What are these control hierarchies?

Chung-chieh Shan
Rutgers University

29 May 2011

theory $\rightarrow \times + ?$ practice
What are these control hierarchies?

Chung-chieh Shan
Rutgers University

29 May 2011
Dijkstra's case against the GOTO statement [37] has been mostly interpreted in the negative, as the forceful denunciation of a programming sin. It seems to us, though, that his message, 40 years on, can also be understood as an invitation to mindfulness when programming.

Indeed consider a compiler from a structured language (e.g., one with while loops and conditional commands) to an unstructured language (e.g., one with labels and with conditional and unconditional jumps): on the one hand, this compiler yields programs that use GOTO statements; on the other hand, as denotations of structured programs, these unstructured programs only use GOTO statements to implement the control structures of structured programs. In that light, Dijkstra's implicit message is not so much that GOTO statements should be considered harmful, no matter what, than one should be mindful about staying in or straying from the image of the compiler when programming in the unstructured language:

In fact, finding oneself straying with good reason is a clear indication that a useful control construct is missing in the source language. For example, C and Pascal programmers conducted the use of GOTO for error cases because these languages lack an exception mechanism.

Dijkstra's implicit message applies to at least two situations involving a program transformation and its left inverse:

CPS transformation. When programming in continuation-passing style, one should be mindful of the continuation identifiers and of the parameters of continuations to stay in the image of the CPS transformation [23, 27]:

Yet programmers use “the extra expressive power of CPS” to stray with good reason:

- for a simple example, not using the current continuation identifier prevents the computation from continuing and therefore has the effect of aborting it; this effect can be obtained in direct style by adding an “abort” control operator;

\[
\alpha \rightarrow \beta
\]

\[
\Downarrow
\]

\[
\text{CPS}
\]

\[
\alpha \times (\beta \rightarrow \omega) \rightarrow \omega
\]
Dijkstra's case against the GOTO statement [37] has been mostly interpreted in the negative, as the forceful denunciation of a programming sin. It seems to us, though, that his message, 40 years on, can also be understood as an invitation to mindfulness when programming.

Indeed consider a compiler from a structured language (e.g., one with while loops and conditional commands) to an unstructured language (e.g., one with labels and unconditional jumps): on the one hand, this compiler yields programs that use GOTO statements; on the other hand, as denotations of structured programs, these unstructured programs only use GOTO statements to implement the control structures of structured programs. In that light, Dijkstra's implicit message is not so much that GOTO statements should be considered harmful, no matter what, than one should be mindful about staying in or straying from the image of the compiler when programming in the unstructured language:

In fact, finding oneself straying with good reason is a clear indication that a useful control construct is missing in the source language. For example, C and Pascal programmers condone the use of GOTO for error cases because these languages lack an exception mechanism.

Dijkstra's implicit message applies to at least two situations involving a program transformation and its left inverse:

- **CPS transformation.** When programming in continuation-passing style, one should be mindful of the continuation identifiers and of the parameters of continuations to stay in the image of the CPS transformation [23, 27]:

\[
\alpha \to \beta \\
\xrightarrow{\text{CPS}} \\
\alpha \times (\beta \to \omega) \to \omega
\]

Yet programmers use "the extra expressive power of CPS" to stray with good reason:

- for a simple example, not using the current continuation identifier prevents the computation from continuing and therefore has the effect of aborting it; this effect can be obtained in direct style by adding an "abort" control operator;

Danvy & Millikin
Dijkstra's case against the GOTO statement [37] has been mostly interpreted in the negative, as the forceful denunciation of a programming sin. It seems to us, though, that his message, 40 years on, can also be understood as an invitation to mindfulness when programming. Indeed consider a compiler from a structured language (e.g., one with while loops and conditional commands) to an unstructured language (e.g., one with labels and unconditional jumps): on the one hand, this compiler yields programs that use GOTO statements; on the other hand, as denotations of structured programs, these unstructured programs only use GOTO statements to implement the control structures of structured programs. In that light, Dijkstra's implicit message is not so much that GOTO statements should be considered harmful, no matter what, than one should be mindful about staying in or straying from the image of the compiler when programming in the unstructured language: 

In fact, finding oneself straying with good reason is a clear indication that a useful control construct is missing in the source language. For example, C and Pascal programmers done the use of GOTO for error cases because these languages lack an exception mechanism.

Dijkstra's implicit message applies to at least two situations involving a program transformation and its left inverse:

**CPS transformation.** When programming in continuation-passing style, one should be mindful of the continuation identifiers and of the parameters of continuations to stay in the image of the CPS transformation [23, 27]:

Yet programmers use "the extra expressive power of CPS" to stray with good reason:

- for a simple example, not using the current continuation identifier prevents the computation from continuing and therefore has the effect of aborting it; this effect can be obtained in direct style by adding an "abort" control operator;

Logic guides codifying the pattern:

- Decompose positive vs negative expressions/variables
  function vs context connectives/constructions

- Decompose \( \omega_1 \rightarrow \beta/\omega_2 \) into \((\beta \rightarrow \bullet \omega_1) \triangleright \omega_2\)
  \(\alpha/\omega_1 \rightarrow \beta/\omega_2 \) into ((\(\alpha \rightarrow \beta) \rightarrow \bullet \omega_1) \triangleright \omega_2\)

- Negatives are polymorphic in the answer type;
  positives are specific in the answer type?

Make nonsense impossible, common sense easy (reflection)?
Dijkstra's case against the GOTO statement [37] has been mostly interpreted in the negative, as the forceful denunciation of a programming sin. It seems to us, though, that his message, 40 years on, can also be understood as an invitation to mindfulness when programming. Indeed consider a compiler from a structured language (e.g., one with while loops and conditional commands) to an unstructured language (e.g., one with labels and with conditional and unconditional jumps): on the one hand, this compiler yields programs that use GOTO statements; on the other hand, as denotations of structured programs, these unstructured programs only use GOTO statements to implement the control structures of structured programs. In that light, Dijkstra's implicit message is not so much that GOTO statements should be considered harmful, no matter what, than one should be mindful about staying in or straying from the image of the compiler when programming in the unstructured language.

In fact, finding oneself straying with good reason is a clear indication that a useful control construct is missing in the source language. For example, C and Pascal programmers condone the use of GOTO for error cases because these languages lack an exception mechanism.

Dijkstra's implicit message applies to at least two situations involving a program transformation and its left inverse:

1. **CPS transformation.** When programming in continuation-passing style, one should be mindful of the continuation identifiers and of the parameters of continuations to stay in the image of the CPS transformation [23, 27]:

   \[
   \begin{align*}
   \alpha &\rightarrow \beta \\
   \downarrow \text{CPS} \\
   \alpha \times (\beta \rightarrow \omega) &\rightarrow \omega \\
   \downarrow \text{CPS} \\
   \alpha \times (\beta \times (\omega \rightarrow \varpi) \rightarrow \varpi) \times (\omega \rightarrow \varpi) &\rightarrow \varpi
   \end{align*}
   \]

2. **Logic guides codifying the pattern:** Zeilberger ▶ Decompose positive vs negative expressions/variables
   function vs context connectives/constructions ▶ Decompose \( \alpha \rightarrow \beta \) into \( \alpha \rightarrow (\beta \rightarrow \omega) \rightarrow \omega \)
   ▶ Negatives are polymorphic in the answer type; positives are specific in the answer type?

Make nonsense impossible, common sense easy (reflection)?

Danvy & Filinski

\[ \alpha \rightarrow \beta \]

\[ \downarrow \text{CPS} \]

\[ \alpha \times (\beta \rightarrow \omega) \rightarrow \omega \]

\[ \downarrow \text{CPS} \]

\[ \alpha \times (\beta \times (\omega \rightarrow \varpi) \rightarrow \varpi) \times (\omega \rightarrow \varpi) \rightarrow \varpi \]
What is the total population of the ten largest capitals in the US? Answering these types of complex questions compositionally involves first mapping the questions into logical forms (semantic parsing).

Liang, Jordan & Klein
What is the total population of the ten largest capitals in the US? Answering these types of complex questions compositionally involves first mapping the questions into logical forms (semantic parsing).

The filtering function $F$ rules out improperly-typed trees . . . To further reduce the search space . . .

Think of DCS as a higher-level programming language tailored to natural language, which results in programs which are much simpler than the logically-equivalent lambda calculus formulae.

Liang, Jordan & Klein
Alice knows Bob

Alice :: E
Bob :: E
know :: E → E → Bool

Alice \$ (know \$ Bob) :: Bool
Alice knows Bob

Alice :: E
Bob :: E
know :: E → E → Bool

Alice $ (know $ Bob) :: Bool

\[
\frac{E \rightarrow E \rightarrow \text{Bool}}{E \rightarrow \text{Bool}} \frac{\text{know}}{E} \frac{\neg \text{Bob}}{E} \frac{\neg}{\$} \frac{\text{Alice}}{E} \frac{E}{E \rightarrow \text{Bool}} \frac{\$}{\text{Bool}}
\]
Alice knows everyone

Alice :: E
Bob :: E
know :: E → E → Bool

type M α = (α → Bool) → Bool
everyone :: M E
everyone c = all c [Alice, Bob, ..]
Alice knows everyone

Alice :: E
Bob :: E
know :: E → E → Bool

\[
\text{type } M \alpha = (\alpha \to \text{Bool}) \to \text{Bool}
\]

everyone :: M E
everyone \ c = \text{all } c [\text{Alice, Bob, . . .}]

\[
\begin{array}{c}
\frac{E}{M E} \quad \frac{E \to E \to \text{Bool}}{M(E \to E \to \text{Bool})} \quad \frac{\text{return}}{\text{M(E \to \text{Bool})}} \quad \frac{\text{everyone}}{\text{M(E \to \text{Bool})}} \\
\frac{\text{M Bool}}{\text{Bool}} \quad \frac{\text{($id$)}}{\text{($id$)}}
\end{array}
\]

Barker, de Groote, . . .
Alice knows everyone

Alice :: E
Bob :: E
know :: E → E → Bool

type \( M \alpha = (\alpha \to \text{Bool}) \to \text{Bool} \)

everyone :: M E
someone :: M E
most :: [E] → M E

Someone knows everyone
Alice knows everyone

Alice :: E  
Bob :: E  
know :: E → E → Bool  

type M α = (α → Bool) → Bool  

every :: [E] → ME  
some :: [E] → ME  
most :: [E] → ME  
logician :: [E]  
programmer :: [E]  

Someone knows everyone  
Most logicians know some programmer
Alice knows everyone

Alice :: E
Bob :: E
know :: E → E → Bool

type $M\alpha = (\alpha \rightarrow \text{Bool}) \rightarrow \text{Bool}$

every :: [E] → $M\ E$
some :: [E] → $M\ E$
most :: [E] → $M\ E$
logician :: [E]
programmer :: [E]
from :: [E] → E → [E]

Someone knows everyone
Most logicians know some programmer
Most logicians know some programmer from Novi Sad
Inverse scope: Someone knows everyone

\[
\begin{array}{c}
\text{ME} \\
\text{someone} \\
\begin{array}{c}
\text{ME} \\
\text{return} \\
\text{everyone} \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{M(}E \rightarrow E \rightarrow \text{Bool}) \\
\text{know} \\
\text{M(E \rightarrow E \rightarrow \text{Bool})} \\
\text{return} \\
\text{M(E \rightarrow \text{Bool})} \\
\text{liftM2 (\$)} \\
\text{M(\text{Bool})} \\
\text{liftM2 (\$)} \\
\text{M(\text{Bool})} \\
\text{($ id$)} \\
\text{Bool}
\end{array}
\]
Inverse scope: Someone knows everyone

\[
\begin{align*}
\text{someone} & \quad \text{know} \\
E \to E \to \text{Bool} & \quad \text{return} \\
M(E \to E \to \text{Bool}) & \quad \text{everyone} \\
M\left(E \to \text{Bool}\right) & \quad \text{liftM2 (\$)} \\
M \text{ Bool} & \\
\text{Bool (\$ id)} & \\
\text{someone} & \quad \text{know} \\
E \to E \to \text{Bool} & \quad \text{return} \\
M(E \to E \to \text{Bool}) & \quad \text{return} \\
M(M(E \to E \to \text{Bool})) & \quad \text{everyone} \\
M(M(M(E \to \text{Bool}))) & \quad \text{liftM2(liftM2 (\$))} \\
M(\text{M Bool}) & \\
\text{M Bool} & \quad \text{liftM (\$ id)} \\
\text{Bool} & \\
\text{Bool (\$ id)} &
\end{align*}
\]
Inverse linking: Combining hierarchies?

some programmer from Novi Sad
some programmer from every city

\[
\begin{align*}
\text{[E] } & \rightarrow \text{ ME} & \text{every} & \text{[E]} & \text{city} \\
\text{ME} & \rightarrow \text{ME} & \text{M}[E] & \text{??}
\end{align*}
\]
Inverse linking: Combining hierarchies?

some programmer from Novi Sad
some programmer from every city

\[
\begin{align*}
\text{[E]} & \rightarrow \text{ME} \quad \text{every} \quad \text{[E]} \quad \text{city} \\
\text{ME} & \quad \text{M[E]} \\
\text{ME} \quad \text{??}
\end{align*}
\]

Someone knows everyone
Most logicians know some programmer
Most logicians know some programmer from every city
From in-situ quantification to in-situ let-insertion

Write domain-specific code generators in multilevel languages
Nielsen & Nielson, Taha

Continuations for code generation, especially let-insertion
Danvy & Filinski, Bondorf, Lawall & Danvy

\[ \cdots \pm \cdots \leadsto \text{let } t_1 = \cdots \text{ and } t_2 = \cdots \text{ and } t_3 = \cdots \text{ in } \cdots \]
From in-situ quantification to in-situ let-insertion

Write domain-specific code generators in multilevel languages

Nielson & Nielson, Taha

Continuations for code generation, especially let-insertion

Danvy & Filinski, Bondorf, Lawall & Danvy

\[ \ldots + \ldots \leadsto \text{let } t_1 = \ldots \text{ and } t_2 = \ldots \text{ and } t_3 = \ldots \text{ in } \ldots \]

\[
(\lambda e : \langle \alpha \rangle.
\lambda c : \langle \alpha \rangle \to \langle \beta \rangle.
\text{let } x = e \text{ in } cx)
: \langle \alpha \rangle \to (\langle \alpha \rangle \to \langle \beta \rangle) \to \langle \beta \rangle
\]
From in-situ quantification to in-situ let-insertion

Write domain-specific code generators in multilevel languages
Nielson & Nielson, Taha

Continuations for code generation, especially let-insertion
Danvy & Filinski, Bondorf, Lawall & Danvy

\[ \cdots \pm \cdots \leadsto \quad \text{let } t_1 = \cdots \text{ and } t_2 = \cdots \text{ and } t_3 = \cdots \text{ in } \cdots \]

\[
(\lambda e : \langle \alpha \rangle.
\begin{align*}
\lambda c : \langle \alpha \rangle & \to \langle \beta \rangle. \\
\text{let } x = e \text{ in } cx
\end{align*}
) \\
: \langle \alpha \rangle \to (\langle \alpha \rangle \to \langle \beta \rangle) \to \langle \beta \rangle
\]
From in-situ quantification to in-situ let-insertion

Write domain-specific code generators in multilevel languages
Nielson & Nielson, Taha

Continuations for code generation, especially let-insertion
Danvy & Filinski, Bondorf, Lawall & Danvy

\[
\cdots \mathbin{\vdash} \cdots \; \leadsto \quad \text{let } t_1 = \cdots \text{ and } t_2 = \cdots \text{ and } t_3 = \cdots \text{ in } \cdots
\]

\[
(\lambda e : \langle \alpha \rangle^\pi. \\
\lambda c : \langle \alpha \rangle \to \langle \beta \rangle. \\
\text{let } x = e \text{ in } c x) \\
: \langle \alpha \rangle \to (\langle \alpha \rangle \to \langle \beta \rangle) \to \langle \beta \rangle
\]
From in-situ quantification to in-situ let-insertion

Write domain-specific code generators in multilevel languages
Nielson & Nielson, Taha

Continuations for code generation, especially let-insertion
Danvy & Filinski, Bondorf, Lawall & Danvy

\[
\cdots + \cdots \leadsto \text{let } t_1 = \cdots \text{ and } t_2 = \cdots \text{ and } t_3 = \cdots \text{ in } \cdots
\]

\[
(\lambda e : \langle \alpha \rangle^\pi.
\lambda c : \forall \rho. \langle \alpha \rangle^\pi,\rho \rightarrow \langle \beta \rangle^\pi,\rho.
\text{let } x = e \text{ in } cx)
: \langle \alpha \rangle \rightarrow (\langle \alpha \rangle \rightarrow \langle \beta \rangle) \rightarrow \langle \beta \rangle
\]
From in-situ quantification to in-situ let-insertion

Write domain-specific code generators in multilevel languages

Nielson & Nielson, Taha

Continuations for code generation, especially let-insertion

Danvy & Filinski, Bondorf, Lawall & Danvy

\[
\cdots \pm \cdots \leadsto \text{let } t_1 = \cdots \text{ and } t_2 = \cdots \text{ and } t_3 = \cdots \text{ in } \cdots
\]

\[
(\lambda e : \langle \alpha \rangle^{\pi,\sigma}. \lambda c : \forall \rho. \langle \alpha \rangle^{\pi,\sigma,\rho} \rightarrow \langle \beta \rangle^{\pi,\sigma,\rho}. \text{let } x = e \text{ in } cx) : \forall \sigma. \langle \alpha \rangle^{\pi,\sigma} \rightarrow (\forall \rho. \langle \alpha \rangle^{\pi,\sigma,\rho} \rightarrow \langle \beta \rangle^{\pi,\sigma,\rho}) \rightarrow \langle \beta \rangle^{\pi,\sigma}
\]
From in-situ quantification to in-situ let-insertion

Write domain-specific code generators in multilevel languages
Nielson & Nielson, Taha

Continuations for code generation, especially let-insertion
Danvy & Filinski, Bondorf, Lawall & Danvy

\[ \cdots \; \vdash \; \cdots \; \leadsto \; \text{let } t_1 = \cdots \; \text{and } t_2 = \cdots \; \text{and } t_3 = \cdots \; \text{in } \cdots \]

\[
(\lambda e : \langle \alpha \rangle_{\pi, \sigma}. \\
\lambda c : \forall \rho. \; \langle \alpha \rangle_{\pi, \sigma, \rho} \rightarrow \langle \beta \rangle_{\pi, \sigma, \rho}. \\
\text{let } x = e \text{ in } cx)
\]

: \forall \sigma. \; \langle \alpha \rangle_{\pi, \sigma} \rightarrow (\forall \rho. \; \langle \alpha \rangle_{\pi, \sigma, \rho} \rightarrow \langle \beta \rangle_{\pi, \sigma, \rho}) \rightarrow \langle \beta \rangle_{\pi, \sigma}

Systematic translation, Kameyama, Kiselyov & Shan

but how does it fit CPS?
Want let-insertion at different scopes.
Summary

Hierarchy 0: composing monad transformers
Hierarchy 1: composing monads (applicative functors)
Hierarchy 2: additional polymorphism at each level

Make nonsense impossible, common sense easy?