D_0\,\overline{B}$ are related if

  - d and d' have the same shape

  and

  - every data element in d is related by R to the corresponding data element in d'

Examples

- The lists $[1, 2, 3, 4]$ and $[5, 6, 7, 8]$ are related by the relation $List_1\ P$ where

$$P = (\mathbb{N}, \mathbb{N}, \{(n, m) : \mathbb{N} \times \mathbb{N} \mid n \text{ and } m \text{ have the same parity}\})$$

- The binary trees

$$\text{node}\,(\text{leaf}\ 1)\ \text{false}\,(\text{node}\,(\text{leaf}\ 2)\ \text{true}\,(\text{leaf}\ 3))$$

and

$$\text{node}\,(\text{leaf}\ 7)\ \text{true}\,(\text{node}\,(\text{leaf}\ 8)\ \text{true}\,(\text{leaf}\ 9))$$

are related by the relation $Tree_1\ P\ \leq_{Bool}$

- The perfect trees

$$\text{pnode}\,(\text{pnode}\,(\text{pleaf}\ 1)\,(\text{pleaf}\ 2))\,(\text{pnode}\,(\text{pleaf}\ 3)\,(\text{pleaf}\ 4))$$

and

$$\text{pnode}\,(\text{pnode}\,(\text{pleaf}\ 5)\,(\text{pleaf}\ 6))\,(\text{pnode}\,(\text{pleaf}\ 7)\,(\text{pleaf}\ 8))$$

are related by the relation $PTree_1\ \leq_{\mathbb{N}}$.

Examples

- The lists $[1, 2, 3, 4]$ and $[5, 6, 7, 8]$ are related by the relation $List_1\ P$ where

$$P = (\mathbb{N}, \mathbb{N}, \{(n, m) : \mathbb{N} \times \mathbb{N} \mid n \text{ and } m \text{ have the same parity}\})$$

- The binary trees

$$\text{node (leaf 1) false (node (leaf 2) true (leaf 3))}$$

  and

$$\text{node (leaf 7) true (node (leaf 8) true (leaf 9))}$$

  are related by the relation $Tree_1\ P\ \leq_{Bool}$

- The perfect trees

$$\text{pnode (pnode (pleaf 1) (pleaf 2)) (pnode (pleaf 3) (pleaf 4))}$$

  and

$$\text{pnode (pnode (pleaf 5) (pleaf 6)) (pnode (pleaf 7) (pleaf 8))}$$

  are related by the relation $PTree_1\ \leq_{\mathbb{N}}$.

Examples

- The lists $[1, 2, 3, 4]$ and $[5, 6, 7, 8]$ are related by the relation $List_1\ P$ where

$$P = (\mathbb{N}, \mathbb{N}, \{(n, m) : \mathbb{N} \times \mathbb{N} \mid n \text{ and } m \text{ have the same parity}\})$$

- The binary trees

$$\mathsf{node}\,(\mathsf{leaf}\ 1)\ \mathsf{false}\,(\mathsf{node}\,(\mathsf{leaf}\ 2)\ \mathsf{true}\,(\mathsf{leaf}\ 3))$$

  and

$$\mathsf{node}\,(\mathsf{leaf}\ 7)\ \mathsf{true}\,(\mathsf{node}\,(\mathsf{leaf}\ 8)\ \mathsf{true}\,(\mathsf{leaf}\ 9))$$

  are related by the relation $Tree_1\ P\ \leq_{Bool}$

- The perfect trees

$$\mathsf{pnode}\,(\mathsf{pnode}\,(\mathsf{pleaf}\ 1)\,(\mathsf{pleaf}\ 2))\,(\mathsf{pnode}\,(\mathsf{pleaf}\ 3)\,(\mathsf{pleaf}\ 4))$$

  and

$$\mathsf{pnode}\,(\mathsf{pnode}\,(\mathsf{pleaf}\ 5)\,(\mathsf{pleaf}\ 6))\,(\mathsf{pnode}\,(\mathsf{pleaf}\ 7)\,(\mathsf{pleaf}\ 8))$$

  are related by the relation $PTree_1\ \leq_{\mathbb{N}}$.

A Parametric Model for ADTs and Nested Types

- We have set and relational interpretations for ADTs and nested types.
- We can also define a term calculus and its set or relational semantics in such a way that the resulting model is parametric, i.e., that the IEL and AT hold.

- The IEL says that if T[A] is a type, then $T_1\ Eq_A = Eq_{T_0\,A}$.
- The AT states that if t(A, x) :: G[A] is a term with one free term variable x :: F[A] then, for any $R : Rel(A, B)$, $(t_0\ A, t_0\ B)$ is a morphism from $(F_0\ A, F_0\ B, F_1\ R)$ to $(T_0\ A, T_0\ B, T_1\ R)$.

A Parametric Model for ADTs and Nested Types

- We have set and relational interpretations for ADTs and nested types.
- We can also define a term calculus and its set or relational semantics in such a way that the resulting model is parametric, i.e., that the IEL and AT hold.

- The IEL says that if T[A] is a type, then $T_1\, Eq_A = Eq_{T_0\, A}$.
- The AT states that if t(A, x) :: G[A] is a term with one free term variable x :: F[A] then, for any $R : Rel(A, B)$, $(t_0\, A, t_0\, B)$ is a morphism from $(F_0\, A, F_0\, B, F_1\, R)$ to $(T_0\, A, T_0\, B, T_1\, R)$.

A Parametric Model for ADTs and Nested Types

- We have set and relational interpretations for ADTs and nested types.
- We can also define a term calculus and its set or relational semantics in such a way that the resulting model is parametric, i.e., that the IEL and AT hold.

- The IEL says that if T[A] is a type, then $T_1\ Eq_A = Eq_{T_0 A}$.
- The AT states that if t(A, x) :: G[A] is a term with one free term variable x :: F[A] then, for any $R : Rel(A, B)$, $(t_0\ A, t_0\ B)$ is a morphism from $(F_0\ A, F_0\ B, F_1\ R)$ to $(T_0\ A, T_0\ B, T_1\ R)$.

A Free Theorem

- Consider $\mathsf{flatten} : \forall\,\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{PTree}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{flatten}_0\ A, \mathit{flatten}_0\ B) : PTree_1\ R \to List_1\ R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $\mathit{flatten}_0\ A\,p$ and $\mathit{flatten}_0\ B\,p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\,x)\}$ for $f : A \to B$ gives:

If $map_{PTree}\ f\,p = p'$ then $map_{List}\ f\,(\mathit{flatten}\ A\,p) = \mathit{flatten}\ B\,p'$.

- That is, $map_{List}\ f\,(\mathit{flatten}\ A\,p) = \mathit{flatten}\ B\,(map_{PTree}\ f\,p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,(\mathsf{flatten}\,\mathsf{p}) = \mathsf{flatten}\,(\mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,\mathsf{p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider $\mathsf{flatten} : \forall\,\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{PTree\,A} \to \mathsf{List\,A}$.
- By the AT, for any $R : \mathit{Rel}(A, B)$,

$$(\mathit{flatten}_0\,A, \mathit{flatten}_0\,B) : \mathit{PTree}_1\,R \to \mathit{List}_1\,R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $\mathit{flatten}_0\,A\,p$ and $\mathit{flatten}_0\,B\,p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\,x)\}$ for $f : A \to B$ gives:

If $\mathit{map}_{\mathit{PTree}}\,f\,p = p'$ then $\mathit{map}_{\mathit{List}}\,f\,(\mathit{flatten}\,A\,p) = \mathit{flatten}\,B\,p'$.

- That is, $\mathit{map}_{\mathit{List}}\,f\,(\mathit{flatten}\,A\,p) = \mathit{flatten}\,B\,(\mathit{map}_{\mathit{PTree}}\,f\,p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map_{List}\,f\,(flatten\,p) = flatten\,(map_{PTree}\,f\,p)}$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider flatten : $\forall$ {A : Set} $\rightarrow$ PTree A $\rightarrow$ List A.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{flatten}_0\, A, \mathit{flatten}_0\, B) : PTree_1\, R \rightarrow List_1\, R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $\mathit{flatten}_0\, A\, p$ and $\mathit{flatten}_0\, B\, p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\, x)\}$ for $f : A \rightarrow B$ gives:

If $map_{PTree}\, f\, p = p'$ then $map_{List}\, f\, (\mathit{flatten}\, A\, p) = \mathit{flatten}\, B\, p'$.

- That is, $map_{List}\, f\, (\mathit{flatten}\, A\, p) = \mathit{flatten}\, B\, (map_{PTree}\, f\, p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathrm{map}_{\mathrm{List}}\, \mathrm{f}\, (\mathrm{flatten}\, \mathrm{p}) = \mathrm{flatten}\, (\mathrm{map}_{\mathrm{PTree}}\, \mathrm{f}\, \mathrm{p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider flatten : ∀ {A : Set} → PTree A → List A.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{flatten}_0\, A, \mathit{flatten}_0\, B) : PTree_1\, R \to List_1\, R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $\mathit{flatten}_0\, A\, p$ and $\mathit{flatten}_0\, B\, p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\, x)\}$ for $f : A \to B$ gives:

If $map_{PTree}\, f\, p = p'$ then $map_{List}\, f\, (\mathit{flatten}\, A\, p) = \mathit{flatten}\, B\, p'$.

- That is, $map_{List}\, f\, (\mathit{flatten}\, A\, p) = \mathit{flatten}\, B\, (map_{PTree}\, f\, p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ (\mathsf{flatten}\ \mathsf{p}) = \mathsf{flatten}\ (\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f}\ \mathsf{p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider flatten : ∀ {A : Set} → PTree A → List A.
- By the AT, for any $R : Rel(A, B)$,

$$(flatten_0\, A, flatten_0\, B) : PTree_1\, R \to List_1\, R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $flatten_0\, A\, p$ and $flatten_0\, B\, p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\, x)\}$ for $f : A \to B$ gives:

If $map_{PTree}\, f\, p = p'$ then $map_{List}\, f\, (flatten\, A\, p) = flatten\, B\, p'$.

- That is, $map_{List}\, f\, (flatten\, A\, p) = flatten\, B\, (map_{PTree}\, f\, p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map}_{\mathsf{List}}\, \mathsf{f}\, (\mathsf{flatten}\, \mathsf{p}) = \mathsf{flatten}\, (\mathsf{map}_{\mathsf{PTree}}\, \mathsf{f}\, \mathsf{p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider flatten : $\forall$ {A : Set} $\rightarrow$ PTree A $\rightarrow$ List A.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{flatten}_0\, A, \mathit{flatten}_0\, B) : PTree_1\, R \rightarrow List_1\, R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $\mathit{flatten}_0\, A\, p$ and $\mathit{flatten}_0\, B\, p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\, x)\}$ for $f : A \rightarrow B$ gives:

If $map_{PTree}\, f\, p = p'$ then $map_{List}\, f\, (\mathit{flatten}\, A\, p) = \mathit{flatten}\, B\, p'$.

- That is, $map_{List}\, f\, (\mathit{flatten}\, A\, p) = \mathit{flatten}\, B\, (map_{PTree}\, f\, p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ (\mathsf{flatten}\ \mathsf{p}) = \mathsf{flatten}\ (\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f}\ \mathsf{p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider flatten : $\forall$ {A : Set} $\rightarrow$ PTree A $\rightarrow$ List A.
- By the AT, for any $R : Rel(A, B)$,

$$(flatten_0\ A, flatten_0\ B) : PTree_1\ R \rightarrow List_1\ R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $flatten_0\ A\, p$ and $flatten_0\ B\, p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\ x)\}$ for $f : A \rightarrow B$ gives:

If $map_{PTree}\ f\ p = p'$ then $map_{List}\ f\ (flatten\ A\, p) = flatten\ B\, p'$.

- That is, $map_{List}\ f\ (flatten\ A\, p) = flatten\ B\ (map_{PTree}\ f\ p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ (\mathsf{flatten}\ \mathsf{p}) = \mathsf{flatten}\ (\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f}\ \mathsf{p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider flatten : $\forall \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{PTree\,A} \to \mathsf{List\,A}$.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{flatten}_0\ A, \mathit{flatten}_0\ B) : PTree_1\ R \to List_1\ R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $\mathit{flatten}_0\ A\,p$ and $\mathit{flatten}_0\ B\,p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\ x)\}$ for $f : A \to B$ gives:

If $map_{PTree}\ f\ p = p'$ then $map_{List}\ f\ (\mathit{flatten}\ A\,p) = \mathit{flatten}\ B\,p'$.

- That is, $map_{List}\ f\ (\mathit{flatten}\ A\,p) = \mathit{flatten}\ B\ (map_{PTree}\ f\ p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ (\mathsf{flatten\,p}) = \mathsf{flatten}\ (\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f\ p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

A Free Theorem

- Consider flatten : $\forall\,\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{PTree}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$.
- By the AT, for any $R : \mathit{Rel}(A, B)$,

$$(\mathit{flatten}_0\; A, \mathit{flatten}_0\; B) : \mathit{PTree}_1\; R \to \mathit{List}_1\; R$$

- If p and p′ are perfect trees of the same shape, and if their p and p′ have R-related data in corresponding positions, then $\mathit{flatten}_0\; A\, p$ and $\mathit{flatten}_0\; B\, p'$ interpret lists with the same length and R-related data in corresponding positions.
- Taking $R = \{(x, f\,x)\}$ for $f : A \to B$ gives:

$$\text{If } \mathit{map}_{\mathit{PTree}}\; f\; p = p' \text{ then } \mathit{map}_{\mathit{List}}\; f\; (\mathit{flatten}\; A\, p) = \mathit{flatten}\; B\, p'.$$

- That is, $\mathit{map}_{\mathit{List}}\; f\; (\mathit{flatten}\; A\, p) = \mathit{flatten}\; B\; (\mathit{map}_{\mathit{PTree}}\; f\; p)$
- This is the semantic equivalent of our naturality property from Lecture 1.
- Reflecting back into syntax (and eliding the type arguments) gives

$$\mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,(\mathsf{flatten}\,\mathsf{p}) = \mathsf{flatten}\,(\mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,\mathsf{p})$$

- We can also get this result from naturality.
- The AT can prove other kinds of results — e.g., inhabitation results and short cut fusion and the existence of deep induction rules for ADTs and nested types — that are not merely syntactic reflections of naturality.

Free Theorem for filter's Type, Formally

- Consider filter : $\forall\{A : Set\} \to (A \to Bool) \to List\,A \to List\,A$.
- By the AT, for any $R : Rel(A, B)$,

$$(filter_0\,A, filter_0\,B) : (R \to Equal_{Bool}) \to List_1\,R \to List_1\,R$$

- Taking $R = \{(x, f\,x)\}$ for $f : A \to B$ gives:

  If (p, p') are such that $f\,y = y'$ implies $p\,y = p'\,y'$, and if $map\,f\,xs = xs'$, then $map\,f\,(filter\,p\,xs) \;=\; filter\,p'\,xs'$

- That is, if $p\,y = p'\,(f\,y)$ then $map\,f\,(filter\,p\,xs) \;=\; filter\,p'\,(map\,f\,xs)$
- That is, $map\,f\,(filter\,(p' \circ f)\,xs) \;=\; filter\,p'\,(map\,f\,xs)$
- This is the non-naturality free theorem we informally argued correct at the start of today's lecture.

Free Theorem for filter's Type, Formally

- Consider filter : ∀{A : Set} → (A → Bool) → List A → List A.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{filter}_0\, A, \mathit{filter}_0\, B) : (R \to Equal_{Bool}) \to List_1\, R \to List_1\, R$$

- Taking $R = \{(x, f\,x)\}$ for $f : A \to B$ gives:

  If (p, p') are such that $f\,y = y'$ implies $p\,y = p'\,y'$, and if $map\,f\,xs = xs'$, then
  $$map\,f\,(\mathit{filter}\,p\,xs)\ =\ \mathit{filter}\,p'\,xs'$$

- That is, if $p\,y = p'\,(f\,y)$ then $map\,f\,(\mathit{filter}\,p\,xs)\ =\ \mathit{filter}\,p'\,(map\,f\,xs)$
- That is, $map\,f\,(\mathit{filter}\,(p' \circ f)\,xs)\ =\ \mathit{filter}\,p'\,(map\,f\,xs)$
- This is the non-naturality free theorem we informally argued correct at the start of today's lecture.

Free Theorem for filter's Type, Formally

- Consider filter : $\forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{Bool}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A}$.
- By the AT, for any $R : Rel(A, B)$,

$$(filter_0\; A, filter_0\; B) : (R \to Equal_{Bool}) \to List_1\; R \to List_1\; R$$

- Taking $R = \{(x, f\,x)\}$ for $f : A \to B$ gives:

  If (p, p') are such that $f\,y = y'$ implies $p\,y = p'\,y'$, and if $map\,f\,xs = xs'$, then $map\,f\,(filter\,p\,xs) \;=\; filter\,p'\,xs'$

- That is, if $p\,y = p'\,(f\,y)$ then $map\,f\,(filter\,p\,xs) \;=\; filter\,p'\,(map\,f\,xs)$
- That is, $map\,f\,(filter\,(p' \circ f)\,xs) \;=\; filter\,p'\,(map\,f\,xs)$
- This is the non-naturality free theorem we informally argued correct at the start of today's lecture.

Free Theorem for filter's Type, Formally

- Consider filter : ∀{A : Set} → (A → Bool) → List A → List A.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{filter}_0\, A, \mathit{filter}_0\, B) : (R \to \mathit{Equal}_{Bool}) \to \mathit{List}_1\, R \to \mathit{List}_1\, R$$

- Taking $R = \{(x, f\, x)\}$ for $f : A \to B$ gives:

  If (p, p') are such that $f\, y = y'$ implies $p\, y = p'\, y'$, and if $map\, f\, xs = xs'$, then
  $$map\, f\, (\mathit{filter}\, p\, xs) = \mathit{filter}\, p'\, xs'$$

- That is, if $p\, y = p'\, (f\, y)$ then $map\, f\, (\mathit{filter}\, p\, xs) = \mathit{filter}\, p'\, (map\, f\, xs)$
- That is, $map\, f\, (\mathit{filter}\, (p' \circ f)\, xs) = \mathit{filter}\, p'\, (map\, f\, xs)$
- This is the non-naturality free theorem we informally argued correct at the start of today's lecture.

Free Theorem for filter's Type, Formally

- Consider filter : ∀{A : Set} → (A → Bool) → List A → List A.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{filter}_0\, A, \mathit{filter}_0\, B) : (R \to \mathit{Equal}_{\mathit{Bool}}) \to \mathit{List}_1\, R \to \mathit{List}_1\, R$$

- Taking $R = \{(x, f\, x)\}$ for $f : A \to B$ gives:

  If (p, p') are such that $f\, y = y'$ implies $p\, y = p'\, y'$, and if $\mathit{map}\, f\, \mathit{xs} = \mathit{xs}'$, then
  $$\mathit{map}\, f\, (\mathit{filter}\, p\, \mathit{xs}) \; = \; \mathit{filter}\, p'\, \mathit{xs}'$$

- That is, if $p\, y = p'\, (f\, y)$ then $\mathit{map}\, f\, (\mathit{filter}\, p\, \mathit{xs}) \; = \; \mathit{filter}\, p'\, (\mathit{map}\, f\, \mathit{xs})$
- That is, $\mathit{map}\, f\, (\mathit{filter}\, (p' \circ f)\, \mathit{xs}) \; = \; \mathit{filter}\, p'\, (\mathit{map}\, f\, \mathit{xs})$
- This is the non-naturality free theorem we informally argued correct at the start of today's lecture.

Free Theorem for filter's Type, Formally

- Consider filter : ∀{A : Set} → (A → Bool) → List A → List A.
- By the AT, for any $R : Rel(A, B)$,

$$(\mathit{filter}_0\, A, \mathit{filter}_0\, B) : (R \to \mathit{Equal}_{\mathit{Bool}}) \to \mathit{List}_1\, R \to \mathit{List}_1\, R$$

- Taking $R = \{(x, f\, x)\}$ for $f : A \to B$ gives:

  If (p, p') are such that $f\, y = y'$ implies $p\, y = p'\, y'$, and if $\mathit{map}\, f\, \mathit{xs} = \mathit{xs}'$, then
  $$\mathit{map}\, f\, (\mathit{filter}\, p\, \mathit{xs}) \;=\; \mathit{filter}\, p'\, \mathit{xs}'$$

- That is, if $p\, y = p'\, (f\, y)$ then $\mathit{map}\, f\, (\mathit{filter}\, p\, \mathit{xs}) \;=\; \mathit{filter}\, p'\, (\mathit{map}\, f\, \mathit{xs})$
- That is, $\mathit{map}\, f\, (\mathit{filter}\, (p' \circ f)\, \mathit{xs}) \;=\; \mathit{filter}\, p'\, (\mathit{map}\, f\, \mathit{xs})$
- This is the non-naturality free theorem we informally argued correct at the start of today's lecture.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall\,\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{B} & & \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{Nil} & = & \mathsf{n} \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{x} :: \mathsf{xs}) & = & \mathsf{c}\,\mathsf{x}\,(\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{xs}) \end{array}$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{g}\,\mathsf{Nil}\,(::)) \;=\; \mathsf{g}\,\mathsf{n}\,\mathsf{c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\,n\,c\,xs = r\} \in Rel(List\,T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\ ((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys) \end{array}$$

- So

$$(g_{List\,T}\;Nil\,(::),\; g_{T'}\;n\,c) \in R$$

i.e.,

$$fold\,n\,c\,(g_{List\,T}\;Nil\,(::)) = g_{T'}\;n\,c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall\,\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{B} & & \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{Nil} & = & \mathsf{n} \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{x} :: \mathsf{xs}) & = & \mathsf{c}\,\mathsf{x}\,(\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{xs}) \end{array}$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{g}\,\mathsf{Nil}\,(::)) \;=\; \mathsf{g}\,\mathsf{n}\,\mathsf{c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\,n\,c\,xs = r\} \in Rel(List\,T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\ ((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys) \end{array}$$

- So

$$(g_{List\,T}\,Nil\,(::),\, g_{T'}\,n\,c) \in R$$

i.e.,

$$fold\,n\,c\,(g_{List\,T}\,Nil\,(::)) = g_{T'}\,n\,c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall\, \{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{B} & & \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{Nil} & = & \mathsf{n} \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{x} :: \mathsf{xs}) & = & \mathsf{c}\,\mathsf{x}\,(\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{xs}) \end{array}$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{g}\,\mathsf{Nil}\,(::)) \;=\; \mathsf{g}\,\mathsf{n}\,\mathsf{c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\,n\,c\,xs = r\} \in Rel(List\,T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\ ((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys) \end{array}$$

- So

$$(g_{List\,T}\;Nil\,(::),\; g_{T'}\;n\,c) \in R$$

i.e.,

$$fold\,n\,c\,(g_{List\,T}\;Nil\,(::)) = g_{T'}\;n\,c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall\,\{\mathsf{A}\;\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\;\mathsf{A} \to \mathsf{B} & & \\ \mathsf{fold}\;\mathsf{n}\;\mathsf{c}\;\mathsf{Nil} & = & \mathsf{n} \\ \mathsf{fold}\;\mathsf{n}\;\mathsf{c}\;(\mathsf{x} :: \mathsf{xs}) & = & \mathsf{c}\;\mathsf{x}\;(\mathsf{fold}\;\mathsf{n}\;\mathsf{c}\;\mathsf{xs}) \end{array}$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A}\;\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold}\;\mathsf{n}\;\mathsf{c}\;(\mathsf{g}\;\mathsf{Nil}\;(::)) \;=\; \mathsf{g}\;\mathsf{n}\;\mathsf{c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\;n\;c\;xs = r\} \in Rel(List\;T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\;n\;c\;Nil = n \\ ((::), c) & \in & R \text{ since } fold\;n\;c\;(y :: ys) = c\;y\;(fold\;n\;c\;ys) \end{array}$$

- So

$$(g_{List\;T}\;Nil\;(::),\; g_{T'}\;n\;c) \in R$$

  i.e.,

$$fold\;n\;c\;(g_{List\;T}\;Nil\;(::)) = g_{T'}\;n\;c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall \{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{B} & & \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{Nil} & = & \mathsf{n} \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{x} :: \mathsf{xs}) & = & \mathsf{c}\,\mathsf{x}\,(\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{xs}) \end{array}$$

- Theorem: If $\mathsf{g} : \forall \{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{g}\,\mathsf{Nil}\,(::)) \;=\; \mathsf{g}\,\mathsf{n}\,\mathsf{c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\,n\,c\,xs = r\} \in Rel(List\,T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\ ((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys) \end{array}$$

- So

$$(g_{List\,T}\,Nil\,(::),\; g_{T'}\,n\,c) \in R$$

i.e.,

$$fold\,n\,c\,(g_{List\,T}\,Nil\,(::)) = g_{T'}\,n\,c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall\,\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{B} & & \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{Nil} & = & \mathsf{n} \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{x} :: \mathsf{xs}) & = & \mathsf{c}\,\mathsf{x}\,(\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{xs}) \end{array}$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{g}\,\mathsf{Nil}\,(::)) \;=\; \mathsf{g}\,\mathsf{n}\,\mathsf{c} \qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\, n\, c\, xs = r\} \in Rel(List\,T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\ ((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys) \end{array}$$

- So

$$(g_{List\,T}\; Nil\,(::),\; g_{T'}\; n\, c) \in R$$

i.e.,

$$fold\,n\,c\,(g_{List\,T}\; Nil\,(::)) = g_{T'}\; n\, c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall\,\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{B} & & \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{Nil} & = & \mathsf{n} \\ \mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{x} :: \mathsf{xs}) & = & \mathsf{c}\,\mathsf{x}\,(\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,\mathsf{xs}) \end{array}$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold}\,\mathsf{n}\,\mathsf{c}\,(\mathsf{g}\,\mathsf{Nil}\,(::)) \;=\; \mathsf{g}\,\mathsf{n}\,\mathsf{c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\,n\,c\,xs = r\} \in Rel(List\,T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\ ((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys) \end{array}$$

- So

$$(g_{List\,T}\;Nil\,(::),\; g_{T'}\;n\,c) \in R$$

i.e.,

$$fold\,n\,c\,(g_{List\,T}\;Nil\,(::)) = g_{T'}\;n\,c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$
\begin{array}{lcl}
\mathsf{fold} : \forall\,\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List}\,\mathsf{A} \to \mathsf{B} \\
\mathsf{fold\,n\,c\,Nil} & = & \mathsf{n} \\
\mathsf{fold\,n\,c\,(x :: xs)} & = & \mathsf{c\,x\,(fold\,n\,c\,xs)}
\end{array}
$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold\,n\,c\,(g\,Nil\,(::))} \;=\; \mathsf{g\,n\,c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\,n\,c\,xs = r\} \in Rel(List\,T, T')$.
- Then

$$
\begin{array}{lcl}
(Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\
((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys)
\end{array}
$$

- So

$$(g_{List\,T}\; Nil\,(::),\; g_{T'}\; n\,c) \in R$$

  i.e.,

$$fold\,n\,c\,(g_{List\,T}\; Nil\,(::)) = g_{T'}\; n\,c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Lists

- Let

$$\begin{array}{lcl} \mathsf{fold} : \forall\,\{\mathsf{A\,B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{List\,A} \to \mathsf{B} & & \\ \mathsf{fold\,n\,c\,Nil} & = & \mathsf{n} \\ \mathsf{fold\,n\,c\,(x :: xs)} & = & \mathsf{c\,x\,(fold\,n\,c\,xs)} \end{array}$$

- Theorem: If $\mathsf{g} : \forall\{\mathsf{A\,B} : \mathsf{Set}\} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ and $\mathsf{n} : \mathsf{T}'$ and $\mathsf{c} : \mathsf{T} \to \mathsf{T}' \to \mathsf{T}'$, then

$$\mathsf{fold\,n\,c\,(g\,Nil\,(::))} \;=\; \mathsf{g\,n\,c} \qquad\qquad (*)$$

- Proof: The AT for System F with ADTs says that, for any $R \in Rel(S, T')$,

$$(g_S,\, g_{T'}) \in R \to (Equal_T \to R \to R) \to R$$

- Let $R = \{(xs, r) \mid fold\,n\,c\,xs = r\} \in Rel(List\,T, T')$.
- Then

$$\begin{array}{lcl} (Nil, n) & \in & R \text{ since } fold\,n\,c\,Nil = n \\ ((::), c) & \in & R \text{ since } fold\,n\,c\,(y :: ys) = c\,y\,(fold\,n\,c\,ys) \end{array}$$

- So

$$(g_{List\,T}\; Nil\,(::),\; g_{T'}\; n\,c) \in R$$

  i.e.,

$$fold\,n\,c\,(g_{List\,T}\; Nil\,(::)) = g_{T'}\; n\,c$$

- Reflecting back into syntax gives $(*)$.
- This program transformation — known as short cut fusion — is not a "naturality style" theorem. It requires the full power of parametricity.

Short Cut Fusion for Nested Types

- We have a similar theorem for every ADT and nested type.
- Since the functors underlying nested types are higher-order, so are their folds.
- Let

$$
\begin{aligned}
&\mathsf{foldP} : \forall\,\{A : \mathsf{Set}\} \to \forall\,\{F : \mathsf{Set} \to \mathsf{Set}\} \to \\
&\qquad\qquad (\forall A.\, A \to F\,A) \to (\forall A.\, F(A \times A) \to F\,A) \to \mathsf{PTree}\,A \to F\,A \\
&\mathsf{foldP}\,l\,n\,(\mathsf{pleaf}\,x) \;=\; l\,n \\
&\mathsf{foldP}\,l\,n\,(\mathsf{pnode}\,xs) \;=\; n\,(\mathsf{foldP}\,l\,n\,xs)
\end{aligned}
$$

- Theorem: If

$$
\begin{aligned}
&g : \forall\,\{A : \mathsf{Set}\} \to \\
&\qquad (\forall\,\{F : \mathsf{Set} \to \mathsf{Set}\} \to (\forall\,A.\,A \to F\,A) \to (\forall\,A.\,F\,(A \times A) \to F\,A) \to F\,A) \to \\
&\qquad\qquad \mathsf{PTree}\,A
\end{aligned}
$$

and $l : \forall\,\{A : \mathsf{Set}\} \to A \to G\,A$ and $n : \forall\,\{A : \mathsf{Set}\} \to G(A \times A) \to GA)$, then

$$\mathsf{foldP}\,l\,n\,(g\,\mathsf{pleaf}\,\mathsf{pnode}) \;=\; g\,l\,n$$

Short Cut Fusion for Nested Types

- We have a similar theorem for every ADT and nested type.
- Since the functors underlying nested types are higher-order, so are their folds.
- Let

$$
\begin{aligned}
&\mathsf{foldP} : \forall\,\{A : \mathsf{Set}\} \to \forall\,\{F : \mathsf{Set} \to \mathsf{Set}\} \to \\
&\qquad\qquad (\forall A.\, A \to F\,A) \to (\forall A.\, F(A \times A) \to F\,A) \to \mathsf{PTree}\,A \to F\,A \\
&\mathsf{foldP}\,l\,n\,(\mathsf{pleaf}\,x) \;=\; l\,n \\
&\mathsf{foldP}\,l\,n\,(\mathsf{pnode}\,xs) \;=\; n\,(\mathsf{foldP}\,l\,n\,xs)
\end{aligned}
$$

- Theorem: If

$$
\begin{aligned}
&g : \forall\,\{A : \mathsf{Set}\} \to \\
&\quad (\forall\,\{F : \mathsf{Set} \to \mathsf{Set}\} \to (\forall\,A.\,A \to F\,A) \to (\forall\,A.\,F\,(A \times A) \to F\,A) \to F\,A) \to \\
&\qquad\quad \mathsf{PTree}\,A
\end{aligned}
$$

and $l : \forall\,\{A : \mathsf{Set}\} \to A \to G\,A$ and $n : \forall\,\{A : \mathsf{Set}\} \to G(A \times A) \to GA)$, then

$$\mathsf{foldP}\,l\,n\,(g\,\mathsf{pleaf}\,\mathsf{pnode}) \;=\; g\,l\,n$$

Short Cut Fusion for Nested Types

- We have a similar theorem for every ADT and nested type.
- Since the functors underlying nested types are higher-order, so are their folds.
- Let

$$
\begin{aligned}
&\mathsf{foldP} : \forall\,\{\mathsf{A} : \mathsf{Set}\} \to \forall\,\{\mathsf{F} : \mathsf{Set} \to \mathsf{Set}\} \to \\
&\qquad (\forall \mathsf{A}.\, \mathsf{A} \to \mathsf{F\,A}) \to (\forall \mathsf{A}.\, \mathsf{F}(\mathsf{A} \times \mathsf{A}) \to \mathsf{F\,A}) \to \mathsf{PTree\,A} \to \mathsf{F\,A} \\
&\mathsf{foldP\,l\,n\,(pleaf\,x)} \;=\; \mathsf{l\,n} \\
&\mathsf{foldP\,l\,n\,(pnode\,xs)} \;=\; \mathsf{n\,(foldP\,l\,n\,xs)}
\end{aligned}
$$

- Theorem: If

$$
\begin{aligned}
&\mathsf{g} : \forall\,\{\mathsf{A} : \mathsf{Set}\} \to \\
&\quad (\forall\,\{\mathsf{F} : \mathsf{Set} \to \mathsf{Set}\} \to (\forall\,\mathsf{A}.\,\mathsf{A} \to \mathsf{F\,A}) \to (\forall\,\mathsf{A}.\,\mathsf{F}\,(\mathsf{A} \times \mathsf{A}) \to \mathsf{F\,A}) \to \mathsf{F\,A}) \to \\
&\qquad \mathsf{PTree\,A}
\end{aligned}
$$

and $\mathsf{l} : \forall\,\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{G\,A}$ and $\mathsf{n} : \forall\,\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{G}(\mathsf{A} \times \mathsf{A}) \to \mathsf{GA})$, then

$$
\mathsf{foldP\,l\,n\,(g\,pleaf\,pnode)} \;=\; \mathsf{g\,l\,n}
$$

Short Cut Fusion for Nested Types

- We have a similar theorem for every ADT and nested type.
- Since the functors underlying nested types are higher-order, so are their folds.
- Let

$$
\begin{aligned}
&\mathsf{foldP} : \forall\,\{A : \mathsf{Set}\} \to \forall\,\{F : \mathsf{Set} \to \mathsf{Set}\} \to \\
&\qquad\qquad (\forall A.\, A \to F\,A) \to (\forall A.\, F(A \times A) \to F\,A) \to \mathsf{PTree}\,A \to F\,A \\
&\mathsf{foldP}\,l\,n\,(\mathsf{pleaf}\,x) \;=\; l\,n \\
&\mathsf{foldP}\,l\,n\,(\mathsf{pnode}\,xs) \;=\; n\,(\mathsf{foldP}\,l\,n\,xs)
\end{aligned}
$$

- Theorem: If

$$
\begin{aligned}
&g : \forall\,\{A : \mathsf{Set}\} \to \\
&\qquad (\forall\,\{F : \mathsf{Set} \to \mathsf{Set}\} \to (\forall\,A.\,A \to F\,A) \to (\forall\,A.\,F\,(A \times A) \to F\,A) \to F\,A) \to \\
&\qquad\qquad \mathsf{PTree}\,A
\end{aligned}
$$

and $l : \forall\,\{A : \mathsf{Set}\} \to A \to G\,A$ and $n : \forall\,\{A : \mathsf{Set}\} \to G(A \times A) \to GA)$, then

$$
\mathsf{foldP}\,l\,n\,(g\,\mathsf{pleaf}\,\mathsf{pnode}) \;=\; g\,l\,n
$$

Summary

- We have seen that we can construct parametric models for languages supporting ADTs and nested types.
- We have seen how to use such a model to derive naturality results and program transformations. We can also derive other standard consequences of parametricity — such as inhabitation results and deep induction rules — in the presence of ADTs and nested types.
- Question: Can we construct parametric models — and thus derive naturality results, program transformations, deep induction rules, and inhabitation results — for GADTs?
- Next time we'll see that:

  - We can construct parametric models for discrete GADTs — but of course these do not have naturality theorems.

  - We can construct models in which GADTs have functorial set and relational interpretations — but these cannot be parametric.
- Question: What should it mean for two GADTs to be related, given that the shape depends on the type of the data it contains?

Summary

- We have seen that we can construct parametric models for languages supporting ADTs and nested types.
- We have seen how to use such a model to derive naturality results and program transformations. We can also derive other standard consequences of parametricity — such as inhabitation results and deep induction rules — in the presence of ADTs and nested types.
- Question: Can we construct parametric models — and thus derive naturality results, program transformations, deep induction rules, and inhabitation results — for GADTs?
- Next time we'll see that:

  - We can construct parametric models for discrete GADTs — but of course these do not have naturality theorems.

  - We can construct models in which GADTs have functorial set and relational interpretations — but these cannot be parametric.
- Question: What should it mean for two GADTs to be related, given that the shape depends on the type of the data it contains?

Summary

- We have seen that we can construct parametric models for languages supporting ADTs and nested types.
- We have seen how to use such a model to derive naturality results and program transformations. We can also derive other standard consequences of parametricity — such as inhabitation results and deep induction rules — in the presence of ADTs and nested types.
- Question: Can we construct parametric models — and thus derive naturality results, program transformations, deep induction rules, and inhabitation results — for GADTs?
- Next time we'll see that:

  - We can construct parametric models for discrete GADTs — but of course these do not have naturality theorems.

  - We can construct models in which GADTs have functorial set and relational interpretations — but these cannot be parametric.
- Question: What should it mean for two GADTs to be related, given that the shape depends on the type of the data it contains?

Summary

- We have seen that we can construct parametric models for languages supporting ADTs and nested types.
- We have seen how to use such a model to derive naturality results and program transformations. We can also derive other standard consequences of parametricity — such as inhabitation results and deep induction rules — in the presence of ADTs and nested types.
- Question: Can we construct parametric models — and thus derive naturality results, program transformations, deep induction rules, and inhabitation results — for GADTs?
- Next time we'll see that:

  - We can construct parametric models for discrete GADTs — but of course these do not have naturality theorems.

  - We can construct models in which GADTs have functorial set and relational interpretations — but these cannot be parametric.

- Question: What should it mean for two GADTs to be related, given that the shape depends on the type of the data it contains?

Summary

- We have seen that we can construct parametric models for languages supporting ADTs and nested types.
- We have seen how to use such a model to derive naturality results and program transformations. We can also derive other standard consequences of parametricity — such as inhabitation results and deep induction rules — in the presence of ADTs and nested types.
- Question: Can we construct parametric models — and thus derive naturality results, program transformations, deep induction rules, and inhabitation results — for GADTs?
- Next time we'll see that:

  - We can construct parametric models for discrete GADTs — but of course these do not have naturality theorems.

  - We can construct models in which GADTs have functorial set and relational interpretations — but these cannot be parametric.
- Question: What should it mean for two GADTs to be related, given that the shape depends on the type of the data it contains?

Summary

- We have seen that we can construct parametric models for languages supporting ADTs and nested types.
- We have seen how to use such a model to derive naturality results and program transformations. We can also derive other standard consequences of parametricity — such as inhabitation results and deep induction rules — in the presence of ADTs and nested types.
- Question: Can we construct parametric models — and thus derive naturality results, program transformations, deep induction rules, and inhabitation results — for GADTs?
- Next time we'll see that:

  - We can construct parametric models for discrete GADTs — but of course these do not have naturality theorems.

  - We can construct models in which GADTs have functorial set and relational interpretations — but these cannot be parametric.
- Question: What should it mean for two GADTs to be related, given that the shape depends on the type of the data it contains?