Chapter 19

The Untyped λ-Calculus

Types are the central organizing principle in the study of programming languages. Yet many languages of practical interest are said to be *untyped*. Have we missed something important? The answer is *no*! The supposed opposition between typed and untyped languages turns out to be illusory. In fact, untyped languages are special cases of typed languages with a single, pre-determined recursive type. Far from being *untyped*, such languages are instead *uni-typed*.[1]

In this chapter we study the premier example of a uni-typed programming language, the *(untyped) λ-calculus*. This formalism was introduced by Church in the 1930's as a universal language of computable functions. It is distinctive for its austere elegance. The λ-calculus has but one "feature", the higher-order function, with which to compute. Everything is a function, hence every expression may be applied to an argument, which must itself be a function, with the result also being a function. To borrow a well-worn phrase, in the λ-calculus it's functions all the way down!

19.1 The λ-Calculus

The abstract syntax of $\mathcal{L}\{\lambda\}$ is given by the following grammar:

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

The statics of $\mathcal{L}\{\lambda\}$ is defined by general hypothetical judgements of the form $x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash u\ \mathsf{ok}$, stating that u is a well-formed expression

[1] An apt description of Dana Scott's.

involving the variables $x_1, \ldots, x_n$. (As usual, we omit explicit mention of the parameters when they can be determined from the form of the hypotheses.) This relation is inductively defined by the following rules:

$$\frac{}{\Gamma, x \text{ ok} \vdash x \text{ ok}} \tag{19.1a}$$

$$\frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok}}{\Gamma \vdash \mathsf{ap}(u_1; u_2) \text{ ok}} \tag{19.1b}$$

$$\frac{\Gamma, x \text{ ok} \vdash u \text{ ok}}{\Gamma \vdash \lambda(x.u) \text{ ok}} \tag{19.1c}$$

The dynamics is given by the following rules:

$$\frac{}{\lambda(x.u) \text{ val}} \tag{19.2a}$$

$$\frac{}{\mathsf{ap}(\lambda(x.u_1); u_2) \mapsto [u_2/x]u_1} \tag{19.2b}$$

$$\frac{u_1 \mapsto u_1'}{\mathsf{ap}(u_1; u_2) \mapsto \mathsf{ap}(u_1'; u_2)} \tag{19.2c}$$

In the λ-calculus literature this judgement is called *weak head reduction*. The first rule is called *β-reduction*; it defines the meaning of function application as substitution of argument for parameter.

Despite the apparent lack of types, $\mathcal{L}\{\lambda\}$ is nevertheless type safe!

Theorem 19.1. *If u ok, then either u val, or there exists u' such that $u \mapsto u'$ and u' ok.*

Proof. Exactly as in preceding chapters. We may show by induction on transition that well-formation is preserved by the dynamics. Since every closed value of $\mathcal{L}\{\lambda\}$ is a λ-abstraction, every closed expression is either a value or can make progress. □

Definitional equivalence for $\mathcal{L}\{\lambda\}$ is a judgement of the form $\Gamma \vdash u \equiv u'$, where $\Gamma = x_1 \text{ ok}, \ldots, x_n \text{ ok}$ 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:

$$\frac{}{\Gamma, u \text{ ok} \vdash u \equiv u} \tag{19.3a}$$

$$\frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \tag{19.3b}$$

$$\frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \tag{19.3c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash \mathsf{ap}(e_1; e_2) \equiv \mathsf{ap}(e_1'; e_2')} \tag{19.3d}$$

$$\frac{\Gamma, x\ \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x.u) \equiv \lambda(x.u')} \tag{19.3e}$$

$$\frac{}{\Gamma \vdash \mathsf{ap}(\lambda(x.e_2); e_1) \equiv [e_1/x]e_2} \tag{19.3f}$$

We often write just $u \equiv u'$ when the variables involved need not be emphasized or are clear from context.

19.2 Definability

Interest in the untyped λ-calculus stems from its surprising expressiveness. It is a *Turing-complete* language in the sense that it has the same capability to expression computations on the natural numbers as does any other known programming language. Church's Law states that any conceivable notion of computable function on the natural numbers is equivalent to the λ-calculus. This is certainly true for all *known* means of defining computable functions on the natural numbers. The force of Church's Law is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the λ-calculus. Church's Law is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation, which makes a prediction about all future measurements of the acceleration in a gravitational field.[2]

We will sketch a proof that the untyped λ-calculus is as powerful as the language PCF described in Chapter 12. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped λ-calculus. This means, in particular, that we must show that the natural numbers are definable as λ-terms in such a way that case analysis, which discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

[2] Unfortunately, it is common in Computer Science to put forth as "laws" assertions that are not scientific laws at all. For example, Moore's Law is merely an observation about a near-term trend in microprocessor fabrication that is certainly not valid over the long term, and Amdahl's Law is but a simple truth of arithmetic. Worse, Church's Law, which is a proper scientific law, is usually called *Church's Thesis*, which, to the author's ear, suggests something less than the full force of a scientific law.

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

$$\overline{0} = \lambda\, b.\, \lambda\, s.\, b \tag{19.4a}$$

$$\overline{n+1} = \lambda\, b.\, \lambda\, s.\, s(\overline{n}(b)(s)) \tag{19.4b}$$

It follows that

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

the n-fold application of u_2 to u_1. That is, $\overline{n}$ iterates its second argument (the induction step) n times, starting with its first argument (the basis).

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:

$$\mathtt{succ} = \lambda\, x.\, \lambda\, b.\, \lambda\, s.\, s(x(b)(s)) \tag{19.5}$$

$$\mathtt{plus} = \lambda\, x.\, \lambda\, y.\, y(x)(\mathtt{succ}) \tag{19.6}$$

$$\mathtt{times} = \lambda\, x.\, \lambda\, y.\, y(\overline{0})(\mathtt{plus}(x)) \tag{19.7}$$

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 $\mathtt{ifz}(u; u_0; x.\, u_1)$ requires a bit of ingenuity. We wish to find a term $\mathtt{pred}$ such that

$$\mathtt{pred}(\overline{0}) \equiv \overline{0} \tag{19.8}$$

$$\mathtt{pred}(\overline{n+1}) \equiv \overline{n}. \tag{19.9}$$

To compute the predecessor using Church numerals, we must show how to compute the result for $\overline{n+1}$ as a function of its value for $\overline{n}$. At first glance this seems straightforward—just take the successor—until we consider the base case, in which we define the predecessor of $\overline{0}$ to be $\overline{0}$. This invalidates the obvious strategy of taking successors at inductive steps, and necessitates some other approach.

What to do? A useful intuition is to think of the computation in terms of a pair of "shift registers" satisfying the invariant that on the nth iteration the registers contain the predecessor of n and n itself, respectively. Given the result for n, namely the pair $(n-1, n)$, we pass to the result for $n+1$ by shifting left and incrementing to obtain $(n, n+1)$. For the base case, we initialize the registers with $(0, 0)$, reflecting the stipulation that the predecessor of zero be zero. To compute the predecessor of n we compute the pair $(n-1, n)$ by this method, and return the first component.

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

$$\langle u_1, u_2 \rangle = \lambda f.\, f(u_1)(u_2) \tag{19.10}$$
$$u \cdot \mathtt{l} = u(\lambda x.\, \lambda y.\, x) \tag{19.11}$$
$$u \cdot \mathtt{r} = u(\lambda x.\, \lambda y.\, y) \tag{19.12}$$

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:

$$u'_p = \lambda x.\, x(\langle \overline{0}, \overline{0} \rangle)(\lambda y.\, \langle y \cdot \mathtt{r}, \mathtt{s}(y \cdot \mathtt{r}) \rangle) \tag{19.13}$$
$$u_p = \lambda x.\, u(x) \cdot \mathtt{l} \tag{19.14}$$

It is easy to check that this gives us the required behavior. Finally, we may define $\mathtt{ifz}(u; u_0; x.\, u_1)$ to be the untyped term

$$u(u_0)(\lambda \_.\, [u_p(u)/x]u_1).$$

This gives us all the apparatus of PCF, apart from general recursion. But this is also definable using a *fixed point combinator*. There are many choices of fixed point combinator, of which the best known is the **Y** *combinator*:

$$\mathbf{Y} = \lambda F.\, (\lambda f.\, F(f(f)))(\lambda f.\, F(f(f))). \tag{19.15}$$

Observe that

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

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

19.3 Scott's Theorem

Definitional equivalence for the untyped λ-calculus is undecidable: there is no algorithm to determine whether or not two untyped terms are definitionally equivalent. The proof of this result is based on two key lemmas:

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 11 for a discussion of Gödel-numbering.)

2. Any two non-trivial[3] properties $\mathcal{A}_0$ and $\mathcal{A}_1$ of untyped terms that respect definitional equivalence are *inseparable*. This means that there is no decidable property $\mathcal{B}$ of untyped terms such that $\mathcal{A}_0\ u$ implies that $\mathcal{B}\ u$ and $\mathcal{A}_1\ u$ implies that it is *not* the case that $\mathcal{B}\ u$. In particular, if $\mathcal{A}_0$ and $\mathcal{A}_1$ are inseparable, then neither is decidable.

For a property $\mathcal{B}$ of untyped terms to respect definitional equivalence means that if $\mathcal{B}\ u$ and $u \equiv u'$, then $\mathcal{B}\ u'$.

Lemma 19.2. *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. $\mathbf{ap}(\overline{\ulcorner u_1 \urcorner})\,(\overline{\ulcorner u_2 \urcorner}) \equiv \overline{\ulcorner u_1\,(u_2) \urcorner}$.
2. $\mathbf{nm}(\overline{n}) \equiv \overline{\ulcorner \overline{n} \urcorner}$.

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 $\overline{n}$. Given these, we may find the required term v by defining $v = w(\overline{\ulcorner w \urcorner})$, where $w = \lambda\, x.\, u(\mathbf{ap}(x)(\mathbf{nm}(x)))$. We have

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

The definition is very similar to that of $\mathbf{Y}(u)$, 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. □

Lemma 19.3. *Suppose that $\mathcal{A}_0$ and $\mathcal{A}_1$ are two non-vacuous properties of untyped terms that respect definitional equivalence. 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] A property of untyped terms is said to be *trivial* if it either holds for all untyped terms or never holds for any untyped term.

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 $\lambda\, x.\, \mathtt{ifz}(w(x); u_1; \_.\, u_0)$, where $\mathcal{A}_0\ u_0$ and $\mathcal{A}_1\ u_1$. By Lemma 19.2 on the preceding page 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 $\mathcal{A}_1\ t$, since $\mathcal{A}_1$ respects definitional equivalence 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 19.4. *There is no algorithm to decide whether or not* $u \equiv u'$.

Proof. For fixed u consider the property $\mathcal{E}_u\ u'$ defined by $u' \equiv u$. This is non-vacuous and respects definitional equivalence, and hence is undecidable. □

19.4 Untyped Means Uni-Typed

The untyped λ-calculus may be faithfully embedded in a typed language with recursive types. This means that every untyped λ-term has a representation as a typed expression in such a way that execution of the representation of a λ-term corresponds to execution of the term itself. This embedding is *not* a matter of writing an interpreter for the λ-calculus in $\mathcal{L}\{+\times\rightharpoonup\mu\}$ (which we could surely do), but rather a direct representation of untyped λ-terms as typed expressions in a language with recursive types.

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, namely the recursive type

$$D = \mu t\,.\, t \to t.$$

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

$$D \cong D \to D.$$

This specifies that D is a type that is isomorphic to the space of functions on D itself, something that is impossible in conventional set theory, but is feasible in the computationally-based setting of the λ-calculus.

This isomorphism leads to the following translation, of $\mathcal{L}\{\lambda\}$ into $\mathcal{L}\{+\times\rightharpoonup\mu\}$:

$$x^\dagger = x \tag{19.16a}$$
$$\lambda\, x.\, u^\dagger = \mathtt{fold}(\lambda\, (x{:}D.\, u^\dagger)) \tag{19.16b}$$
$$u_1(u_2)^\dagger = \mathtt{unfold}(u_1^\dagger)(u_2^\dagger) \tag{19.16c}$$

Observe that the embedding of a λ-abstraction is a value, and that the embedding of an application exposes the function being applied by unrolling the recursive type. Consequently,

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

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

Thus we see that the canonical *untyped* language, $\mathcal{L}\{\lambda\}$, which by dint of terminology stands in opposition to *typed* languages, turns out to be but a typed language after all! Rather than eliminating types, an untyped language consolidates an infinite collection of types into a single recursive type. Doing so renders static type checking trivial, at the expense of incurring substantial dynamic overhead to coerce values to and from the recursive type. In Chapter 20 we will take this a step further by admitting many different types of data values (not just functions), each of which is a component of a "master" recursive type. This shows that so-called *dynamically typed* languages are, in fact, *statically typed*. Thus a traditional distinction can hardly be considered an opposition, since dynamic languages are but particular forms of static language in which (undue) emphasis is placed on a single recursive type.

19.5 Notes

The untyped λ-calculus was introduced by Church [20] in the 1930's as a codification of the informal concept of a computable function. Unlike

the well-known machine models, such as the Turing machine or the random access machine, the λ-calculus directly codifies mathematical and programming practice. The definitive reference for all aspects of the untyped λ-calculus is Barendregt's text [10]. In particular, the proof of Scott's theorem given here is adapted from Barendregt's account. The reduction of untyped to typed λ-calculus via the concept of a recursive type was achieved by Scott in his pioneering work on the semantics of the λ-calculus [85].

Chapter 20

Dynamic Typing

We saw in Chapter 19 that an untyped language may be viewed as a unityped language in which the so-called untyped terms are terms of a distinguished recursive type. In the case of the untyped λ-calculus this recursive type has a particularly simple form, expressing that every term is isomorphic to a function. Consequently, no run-time errors can occur due to the misuse of a value—the only elimination form is application, and its first argument can only be a function. Obviously this property breaks down once more than one class of value is permitted into the language. For example, if we add natural numbers as a primitive concept to the untyped λ-calculus (rather than defining them via Church encodings), then it is possible to incur a run-time error arising from attempting to apply a number to an argument, or to add a function to a number. One school of thought in language design is to turn this vice into a virtue by embracing a model of computation that has multiple classes of value of a single type. Such languages are said to be *dynamically typed*, in purported opposition to *statically typed* languages. But the supposed opposition is illusory. Just as the untyped λ-calculus is really unityped, so dynamic languages are special cases of static languages.

20.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, called $\mathcal{L}\{\mathit{dyn}\}$. The abstract syntax of $\mathcal{L}\{\mathit{dyn}\}$ is given by the

following grammar:

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

There are two classes of values in $\mathcal{L}\{dyn\}$, the *numbers*, which have the form $\overline{n}$,[1] and the *functions*, which have the form $\lambda(x.d)$. The expressions $\mathtt{zero}$ and $\mathtt{succ}(d)$ are not in themselves values, but rather are operations that evaluate to classified values.

The concrete syntax of $\mathcal{L}\{dyn\}$ is somewhat deceptive, in keeping with common practice in dynamic languages. For example, the concrete syntax for a number is a bare numeral, $\overline{n}$, but in fact it is just a convenient notation for the classified value, $\mathtt{num}(\overline{n})$, of class $\mathtt{num}$. Similarly, the concrete syntax for a function is a λ-abstraction, $\lambda(x.d)$, which must be regarded as standing for the classified value $\mathtt{fun}(\lambda\, x.d)$ of class $\mathtt{fun}$.

The statics of $\mathcal{L}\{dyn\}$ is essentially the same as that of $\mathcal{L}\{\lambda\}$ given in Chapter 19; it merely checks that there are no free variables in the expression. The judgement

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

states that d is a well-formed expression with free variables among those in the hypothesis list.

The dynamics of $\mathcal{L}\{dyn\}$ checks for errors that would never arise in a safe statically typed language. For example, function application must ensure that its first argument is a function, signaling an error in the case that it is not, and similarly the case analysis construct must ensure that its first argument is a number, signaling an error if not. The reason for having classes labelling values is precisely to make this run-time check possible.

The value judgement, d val, states that d is a fully evaluated (closed) expression:

$$\frac{}{\mathtt{num}(\overline{n})\ \mathsf{val}} \tag{20.1a}$$

$$\frac{}{\mathtt{fun}(\lambda\, x.d)\ \mathsf{val}} \tag{20.1b}$$

[1] The numerals, $\overline{n}$, are n-fold compositions of the form $\mathtt{s}(\mathtt{s}(\ldots\mathtt{s}(\mathtt{z})\ldots))$.

The dynamics makes use of judgements that check the class of a value, and recover the underlying λ-abstraction in the case of a function.

$$\frac{}{\mathtt{num}(\overline{n})\ \mathsf{is\_num}\ \overline{n}} \tag{20.2a}$$

$$\frac{}{\mathtt{fun}(\lambda\, x.d)\ \mathsf{is\_fun}\ x.d} \tag{20.2b}$$

The second argument of each of these judgements has a special status—it is not an expression of $\mathcal{L}\{dyn\}$, but rather just a special piece of syntax used internally to the transition rules given below.

We also will need the "negations" of the class-checking judgements in order to detect run-time type errors.

$$\frac{}{\mathtt{num}(\_)\ \mathsf{isnt\_fun}} \tag{20.3a}$$

$$\frac{}{\mathtt{fun}(\_)\ \mathsf{isnt\_num}} \tag{20.3b}$$

The transition judgement, $d \mapsto d'$, and the error judgement, d err, are defined simultaneously by the following rules:[2]

$$\frac{}{\mathtt{zero} \mapsto \mathtt{num(z)}} \tag{20.4a}$$

$$\frac{d \mapsto d'}{\mathtt{succ}(d) \mapsto \mathtt{succ}(d')} \tag{20.4b}$$

$$\frac{d\ \mathsf{is\_num}\ \overline{n}}{\mathtt{succ}(d) \mapsto \mathtt{num(s}(\overline{n})\mathtt{)}} \tag{20.4c}$$

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{succ}(d)\ \mathsf{err}} \tag{20.4d}$$

$$\frac{d \mapsto d'}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto \mathtt{ifz}(d'; d_0; x.d_1)} \tag{20.4e}$$

$$\frac{d\ \mathsf{is\_num}\ \mathtt{z}}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto d_0} \tag{20.4f}$$

$$\frac{d\ \mathsf{is\_num}\ \mathtt{s}(\overline{n})}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto [\mathtt{num}(\overline{n})/x]d_1} \tag{20.4g}$$

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{ifz}(d; d_0; x.d_1)\ \mathsf{err}} \tag{20.4h}$$

$$\frac{d_1 \mapsto d_1'}{\mathtt{dap}(d_1; d_2) \mapsto \mathtt{dap}(d_1'; d_2)} \tag{20.4i}$$

[2] The obvious error propagation rules discussed in Chapter 8 are omitted here for the sake of concision.

$$\frac{d_1 \text{ is\_fun } x.d}{\texttt{dap}(d_1;d_2) \mapsto [d_2/x]d} \tag{20.4j}$$

$$\frac{d_1 \text{ isnt\_fun}}{\texttt{dap}(d_1;d_2)\ \texttt{err}} \tag{20.4k}$$

$$\frac{}{\texttt{fix}(x.d) \mapsto [\texttt{fix}(x.d)/x]d} \tag{20.4l}$$

Rule (20.4g) labels the predecessor with the class `num` to maintain the invariant that variables are bound to expressions of $\mathcal{L}\{dyn\}$.

The language $\mathcal{L}\{dyn\}$ enjoys essentially the same safety properties as $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, except that there are more opportunities for errors to arise at run-time.

Theorem 20.1 (Safety). *If d ok, then either d val, or d err, or there exists d' such that $d \mapsto d'$.*

Proof. By rule induction on Rules (20.4). The rules are designed so that if d ok, then some rule, possibly an error rule, applies, ensuring progress. Since well-formedness is closed under substitution, the result of a transition is always well-formed. □

20.2 Variations and Extensions

The dynamic language $\mathcal{L}\{dyn\}$ defined in Section 20.1 on page 177 closely parallels the static language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ defined in Chapter 12. One discrepancy, however, is in the treatment of natural numbers. Whereas in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ the zero and successor operations are introductory forms for the type `nat`, in $\mathcal{L}\{dyn\}$ they are elimination forms that act on separately-defined numerals. The point of this representation is to ensure that there is a well-defined class of *numbers* in the language.

It is worthwhile to explore an alternative representation that, superficially, is even closer to $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. Suppose that we eliminate the expression $\texttt{num}(\overline{n})$ from the language, but retain `zero` and $\texttt{succ}(d)$, with the idea that these are to be thought of as introductory forms for numbers in the language. We are faced with the problem that such an expression is well-formed for *any* well-formed d. So, in particular, the expression $\texttt{succ}(\lambda(x.d))$ is a value, as is $\texttt{succ}(\texttt{zero})$. There is no longer a well-defined class of *numbers*, but rather two separate classes of values, zero and successor, with no assurance that the successor is of a number.

The dynamics of the conditional branch changes only slightly, as described by the following rules:

$$\frac{d \mapsto d'}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto \texttt{ifz}(d'; d_0; x\,.\,d_1)} \tag{20.5a}$$

$$\frac{d \text{ is\_zero}}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto d_0} \tag{20.5b}$$

$$\frac{d \text{ is\_succ } d'}{\texttt{ifz}(d; d_0; x\,.\,d_1) \mapsto [d'/x]d_1} \tag{20.5c}$$

$$\frac{d \text{ isnt\_zero} \quad d \text{ isnt\_succ}}{\texttt{ifz}(d; d_0; x\,.\,d_1) \text{ err}} \tag{20.5d}$$

The foregoing rules are to be augmented by the following rules that check whether a value is of class zero or successor:

$$\frac{}{\texttt{zero} \text{ is\_zero}} \tag{20.6a}$$

$$\frac{}{\texttt{succ}(d) \text{ isnt\_zero}} \tag{20.6b}$$

$$\frac{}{\texttt{succ}(d) \text{ is\_succ } d} \tag{20.6c}$$

$$\frac{}{\texttt{zero} \text{ isnt\_succ}} \tag{20.6d}$$

A peculiarity of this formulation of the conditional is that it can only be understood as distinguishing `zero` from `succ(_)`, rather than as distinguishing `zero` from non-`zero`. The reason is that if d is not `zero`, it might be either a successor or a function, and hence its "predecessor" is not well-defined.

Similar considerations arise when enriching $\mathcal{L}\{dyn\}$ with structured data. The classic example is to enrich the language as follows:

Expr	d	::=	`nil`	`nil` null
			$\texttt{cons}(d_1; d_2)$	$\texttt{cons}(d_1; d_2)$ pair
			$\texttt{ifnil}(d; d_0; x, y\,.\,d_1)$	$\texttt{ifnil}\, d\, \{\texttt{nil} \Rightarrow d_0 \mid \texttt{cons}(x; y) \Rightarrow d_1\}$ conditional

The expression $\texttt{ifnil}(d; d_0; x, y\,.\,d_1)$ distinguishes the null structure from the pair of two structures. We leave to the reader the exercise of formulating the dynamics of this extension.

An advantage of dynamic typing is that the constructors `nil` and `cons(`d_1`;`d_2`)` are sufficient to build unbounded, as well as bounded, data structures such as lists or trees. For example, the list consisting of three zero's may be represented by the value

```
cons(zero;cons(zero;cons(zero;nil))).
```

But what to make of this beast?

```
cons(zero;cons(zero;cons(zero;λ(x)x))).
```

It is a perfectly valid expression, but does not correspond to any natural data structure.

The disadvantage of this representation becomes apparent as soon as one wishes to define operations on lists, such as the append function:

```
fix a is λ(x.λ(y.ifnil(x;y;x1,x2.cons(x1;a(x2)(y)))))
```

What if x is the second list-like value given above? As it stands, the append function will signal an error upon reaching the function at the end of the list. If, however, y is this value, no error is signalled. This asymmetry may seem innocuous, but it is only one simple manifestation of a pervasive problem with dynamic languages: it is impossible to state within the language even the most rudimentary assumptions about the inputs, such as the assumption that both arguments to the append function ought to be genuine lists.

The conditional expression `ifnil(`d`;`d_0`;`x`,`y`.`d_1`)` is rather *ad hoc* in that it makes a distinction between `nil` and all other values. Why not distinguish successors from non-successors, or functions from non-functions? A more systematic approach is to enrich the language with *predicates* and *destructors*. Predicates determine whether a value is of a specified class, and destructors recover the value labelled with a given class.

Expr	d	::=	`cond(`d`;`d_0`;`d_1`)`	conditional
			`nil?(`d`)`	nil test
			`cons?(`d`)`	pair test
			`car(`d`)`	first projection
			`cdr(`d`)`	second projection

The conditional `cond(`d`;`d_0`;`d_1`)` distinguishes d between `nil` and *all other values*. If d is not `nil`, the conditional evaluates to d_0, and otherwise evaluates to d_1. In other words the value `nil` represents boolean falsehood,

and all other values represent boolean truth. The predicates `nil?(`d`)` and `cons?(`d`)` test the class of their argument, yielding `nil` if the argument is not of the specified class, and yielding some non-`nil` if so. The destructors `car(`d`)` and `cdr(`d`)`[3] decompose `cons(`d_1`;`d_2`)` into d_1 and d_2, respectively. As an example, the append function may be defined using predicates as follows:

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

20.3 Critique of Dynamic Typing

The safety theorem for $\mathcal{L}\{dyn\}$ is often promoted as an advantage of dynamic over static typing. Unlike static languages, which rule out some candidate programs as ill-typed, essentially every piece of abstract syntax in $\mathcal{L}\{dyn\}$ is well-formed, and hence, by Theorem 20.1 on page 180, has a well-defined dynamics. But this can also be seen as a disadvantage, since errors that could be ruled out at compile time by type checking are not signalled until run time in $\mathcal{L}\{dyn\}$. To make this possible, the dynamics of $\mathcal{L}\{dyn\}$ must enforce conditions that need not be checked in a statically typed language.

Consider, for example, the addition function in $\mathcal{L}\{dyn\}$, whose specification is that, when passed two values of class `num`, returns their sum, which is also of class `num`:[4]

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

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

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

It is deceptive, because the concrete syntax obscures the class tags on values, and obscures the use of primitives that check those tags. Let us now examine the costs of these operations in a bit more detail.

First, observe that the body of the fixed point expression is labelled with class `fun`. The dynamics of the fixed point construct binds p to this function. This means that the dynamic class check incurred by the application of p in

[3] This terminology for the projections is archaic, but firmly established in the literature.

[4] This specification imposes no restrictions on the behavior of addition on arguments that are not classified as numbers, but one could make the further demand that the function abort when applied to arguments that are not classified by `num`.

the recursive call is guaranteed to succeed. But $\mathcal{L}\{dyn\}$ offers no means of suppressing this redundant check, because it cannot express the invariant that p is always bound to a value of class `fun`.

Second, observe 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 is guaranteed 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 observe that it is otherwise a loop invariant that the result of applying the inner function is of this class. However, $\mathcal{L}\{dyn\}$ 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 $\mathcal{L}\{dyn\}$, and hence there is no way to avoid the class check imposed by the conditional branch.

Classification is not free—storage is required for the class label, and it takes time to detach the class from a value each time it is used and to attach a class to a value whenever it is created. Although the overhead of classification is not asymptotically significant (it slows down the program only by a constant factor), it is nevertheless non-negligible, and should be eliminated whenever possible. But this is impossible within $\mathcal{L}\{dyn\}$, because it cannot enforce the restrictions required to express the required invariants. For that we need a static type system.

20.4 Notes

The earliest dynamically typed language is Lisp [56], which continues to influence language design a half decade after its invention. Dynamic PCF is essentially the core of Lisp, but with a proper treatment of variable binding, correcting what McCarthy himself as described as an error in the original. Informal discussions of dynamic languages are often confused by the ellision of the dynamic checks that are made explicit here. While the surface syntax of dynamic PCF is essentially the same as that for PCF, minus the type annotations, the underlying dynamics is fundamentally different. It

is for this reason that static PCF cannot be properly seen as a restriction of dynamic PCF by the imposition of a type system, contrary to what seems to be a widely held belief. It is simply not accurate to state that a static type sytem is a *post hoc* restriction imposed on a dynamically typed language.

Chapter 21

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 considered in Chapter 20 may be embedded into the hybrid language by regarding a dynamically typed program as a statically typed program of type `dyn`. This shows that static and dynamic types are not opposed to one another, but may coexist harmoniously.

The notion of a hybrid language, however, is itself illusory, because the type `dyn` is really a particular recursive type. This shows that there is no need for any special mechanisms to support dynamic typing. Rather, they may be derived from the more general concept of a recursive type. Moreover, this shows that *dynamic typing is but a mode of use of static typing*! The supposed opposition between dynamic and static typing is, therefore, a fallacy: dynamic typing can hardly be opposed to that of which it is but a special case!

21.1 A Hybrid Language

Consider the language $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$, which extends $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ (defined in Chapter 12) with the following additional constructs:

Type	τ	::=	$\texttt{dyn}$	$\texttt{dyn}$	dynamic
Expr	e	::=	$\texttt{new}[l](e)$	$l \cdot e$	construct
			$\texttt{cast}[l](e)$	$e \cdot l$	destruct
Class	l	::=	$\texttt{num}$	$\texttt{num}$	number
			$\texttt{fun}$	$\texttt{fun}$	function

The type dyn is the type of dynamically classified values. The new operation attaches a classifier to a value, and the cast operation checks the classifier and returns the associated value.

The statics of $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ extends that of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the following additional rules:

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{new[num]}\,(e) : \texttt{dyn}} \tag{21.1a}$$

$$\frac{\Gamma \vdash e : \texttt{dyn} \rightharpoonup \texttt{dyn}}{\Gamma \vdash \texttt{new[fun]}\,(e) : \texttt{dyn}} \tag{21.1b}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[num]}\,(e) : \texttt{nat}} \tag{21.1c}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[fun]}\,(e) : \texttt{dyn} \rightharpoonup \texttt{dyn}} \tag{21.1d}$$

The statics ensures that class labels are applied to objects of the appropriate type, namely num for natural numbers, and fun for functions defined over labelled values.

The dynamics of $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ extends that of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the following rules:

$$\frac{e \ \mathsf{val}}{\texttt{new}[l]\,(e) \ \mathsf{val}} \tag{21.2a}$$

$$\frac{e \mapsto e'}{\texttt{new}[l]\,(e) \mapsto \texttt{new}[l]\,(e')} \tag{21.2b}$$

$$\frac{e \mapsto e'}{\texttt{cast}[l]\,(e) \mapsto \texttt{cast}[l]\,(e')} \tag{21.2c}$$

$$\frac{\texttt{new}[l]\,(e) \ \mathsf{val}}{\texttt{cast}[l]\,(\texttt{new}[l]\,(e)) \mapsto e} \tag{21.2d}$$

$$\frac{\texttt{new}[l']\,(e) \ \mathsf{val} \quad l \neq l'}{\texttt{cast}[l]\,(\texttt{new}[l']\,(e)) \ \mathsf{err}} \tag{21.2e}$$

Casting compares the class of the object to the required class, returning the underlying object if these coincide, and signalling an error otherwise.

Lemma 21.1 (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 a straightforward rule induction on the statics of $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$. □

Theorem 21.2 (Safety). *The language $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ is safe:*

1. *If* $e : \tau$ *and* $e \mapsto e'$, *then* $e' : \tau$.

2. *If* $e : \tau$, *then either* e val, *or* e err, *or* $e \mapsto e'$ *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 $\mathcal{L}\{dyn\}$—a well-typed cast might fail at run-time if the class of the cast does not match the class of the value. □

21.2 Optimization of Dynamic Typing

The language $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ combines static and dynamic typing by enriching $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the type, `dyn`, of classified values. It is, for this reason, called a *hybrid* language. Unlike a purely dynamic type system, a hybrid type system can express invariants that are crucial to the optimization of programs in $\mathcal{L}\{dyn\}$.

Let us examine this in the case of the addition function, which may be defined in $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ as follows:

$$\texttt{fun} \cdot \lambda\,(x{:}\texttt{dyn}.\,\texttt{fix}\,p{:}\texttt{dyn}\,\texttt{is}\,\texttt{fun} \cdot \lambda\,(y{:}\texttt{dyn}.\,e_{x,p,y})),$$

where

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

is defined to be the expression

$$\texttt{ifz}\,(y \cdot \texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num} \cdot (\texttt{s}((p \cdot \texttt{fun})(\texttt{num} \cdot y') \cdot \texttt{num}))\}.$$

This is a reformulation of the dynamic addition function given in Section 20.3 on page 183 in which we have made explicit the checking and imposition of classes on values. We will exploit the static type system of $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ to optimize this dynamically typed implementation of addition in accordance with the specification given in Section 20.3 on page 183.

First, note that the body of the `fix` expression is an explicitly labelled function. This means that when the recursion is unwound, the variable p is bound to this value of type `dyn`. Consequently, the check that p is labelled with class `fun` is redundant, and can be eliminated. This is achieved by rewriting the function as follows:

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

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

$$\texttt{ifz}\,(y \cdot \texttt{num})\,\{\texttt{zero} \Rightarrow x \mid \texttt{succ}(y') \Rightarrow \texttt{num} \cdot (\texttt{s}(p(\texttt{num} \cdot y') \cdot \texttt{num}))\}.$$

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 $\texttt{dyn} \rightharpoonup \texttt{dyn}$, reflecting that the body is now a "bare function", rather than a labelled function value of type dyn.

Next, observe that the parameter y of type dyn is cast to a number on each iteration of the loop before it is tested for zero. Since this function is recursive, the bindings of y arise in one of two ways, at the initial call to the addition function, and on each recursive call. But the recursive call is made on the predecessor of y, which is a true natural number that is labelled with num at the call site, only to be removed by the class check at the conditional on the next iteration. This suggests that we hoist the check on y outside of the loop, and avoid labelling the argument to the recursive call. Doing so changes the type of the function, however, from $\texttt{dyn} \rightharpoonup \texttt{dyn}$ to $\texttt{nat} \rightharpoonup \texttt{dyn}$. Consequently, further changes are required to ensure that the entire function remains well-typed.

Before doing so, let us make another observation. The result of the recursive call is checked to ensure that it has class num, and, if so, the underlying value is incremented and labelled with class num. If the result of the recursive call came from an earlier use of this branch of the conditional, then obviously the class check is redundant, because we know that it must have class num. But what if the result came from the other branch of the conditional? In that case the function returns x, which need not be of class num because it is provided by the caller of the function. However, we may reasonably insist that it is an error to call addition with a non-numeric argument. This canbe enforced by replacing x in the zero branch of the conditional by $x \cdot \texttt{num}$.

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

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

This function has type $\texttt{nat} \rightharpoonup \texttt{nat}$, and runs at full speed when applied to a natural number—all checks have been hoisted out of the inner loop.

Finally, recall that the overall goal is to define a version of addition that works on values of type dyn. Thus we require a value of type $\texttt{dyn} \rightharpoonup \texttt{dyn}$, but what we have at hand is a function of type $\texttt{nat} \rightharpoonup \texttt{nat}$. This can be

converted to the required form by pre-composing with a cast to `num` and post-composing with a coercion to `num`:

$$\texttt{fun} \cdot \lambda\,(x{:}\texttt{dyn}.\,\texttt{fun} \cdot \lambda\,(y{:}\texttt{dyn}.\,\texttt{num} \cdot (e_x''(y \cdot \texttt{num})))).$$

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`.

21.3 Static "Versus" Dynamic Typing

There are many attempts to distinguish dynamic from static typing, all of which are misleading or wrong. For example, it is often said that static type systems associate types with variables, but dynamic type systems associate types with values. This oft-repeated characterization appears to be justified by the absence of type annotations on λ-abstractions, and the presence of classes on values. But it is based on a confusion of classes with types—the *class* of a value (`num` or `fun`) is not its *type*. Moreover, a static type system assigns types to values just as surely as it does to variables, so the description fails on this account as well.

Another way to differentiate dynamic from static languages is to say that whereas static languages check types at compile time, dynamic languages check types at run time. But to say that static languages check types statically is to state a tautology, and to say that dynamic languages check types at run-time is to utter a falsehood. Dynamic languages perform *class checking*, not *type checking*, at run-time. For example, application checks that its first argument is labelled with `fun`; it does not type check the body of the function. Indeed, at no point does the dynamics compute the *type* of a value, rather it checks its class against its expectations before proceeding. Here again, a supposed contrast between static and dynamic languages evaporates under careful analysis.

Another characterization is to assert that dynamic languages admit heterogeneous collections, whereas static languages admit only homogeneous collections. For example, in a dynamic language the elements of a list may be of disparate *classes*, as illustrated by the expression

```
cons(s(z);cons(λ(λ(x.x));nil)).
```

But they are nevertheless all of the same *type*! Put the other way around, a static language with a dynamic type is just as capable of representing a heterogeneous collection as is a dynamic language with only one type.

What, then, are we to make of the traditional distinction between dynamic and static languages? Rather than being in opposition to each other, we see that *dynamic languages are a mode of use of static languages*. If we have a type `dyn` in the language, then we have all of the apparatus of dynamic languages at our disposal, so there is no loss of expressive power. But there is a very significant gain from embedding dynamic typing within a static type discipline! We can avoid much of the overhead of dynamic typing by simply limiting our use of the type `dyn` in our programs, as was illustrated in Section 21.2 on page 189.

21.4 Reduction to Recursive Types

The type `dyn` codifies the use of dynamic typing within a static language. Its introduction form labels an object of the appropriate type, and its elimination form is a (possibly undefined) casting operation. Rather than treating `dyn` as primitive, we may derive it as a particular use of recursive types, according to the following definitions:

$$\texttt{dyn} = \mu t.\,[\texttt{num}:\texttt{nat}, \texttt{fun}:t \rightharpoonup t] \tag{21.3}$$

$$\texttt{new}[\texttt{num}]\,(e) = \texttt{fold}(\texttt{num}\cdot e) \tag{21.4}$$

$$\texttt{new}[\texttt{fun}]\,(e) = \texttt{fold}(\texttt{fun}\cdot e) \tag{21.5}$$

$$\texttt{cast}[\texttt{num}]\,(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num}\cdot x \Rightarrow x \mid \texttt{fun}\cdot x \Rightarrow \texttt{error}\} \tag{21.6}$$

$$\texttt{cast}[\texttt{fun}]\,(e) = \texttt{case}\,\texttt{unfold}(e)\,\{\texttt{num}\cdot x \Rightarrow \texttt{error} \mid \texttt{fun}\cdot x \Rightarrow x\} \tag{21.7}$$

One may readily check that the static and dynamics for the type `dyn` are derivable according to these definitions.

This encoding readily generalizes to any number of classes of values: we need only consider additional summands corresponding to each class. For example, to account for the constructors `nil` and $\texttt{cons}(d_1; d_2)$ considered in Chapter 20, the definition of `dyn` is expanded to the recursive type

$$\mu t.\,[\texttt{num}:\texttt{nat}, \texttt{fun}:t \rightharpoonup t, \texttt{nil}:\texttt{unit}, \texttt{cons}:t \times t],$$

with corresponding definitions for the `new` and `cast` operations. This exemplifies the general case: dynamic typing is a mode of use of static types in which classes of values are simply names of summands in a recursive type of dynamic values.

21.5 Notes

The concept of a "hybrid" type system is wholly artificial. It is introduced here as an explanatory bridge between dynamic and static languages. The reality is that dynamic languages are statically typed. The only point of discussing hybrid typing is to show that one can expose more or less of the underlying static type system in a so-called dynamic language. In the general case dynamic typing is but a particular mode of use of static typing, rather than being, as commonly thought, opposed to it. This point of view is essentially due to Scott [85], who also proposed replacing the word "untyped" by "unityped" in programming languages.