CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Lambda Calculus; Dynamic Types

April 10, 2025

The Untyped Lambda Calculus

The abstract syntax of the untyped λ-calculus, called $\mathbf{\Lambda}$, is given by the following grammar:

Exp	u	$::=$	x	x	variable
			$\lambda(x.u)$	$\lambda\,(x)\,u$	λ-abstraction
			$\mathtt{ap}(u_1;u_2)$	$u_1(u_2)$	application

The statics of $\mathbf{\Lambda}$ is defined by general hypothetical judgments of the form $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that u is a well-formed expression involving the variables $x_1, \ldots, x_n$. (As usual, we omit explicit mention of the variables when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x\ \mathsf{ok} \vdash x\ \mathsf{ok}} \tag{21.1a}$$

$$\frac{\Gamma \vdash u_1\ \mathsf{ok} \quad \Gamma \vdash u_2\ \mathsf{ok}}{\Gamma \vdash u_1(u_2)\ \mathsf{ok}} \tag{21.1b}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u\ \mathsf{ok}}{\Gamma \vdash \lambda\,(x)\,u\ \mathsf{ok}} \tag{21.1c}$$

The Untyped Lambda Calculus

The dynamics of $\mathbf{\Lambda}$ is given equationally, rather than via a transition system. Definitional equality for $\mathbf{\Lambda}$ is a judgment of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1 \ \mathsf{ok}, \ldots, x_n \ \mathsf{ok}$ for some $n \geq 0$, and u and u' are terms having at most the variables $x_1, \ldots, x_n$ free. It is inductively defined by the following rules:

$$\frac{}{\Gamma, u \ \mathsf{ok} \vdash u \equiv u} \tag{21.2a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{21.2b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{21.2c}$$

$$\frac{\Gamma \vdash u_1 \equiv u_1' \quad \Gamma \vdash u_2 \equiv u_2'}{\Gamma \vdash u_1(u_2) \equiv u_1'(u_2')} \tag{21.2d}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda\,(x)\,u \equiv \lambda\,(x)\,u'} \tag{21.2e}$$

$$\frac{\Gamma, x \ \mathsf{ok} \vdash u_2 \ \mathsf{ok} \quad \Gamma \vdash u_1 \ \mathsf{ok}}{\Gamma \vdash (\lambda\,(x)\,u_2)(u_1) \equiv [u_1/x]u_2} \tag{21.2f}$$

The Lambda Calculus: Definability

It is a Turing-complete language --- it can express computations on natural numbers as in any other programming language

It is as powerful as PCF

The first task is to represent the natural numbers as certain λ-terms, called the *Church numerals*.

$$\overline{0} \triangleq \lambda\,(b)\,\lambda\,(s)\,b \tag{21.3a}$$

$$\overline{n+1} \triangleq \lambda\,(b)\,\lambda\,(s)\,s(\overline{n}(b)(s)) \tag{21.3b}$$

It follows that

$$\overline{n}(u_1)(u_2) \equiv u_2(\ldots(u_2(u_1))),$$

The Lambda Calculus: Definability

Using this definition, it is not difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped λ-terms:

$$\mathtt{succ} \triangleq \lambda\,(x)\,\lambda\,(b)\,\lambda\,(s)\,s(x(b)(s)) \tag{21.4}$$

$$\mathtt{plus} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(x)(\mathtt{succ}) \tag{21.5}$$

$$\mathtt{times} \triangleq \lambda\,(x)\,\lambda\,(y)\,y(\overline{0})(\mathtt{plus}(x)) \tag{21.6}$$

It is easy to check that $\mathtt{succ}(\overline{n}) \equiv \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

To define $\mathtt{ifz}\{u_0; x.u_1\}(u)$ requires a bit of ingenuity. The key is to define the "cut-off predecessor," `pred`, such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{21.7}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{21.8}$$

The Lambda Calculus: Definability

To make this precise, we must first define a Church-style representation of ordered pairs.

$$\langle u_1, u_2 \rangle \triangleq \lambda\,(f)\; f(u_1)(u_2) \tag{21.9}$$

$$u \cdot \mathtt{l} \triangleq u(\lambda\,(x)\,\lambda\,(y)\,x) \tag{21.10}$$

$$u \cdot \mathtt{r} \triangleq u(\lambda\,(x)\,\lambda\,(y)\,y) \tag{21.11}$$

It is easy to check that under this encoding $\langle u_1, u_2 \rangle \cdot \mathtt{l} \equiv u_1$, and that a similar equivalence holds for the second projection. We may now define the required representation, u_p, of the predecessor function:

$$u'_p \triangleq \lambda\,(x)\,x(\langle \overline{0}, \overline{0} \rangle)(\lambda\,(y)\,\langle y \cdot \mathtt{r}, \mathtt{succ}\,(y \cdot \mathtt{r}) \rangle) \tag{21.12}$$

$$u_p \triangleq \lambda\,(x)\,u'_p(x) \cdot \mathtt{l} \tag{21.13}$$

The Lambda Calculus: Definability

This definition gives us all the apparatus of PCF, apart from general recursion. But general recursion is also definable in Λ using a *fixed point combinator*. There are many choices of fixed point combinator, of which the best known is the $\mathtt{Y}$ *combinator*:

$$\mathtt{Y} \triangleq \lambda\,(F)\,(\lambda\,(f)\,F(f(f)))(\lambda\,(f)\,F(f(f))).$$

It is easy to check that

$$\mathtt{Y}(F) \equiv F(\mathtt{Y}(F)).$$

Using the $\mathtt{Y}$ combinator, we may define general recursion by writing $\mathtt{Y}(\lambda\,(x)\,u)$, where x stands for the recursive expression itself.

The Lambda Calculus: Definability

If a function is recursive, it is given an extra first argument, which is arranged at call sites to be the function itself. Whenever we wish to call a self-referential function with an argument, we apply the function first to itself and then to its argument; this protocol is imposed on both the "external" calls to the function and on the "internal" calls that the function may make to itself. For this reason, the first argument is often called this or self, to remind you that it will be, by convention, bound to the function itself.

With this in mind, it is easy to see how to derive the definition of Y. If F is the function whose fixed point we seek, then the function $F' = \lambda(f)\, F(f(f))$ is a variant of F in which the self-application convention has been imposed internally by substituting for each occurrence of f in $F(f)$ the self-application $f(f)$. Now check that $F'(F') \equiv F(F'(F'))$, so that $F'(F')$ is the desired fixed point of F. Expanding the definition of F', we have derived that the desired fixed point of F is

$$\lambda(f)\, F(f(f))(\lambda(f)\, F(f(f))).$$

Scott's Theorem

Scott's Theorem states that definitional equality for the untyped λ-calculus is undecidable: there is no algorithm to determine whether two untyped terms are definitionally equal. The proof uses the concept of *inseparability*. Any two properties, $\mathcal{A}_0$ and $\mathcal{A}_1$, of λ-terms are *inseparable* if there is no decidable property, $\mathcal{B}$, such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ holds, and $\mathcal{A}_1\ u$ implies that $\mathcal{B}\ u$ does *not* hold. We say that a property, $\mathcal{A}$, of untyped terms is *behavioral* iff whenever $u \equiv u'$, then $\mathcal{A}\ u$ iff $\mathcal{A}\ u'$.

The proof of Scott's Theorem decomposes into two parts:

1. For any untyped λ-term u, we may find an untyped term v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$, where $\ulcorner v \urcorner$ is the Gödel number of v, and $\overline{\ulcorner v \urcorner}$ is its representation as a Church numeral. (See Chapter 9 for a discussion of Gödel-numbering.)
2. Any two non-trivial[2] behavioral properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms are inseparable.

Scott's Theorem

Lemma 21.1. *For any u, there exists v such that $u(\overline{\ulcorner v \urcorner}) \equiv v$.*

Proof Sketch The proof relies on the definability of the following two operations in the untyped λ-calculus:

1. $\mathtt{ap}(\overline{\ulcorner u_1 \urcorner})(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1(u_2) \urcorner}$.
2. $\mathtt{nm}(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

Intuitively, the first takes the representations of two untyped terms and builds the representation of the application of one to the other. The second takes a numeral for n, and yields the representation of the Church numeral $\overline{n}$. Given these, we may find the required term v by defining $v \triangleq w(\overline{\ulcorner w \urcorner})$, where $w \triangleq \lambda\,(x)\,u(\mathtt{ap}(x)(\mathtt{nm}(x)))$. We have

$$\begin{aligned} v &= w(\overline{\ulcorner w \urcorner}) \\ &\equiv u(\mathtt{ap}(\overline{\ulcorner w \urcorner})(\mathtt{nm}(\overline{\ulcorner w \urcorner}))) \\ &\equiv u(\overline{\ulcorner w(\overline{\ulcorner w \urcorner}) \urcorner}) \\ &\equiv u(\overline{\ulcorner v \urcorner}). \end{aligned}$$

The definition is very similar to that of $\mathtt{Y}(u)$, except that u takes as input the representation of a term, and we find a v such that, when applied to the representation of v, the term u yields v itself. □

Scott's Theorem

Lemma 21.2. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-trivial behavioral properties of untyped terms. Then there is no untyped term w such that*

1. *For every u, either $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$ or $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*
2. *If $\mathcal{A}_0\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{0}$.*
3. *If $\mathcal{A}_1\ u$, then $w(\overline{\ulcorner u \urcorner}) \equiv \overline{1}$.*

Proof Suppose there is such an untyped term w. Let v be the untyped term

$$\lambda\,(x)\,\mathtt{ifz}\{u_1; \_.u_0\}(w(x)),$$

where u_0 and u_1 are chosen such that $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. (Such a choice must exist by non-triviality of the properties.) By Lemma 21.1 there is an untyped term t such that $v(\overline{\ulcorner t \urcorner}) \equiv t$. If $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, then $t \equiv v(\overline{\ulcorner t \urcorner}) \equiv u_1$, and so $\mathcal{A}_1\ t$, because $\mathcal{A}_1$ is behavioral and $\mathcal{A}_1\ u_1$.

But then $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$ by the defining properties of w, which is a contradiction. Similarly, if $w(\overline{\ulcorner t \urcorner}) \equiv \overline{1}$, then $\mathcal{A}_0\ t$, and hence $w(\overline{\ulcorner t \urcorner}) \equiv \overline{0}$, again a contradiction. □

Corollary 21.3. *There is no algorithm to decide whether $u \equiv u'$.*

Proof For fixed u, the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$ is a non-trivial behavioral property of untyped terms. So it is inseparable from its negation, and hence is undecidable. □

Untyped = Uni-Typed

The key observation is that the *untyped* λ-calculus is really the *uni-typed* λ-calculus. It is not the *absence* of types that gives it its power, but rather that it has *only one* type, the recursive type

$$D \triangleq \mathtt{rec}\, t\, \mathtt{is}\, t \rightharpoonup t.$$

A value of type D is of the form $\mathtt{fold}(e)$ where e is a value of type $D \rightharpoonup D$—a function whose domain and range are both D. Any such function can be regarded as a value of type D by "folding", and any value of type D can be turned into a function by "unfolding". As usual, a recursive type is a solution to a type equation, which in the present case is the equation

$$D \cong D \rightharpoonup D.$$

This isomorphism specifies that D is a type that is isomorphic to the space of partial functions on D itself, which is impossible if types are just sets.

Untyped = Uni-Typed

This isomorphism leads to the following translation, of Λ into **FPC**:

$$x^\dagger \triangleq x \tag{21.14a}$$

$$\lambda\,(x)\,u^\dagger \triangleq \mathtt{fold}(\lambda\,(x : D)\,u^\dagger) \tag{21.14b}$$

$$u_1(u_2)^\dagger \triangleq \mathtt{unfold}(u_1^\dagger)(u_2^\dagger) \tag{21.14c}$$

we have

$$\begin{aligned}
\lambda\,(x)\,u_1(u_2)^\dagger &= \mathtt{unfold}(\mathtt{fold}(\lambda\,(x : D)\,u_1^\dagger))(u_2^\dagger) \\
&\equiv \lambda\,(x : D)\,u_1^\dagger(u_2^\dagger) \\
&\equiv [u_2^\dagger/x]u_1^\dagger \\
&= ([u_2/x]u_1)^\dagger.
\end{aligned}$$

The last step, stating that the embedding commutes with substitution, is proved by induction on the structure of u_1. Thus β-reduction is implemented by evaluation of the embedded terms.

Dynamic Typing

An untyped language is a uni-typed language in which "untyped" terms are just terms of single recursive type.

No application can get stuck, because every value is a function that may be applied to an argument.

This safety property breaks down once more than one class of value is admitted. For example, if the natural numbers are added as a primitive, it is possible to incur a run-time error by attempting to apply a number to an argument.

One way to manage this is to embrace the possibility, treating class mismatches as checked errors.

Such languages are called *dynamic languages* because an error such as the one described is postponed to run-time, rather than precluded at compile time by type checking.

Dynamically Typed PCF

To illustrate dynamic typing, we formulate a dynamically typed version of **PCF**, called **DPCF**. The abstract syntax of **DPCF** is given by the following grammar:

Exp	d	::=	x	x	variable
			$\mathtt{num}[n]$	$\overline{n}$	numeral
			$\mathtt{zero}$	$\mathtt{zero}$	zero
			$\mathtt{succ}(d)$	$\mathtt{succ}(d)$	successor
			$\mathtt{ifz}\{d_0; x.d_1\}(d)$	$\mathtt{ifz}\, d\, \{\mathtt{zero} \hookrightarrow d_0 \mid \mathtt{succ}(x) \hookrightarrow d_1\}$	zero test
			$\mathtt{fun}(x.d)$	$\lambda\,(x)\,d$	abstraction
			$\mathtt{ap}(d_1; d_2)$	$d_1(d_2)$	application
			$\mathtt{fix}(x.d)$	$\mathtt{fix}\, x\, \mathtt{is}\, d$	recursion

There are two classes of values in **DPCF**, the *numbers*, which have the form $\mathtt{num}[n]$, and the *functions*, which have the form $\mathtt{fun}(x.d)$. The expressions $\mathtt{zero}$ and $\mathtt{succ}(d)$ are not themselves values, but rather are *constructors* that evaluate to values.

DPCF Statics

The statics of **DPCF** is like that of **Λ**; it merely checks that there are no free variables in the expression. The judgment

$$x_1 \ \mathsf{ok}, \ldots x_n \ \mathsf{ok} \vdash d \ \mathsf{ok}$$

states that d is a well-formed expression with free variables among those in the hypotheses. If the assumptions are empty, then we write just $d \ \mathsf{ok}$ to mean that d is a closed expression of **DPCF**.

DPCF Dynamics

d val	d is a (closed) value
$d \longmapsto d'$	d evaluates in one step to d'
d err	d incurs a run-time error
d is_num n	d is of class `num` with value n
d isnt_num	d is not of class `num`
d is_fun $x.d$	d is of class `fun` with body $x.d$
d isnt_fun	d is not of class `fun`

The value judgment d val states that d is a evaluated (closed) expression:

$$\frac{}{\texttt{num}[n]\ \mathsf{val}} \tag{22.1a}$$

$$\frac{}{\texttt{fun}(x.d)\ \mathsf{val}} \tag{22.1b}$$

DPCF Dynamics

The affirmative class-checking judgments are defined by the following rules:

$$\frac{}{\texttt{num}[n]\ \mathsf{is\_num}\ n} \tag{22.2a}$$

$$\frac{}{\texttt{fun}(x.d)\ \mathsf{is\_fun}\ x.d} \tag{22.2b}$$

The negative class-checking judgments are correspondingly defined by these rules:

$$\frac{}{\texttt{num}[n]\ \mathsf{isnt\_fun}} \tag{22.3a}$$

$$\frac{}{\texttt{fun}(x.d)\ \mathsf{isnt\_num}} \tag{22.3b}$$

The transition judgment $d \longmapsto d'$ and the error judgment $d\ \mathsf{err}$ are defined simultaneously by the following rules:

$$\frac{}{\texttt{zero} \longmapsto \texttt{num[z]}} \tag{22.4a}$$

$$\frac{d \longmapsto d'}{\texttt{succ}(d) \longmapsto \texttt{succ}(d')} \tag{22.4b}$$

$$\frac{d\ \mathsf{err}}{\texttt{succ}(d)\ \mathsf{err}} \tag{22.4c}$$

$$\frac{d\ \mathsf{is\_num}\ n}{\texttt{succ}(d) \longmapsto \texttt{num[s}(n)\texttt{]}} \tag{22.4d}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{succ}(d)\ \mathsf{err}} \tag{22.4e}$$

DPCF Dynamics

$$\frac{d \longmapsto d'}{\texttt{ifz}\{d_0; x.d_1\}(d) \longmapsto \texttt{ifz}\{d_0; x.d_1\}(d')} \quad (22.4f)$$

$$\frac{d\ \mathsf{err}}{\texttt{ifz}\{d_0; x.d_1\}(d)\ \mathsf{err}} \quad (22.4g)$$

$$\frac{d\ \mathsf{is\_num}\ 0}{\texttt{ifz}\{d_0; x.d_1\}(d) \longmapsto d_0} \quad (22.4h)$$

$$\frac{d\ \mathsf{is\_num}\ n+1}{\texttt{ifz}\{d_0; x.d_1\}(d) \longmapsto [\texttt{num}[n]/x]d_1} \quad (22.4i)$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{ifz}\{d_0; x.d_1\}(d)\ \mathsf{err}} \quad (22.4j)$$

$$\frac{d_1 \longmapsto d_1'}{\mathtt{ap}(d_1; d_2) \longmapsto \mathtt{ap}(d_1'; d_2)} \quad (22.4k)$$

$$\frac{d_1\ \mathsf{err}}{\mathtt{ap}(d_1; d_2)\ \mathsf{err}} \quad (22.4l)$$

$$\frac{d_1\ \mathsf{is\_fun}\ x.d}{\mathtt{ap}(d_1; d_2) \longmapsto [d_2/x]d} \quad (22.4m)$$

$$\frac{d_1\ \mathsf{isnt\_fun}}{\mathtt{ap}(d_1; d_2)\ \mathsf{err}} \quad (22.4n)$$

$$\frac{}{\texttt{fix}(x.d) \longmapsto [\texttt{fix}(x.d)/x]d} \quad (22.4o)$$

DPCF Safety

Lemma 22.1 (Class Checking)**.** *If d val, then*

1. *either d $\mathsf{is\_num}$ n for some n, or d $\mathsf{isnt\_num}$;*
2. *either d $\mathsf{is\_fun}$ $x.d'$ for some x and d', or d $\mathsf{isnt\_fun}$.*

Proof By inspection of the rules defining the class-checking judgments. □

Theorem 22.2 (Progress)**.** *If d ok, then either d val, or d err, or there exists d' such that $d \longmapsto d'$.*

Proof By induction on the structure of d. For example, if $d = \mathtt{succ}(d')$, then we have by induction either d' val, or d' err, or $d' \longmapsto d''$ for some d''. In the last case, we have by rule (22.4b) that $\mathtt{succ}(d') \longmapsto \mathtt{succ}(d'')$, and in the second-to-last case, we have by rule (22.4c) that $\mathtt{succ}(d')$ err. If d' val, then by Lemma 22.1, either d' $\mathsf{is\_num}$ n or d' $\mathsf{isnt\_num}$. In the former case $\mathtt{succ}(d') \longmapsto \mathtt{num}[\mathtt{s}(n)]$, and in the latter $\mathtt{succ}(d')$ err. The other cases are handled similarly. □

Lemma 22.3 (Exclusivity)**.** *For any d in* **DPCF***, exactly one of the following holds: d val, or d err, or $d \longmapsto d'$ for some d'.*

DPCF Variations and Extensions

One could instead treat `zero` and `succ`(d) as values of separate classes and introduce the obvious class-checking judgments for them. When written in this style, the dynamics of the conditional branch is given as follows:

$$\frac{d \longmapsto d'}{\mathtt{ifz}\{d_0; x.d_1\}(d) \longmapsto \mathtt{ifz}\{d_0; x.d_1\}(d')} \tag{22.5a}$$

$$\frac{d \text{ is\_zero}}{\mathtt{ifz}\{d_0; x.d_1\}(d) \longmapsto d_0} \tag{22.5b}$$

$$\frac{d \text{ is\_succ } d'}{\mathtt{ifz}\{d_0; x.d_1\}(d) \longmapsto [d'/x]d_1} \tag{22.5c}$$

$$\frac{d \text{ isnt\_zero} \quad d \text{ isnt\_succ}}{\mathtt{ifz}\{d_0; x.d_1\}(d) \text{ err}} \tag{22.5d}$$

DPCF Variations and Extensions

DPCF can be extended with structured data similarly. A classic example is to consider a class `nil`, consisting of a "null" value, and a class `cons`, consisting of pairs of values.

Exp	d	::=	`nil`	`nil`	null
			`cons`$(d_1; d_2)$	`cons`$(d_1; d_2)$	pair
			`ifnil`$(d; d_0; x, y.d_1)$	`ifnil` d {`nil` $\hookrightarrow d_0$ | `cons`$(x; y) \hookrightarrow d_1$}	conditional

The expression `ifnil`$(d; d_0; x, y.d_1)$ distinguishes the null value from a pair, and signals an error on any other class of value.

Lists (finite sequences) can be encoded using null and pairing. For example, the list consisting of three zeros can berepresented by the value

$$\texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \texttt{nil}))).$$

But what to make of the following value?

$$\texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \texttt{cons}(\texttt{zero}; \lambda\,(x)\,x)))$$

It is not a list, because it does not end with `nil`, but it is a permissible value in the enriched language.

DPCF Variations and Extensions

It might be argued that the conditional branch that distinguishes null from a pair is inappropriate in **DPCF**, because there are more than just these two classes in the language. One approach that avoids this criticism is to abandon pattern matching on the class of data, replacing it by a general conditional branch that distinguishes null from all other values, and adding to the language *predicates*[1] that test the class of a value and *destructors* that invert the constructors of each class.

We could instead reformulate null and and pairing as follows:

Exp	d	::=	$\texttt{cond}(d; d_0; d_1)$	$\texttt{cond}(d; d_0; d_1)$	conditional
			$\texttt{nil?}(d)$	$\texttt{nil?}(d)$	nil test
			$\texttt{cons?}(d)$	$\texttt{cons?}(d)$	pair test
			$\texttt{car}(d)$	$\texttt{car}(d)$	first projection
			$\texttt{cdr}(d)$	$\texttt{cdr}(d)$	second projection

Written in this form, the function `append` is given by the expression

$$\texttt{fix}\, a\, \texttt{is}\, \lambda\,(x)\, \lambda\,(y)\, \texttt{cond}(x; \texttt{cons}(\texttt{car}(x); a(\texttt{cdr}(x))(y)); y).$$

Critique of Dynamic Typing

Consider, for example, the addition function in **DPCF**, whose specification is that, when passed two values of class `num`, returns their sum, which is also of class `num`:[3]

$$\texttt{fun}(x.\texttt{fix}(p.\texttt{fun}(y.\texttt{ifz}\{x; y'.\texttt{succ}(p(y'))\}))).$$

The addition function may, deceptively, be written in concrete syntax as follows:

$$\lambda\,(x)\,\texttt{fix}\,p\,\texttt{is}\,\lambda\,(y)\,\texttt{ifz}\,y\,\{\texttt{zero} \hookrightarrow x \mid \texttt{succ}(y') \hookrightarrow \texttt{succ}(p(y'))\}.$$

First, note that the body of the fixed point expression is labeled with class `fun`. The dynamics of the fixed point construct binds p to this function. Consequently, the dynamic class check incurred by the application of p in the recursive call is guaranteed to succeed. But **DPCF** offers no means of suppressing the redundant check, because it cannot express the invariant that p is always bound to a value of class `fun`.

Critique of Dynamic Typing

Second, note that the result of applying the inner λ-abstraction is either x, the argument of the outer λ-abstraction, or the successor of a recursive call to the function itself. The successor operation checks that its argument is of class `num`, even though this condition is guaranteed to hold for all but the base case, which returns the given x, which can be of any class at all. In principle, we can check that x is of class `num` once, and note that it is otherwise a loop invariant that the result of applying the inner function is of this class. However, **DPCF** gives us no way to express this invariant; the repeated, redundant tag checks imposed by the successor operation cannot be avoided.

Third, the argument y to the inner function is either the original argument to the addition function, or is the predecessor of some earlier recursive call. But as long as the original call is to a value of class `num`, then the dynamics of the conditional will ensure that all recursive calls have this class. And again there is no way to express this invariant in **DPCF**, and hence, there is no way to avoid the class check imposed by the conditional branch.

Hybrid Typing

A *hybrid* language is one that combines static and dynamic typing by enriching a statically typed language with a distinguished type dyn of dynamic values.

The dynamically typed language can be embedded into the hybrid language by viewing a dynamically typed program as a statically typed program of type dyn.

Static and dynamic types are not opposed to one another but may coexist harmoniously.

The *ad hoc* device of adding the type dyn to a static language is unnecessary in a language with recursive types, wherein it is definable as a particular recursive type.

Thus, *dynamic typing is a mode of use of static typing*, reconciling an apparent opposition between them.

Hybrid PCF (HPCF)

Consider the language **HPCF**, which extends **PCF** with the following constructs:

Typ	τ	::=	`dyn`	`dyn`	dynamic
Exp	e	::=	`new`$[l](e)$	$l \,!\, e$	construct
			`cast`$[l](e)$	$e \,@\, l$	destruct
			`inst`$[l](e)$	$l \,?\, e$	discriminate
Cls	l	::=	`num`	`num`	number
			`fun`	`fun`	function

The type `dyn` is the type of dynamically classified values. The constructor attaches a classifier to a value of a type associated to that classifer, the destructor recovers the value classified with the given classifier, and the discriminator tests the class of a classified value.

HPCF Statics

The statics of **HPCF** extends that of **PCF** with the following rules:

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{new}[\mathtt{num}](e) : \mathtt{dyn}} \tag{23.1a}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\Gamma \vdash \mathtt{new}[\mathtt{fun}](e) : \mathtt{dyn}} \tag{23.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{num}](e) : \mathtt{nat}} \tag{23.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{fun}](e) : \mathtt{dyn} \rightharpoonup \mathtt{dyn}} \tag{23.1d}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{inst}[\mathtt{num}](e) : \mathtt{bool}} \tag{23.1e}$$

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{inst}[\mathtt{fun}](e) : \mathtt{bool}} \tag{23.1f}$$

HPCF Dynamics

$$\frac{e\ \mathsf{val}}{\mathtt{new}[l](e)\ \mathsf{val}} \tag{23.2a}$$

$$\frac{e \longmapsto e'}{\mathtt{new}[l](e) \longmapsto \mathtt{new}[l](e')} \tag{23.2b}$$

$$\frac{e \longmapsto e'}{\mathtt{cast}[l](e) \longmapsto \mathtt{cast}[l](e')} \tag{23.2c}$$

$$\frac{\mathtt{new}[l](e)\ \mathsf{val}}{\mathtt{cast}[l](\mathtt{new}[l](e)) \longmapsto e} \tag{23.2d}$$

$$\frac{\mathtt{new}[l'](e)\ \mathsf{val} \quad l \neq l'}{\mathtt{cast}[l](\mathtt{new}[l'](e))\ \mathsf{err}} \tag{23.2e}$$

$$\frac{e \longmapsto e'}{\mathtt{inst}[l](e) \longmapsto \mathtt{inst}[l](e')} \tag{23.2f}$$

$$\frac{\mathtt{new}[l](e)\ \mathsf{val}}{\mathtt{inst}[l](\mathtt{new}[l](e)) \longmapsto \mathtt{true}} \tag{23.2g}$$

$$\frac{\mathtt{new}[l](e)\ \mathsf{val} \quad l \neq l'}{\mathtt{inst}[l'](\mathtt{new}[l](e)) \longmapsto \mathtt{false}} \tag{23.2h}$$

HPCF Safety

Lemma 23.1 (Canonical Forms)**.** *If* $e : \texttt{dyn}$ *and* e val*, then* $e = \texttt{new}[l](e')$ *for some class* l *and some* e' val*. If* $l = \texttt{num}$*, then* $e' : \texttt{nat}$*, and if* $l = \texttt{fun}$*, then* $e' : \texttt{dyn} \rightharpoonup \texttt{dyn}$*.*

Proof By rule induction on the statics of **HPCF**. □

Theorem 23.2 (Safety)**.** *The language* **HPCF** *is safe:*

1. *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$*.*
2. *If* $e : \tau$*, then either* e val*, or* e err*, or* $e \longmapsto e'$ *for some* e'*.*

Proof Preservation is proved by rule induction on the dynamics, and progress is proved by rule induction on the statics, making use of the canonical forms lemma. The opportunities for run-time errors are the same as those for **DPCF**—a well-typed cast might fail at run-time if the class of the cast does not match the class of the value. □

HPCF in FPC

In a language such as **FPC** (Chapter 20) with recursive types, there is no need to add `dyn` as a primitive type. Instead, it is defined to be type

$$\texttt{rec}\, t\, \texttt{is}\, [\texttt{num} \hookrightarrow \texttt{nat}, \texttt{fun} \hookrightarrow t \rightharpoonup t]. \tag{23.3}$$

The introduction and elimination forms for this definition of `dyn` are definable as follows:[2]

$$\texttt{new}[\texttt{num}](e) \triangleq \texttt{fold}(\texttt{num} \cdot e) \tag{23.4}$$

$$\texttt{new}[\texttt{fun}](e) \triangleq \texttt{fold}(\texttt{fun} \cdot e) \tag{23.5}$$

$$\texttt{cast}[\texttt{num}](e) \triangleq \texttt{case}\, \texttt{unfold}(e)\, \{\texttt{num} \cdot x \hookrightarrow x \mid \texttt{fun} \cdot x \hookrightarrow \texttt{error}\} \tag{23.6}$$

$$\texttt{cast}[\texttt{fun}](e) \triangleq \texttt{case}\, \texttt{unfold}(e)\, \{\texttt{num} \cdot x \hookrightarrow \texttt{error} \mid \texttt{fun} \cdot x \hookrightarrow x\}. \tag{23.7}$$

These definition simply decompose the class operations for `dyn` into recursive unfoldings and case analyses on values of a sum type.

Dynamic as Static Typing

The language **DPCF** of Chapter 22 can be embedded into **HPCF** by a simple translation that makes explicit the class checking in the dynamics of **DPCF**. Specifically, we may define a translation $d^\dagger$ of expressions of **DPCF** into expressions of **HPCF** according to the following static correctness criterion:

Theorem 23.3. *If* $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash d\ \mathsf{ok}$ *according to the statics of* **DPCF**, *then* $x_1 : \mathtt{dyn}, \ldots, x_n : \mathtt{dyn} \vdash d^\dagger : \mathtt{dyn}$ *in* **HPCF**.

The proof of Theorem 23.3 is given by induction on the structure of d based on the following translation:

$$
\begin{aligned}
x^\dagger &\triangleq x \\
\mathtt{num}[n]^\dagger &\triangleq \mathtt{new}[\mathtt{num}](\overline{n}) \\
\mathtt{zero}^\dagger &\triangleq \mathtt{new}[\mathtt{num}](\mathtt{z}) \\
\mathtt{succ}(d)^\dagger &\triangleq \mathtt{new}[\mathtt{num}](\mathtt{s}(\mathtt{cast}[\mathtt{num}](d^\dagger))) \\
\mathtt{ifz}\{d_0; x.d_1\}(d) &\triangleq \mathtt{ifz}\{d_0^\dagger; x.[\mathtt{new}[\mathtt{num}](x)/x]d_1^\dagger\}(\mathtt{cast}[\mathtt{num}](d^\dagger)) \\
(\mathtt{fun}(x.d))^\dagger &\triangleq \mathtt{new}[\mathtt{fun}](\lambda\,(x : \mathtt{dyn})\,d^\dagger) \\
(\mathtt{ap}(d_1; d_2))^\dagger &\triangleq \mathtt{cast}[\mathtt{fun}](d_1^\dagger)(d_2^\dagger) \\
\mathtt{fix}(x.d) &\triangleq \mathtt{fix}\{\mathtt{dyn}\}(x.d^\dagger)
\end{aligned}
$$

Optimization of Dynamic Typing in HPCF

Consider the addition function in **DPCF** given in Section 22.3, which we transcribe here for convenience:

$$\lambda\,(x)\,\mathtt{fix}\,p\,\mathtt{is}\,\lambda\,(y)\,\mathtt{ifz}\,y\,\{\mathtt{zero} \hookrightarrow x \mid \mathtt{succ}(y') \hookrightarrow \mathtt{succ}(p(y'))\}.$$

It is a value of type `dyn` in **HPCF** given as follows:

$$\mathtt{fun}\,!\,\lambda\,(x : \mathtt{dyn})\,\mathtt{fix}\,p : \mathtt{dyn}\,\mathtt{is}\,\mathtt{fun}\,!\,\lambda\,(y : \mathtt{dyn})\,e_{x,p,y}, \tag{23.8}$$

within which the fragment

$$x : \mathtt{dyn},\ p : \mathtt{dyn},\ y : \mathtt{dyn} \vdash e_{x,p,y} : \mathtt{dyn}$$

stands for the expression

$$\mathtt{ifz}\,(y\ @\ \mathtt{num})\,\{\mathtt{zero} \hookrightarrow x \mid \mathtt{succ}(y') \hookrightarrow \mathtt{num}\,!\,(\mathtt{s}((p\ @\ \mathtt{fun})(\mathtt{num}\,!\,y')\ @\ \mathtt{num}))\}.$$

The embedding into **HPCF** makes explicit the run-time checks that are implicit in the dynamics of **DPCF**.

Optimization of Dynamic Typing in HPCF

The first redundancy arises from the use of recursion in a dynamic language. In the above example, we use recursion to define the inner loop p of the computation. The value p is, by definition, a λ-abstraction, which is explicitly tagged as a function. Yet the call to p within the loop checks at run-time whether p is in fact a function before applying it. Because p is an internally defined function, all of its call sites are under the control of the addition function, which means that there is no need for such pessimism at calls to p, provided that we change its type to $\texttt{dyn} \rightharpoonup \texttt{dyn}$, which directly expresses the invariant that p is a function acting on dynamic values.

Performing this transformation, we obtain the following reformulation of the addition function that eliminates this redundancy:

$$\texttt{fun} \mathbin{!} \lambda\,(x : \texttt{dyn})\,\texttt{fun} \mathbin{!} \texttt{fix}\; p : \texttt{dyn} \rightharpoonup \texttt{dyn}\; \texttt{is}\; \lambda\,(y : \texttt{dyn})\, e'_{x,p,y},$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y \mathbin{@} \texttt{num})\,\{\texttt{zero} \hookrightarrow x \mid \texttt{succ}(y') \hookrightarrow \texttt{num} \mathbin{!} (\texttt{s}(p(\texttt{num} \mathbin{!} y') \mathbin{@} \texttt{num}))\}.$$

We have "hoisted" the function class label out of the loop and suppressed the cast inside the loop. Correspondingly, the type of p has changed to $\texttt{dyn} \rightharpoonup \texttt{dyn}$.

Optimization of Dynamic Typing in HPCF

Two observations:

- Variable y of type dyn can be made into nat
- The result of the recursive call can be made to nat as well

Combining these optimizations we obtain the inner loop e''_x defined as follows:

$$\texttt{fix}\, p : \texttt{nat} \rightharpoonup \texttt{nat}\, \texttt{is}\, \lambda\, (y : \texttt{nat})\, \texttt{ifz}\, y\, \{\texttt{zero} \hookrightarrow x\ @\ \texttt{num} \mid \texttt{succ}(y') \hookrightarrow \texttt{s}(p(y'))\}.$$

It has the type $\texttt{nat} \rightharpoonup \texttt{nat}$, and runs without class checks when applied to a natural number.

Finally, recall that the goal is to define a version of addition that works on values of type dyn. Thus, we need a value of type $\texttt{dyn} \rightharpoonup \texttt{dyn}$, but what we have at hand is a function of type $\texttt{nat} \rightharpoonup \texttt{nat}$. It can be converted to the needed form by pre-composing with a cast to num and post-composing with a coercion to num:

$$\texttt{fun}\, !\, \lambda\, (x : \texttt{dyn})\, \texttt{fun}\, !\, \lambda\, (y : \texttt{dyn})\, \texttt{num}\, !\, (e''_x(y\ @\ \texttt{num})).$$

The innermost λ-abstraction converts the function e''_x from type $\texttt{nat} \rightharpoonup \texttt{nat}$ to type $\texttt{dyn} \rightharpoonup \texttt{dyn}$ by composing it with a class check that ensures that y is a natural number at the initial call site, and applies a label to the result to restore it to type dyn.

Static vs. Dynamic Typing

The seeming opposition btw static & dynamic typing is an illusion.

- *Dynamic lang. associate types with values, whereas static lang. associate types to variables?*
- *Dynamic lang. check types at run-time, whereas static lang. check types at compile time?*
- *Dynamic lang. support heterogeneous collections, whereas static lang. support homogeneous collections?*

The question is not *whether* to have static typing, but rather how best to embrace it.