Chapter 6

Types and declarations

6.1 Types

The only types we have met so far are various numbers, like the reals $\mathbb{R}$, the integers $\mathbb{Z}$ and the natural numbers $\mathbb{N}$. They are examples that we borrow directly from mathematics, more or less taking their existence for granted. Figure 6.1 gives other examples of standard mathematical types; but in general we can use *any* set as a type.

In code, however, the available types are restricted. We make the (idealised) assumption that types $\mathbb{N}$ and $\mathbb{Z}$, at least, are available in code, with the intention that they correspond roughly to types `INTEGER` or *int* in some everyday programming language. Other types can be constructed from them, as we will see in Chapters 9 and 15. The empty type $\{\}$ is not code, however.

Every type brings with it certain functions and relations which can be applied to its elements. For the types of Figure 6.1 we may use all those from arithmetic, some examples of which are given in Figures 6.2 and 6.3.

Why do we bother with types? They affect program development in several ways. One way is that types provide information about the possible values that variables can take, and the information can be used to make program development easier. For example, the following refinement is not valid in general, but it is valid if m and n are known to be natural numbers, elements of $\mathbb{N}$:

$$n\colon [m \neq 0\ ,\ 0 \leq n < m] \quad \sqsubseteq \quad n\colon [m > 0\ ,\ n < m]\ .$$

The precondition has been weakened because $m \neq 0 \Rrightarrow m > 0$ for any m in $\mathbb{N}$. The postcondition has been strengthened because $n < m \Rrightarrow 0 \leq n < m$ for any n in $\mathbb{N}$. (Actually the two programs are equal, which is a special case of refinement.) Thus having the types declared means that the fact $m, n \in \mathbb{N}$ is available anywhere the usual scope rules allow, and we do not have to carry the information explicitly from place to place.

Another way is that types restrict the values that can be assigned to variables, and that makes program development harder: there are fewer assignments to

$\mathbb{N}$ The *natural numbers*, or non-negative integers.

$\mathbb{Z}$ The *integers*, positive, negative, and zero.

$\mathbb{Q}$ The *rational numbers*.

$\mathbb{R}$ The *real numbers*.

$\mathbb{C}$ The *complex numbers*.

Each of the types is a proper subset of the one below it: thus $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$. For any type T above except the complex numbers, we write T^+ for the set of *positive* elements of the type, and T^- for the set of *negative* elements of the type. Thus $\mathbb{N}^+ = \mathbb{Z}^+ = \{1, 2, 3, \cdots\}$, and $\mathbb{Z}^- = \{-1, -2, -3, \cdots\}$.

Figure 6.1 Some standard mathematical types

choose from. For example, the code $m := -1$ should not occur in a program where m has type $\mathbb{N}$. That is partly because everyday languages have explicit typing, which therefore we must accommodate.

6.2 Declarations

6.2.1 Variable declarations

To associate a type with a variable we use a *variable declaration*, and for variable x and type T that is written **var** $x : T$. We also have multiple declarations like this:

$$\textbf{var } x, y : T;\ \ z : U\ .$$

It declares x and y to have type T, and z to have type U.

We have earlier met informal declarations, such as at the beginning of Chapter 1: 'let our variables be just x, y, and z; and let their values be real numbers' (p.4). Now we could write that **var** $x, y, z : \mathbb{R}$.

© Carroll Morgan 1990, 1994, 1998

$+$	Addition.
$-$	Subtraction.
$\times$	Multiplication. We allow the conventional $2n$ to abbreviate $2 \times n$.
$/$	Division. Note that dividing two integers does not necessarily yield an integer.
$\lceil\ \rceil$	Ceiling: the least integer no less than.
$\lfloor\ \rfloor$	Floor: the greatest integer no more than.
$\div$	Integer division: $a \div b = \lfloor a/b \rfloor$.
$\ominus$	Natural number subtraction: $a \ominus b = a - b$, provided $a - b \geq 0$.
mod	Modulus: $a = b \times (a \div b) + (a \,\mathsf{mod}\, b)$, provided $b \neq 0$.
abs	Absolute value.
$\sqcup$	Maximum.
$\sqcap$	Minimum.

Figure 6.2 Some standard arithmetic functions

$<$	Less than.
$\leq$	Less than or equal to.
$>$	Greater than.
$\geq$	Greater than or equal to.
$\vert$	Divides exactly: $(a \vert b) \equiv (b \,\mathsf{mod}\, a = 0)$.

Figure 6.3 Some standard arithmetic relations

6.2.2 Invariant declarations

A more advanced form of declaration is the local invariant: instead of writing **var** $x : \mathbb{N}$ we could write

$$\textbf{var}\ x : \mathbb{Z};\ \textbf{and}\ x \geq 0\ .$$

© Carroll Morgan 1990, 1994, 1998

We could also write just **var** x **and** $x \in \mathbb{N}$, using an untyped declaration of x.

A *local invariant* is any formula written after the keyword **and**, as a declaration. The declaration

and *inv*

allows us subsequently to assume *inv*, as well as any typing information, when applying laws or definitions.

In fact the above examples make it clear that local invariants subsume types: typing a variable just makes invariant that it is an element of its type.

We can write more interesting invariants too. The declaration **and** $2 \mid n$ means that n must always be even. And we can relate different variables to each other; for example, the following declares a rational q, and two integers n and d that always represent it as a fraction:

var $q : \mathbb{Q};\ \ n : \mathbb{Z};\ \ d : \mathbb{N};$ **and** $q = n/d$.

Finally, invariants can be used to make constants: given a type declaration $g : \mathbb{R}^+$, the additional declaration **and** $g^2 - g - 1 = 0$ makes g the golden ratio $1.618\cdots$. Unlike the conventional constant declaration (for example **const** $g = 1.618$), the distinguishing property of g is declared as well.

Unfortunately, local invariants are not code: they are useful during development, but must at some stage be removed. We return to that later, in Section 6.6.

6.2.3 Logical constants

We have seen in some earlier exercises that it is sometimes useful to have a name, not necessarily a normal program variable, that can be used to refer to values of interest during a development (Exercises 5.2, 5.7 and 5.3). For example, although it is easy enough to write the specification $f, n := n!, ?$ for a program that sets f to the factorial of n (possibly changing n), during the development one might need an invariant along the lines of

$n! \times f \quad = \quad$ 'the factorial of the initial value of n'.

Rewriting the specification as $f, n\colon [F = n!\ ,\ f = n_0!]$ is a step in the right direction, since — as long as we do not change F — we can now write the invariant as

$$n! \times f \quad = \quad F\ .$$

In fact, a slightly neater specification would be just $f, n\colon [F = n!\ ,\ f = F]$ (as in Exercise 5.3).

But, strictly speaking, a precondition $F = n!$ means 'abort if $F \neq n!$', and that certainly is not what we want: we need F to take a value such that the precondition holds. In this case that value would be $n!$.

© Carroll Morgan 1990, 1994, 1998

That 'taking a value such that the precondition holds' is what logical constants are for. Like variables, they are declared — but we indicate their different nature with the keyword **con**: the notation

con F

declares F to be a logical constant, rather than a variable. (Since we declare logical constants explicitly, we can use either upper or lower case for them — but upper case is conventional.)

Unlike **var**, a logical constant declaration **con** is not code, and hence at some later stage of refinement it must be removed. Naturally, that can occur only when all references to those logical constants have been eliminated, since otherwise they would become undeclared. And since logical constants are not code, typing for them is optional.

Our specification above is thus to be interpreted in the context of the declarations **var** $f, n : \mathbb{N}$; **con** F. The development, incidentally, is then

$$
\begin{array}{ll}
 & f, n\colon [F = n!\ ,\ f = F] \\
\sqsubseteq & \text{“establish invariant”} \\
 & f := 1; \\
 & f, n\colon [F = n! \times f\ ,\ F = n! \times f \wedge n = 0] \qquad \triangleleft \\
\sqsubseteq & \text{“Note we assume } n \geq 0 \text{ because of its declaration.”} \\
 & \mathbf{do}\ n \neq 0 \rightarrow \\
 & \qquad f, n\colon [n > 0\ ,\ F = n! \times f\ ,\ n < n_0] \qquad \triangleleft \\
 & \mathbf{od} \\
\sqsubseteq & \text{“How do we know } n \text{ stays non-negative?”} \\
 & f\colon [n > 0 \wedge F = n! \times f\ ,\ F = (n-1)! \times f]\,; \qquad \triangleleft \\
 & n := n - 1 \\
\sqsubseteq & \text{“And now the logical constant } F \text{ disappears, as it must.”} \\
 & f := f \times n\ .
\end{array}
$$

(What refinement rules were used in the above?)

6.3 Local blocks

6.3.1 Variables and invariants

Declarations of variables, invariants and logical constants are made within local blocks, which indicate precisely the part of the program affected. A *local block* is a program fragment enclosed in the block brackets |[and]|; any of the declarations of Section 6.2 may be placed immediately after the opening bracket. They are separated from the following program, the block *body*, by a spot • .

© Carroll Morgan 1990, 1994, 1998

Declarations are limited in effect to the block in which they are declared; such limitations are necessary, for example, whenever a programmer uses extra variables not mentioned in the specification agreed with his client. The specification

$$x, y\colon [x = X \wedge y = Y \ , \ x = Y \wedge y = X] \ ,$$

which swaps the values of x and y, is *not* refined by

$$t := x; \ \ x := y; \ \ y := t \ .$$

(Compare Section 3.7, where t was in the frame.)

The specification does not allow t to change, and so its implementation must not either. We must use a local block, and a correct refinement is

$$|[\ \mathbf{var}\ t : T \bullet t := x; \ \ x := y; \ \ y := t\ ]| \ ,$$

where T is the type of x and y. The variable t is significant only within the local block; it is wholly divorced from any variable t declared *outside* the local block. Its initial value is an arbitrary member of its type.

The names of local variables can be systematically altered throughout the block in which they are declared — thus this program also swaps x and y:

$$|[\ \mathbf{var}\ s : T \bullet s := x; \ \ x := y; \ \ y := s\ ]| \ .$$

It is because of that essential arbitrariness in the name of a local variable that it is clear it cannot have anything to do with variables declared outside the block.

In general examples like the above, we may omit types from declarations; in that case, we assume all variables to have the same type.

Here is the law for introducing a local variable, and optionally a local invariant as well:

Law 6.1 <u>introduce local variable</u> If x does not occur in w, *pre* or *post* then

$$w\colon [pre \ , \ post] \quad \sqsubseteq \quad |[\ \mathbf{var}\ x : T;\ \mathbf{and}\ inv \bullet w, x\colon [pre \ , \ post]\ ]| \ .$$

□

It is the proviso of Law 6.1 — that x is 'fresh' — that ensures there is no confusion between the 'new' x and any 'existing' x's. Without it, a reference to an existing x could be captured by the new declaration.

The invariant part of Law 6.1 is optional, of course, and is considered to be true if omitted. Remember however that an invariant is not code, and so cannot appear in the final program. (See Section 6.6.)

For laws like *introduce local variable* 6.1, which introduce a block, we can use an abbreviation when setting out refinements: the declaration decorates the refinement step, and the block brackets are omitted. We use the spot $\bullet$ again to separate the decoration from the refined program. Thus we have this alternative layout of the law:

© Carroll Morgan 1990, 1994, 1998

$$\begin{array}{l} \quad w\colon [pre\ ,\ post] \\ \sqsubseteq\ \textbf{var}\ x : T;\ \textbf{and}\ inv\bullet \\ \\ \quad w, x\colon [pre\ ,\ post]\,. \end{array}$$

The advantage of that is a more concise notation, no need for block brackets during the development, and no indentation. (The block brackets are required only if we collect the code and present it linearly: similarly '$2 \times (3+4)$' needs parentheses, although expression trees do not contain them.)

6.3.2 Logical constants and blocks

Logical constants are introduced, in general, with this law:

Law 6.2 <u>introduce logical constant</u> If $pre \Rrightarrow (\exists\, c : T \bullet pre')$, and c does not occur in w, pre or $post$, then

$$\begin{array}{l} \quad w\colon [pre\ ,\ post] \\ \sqsubseteq\ \textbf{con}\ c : T\bullet \\ \quad w\colon [pre'\ ,\ post]\,. \end{array}$$

If the optional type T is omitted, then the quantification in the proviso should be untyped.

□

Note that we use the abbreviated layout, just as for **var**.

Later we shall see direct applications of Law 6.2; but more often we use this more specialised law, an immediate consequence of it:

Law 6.3 <u>fix initial value</u> For any term E such that $pre \Rrightarrow E \in T$, and fresh name c,

$$\begin{array}{l} \quad w\colon [pre\ ,\ post] \\ \sqsubseteq\ \textbf{con}\ c : T\bullet \\ \quad w\colon [pre \wedge c = E\ ,\ post]\,. \end{array}$$

Proof: Law 6.2 requires

$$\begin{array}{ll} & pre \\ \Rrightarrow & pre \wedge E \in T \\ \equiv & \text{“}c\text{ fresh”} \\ & (pre \wedge c \in T)[c\backslash E] \\ \equiv & \text{“Predicate law A.56”} \\ & (\exists\, c \bullet pre \wedge c \in T \wedge c = E) \\ \equiv & (\exists\, c : T \bullet pre \wedge c = E)\ . \end{array}$$

□

© Carroll Morgan 1990, 1994, 1998

Law 6.3 is used when it is necessary later in a development to refer to the value that some term E had initially, where the constituent variables of E may be changed by assignments; the logical constant c retains that initial value. Returning to our factorial example above, we could start from f, n: $[f = n_0!]$, and use *fix initial value* 6.3 to introduce F, as follows:

$$\begin{aligned}
&\sqsubseteq f, n\colon [f = n_0!] \\
&\sqsubseteq \textbf{con } F \bullet \\
&\quad f, n\colon [F = n!\ ,\ f = n_0!] \\
&\sqsubseteq \text{"\textit{strengthen postcondition} 5.1"} \\
&\quad f, n\colon [F = n!\ ,\ f = F]\ .
\end{aligned}$$

In practice, of course, we would not dream of setting out all that detail: it would be sufficient to go from f, n: $[f = n_0!]$ (or even $f, n := n!, ?$) to f, n: $[F = n!\ ,\ f = F]$ in one step directly, quoting '**con** F' as the justification.

Finally, for removing logical constants, we have this law:

Law 6.4 remove logical constant If c occurs nowhere in program *prog*, then

$$|[\ \textbf{con}\ c : T \bullet \mathit{prog}\ ]| \quad \sqsubseteq \quad \mathit{prog}\ .$$

□

Law 6.4 is the justification for removing declarations **con** c when all occurrences of c have been removed. We will not use it explicitly, however. (We could formulate a similar law for variables, but since they *are* code it would be very seldom that we would want to remove them.)

6.4 Using types and invariants

Within a local block such as

$$|[\ \textbf{var}\ x : T;\ \ \textbf{and}\ \mathit{inv} \bullet \cdots\ ]|\ ,$$

the formula $x \in T \wedge \mathit{inv}$ may be used when proving the provisos of refinement rules. Within several nested blocks, all the enclosing formulae may be used. They are known collectively as the *context.*

Thus in our original example,

$$n\colon [m \neq 0\ ,\ 0 \leq n < m] \quad \sqsubseteq \quad n\colon [m > 0\ ,\ n < m]\ ,$$

we may use the context $m, n \in \mathbb{N}$. The provisos for the two refinements, weakening the precondition and strengthening the postcondition, are then

© Carroll Morgan 1990, 1994, 1998

$$\begin{array}{lll} m \in \mathbb{N} \wedge m \neq 0 & \Rrightarrow & m > 0 \\ n \in \mathbb{N} \wedge n < m & \Rrightarrow & 0 \leq n < m \ . \end{array}$$

Naturally we need not write in the whole context: only $m \in \mathbb{N}$ was required in the first case, and $n \in \mathbb{N}$ in the second.

A slight difficulty arises if there is danger of variable capture, as in this example:

$$\begin{array}{l} |[\ \mathbf{var}\ a, b : \mathbb{N} \bullet \\ \quad |[\ \mathbf{var}\ a : \mathbb{Z} \bullet \\ \qquad \cdots \\ \quad]| \\ ]| . \end{array}$$

At the point $\cdots$, in the inner block, we cannot refer to the context formula $a \in \mathbb{N}$, because its free variable a would be captured by the inner declaration. One must either rename the inner bound variable, or use only the weaker $b \in \mathbb{N}$ from the outer declaration. In general, we can always use a weaker context than the one we are given, and that allows us to use $a \in \mathbb{Z} \wedge b \in \mathbb{N}$, for example, in the inner block.

6.5 A final note on feasibility

With types and invariants now explicit, and initial variables, we can present a more general definition of feasibility that takes them all into account. It is

Definition 6.5 feasibility The specification $w\colon [pre\ ,\ post]$ is *feasible* in context *inv* iff

$$(w = w_0) \wedge pre \wedge inv \quad \Rrightarrow \quad (\exists\, w : T \bullet inv \wedge post)\,,$$

where T is the type of w.

□

As an example, recall that we have seen that the specification

$$y\colon \left[y^2 = x\right] \tag{6.1}$$

is infeasible if x, y range over $\mathbb{R}$: the formula $(\exists\, y : \mathbb{R} \bullet y^2 = x)$ is not true for all x in $\mathbb{R}$. But if x, y had type $\mathbb{C}$, the feasibility formula from Definition 6.5 would reduce to $x \in \mathbb{C} \Rrightarrow (\exists\, y : \mathbb{C} \bullet y^2 = x)$ instead, quite a different matter.

6.6 Checking types and invariants

Section 6.4 explained the use of context in checking the provisos of refinements, and how that makes proposed refinements more likely to be valid. But there is a price to pay: at some stage we must check that those types are respected and

© Carroll Morgan 1990, 1994, 1998

invariants maintained. We call that activity *type checking*, and say that a program successfully checked is *well-typed.* If it fails the check, we call it *ill-typed.*

Experience suggests that type checking is best done at the end of development, rather than during it. Often it is obvious that a program is well-typed; and if types are used in a reasonable way, much of type checking can be done automatically by computer.

We must type-check both types and invariants; we deal first with types. Only assignments can violate types, and so we must be able to tell whether any assignment is well- or ill-typed. In an assignment $w := E$, we always know the type of w because that is given in its declaration. We arrange, as explained below, that we always know a type for E as well. If the type of w is T, and of E is U, then the assignment is well-typed if $U \subseteq T$.

A type for any expression can be deduced provided we know the types of the variables and constants that it contains, and provided the operators have certain properties. For example, we know that the natural numbers are closed under addition, and that 1 is a natural number; therefore we know that in a context containing $a \in \mathbb{N}$ the expression $a + 1$ has type $\mathbb{N}$. Therefore $a := a + 1$ is well-typed. The same applies to $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$: the constant 1 is an element of them all, and they all are closed under addition.

But type checking is not always so straightforward. Given $m, n : \mathbb{N}$, consider this refinement:

$$m\colon [n > 0\ ,\ m = n - 1] \quad \sqsubseteq \quad m := n - 1 .$$

Though the refinement is valid, the assignment is ill-typed, because $n \in \mathbb{N} \not\Rrightarrow n - 1 \in \mathbb{N}$. Therefore we do not allow subtraction '$-$' of natural numbers in assignments.

For subtraction of natural numbers we use instead the operator $\ominus$, which agrees with ordinary subtraction '$-$' as far as possible: $2 \ominus 1 = 2 - 1 = 1$. But $1 - 2$ is not a natural number, while $1 \ominus 2$ *is* a natural number (though we choose not to know which one): therefore they cannot be equal. Still, in the situation above we have this alternative refinement:

$$m\colon [n > 0\ ,\ m = n - 1] \quad \sqsubseteq \quad m := n \ominus 1 .$$

It too is valid, and this time the resulting assignment is well-typed. The validity rests on the proviso required by *assignment* 1.3, which for the above is

$$n > 0 \quad \Rrightarrow \quad n \ominus 1 = n - 1 .$$

That is true given our declaration $m, n : \mathbb{N}$.

Operationally, one would argue that when n is 0 initially, and thus $m := n \ominus 1$ assigns an unknown natural number to m, the refinement still holds: the precondition of the left-hand side is false.

Now we turn to the type-checking of local invariants. Recall that declarations '**and** *inv*' are not code, and so must be removed once they have served their

© Carroll Morgan 1990, 1994, 1998

purpose (of providing extra context). There are general laws for that; but we do not show them, because our use of invariants will be very modest. Either they refer to variables that are never changed — that do not appear in any frame — or we use operators that maintain them trivially. Under those strong conditions, they can be removed without further checking; we draw attention to that as it arises below (for example, on page 100).

6.7 Undefined expressions

Consider this refinement:

$$x\colon [x = 1/0] \quad \sqsubseteq \quad x := 1/0.$$

Since the specification terminates (precondition true), so must the assignment. Yet that is not the conventional view. Usually, division by 0 is said to be 'undefined', causing assignments like the above to abort.

But recall $1 \ominus 2$ from Section 6.6. It is defined, and is even a natural number; but we do not know which one. Similarly, we say that $1/0$ is a rational number, but we do not know which one. The assignment $x := 1/0$ does terminate, but we simply do not know what value of x results.

Our novel view simplifies program development considerably, but of course complicates programming language implementations. We insist that divisions a/b return a result in all circumstances; they cannot abort when $b = 0$. Thus in programming languages without that property (that is, in most programming languages, regrettably), the command $x := a/b$ cannot be code on its own. Instead, we would have to allow certain specifications as code; in this case, we would allow only

$$\begin{array}{rl} & x\colon [B \neq 0\ ,\ x = A/B] \\ \text{or the equivalent} & \{B \neq 0\}\ x := A/B \end{array} \qquad (6.2)$$

for any variable x and expressions A and B over suitable types. Other partial operators would be handled similarly. Adherence to the strict form could easily be enforced by a compiler, a major part of which is dedicated to syntax checking in any case. (See Exercises 9.15 and following for further discussion of undefinedness.)

6.8 Exercises

Ex. 6.1 Consider this factorial program, in which we assume $f, n : \mathbb{N}$. What laws are used for the first refinement step (shown)?

$$f\colon [f = n!]$$

© Carroll Morgan 1990, 1994, 1998

$$\begin{array}{ll} \sqsubseteq & \textbf{var } i : \mathbb{N} \bullet \\ & f, i\colon [i \le n \wedge f = i!] \, ; \qquad\qquad \text{(i)} \\ & f, i\colon [i \le n \wedge f = i! \ , \ f = i! \wedge i = n] \ . \qquad \text{(ii)} \end{array}$$

Complete the refinement to code.

Ex. 6.2 Repeat Exercise 4.2, but this time assume that $\{x \ge 0\}\ y := \sqrt{x}$ is code, while $y := \sqrt{x}$ (on its own) is not. (Recall Section 6.7.)

Ex. 6.3 ♡ Assuming the context $z : \mathbb{Z}$, show the following to be a valid refinement:

$$\begin{array}{ll} & z\colon [z \ge 0] \\ \sqsubseteq & \textbf{var } n : \mathbb{N} \bullet \\ & z := n \ . \end{array}$$

Ex. 6.4 Show that this refinement is valid:

$$\begin{array}{ll} & w\colon [\mathit{post}] \\ \sqsubseteq & \textbf{and } \mathit{post} \bullet \\ & \textbf{choose } w \ . \end{array}$$

Why does that mean that invariants cannot be code? *Hint*: Use the context when checking the proviso of *strengthen postcondition* 1.1.

Ex. 6.5 ♡ Some programming languages allow declarations of *constants* as follows:

$$\textbf{const } c = 3.$$

How can that effect be achieved with the declarations of this chapter? What are the remaining differences?

Ex. 6.6 ♡ Suppose we have context $n : \mathbb{N}$. Is the following a valid refinement?

$$n\colon \left[n^2 = 1\right] \quad \sqsubseteq \quad n := -1.$$

Ex. 6.7 ♡ Which of the following specifications are feasible, assuming the declarations $n : \mathbb{N};\ \ z : \mathbb{Z};\ \ r : \mathbb{R};\ \ c : \mathbb{C}$?

1. $n\colon [n = z]$
2. $z\colon [z = n]$
3. $r\colon [z \ge 0 \ , \ r^2 = z]$
4. $n\colon [z \ge 0 \ , \ n^2 = z]$
5. $r\colon [c^n = 1 \ , \ r = c + 1/c]$

© Carroll Morgan 1990, 1994, 1998

Ex. 6.8 Assume that the invariant is $x = 0$. Which of these specifications are feasible?

1. x: $[x = 0]$
2. x: $[x = 1]$
3. x: $[x = 0\ ,\ x = 0]$
4. x: $[x = 1\ ,\ x = 1]$
5. **choose** x

Ex. 6.9 Suppose we have *type coercion* functions for taking types into their subsets. For example, the function nat takes any natural number to itself, and any other number to *some* natural number. It satisfies these two properties:

1. $n \in \mathbb{N} \Rrightarrow n = \mathsf{nat}\, n$
2. $\mathsf{nat}\, c \in \mathbb{N}$, for any c.

The other coercion functions are int, rat, real, and cpx.

Assuming the types $n : \mathbb{N}$; $z : \mathbb{Z}$; $q : \mathbb{Q}$; $r : \mathbb{R}$; $c : \mathbb{C}$, determine whether the following are valid refinements:

1. n: $[n = z] \quad \sqsubseteq \quad n := \mathsf{nat}\, z$
2. n: $[z \geq 0\ ,\ n = z] \quad \sqsubseteq \quad n := \mathsf{nat}\, z$
3. n: $[r \geq 0\ ,\ n = r] \quad \sqsubseteq \quad n := \mathsf{nat}\, r$
4. z: $[n < 2\ ,\ z^2 = n] \quad \sqsubseteq \quad z := \mathsf{int}\, \sqrt{n}$
5. q: $[r \geq 0\ ,\ q^2 = r] \quad \sqsubseteq \quad q := \mathsf{rat}\, \sqrt{r}$
6. q: $[r \neq 0\ ,\ q = 2/r] \quad \sqsubseteq \quad q := \mathsf{rat}(2/r)$
7. r: $[c^n = 1\ ,\ r = c + 1/c] \quad \sqsubseteq \quad r := \mathsf{real}(c + 1/c)$.

Ex. 6.10 ♡ What is wrong with the following 'law' for iterations?

$$
\begin{array}{ll}
 & w\colon [I\ ,\ I \wedge \neg G] \\
\sqsubseteq? & \mathbf{con}\ e \bullet \\
 & \mathbf{do}\ G \rightarrow \\
 & \qquad w\colon [G \wedge (e = E) \wedge I\ ,\ I \wedge (0 \leq E < e)] \\
 & \mathbf{od}\ .
\end{array}
$$

Expression E is the variant, captured before each iteration by the logical constant e.

Hint: Unfold the iteration.

© Carroll Morgan 1990, 1994, 1998