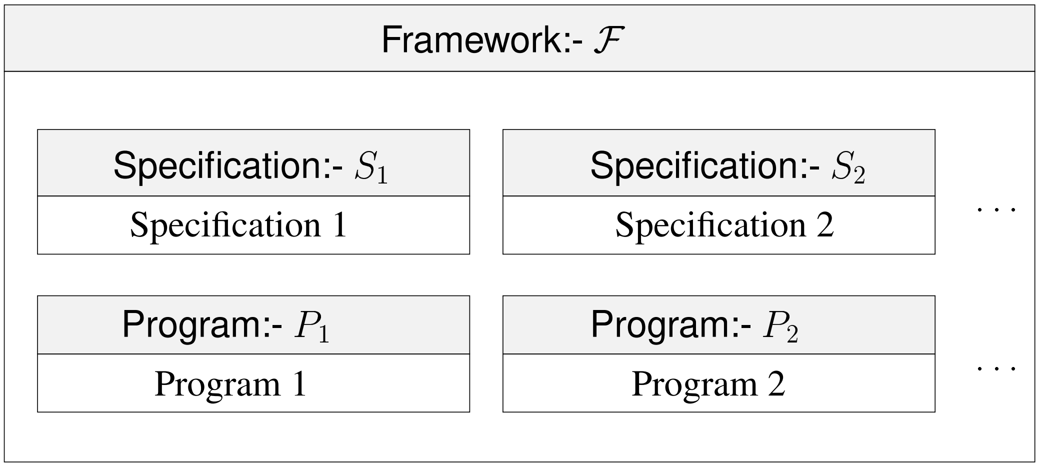
2. Our Definition of Correct Schemas

Our Approach to (Logic) Program Development

A three-tier formalism (with model-theoretic semantics):



- **Programs**: for computing (specified) relations.

  Programs are pure (standard or constraint) logic programs.

- **Specifications**: for defining new relations (and functions).

- **Specification Framework**: first-order axiomatisation of (all the relevant knowledge of) the problem domain,

  i.e. a first-order theory with *isoinitial* models.

  The framework provides an unambiguous semantic underpinning for *specifications* and *programs*,

  as well as the **correctness relationship** between them.

Parametric Frameworks

Framework:- $\mathcal{F} : \Delta \Leftarrow \Pi$		
Specification:- S_1 Specification 1	Specification:- S_2 Specification 2	$\ldots$
Program:- $P_1 : \delta_1 \Leftarrow \pi_1$ Program 1	Program:- $P_2 : \delta_2 \Leftarrow \pi_2$ Program 2	$\ldots$

- Δ are defined/axiomatised symbols
  Π are **framework parameters**
- Specs S_1, S_2, $\ldots$, may contain parameters from Π
- $\delta_1, \delta_2, \ldots$ are defined symbols
  $\pi_1, \pi_2, \ldots, \pi_i \subseteq \Pi$, are **program parameters** of (open) programs P_1, P_2, $\ldots$

Parametric Frameworks — An Example

Specification Framework $\mathcal{LIST}(Elem, \lhd)$;

$$
\begin{array}{rl}
\text{IMPORT:} & \mathcal{NAT}; \\
\text{SORTS:} & Nat, Elem, List; \\
\text{FUNCTIONS:} & nil \; : \; \to List; \\
 & \cdot \; : \; (Elem, List) \to List; \\
 & nocc \; : \; (Elem, List) \to Nat; \\
\text{RELATIONS:} & elemi \; : \; (List, Nat, Elem); \\
 & \lhd \; : \; (Elem, Elem); \\
\text{AXIOMS:} & \text{C-AXS}(nil, \cdot); \\
 & elemi(L, i, a) \leftrightarrow \exists h, T, j \,.\, L = h \cdot T \wedge \\
 & (i = 0 \wedge a = h \vee i = s(j) \wedge elemi(T, j, a)); \\
 & nocc(x, nil) = 0; \\
 & a = b \to nocc(a, b \cdot L) = nocc(a, L) + 1; \\
 & \neg a = b \to nocc(a, b \cdot L) = nocc(a, L); \\
\text{CONSTRAINTS:} & \text{(total ordering axioms for } \lhd) \\
 & x \lhd y \wedge y \lhd x \leftrightarrow x = y; \\
 & x \lhd y \wedge y \lhd z \to x \lhd z; \\
 & x \lhd y \vee y \lhd x.
\end{array}
$$

C-AXS$(nil, \cdot)$ contains:

- Clark's Equality Theory for the list constructors $\cdot$ and nil
- the first-order induction schema $H(nil) \wedge (\forall a, J \,.\, H(J) \to H(a \cdot J)) \to \forall L \,.\, H(L)$

$nocc(a, L)$ gives the number of occurrences of a in L

$elemi(L, i, a)$ means a occurs at position i in L.

Specification of Programs

In a specification framework $\mathcal{F}(\Pi)$,

a (**program**) **specification** S_δ is

a set of sentences that define new function or relation symbols δ

in terms of the symbols Σ of $\mathcal{F}$.

If S_δ contains symbols of Π, then it is an **open** specification.

Example

In $\mathcal{LIST}(Elem, \lhd)$, we can have the following specs:

SPECIFICATIONS:

$$
\begin{array}{l}
mem(e, L) \leftrightarrow \exists i \, . \, elemi(L, i, e); \\
n = l(L) \leftrightarrow \forall i \, . \, i < n \leftrightarrow \exists a \, . \, elemi(L, i, a); \\
append(A, B, L) \leftrightarrow \forall i, a \, . \\
\qquad\qquad\qquad (elemi(A, i, a) \leftrightarrow elemi(L, i, a) \wedge i < l(A)) \wedge \\
\qquad\qquad\qquad (elemi(B, i, a) \leftrightarrow elemi(L, i + l(A), a)); \\
perm(A, B) \leftrightarrow \forall e \, . \, nocc(e, A) = nocc(e, B); \\
C = A|B \leftrightarrow append(A, B, C); \\
ord(L) \leftrightarrow \forall i \, . \, elemi(L, i, e_1) \wedge elemi(L, s(i), e_2) \rightarrow e_1 \lhd e_2 \\
sort(L, S) \leftrightarrow perm(L, S) \wedge ord(S).
\end{array}
$$

Program Schemas

Framework:- $\mathcal{F} : \Delta \Leftarrow \Pi$

Specification:- S_r

Specification of r

Specification:- S_π

Specification of π

Program:- $P_r : r \Leftarrow \pi$

Program to compute r

$$\begin{array}{c}\text{Schema}\\ \text{(for relation } r\text{)}\end{array} = \left\{\begin{array}{c}\text{Framework}\\ +\\ \text{Open program, } \textbf{template} \text{ (for } r\text{)}\\ +\\ \text{Specifications of template predicates}\end{array}\right.$$

A Simple Divide-and-Conquer Schema

Schema $\mathcal{DC}(\mathsf{I}, \mathsf{O}, I_r, O_r, O_d, \prec)$;

IMPORT: $\mathcal{LIST}(\mathsf{I}), \mathcal{LIST}(\mathsf{O})$;

DECLARATIONS:

$$\begin{array}{lcl} I_r & : & [\mathsf{I}]; \\ O_r & : & [\mathsf{I}, \mathsf{O}]; \\ O_d & : & [\mathsf{I}, List(\mathsf{I}), List(\mathsf{I})]; \\ \prec & : & [\mathsf{I}, \mathsf{I}]; \end{array}$$

AXIOMS:

$$\begin{array}{lcl} \mathrm{A1} & : & M(a,b) \leftrightarrow l(a) = l(b) \wedge \\ & & \forall x, y, i \,.\, elemi(a,i,x) \wedge elemi(b,i,y) \rightarrow O_r(x,y); \end{array}$$

CONSTRAINTS:

$$\begin{array}{lcl} \mathrm{C1} & : & I_r(x) \rightarrow \exists h, a \,.\, O_d(x,h,a); \\ \mathrm{C2} & : & I_r(x) \wedge O_d(x,h,a) \rightarrow (\forall y \,.\, mem(y,a) \rightarrow I_r(y) \wedge y \prec x); \\ \mathrm{C3} & : & WellFounded(\prec); \\ \mathrm{C4} & : & I_r(x) \wedge O_d(x,h,a) \wedge O_r(x,y) \rightarrow \exists b \,.\, M(a,b); \end{array}$$

SPECIFICATIONS:

$$\begin{array}{rcl} r & : & [\mathsf{I}, \mathsf{O}]; \\ \mathrm{S}_r^c & : & I_r(x) \rightarrow (r(x,y) \leftrightarrow O_r(x,y)); \\ d & : & [\mathsf{I}, List(\mathsf{I}), List(\mathsf{I})]; \\ \mathrm{S}_d^{sl} & : & I_r(x) \rightarrow (d(x,h,a) \rightarrow O_d(x,h,a)); \\ & & I_r(x) \rightarrow \exists h, a \,.\, d(x,h,a); \\ rec & : & [List(\mathsf{I}), List(\mathsf{O})]; \\ \mathrm{S}_{rec}^c & : & (\forall x \,.\, mem(x,a) \rightarrow I_r(x)) \rightarrow (rec(a,b) \leftrightarrow M(a,b)); \\ c & : & [List(\mathsf{I}), List(\mathsf{O}), \mathsf{O}]; \\ \mathrm{S}_c^{gc} & : & I_r(x) \wedge O_d(x,h,a) \wedge M(a,b) \rightarrow (c(h,b,y) \leftrightarrow O_r(x,y)); \end{array}$$

TEMPLATE:

$$r(x,y) \ \leftarrow \ d(x,h,a), rec(a,b), c(h,b,y).$$

Using (Generic) Schemas

A (generic) schema can be **specialised** into different (specific) schemas.

E.g. the schema $\mathcal{DC}$ can be specialised into divide-and-conquer schemas with *arbitrary* numbers of base cases, step cases and recursive calls in the templates.

Example 1

An instance of $\mathcal{DC}$, with 1 base case and 1 step case in the template (we show only the template):

$$
\begin{array}{lll}
r(x,y) & \leftarrow & prim(x), solve(x,y) \\
r(x,y) & \leftarrow & \neg prim(x), dec(x,h,x1,x2), r(x1,y1), r(x2,y2), \\
 & & comp(h,y1,y2,y)
\end{array}
$$

Note that in the step case, there are 2 recursive calls (to r).

Example 2

An instance of $\mathcal{DC}$ with 1 base case and 2 step cases in the template:

$$
\begin{array}{lll}
r(0,y,z) & \leftarrow & c1(y,z) \\
r(s(x),y,z) & \leftarrow & sum(v,v,s(x)), r(v,y,w), c2(0,y,w,z) \\
r(s(x),y,z) & \leftarrow & sum(v,v,x), r(v,y,w), c2(s(0),y,w,z).
\end{array}
$$

Schema Correctness

Framework:- $\mathcal{F} : \Delta \Leftarrow \Pi$

Specification:- S_r	Specification:- S_π
Specification of r	Specification of π

Program:- $P_r : r \Leftarrow \pi$
Program to compute r

$$\begin{array}{c}\textbf{Correct Schema} \\ \text{(for } r\text{)}\end{array} = \left\{ \begin{array}{c} \text{Adequate (Open) Framework} \\ + \\ \textbf{Steadfast} \text{ template for } r \\ + \\ \text{Specs of template predicates} \end{array} \right.$$

$$\textbf{Steadfastness} = \left\{ \begin{array}{c} \text{Correctness of } \textit{open programs} \\ \text{in} \\ \text{classes of interpretations of the } \textit{specs} \\ \text{in} \\ \text{the context of the } \textit{framework} \end{array} \right.$$

Steadfastness $\Rightarrow$ **Correct Reusability**

Adequate Frameworks

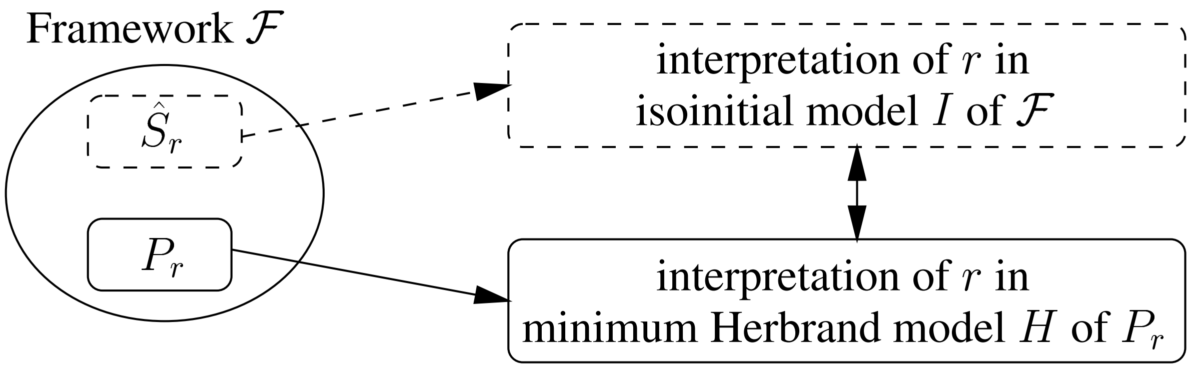
- A *closed* framework $\mathcal{F}$ is *adequate* ...
  ... if it has a reachable isoinitial model.

- An *open* framework $\mathcal{F}(\Pi)$ is *adequate* ...
  ... if, for every adequate closed framework $\mathcal{G}$,
  the *instance* $\mathcal{F}(\Pi)[\mathcal{G}]$ is an adequate closed framework.

  We use only adequate frameworks ...
  ... constructed incrementally

Specification and Correctness of Closed Programs

Strict Specifications



Example

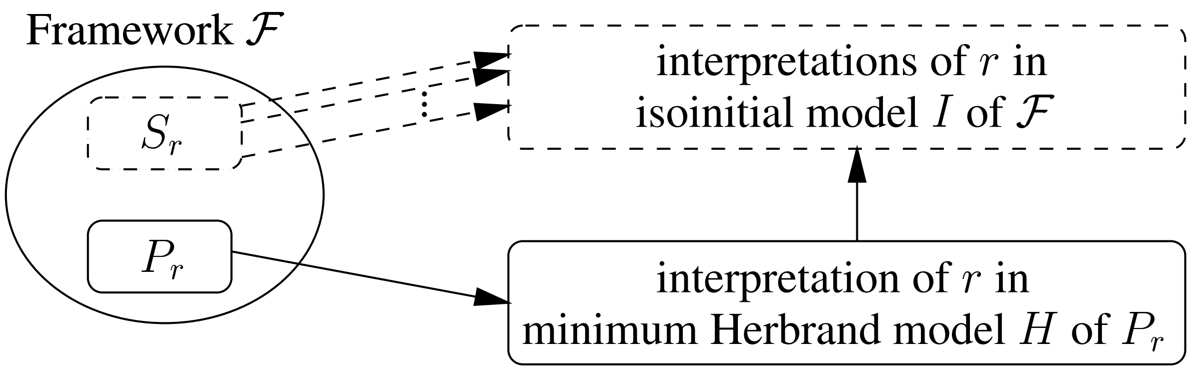
An **if-and-only-if specification** $\hat{S}_r$ of a new *relation* r in a framework $\mathcal{F}$:

$$r(x) \leftrightarrow R(x)$$

where $R(x)$ is any formula of the language of $\mathcal{F}$.

Specification and Correctness of Closed Programs

Non-Strict Specifications



Example

A **super-and-sub specification**:

$$\forall x \,.\, (R_{sub}(x) \rightarrow r(x)) \wedge (r(x) \rightarrow R_{super}(x))$$

where $R_{sub}(x)$ and $R_{super}(x)$ are two formulas of the language of $\mathcal{F}$ such that $\mathcal{F} \vdash \forall x \,.\, R_{sub}(x) \rightarrow R_{super}(x)$.

Specification and Correctness of Closed Programs

Conditional Specifications

A **conditional specification** of a new relation r in a framework $\mathcal{F}$:

$$\forall x, y \,.\, IC(x) \rightarrow (r(x,y) \leftrightarrow OC(y) \wedge R(x,y))$$

where

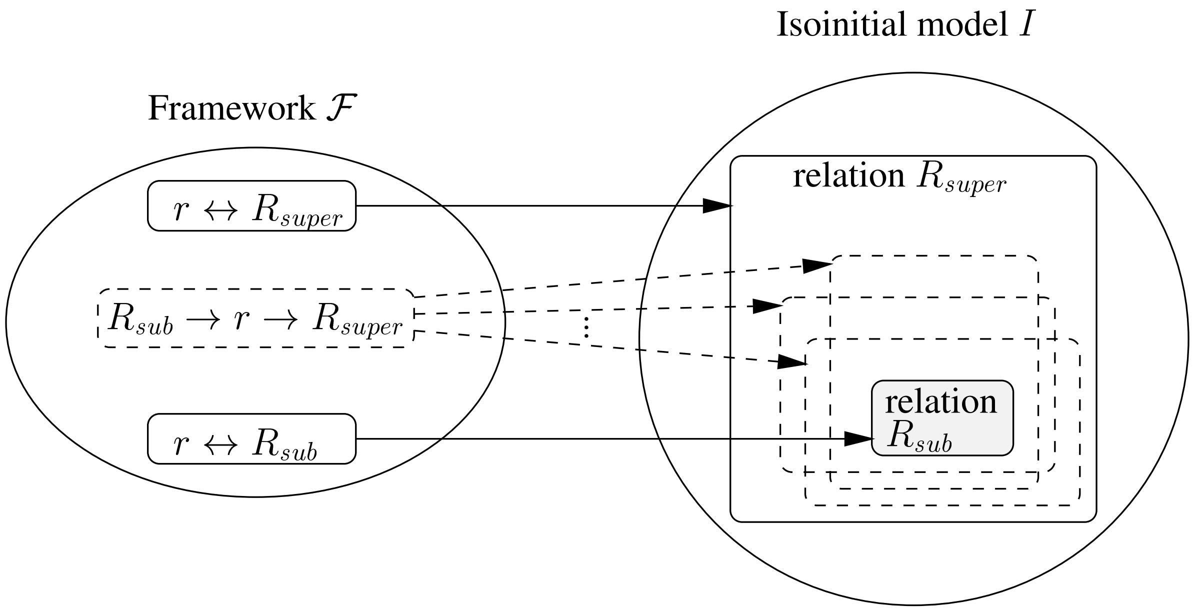
$IC(x)$ is a condition on x,

$OC(y)$ is a condition on y, and

$R(x,y)$ is a formula of the language of $\mathcal{F}$.

A conditional spec is like a **pre-post-condition** style spec as in VDM, Z, and B.

A conditional spec is a special case of a super-and-sub spec:



Specification and Correctness of Open Programs

Let $\mathcal{F}(\Pi)$ be a framework with a signature Σ.

An **open program** $P_r(\pi)$ in $\mathcal{F}(\Pi)$ is a program with

- *defined* predicates r (specified by S_r)
- *open* predicates (or *parameters*) π (specified by S_π)

The **specification** of an **open program** $P_r(\pi)$ in a framework $\mathcal{F}(\Pi)$ is a pair (S_r, S_π)

where S_r is a specification of the relation r

S_π is a specification of P_r's parameter π ($\pi \subseteq \Pi$)

Correctness can be defined in a similar manner to closed programs, but …

… we cannot use minimum Herbrand models (closed world assumption)

… so we use **minimum j-models** where

j is a *pre-interpretation*, i.e. a Π-interpretation

… and we base correctness on **steadfastness**

Open Programs – An Example

In the open framework $\mathcal{LIST}(Int, \lhd)$,

(as well as any instance, e.g. the closed framework $\mathcal{LIST}(Int, \leq))$

we can have the following open program P_{sort}:

$$\begin{array}{rcl} sort(nil, nil) & \leftarrow & \\ sort(x.nil, x.nil) & \leftarrow & \\ sort(x.y.A)), W) & \leftarrow & split(x.y.A)), I, J) \wedge \\ & & sort(I, U) \wedge sort(J, V) \wedge \\ & & merge(U, V, W) \end{array} \qquad (P_{sort})$$

with

– the following (if-and-only-if) spec of $sort$:

$$sort(X, Y) \leftrightarrow perm(X, Y) \wedge ord(Y) \qquad (S_{sort})$$

– the following (*selection*) spec of $split$:

$$\begin{array}{l} l(X) > 1 \wedge split(X, Y, Z) \rightarrow \\ perm(X, Y|Z) \wedge l(Y) < l(X) \wedge l(Z) < l(X) \\ l(X) > 1 \rightarrow \exists Y, Z \,.\, split(X, Y, Z) \end{array} \qquad (S_{split})$$

– the following (conditional) spec of $merge$:

$$\begin{array}{ll} ord(X) \wedge ord(Y) \rightarrow & \\ (merge(X, Y, Z) \leftrightarrow & ord(Z) \wedge \\ & perm(X|Y, Z)) \end{array} \qquad (S_{merge})$$

Steadfastness: Correctness of Open Programs

- In a *closed* framework $\mathcal{F}$,
  let P_r be an open program for r,
  with parameters $\pi_1, \ldots, \pi_n$, and specifications $S_r, S_1, \ldots, S_n$,
  P_r is **steadfast** in $\mathcal{F}$
  if, for any closed programs $P_1, \ldots, P_n$ that compute $\pi_1, \ldots, \pi_n$ such that P_i is correct wrt S_i,
  the (closed) program $P_r \cup P_1 \cup \ldots \cup P_n$ is correct wrt S_r.

- In an *open* framework $\mathcal{F}(\Pi)$, P_r is **steadfast**
  if it is steadfast in every instance $\mathcal{F}[\mathcal{G}]$.

Example

In the open framework $\mathcal{LIST}(Int, \triangleleft))$
the open program P_{sort} is steadfast:

in every instance of $\mathcal{LIST}(Int, \triangleleft)$,
(e.g. the closed framework $\mathcal{LIST}(Int, \leq))$
P_{sort} is always correct wrt its spec S_{sort},
provided that the programs for its parameters $merge$ and $split$ are correct wrt to their respective specs S_{split} and S_{merge}.
If we have steadfast programs for $merge$ and $split$,
then their composition with P_{sort} will also be steadfast.

Steadfastness — A Simple Example

Consider the specification framework, $\mathcal{ITER}(D, \circ)$, containing (at least) the following:

(i) a (generic) domain D, with a binary operation $\circ$ and a distinguished element e;

(ii) the usual structure of natural numbers, to express the number n;

(iii) the iteration operation $\times(a, n) = e \circ a \underbrace{\circ \cdots \circ}_{(n \text{ times})} a$.

Suppose we want a program to compute a relation $iterate$, that iterates n times the binary operation $\circ$ on D.

The spec of $iterate$ is:

$$S_{iterate}(D, \times): \qquad iterate(x, n, z) \leftrightarrow z = \times(a, n) \tag{5}$$

Consider the following program $P_{iterate}$:

$$\begin{array}{lcl} iterate(a, 0, v) & \leftarrow & unit(v) \\ iterate(a, s(n), v) & \leftarrow & iterate(a, n, w), op(w, a, v) \end{array}$$

where s is the successor function for natural numbers,

and the predicates $unit$ and op are specified in $\mathcal{ITER}$ by:

$$\begin{array}{llcl} S_{unit}(D, e) : & unit(u) & \leftrightarrow & u = e \\ S_{op}(D, \circ) : & op(x, y, z) & \leftrightarrow & z = x \circ y \end{array} \tag{6}$$

$P_{iterate}$ is *correct* with respect to the specs (5) and (6)

because it always computes $iterate$ correctly

in terms of correct computations of $unit$ and op in *any* interpretation of $\mathcal{ITER}$.

Steadfastness – A Simple Example (contd)

- For example, if

  D is the set of natural numbers, $\circ$ is $+$, e is 0,

  then $\times(a, n) = 0 + a + \cdots + a = na$.

  Now, if $Q^{+}_{unit(u)}$ computes $u = 0$ and $Q^{+}_{op(x,y,z)}$ computes $z = x + y$,

  then $P_{iterate} \cup Q^{+}_{unit(u)} \cup Q^{+}_{op(x,y,z)}$ will compute na,

  i.e., $P_{iterate}$ is correct in this first interpretation.

- As another example, if

  D is the set of integers, $\circ$ is $-$, e is 0,

  then $\times(a, n) = 0 - a - \cdots - a = -na$.

  If $Q^{-}_{unit(u)}$ computes $u = 0$ and $Q^{-}_{op(x,y,z)}$ computes $z = x - y$, then $P_{iterate} \cup Q^{-}_{unit(u)} \cup Q^{-}_{op(x,y,z)}$ will compute $-na$ for an integer a,

  i.e., $P_{iterate}$ is also correct in this second interpretation.

  We say $P_{iterate}$ is **steadfast** in $\mathcal{ITER}$.

Steadfastness – A Simple Example (contd)

As an example of a non-steadfast program in $\mathcal{ITER}$, consider the following program $P^*_{iterate}$:

$$
\begin{array}{lcl}
iterate(a,0,v) & \leftarrow & unit(v) \\
iterate(a,n,v) & \leftarrow & m+m=n, iterate(a,m,w), op(w,w,v) \\
iterate(a,n,v) & \leftarrow & m+s(m)=n, iterate(a,m,w), \\
 & & op(w,w,z), op(z,a,v)
\end{array}
$$

$P^*_{iterate}$ is more efficient than $P_{iterate}$: the number of recursive calls is linear in $P_{iterate}$, whereas it is logarithmic in $P^*_{iterate}$.

$P^*_{iterate}$ is correct with respect to (5) and (6) if D is the set of natural numbers with $+$ as $\circ$ and 0 as e.

However, it is incorrect if D is the set of integers with $-$ as $\circ$ and 0 as e.

For instance, for $iterate(a, s(s(s(s(0)))), v)$, $P^*_{iterate}$ computes 0 instead of the correct answer $-4a$.

Thus $P^*_{iterate}$ is *not* steadfast in $\mathcal{ITER}$.

Steadfastness – A Simple Example (contd)

However, if we require that e and $\circ$ satisfy the additional (associativity) axioms

$\forall x \,.\, e \circ x = x$ and

$\forall x, y, z \,.\, x \circ (y \circ z) = (x \circ y) \circ z$,

then we can prove that:

$$\begin{cases} \times(a,n) &=& \times(a, n \div 2) \circ \times(a, n \div 2) \circ a & \text{if } n \text{ is odd} \\ \times(a,n) &=& \times(a, n \div 2) \circ \times(a, n \div 2) & \text{if } n \text{ is even} \end{cases}$$

and in the subclass $\mathcal{ITER}^*$ of interpretations of $\mathcal{ITER}$ that satisfy the additional axioms,

$P^*_{iterate}$ computes correctly, and thus it is steadfast in $\mathcal{ITER}^*$.

For instance,

if D is the set of m-dimensional square matrices,

with the m-dimensional identity matrix as e,

then since matrix multiplication $\times$ is associative,

$P^*_{iterate}$ is correct, where op computes matrix products.

Steadfastness $\Rightarrow$ Correct Reusability

Steadfastness yields 2 levels of **correct reusability**:

Inheritance

If $P_r(\pi)$ is correct in an open framework $\mathcal{F}(\Pi)$ wrt a specification (S_r, S_π),

then it is correct wrt (S_r, S_π), in all the instances $\mathcal{F}[\mathcal{G}]$.

$\Rightarrow$ reusability of correct programs

via framework *composition*, and *morphisms* (subclassing, instantiation).

Compositionality

If $P_r(p)$ is correct in a framework $\mathcal{F}$ wrt a specification (S_r, S_p),

and Q is a program for computing p that is correct wrt S_p,

then $P \cup Q$ is a program that is correct (in $\mathcal{F}$) wrt (S_r).

$\Rightarrow$ guarantees the correctness of the composition of inherited correct open programs