Finite Functional Programming
or, LAMBDA: The Ultimate Predicate

Michael Arntzenius[0009−0002−0417−5636] and Max Willsey[0000−0001−8066−4218]

University of California, Berkeley, USA
daekharel@gmail.com mwillsey@berkeley.edu

Abstract. We unify functional and logic programming by treating predicates as functions equipped with their support: the set of inputs whose output is nonzero. Datalog, for instance, is a language of finitely supported boolean functions. Finite support allows representing functions as input-output tables. Generalizing from boolean functions to other pointed sets neatly handles aggregation and weighted logic programming. We refer to the combination of finitely supported functions, represented as data, with higher order functions, represented as code, as *finite functional programming*. We give a simple type system to check finite support, using graded effects to check variable grounding and relevance types to model pointed sets.

Keywords: Logic programming · Functional programming · Categorical semantics · Linear types · Relevance types · Graded effects

1 Logic programming is (not) functional programming

Consider the following simple logic program to find mutual follows in a social network graph:

```
follows(john, mary).
follows(mary, john).
% ... many more follows() facts.
mutuals(X,Y) :- follows(X,Y), follows(Y,X).
```

Predicates like `follows` and `mutuals` denote functions into booleans. Following this intuition, we can translate this logic program directly into a functional one:

$$
\begin{aligned}
&\mathit{follows}, \mathit{mutuals} : \mathit{user} \to \mathit{user} \to \mathit{bool} \\
&\mathit{follows}\ x\ y = (\mathit{john} == x \text{ and } \mathit{mary} == y) \\
&\qquad\qquad \text{or } (\mathit{mary} == x \text{ and } \mathit{john} == y) \\
&\qquad\qquad \text{or } \ldots \\
&\mathit{mutuals}\ x\ y = \mathit{follows}\ x\ y \text{ and } \mathit{follows}\ y\ x
\end{aligned}
$$

However, functions in functional and imperative languages are unidirectional: they take inputs to outputs. Evaluating $\mathit{mutuals}\ x\ y$ will test whether fixed users x, y are

mutuals—but unlike in logic programming, we cannot use it to enumerate mutual follows, for instance, to find the mutuals y of a fixed user x.

For the same reason, our functional translation cannot handle existential quantifiers (also called 'projection' in databases). Consider this logic program to find actors who have appeared in the same film:

```
costars(Actor1, Actor2) :-
  stars(Film, Actor1),
  stars(Film, Actor2).
```

Since `Film` does not appear in the head of the rule, the head is derivable if there exists a `Film` satisfying the body. To embed this functionally, we could assume a function $\mathit{exists} : \forall\alpha.\ (\alpha \to \mathit{bool}) \to \mathit{bool}$:

$\mathit{costars} : \mathit{person} \to \mathit{person} \to \mathit{bool}$
$\mathit{costars\ actor1\ actor2} = \mathit{exists}\ (\lambda \mathit{film}.\ \mathit{stars\ film\ actor1}$ and $\mathit{stars\ film\ actor2})$

But how can we implement *exists* for an arbitrary type α and function $f : \alpha \to \mathit{bool}$? We are stymied by unidirectionality: all we can do with f is give it inputs; since the input type is arbitrary, we cannot generate any.[1]

Logic programming implements existential quantification by rejecting input-output directionality: much as a function is a set of input-output tuples, a predicate can enumerate its tuples—or rather, those whose output is true (any non-enumerated inputs are implicitly false). We call these inputs the *support* of a predicate. If we know f's support, calculating *exists* f is simple: is the support nonempty?

This suggests we can make the direct functional translation of logic programs behave correctly if we equip functions with their support. Inspired by Datalog, we focus on finite support. Since finitely supported functions can be represented as key-value tables, we also call them *finite maps.* We make the following contributions:

1. We observe that defining a function's support requires a point in its codomain, suggesting the category Set_* of pointed sets and point preserving maps is a suitable site for the semantics of a functional logic programming language.
2. We show Set_* forms a model of relevance, a relaxation of linearity in which variables must be used at least once rather than exactly once.
3. We show finite maps form a graded monad/comonad on Set_*.
4. We construct a simply typed language λ_{FS} that can express relational algebra and some forms of aggregation, whose type system guarantees finite support.
5. We give a denotational semantics of λ_{FS} in Set_*.

Along the way we will observe other curious connections:

1. Database inner joins are point preserving maps out of the smash product $A \otimes B$ of pointed sets, while outer joins map out of the direct product $A \,\&\, B$.

[1] Of course, one can define *exists* for any finite type α by enumeration, and even for some infinite types by exploiting continuity of f [9,8]. However, our approach will be to presume, not static knowledge of the type, but dynamic knowledge of f's support.

2. Input/output modes in bottom-up logic programming correspond to procedural functions and finite maps respectively.
3. Checking a rule is well moded corresponds to an effect system for the graded monad of finite maps.

さあ始めましょう！ Let's get started!

2 Support, pointedly

The support of a boolean function f is the set $\{x \in A : f(x) \neq \mathit{false}\}$. It will be useful to generalize beyond booleans, firstly to expose compositional structure, for instance, to define the support of a curried function $f : A \to B \to \mathit{bool}$; and secondly to generalize from boolean logic and existential quantification (or relational algebra and projection, respectively) to weighted logic programming and aggregation (or tensor algebra and contraction).

To this end we write P, Q, R for sets with a designated element ('pointed sets') and $\mathit{nil}_P \in P$ for the designated element (the 'point'). (We use A, B, C for ordinary sets.) For instance, we regard bool as pointed with $\mathit{nil}_{\mathit{bool}} = \mathit{false}$. We define the support of a function $f : A \to P$ into a pointed set to be $\operatorname{supp} f = \{x \in A : f(x) \neq \mathit{nil}_P\}$. If one regards nil_P as an uninteresting, default value, a function's support contains those inputs with interesting, nondefault outputs.

We write $A \Rightarrow P = \{f \in A \to P : \operatorname{supp} f \text{ is finite}\}$ for the finitely supported functions from A to P. This is a pointed set with $\mathit{nil}_{A \Rightarrow P} = \lambda x.\ \mathit{nil}_P$; the boring, default finite map is the one with empty support. What can we do with finite maps? We can of course apply them—this is lookup in a key-value table. We can also curry and uncurry them: $A \times B \Rightarrow P \cong A \Rightarrow (B \Rightarrow P)$. This converts between a single flat table $A \times B \Rightarrow P$ and a trie-like nesting of tables $A \Rightarrow B \Rightarrow P$.

But how can we manipulate the output P of a finite map $A \Rightarrow P$? Can we, for instance, compose finite maps? Unfortunately, the composition of two finite maps may not have finite support: e.g. take $\lambda x.\ x < 3 : \mathbb{N} \Rightarrow \mathit{bool}$ followed by $\mathit{not} : \mathit{bool} \Rightarrow \mathit{bool}$. The set of naturals less than three is finite; its complement is not. For the same reason, we cannot compose a finite map $g : A \Rightarrow P$ with an arbitrary function $f : P \to Q$ and expect the result to be finitely supported.

However, we *can* compose a finite map with a function f if $f(\mathit{nil}_P) = \mathit{nil}_Q$; this preserves finiteness because it can only contract our support. These are the *point preserving maps*, which we notate $P \multimap Q = \{f \in P \to Q : f(\mathit{nil}_P) = \mathit{nil}_Q\}$. These are pointed with $\mathit{nil}_{P \multimap Q} = \lambda x.\ \mathit{nil}_Q$. The category Set_* has pointed sets P, Q as objects and point preserving maps as morphisms; what we have just observed is that, for any set A, there is a functor $F_A : \mathrm{Set}_* \to \mathrm{Set}_*$ given by $F_A(P) = A \Rightarrow P$. If finitely supported maps are this work's *raison d'être,* point preserving maps are how we will manipulate and combine them.

2.1 Direct or/and smash

Following logic programming, we take particular interest in two point preserving maps: boolean disjunction and conjunction. Let's start with their types. Recall that $nil_{bool} = false$. Observe that *or* yields *nil* when both its arguments are *nil*, while *and* yields *nil* when either argument is. We can reflect this both/either distinction using distinct types. Disjunction accepts a pair which is *nil* when both components are. This is the direct product of pointed sets, $P \,\&\, Q = \{\langle p, q\rangle : p \in P, q \in Q\}$ where $nil_{P\&Q} = \langle nil_P, nil_Q\rangle$. Conversely, conjunction accepts a pair which is *nil* if either component is. This is the smash product of pointed sets, $P \otimes Q = \{(p, q) : p \in P, q \in Q\}$ quotiented by $nil_{P\otimes Q} = (nil_P, y) = (x, nil_Q)$. Thus:

$$\text{or} : bool \,\&\, bool \multimap bool \qquad\qquad \text{and} : bool \otimes bool \multimap bool$$

It will be useful to generalize 'and' to type $bool \otimes P \multimap P$, along with its mirror image 'when' of type $P \otimes bool \multimap P$, defined:[2]

$$\begin{aligned} true \text{ and } x &= x \text{ when } true &&= x \\ false \text{ and } x &= x \text{ when } false &&= nil_P \end{aligned}$$

Finally, we must note an important asymmetry: curried maps $f : P \multimap Q \multimap R$ must preserve *nil* in each argument separately, like 'and' and unlike 'or':

$$\begin{aligned} &(f\ nil_P)\ q = nil_{Q\multimap R}\ q = nil_R && \text{since } f : P \multimap (Q \multimap R) \text{ preserves } nil \\ &(f\ p)\ nil_Q = nil_R && \text{since } f\ p : Q \multimap R \text{ preserves } nil \end{aligned}$$

In fact, currying and uncurrying of point preserving maps forms an isomorphism $P \otimes Q \multimap R \cong P \multimap Q \multimap R$. But there is no corresponding way to curry maps $P \,\&\, Q \multimap R$ (like 'or') that yield *nil* only when both arguments are *nil*.[3]

3 A finitely supported map from examples to booleans

Now that we've developed enough notation (summarized in figure 1, along with a few more pointed sets we will introduce later), we can consider some example programs and their types. For now we rely on intuition to see that they are semantically well typed; in sections 4–6 we will develop typing rules. First, let's consider some basic uses of conjunction (inner joins, in database parlance), starting with the simplest, cross product:

$cross : (A \Rightarrow bool) \multimap (B \Rightarrow bool) \multimap (A \Rightarrow B \Rightarrow bool)$
$cross\ f\ g\ x\ y = f\ x \text{ and } g\ y$

[2] An edifying exercise is to verify the generalized 'and' is one leg of an isomorphism $bool \otimes P \cong P$; its inverse is $\lambda x.\ (true, x) : P \multimap bool \otimes P$.

[3] Phrased categorically, Set_* does not have all exponential objects.

Notation	*Name*	*Elements*	*nil*
$P \,\&\, Q$	direct product	$\langle p, q\rangle$ for $p \in P, q \in Q$	$\langle nil_P, nil_Q\rangle$
$P \otimes Q$	smash product	(p, q) for $p \in P, q \in Q$ modulo:	$(nil_P, q) = (p, nil_Q)$
$P \multimap Q$	point preserving maps	$f : P \to Q$ with $f(nil_P) = nil_Q$	$\lambda\_.\ nil_Q$
$A \Rightarrow P$	finite maps	$f : A \to P$ with supp f finite	$\lambda\_.\ nil_P$
bool	booleans	*true*, *false*	*false*
$\mathbb{N}_0$	natural numbers	0, 1, 2, 3, 4...	0
maybe A	maybe type	just a for $a \in A$, none	none

Fig. 1. Pointed sets I have known and loved

$$
\begin{array}{rrcl}
\text{types} & A, B & ::= & P \mid A \to B \mid A \times B \mid 1 \mid \ldots \\
\text{pointed types} & P, Q & ::= & P \,\&\, Q \mid P \otimes Q \mid P \multimap Q \mid A \Rightarrow Q \mid \mathit{maybe}\ A \mid \mathbb{N}_0 \\
\text{expressions} & e & ::= & t \mid x \mid \lambda x.\, e \mid e_1\, e_2 \mid (e_1, e_2) \mid \pi_i\, e \mid () \\
 & & & \text{case } e_1 \text{ of just } x \to e_2;\ \text{none} \to e_3 \mid \ldots \\
\text{pointed terms} & t, u & ::= & \mathit{nil} \mid x \mid \lambda x.\, t \mid t\, u \mid t\, x \mid t\, e \mid \langle t, u\rangle \mid \pi_i\, t \\
 & & & (t, u) \mid \text{let } (x, y) = t \text{ in } u \mid \text{just } e \mid \text{let just } x = t \text{ in } u
\end{array}
$$

Fig. 2. Syntax of λ_{FS}

The type of *cross* captures something important: curried functions preserve *nil* in all arguments separately, and since $nil_{A \Rightarrow bool} = \lambda\_.\mathit{false}$ is the empty relation, we know from its type alone that the cross product of an empty set with any other relation is empty. Inner joins in general have this property because they use conjunction. For instance, intersection:

$\mathit{intersect} : (A \Rightarrow \mathit{bool}) \multimap (A \Rightarrow \mathit{bool}) \multimap (A \Rightarrow \mathit{bool})$
$\mathit{intersect}\ f\ g\ x = f\ x$ and $g\ x$

This intersects two finite sets, but we can also more generally 'intersect' (i.e. filter) a finite set with an arbitrary function $A \to \mathit{bool}$:

$\mathit{filter} : (A \Rightarrow \mathit{bool}) \multimap (A \to \mathit{bool}) \multimap (A \Rightarrow \mathit{bool})$
$\mathit{filter}\ f\ g\ x = f\ x$ and $g\ x$

We can also define the intersection of two arbitrary functions $f, g : A \to \mathit{bool}$, but this will only yield another function, not a finite map.

Let's move on to considering what we can express with the function $\mathit{exists} : (A \Rightarrow \mathit{bool}) \multimap \mathit{bool}$, now justified by the finite support of its argument:

$\mathit{costars} : \mathit{person} \Rightarrow \mathit{person} \Rightarrow \mathit{bool}$
$\mathit{costars}\ x\ y = \mathit{exists}\ (\lambda \mathit{film}.\ \mathit{stars}\ \mathit{film}\ x$ and $\mathit{stars}\ \mathit{film}\ y)$

Functions of the shape $(A \Rightarrow P) \multimap P$ represent *aggregations* into P. For instance, $\mathit{sum} : (A \Rightarrow \mathbb{N}_0) \multimap \mathbb{N}_0$ lets us count the number of films someone has starred in:

$$
\begin{array}{ll}
\mathit{bool} \longrightarrow \mathit{maybe}\ 1 & t \text{ and } u \longrightarrow \text{let just } \_ = t \text{ in } u \\
\mathit{true} \longrightarrow \text{just } () & \text{let } x = t \text{ in } u \longrightarrow \text{let } (x, y) = (t, \mathit{true}) \text{ in } (y \text{ and } u) \\
\mathit{false} \longrightarrow \mathit{nil} & t \text{ when } u \longrightarrow \text{let } x = t \text{ in } (u \text{ and } x) \\
\text{or} : \mathit{bool} \,\&\, \mathit{bool} \multimap \mathit{bool} & \mathit{exists} : (A \Rightarrow \mathit{bool}) \multimap \mathit{bool} \\
(+) : \mathbb{N}_0 \,\&\, \mathbb{N}_0 \multimap \mathbb{N}_0 & \mathit{sum} : (A \Rightarrow \mathbb{N}_0) \multimap \mathbb{N}_0 \\
(\times) : \mathbb{N}_0 \otimes \mathbb{N}_0 \multimap \mathbb{N}_0 & (=) : A \to (A \Rightarrow \mathit{bool})
\end{array}
$$

Fig. 3. Syntax sugar and primitive functions in λ_{FS}

$\mathit{filmCount} : \mathit{person} \Rightarrow \mathbb{N}_0$
$\mathit{filmCount}\ \mathit{actor} = \mathit{sum}\ (\lambda \mathit{film}.\ 1 \text{ when } \mathit{stars}\ \mathit{film}\ \mathit{actor})$

In general, for any commutative monoid of the form $(P, \oplus, \mathit{nil}_P)$, since $\mathit{nil} \oplus \mathit{nil} = \mathit{nil}$ we have $\oplus : P \,\&\, P \multimap P$. This extends to an aggregation $\bigoplus_P : (A \Rightarrow P) \multimap P$ that takes a finite map $f : A \Rightarrow P$ to the monoid sum $\bigoplus_{x \in \text{supp}\, f} f(x)$. For instance, the aggregation of the monoid $(\mathit{bool}, \mathit{or}, \mathit{false})$ is *exists*, while the aggregation of $(\mathbb{N}_0, +, 0)$ is *sum*.

Just as *exists* combined with $\mathit{and} : \mathit{bool} \otimes \mathit{bool} \multimap \mathit{bool}$ gave us relational composition in *costars*, summation combined with $\times : \mathbb{N}_0 \otimes \mathbb{N}_0 \multimap \mathbb{N}_0$ (which holds since $0 \times y = x \times 0 = 0 = \mathit{nil}_{\mathbb{N}_0}$) gives us matrix multiplication:

$\mathit{matMul} : (A \times B \Rightarrow \mathbb{N}_0) \multimap (B \times C \Rightarrow \mathbb{N}_0) \multimap A \times C \Rightarrow \mathbb{N}_0$
$\mathit{matMul}\ m\ n\ i\ k = \mathit{sum}\ (\lambda j.\ m\ i\ j \times n\ j\ k)$

We have already seen that $A \Rightarrow P$ is functorial in P. Given a commutative monoid aggregation over P, it is also functorial in the key space A, by aggregating the values of keys that collide under some map $f : A \to B$. For $\mathbb{N}_0$ under *sum* this is:

$\mathit{map} : (A \to B) \to (A \Rightarrow \mathbb{N}_0) \multimap (B \Rightarrow \mathbb{N}_0)$
$\mathit{map}\ f\ \mathit{count}\ b = \mathit{sum}\ (\lambda a.\ \mathit{count}\ a \text{ when } f\ a = b)$

Of course, finite sets $A \Rightarrow \mathit{bool}$ and bags $A \Rightarrow \mathbb{N}_0$ are not only functors but monads. If besides an aggregation *exists/sum* we have a monoid $(P, \circledast : P \otimes P \multimap P, \mathbf{1})$—for *bool* this is conjunction $(\mathit{bool}, \mathit{and}, \mathit{true})$, for $\mathbb{N}_0$ it is multiplication $(\mathbb{N}_0, \times, 1)$—then we can define monadic pure and join:

$$
\begin{array}{ll}
\mathit{pure} : A \to (A \Rightarrow \mathbb{N}_0) & \mathit{join} : ((A \Rightarrow \mathbb{N}_0) \Rightarrow \mathbb{N}_0) \multimap (A \Rightarrow \mathbb{N}_0) \\
\mathit{pure}\ x\ a = 1 \text{ when } x = a & \mathit{join}\ \mathit{nested}\ a = \mathit{sum}\ (\lambda t.\ \mathit{nested}\ t \times t\ a)
\end{array}
$$

We might even dare to consider defining finitely supported predicates recursively:

$\mathit{connected} : \mathit{person} \Rightarrow \mathit{person} \Rightarrow \mathit{bool}$
$\mathit{connected}\ x\ y = \mathit{costars}\ x\ y \text{ or } \mathit{exists}\ (\lambda z.\ \mathit{connected}\ x\ z \text{ and } \mathit{connected}\ z\ y)$

Unfortunately we will not be able to give typing rules or semantics for this kind of recursion here. Ensuring the existence of a fixed point would seem to require some

sort of monotonicity à la Datafun [3] or $\lambda_\vee$ [26], and ensuring it remains finite will be even more difficult; we leave this to future work.

Finally, let's turn our attention to the prototypical outer join: union.

$$\begin{array}{l} \mathit{union} : (A \Rightarrow \mathit{bool}) \mathbin{\&} (A \Rightarrow \mathit{bool}) \multimap (A \Rightarrow \mathit{bool}) \\ \mathit{union}\, \mathit{fg}\, x = \pi_1\, \mathit{fg}\, x \text{ or } \pi_2\, \mathit{fg}\, x \end{array}$$

We saw in section 2 why *union*, being a function on & pairs, cannot be curried. But why can't we destructure the pair *fg*? To explain this, we must understand the variable usage discipline that ensures functions preserve *nil*.

4 The relevance of being relevant

We've seen that point preserving maps can be applied to the outputs of finitely supported maps while preserving finite support, and allow us to combine multiple maps via inner joins (maps out of ⊗ pairs) and outer joins (maps out of & pairs). How can we check that maps preserve *nil*? To gain intuition, let's look at a few simple examples:

$$\begin{array}{rll} \mathit{id} = \lambda x.\, x & : P \multimap P & \checkmark \\ \mathit{three} = \lambda x.\, 3 & : P \multimap \mathbb{N}_0 & \text{✗ NOT POINT PRESERVING} \\ \mathit{dup}_\otimes = \lambda x.\, (x, x) & : P \multimap P \otimes P & \checkmark \\ \mathit{dup}_\& = \lambda x.\, \langle x, x \rangle & : P \multimap P \mathbin{\&} P & \checkmark \end{array}$$

Plainly the identity function preserves *nil*, but constant functions do not (except for $\lambda x.\, \mathit{nil}$). This suggests a linear type system, which ensures each variable is used exactly once; constant functions do not use their argument and so are prohibited. However, duplication does preserve *nil*, whether into ⊗ or & pairs: $(\mathit{nil}, \mathit{nil}) = \mathit{nil}$ and $\langle \mathit{nil}, \mathit{nil} \rangle = \mathit{nil}$. Or, consider the intersection ($\lambda x.\, f\, x$ and $g\, x$) of two maps $f, g : P \multimap \mathit{bool}$, which uses x twice yet still preserves *nil*. What we need is a *relevant* type system, which ensures variables are used *at least* once—although as we'll see presently, what counts as 'used' can be subtle.

So far our examples fail to distinguish the behavior of ⊗ from &; let's fix that:

$$\begin{array}{rll} \mathit{fst}_\otimes = \lambda p.\, \text{let } (x, y) = p \text{ in } x & : P \otimes Q \multimap P & \text{✗ NOT POINT PRESERVING} \\ \mathit{fst}_\& = \lambda p.\, \pi_1\, p & : P \mathbin{\&} Q \multimap P & \checkmark \\ \mathit{pair}_\otimes = \lambda x.\, \lambda y.\, (x, y) & : P \multimap Q \multimap P \otimes Q & \checkmark \\ \mathit{pair}_\& = \lambda x.\, \lambda y.\, \langle x, y \rangle & : P \multimap Q \multimap P \mathbin{\&} Q & \text{✗ NOT POINT PRESERVING} \\ \mathit{and3}_\otimes = \lambda x.\, (x, 3) & : P \multimap P \otimes \mathbb{N}_0 & \checkmark \\ \mathit{and3}_\& = \lambda x.\, \langle x, 3 \rangle & : P \multimap P \mathbin{\&} \mathbb{N}_0 & \text{✗ NOT POINT PRESERVING} \end{array}$$

One can verify these examples mechanically using the definitions of ⊗ and &, but some intuition may be helpful. To use an ⊗ pair (x, y) we must use both x and y,

to guarantee that if either is *nil* we will propagate it. By contrast, if a & pair is *nil* then both of its components are, so we are free to use only one. This is why $fst_{\otimes}$ is invalid but $fst_{\&}$ is correct—and also why we use destructuring to eliminate $\otimes$, but projection for &. Conversely, constructing an $\otimes$ pair (t, u) uses anything used by either t or u, since if either t or u is *nil* the whole pair is, but constructing a & pair $\langle t, u \rangle$ uses only what both t and u use, since only these will force both t and u to be *nil*. This is why $pair_{\otimes}$ and $and3_{\otimes}$ are correct but $pair_{\&}$ and $and3_{\&}$ are invalid.

Since we need to mix a cartesian, structural type system for sets A, B and functions $A \to B$ with a substructural, relevant type system for pointed sets P, Q and point preserving maps $P \multimap Q$, we adapt the rules of Benton and Wadler's mixed linear/nonlinear logic LNL [6,5] to relevance rather than linearity. We use two contexts, Γ containing ordinary variables $x : A$ and Δ containing pointed set variables $x : P$; two syntactic classes of terms, ordinary e and point preserving t; and two typing judgments, $\Gamma \vdash e : A$ for functions and $\Gamma \mid \Delta \vdash t : P$ for point preserving maps. A few example rules:

$$\frac{\Gamma \mid \Delta \vdash t : P \qquad \Gamma \mid \Delta \vdash u : Q}{\Gamma \mid \Delta \vdash \langle t, u \rangle : P \,\&\, Q} \qquad \frac{\Gamma \mid \Delta_1 \vdash t : P \qquad \Gamma \mid \Delta_2 \vdash u : Q}{\Gamma \mid \Delta_1 \cup \Delta_2 \vdash (t, u) : P \otimes Q}$$

$$\frac{}{\Gamma \mid x : P \vdash x : P} \qquad \frac{\Gamma \mid \Delta, x : P \vdash t : Q}{\Gamma \mid \Delta \vdash \lambda x.\, t : P \multimap Q}$$

Observe that in the rule for & pairs $\langle t, u \rangle$ each component is checked in the same relevant context Δ, and therefore must use the same variables; but in $\otimes$ pairs (t, u), we give them different contexts Δ_1, Δ_2 which must *union* to produce the context of the whole pair. This union is a key difference from standard linear logic, which would split the contexts disjointly; it allows a variable to be used in both branches—but, unlike the & rule, does not *require* it.

5 The other side of the tracks

Alas, these are not yet the typing rules we are looking for. A scant two contexts will not suffice; we have *three* kinds of function we'd like to introduce—ordinary $A \to B$, point preserving $P \multimap Q$, and finitely supported $A \Rightarrow P$—and therefore three kinds of variable, needing three separate contexts. Besides Γ and Δ we need a context Ω of finitely supported variables $x : A, y : B, \ldots$, and our typing judgement takes the form $\Gamma \mid \Delta \mid \Omega \vdash t : P$. To understand how Ω behaves, let's revisit the examples that opened the previous section:

$$\begin{aligned}
id &= \lambda x.\, x && : P \Rightarrow P && \text{✗ NOT FINITELY SUPPORTED} \\
three &= \lambda x.\, 3 && : A \Rightarrow \mathbb{N}_0 && \text{✗ NOT FINITELY SUPPORTED} \\
dup_{\otimes} &= \lambda x.\, (x, x) && : P \Rightarrow P \otimes P && \text{✗ NOT FINITELY SUPPORTED} \\
dup_{\&} &= \lambda x.\, \langle x, x \rangle && : P \Rightarrow P \,\&\, P && \text{✗ NOT FINITELY SUPPORTED}
\end{aligned}$$

Oh no! What went wrong? Well, constant functions like *three* are not finite because their support is their entire possibly infinite domain ($\lambda x.\ nil$ is the only exception). Similarly, the support of *id*, $dup_{\otimes}$, and $dup_{\&}$ is the entire domain minus *nil*; we cannot generally use a finite map's input directly in its output.[4]

If $\lambda x.\ x$ is not well typed, what does our variable rule look like? How do we use variables in our finitely supported context Ω? Let's return to our source of inspiration. In bottom-up logic programming, we generate nonempty, finite relations by either (1) combining other nonempty relations or (2) using rules that refer to constants. We saw in section 3 that in λ_{FS}, (1) means using point preserving maps to transform finite maps, and (2) means using equality $(=) : A \to A \Rightarrow bool$ to generate singleton finite maps. Let's consider a prototypical example of each kind:

$$actorOrDirector = \lambda x.\ actor\ x \text{ or } director\ x : person \Rightarrow bool \qquad \checkmark$$
$$hitchcockAlone = \lambda x.\ hitchcock = x \qquad : person \Rightarrow bool \qquad \checkmark$$

In each case, we use the finitely supported variable x by applying a finite map to it. This, then, will serve both as our elimination rule for finite maps and our usage rule for finitely supported variables:

$$\frac{\Gamma \mid \Delta \mid \Omega, x : A \vdash t : P}{\Gamma \mid \Delta \mid \Omega \vdash \lambda x.\ t : A \Rightarrow P}\ \Rightarrow \text{I} \qquad \frac{\Gamma \mid \Delta \mid \Omega \vdash t : A \Rightarrow P}{\Gamma \mid \Delta \mid \Omega, x : A \vdash t\ x : P}\ \Rightarrow \text{E}$$

These rules structuralize the isomorphism $A \Rightarrow B \Rightarrow P \cong A \times B \Rightarrow P$ (which is more obvious after renaming: $\Omega \Rightarrow A \Rightarrow P \cong \Omega \times A \Rightarrow P$). Along with the isomorphism $P \cong (1 \Rightarrow P)$, this shows that finitely supported maps form not just a functor but a *graded monad* on Set_*.

A (non-graded) monad is an endofunctor F with natural transformations $pure_\alpha : \alpha \to F\alpha$ and $join_\alpha : FF\alpha \to F\alpha$, satisfying laws which we omit for brevity. For our purposes, a graded monad [28,16] is a family of functors F_m where $m \in M$ is drawn from some monoid of grades $(M, \cdot, 1)$, with analogues of *pure* and *join* that interact with the grading monoid: $pure_\alpha : \alpha \to F_1\alpha$ and $join_\alpha : F_mF_n\alpha \to F_{mn}\alpha$, again satisfying certain omitted laws.

In our case, the graded monad is $F_AP = A \Rightarrow P$, and the monoid of grades is sets under cross product.[5] This makes $pure_P : P \multimap (1 \Rightarrow P)$ and $join_P : (A \Rightarrow B \Rightarrow P) \multimap (A \times B \Rightarrow P)$ merely the forward legs of our two isomorphisms. The laws we omitted hold trivially because these maps are isomorphisms—indeed, the isomorphisms' reverse legs also make F_A a graded comonad [24]. Although almost trivial, this graded (co)monad is of practical interest because type systems for graded monads (effect systems) have been extensively studied [12].

[4] We could in principle make a carve-out for maps with finite domains. As we intend to represent finite maps by tables, however, this poses a usability hazard: enumerating all 64-bit integers just because a programmer wrote $\lambda x.\ x$ is not desirable.

[5] Technically this is not a monoid, since $(A \times B) \times C$ and $A \times (B \times C)$ are only isomorphic, not equal. This technicality is not worth getting hung up on; it can be defeated, for instance, by taking grades to be contexts Ω under concatenation rather than sets.

Unfortunately, our typing rules do not all follow neatly from this connection to grading. Let's examine the rules for & and ⊗. Both are informative: the former because it is straightforward, the latter because it is not. The rules for & are:

$$\frac{\Gamma \mid \Delta \mid \Omega \vdash t : P \qquad \Gamma \mid \Delta \mid \Omega \vdash u : Q}{\Gamma \mid \Delta \mid \Omega \vdash \langle t, u \rangle : P \,\&\, Q}\ \&\text{I} \qquad \frac{\Gamma \mid \Delta \mid \Omega \vdash t : P_1 \,\&\, P_2}{\Gamma \mid \Delta \mid \Omega \vdash \pi_i\, t : P_i}\ \&\text{E}$$

Now let's consider what rules we would need to type the following examples involving ⊗ (by way of $and : bool \otimes bool \multimap bool$):

$$f : A \Rightarrow bool, g : A \Rightarrow bool \mid \cdot \mid x : A, y : A \vdash f\ x \text{ and } g\ y : bool$$
$$f : A \Rightarrow bool, g : A \Rightarrow bool \mid \cdot \mid x : A \qquad \vdash f\ x \text{ and } g\ x : bool$$
$$f : A \Rightarrow bool, g : A \to bool \mid \cdot \mid x : A \qquad \vdash f\ x \text{ and } g\ x : bool$$

Since we wish to guarantee finite support, we first ask: what is the support of each example (as a function of the supports of f, g)?

The support of $f\ x$ and $g\ y$ is the cross product of the supports of $f : A \Rightarrow bool$ and $g : A \Rightarrow bool$. Fortunately, the cross product of finite sets is finite. So what typing rule does this example need? We have used each variable x, y in our finite support context Ω exactly once, so it suffices to be able to split this context between the (implicit) ⊗ pair of arguments to 'and':

$$\frac{\Gamma \mid \Delta_1 \mid \Omega_1 \vdash t : P \qquad \Gamma \mid \Delta_2 \mid \Omega_2 \vdash u : Q}{\Gamma \mid \Delta_1 \cup \Delta_2 \mid \Omega_1, \Omega_2 \vdash (t, u) : P \otimes Q}$$

This rule does not suffice for our second example, $f\ x$ and $g\ x$, which uses the same finitely supported variable x twice. Nonetheless this has finite support, namely, the intersection of f's and g's supports. So we can relax our rules to union contexts, much like the rules for the relevant context Δ:

$$\frac{\Gamma \mid \Delta_1 \mid \Omega_1 \vdash t : P \qquad \Gamma \mid \Delta_2 \mid \Omega_2 \vdash u : Q}{\Gamma \mid \Delta_1 \cup \Delta_2 \mid \Omega_1 \cup \Omega_2 \vdash (t, u) : P \otimes Q}$$

Yet this still fails to check our final example, which filters the finite set $f : A \Rightarrow bool$ by an arbitrary boolean function $g : A \to bool$. This is unproblematic semantically: we can filter a finite set by any predicate we like and the result remains finite! The solution is a rule that breaks the rules: any variable finitely supported by t may be used unrestricted in u, so the finite support context Ω_1 for t will *jump* across the railway tracks into the unrestricted context for u:

$$\frac{\Gamma \mid \Delta_1 \mid \Omega_1 \vdash t : P \qquad \Gamma, \Omega_1 \mid \Delta_2 \mid \Omega_2 \vdash u : Q}{\Gamma \mid \Delta_1 \cup \Delta_2 \mid \Omega_1, \Omega_2 \vdash (t, u) : P \otimes Q}\ \otimes\text{I}$$

This rule is asymmetric: it 'grounds' variables from left to right. This choice is arbitrary—right to left is just as sound—but the symmetric variant, which allows

variables finitely supported by either t or u to be used unrestricted in the other, allows circular dataflow and is unsound. For instance, there may be infinitely many x, y such that $x = y$, but for fixed x there is exactly one such y; this is why we give equality the type $A \to (A \Rightarrow bool)$. However, if we allow each branch to ground variables used in the other, then ($x = y$ and $y = x$), though semantically identical to $x = y$, would incorrectly appear to finitely support both x and y: in the left branch, x is used to ground y, and vice-versa.

To ensure our last rule generalizes the two prior ones, we must be able to weaken a finitely supported hypothesis $x : A \in \Omega$ to an unrestricted one $x : A \in \Gamma$. Then a variable finitely supported by both t and u can simply be placed into Ω_1 in $\otimes$ I. Unfortunately this doesn't yet hold: $\Rightarrow$ E lets us apply finite maps $A \Rightarrow P$ to variables in Ω but not in Γ! We need an additional rule that lets us apply a finite map to an unrestricted variable x, or more generally an expression e which (as usual) may use unrestrictedly any variables finitely supported by t:[6]

$$\frac{\Gamma \,/\, \Delta \,/\, \Omega \vdash t : A \Rightarrow P \qquad \Gamma, \Omega \vdash e : A}{\Gamma \,/\, \Delta \,/\, \Omega \vdash t\ e : P} \Rightarrow \text{E}_2$$

6 Types and semantics

We give the typing rules of λ_{FS} in figure 4. The attentive reader will find few surprises. As mentioned in section 4, our rules borrow from adjoint calculi such as LNL [6,5] to handle the interaction between ordinary functions and point preserving functions. This corresponds semantically to an adjunction between the cartesian closed category Set (LNL's $\mathcal{C}$) and the symmetric monoidal closed category Set_* (LNL's $\mathcal{L}$). Additionally, Set_* possesses a natural family of diagonal maps $\lambda x.(x, x) : P \multimap P \otimes P$, making it a relevant monoidal category [23,18]. The adjunction in question consists of the free functor *maybe* : Set $\to \text{Set}_*$ and the forgetful functor $U : \text{Set}_* \to$ Set. Our main difference from a standard adjoint calculus is our additional context Ω of finitely supported variables, which in most typing rules follows the context-hopping pattern of $\otimes$ introduction that we explored in section 5.

Unsurprisingly, we interpret 'set' types as sets $[\![A]\!] \in$ Set, and 'pointed set' types as pointed sets $[\![P]\!] \in \text{Set}_*$. We have deliberately used almost identical notation for the syntax of types and for the mathematical objects they denote. See figure 5 for the semantics of types and contexts. The n-ary smash product $\bigotimes_i P_i$ deserves some explanation: this denotes the pointed set $\{(x_1, \ldots, x_n) : x_1 \in P_1, \ldots, x_n \in P_n\} \uplus \{nil\}$ quotiented by $\forall i.\ x_i = nil \implies (x_1, \ldots, x_n) = nil$. The explicit addition of *nil* means that the nullary smash product, $\bigotimes\{\}$, has two elements: *nil* and the empty tuple $()$, which is important for interpreting the case $\Delta = \cdot$ of an empty pointed set context.

[6] Adding Ω to e's context is also needed to allow multiple uses of a finitely supported variable among the arguments to a nested finite map, e.g. $\cdot \,/\, f : A \Rightarrow A \Rightarrow P \,/\, x : A \vdash f\ x\ x : P$.

$\boxed{\Gamma \vdash e : A}$

$$\frac{\Gamma \,/\, \cdot \,/\, \cdot \vdash t : P}{\Gamma \vdash t : P}\ \textsc{ui} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{evar} \qquad \frac{}{\Gamma \vdash () : 1}\ 1\textsc{i} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\ \to\textsc{i}$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\ \to\textsc{e} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}\ \times\textsc{i}$$

$$\frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\, e : A_i}\ \times\textsc{e}_i \qquad \frac{\Gamma \vdash e_1 : \mathit{maybe}\ A \qquad \Gamma, x : A \vdash e_2 : B \qquad \Gamma \vdash e_3 : B}{\Gamma \vdash \text{case } e_1 \text{ of just } x \to e_2;\ \text{none} \to e_3 : B}\ \textsc{case}$$

$\boxed{\Gamma \,/\, \Delta \,/\, \Omega \vdash t : P}$

$$\frac{\Gamma \vdash e : P}{\Gamma \,/\, \cdot \,/\, \cdot \vdash e : P}\ \textsc{ue} \qquad \frac{}{\Gamma \,/\, x : P \,/\, \cdot \vdash x : P}\ \textsc{var} \qquad \frac{}{\Gamma \,/\, \Delta \,/\, \Omega \vdash \mathit{nil} : P}\ \textsc{nil}$$

$$\frac{\Gamma \,/\, \Delta, x : P \,/\, \cdot \vdash t : Q}{\Gamma \,/\, \Delta \,/\, \cdot \vdash \lambda x.\, t : P \multimap Q}\ \multimap\textsc{i} \qquad \frac{\Gamma \,/\, \Delta \,/\, \Omega, x : A \vdash t : P}{\Gamma \,/\, \Delta \,/\, \Omega \vdash \lambda x.\, t : A \Rightarrow P}\ \Rightarrow\textsc{i}$$

$$\frac{\begin{array}{c}\Gamma \,/\, \Delta_1 \,/\, \Omega_1 \vdash t : P \multimap Q \\ \Gamma, \Omega_1 \,/\, \Delta_2 \,/\, \Omega_2 \vdash u : P\end{array}}{\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash t\, u : Q}\ \multimap\textsc{e} \qquad \frac{\Gamma \,/\, \Delta \,/\, \Omega \vdash t : A \Rightarrow P}{\Gamma \,/\, \Delta \,/\, \Omega, x : A \vdash t\, x : P}\ \Rightarrow\textsc{e}$$

$$\frac{\begin{array}{c}\Gamma \,/\, \Delta \,/\, \Omega \vdash t : P \\ \Gamma \,/\, \Delta \,/\, \Omega \vdash u : Q\end{array}}{\Gamma \,/\, \Delta \,/\, \Omega \vdash \langle t, u \rangle : P \,\&\, Q}\ \&\textsc{i} \qquad \frac{\begin{array}{c}\Gamma \,/\, \Delta_1 \,/\, \Omega_1 \vdash t : P \\ \Gamma, \Omega_1 \,/\, \Delta_2 \,/\, \Omega_2 \vdash u : Q\end{array}}{\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash (t, u) : P \otimes Q}\ \otimes\textsc{i}$$

$$\frac{\Gamma \,/\, \Delta \,/\, \Omega \vdash t : P_1 \,\&\, P_2}{\Gamma \,/\, \Delta \,/\, \Omega \vdash \pi_i\, t : P_i}\ \&\textsc{e} \qquad \frac{\begin{array}{c}\Gamma \,/\, \Delta_1 \,/\, \Omega_1 \vdash t : P \otimes Q \\ \Gamma, \Omega_1 \,/\, \Delta_2, x : P, y : Q \,/\, \Omega_2 \vdash u : Q\end{array}}{\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash \text{let } (x, y) = t \text{ in } u : Q}\ \otimes\textsc{e}$$

$$\frac{\Gamma \,/\, \Delta \,/\, \Omega \vdash t : A \Rightarrow P \qquad \Gamma, \Omega \vdash e : A}{\Gamma \,/\, \Delta \,/\, \Omega \vdash t\, e : P}\ \Rightarrow\textsc{e}_2 \qquad \frac{\Gamma \vdash e : A}{\Gamma \,/\, \cdot \,/\, \cdot \vdash \text{just } e : \mathit{maybe}\ A}\ \textsc{maybe i}$$

$$\frac{\Gamma \,/\, \Delta_1 \,/\, \Omega_1 \vdash t : \mathit{maybe}\ A \qquad \Gamma, \Omega_1, x : A \,/\, \Delta_2 \,/\, \Omega_2 \vdash u : P}{\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash \text{let just } x = t \text{ in } u : P}\ \textsc{maybe e}$$

Fig. 4. λ_{FS} typing rules

$$[\![1]\!] = 1 \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!] \qquad [\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \qquad [\![P]\!] = U[\![P]\!]$$

$$[\![P \,\&\, Q]\!] = [\![P]\!] \,\&\, [\![Q]\!] \qquad [\![P \otimes Q]\!] = [\![P]\!] \otimes [\![Q]\!] \qquad [\![P \multimap Q]\!] = [\![P]\!] \multimap [\![Q]\!]$$
$$[\![A \Rightarrow P]\!] = [\![A]\!] \Rightarrow [\![P]\!] \qquad [\![\mathit{maybe}\ A]\!] = \mathit{maybe}\ [\![A]\!] \qquad [\![\mathbb{N}_0]\!] = \mathbb{N}_0$$

$$[\![\Gamma]\!] = \prod_{x:A \in \Gamma} [\![A]\!] \qquad [\![\Delta]\!] = \bigotimes_{x:P \in \Delta} [\![P]\!] \qquad [\![\Omega]\!] = \prod_{x:A \in \Gamma} [\![A]\!]$$

Fig. 5. Semantics of types and contexts in λ_{FS}

SEMANTICS OF $\Gamma \vdash e : A$

$$\begin{aligned}
[\![t]\!]\,\gamma &= [\![t]\!]\,\gamma\,()\,() \\
[\![x]\!]\,\gamma &= \pi_x \gamma \\
[\![()]\!]\,\gamma &= () \\
[\![\lambda x.\, e]\!]\,\gamma &= \lambda x.\, [\![e]\!]\,(\gamma, x) \\
[\![e_1\ e_2]\!]\,\gamma &= [\![e_1]\!]\,\gamma\,([\![e_2]\!]\,\gamma) \\
[\![(e_1, e_2)]\!]\,\gamma &= ([\![e_1]\!]\,\gamma,\ [\![e_2]\!]\,\gamma) \\
[\![\pi_i\ e]\!]\,\gamma &= \pi_i([\![e]\!]\,\gamma) \\
[\![\text{case } e_1 \text{ of just } x \to e_2;\ \text{none} \to e_3]\!] &= \begin{cases} [\![e_2]\!]\,(\gamma, x) & \text{if } [\![e_1]\!]\,\gamma = \text{just } x \\ [\![e_3]\!]\,\gamma & \text{otherwise} \end{cases}
\end{aligned}$$

SEMANTICS OF $\Gamma \,/\, \Delta \,/\, \Omega \vdash t : P$

$$\begin{aligned}
[\![\Gamma \,/\, \cdot \,/\, \cdot \vdash e : P]\!]\,\gamma\,\delta &= \{() \mapsto [\![e]\!]\,\gamma : \delta \neq \mathit{nil}\} \\
[\![\Gamma \,/\, \Delta \,/\, \Omega \vdash \mathit{nil} : P]\!]\,\gamma\,\delta &= \{\} \\
[\![\Gamma \,/\, x : P \,/\, \cdot \vdash x : P]\!]\,\gamma\,x &= \{() \mapsto x\} \\
[\![\Gamma \,/\, \Delta \,/\, \cdot \vdash \lambda x.\, t : P \multimap Q]\!]\,\gamma\,\delta &= \{() \mapsto \lambda x.\, [\![t]\!]\,\gamma\,(\delta, x)\} \\
[\![\Gamma \,/\, \Delta \,/\, \Omega \vdash \lambda x.\, t : A \Rightarrow P]\!]\,\gamma\,\delta &= \{\omega \mapsto \{x \mapsto y : (\omega, x) \mapsto y \in [\![t]\!]\,\gamma\,\delta\} : \exists x, y.\ (\omega, x) \mapsto y \in [\![t]\!]\,\gamma\,\delta\} \\
[\![\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash t\ u : Q]\!]\,\gamma\,\delta &= \{(\omega_1, \omega_2) \mapsto x\ y : \omega_1 \mapsto x \in [\![t]\!]\,\gamma\,(\pi_{\Delta_1}\delta),\ \omega_2 \mapsto y \in [\![u]\!]\,(\gamma, \omega_1)\,(\pi_{\Delta_2}\delta)\} \\
[\![\Gamma \,/\, \Delta \,/\, \Omega, x : A \vdash t\ x : P]\!]\,\gamma\,\delta &= \{(\omega, x) \mapsto y : \omega \mapsto f \in [\![t]\!]\,\gamma\,\delta,\ x \mapsto y \in f\} \\
[\![\Gamma \,/\, \Delta \,/\, \Omega \vdash t\ e : P]\!]\,\gamma\,\delta &= \{\omega \mapsto f([\![e]\!]\,(\gamma, \omega)) : \omega \mapsto f \in [\![t]\!]\,\gamma\,\delta\} \\
[\![\Gamma \,/\, \Delta \,/\, \Omega \vdash \langle t, u\rangle : P \,\&\, Q]\!]\,\gamma\,\delta &= \{\omega \mapsto \langle x, y\rangle : \omega \mapsto x \in [\![t]\!]\,\gamma\,\delta,\ \omega \mapsto y \in [\![u]\!]\,\gamma\,\delta\} \\
[\![\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash (t, u) : P \otimes Q]\!]\,\gamma\,\delta &= \{(\omega_1, \omega_2) \mapsto (x, y) : \omega_1 \mapsto x \in [\![t]\!]\,\gamma\,(\pi_{\Delta_1}\delta),\ \omega_2 \mapsto y \in [\![u]\!]\,(\gamma, \omega_1)\,(\pi_{\Delta_2}\delta)\} \\
[\![\Gamma \,/\, \Delta \,/\, \Omega \vdash \pi_i\ t : P_i]\!]\,\gamma\,\delta &= \{\omega \mapsto \pi_i\, x : \omega \mapsto x \in [\![t]\!]\,\gamma\,\delta\} \\
[\![\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash \text{let } (x, y) = t \text{ in } u : Q]\!]\,\gamma\,\delta &= \{(\omega_1, \omega_2) \mapsto z : \omega_1 \mapsto (x, y) \in [\![t]\!]\,\gamma\,(\pi_{\Delta_1}\delta),\ \omega_2 \mapsto z \in [\![u]\!]\,(\gamma, \omega_1)\,(\pi_{\Delta_2}\delta, x, y)\} \\
[\![\Gamma \,/\, \cdot \,/\, \cdot \vdash \text{just } e : \mathit{maybe}\ A]\!]\,\gamma\,\delta &= \{() \mapsto \text{just}\,([\![e]\!]\,\gamma) : \delta \neq \mathit{nil}\} \\
[\![\Gamma \,/\, \Delta_1 \cup \Delta_2 \,/\, \Omega_1, \Omega_2 \vdash \text{let just } x = t \text{ in } u : P]\!]\,\gamma\,\delta &= \{(\omega_1, \omega_2) \mapsto y : \omega_1 \mapsto \text{just } x \in [\![t]\!]\,\gamma\,(\pi_{\Delta_1}\delta),\ \omega_2 \mapsto y \in [\![u]\!]\,(\gamma, \omega_1, x)\,(\pi_{\Delta_2}\delta)\}
\end{aligned}$$

Fig. 6. Semantics of expressions and terms in λ_{FS}

$$[\![\Gamma \vdash e : A]\!] \in [\![\Gamma]\!] \to [\![A]\!]$$
$$[\![\Gamma \mid \Delta \mid \Omega \vdash t : P]\!] \in [\![\Gamma]\!] \to [\![\Delta]\!] \multimap [\![\Omega]\!] \Rightarrow [\![P]\!]$$

We give semantics to expressions e and terms t in figure 6. The term semantics often depend on the splitting of the Δ and Ω contexts, so we include the conclusion of the typing judgment in the semantic brackets. For space reasons we could not include the premises, so cross-referencing with figure 4 is suggested. To make it clear that terms are finitely supported with respect to the context Ω, we construct all finite maps explicitly using set comprehensions as sets of input-output pairs, $x \mapsto y$.

7 Looking back and forward

There are many ways to combine logic and functional programming. Mercury [29] integrates functional programming into top-down logic programming. Functional IncA [22] embeds functional programming into Datalog via a demand transform. Flix [20] has two sublanguages, logical and functional, which can each invoke the other.

λ_{FS}, however, belongs to the subfamily of languages that integrate logical features into a functional substrate. Perhaps its best known exemplars are 'functional logic programming' languages like Curry [1] and Verse [4], where relations $A \times B \to$ *bool* become 'functions' $A \to B$ that may yield many Bs for a given A. We find this undesirable because it (a) makes multiple returns and unification into pervasive ambient effects and (b) hides the boolean-ness, losing the opportunity to generalize from *bool* to other types. λ_{FS} is closer to Datafun [3] and $\lambda_{\vee}$ [26], which avoid ambient effects by representing relations as finite sets, a separate type from normal functions. However, they express logical operations via loops or set comprehensions. In contrast λ_{FS} is inspired by Rel [2], where relations are defined pointwise like functions, using 'and', 'or', and 'exists'. Rel also supports arithmetic operators and aggregations, as if relations had return values. However, Rel implements this by desugaring to a Datalog-like core where the 'value' of a Rel expression is just the relation's final column. λ_{FS} reimagines this syntactic sugar semantically.

By generalizing beyond booleans, λ_{FS} moves closer to weighted logic programming languages like Dyna [10,11], ProbLog [25], or Datalog$^\circ$ [17]. Indeed, λ_{FS} is inspired by and indebted to databases research on K-relations [14], generalizing Datalog semantics from booleans to semirings, which has in turn inspired functional query languages [15,27]. Other Datalog-inspired languages like Datafun, Flix, and $\lambda_{\vee}$ generalize to semilattices. However, so far as we know only λ_{FS} uses pointed sets—the minimal, most general structure required to define a function's support. Semirings then emerge as a natural structure over pointed sets: we can generalize *or* and *exists* to an 'additive' commutative monoid $(P, \oplus : P \,\&\, P \multimap P, nil_P)$ and its aggregation operator $\bigoplus : (A \Rightarrow P) \multimap P$, and similarly *and* generalizes to a 'multiplicative' monoid $(P, \circledast : P \otimes P \multimap P, \mathbf{1})$ distributing over $\oplus$.

λ_{FS} has many limitations which we would like to lift in future work:

Recursion There are two kinds of recursion we might want in λ_{FS}: lazy functional recursion (as in recursively defined functions or codata) and bottom-up iteration until reaching a fixed point (as in Datalog's recursively defined relations). It's not obvious how these should interact semantically. λ_{V} [26] shows how to unify them, but requires a lot of structure (semilattices, dcpos, and continuity); moreover, it seemingly cannot do stratified computation, where we compute a fixed point and then use its results in a non-monotone way.

Implementation We did not show how to implement either our type system or our semantics, although we have built a prototype (https://github.com/rntz/fslang/blob/main/fslang.rkt) which we lack space to discuss here. As our finite map operations are a form of tensor algebra, perhaps indexed streams [19] could provide an efficient implementation technique.

Interleaving Δ/Ω contexts We use three contexts, $\Gamma \mid \Delta \mid \Omega \vdash P$, glossed as $\Gamma \to \Delta \multimap \Omega \Rightarrow P$. This is not enough: since $P \multimap (A \Rightarrow Q) \not\cong A \Rightarrow (P \multimap Q)$, writing programs of the latter type naturally requires another context, $\Gamma \mid \Delta_1 \mid \Omega \mid \Delta_2 \vdash P$. In general we may need any number of interleavings of Δ and Ω contexts. Since our contexts handle the interaction of a graded effect (Ω for finite maps) and a coeffect (Γ for the *maybe* comonad) on the category $\text{Set}_*(\Delta)$, we hope to apply or extend prior work on combining graded effects and coeffects [13].

Free grounding order Following the pattern of $\otimes\,\text{I}$, our rules ground finitely supported variables left to right. This choice is arbitrary: there need only be some order which grounds all variables without circularity. This ordering can discovered automatically for Datalog programs [7]. Extending this to λ_{FS} would free the programmer from thinking about grounding order.

Destructuring & pairs Outer joins like *union* are most naturally expressed as destructuring their & pair argument, but standard linear/relevance typing rules do not allow this. We suspect it may be possible by stealing rules from the logic of bunched implications [21].

Patterns and finite maps We would like to extend finite map λ and application to patterns, such as tuples $\lambda(x, y).\ t$ and $t\ (x, y)$. Otherwise, for example, currying/uncurrying of finite maps is quite tricky.[7] Ideally, application patterns could mix grounding finitely supported variables (as in $\Rightarrow \text{E}$) with looking up expressions (as in $\Rightarrow \text{E}_2$), e.g. $x : A \mid f : A \times A \Rightarrow P \mid y : A \vdash f\ (x, y) : P$. However, the unusual behavior of finite support contexts Ω makes specifying and implementing finite support patterns challenging.

[7] The confident reader may take this as a brainteaser. Make sure your solution has a typing derivation! Our solution (upside-down):

$$\begin{aligned}
&\mathit{curry} : (A \times B \Rightarrow P) \multimap (A \Rightarrow B \Rightarrow P)\\
&\mathit{curry}\ f\ a\ b = (\lambda x.\ f\ x \text{ when } \pi_1\ x = a \text{ and } \pi_2\ x = b)\ (a, b)\\
&\mathit{uncurry} : (A \Rightarrow B \Rightarrow P) \multimap (A \times B \Rightarrow P)\\
&\mathit{uncurry}\ f\ ab = (\lambda a.\ \lambda b.\ f\ a\ b \text{ when } (a, b) = ab)\ (\pi_1\ ab)\ (\pi_2\ ab)
\end{aligned}$$

The secret is to combine left-to-right grounding with an immediately applied finite λ:

Metatheory λ_{FS} lacks metatheory. Our semantics is a sketch; we have not proven semantic finite support or point preservation, nor syntactic weakening or substitution. The need for $\Rightarrow \text{E}_2$ should show this is surprisingly subtle. Moreover, because of left-to-right grounding, substitution fails for finitely supported variables: $(\lambda x.\ t)\ u$ may have a typing derivation when $t\{x \mapsto u\}$ does not (see previous footnote's solution for an example).

References

1. Antoy, S., Hanus, M.: Functional logic programming. Communications of the ACM **53**(4), 74–85 (Apr 2010)
2. Aref, M., Guagliardo, P., Kastrinis, G., Libkin, L., Marsault, V., Martens, W., McGrath, M., Murlak, F., Nystrom, N., Peterfreund, L., Rogers, A., Sirangelo, C., Vrgoč, D., Zhao, D., Zreika, A.: Rel: A programming language for relational data. In: Companion of the 2025 International Conference on Management of Data. p. 283–296. SIGMOD/PODS '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3722212.3724450
3. Arntzenius, M., Krishnaswami, N.R.: Datafun: A functional Datalog. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 214–227. ICFP 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2951913.2951948
4. Augustsson, L., Breitner, J., Claessen, K., Jhala, R., Jones, S.P., Shivers, O., Jr., G.L.S., Sweeney, T.: The verse calculus: A core calculus for deterministic functional logic programming. Proc. ACM Program. Lang. **7**(ICFP), 417–447 (2023). https://doi.org/10.1145/3607845
5. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In: Pacholski, L., Tiuryn, J. (eds.) Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers. Lecture Notes in Computer Science, vol. 933, pp. 121–135. Springer (1994). https://doi.org/10.1007/BFB0022251
6. Benton, P.N., Wadler, P.: Linear logic, monads and the lambda calculus. In: Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996. pp. 420–431. IEEE Computer Society (1996). https://doi.org/10.1109/LICS.1996.561458
7. Contrastin, M., Orchard, D.A., Rice, A.C.: Automatic reordering for dataflow safety of datalog. In: Sabel, D., Thiemann, P. (eds.) Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018. pp. 9:1–9:17. ACM (2018). https://doi.org/10.1145/3236950.3236954
8. Escardó, M.H.: Infinite sets that admit fast exhaustive search. In: 22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings. pp. 443–452. IEEE Computer Society (2007). https://doi.org/10.1109/LICS.2007.25
9. Escardó, M.H.: Seemingly impossible functional programs. Blog post (2007), https://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/
10. Filardo, N.W.: Dyna 2: Towards a General Weighted Logic Language. Ph.D. thesis, Johns Hopkins University (10 2017), https://www.ietfng.org/nwf/publications-and-talks/2017-thesis.html

11. Francis-Landau, M.: Declarative Programming via Term Rewriting. Ph.D. thesis, Johns Hopkins University (1 2024), https://matthewfl.com/research#phd
12. Gaboardi, M., Katsumata, S.y., Orchard, D., Breuvart, F., Uustalu, T.: Combining effects and coeffects via grading. SIGPLAN Not. **51**(9), 476–489 (Sep 2016). https://doi.org/10.1145/3022670.2951939
13. Gaboardi, M., Katsumata, S.y., Orchard, D., Breuvart, F., Uustalu, T.: Combining effects and coeffects via grading. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. p. 476–489. ICFP 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2951913.2951939, https://doi.org/10.1145/2951913.2951939
14. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Libkin, L. (ed.) Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China. pp. 31–40. ACM (2007). https://doi.org/10.1145/1265530.1265535, https://doi.org/10.1145/1265530.1265535
15. Henglein, F., Kaarsgaard, R., Mathiesen, M.K.: The programming of algebra. In: Gibbons, J., New, M.S. (eds.) Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April 2022. EPTCS, vol. 360, pp. 71–92 (2022). https://doi.org/10.4204/EPTCS.360.4
16. Katsumata, S.y.: Parametric effect monads and semantics of effect systems. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 633–645. POPL '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2535838.2535846
17. Khamis, M.A., Ngo, H.Q., Pichler, R., Suciu, D., Wang, Y.R.: Convergence of datalog over (pre-) semirings. J. ACM **71**(2), 8:1–8:55 (2024). https://doi.org/10.1145/3643027
18. Kosta, D., Petrić, Z.: Relevant categories and partial functions. Publications de l'Institut Mathematique **96**, 17–23 (2007). https://doi.org/10.2298/PIM0796017D
19. Kovach, S., Kolichala, P., Gu, T., Kjolstad, F.: Indexed streams: A formal intermediate representation for fused contraction programs. Proc. ACM Program. Lang. **7**(PLDI) (Jun 2023). https://doi.org/10.1145/3591268
20. Madsen, M., Lhoták, O.: Fixpoints for the masses: programming with first-class datalog constraints. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428193
21. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symb. Log. **5**(2), 215–244 (1999). https://doi.org/10.2307/421090
22. Pacak, A., Erdweg, S.: Functional programming with datalog. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, Berlin, Germany, June 6-10, 2022. LIPIcs, vol. 222, pp. 7:1–7:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.ECOOP.2022.7
23. Petrić, Z.: Coherence in substructural categories. Stud Logica **70**(2), 271–296 (2002). https://doi.org/10.1023/A:1015186718090
24. Petricek, T., Orchard, D.A., Mycroft, A.: Coeffects: a calculus of context-dependent computation. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014. pp. 123–135. ACM (2014). https://doi.org/10.1145/2628136.2628160
25. Raedt, L.D., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 2462–2467 (2007), http://ijcai.org/Proceedings/07/Papers/396.pdf

26. Rioux, N., Zdancewic, S.: Functional meaning for parallel streaming. Proc. ACM Program. Lang. **9**(PLDI), 1220–1244 (2025). https://doi.org/10.1145/3729299
27. Shaikhha, A., Huot, M., Smith, J., Olteanu, D.: Functional collection programming with semi-ring dictionaries. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–33 (2022). https://doi.org/10.1145/3527333
28. Smirnov, A.L.: Graded monads and rings of polynomials. Journal of Mathematical Sciences **151**, 3032–3051 (6 2008). https://doi.org/10.1007/s10958-008-9013-7, translated from Zapiski Nauchnykh Seminarov POMI, Vol. 349, 2007, pp. 174–210
29. Somogyi, Z., Henderson, F., Conway, T.C.: The execution algorithm of mercury, an efficient purely declarative logic programming language. J. Log. Program. **29**(1-3), 17–64 (1996). https://doi.org/10.1016/S0743-1066(96)00068-4