Algebra of Programming in Agda

Dependent Types for Relational Program Derivation

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson

To appear in *Journal of Functional Programming*

Terms

Algebra of Programming A semi-formal approach to deriving *functional* programs from *relational* specifications.

Agda A *dependently-typed* functional programming language being actively developed.

Dependent types A family of type systems capable of expressing various correctness properties.

In this introductory presentation

> **introductory** *serving as an introduction to a subject or topic [...] intended to persuade someone to purchase something for the first time*
>
> — *Oxford American Dictionaries*

I will (casually) introduce how *squiggolists* treat algorithm design,

and report an experiment that faithfully encodes AoP-style derivations in Agda.

By "faithfully" I mean correctness of a derivation is guaranteed by Agda's type system, while certain degree of readability is retained.

Inductive datatypes

The most natural datatypes in functional programming are *inductive datatypes*.

$$\begin{array}{l}\textbf{data}\ \mathbb{N} : \mathit{Set}\ \textbf{where}\\ \quad \mathit{zero} : \mathbb{N}\\ \quad \mathit{suc} : \mathbb{N} \to \mathbb{N}\end{array}$$

$$\begin{array}{l}\textbf{data}\ [\_]\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where}\\ \quad [] : [A]\\ \quad \_::\_ : A \to [A] \to [A]\end{array}$$

Pattern matching

Functions on inductive datatypes can be defined by *pattern matching*.

$$
\begin{aligned}
&\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&zero + n = n \\
&(suc\ m) + n = suc\ (m + n)
\end{aligned}
$$

$$
\begin{aligned}
&sum : [\mathbb{N}] \to \mathbb{N} \\
&sum\ [] = zero \\
&sum\ (x :: xs) = x + sum\ xs
\end{aligned}
$$

Fold

The pattern of inductive definitions is captured in a higher-order function *foldr*.

$$
\begin{aligned}
&\mathit{foldr} : (A \to B \to B) \to B \to [\,A\,] \to B \\
&\mathit{foldr}\ f\ e\ [\,] = e \\
&\mathit{foldr}\ f\ e\ (x :: xs) = f\ x\ (\mathit{foldr}\ f\ e\ xs)
\end{aligned}
$$

Then alternatively *sum* can be defined as *foldr* _+_ *zero*.

The Fold-Fusion Theorem

It is natural to state theorems about functional programs. One of the most useful theorems is the *Fold-Fusion Theorem*.

Theorem (Fold-Fusion)

Let $f : A \to B \to B$, $e : B$, $g : A \to C \to C$, *and* $h : B \to C$. *If*

$$h\ (f\ x\ y) = g\ x\ (h\ y),$$

then

$$h \cdot foldr\ f\ e = foldr\ g\ (h\ e).$$

Proof of the Fold-Fusion Theorem

The theorem can be inductively proved by simple equational reasoning.

Base case.

$$
\begin{aligned}
& h\ (\mathit{foldr}\ f\ e\ [\,]) \\
= \quad & \{\,\text{definition of } \mathit{foldr}\,\} \\
& h\ e \\
= \quad & \{\,\text{definition of } \mathit{foldr}\,\} \\
& \mathit{foldr}\ g\ (h\ e)\ [\,].
\end{aligned}
$$

Inductive case

$$h\ (foldr\ f\ e\ (x :: xs))$$

Inductive case

$$
\begin{aligned}
& h\ (\mathit{foldr}\ f\ e\ (x :: xs)) \\
= \quad & \{\ \text{definition of } \mathit{foldr}\ \} \\
& h\ (f\ x\ (\mathit{foldr}\ f\ e\ xs))
\end{aligned}
$$

Inductive case

$$
\begin{aligned}
& h\ (\mathit{foldr}\ f\ e\ (x :: xs)) \\
= \quad & \{\ \text{definition of } \mathit{foldr}\ \} \\
& h\ (f\ x\ (\mathit{foldr}\ f\ e\ xs)) \\
= \quad & \{\ \text{fusion condition}\ \} \\
& g\ x\ (h\ (\mathit{foldr}\ f\ e\ xs))
\end{aligned}
$$

Inductive case

$$
\begin{aligned}
& h\ (\mathit{foldr}\ f\ e\ (x :: xs)) \\
= \quad & \{\ \text{definition of } \mathit{foldr}\ \} \\
& h\ (f\ x\ (\mathit{foldr}\ f\ e\ xs)) \\
= \quad & \{\ \text{fusion condition}\ \} \\
& g\ x\ (h\ (\mathit{foldr}\ f\ e\ xs)) \\
= \quad & \{\ \text{induction hypothesis}\ \} \\
& g\ x\ (\mathit{foldr}\ g\ (h\ e)\ xs)
\end{aligned}
$$

Inductive case

$$
\begin{aligned}
& h\ (\mathit{foldr}\ f\ e\ (x :: xs)) \\
= \quad & \{\ \text{definition of } \mathit{foldr}\ \} \\
& h\ (f\ x\ (\mathit{foldr}\ f\ e\ xs)) \\
= \quad & \{\ \text{fusion condition}\ \} \\
& g\ x\ (h\ (\mathit{foldr}\ f\ e\ xs)) \\
= \quad & \{\ \text{induction hypothesis}\ \} \\
& g\ x\ (\mathit{foldr}\ g\ (h\ e)\ xs) \\
= \quad & \{\ \text{definition of } \mathit{foldr}\ \} \\
& \mathit{foldr}\ g\ (h\ e)\ (x :: xs). \qquad \square
\end{aligned}
$$

Sketch of a derivation of radix sort

Gibbons, J. 1999. A Pointless Derivation of Radix Sort. *J. Funct. Program.* 9, 3 (May. 1999), 339–346.

Sketch of a derivation of radix sort

Gibbons, J. 1999. A Pointless Derivation of Radix Sort. *J. Funct. Program.* 9, 3 (May. 1999), 339–346.

- Imagine a datatype *Composite* consisting of a fixed number of *Field*s, where the type *Field* is finite and totally-ordered.

Sketch of a derivation of radix sort

Gibbons, J. 1999. A Pointless Derivation of Radix Sort. *J. Funct. Program.* 9, 3 (May. 1999), 339–346.

- Imagine a datatype *Composite* consisting of a fixed number of *Field*s, where the type *Field* is finite and totally-ordered.
- The function

$$mktree : [\, Composite \longrightarrow Field \,] \longrightarrow [\, Composite \,] \longrightarrow Tree\ [\, Composite \,]$$

receives a significance order of fields and classifies the input list first by the most significant field, and for each bucket classifies the elements by the second-most significant field, and so on.

Sketch of a derivation of radix sort

- It is apparent that

  $$treesort : [\, Composite \longrightarrow Field \,] \longrightarrow [\, Composite \,] \longrightarrow [\, Composite \,]$$
  $$treesort\ ds = flatten \cdot mktree\ ds$$

  correctly sorts a list, where $flatten : Tree\ A \longrightarrow [\, A \,]$ collects elements stored in the leaves in order.

Sketch of a derivation of radix sort

- It is apparent that

  $$treesort : [\, Composite \longrightarrow Field \,] \longrightarrow [\, Composite \,] \longrightarrow [\, Composite \,]$$
  $$treesort\ ds = flatten \cdot mktree\ ds$$

  correctly sorts a list, where $flatten : Tree\ A \longrightarrow [\, A \,]$ collects elements stored in the leaves in order.
- By writing *mktree* as a (higher-order) fold, we get

  $$treesort = (flatten\,\cdot) \cdot mktree.$$

  The Fold-Fusion Theorem is now applicable and radix sort is immediately derived. Moreover, the fusion condition is a statement of stability.

Richard Bird

Unlike other formalisms, functional programming offers a unique opportunity to exploit a compositional approach to Algorithm Design, and to demonstrate the effectiveness of the mathematics of program construction in the presentation of many algorithms[.]

Functional Algorithm Design. *Sci. Comput. Program.* 26, 1-3 (May. 1996), 15–31.

Relations

Relations can be regarded as potentially partial and non-deterministic functions. Then the usual deterministic, total functions become a special case.

A relation R mapping values of A to values of B is denoted by $R : B \leftarrow A$. We write "$b\ R\ a$" if R maps a to b.

For example, $\leq : \mathbb{N} \leftarrow \mathbb{N}$ maps a natural number to itself or a smaller natural number, and $\in : A \leftarrow \wp\, A$ maps a set of A's to one of its elements.

Operators on relations

Relation composition $c\ (R \cdot S)\ a \equiv \exists b.\ c\ R\ b \wedge b\ S\ a$

Converse $a\ R^{\circ}\ b \equiv b\ R\ a$

Power transpose $\Lambda R\ a = \{\, b \mid b\ R\ a \,\}$

Meet $b\ (R \cap S)\ a \equiv b\ R\ a \wedge b\ S\ a$

Join $b\ (R \cup S)\ a \equiv b\ R\ a \vee b\ S\ a$

Product relator $(b, d)\ (R \times S)\ (a, c) \equiv b\ R\ a \wedge d\ S\ c$

Left/right division Division introduces a form of universal quantification into the relational language. We will skip its definition and expand relations defined with division to pointwise formulas in this presentation.

Relational fold

Relational fold is a generalisation of functional fold, and the former can be defined in terms of the latter.

Given $R : B \leftarrow A \times B$ and $s : \wp\, B$, define

$$foldR\ R\ s\ =\ \in \cdot foldr\ (\Lambda(R \cdot (id \times \in)))\ s,$$

where *id* is the identity function. Its type is $B \leftarrow [\,A\,]$.

An operational explanation

$$foldR\ R\ s\ =\ \in \cdot foldr\ (\Lambda(R \cdot (id \times \in)))\ s$$

The set s records possible base cases.

Given a new element $a : A$ and an inductively-computed set $bs : \wp\, B$, $id \times \in$ chooses an element b from bs and then $R\ (a, b)$ is non-deterministically computed. All results that can be generated from such a process are collected by the Λ operator into a set.

Finally $\in$ takes one result from the inductively-computed set.

Compare this procedure with the powerset (subset) construction that converts a non-deterministic finite automaton into a deterministic one.

Example: subsequence

The relation

$$subseq : [A] \leftarrow [A]$$
$$subseq = foldR\ (outr \cup cons)\ nil$$

maps a list to one of its subsequences, where $nil = \{[]\}$ and

$$outr : B \leftarrow A \times B$$
$$outr\ (a, b)\ =\ b$$

$$cons : [A] \leftarrow A \times [A]$$
$$cons\ (x, xs)\ =\ x :: xs.$$

Example: orderedness

The relation

$$ordered : [A] \leftarrow [A]$$
$$ordered = foldR\ (cons \cdot lbound)\ nil$$

maps a list to itself if it is increasingly sorted, where *lbound* maps a pair (x, xs) to itself if x is a lower bound of xs.

If a relation R satisfies $R \subseteq id$, it is called a *coreflexive relation* or a *coreflexive* for short. Coreflexives are used in specifications to filter out those values with some particular property. One can see that both *ordered* and *lbound* are coreflexives.

The activity-selection problem

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$). The activity-selection problem is to select a maximum-size subset of mutually compatible activities.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001 *Introduction to Algorithms*. MIT Press.

If the activities are decreasingly sorted by finish time, there is a simple algorithm.

Maximum

Given an order $R : A \leftarrow A$, the relation $max\, R : A \leftarrow \wp\, A$ maps a set to one of its maximum elements with respect to R:

$$max\, R \;=\; \in \cap\, (R^{\circ} / \ni).$$

Expanding the point-free definition,

$$x\ (max\, R)\ s \;\equiv\; x \in s \;\wedge\; \forall y.\, y \in s \Rightarrow y\, R\, x.$$

Specification

We can give a relational specification for the activity-selection problem:

$$max \leq_\ell \cdot \Lambda(mutex \cdot subseq).$$

- Generate an arbitrary plan by *subseq*.
- Ensure the plan is legitimate by the coreflexive *mutex*.
- Collect all such plans by the Λ operator.
- Choose one of the longest plans by $max \leq_\ell$.

Any function contained in this relation is considered a correct implementation.

The Greedy Theorem

Theorem (Bird and de Moor)

Let $R : A \leftarrow A$ *be a preorder and* $S : A \leftarrow \mathbf{F}A$ *be monotonic on* R, *i.e.,*

$$S \cdot \mathbf{F}R \subseteq R \cdot S.$$

Then

$$max\, R \cdot \Lambda(\!|S|\!) \supseteq (\!|max\, R \cdot \Lambda S|\!).$$

Bird, R. and de Moor, O. 1997 *Algebra of Programming*. Prentice-Hall, Inc.

Specialisation to lists

Corollary

Let $R : A \leftarrow A$ *be a preorder,* $S : A \leftarrow B \times A$ *be monotonic on* R*, i.e.,*

$$S \cdot (id \times R) \ \subseteq \ R \cdot S,$$

and $s : \wp\, A$ *be a set. Then*

$$max\, R \cdot \Lambda(foldR\ S\ s) \ \supseteq \ foldR\ (max\, R \cdot \Lambda S)\ (\Lambda(max\, R)\ s).$$

Fusing $mutex \cdot subseq$ into a fold

The coreflexive *mutex* is defined by

$$mutex \;=\; foldR\ (cons \cdot compatible)\ nil$$

where the coreflexive *compatible* lets a pair (x, xs) of type $Activity \times [\,Activity\,]$ go through if x does not overlap with any of the activities in xs.

By relational fold-fusion, $mutex \cdot subseq$ can be fused into $foldR\ R\ nil$ if we can find a relation R such that

$$mutex \cdot (outr \cup cons) \;=\; R \cdot (id \times mutex).$$

Synthesis of R

We calculate:

$$mutex \cdot (outr \cup cons)$$

Synthesis of R

We calculate:

$$
\begin{array}{ll}
 & mutex \cdot (outr \cup cons) \\
= & \quad \{ \text{composition distributes over join} \} \\
 & (mutex \cdot outr) \cup (mutex \cdot cons)
\end{array}
$$

Synthesis of R

We calculate:

$$
\begin{array}{ll}
 & mutex \cdot (outr \cup cons) \\
= & \quad \{ \text{composition distributes over join} \} \\
 & (mutex \cdot outr) \cup (mutex \cdot cons) \\
= & \quad \{ \text{product relator; catamorphisms} \} \\
 & (outr \cdot (id \times mutex)) \cup (cons \cdot compatible \cdot (id \times mutex))
\end{array}
$$

Synthesis of R

We calculate:

$$
\begin{array}{ll}
 & mutex \cdot (outr \cup cons) \\
= & \quad \{ \text{composition distributes over join} \} \\
 & (mutex \cdot outr) \cup (mutex \cdot cons) \\
= & \quad \{ \text{product relator; catamorphisms} \} \\
 & (outr \cdot (id \times mutex)) \cup (cons \cdot compatible \cdot (id \times mutex)) \\
= & \quad \{ \text{composition distributes over join} \} \\
 & (outr \cup (cons \cdot compatible)) \cdot (id \times mutex).
\end{array}
$$

So $mutex \cdot subseq = foldR\ (outr \cup (cons \cdot compatible))\ nil$.

Checking the monotonicity condition

Now we wish to apply the Greedy Theorem. That means we need to check the monotonicity condition

$$(outr \cup (cons \cdot compatible)) \cdot (id \times \leq_\ell) \subseteq \leq_\ell \cdot (outr \cup (cons \cdot compatible)),$$

which says for two partial plans xs and ys with $length\ xs \leq length\ ys$, no matter which complete plan xs' is computed from xs, there is always a complete plan ys' computed from ys such that $length\ xs' \leq length\ ys'$.

This is not true, however.

Strengthening the order

A typical strategy is to refine the order $\leq_\ell$ to a stronger one. Since the problematic case happens when the two partial plans under consideration are of the same length, we define

$$\trianglelefteq \;=\; <_\ell \cup (=_\ell \cap (\mathit{post\text{-}compatible} \backslash \mathit{post\text{-}compatible}))$$

where $\mathit{post\text{-}compatible} : \mathit{Activity} \leftarrow [\,\mathit{Activity}\,]$ maps a plan to a compatible activity that finishes later than all activities in the plan.

Now the monotonicity condition with respect to $\trianglelefteq$ holds and the Greedy Theorem gives us

$$\mathit{foldR}\ (\mathit{max}\trianglelefteq \cdot \Lambda(\mathit{outr} \cup (\mathit{cons} \cdot \mathit{compatible})))\ \mathit{nil}.$$

Cleaning up

Take a closer look at

$$max \trianglelefteq \cdot \Lambda(outr \cup (cons \cdot compatible)).$$

Given a partial plan xs and an incoming activity x, the relation picks a best plan with respect to $\trianglelefteq$ from, depending on whether x is compatible with xs, the set $\{xs, x :: xs\}$ or the set $\{xs\}$.

In the first case $x :: xs$ should be chosen since it is strictly longer than xs. In the second case there is only one candidate.

Since x finishes later than xs, it suffices to check whether x starts later than xs.

The final program

Define

$$
\begin{aligned}
&\mathit{compatible\text{-}cons} : \mathit{Activity} \to [\,\mathit{Activity}\,] \to [\,\mathit{Activity}\,] \\
&\mathit{compatible\text{-}cons}\ x\ [\,] = x :: [\,] \\
&\mathit{compatible\text{-}cons}\ x\ (y :: xs) = \\
&\qquad \textbf{if}\ \mathit{start}\ x \geq \mathit{finish}\ y\ \textbf{then}\ x :: y :: xs\ \textbf{else}\ y :: xs.
\end{aligned}
$$

The final program is a simple fold:

$$\mathit{foldr}\ \mathit{compatible\text{-}cons}\ [\,].$$

The derivation

$$
\begin{array}{ll}
 & \mathit{max} \leq_\ell \cdot \Lambda(\mathit{mutex} \cdot \mathit{subseq}) \\
= & \quad \{\, \text{fusion} \,\} \\
 & \mathit{max} \leq_\ell \cdot \Lambda(\mathit{foldR}\ (\mathit{outr} \cup (\mathit{cons} \cdot \mathit{compatible}))\ \mathit{nil}) \\
\supseteq & \quad \{\, \text{strengthening} \,\} \\
 & \mathit{max} \trianglelefteq \cdot \Lambda(\mathit{foldR}\ (\mathit{outr} \cup (\mathit{cons} \cdot \mathit{compatible}))\ \mathit{nil}) \\
\supseteq & \quad \{\, \text{Greedy Theorem} \,\} \\
 & \mathit{foldR}\ (\mathit{max} \trianglelefteq \cdot \Lambda(\mathit{outr} \cup (\mathit{cons} \cdot \mathit{compatible})))\ \mathit{nil} \\
= & \quad \{\, \text{cleaning up} \,\} \\
 & \mathit{foldr}\ \mathit{compatible\text{-}cons}\ [].
\end{array}
$$

Donald E. Knuth

> *Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it. [...] we should continually be striving to transform every art into a science: in the process, we advance the art.*

Computer Programming as an Art. *Commun. ACM* 17, 12 (Dec. 1974), 667–673.

Curry-Howard correspondence

Observe the following two logic inference rules:

$$(\wedge\mathrm{E})\ \frac{\varphi \wedge \psi}{\psi} \qquad\qquad (\rightarrow\mathrm{E})\ \frac{\varphi \rightarrow \psi \qquad \varphi}{\psi}$$

The left one resembles the type of $outr : A \times B \rightarrow B$, and the right one looks like the types involved in function application.

Curry-Howard correspondence is a family of formal correspondences between logic deduction systems and computational calculi. Its famous slogan is "propositions are types, and proofs are programs".

Thus writing proofs is no different from writing programs! All we need is an expressive enough programming language.

Roadmap

We will go through the following:

1. first-order intuitionistic logic,
2. preorder reasoning combinators,
3. modelling of sets and relations,
4. the specification (in detail),
5. the Partial Greedy Theorem, and
6. finally, the derivation.

More complicated proofs cannot be explained in this presentation since more background is required. However it is worth noting that important theorems such as Fold-Fusion Theorem and Greedy Theorem can be proved in an "equational" style close to the proofs in the AoP book.

Encoding first-order intuitionistic logic in Agda

absurdity

$$\textbf{data}\ \bot : \mathit{Set}\ \textbf{where}$$

truth

$$\textbf{data}\ \top : \mathit{Set}\ \textbf{where}$$
$$tt : \top$$

conjunction

$$\textbf{data}\ \_\times\_\ (A\ B : \mathit{Set}) : \mathit{Set}\ \textbf{where}$$
$$\_,\_ : A \to B \to A \times B$$

disjunction

$$\textbf{data}\ \_\uplus\_\ (A\ B : \mathit{Set}) : \mathit{Set}\ \textbf{where}$$
$$inj_1 : A \to A \uplus B$$
$$inj_2 : B \to A \uplus B$$

implication

$$A \to B$$

Encoding first-order intuitionistic logic in Agda

exists

$$\textbf{data}\ \exists\ \{A : Set\}\ (P : A \to Set) : Set\ \textbf{where}$$
$$\_,\_ : (witness : A) \to P\ witness \to \exists\ P$$

forall

$$(x : A) \to P\ x \quad \text{or} \quad \forall\ x \to P\ x$$

equality

$$\textbf{data}\ \_\equiv\_\ \{A : Set\}\ : A \to A \to Set\ \textbf{where}$$
$$refl : \{x : A\} \to x \equiv x$$

For example, the type

$$(n : \mathbb{N}) \to (n \equiv zero\ \uplus\ \exists\ (\lambda\, m \to n \equiv suc\ m))$$

says every natural number is either zero or a successor of some natural number.

More on products

In fact, $\_\times\_$ and $\exists$ are special cases of the following Σ-*type*:

$$\begin{aligned}&\textbf{data}\ \Sigma\ (A : Set)\ (B : A \to Set) : Set\ \textbf{where}\\ &\quad \_,\_ : (a : A) \to B\ a \to \Sigma\ A\ B\end{aligned}$$

which can be seen as a kind of generalised products whose second component's type depends on the first component. We can define extractor functions

$$\begin{aligned}&proj_1 : \{A : Set\}\ \{B : A \to Set\} \to \Sigma\ A\ B \to A\\ &proj_1\ (a\,,b) = a\end{aligned}$$

$$\begin{aligned}&proj_2 : \{A : Set\}\ \{B : A \to Set\} \to (\sigma : \Sigma\ A\ B) \to B\ (proj_1\ \sigma)\\ &proj_2\ (a\,,b) = b.\end{aligned}$$

More on equality

_≡_ is reflexive by definition, and can be proven to be transitive and symmetric. It is also a congruence with respect to function application.

$$trans : \{A : Set\}\ \{x\ y\ z : A\} \to x \equiv y \to y \equiv z \to x \equiv z$$
$$trans\ refl\ refl = refl$$

$$sym : \{A : Set\}\ \{x\ y : A\} \to x \equiv y \to y \equiv x$$
$$sym\ refl = refl$$

$$cong : \{A\ B : Set\}\ (f : A \to B)\ \{x\ y : A\} \to x \equiv y \to f\ x \equiv f\ y$$
$$cong\ f\ refl = refl$$

Equational reasoning

We can write a chain of equality in Agda as

$$
\begin{aligned}
& expression_1 \\
\equiv\langle\ & exp_1 \equiv exp_2\ \rangle \\
& expression_2 \\
\equiv\langle\ & exp_2 \equiv exp_3\ \rangle \\
& \ldots \\
\equiv\langle\ & exp_{n-1} \equiv exp_n\ \rangle \\
& expression_n \quad \square
\end{aligned}
$$

Equational reasoning

We can write a chain of equality in Agda as

$$
\begin{aligned}
& expression_1 \\
\equiv\langle\ & exp_1 \equiv exp_2\ \rangle \\
& expression_2 \\
\equiv\langle\ & exp_2 \equiv exp_3\ \rangle \\
& \ldots \\
\equiv\langle\ & exp_{n-1} \equiv exp_n\ \rangle \\
& expression_n \quad \square
\end{aligned}
$$

Equational reasoning

We can write a chain of equality in Agda as

$$
\begin{aligned}
& expression_1 \\
\equiv\langle\ & exp_1 \equiv exp_2 \ \rangle \\
& expression_2 \\
\equiv\langle\ & exp_2 \equiv exp_3 \ \rangle \\
& \ldots \\
\equiv\langle\ & exp_{n-1} \equiv exp_n \ \rangle \\
& expression_n \quad \square
\end{aligned}
$$

Equational reasoning

We can write a chain of equality in Agda as

$$
\begin{aligned}
& expression_1 \\
\equiv\langle\ & exp_1 \equiv exp_2 \ \rangle \\
& expression_2 \\
\equiv\langle\ & exp_2 \equiv exp_3 \ \rangle \\
& \ldots \\
\equiv\langle\ & exp_{n-1} \equiv exp_n \ \rangle \\
& expression_n \quad \square
\end{aligned}
$$

Equational reasoning

We can write a chain of equality in Agda as

$$
\begin{aligned}
& expression_1 \\
\equiv\langle\ & exp_1 \equiv exp_2\ \rangle \\
& expression_2 \\
\equiv\langle\ & exp_2 \equiv exp_3\ \rangle \\
& \ldots \\
\equiv\langle\ & exp_{n-1} \equiv exp_n\ \rangle \\
& expression_n \quad \square
\end{aligned}
$$

Preorder reasoning combinators

Concretely,

$$\_\equiv\langle\_\rangle\_ : \{A : Set\}\ (x : A)\ \{y\ z : A\} \to x \equiv y \to y \equiv z \to x \equiv z$$
$$x \equiv\langle\ x{\equiv}y\ \rangle\ y{\equiv}z\ =\ trans\ x{\equiv}y\ y{\equiv}z$$

$$\_\Box : \{A : Set\}\ (x : A) \to x \equiv x$$
$$x\ \Box\ =\ refl.$$

The same technique works for any preorder.

Proving the Fold-Fusion Theorem

```
foldr-fusion : {A B C : Set}
  (h : B → C) {f : A → B → B} {g : A → C → C} {e : B} →
    (∀ x y → h (f x y) ≡ g x (h y)) →
      ∀ xs → h (foldr f e xs) ≡ foldr g (h e) xs
foldr-fusion h {f} {g} {e} fusion-condition [] = refl
foldr-fusion h {f} {g} {e} fusion-condition (x :: xs) =
    h (foldr f e (x :: xs))
  ≡⟨ refl ⟩
    h (f x (foldr f e xs))
  ≡⟨ fusion-condition x (foldr f e xs) ⟩
    g x (h (foldr f e xs))
  ≡⟨ cong (g x) (foldr-fusion h fusion-condition xs) ⟩
    g x (foldr g (h e) xs)
  ≡⟨ refl ⟩
    foldr g (h e) (x :: xs)  □
```

Modelling sets

A set of A's can be specified by its characteristic function of type $A \longrightarrow Set$. Concretely, if $s : A \longrightarrow Set$, then $x : A$ is considered a member of s if $s\ x$ has a proof, i.e., there is a term that has type $s\ x$. Thus we define

$$\wp\, A \;=\; A \longrightarrow Set.$$

For example, define

$$singleton : \{A : Set\} \longrightarrow A \longrightarrow \wp\, A$$
$$singleton\ x = \lambda\, y \longrightarrow x \equiv y.$$

Then *singleton* x is the set containing a single element x.

Modelling relations

Set-theoretically, a relation of type $B \leftarrow A$ is a subset of $A \times B$, so we may define $B \leftarrow A$ as $\wp(A \times B) = A \times B \rightarrow Set$. But we found the following isomorphic representation more convenient:

$$B \leftarrow A \;=\; B \rightarrow A \rightarrow Set.$$

For example, an Agda function is lifted to a relation by the combinator

$$\begin{aligned} &fun : \{A\ B : Set\} \rightarrow (A \rightarrow B) \rightarrow (B \leftarrow A) \\ &fun\ f \;=\; \lambda\, y\, x \rightarrow f\, x \equiv y. \end{aligned}$$

Operators on relations revisited

Operators on relations revisited

composition

$$\_\cdot\_ : \{A\ B\ C : Set\} \rightarrow (C \leftarrow B) \rightarrow (B \leftarrow A) \rightarrow (C \leftarrow A)$$

$$(\_R\_ \cdot \_S\_)\ c\ a\ =\ \exists\ (\lambda\, b \rightarrow c\ R\ b \times b\ S\ a)$$

Operators on relations revisited

composition

$$\_\cdot\_ : \{A\ B\ C : Set\} \rightarrow (C \leftarrow B) \rightarrow (B \leftarrow A) \rightarrow (C \leftarrow A)$$

$$(\_R\_ \cdot \_S\_)\ c\ a = \exists\ (\lambda\, b \rightarrow c\ R\ b \times b\ S\ a)$$

converse

$$\_^{\circ} : \{A\ B : Set\} \rightarrow (B \leftarrow A) \rightarrow (A \leftarrow B)$$

$$(R\ ^{\circ})\ a\ b = R\ b\ a$$

Operators on relations revisited

composition

$$\_\cdot\_ : \{A\ B\ C : Set\} \rightarrow (C \leftarrow B) \rightarrow (B \leftarrow A) \rightarrow (C \leftarrow A)$$

$$(\_R\_ \cdot \_S\_)\ c\ a = \exists\ (\lambda\, b \rightarrow c\ R\ b \times b\ S\ a)$$

converse

$$\_^{\circ} : \{A\ B : Set\} \rightarrow (B \leftarrow A) \rightarrow (A \leftarrow B)$$

$$(R\ ^{\circ})\ a\ b = R\ b\ a$$

power transpose

$$\Lambda : \{A\ B : Set\} \rightarrow (B \leftarrow A) \rightarrow (A \rightarrow \wp\, B)$$

$$\Lambda\ \_R\_\ a = \lambda\, b \rightarrow b\ R\ a$$

Operators on relations revisited

composition

$$\_\cdot\_ : \{A\ B\ C : Set\} \rightarrow (C \leftarrow B) \rightarrow (B \leftarrow A) \rightarrow (C \leftarrow A)$$
$$(\_R\_ \cdot \_S\_)\ c\ a = \exists\ (\lambda\, b \rightarrow c\ R\ b \times b\ S\ a)$$

converse

$$\_^{\circ} : \{A\ B : Set\} \rightarrow (B \leftarrow A) \rightarrow (A \leftarrow B)$$
$$(R\ ^{\circ})\ a\ b = R\ b\ a$$

power transpose

$$\Lambda : \{A\ B : Set\} \rightarrow (B \leftarrow A) \rightarrow (A \rightarrow \wp\, B)$$
$$\Lambda\ \_R\_\ a = \lambda\, b \rightarrow b\ R\ a$$

join

$$\_\sqcup\_ : \{A\ B : Set\} \rightarrow (B \leftarrow A) \rightarrow (B \leftarrow A) \rightarrow (B \leftarrow A)$$
$$(R \sqcup S)\ b\ a = R\ b\ a \uplus S\ b\ a$$

Inclusion

We can define set inclusion

$$\begin{aligned} \_\subseteq\_ &: \{A : Set\} \to \wp\, A \to \wp\, A \to Set \\ s_1 \subseteq s_2 &= \forall\, x \to (s_1\ x \to s_2\ x) \end{aligned}$$

and relational inclusion

$$\begin{aligned} \_\sqsubseteq\_ &: \{A\ B : Set\} \to (B \leftarrow A) \to (B \leftarrow A) \to Set \\ R \sqsubseteq S &= \forall\ b\ a \to (R\ b\ a \to S\ b\ a). \end{aligned}$$

Both can be shown to be reflexive and transitive, so we can build preorder reasoning combinators for them.

Encoding the specification

Our goal is to build up the relation

$$act\text{-}sel\text{-}spec : [\, Activity \,] \leftarrow [\, Activity \,]$$
$$act\text{-}sel\text{-}spec = max\ \_\leq_\ell\_ \bullet \Lambda(mutex \cdot subseq).$$

If we can construct a term of type

$$\exists\ (\lambda\, f \rightarrow f \cdot fin\text{-}ordered \ \sqsubseteq\ act\text{-}sel\text{-}spec \cdot fin\text{-}ordered)$$

where *fin-ordered* is a coreflexive explicitly stating that the input list is required to be sorted by finish time, the witness to this existential proposition is an algorithm solving the activity-selection problem, correctness guaranteed by the type checker.

The "maximum" part

Maximum is easily defined:

$$max : \{A : Set\} \to (A \leftarrow A) \to (A \leftarrow \wp\, A)$$
$$(max\ \_R\_)\ a\ s\ =\ s\ a\ \times\ (\forall\ a' \to s\ a' \to a'\ R\ a).$$

In our library it is defined in point-free style.

Assuming $\_\leq\_ : \mathbb{N} \leftarrow \mathbb{N}$ is given,

$$\_\leq_\ell\_ : \{A : Set\} \to ([\,A\,] \leftarrow [\,A\,])$$
$$xs \leq_\ell ys\ =\ length\ xs \leq length\ ys.$$

Fold, the third time

Define

$$foldR : \{A\ B : Set\} \to (B \leftarrow A \times B) \to \wp\, B \to (B \leftarrow [\,A\,])$$
$$foldR\ R\ s\ =\ \in \bullet foldr\ (\Lambda(R \cdot (id_R \times \in)))\ s$$

where _•_ is a special type of composition defined by

$$\_\bullet\_ : \{A\ B\ C : Set\} \to (C \leftarrow \wp\, B) \to (A \to \wp\, B) \to (C \leftarrow A)$$
$$(R \bullet f)\ c\ a\ =\ R\ c\ (f\ a).$$

The reason we need such composition is due to *predicativity*, an issue we will ignore in this presentation.

Subsequence, identically

Define

$$subseq : \{A : Set\} \to ([\,A\,] \leftarrow [\,A\,])$$
$$subseq = foldR\ (outr \sqcup cons)\ nil$$

where

$$cons : \{A : Set\} \to (A \leftarrow (A \times [\,A\,]))$$
$$cons = fun\ (uncurry\ \_::\_)$$

$$outr : \{A\ B : Set\} \to (B \leftarrow (A \times B))$$
$$outr = fun\ proj_2$$

$$nil : \{A : Set\} \to \wp\, A$$
$$nil = singleton\ [].$$

Junctional check

Recall that

$$mutex = foldR\ (cons \cdot compatible)\ nil$$

where *compatible* is a coreflexive which, given a pair (x, xs) of type $Activity \times [Activity]$, checks whether x is disjoint from every activity in xs. The relation *fin-ordered* has the same structure. Define

$$check\ C = foldR\ (cons \cdot C)\ nil$$

to capture this pattern. Then $mutex = check\ compatible$.

The only component we haven't constructed is *compatible*.

Compatibility between an activity and a plan

Define

$$
\begin{aligned}
&\mathit{compatible\text{-}set} : \wp\,(\mathit{Activity} \times [\,\mathit{Activity}\,]) \\
&\mathit{compatible\text{-}set}\ (a\,, [\,]) \ = \ \top \\
&\mathit{compatible\text{-}set}\ (a\,, x :: \mathit{xs}) \ = \ \mathit{disjoint}\ a\ x \ \times \ \mathit{compatible\text{-}set}\ (a\,, \mathit{xs})
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{disjoint} : \mathit{Activity} \leftarrow \mathit{Activity} \\
&\mathit{disjoint}\ a\ x \ = \ \mathit{finish}\ x \leq \mathit{start}\ a \ \uplus \ \mathit{finish}\ a \leq \mathit{start}\ x.
\end{aligned}
$$

Then

$$
\begin{aligned}
&\mathit{compatible} : (\mathit{Activity} \times [\,\mathit{Activity}\,]) \leftarrow (\mathit{Activity} \times [\,\mathit{Activity}\,]) \\
&\mathit{compatible}\ (y\,, \mathit{ys})\ (x\,, \mathit{xs}) \ = \\
&\qquad (y\,, \mathit{ys}) \equiv (x\,, \mathit{xs}) \ \times \ \mathit{compatible\text{-}set}\ (x\,, \mathit{xs}).
\end{aligned}
$$

The Partial Greedy Theorem

Since the domain of $\mathit{act\text{-}sel\text{-}spec} = \mathit{max}\ \_\leq_\ell\_ \bullet \Lambda(\mathit{mutex} \cdot \mathit{subseq})$ is restricted by the coreflexive *fin-ordered*, we need a variant of the Greedy Theorem to deal with partiality of input.

$$
\begin{aligned}
&\mathit{partial\text{-}greedy\text{-}thm} : \{A\ B : \mathit{Set}\} \\
&\quad \{S : B \leftarrow (A \times B)\}\ \{s : \wp\, B\}\ \{R : B \leftarrow B\} \\
&\quad \{C : (A \times [A]) \leftarrow (A \times [A])\}\ \{D : (A \times B) \leftarrow (A \times B)\} \rightarrow \\
&\qquad R \cdot R \sqsubseteq R \rightarrow C \sqsubseteq \mathit{id}_R \rightarrow D \sqsubseteq \mathit{id}_R \rightarrow \\
&\qquad S \cdot (\mathit{id}_R \times R) \cdot D \sqsubseteq R \cdot S \rightarrow \\
&\qquad (\mathit{id}_R \times \mathit{foldR}\ S\ s) \cdot C \sqsubseteq D \cdot (\mathit{id}_R \times \mathit{foldR}\ S\ s) \rightarrow \\
&\qquad\quad \mathit{foldR}\ (\mathit{max}\, R \bullet \Lambda S)\ (\Lambda(\mathit{max}\, R)\ s) \cdot \mathit{check}\ C \\
&\qquad\qquad \sqsubseteq (\mathit{max}\, R \bullet \Lambda(\mathit{foldR}\ S\ s)) \cdot \mathit{check}\ C
\end{aligned}
$$

The Partial Greedy Theorem

Since the domain of $\mathit{act\text{-}sel\text{-}spec} = \mathit{max}\ \_\leq_\ell\_ \bullet \Lambda(\mathit{mutex} \cdot \mathit{subseq})$ is restricted by the coreflexive *fin-ordered*, we need a variant of the Greedy Theorem to deal with partiality of input.

$$
\begin{aligned}
&\mathit{partial\text{-}greedy\text{-}thm} : \{A\ B : \mathit{Set}\} \\
&\quad \{S : B \leftarrow (A \times B)\}\ \{s : \wp B\}\ \{R : B \leftarrow B\} \\
&\quad \{C : (A \times [A]) \leftarrow (A \times [A])\}\ \{D : (A \times B) \leftarrow (A \times B)\} \rightarrow \\
&\qquad R \cdot R \sqsubseteq R \rightarrow C \sqsubseteq \mathit{id}_R \rightarrow D \sqsubseteq \mathit{id}_R \rightarrow \\
&\qquad S \cdot (\mathit{id}_R \times R) \cdot D \sqsubseteq R \cdot S \rightarrow \\
&\qquad (\mathit{id}_R \times \mathit{foldR}\ S\ s) \cdot C \sqsubseteq D \cdot (\mathit{id}_R \times \mathit{foldR}\ S\ s) \rightarrow \\
&\qquad\quad \mathit{foldR}\ (\mathit{max}\ R \bullet \Lambda S)\ (\Lambda(\mathit{max}\ R)\ s) \cdot \mathit{check}\ C \\
&\qquad\qquad \sqsubseteq (\mathit{max}\ R \bullet \Lambda(\mathit{foldR}\ S\ s)) \cdot \mathit{check}\ C
\end{aligned}
$$

The Partial Greedy Theorem

Since the domain of $\mathit{act\text{-}sel\text{-}spec} = \mathit{max}\ \_\leq_\ell\_ \bullet \Lambda(\mathit{mutex} \cdot \mathit{subseq})$ is restricted by the coreflexive *fin-ordered*, we need a variant of the Greedy Theorem to deal with partiality of input.

$$
\begin{aligned}
&\mathit{partial\text{-}greedy\text{-}thm} : \{A\ B : \mathit{Set}\} \\
&\quad \{S : B \leftarrow (A \times B)\}\ \{s : \wp B\}\ \{R : B \leftarrow B\} \\
&\quad \{C : (A \times [A]) \leftarrow (A \times [A])\}\ \{D : (A \times B) \leftarrow (A \times B)\} \rightarrow \\
&\qquad R \cdot R \sqsubseteq R \rightarrow C \sqsubseteq \mathit{id}_R \rightarrow D \sqsubseteq \mathit{id}_R \rightarrow \\
&\qquad S \cdot (\mathit{id}_R \times R) \cdot D \sqsubseteq R \cdot S \rightarrow \\
&\qquad (\mathit{id}_R \times \mathit{foldR}\ S\ s) \cdot C \sqsubseteq D \cdot (\mathit{id}_R \times \mathit{foldR}\ S\ s) \rightarrow \\
&\qquad\quad \mathit{foldR}\ (\mathit{max}\ R \bullet \Lambda S)\ (\Lambda(\mathit{max}\ R)\ s) \cdot \mathit{check}\ C \\
&\qquad\qquad \sqsubseteq (\mathit{max}\ R \bullet \Lambda(\mathit{foldR}\ S\ s)) \cdot \mathit{check}\ C
\end{aligned}
$$

The Partial Greedy Theorem

Since the domain of $\mathit{act\text{-}sel\text{-}spec} = \mathit{max}\ \_\leq_\ell\_ \bullet \Lambda(\mathit{mutex} \cdot \mathit{subseq})$ is restricted by the coreflexive *fin-ordered*, we need a variant of the Greedy Theorem to deal with partiality of input.

$$
\begin{aligned}
&\mathit{partial\text{-}greedy\text{-}thm} : \{A\ B : \mathit{Set}\} \\
&\quad \{S : B \leftarrow (A \times B)\}\ \{s : \wp B\}\ \{R : B \leftarrow B\} \\
&\quad \{C : (A \times [A]) \leftarrow (A \times [A])\}\ \{D : (A \times B) \leftarrow (A \times B)\} \rightarrow \\
&\qquad R \cdot R \sqsubseteq R \rightarrow C \sqsubseteq \mathit{id}_R \rightarrow D \sqsubseteq \mathit{id}_R \rightarrow \\
&\qquad S \cdot (\mathit{id}_R \times R) \cdot D \sqsubseteq R \cdot S \rightarrow \\
&\qquad (\mathit{id}_R \times \mathit{foldR}\ S\ s) \cdot C \sqsubseteq D \cdot (\mathit{id}_R \times \mathit{foldR}\ S\ s) \rightarrow \\
&\qquad\quad \mathit{foldR}\ (\mathit{max}\ R \bullet \Lambda S)\ (\Lambda(\mathit{max}\ R)\ s) \cdot \mathit{check}\ C \\
&\qquad\qquad \sqsubseteq (\mathit{max}\ R \bullet \Lambda(\mathit{foldR}\ S\ s)) \cdot \mathit{check}\ C
\end{aligned}
$$

The Partial Greedy Theorem

Since the domain of $\mathit{act\text{-}sel\text{-}spec} = \mathit{max}\ \_\leq_\ell\_ \bullet \Lambda(\mathit{mutex} \cdot \mathit{subseq})$ is restricted by the coreflexive *fin-ordered*, we need a variant of the Greedy Theorem to deal with partiality of input.

$$
\begin{aligned}
&\mathit{partial\text{-}greedy\text{-}thm} : \{A\ B : \mathit{Set}\} \\
&\quad \{S : B \leftarrow (A \times B)\}\ \{s : \wp B\}\ \{R : B \leftarrow B\} \\
&\quad \{C : (A \times [A]) \leftarrow (A \times [A])\}\ \{D : (A \times B) \leftarrow (A \times B)\} \rightarrow \\
&\qquad R \cdot R \sqsubseteq R \rightarrow C \sqsubseteq \mathit{id}_R \rightarrow D \sqsubseteq \mathit{id}_R \rightarrow \\
&\qquad S \cdot (\mathit{id}_R \times R) \cdot D \sqsubseteq R \cdot S \rightarrow \\
&\qquad (\mathit{id}_R \times \mathit{foldR}\ S\ s) \cdot C \sqsubseteq D \cdot (\mathit{id}_R \times \mathit{foldR}\ S\ s) \rightarrow \\
&\qquad\quad \mathit{foldR}\ (\mathit{max}\ R \bullet \Lambda S)\ (\Lambda(\mathit{max}\ R)\ s) \cdot \mathit{check}\ C \\
&\qquad\qquad \sqsubseteq (\mathit{max}\ R \bullet \Lambda(\mathit{foldR}\ S\ s)) \cdot \mathit{check}\ C
\end{aligned}
$$

The main derivation

(Let $S = outr \sqcup (cons \cdot compatible)$.)

$$
\begin{aligned}
&activity\text{-}selection\text{-}derivation : \\
&\quad \exists\, (\lambda\, f \rightarrow fun\ f \cdot fin\text{-}ordered \ \sqsubseteq\ act\text{-}sel\text{-}spec \cdot fin\text{-}ordered) \\
&activity\text{-}selection\text{-}derivation = \_\,, \\
&\quad (max\ \_\leq_\ell\_ \bullet \Lambda(mutex \cdot subseq)) \cdot fin\text{-}ordered \\
&\sqsupseteq\langle\ comp\text{-}monotonic\text{-}l\ (minΛ\text{-}cong\text{-}\sqsupseteq\ mutex{\cdot}subseq\text{-}is\text{-}fold)\ \rangle \\
&\quad (max\ \_\leq_\ell\_ \bullet \Lambda(foldR\ S\ nil)) \cdot fin\text{-}ordered \\
&\sqsupseteq\langle\ comp\text{-}monotonic\text{-}l\ (max\text{-}monotonic\ \trianglelefteq\text{-}refines\text{-}\leq_\ell)\ \rangle \\
&\quad (max\ \_\trianglelefteq\_ \bullet \Lambda(foldR\ S\ nil)) \cdot fin\text{-}ordered \\
&\quad \ldots
\end{aligned}
$$

Applying the Partial Greedy Theorem

(Assume $fin\text{-}ordered = check\ fin\text{-}ubound$.)

$$
\begin{array}{l}
\ldots \\
\quad (max\ \_\trianglelefteq\_ \bullet \Lambda(foldR\ S\ nil)) \cdot fin\text{-}ordered \\
\sqsupseteq \langle\ fin\text{-}ubound\text{-}promotion\ \rangle \\
\quad (max\ \_\trianglelefteq\_ \bullet \Lambda(foldR\ (S \cdot fin\text{-}ubound)\ nil)) \cdot fin\text{-}ordered \\
\sqsupseteq \langle\ partial\text{-}greedy\text{-}thm\ \trianglelefteq\text{-}trans\ fin\text{-}ubound{\sqsubseteq}id_R\ fin\text{-}ubound{\sqsubseteq}id_R \\
\qquad monotonicity\ fin\text{-}ubound\text{-}homo\ \rangle \\
\quad foldR\ (max\ \_\trianglelefteq\_ \bullet \Lambda(S \cdot fin\text{-}ubound))\ (\Lambda(max\ \_\trianglelefteq\_)\ nil) \cdot fin\text{-}ordered \\
\sqsupseteq \langle\ comp\text{-}monotonic\text{-}I\ (foldR\text{-}monotonic\ \sqsubseteq\text{-}refl\ max\text{-}\trianglelefteq\text{-}nil{\sqsupseteq}nil)\ \rangle \\
\quad foldR\ (max\ \_\trianglelefteq\_ \bullet \Lambda(S \cdot fin\text{-}ubound))\ nil \cdot fin\text{-}ordered \\
\ldots
\end{array}
$$

Cleaning up

$$
\begin{array}{ll}
\ldots & \\
\quad & \mathit{foldR}\ (\mathit{max}\ \_\trianglelefteq\_ \bullet \Lambda(S \cdot \mathit{fin\text{-}ubound}))\ \mathit{nil} \cdot \mathit{fin\text{-}ordered} \\
\sqsupseteq\langle & \mathit{comp\text{-}monotonic\text{-}I}\ \mathit{algebra\text{-}refinement}\ \rangle \\
\quad & \mathit{foldR}\ (\mathit{fun}\ (\mathit{uncurry}\ \mathit{compatible\text{-}cons}) \cdot \mathit{fin\text{-}ubound})\ \mathit{nil} \cdot \mathit{fin\text{-}ordered} \\
\sqsupseteq\langle & \mathit{fin\text{-}ubound\text{-}demotion}\ \rangle \\
\quad & \mathit{foldR}\ (\mathit{fun}\ (\mathit{uncurry}\ \mathit{compatible\text{-}cons}))\ \mathit{nil} \cdot \mathit{fin\text{-}ordered} \\
\sqsupseteq\langle & \mathit{comp\text{-}monotonic\text{-}I}\ (\mathit{foldR\text{-}to\text{-}foldr}\ \mathit{compatible\text{-}cons}\ [])\ \rangle \\
\quad & \mathit{fun}\ (\mathit{foldr}\ \mathit{compatible\text{-}cons}\ []) \cdot \mathit{fin\text{-}ordered} \\
\square &
\end{array}
$$

Advantages of program derivation

- Insights are reified as theorems, which can be discovered, proved, collected, and systematically studied.
- These theorems then provide hints and directions to solve other problems.
- A derivation is fun to read, shows the connection between the specification and the derived algorithm, and can likely point out why the algorithm works.
- Beautiful theories about structures and semantics of programs are built up along with development of program derivation.

Disadvantages of program derivation

- The theory is not yet rich enough to cover all classical textbook algorithms.
- The work of laying problems in suitable forms and discovering theorems is extremely hard, and perhaps too abstract for a young field like Algorithm Design.
- That perhaps explains partially why the community is shrinking.
- Relations are inherently harder to manipulate, since there are a great number of operators and associated laws, and inequational reasoning is less deterministic. Even when we step back to functional derivation, low-level calculations can still be very tedious.

Jeremy Gibbons

> *We are interested in extending what can be calculated precisely because we are not that interested in the calculations themselves.*

Calculating Functional Programs. In Keiichi Nakata, editor, *Proceedings of ISRG/SERG Research Colloquium*. School of Computing and Mathematical Sciences, Oxford Brookes University, November 1997. Technical Report CMS-TR-98-01.

Advantages of mechanised formalisation

- Proving theorems by hand is prone to errors. Automatic theorem proving does not make human readers gain intuition and thus confidence. Machine-assisted theorem proving strikes a balance between the two.
- That is, we still write proofs ourselves and become confident about the theorem, while the machine assists us to ensure not a single detail is overlooked, making the proof practically infallible.

Disadvantages of mechanised formalisation

- Formalisation is difficult.
- "Not a single detail is overlooked" means every detail needs to be taken care about, including, say, associativity of natural number addition.
- Formal proofs are unlikely to be written in the traditional essay-like style and often harder to read.

Leslie Lamport

> *The proof style I advocate is a refinement of one, called natural deduction, [..., which] has been viewed primarily as a method of writing proofs in formal logic. What I will describe is a practical method for writing the less formal proofs of ordinary mathematics. It is based on hierarchical structuring — a successful tool for managing complexity.*

How to Write a Proof. *American Mathematical Monthly* 102, 7 (August–September 1993) 600–608. Also appeared in *Global Analysis in Modern Mathematics*, Karen Uhlenbeck, editor. Publish or Perish Press, Houston. Also appeared as SRC Research Report 94.

Robert Pollack

Many people who pursue formal mathematics are seeking the beauty of complete concreteness, which contributes to their own appreciation of the material being formalised, while to many outside the field formalisation is "just filling in the details" of conventional mathematics. But "just" might be infeasible unless serious thought is given to representation of both the logic of formalisation and the mathematics being formalised.

How to Believe a Machine-Checked Proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.

Immanuel Kant

Now it does indeed seem natural that, as soon as we have left the ground of experience, we should, through careful enquiries, assure ourselves as to the foundations of any building that we propose to erect, not making use of any knowledge that we possess without first determining whence it has come, and not trusting to principles without knowing their origin.

Critique of Pure Reason, 1929, Trans. N. Kemp Smith, New York: St. Martin's Press.