0. The Issues
a. (D)SP ISSUE: Declarativity over Algorithms
   - Advantages:
     More compact, clearer (no extraneous detail)
     Problem-oriented (algorithm = implementation)
   - Crucial: best suited to reasoning & calculation
   - Sloppy nomenclature ("smurf") is not harmless:
     E.g., passing off algorithms as "specifications"
     ⇒ loss of opportunities (in modeling, design etc.)
b. BROADER ISSUE: Unified Basis for Engineering
   - Unifying classical Engineering with CS (→ ECE):
     Similar models for circuits and programs
   - Unified basis for "continuous" and "discrete" math

1. A formalism, part I: the language
Four constructs suffice due to functional basis.
0. Identifier: almost anything (a lexical technicality)
   Introduced by a binding: \( i : S \land p \)   Legend:
   new identifier(s) \( i \), set \( S \), (optional) proposition \( p \)
Example: \( n : \mathbb{N} \) is the same as \( n : \mathbb{Z} \land 0 \leq n \)
1. Function application: \( f \ e \) (or \( f(e) \) traditionally)
   Other forms declarable in binding (e.g., \(- \ast + \) )
2. Function abstraction: \( \text{binding} . \text{expr} \)   Axioms:
   Domain: \( v \in D(v : S \land p . e) \equiv v \in S \land p \)
   Map: \( v \in D(v : S \land p . e) \Rightarrow (v : S \land p . e) v = e \)
3. Tupling: \( e, e', e'' \)   Domain: \( D(e, e', e'') = \{0,1,2\} \)
   Map: \( (e, e', e'') 0 = e \) and \( (e, e', e'') 1 = e' \) etc.
   Synthesizes all familiar forms without their defects.

2. A formalism, part II: the rules
a. Main aspect: support calculational reasoning
   \[ expression \ R \ \langle \text{justification} \rangle \ expression' \]
   \[ R' \ \langle \text{justification} \rangle \ expression'' \]
   \( R, R' \) transitive, e.g., arithmetic \( =, \leq, \) logic \( \equiv \Rightarrow \).
   Common in (applications of) algebra, calculus etc.
   Lacking in logical arguments with synkopation, i.e., using symbols (\( \forall, \exists \)) as mere notation, w/o rules
b. Two main elements of the formalism
   0. (Concrete) Generic Functionals
   Generalizes function composition, inverse etc.
   1. Functional Predicate Calculus: practical rules
   Allows engineers to calculate smoothly with \( \forall, \exists \).
   A necessity in CS, a valuable opportunity in EE.

3. Illustration: a few typical rules
a. Generic Functionals: filtering (\( \{ \cdot \} \)) , extension (\( \cdot \))
   \[ D(f \circ P) = \{ x : f \land D P \mid P(x) \} \]
   \[ D(f \circ g) = \{ x : D(f \land D g) \mid (f, g, x) \in D(\ast) \} \]
b. Functional Predicate Calculus: a few rules for \( \forall \)
   Axiom: \( \forall P \equiv P = D P = P \) 1)
   Typical derived rules:
   \[
   \begin{array}{|c|c|}
   \hline
   \text{Rule name} & \text{Point-free form} \\
   \hline
   \text{L-dist.} & \equiv (q \Rightarrow \equiv (q \Rightarrow P)) \\
   \hline
   \text{R-dist.} & \equiv (P \Rightarrow \equiv (P \Rightarrow q)) \\
   \hline
   \text{1-pnt. rule} & \equiv (P \equiv (\forall (\exists q)) \\
   \hline
   \text{Trading} & \equiv (P \equiv (\forall (\exists q)) \\
   \hline
   \end{array}
   \]

4. Examples I: Mathematical Analysis
Illustrates how calculation replaces syncopation
From the definitions (see paper), show by calculation \( \equiv \text{ad } P = P \).
\[
\begin{align*}
\text{closed } P \equiv (\text{"closed"} \equiv (\equiv P)) \\
& \equiv (\forall v : R_{\ast}, v \in R \Rightarrow (v \in R \land (\ast - v < \ast \Rightarrow P v)) \\
& \equiv (\text{Trig.} \equiv (\forall v : \mathbb{R} \Rightarrow (\ast - v < \ast \Rightarrow P v)) \\
& \equiv (\text{Crisps.} \equiv (\forall v : \mathbb{R} \Rightarrow (\ast - v < \ast \Rightarrow P v)) \\
& \equiv (\text{Duality} \equiv (\forall v : \mathbb{R} \Rightarrow (v \in R \land (\ast - v < \ast \Rightarrow P v)) \\
& \equiv (\text{Def. ad} \equiv (\forall v : \mathbb{R} \Rightarrow \equiv (P v \equiv P v) \\
& \equiv (\text{Lemma} \equiv (\forall v : \mathbb{R} \Rightarrow P v \equiv P v) \\
\end{align*}
\]

5. Examples II: Signals and Systems
Illustrates calculation with [generic] functionals
Calculate the response of an LTI system \( x(t) \) see paper \( E \). \( E(t) = e^{t} \).
\[
\begin{align*}
\ast E(t) = (\text{End}) t = (\text{End}) (e^{t}) t \\
= (\text{End}) (e^{t}) t \\
\end{align*}
\]

6. Examples III: Program Dynamics
Illustrates how the same formalism covers programs
0. Describing program behaviour by program equations
State space \( S \); set of commands \( C \); define \( R_{\ast} \) and \( T_{\ast} \) (type \( C \rightarrow S \rightarrow \mathbb{B} \))
\[
\begin{align*}
\text{Command} & = v, c \Rightarrow e, c' \Rightarrow e' \\
\text{State change} & = \text{if } i, b \Rightarrow f i \\
\text{Termination} & = \text{if } i, b \Rightarrow f i \\
\end{align*}
\]
1. Hoare semantics: \( (\{ A \} \ast P) \equiv \forall v, v', R_{\ast}, (s, s') \Rightarrow A \Rightarrow P \)
2. Calculating Dijkstra semantics from Hoare semantics
\[
\begin{align*}
W_{\text{Ho}} & \equiv (\exists v, v', R_{\ast}, (s, s') \Rightarrow A \Rightarrow P \Rightarrow P) \\
& \equiv (\text{Schutte} \Rightarrow (\forall v, v', R_{\ast}, (s, s') \Rightarrow A \Rightarrow P \Rightarrow P) \\
& \equiv (\text{Distrib.} \Rightarrow (\forall v, v', R_{\ast}, (s, s') \Rightarrow A \Rightarrow P \Rightarrow P) \\
& \equiv (\text{Var. change} \Rightarrow (\forall v, v', R_{\ast}, (s, s') \Rightarrow A \Rightarrow P \Rightarrow P) \\
\end{align*}
\]

\( \ast R. \) Bourne is with the Department of Information Technology, Ghent University, Ghent, Belgium.