

Legend Expressions: d, e : arbitrary; f, g : function; p, q : boolean; X, Y : set; P : predicate; F : family of functions; S : family of sets. Variables: new (not free in these expressions) x, y ; unrestricted: z .

The *only* concept that is not familiar from earlier courses is *function abstraction*. Funmath syntax: $z : X \Delta p . e$, with z not free in X (a style decision); Δp is optional, i.e., $(z : X . e) = z : X \Delta 1 . e$. Generalization to tuples is evident. Variants: $e \mid z : X \Delta p$ stands for $z : X \Delta p . e$, and $z : X \mid p$ stands for $z : X \Delta p . z$. Axioms: domain $x \in \mathcal{D}(z : X \Delta p . e) \equiv x \in X \wedge p|_x^z$ and mapping $x \in \mathcal{D}(z : X \Delta p . e) \Rightarrow (z : X \Delta p . e) x = e|_x^z$.

Note that the operators are not part of the Funmath language, but constitute the basic ‘user library’.

Operators for defining constant functions (basic form, empty function, 1-point function definer)	
$\bullet, \varepsilon, \mapsto$	$X \bullet e = x : X . e$ (recall: new x), $\varepsilon = \emptyset \bullet e$ and $d \mapsto e = \iota d \bullet e$ (recall: $y \in \iota x \equiv y = x$)
Quantifying operators: <i>everywhere</i> (\forall) and <i>somewhere</i> (\exists) as predicates over predicates	
\forall, \exists	$\forall P \equiv P = \mathcal{D} P \bullet 1$ and $\exists P \equiv P \neq \mathcal{D} P \bullet 0$
Operators from functions to sets (domain, range, graph, bijective domain and bijective range)	
\mathcal{D}	Defined axiomatically for every function (in abstractions and in the axiom for \rightarrow)
\mathcal{R}	$y \in \mathcal{R} f \equiv \exists x : \mathcal{D} f . f x = y$ If f is an abstraction, we usually write $\mathcal{R} f$ as $\{f\}$.
\mathcal{G}	$\mathcal{G} f = \{x, f x \mid x : \mathcal{D} f\}$ This is already an illustration of the preceding convention.
Bdom	$\text{Bdom } f = \{x : \mathcal{D} f \mid \forall y : \mathcal{D} f . f x = f y \Rightarrow x = y\}$
Bran	$\text{Bran } f = \{f x \mid x : \text{Bdom } f\}$
Function typing (family type, function arrow, FunCart product, dependent type)	
fam	$f \in \text{fam } Y \equiv \forall x : \mathcal{D} f . f x \in Y$
\rightarrow	$f \in X \rightarrow Y \equiv \mathcal{D} f = X \wedge \forall x : \mathcal{D} f \cap X . f x \in Y$
\times	$f \in \times S \equiv \mathcal{D} f = \mathcal{D} S \wedge \forall x : \mathcal{D} f \cap \mathcal{D} S . f x \in S x$. Convenient shorthand: for an abstraction $z : X . Y$ with $z \notin \varphi X$, we write $\times z : X . Y$ as $X \exists z \rightarrow Y$.
Predicates with 1 function argument (constant, injective, surjective)	
con	$\text{con } f \equiv \forall (x, y) : (\mathcal{D} f)^2 . f x = f y$
inj	$\text{inj } f \equiv \forall (x, y) : (\mathcal{D} f)^2 . f x = f y \Rightarrow x = y$
srj	$f \text{ srj } Y \equiv Y = \mathcal{R} f$
Function transformers with 1 function argument (restrict, filter, inverse, transpose)	
\upharpoonright	$f \upharpoonright X = x : \mathcal{D} f \cap X . f x$ (note that $f \upharpoonright X \in \mathcal{D} f \cap X \rightarrow \mathcal{R} f$)
\downarrow	$f \downarrow P = f \upharpoonright (\mathcal{D} f \downarrow P)$, defining \downarrow for sets by $X \downarrow P = \{x : X \cap \mathcal{D} P \mid P x\}$.
\dashv	$f \dashv \in \text{Bran } f \rightarrow \text{Bdom } f$ and $\forall x : \text{Bdom } f . f \dashv (f x) = x$
\dashv^T	$F^T = y : \cap (x : \mathcal{D} F . \mathcal{D}(F x)) . x : \mathcal{D} F . F x y$ (exercise: what is the type?)
Predicates (relations) with 2 function arguments (compatibility, subfunction)	
\odot	$f \odot g \equiv f \upharpoonright \mathcal{D} g = g \upharpoonright \mathcal{D} f$, hence $f \odot g \equiv \forall x : \mathcal{D} f \cap \mathcal{D} g . f x = g x$
\subseteq	$f \subseteq g \equiv f = g \upharpoonright \mathcal{D} f$, hence $f \subseteq g \equiv \mathcal{D} f \subseteq \mathcal{D} g \wedge f \odot g$
Function transformers with 2 function arguments. (override, merge, compose, parallel)	
\otimes	$f \otimes g = x : \mathcal{D} f \cup \mathcal{D} g . (x \in \mathcal{D} g) ? g x \upharpoonright f x$
\cup	$f \cup g = x : \mathcal{D} f \cup \mathcal{D} g \wedge (x \in \mathcal{D} f \cap \mathcal{D} g \Rightarrow f x = g x) . f \otimes g$
\circ	$f \circ g = x : \mathcal{D} g \wedge g x \in \mathcal{D} f . f (g x)$
\parallel	$f \parallel g = (x, y) : \mathcal{D} f \times \mathcal{D} g . f x, g y$
Direct extension operators (for any operator g and any infix operator \star)	
$\overline{}$	$\overline{g} f = g \circ f \quad (= x : \mathcal{D} g \wedge g x \in \mathcal{D} f . f (g x))$
$\widehat{}$	$f \widehat{\star} g = (\star) \circ (f, g)^T \quad (= x : \mathcal{D} f \cap \mathcal{D} g \wedge (f x, g x) \in \mathcal{D}(\star) . f x \star g x)$
Elastic generalizations (direct extension, parallel, compatibility, merge) and type merge	
\leq	$\widehat{\leq} F = g \circ F^T$
\parallel	$\parallel F f = x : \mathcal{D} F \cap \mathcal{D} f \wedge f x \in \mathcal{D}(F x) . F x (f x)$
\odot	$\odot F \equiv \forall (x, y) : (\mathcal{D} F)^2 . F x \odot F y$
\cup	$\mathcal{D}(\cup F) = \{y : \cup(\mathcal{D} \circ F) \mid \forall (x, x') : (\mathcal{D} F)^2 . y \in \mathcal{D}(F x) \cap \mathcal{D}(F x') \Rightarrow F x y = F x' y\}$ $y \in \mathcal{D}(\cup F) \Rightarrow \forall x : \mathcal{D} F . y \in \mathcal{D}(F x) \Rightarrow \cup F y = F x y$
\otimes	$\otimes G = \{\cup F \mid F : \times G \wedge \odot F\}$ for any family G of function types

Funmath LRRL (Language Rationale and Reference Leaflet)

R. Boute — INTEC, Universiteit Gent

Version 2.5

Rationale A formal mathematical language is valuable insofar as it supports the design of precise calculation rules that are convenient in everyday practice.

In this sense, common mathematical conventions are strong in Algebra and Analysis (e.g., rules for \int in every introductory Analysis text), weaker in Discrete Mathematics (e.g., rules for \sum only in very few texts), and poor in Predicate Logic (e.g., disparate conventions for \forall and \exists , rules in most logic texts not suited for practice). This is reflected in the degree to which everyday calculation in the respective areas can be called “formal”, and inversely proportional to the needs in Computing Science.

Entirely deficient are the conventions for set comprehension. Common expressions such as $\{m \in \mathbb{N} \mid m < n\}$ and $\{2 \cdot n \mid n \in \mathbb{Z}\}$ may look innocuous, but exposing their structure as $\{v \in X \mid p\}$ and $\{e \mid v \in X\}$ (with the metavariables below) reveals the ambiguity: $\{n \in \mathbb{N} \mid n \in \mathbb{Z}\}$ matches both. Calculation rules are nonexistent.

Funmath (Functional Mathematics) is not “yet another computer language” but an approach to structure formalisms by conceiving mathematical objects as functions whenever convenient — which is quite more often than common practice reflects. Four constructs suffice for synthesizing most (all?) common conventions without their ambiguities and inconsistencies, and also yield new yet useful new forms of expression, such as point-free expressions. This leaflet summarizes only the syntax and main definitions; the calculation rules are treated extensively in the courses listed above (top of page).

Syntax To facilitate adoption of this design in other formalisms, we avoid a formal grammar. Instead, we use the following metavariables: i for a (tuple of) identifiers, and for expressions: v, w : (tuple of) variable(s); d, e : arbitrary; p, q, r : boolean; X, Y : set; f, g : function; P, Q : predicate; F, G : family of functions; S, T : family of sets. By “family of X ” we mean “ X -valued function”. Here are the four constructs.

0. An *identifier* can be any (string of) symbol(s) except markers (binding colon and filter mark, abstraction dot), parentheses (), and a few keywords (**def**, **spec**).

Identifiers are *introduced* by *bindings* $i : X \Delta p$, read “ i in X satisfying p ”. The *filter* Δp (or **with** p) is optional, e.g., $n : \mathbb{N}$ and $n : \mathbb{Z} \wedge n \geq 0$ are interchangeable.

Definitions, of the form **def binding**, introduce *constants*, with global scope.

Existence and uniqueness are proof obligations, which is not the case for *specifications*, of the form **spec binding**. Example: **def** *roto* : $\mathbb{R}_{\geq 0}$ **with** *roto*² = 2. Well-established symbols (e.g., $\mathbb{B}, \Rightarrow, \mathbb{R}, +, \sqrt{}$) are seen as predefined constants.

1. An *abstraction* of the form *binding . expression* denotes a *function*. The identifiers introduced are *variables*, with scope limited to the abstraction. Writing f for $v : X \Delta p . e$, the domain axiom is $d \in \mathcal{D} f \equiv d \in X \wedge p|_d^v$ and the mapping axiom $d \in \mathcal{D} f \Rightarrow f d = e|_d^v$. Here $e|_d^v$ is e with d substituted for v . Example: $n : \mathbb{Z} . 2 \cdot n$.
2. A *function application* has the form $f e$ in the default *prefix* syntax. For a function identifier (*operator*), other conventions can be specified by dashes in its binding, e.g., $\dashv \star \dashv$ for infix. Prefix has precedence over infix. Parentheses are used for overriding precedence rules, *never* as an operator. Application may be partial: if \star is an infix operator, then $(a \star)$ and $(\star b)$ satisfy $(a \star) b = a \star b = (\star b) a$. *Variadic application*, of the form $e \star e' \star e'' \star e'''$, is explained below.

3. *Tupling*, of the form e, e', e'' (any length n), denotes a function with domain $0..n - 1$ and mapping illustrated by $(e, e', e'')0 = e$ and $(e, e', e'')1 = e'$ etc. The conditional expression $(p?e' \dagger e)$ is defined via tuples by $(p?e' \dagger e) = (e, e')p$.

Macros can define shorthands or sugaring in terms of the basic syntax, but very few are needed. Shorthands are d^e for $d \uparrow e$ (exponent) and d_e for $d \downarrow e$ (filtering, see below). Sugaring macros are $e \mid v : X \Delta p$ for $v : X \Delta p . e$ and $v : X \mid p$ for $v : X \Delta p . v$, and finally $v := e$ for $v : \iota e$, using the *singleton set injector* ι with axiom $d \in \iota e \equiv d = e$.

Functions A function f is fully defined by its *domain* $\mathcal{D}f$ and its *mapping* (unique image for every domain element). Skipping a technicality (see course notes), equality is axiomatized by $f = g \Rightarrow \mathcal{D}f = \mathcal{D}g \wedge (e \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f e = g e)$ and its converse, the inference rule $\mathcal{D}f = \mathcal{D}g \wedge (v \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f v = g v) \vdash f = g$ (v not free in f, g).

Example: the *constant function definer* \bullet with $X \bullet e = v : X . e$ (v not free in e). It is near-trivial, but very useful. Special instances: the *empty function* $\varepsilon := \emptyset \bullet e$ (any e , by function equality) and the *one-point function definer* \mapsto with $d \mapsto e = \iota d \bullet e$.

Predicates are \mathbb{B} -valued functions. Here $\mathbb{B} = \{0, 1\}$, others may prefer $\mathbb{B} = \{\text{F}, \text{T}\}$.

Pragmatics We show how to exploit the functional mathematics principle and (re-)synthesise common notations, issues that are not evident from mere syntax.

(a) *Elastic operators* originally are functionals designed to obviate common ad hoc abstractors such as $\sum_{i=m}^n, \forall v : X, \lim_{x \rightarrow a}$, but the idea leads to other designs as well.

The *quantification operators* (\forall, \exists) are defined by $\forall P \equiv P = \mathcal{D}P \bullet 1$ and $\exists P \equiv P \neq \mathcal{D}P \bullet 0$. Observe synthesis of familiar expressions in $\forall P \equiv \forall x : \mathcal{D}P . P x$ and $\forall x : \mathbb{R} . x^2 \geq 0$ but also new forms as in $\forall (p, q) = p \wedge q$ and $\exists (p, q) = p \vee q$.

For every common infix operator \star an *elastic extension* E is designed such that $x \star y = E(x, y)$. Evident are \cup and \cap for \cup and \cap (e.g., $e \in \cap S \equiv \forall x : \mathcal{D}S . e \in S x$), more interesting are \sum for $+$ (see below) and the following extensions for $=$ and \neq .

The *constancy predicate* con with $\text{con } f \equiv \forall x : \mathcal{D}f . \forall y : \mathcal{D}f . f x = f y$ and the *injectivity predicate* inj with $\text{inj } f \equiv \forall x : \mathcal{D}f . \forall y : \mathcal{D}f . f x = f y \Rightarrow x = y$ follow the same design principle. Properties are $\text{con}(d, e) \equiv d = e$ and $\text{inj}(d, e) \equiv d \neq e$.

The (*function*) *range operator* \mathcal{R} has axiom $e \in \mathcal{R}f \equiv \exists x : \mathcal{D}f . f x = e$. Using $\{ _ \}$ as a synonym for \mathcal{R} synthesizes set notations such as $\{m : \mathbb{N} \mid m < n\}$ and $\{2 \cdot n \mid n : \mathbb{Z}\}$. Since we never abuse \in for binding, $\{n : \mathbb{N} \mid n \in \mathbb{Z}\}$ is unambiguous. Expressions like $\{e, e', e''\}$ also have their usual meaning. Rules are derived via \exists . We use \mathcal{R} in defining the *function arrow* \rightarrow by $f \in X \rightarrow Y \equiv \mathcal{D}f = X \wedge \mathcal{R}f \subseteq Y$.

Variadic function application is alternating an infix operator with arguments. We *uniformly* take this as standing for the application of a matching elastic operator to the argument list. Examples: $p \wedge q \wedge r \equiv \forall (p, q, r)$ and $e = e' = e'' \equiv \text{con}(e, e', e'')$. An example of a new opportunity is $e \neq e' \neq e'' \equiv \text{inj}(e, e', e'')$.

Traditional ad hoc abstractors have a “range” attached to them, as in $\sum_{i=m}^n$. Elastic operators subsume this by the domain of the argument. This *domain modulation* principle obtains additional flexibility from the generic *function/set filtering operator* \downarrow defined by $f_P = x : \mathcal{D}f \cap \mathcal{D}P \Delta P x . f x$ and $X_P = \{x : X \cap \mathcal{D}P \mid P x\}$.

(b) *Generic functionals* extend often-used functionals to *arbitrary* functions by lifting restrictions. For instance, function inversion f^- traditionally requires $\text{inj } f$ and composition $f \circ g$ traditionally requires $\mathcal{R}g \subseteq \mathcal{D}f$. We discard all restrictions on the argument functions by defining the domain of the result function such that its image definition is free of out-of-domain applications, e.g., $f \circ g = x : \mathcal{D}g \Delta g x \in \mathcal{D}f . f(g x)$. The main generic functionals and their definitions are listed at the end of this leaflet.

(c) *Two design examples* For function types, a useful refinement of \rightarrow is the Functional Cartesian Product \times with $\times T = \{f : \mathcal{D}T \rightarrow \bigcup T \mid \forall x : \mathcal{D}f \cap \mathcal{D}T . f x \in T x\}$. Some properties: (i) $\times^- X x = \{f x \mid f : X\}$ for nonempty $X : \mathcal{R} \times$ and $x : \mathcal{D}(\times^- X)$; (ii) $X \rightarrow Y = \times(X \bullet Y)$; (iii) $X \times Y = \times(X, Y)$ where $X \times Y$ is the usual Cartesian product defined by $x, y \in X \times Y \equiv x \in X \wedge y \in Y$. Hence we define variadic application of \times by $X \times Y \times Z = \times(X, Y, Z)$ etc. Often we write $X \ni v \rightarrow Y_v$ for $\times v : X . Y_v$. The course notes show many applications, from analog electronics to record types.

A generalized and formal definition of \sum is given via the generic *function merge* \cup defined by $f \cup g = x : \mathcal{D}f \cup \mathcal{D}g \Delta (x \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f x = g x) . x \in \mathcal{D}f ? f x \dagger g x$. We define \sum recursively by the *empty rule* $\sum \varepsilon = 0$, the *one-point rule* $\sum (e \mapsto c) = c$, and the *merge rule* $\sum (f \cup g) = \sum f + \sum g$ for any numeric c and any number-valued functions f and g with finite nonintersecting domains. The function domains may happen to be numeric. As expected, variadic application of $+$ is defined by $x + y + z = \sum(x, y, z)$. With $_ \dots _$ defined as in Pascal by $m .. n = \{i : \mathbb{Z} \mid m \leq i \leq n\}$ for integer n and m , we formalize $\sum_{i=m}^n e$ as standing for $\sum i : m .. n . e$. This formally yields all rules, less the many ambiguities in common conventions. Variants, including one satisfying $\sum_m^k f + \sum_m^n f = \sum_m^n f$ for any integer m, k, n , are derived in the course.

Sequences and sequence types Tuples, arrays, lists, here jointly called *sequences*, are the most ubiquitous aggregate data structures in discrete mathematics. Here we recast them in a functional mold. Letting $\mathbb{N}' := \mathbb{N} \cup \iota \infty$, we define the *block operator* $\square : \mathbb{N}' \rightarrow \mathcal{P} \mathbb{N}$ by $\square n = \{m : \mathbb{N} \mid m < n\}$, so $\square 0 = \emptyset$ and $\square 2 = \mathbb{B}$ and $\square \infty = \mathbb{N}$.

A *sequence* is any function with domain $\square n$ for some $n : \mathbb{N}'$. The *length operator* $\#$ is defined by $\# x = n \equiv \mathcal{D}x = \square n$ for any sequence x and $n : \mathbb{N}'$. The *empty sequence* is ε , and the *singleton sequence injector* τ is defined by $\tau e = 0 \mapsto e$.

An *array of length n over set A* is a function of type $\square n \rightarrow A$, written $A \uparrow n$ or A^n . Note that $A^n = \times(\square n \bullet A)$. *Finite lists over A* are functions of type $\bigcup n : \mathbb{N} . A^n$, written A^* . Note that $A^\infty = \mathbb{N} \rightarrow A$ and $A^* \cap A^\infty = \emptyset$. We define A^ω as $A^* \cup A^\infty$.

Arbitrary tuple types are covered by \times . If T is a list of sets, $\times T \subseteq (\bigcup T) \# T$.

Operators over sequences The most important operator is *concatenation* $++$. For any sequences x and y , we define the domain of $x ++ y$ by $\#(x ++ y) = \#x + \#y$ and the mapping by $(x ++ y)i = (i < \#x) ? x i \dagger y(i - \#x)$ for all $i : \mathcal{D}(x ++ y)$. Useful properties are $x ++ \varepsilon = x = \varepsilon ++ x$ and associativity: $(x ++ y) ++ z = x ++ (y ++ z)$.

Derived operators are *prefixing* \succ and *postfixing* \prec with $e \succ x = \tau e ++ x$ and $x \prec e = x ++ \tau e$. Alternatively, we can let \succ be defined as the basic operator via its mapping and define $++$ recursively by $\varepsilon ++ y = y$ and $(a \succ x) ++ y = a \succ (x ++ y)$.

The induction principle for lists is $\forall P \equiv P \varepsilon \wedge \forall x : A^* . P x \Rightarrow \forall a : A . P(a \succ x)$ for any predicate P over A^* .

Formal calculation rules Deriving and using the formal calculation rules is the major topic of the courses listed above the title of this leaflet. Here is the syllabus.

- (a) Expressions, substitution, formal calculation with equality (reflexivity, symmetry, Leibniz’s principle); basic lambda calculus for handling bound and free variables.
- (b) Proposition calculus from 2 viewpoints: calculating with implication (\Rightarrow) and negation (\neg) as the basic operators, and *binary algebra* with \wedge and \vee as restrictions to \mathbb{B} of the \wedge (infimum) and \vee (supremum) operators on real numbers. Conditionals.
- (c) Formal calculation with sets, Funmath abstractions, tuples, generic functionals.
- (d) Functional predicate calculus, rules for \forall, \exists in point-free and point-wise form.
- (e) Well-foundedness and induction (general, over \mathbb{N} , structural, list induction).