# Presentation Type Theory

## Direct Style Transform*

Camil Staps, June 2018

- Why DS?
  - In the 1980s everybody used CPS in compilers.
  - Trampolining code is only useful in few cases.
  - The CPS language allows ~~fewer β~~ more reductions allowing for more optimizations.
  - With DS, these optimizations can be mapped back to the CBV language.
- Goal: from the CPS language back to the original 'DS' language.
  - Because we may have applied optimizations, this CPS language is closed under reduction.
- For the CPS language, define the grammar with 5 kinds of pseudo-objects:
  - Computations; receiving a continuation

    $\mathcal{M} ::= x \mid \boxed{\lambda k.\mathcal{A}} \mid \mathcal{V}\mathcal{N}$

    recall $\langle x\rangle = \lambda k.xk$

    continuation

    $\lambda x.\lambda k.\mathcal{A}$ or $\lambda\alpha.\lambda k.\mathcal{A}$ receive one to goto $\lambda k.\mathcal{A}$

    type is double-negated
  - Values; ~~the~~ source abstractions

    $\mathcal{V} ::= \lambda x.\lambda k.\mathcal{A} \mid \lambda\alpha.\lambda k.\mathcal{A}$

    type is not double-negated
  - Arguments; arguments to source abstractions

    $\mathcal{N} ::= \boxed{\lambda k.\mathcal{A}} \mid C$
  - Answers; result of applying a continuation to a computation

    $\mathcal{A} ::= \mathcal{K}\mathcal{V} \mid \mathcal{M}\mathcal{K}$

    type $\bot$
  - Continuations: identifier k or an abstraction receiving a value

    $\mathcal{K} ::= \boxed{k} \mid \lambda y.y\mathcal{N}\mathcal{K}$

    fun fact: only one is needed
- To see why $\lambda k.\mathcal{A}$ is both a computation and a value:

$$\begin{aligned}\langle(\lambda x.x)\,z\rangle &= \lambda k.\,(\lambda k'.k'(\lambda x.\lambda k''.xk''))(\lambda y.y\,(\lambda k'.xk')k)\\ &\to \lambda k.\,(\lambda x.\lambda k'.xk')\boxed{(\lambda k'.xk')}k\\ &\to \lambda k.\,\boxed{(\lambda k'.xk')}k\\ &\to \lambda k.xk\end{aligned}$$

argument

computation.

~~I.e., it is a result of $\langle x\rangle = \lambda k.xk$~~.

* Barthe, Hatcliff & Sørensen, 1999. 'CPS Translations and Applications: The Cube and Beyond'. Higher-Order and Symbolic Computation 12, 125-170, sec. 4-4.1.

- To compare the CPS transform:

comp

$\langle x \rangle = \lambda k.\, x\, k$ — answer ~~is a computation / value~~

$\langle \lambda x. O \rangle = \lambda k.\, k\, (\lambda x. \langle O \rangle)$ — value

argument

$\langle O\, O' \rangle = \lambda k.\, \langle O \rangle\, (\lambda y.\, y\, \langle O' \rangle\, k)$ — continuation

continuation

$\langle \lambda \alpha. O \rangle = \lambda k.\, k\, (\lambda \alpha. \langle O \rangle)$ — value

$\langle O C \rangle = \lambda k.\, \langle O \rangle\, (\lambda y.\, y\, \langle C \rangle\, k)$

has to be an argument because $\mathcal{K} ::= \lambda y.\, y\, \mathcal{N}\mathcal{K}$

- Constructors in the CPS language are:

$$C ::= \alpha \mid \lambda x. C_2 \mid \lambda \alpha. C_2 \mid C\mathcal{N} \mid \Pi x{:}\neg\neg C_1. \neg\neg C_2 \mid \Pi \alpha{:}K. \neg\neg C.$$

These relate directly to the domain of the CPS transform for constructors.

- Kinds: $K ::= * \mid \Pi x{:}\neg\neg C. K \mid \Pi \alpha{:}K_1. K_2$

Also relate directly. ML-like strategy, no 'serious' polymorphism

- Contexts: $\Gamma ::= \bot{:}* \mid \Gamma, x{:}\neg\neg C \mid \Gamma, \alpha{:}K$

always! computations are always negated

- The legal objects are defined with derivation rules:

$\Vdash_{com}$, $\Vdash_{val}$, $\Vdash_{arg}$, $\Vdash_{ans}$, $\Vdash_{con}$.

<u>Computations:</u>

(1) $$\frac{\Gamma \Vdash_{con} C : *}{\Gamma, x{:}\neg\neg C \Vdash_{com} x : \neg\neg C}$$ if $x \notin \Gamma$ and $x \in$ Computation-ident[CPS]

I.e., variables must be simple-kinded.

<u>not</u> the angular brackets of the transform!

(2) $$\frac{\langle \Gamma \rangle \langle k{:}\neg C \rangle \Vdash_{ans} \mathcal{A} : \bot}{\Gamma \Vdash_{com} \lambda k. \mathcal{A} : \neg\neg C}$$ If k is a continuation ~~with~~ for type C making an answer well-typed, $\lambda k. \mathcal{A} : \neg\neg C$.

Note: k has special treatment, because ~~[illegible]~~ only one k is needed.*

Note 2: a continuation always contains some k, and an answer always contains some continuation, so the $k{:}\neg C$ <u>must</u> be used.

* This is not required but just gives simpler rules (?)

(3) $$\frac{\Gamma \Vdash_{val} \mathcal{V} : \Pi\alpha{:}K_1 . \neg\neg C_2 \qquad \Gamma \Vdash_{arg} C_1 : K_1}{\Gamma \Vdash_{com} \mathcal{V} C_1 : \neg\neg C_2 \{\alpha := C_1\}}$$ (straightforward)

(4) $$\frac{\Gamma \Vdash_{val} \mathcal{V} : \Pi x{:}\neg\neg C_1 . \neg\neg C_2 \qquad \Gamma \Vdash_{arg} (\lambda k.\mathcal{A}) : \neg\neg C_1}{\Gamma \Vdash_{com} \mathcal{V}(\lambda k.\mathcal{A}) : \neg\neg C_2 \{x := \lambda k.\mathcal{A}\}}$$ (straightforward)

Arguments:

(5) $$\frac{\langle\Gamma\rangle \langle k : \neg C\rangle \Vdash_{ans} \mathcal{A} : \bot}{\Gamma \Vdash_{arg} \lambda k.\mathcal{A} : \neg\neg C}$$ (as in rule (2))

$$\frac{\Gamma \Vdash_{con} C : K}{\Gamma \Vdash_{arg} C : K}$$ (6)

Values:

(7) $$\frac{\Gamma, x : \neg\neg C_1 \Vdash_{com} \lambda k.\mathcal{A} : \neg\neg C_2 \qquad \Gamma \Vdash_{con} \Pi x{:}\neg\neg C_1 . \neg\neg C_2 : *}{\Gamma \Vdash_{val} \lambda x.\lambda k.\mathcal{A} : \cancel{\Pi x : *}\ \Pi x{:}\neg\neg C_1 . \neg\neg C_2}$$

I.e., x is some computation for type $C_1$, and $\overset{\lambda k.\mathcal{A}}{\cancel{C_2}}$ (given x) for $C_2$. Then $\lambda x.\lambda k.\mathcal{A}$ is a function type, as long as it is simple-kinded.

For constructor abstractions the rule is similar (though no double-negation, again because of the ML-like strategy):

(8) $$\frac{\Gamma, \alpha : K_1 \Vdash_{com} \lambda k.\bot : \neg\neg C_2 \qquad \Gamma \Vdash_{con} \Pi\alpha{:}K_1 . \neg\neg C_2 : *}{\Gamma \Vdash_{val} \lambda\alpha.\lambda k.\mathcal{A} : \Pi\alpha{:}K_1 . \neg\neg C_2}$$

Again, the value itself must be simple-kinded. ~~With~~ The left antecedent ~~We do not need to check that $\mathcal{A} : \bot$~~ is probably incorrect and should read $\lambda k.\mathcal{A} : \neg\neg C_2$; we know that $\bot : * \neq \neg\neg C_2$.

Answers:

We may assume k is of some continuation type, i.e. $\neg C_1$. (The actual $C_1$ depends on the transformed program. I guess that this is needed because typing is undecidable, hence we cannot derive $C_1$ (?))

(9) $$\frac{\langle\Gamma\rangle\langle k : \neg C_1\rangle \Vdash_{cnt} \mathcal{K} : \neg C_2 \qquad \Gamma \Vdash_{val} \mathcal{V} : C_2}{\langle\Gamma\rangle\langle k : \neg C_1\rangle \Vdash_{ans} \mathcal{K}\mathcal{V} : \bot}$$

Other than the handling of k, this is an ordinary application rule.

$$(10)\quad \frac{\Gamma \Vdash_{com} \mathcal{M} : \neg\neg C_2 \qquad \langle\Gamma\rangle\langle k:\neg C_1\rangle \Vdash_{cnt} \mathcal{K} : \neg C_2}{\langle\Gamma\rangle\langle k:\neg C_1\rangle \Vdash_{ans} \mathcal{M}\mathcal{K} : \bot}$$

note the typo in the paper, copied from rule (9)

Compared to (9), we only have two added negations $(\mathcal{M}:\neg\neg C_2$ and $\mathcal{K}:\neg C_2)$, due to the types of computations, continuations and values.

<u>Continuations:</u>

We do restrict k to be the continuation of a simply-kinded type:

$$(11)\quad \frac{\Gamma \Vdash_{con} C : *}{\langle\Gamma\rangle\langle k:\neg C\rangle \Vdash_{cnt} k : \neg C}$$

For transformed applications we get:

$$(12)\quad \frac{\begin{array}{l}\Gamma \Vdash_{con} \Pi x:\neg\neg C_1 . \neg\neg C_2 : * \\ \Gamma \Vdash_{arg} \lambda k.\mathcal{A} : \neg\neg C_1 \\ \langle\Gamma\rangle\langle k:\neg C_0\rangle \Vdash_{cnt} \mathcal{K} : \neg C_2\{x := \lambda k.\mathcal{A}\}\end{array}}{\langle\Gamma\rangle\langle k:\neg C_0\rangle \Vdash_{cnt} (\lambda y. y\,(\lambda k.\mathcal{A})\mathcal{K}) : \neg\Pi x:\neg\neg C_1 . \neg\neg C_2} \quad \text{if } y \notin \Gamma \text{ and } y \text{ is a computation identifier}$$

Note that y does not appear in the context. Its type would be $\Pi x:\neg\neg C_1 . \neg\neg C_2$; then applied with $x := \lambda k.\mathcal{A}$ and $\mathcal{K}:\neg C_2$ yields a $\bot$. Since y is immediately applied, it need not appear in the context; compare to a hardcoded language construct.

The case for constructor applications is similar:

$$(13)\quad \frac{\begin{array}{l}\Gamma \Vdash_{con} \Pi \alpha:K_1 . \neg\neg C_2 : * \\ \Gamma \Vdash_{arg} C_1 : K_1 \\ \langle\Gamma\rangle\langle k:\neg C_0\rangle \Vdash_{cnt} \mathcal{K} : \neg C_2\{\alpha := C_1\}\end{array}}{\langle\Gamma\rangle\langle k:\neg C_0\rangle \Vdash_{cnt} (\lambda y. y\, C_1 \mathcal{K}) : \neg\Pi \alpha:K_1 . \neg\neg C_2} \quad \text{if } y \notin \Gamma \text{ and } y \text{ is a computation identifier}$$

Here, $y : \Pi\alpha:K_1 . \neg\neg C_2$; with $C_1 : K_1$ and $\mathcal{K}:\neg C_2\{\alpha := C_1\}$ this gives a $\bot$.

The rules for constructors and kinds ensure that everything is well-kinded, but since we assumed an ML-like strategie (no 'serious' polymorphism) these rules are much like the conventional typing rules.

In addition to these rules, we have weakening and conversion, e.g.

$$(\text{weak.}) \quad \frac{\Gamma \Vdash_{\vartheta} A : B \qquad \Gamma \Vdash_{knd} K : \square}{\Gamma, \alpha : K \Vdash_{\vartheta} A : B} \quad \text{if } \alpha \notin \Gamma \text{ and } \alpha \text{ is a constructor identifier}$$

$\vartheta \in \{com, val, arg, con, knd\}$

$$(\text{weak.}) \quad \frac{\Gamma \Vdash_{\vartheta} A : B \qquad \Gamma \Vdash_{con} C : *}{\Gamma, x : \neg\neg C \Vdash_{\vartheta} A : B} \quad \text{if } x \notin \Gamma \text{ and } x \text{ is a computation identifier}$$

$$(\text{conv.}) \quad \frac{\Gamma \Vdash_{con} C : K \qquad \Gamma \Vdash_{knd} K' : \square}{\Gamma \Vdash_{con} C : K'} \quad \text{if } K =_{\beta} K'$$

$$(\text{conv.}) \quad \frac{\Gamma \Vdash_{com} \mathcal{M} : \neg\neg C \qquad \Gamma \Vdash_{con} C' : *}{\Gamma \Vdash_{com} \mathcal{M} : \neg\neg C'} \quad \text{if } C =_{\beta} C'$$

With the legal terms defined we can show a number of properties:

① Legal CPS terms are legal terms in the DF λ-cube.

$$\Gamma \Vdash_{\vartheta} A : B \quad \text{implies} \quad \Gamma \Vdash A : B \qquad \vartheta \in \{com, val, arg, con, knd\}$$

$$\langle \Gamma \rangle \langle k : \neg C \rangle \Vdash_{\varphi} A : B \quad \text{implies} \quad \Gamma, k : \neg C \Vdash A : B \qquad \varphi \in \{ans, cnt\}$$

The proof is a simple case analysis of the derivation rules. For instance, we can derive $\langle \Gamma \rangle \langle k : \neg \text{Int} \rangle \Vdash_{ans} x\ k : \bot$ with $\Gamma = \bot : *, \text{Int} : *, x : \neg\neg\text{Int}$ using rules (10) and (11). In the DF λ-cube we have a $(\to)$ rule to derive $\Gamma, k : \neg\text{Int} \Vdash x\ k : \bot$ :

$$(\to) \quad \frac{\Gamma \Vdash O : (\Pi x : C . C') \qquad \Gamma \Vdash O' : C}{\Gamma \Vdash O\, O' : C' \{x := O'\}}$$

Recall that $C \to C' \equiv \Pi x : C . C'$ if $x$ is not free in $C'$, i.e. $\neg\text{Int} \equiv \Pi x : \text{Int} . \bot$.

② The CPS language contains the image of the CPS translation.

$\Gamma \Vdash O : C$ implies $\langle\!\langle \Gamma \rangle\!\rangle \Vdash_{com} \langle O \rangle : \langle\!\langle C \rangle\!\rangle = \neg\neg \langle C \rangle$

$\Gamma \Vdash C : K$ implies $\langle\!\langle \Gamma \rangle\!\rangle \Vdash_{con} \langle C \rangle : \langle\!\langle K \rangle\!\rangle = \langle K \rangle$

$\Gamma \Vdash K : \square$ implies $\langle\!\langle \Gamma \rangle\!\rangle \Vdash_{knd} \langle K \rangle : \langle\!\langle \square \rangle\!\rangle = \square$

The proof analyses the cases of the CPS transform.

Essentially, each DF $\lambda$-cube derivation rule has a corresponding CPS derivation rule (the correspondence transcendents the CPS transform). Thus the typing derivation can be mapped to the derivation of the CPS term, modulo administration. See the example under ① for an example; recall that $\langle x \rangle = \lambda k.\, x k$.

③ Legal CPS terms are closed under reduction.

$\Gamma \Vdash_\vartheta A : B$ and $A \to_\beta A'$ imply $\Gamma \Vdash_\vartheta A' : B$ $\quad \vartheta \in \{com, val, arg, con, knd\}$

$\langle \Gamma \rangle \langle k : \neg C \rangle \Vdash_\varphi A : B$ and $A \to_\beta A'$ imply $\langle \Gamma \rangle \langle k : \neg C \rangle \Vdash_\varphi A' : B$ $\quad \varphi \in \{ans, cnt\}$

The proof analyses all redexes that can occur in CPS objects.

- $\mathcal{V}\mathcal{N}$ is always a redex. If $\mathcal{V} = \lambda x.\lambda k.\mathcal{A}$ and $\mathcal{N} = \lambda k.\mathcal{A}'$ this works out since $\lambda k.\mathcal{A}$ is a computation as well, hence $\mathcal{V}\mathcal{N} \to \lambda k.\mathcal{A}\{x := \lambda k.\mathcal{A}'\}$ is a computation. If $\mathcal{V} = \lambda\alpha.\lambda k.\mathcal{A}$ and $\mathcal{N} = C$, we get $\mathcal{V}\mathcal{N} \to \lambda k.\mathcal{A}\{\alpha := C\}$ which is also a computation (both by a substitution lemma).
  If $\mathcal{V} = \lambda k.\lambda k.\mathcal{A}$ and $\mathcal{N} = C$, we need a typing derivation for $\mathcal{V}\mathcal{N}$. By (3), $\mathcal{V} : \Pi\alpha{:}K_1.\neg\neg C_2$ for some $K_1$, $C_2$. But this can only be the case if $\mathcal{V} = \lambda\alpha.\lambda k.\mathcal{A}$. Hence $\mathcal{V}\mathcal{N} = (\lambda x.\lambda k.\mathcal{A})C$ is illegal. $\mathcal{V}\mathcal{N} = (\lambda\alpha.\lambda k.\mathcal{A})(\lambda k.\mathcal{A}')$ is similar.
- $\mathcal{K}\mathcal{V}$ is a redex if $\mathcal{K} = \lambda y.\, y\mathcal{N}\mathcal{K}'$. Then $\mathcal{K}\mathcal{V} \to \mathcal{V}\mathcal{N}\mathcal{K}'$. This is similar to the above case: it works out if $\mathcal{V} = \lambda x.\lambda k.\mathcal{A}$ and $\mathcal{N} = \lambda k.\mathcal{A}'$ or $\mathcal{V} = \lambda\alpha.\lambda k.\mathcal{A}$ and $\mathcal{N} = C$; the other two cases are rejected by $\Vdash_{val}$.
- $\mathcal{M}\mathcal{K}$ is a redex if $\mathcal{M} = \lambda k.\mathcal{A}$. Then $\mathcal{M}\mathcal{K} \to \mathcal{A}\{k := \mathcal{K}\}$. This works out, since (5) allows us to pick any type for $k$.

- Where does this leave us?

- We still need to do the actual DS transform. This is section 4.2. The problem is that we have to translate CPS terms that are not in the image of the CPS transform. For instance, $\lambda y.\, y\,\mathcal{N}\mathcal{K}$ is a CPS term, but only occurs in context in the transform; e.g. $\langle OO' \rangle = \lambda k. \langle O \rangle (\lambda y.\, y \langle O' \rangle k)$. Thus we cannot just invert the CPS transform.

- The solution is to transform to evaluation contexts. Thus we get:*

$$\rangle \lambda y.\, y\,\mathcal{N}\mathcal{K} \langle_{cnt} = \rangle \mathcal{K} \langle_{cnt} \big[ [\cdot]\ \rangle \mathcal{N} \langle_{arg} \big]$$

- Other noteworthy cases:

$$\rangle \lambda k.\, \mathcal{A} \langle_{com} = \rangle \mathcal{A} \langle_{ans}$$

$$\rangle k \langle_{knt} = [\cdot]$$

$$\rangle \lambda x.\lambda k.\, \mathcal{A} \langle_{val} = \lambda x.\, \rangle \lambda k.\, \mathcal{A} \langle_{com} = \lambda x.\, \rangle \mathcal{A} \langle_{ans}$$

- The DS transform is the inverse of the CPS transform, but is more general: it can translate terms outside the image of the CPS transform.

- We can prove correctness for DS as for CPS:

$$\Gamma \Vdash_{\vartheta} A : B \text{ implies } \rangle \Gamma \langle \ \Vdash \ \rangle A \langle : \rangle B \langle \qquad \text{for } \vartheta \in \{\text{com, con, knd}\}.$$

* I use $\rangle \cdot\cdot \langle$ for the DS transform. The paper uses $D\langle \cdot\cdot \rangle$, and $C\langle \cdot\cdot \rangle$ for CPS.