Dynamically Typed Programming Languages

Part 2: Dynamic PCF

Jim Royer

CIS 352

April 16, 2019

Reference

- *Practical Foundations for Programming Languages, 1/e, "Part VI: Dynamic Types"*, by Robert Harper, Cambridge University Press, 2013, pages 127–148.

(Ordinary) PCF

PCF (Programming Computable Functions) Plotkin 1977

A theoretical cousin of ML and Haskell (and close to LFP$^+$).

PCF Raw Syntax

$$\begin{aligned} \mathit{Expr} ::= {} & X \mid \lambda X.\mathit{Expr} \mid \mathit{Expr}(\mathit{Expr}) \\ & \mid N \mid \mathsf{zero} \mid \mathsf{succ}(\mathit{Expr}) \mid \mathsf{fix}\, X\, \mathsf{is}\, \mathit{Expr} \\ & \mid \mathsf{ifz}\, \mathit{Expr}\, \{\mathsf{zero} \Rightarrow \mathit{Expr} \mid \mathsf{succ}(X) \Rightarrow \mathit{Expr}\,\} \end{aligned}$$

Note: For fix, think rec.

PCF Types: $\mathit{Type} ::= \mathsf{nat} \mid \mathit{Type} \to \mathit{Type}$

PCF Typing Rules: The usual thing.

Gordon Plotkin

Dynamic PCF

DPCF Raw Syntax (the same as PCF syntax)

$$\begin{aligned} Expr ::= X \mid \lambda X.Expr \mid Expr(Expr) \mid N \mid \mathsf{zero} \mid \mathsf{succ}(Expr) \\ \mid \mathsf{ifz}\, Expr\, \{\mathsf{zero} \Rightarrow Expr \mid \mathsf{succ}(X) \Rightarrow Expr\,\} \mid \mathsf{fix}\, X\, \mathsf{is}\, Expr \end{aligned}$$

DPCF Statics

$x_1 : \mathsf{ok}, \ldots x_n : \mathsf{ok} \vdash e : ok$

asserts that

- e is a well-formed expression
- with free variables $\subseteq \{\, x_1, \ldots, x_n \,\}$.

However:

$\vdash 3(4) : \mathsf{ok}$

is true but nonsensical (i.e., and error).

DPCF Dynamics

$$\begin{aligned} Expr ::= X \mid \lambda X.Expr \mid Expr(Expr) \mid N \mid \mathsf{zero} \mid \mathsf{succ}(Expr) \\ \mid \mathsf{ifz}\, Expr \,\{\mathsf{zero} \Rightarrow Expr \mid \mathsf{succ}(X) \Rightarrow Expr\,\} \mid \mathsf{fix}\, X \,\mathsf{is}\, Expr \end{aligned}$$

DPCF "classes": num and fun

DPCF judgment forms Note: d is abstract syntax

d val	d is a (closed) value
$d \mapsto d'$	d evaluates to d' in one step
d err	d incurs a run-time error
d isNum n	d is of class num with value n
d isNotNum	d is not of class num
d isFun `(Fun x d)`	d is of class fun with value `(Fun x d)`
d isNotFun	d is not of class fun

DPCF Dynamics Rules

Class checking

$$\frac{}{(\texttt{Num}\, n)\ \textsf{val}} \qquad \frac{}{(\texttt{Num}\, n)\ \textsf{isNum}\ n} \qquad \frac{}{(\texttt{Num}\, n)\ \textsf{isNotFun}}$$

$$\frac{}{(\texttt{Fun}\, x\, d)\ \textsf{val}\ (\texttt{Fun}\, x\, n)} \qquad \frac{}{(\texttt{Fun}\, x\, d)\ \textsf{isNotNum}} \qquad \frac{}{(\texttt{Fun}\, x\, d)\ \textsf{isFun}\ (\texttt{Fun}\, x\, d)}$$

Transition ($\mapsto$) rules

$$\frac{}{\texttt{Zero} \mapsto (\texttt{Num}\, 0)}$$

$$\frac{d \mapsto d'}{(\texttt{Succ}\, d) \mapsto (\texttt{Succ}\, d')} \qquad \frac{d\ \textsf{err}}{(\texttt{Succ}\, d)\ \textsf{err}}$$

$$\frac{d\ \textsf{isNum}\ n}{(\texttt{Succ}\, d) \mapsto (\texttt{Num}\, (n+1))} \qquad \frac{d\ \textsf{isNotNum}}{(\texttt{Succ}\, d)\ \textsf{err}}$$

DPCF Dynamics Rules, continued

More transition ($\mapsto$) rules

[See Harper for the five ifz rules]

$$\frac{d_1 \mapsto d_1'}{(\mathtt{App}\, d_1\, d_2) \mapsto (\mathtt{App}\, d_1'\, d_2)} \qquad \frac{d_1\ \mathsf{err}}{(\mathtt{App}\, d_1\, d_2)\ \mathsf{err})}$$

$$\frac{d_1\ \mathsf{isFun}\ (\mathtt{Fun}\, x\, d')}{(\mathtt{App}\, d_1\, d_2) \mapsto d[d_2/x]} \qquad \frac{d\ \mathsf{isNotFun}}{(\mathtt{App}\, d_1\, d_2)\ \mathsf{err}}$$

$$\frac{}{(\mathtt{Fix}\ x\ d) \mapsto d[(\mathtt{Fix}\ x\ d)/x]}$$

DPCF Dynamics, safety (such as it is)

Lemma (Class Checking)

If $(d\ \mathsf{val})$, *then:*

- *either* $(d\ \mathsf{isNum}\ n)$ *for some* n, *or* $(d\ \mathsf{isNotNum})$.
- *either* $(d\ \mathsf{isFun}\ (\mathtt{Fun}\ x\ d'))$ *for some* x *and* d', *or* $(d\ \mathsf{isNotFun})$.

Theorem (Progress)

If $\vdash d\ \mathsf{ok}$, *then either*

- $d\ \mathsf{val}$ *or*
- $d\ \mathsf{err}$ *or*
- $d \mapsto d'$ *for some* d'.

Static vs. Dynamic Typing

In a statically typed language (e.g., Java, Haskell, ...)

- Type checking happens *before* run time.
- Errors such as $4(3)$ get caught in type-checking.
- **But** you are stuck with the type-system of the language.

In a dynamically typed language (e.g., Scheme, Python, ...)

- $d(d')$ may sometimes be just fine and other times an error (say when d has the value 4).
- Type/class checking happens in ***every step of running the program!***
- This is a mild runtime overhead, but it complicates implementing and reasoning about programs ***a lot!***
- **But** you get to make up *(& enforce)* your own type system with every program you write.

Compromise position: Work in a language with a two-fisted type system.