23

Towards Squiggly Refinement Algebra

D. A. Naumann
Stevens Institute of Technology
Hoboken, NJ 07030 USA. E-mail: naumann@cs.stevens-tech.edu

Abstract

The algebra of functions and relations has been used so successfully in program construction that textbooks have appeared. Despite the importance of predicate transformers in imperative programming, the algebra of transformers has been less explored. To show its promise, we prove results on exponents and recursion on inductive data types, sufficient for carrying out a polytypic derivation that has been given as a substantial example for functions and relations. We also give a data refinement from exponents of specifications to the concrete exponents needed for program semantics.

Keywords

Specification and verification, abstraction and refinement, polytypism, categorical foundations, predicate transformers, data refinement

1 INTRODUCTION

The program-level algebra of functions and relations has proved quite useful in the derivation of functional programs, and has been explored to the extent that there are now textbooks on the subject (Bird & de Moor 1996). In a particularly interesting exercise, de Moor (1996) uses the algebra of functions and relations to derive a solution to the repmin problem. The repmin problem is to replace each node of a tree of numbers with the minimum value in the tree, and to do so in a single pass over the tree. The derivation is *polytypic*, meaning that it is parameterized not just by the data type (numbers could be replaced by any suitably ordered type) but by the type-constructor 'tree-of': any inductive type-constructor will do. More striking is de Moor's observation that the derivation is also paradigm-parameterized in that the resulting program can be interpreted in two different categories, giving quite different programs: a higher-order functional program and a first-order logic program. Just as striking is the absence of an imperative interpretation. Our contribution is to justify such an interpretation, and more broadly to argue that the algebra of predicate transformers may be of practical use. In so arguing, we use a higher-order data

Programming Concepts and Methods D. Gries, & W-P. de Roever (Eds.)

refinement, i.e. refinement of a constructor rather than a primitive type, which is of independent interest.

The reader is supposed to be familiar with constructive functional programming in the categorical style – known as Squiggol – pioneered by Meertens, Backhouse et al. (Bird & de Moor 1996) and with the refinement calculus of Back, Morgan et al. (Morgan 1994). Refinement calculus and Squiggol are both algebraic in the sense that programs are developed from specifications by (in)equational calculations, in a setting where specifications are viewed as abstract programs. Derivations in refinement calculus involve heavy use of state predicates and program expressions involving state variables. By contrast, derivations in Squiggol are entirely at the level of programs, which facilitates concision and generality. But refinement calculus deals with imperative programming, for which predicate-transformer semantics is an accurate model.

There is an important connection between predicate transformers and the algebra of functions and relations. Every relation can be factored as a composite $(f^o \,;\, g)$ of a function g with the *reciprocal* (relative converse) f^o of a function f. This allows the category **Rel** of relations to be constructed from the category **Fun** of functions in such a way that type-structure can be lifted from functions to relations (Bird & de Moor 1996). Refinement calculus is based on the category of what we shall call *powerset transformers*, i.e. monotonic functions $f : \wp A \to \wp B$ between powersets. Every powerset transformer can be factored as a composite $([R] \,;\, \langle S \rangle)$ of the direct-image $\langle S \rangle$ and inverse-image $[R]$ of relations R, S. This allows the type-structure of **Rel** to be lifted to powerset transformers (Gardiner, Martin & de Moor 1994).

Lifting from **Fun** to **Rel** can yield weaker laws; e.g. Cartesian products are not categorical products in **Rel**. Lifting from **Rel** to powerset transformers leads to even weaker laws, as it should in an accurate model of imperative programs. Consider, e.g. the program $A \xrightarrow{f \vartriangle g} B \times C \xrightarrow{\pi} B$ which forms a pair by executing programs f and g and then projects out just the first component of the pair. It is not necessarily equal to f, because g could diverge. The refinement $(f \vartriangle g) \,;\, \pi \sqsubseteq f$ does hold, but only if g obeys the law of the 'excluded miracle'. Deterministic programs satisfy the other fundamental law of categorical products, $h \,;\, \pi_0 \vartriangle h \,;\, \pi_1 = h$, which says that for pair-producing h the components of an output pair can be constructed separately and combined to yield the same pair. But this weakens to an inequality if h is nondeterministic, because two independent executions of h may have different outcomes. Fortunately, a small number of conditional inequalities, generalizing the familiar universal properties, sufficies to axiomatize the weak product for powerset transformers (see Proposition 5).

Coproducts do lift to categorical coproducts of powerset transformers, and de Moor (1992) shows that initial algebras (inductive data types) also lift without weakening. Martin (1995) shows that some of the 'fusion' laws of Squiggol lift in weakened form to powerset transformers. Morgan (1994) uses (monomorphic) inductive data types and Martin (1995) spells out the interpretation of map-like operations and fusion laws in the notation of refinement-calculus, but derivations at the program level

are not given. This author was surprised to succeed at a nontrivial derivation in a category that is only slightly different from powerset transformers.

The repmin derivation uses initial algebras and exponents. The categorical notion of *exponent* internalizes the homset $\mathbf{Fun}(B, C)$ as an object $B \to C$ axiomatized in terms of Currying, i.e. the bijection between $\mathbf{Fun}(A \times B, C)$ and $\mathbf{Fun}(A, B \to C)$. Exponents are usually internalized homsets, but not always. De Moor's logic-programming interpretation of repmin is based on the bijection between $\mathbf{Rel}(A \times B, C)$ and $\mathbf{Rel}(A, B \times C)$ which has the formal properties of an exponent but corresponds to shuffling inputs and outputs in first-order logic programs. Function-spaces lift from **Fun** to **Rel** (and to powerset transformers), yielding a different (weak) exponent that is not the homset of **Rel** (nor of transformers). But we want an imperative interpretation of repmin as a higher-order program. If programs denote transformers, then a data type of programs should be a set of transformers, i.e. a homset of the category of transformers. Homsets of powerset transformers do carry an exponent structure but in quite weak form; by contrast with lifted products there is no simple axiomatization (Naumann 1995*b*). The evaluation rule for exponents, used in the repmin derivation, fails to hold in general. That derivation also uses a recursion theorem which is proved using exponents, and for powerset transformers we do not get a unique solution to the recursion equation (Naumann 1994*b*). Worse yet, data refinement is not preserved by the weak exponent. So much for powerset transformers.

A better-behaved category of predicate transformers (just *transformers* henceforth) can be obtained by changing the base category from **Fun** to the ordered category **MFun** of monotonic functions between preordered sets. As is explained in §2, order-absorbing relations factor into pairs of monotonic functions, and transformers on ordered data types factor into pairs of such relations. Such transformers act on predicates φ that are monotonic in the sense that x' satisfies φ whenever $x' \succeq x$ for x satisfying φ. This is certainly true of predicates used in specification, if $x' \succeq x$ means that x approximates x'. Because predicates on the exponent are restricted to be compatible with the refinement order – i.e. internalization of the homset takes its ordering into account – the properties of the weak exponent are improved. There is a complete axiomatization (Naumann 1996) (usually we shall omit the qualifier 'weak'), and data refinement is sound (Naumann 1995*a*). Transformers give an accurate semantics for a conventional higher order imperative language with specifications as abstract programs and higher types modeled by the weak exponent (Naumann 1994*a*).

This paper's first contribution is to show that inductive data types lift to transformers, and to prove the recursion theorem needed for the repmin derivation. Aside from problem-specific facts, the only other ingredient needed for repmin is a 'banana split' law that expresses a pair of recursions in terms of a single one (loop fusion). The general result weakens somewhat for transformers, but one gets a conditional law that is adequate for repmin. A companion paper (Naumann 1998) uses the present results to prove some general lemmas that are then used along with problem-specific results to derive the repmin program: de Moor's polytypic derivation can indeed be interpreted in an imperative setting.

This paper's other contribution deals with the connection between the model and concrete programs. Because the category of transformers is rich enough to interpret infeasible constructs including pre–post specifications as abstract programs, the homset is not appropriate as a concrete type. So we give a data refinement from that abstract exponent to an exponent of healthy transformers, those having a well-understood connection with operational semantics of conventional languages. This step can always be taken, which means the issue can be ignored in practice: Programs can be derived by calculation using the exponent in the category of specifications, and the final program can be interpreted concretely.

The paper is organized as follows: §2 reviews basic definitions for transformers and order-absorbing relations called ideals; §3 shows that initial algebras for ideals are final coalgebras; §4 reviews the weak product and weak exponent for transformers; §5 lifts initial algebras and proves the recursion theorem; §6 briefly sketches the repmin program; §7 discusses data refinement and defines the concrete exponent; §8 gives the data refinement between exponents; §9 discusses the results.

2 BACKGROUND ON PROSETS, PROCATS, AND IDEAL RELATIONS

This section defines the three categories **MFun**, **IRel**, and **Spec** along with quite a number related notions. Complete definitions and proofs appear in Naumann (1996); many of the ideas are based on Gardiner et al. (1994).

First, some basics. Application of function f to argument x is written fx, and associates to the left. Composition of relation $R : A \to B$ with $S : B \to C$ is denoted by $(R\,;S)$ even in the special case that R and S are functions. Composition binds less tightly than application but more tightly than other binary operators like $\Delta, \otimes, \rightsquigarrow$ introduced later. The pairing comma binds very weakly, e.g. $(x, y \otimes z)$ means $(x, (y \otimes z))$.

Data types (state spaces) are taken to be *prosets*, i.e. sets A equipped with a preorder relation $\preceq_A$. The intended interpretation of $x \preceq y$ is that x approximates y or y refines x. No further properties of $\preceq$ are required; the induced order for transformers suffices for semantics of recursion, and data types in specifications need not be restricted to CPOs. In particular, sets ordered discretely (i.e. by equality) are allowed; it is only for exponents that nontrivial order is needed. Predicates on data type A are taken to be *updeals*, i.e. subsets φ of A that are upward closed: $x \in \varphi \wedge x \preceq y \;\Rightarrow\; y \in \varphi$. (This is called monotonicity in §1.) The proset of updeals in A, ordered by $\subseteq$, is denoted by $\mathcal{U}A$. It is a complete lattice. Note that $\mathcal{U}A$ is the powerset $\wp A$ if and only if A is ordered discretely.

The repmin problem involves nonempty trees, for example defined by

$$Ltree\ B = Leaf\ B \mid Bin(Ltree\ B, Ltree\ B)$$

With B instantiated to be the naturals, the appropriate approximation ordering on

trees is equality. The repmin program involves minimum values with respect to numerical magnitude, not approximation – and the same is true if, for example, B is a type of extensible records, for which the record-extension order induces a nontrivial approximation order on trees. The repmin problem is parameteric not just in B but in the constructor $Ltree$; polytypism is addressed in §3.

A *procat* is a category $\mathcal{C}$ with each homset $\mathcal{C}(A, B)$ preordered and composition monotonic. The procat **MFun** has prosets as objects and monotonic functions as arrows, ordered pointwise. The procat **Rel** has binary relations as arrows, ordered by $\subseteq$. An arrow $m : A \to B$ in a procat is a *map* just if there is an arrow $c : B \to A$, called its *comap*, such that $\mathsf{id}_A \preceq (m \,;\, c)$ and $(c \,;\, m) \preceq \mathsf{id}_B$. The maps of **Rel** are the total functions, and comaps are reciprocals of functions. We write R° for the *reciprocal* or relative converse of any relation R.

A *transformer* is a monotonic function $\mathcal{U}A \to \mathcal{U}B$ for prosets A, B. To facilitate comparison between the algebra of relations and the algebra of transformers, we define a procat **Spec** with arrows the other way around. For prosets A, B, define $\mathbf{Spec}(A, B)$ to be the set of transformers $\mathcal{U}B \to \mathcal{U}A$, ordered pointwise (written $\sqsubseteq$). In short, $\mathbf{Spec}(A, B) = \mathbf{MFun}(\mathcal{U}B, \mathcal{U}A)$. We write the symbol $\mathbin{;^\circ}$ for composition in **Spec**, so $(f \mathbin{;^\circ} g)$ in **Spec** is the composite $(g \,;\, f)$ of functions. A powerset transformer is just a transformer on discretely ordered data types.

As mentioned in §1, every powerset transformer can be written in the form $\langle R\rangle \mathbin{;^\circ} [S]$ where $\langle R\rangle$ is the direct-image function for some relation R and $[S]$ is the inverse-image function for some S. This factorization extends to transformers on non-discrete prosets, but the relations involved must be compatible with order in the following sense. If A, B are prosets, a relation $R : A \to B$ is an *ideal* if $(\preceq \,;\, R \,;\, \preceq) = R$. At the level of points this means $a' \preceq a \wedge aRb \wedge b \preceq b' \;\Rightarrow\; a'Rb'$. Ideals are just a mathematical tool for intermediate purposes: If one tries to interpret the ideal property for relations as a model of nondeterministic state transformers then there is clear justification for $(R \,;\, \preceq) = R$ and no justification for $(\preceq \,;\, R) = R$. Later we explain how the ideal property makes sense for pre–post specifications.

Define the category **IRel** of ideal relations to have prosets for objects and ideals for arrows, with homsets ordered by $\subseteq$. Composition is the same as in **Rel**, but the identity arrow on A is the relation $\preceq_A$ (and we refrain from writing id for that).

Whereas **Fun** is a subcategory of **Rel**, **MFun** is not a subcategory of **IRel** because a monotonic function $f : A \to B$ is not an ideal in general. But $(f \,;\, \preceq)$ is an ideal, and the maps of **IRel** are exactly the ideals of this form. For any f in **MFun**, the comap of $(f \,;\, \preceq)$ is the ideal $(\preceq \,;\, f^\circ)$. Every ideal factors as $(\preceq \,;\, f^\circ \,;\, g \,;\, \preceq)$ for some f, g in **MFun**. Monotonic functors on **IRel** preserve maps.

For ideals $S : A \to C$ and $R : B \to C$ we write S/R for the so-called *quotient* such that $Q \subseteq S/R \equiv Q \,;\, R \subseteq S$ for all Q. At the level of points, quotients in **IRel** are the same as in **Rel**. We use only one property of quotients, namely $Q \,;\, R/S = (Q \,;\, R)/S$ for all maps Q (we let / bind more tightly than ;).

The direct-image function of R in $\mathbf{IRel}(A, B)$ is a transformer $\langle R\rangle : \mathcal{U}A \to \mathcal{U}B$, i.e. an element of $\mathbf{Spec}(B, A)$. The inverse-image $[R]$ is in $\mathbf{Spec}(A, B)$. Direct-images (resp. inverse-images) are universally disjunctive (resp. conjunctive). The

maps of **Spec** are the transformers of the form $[R]$, and comaps have the form $\langle R\rangle$. Readers familiar with junctivity properties of maps (left adjoints) may find this confusing. It may help to keep in mind that **Spec** is the arrow-dual of a category of functions and that, roughly speaking, maps $[R]$ correspond to weakest-precondition transformers for programs just like maps of **Rel** correspond to functional programs.

Several other potentially-confusing complications are the consequence of objects being ordered. For proset A, we write $\widetilde{A}$ for the order-dual proset: $x \preceq_A y \equiv y \preceq_{\widetilde{A}} x$. For R in $\mathbf{IRel}(A, B)$, the reciprocal R^o does not have the ideal property in general; rather, it is an ideal with respect to the order-dual prosets. We define the *op-reciprocal* $R^\ominus$ to be R^o construed as an element of $\mathbf{IRel}(\widetilde{B}, \widetilde{A})$. (Let $^\ominus$ bind more tightly than function application). Op-reciprocation is involutive and order-contravariant. It is also arrow-contravariant: $(R\,;S)^\ominus = S^\ominus\,;R^\ominus$ and $(\preceq_B)^\ominus = \preceq_{\widetilde{B}}$.

For R in $\mathbf{Rel}(A, B)$ there is function $\Lambda R : A \to \wp B$ sending a to the image of a through R. The direct image function $\langle R\rangle : \wp A \to \wp B$ can be defined by $\langle R\rangle = \Lambda(\ni ; R)$ where $\ni$ is the reciprocal of the membership relation $(\in) : A \to \wp A$. The structure $\wp$, Λ, and $\ni$ in **Rel** is characterized by the *power adjunction* law

$$f = \Lambda R \equiv f\,;\ni = R \quad \text{for all relations } R \text{ and functions } f \tag{1}$$

(of appropriate type) (Bird & de Moor 1996). This implies naturality properties

$$\Lambda R\,;\langle S\rangle = \Lambda(R\,;S) \quad \text{and} \quad \Lambda(f\,;R) = f\,;\Lambda R \quad . \tag{2}$$

Updeal lattices can be described by the power adjunction on **IRel**, which facilitates calculational proofs about transformers, but there are complications due to order.

We want to axiomatize the membership relation $\ni : \mathcal{U}A \to A$ restricted to updeals, but $\ni$ does not have the ideal property with respect to the order $\subseteq$ on $\mathcal{U}A$. It is an ideal of type $\widetilde{\mathcal{U}A} \to A$ (recall that $\widetilde{\mathcal{U}A}$ is the set of updeals ordered by $\supseteq$). If R is in $\mathbf{IRel}(A, B)$ and $a \in A$ then $\Lambda R a$ is an updeal, and $a \preceq a'$ implies $\Lambda R a \supseteq \Lambda R a'$ (these are precisely the properties $R\,;\preceq = R$ and $\preceq\,;R = R$). So we can take ΛR to be in $\mathbf{MFun}(A, \widetilde{\mathcal{U}B})$, and (1) holds for ideals R and monotonic functions f.

Just as the power adjunction is used to define $\langle R\rangle = \Lambda(\ni\,;R)$, it is also used to define $[R] = \Lambda(\ni/R)$. Strictly speaking these formulations are type-incorrect; e.g. $\Lambda(\ni/R)$ has type $\widetilde{\mathcal{U}B} \to \widetilde{\mathcal{U}A}$ for R of type $A \to B$, whereas we shall take $[R]$ to be $\mathcal{U}B \to \mathcal{U}A$, so that $\langle R\rangle$ is in $\mathbf{Spec}(B, A)$ and $[R]$ is in $\mathbf{Spec}(A, B)$. Precise formulations require that $\widetilde{\ }$ be extended to a functor on **MFun**, giving for f in $\mathbf{MFun}(A, B)$ the same mapping considered as a function $\widetilde{f}$ in $\mathbf{MFun}(\widetilde{A}, \widetilde{B})$, but we omit that. We also write $[f]$ for $[f\,;\preceq]$ and $\langle f^o\rangle$ for $\langle \preceq\,;f^o\rangle$. In the case $f = \mathrm{id}$ we are more careful, e.g. writing $[\preceq] = \mathrm{id}$ and $[R\,;S] = [R]\,\mathring{;}\,[S]$ to express functoriality of $[-]$.

Lemma 1 *(a) For any ideal R, $\langle R\rangle$ is a comap in* **Spec** *and $[R]$ is a map. (b) If f is in* **MFun** *then $[f]$ is a bimap in* **Spec** *and $[f] = \langle f^o\rangle$.*

Every transformer in $\mathbf{Spec}(A, B)$ factors as $\langle R \rangle \fatsemi [S]$ for some proset C and ideals R in $\mathbf{IRel}(C, A)$ and S in $\mathbf{IRel}(C, B)$. In terms of specifications, R is a precondition and S a postcondition; C is an index set or specification variable linking initial and final states. The ideal properties $R \fatsemi \preceq \;=\; R$ and $S \fatsemi \preceq \;=\; S$ say that the predicates are updeals (monotonic) in their right argument. The ideal properties $\preceq \fatsemi R = R$ and $\preceq \fatsemi S = S$ are hard to justify, except that for specifications one may always choose C to be ordered by equality, making the properties vacuous.

3 INDUCTIVE DATA TYPES FOR IDEAL RELATIONS

The Cartesian product, disjoint sum, and function-space constructs lift from **MFun** to **IRel** in much the same way as they do from **Fun** to **Rel** (Bird & de Moor 1996, Naumann 1996). This section shows that inductive data types also lift.

A functor $F : \mathcal{C}^2 \to \mathcal{C}$ *has fixpoints* if for each object B there is an object $\mathsf{fix}B$ and arrow $\mathsf{in}B : F(B, \mathsf{fix}B) \to \mathsf{fix}B$ that is *initial* for B in the following sense. For each A and $R : F(B, A) \to A$ there is $(\!|R|\!) : \mathsf{fix}B \to A$ such that for all S

$$S = (\!|R|\!) \;\equiv\; \mathsf{in}B \fatsemi S = F(\preceq, S) \fatsemi R \qquad \begin{array}{ccc} F(B, \mathsf{fix}B) & \xrightarrow{\mathsf{in}B} & \mathsf{fix}B \\ {\scriptstyle F(\preceq, S)}\downarrow & & \downarrow{\scriptstyle S} \\ F(B, A) & \xrightarrow[R]{} & A \end{array} \tag{3}$$

We call $(\!|R|\!)$ a *catamorphism*. Dependence on F is suppressed in the notation. Note that $\mathsf{fix}B$ is not the fixpoint of B but rather of the functor $F(B, -)$.

As an example, for each B the type $Ltree\, B$ is the least fixpoint of the functor $F(B, -)$ with F defined by $F(B, A) = B + (A \times A)$. From a function $min : F(\mathbb{N}, \mathbb{N}) \to \mathbb{N}$ giving the numerical minimum of one or two naturals we get function $(\!|min|\!) : Ltree\, \mathbb{N} \to \mathbb{N}$ which gives the numerical minimum of a tree.

If F is a monotonic functor and its restriction to **Fun** (i.e. the maps of **Rel**) has an initial algebra then that algebra is initial in **Rel**. This can be proved using power adjunction without reciprocation (Bird & de Moor 1996). The same proof shows that fixpoints lift from **MFun** to **IRel**, which we state as follows.

Proposition 2 *If F is a monotonic functor $\mathbf{IRel}^2 \to \mathbf{IRel}$, and it has fixpoints in the subcategory of maps, then it has fixpoints in* **IRel**.

It is well known that initial algebras for relations are final coalgebras. The corresponding result for ideals is slightly more complicated. We say a functor F on **IRel** *commutes with op-reciprocation* if $F\widetilde{B} = (FB)^{\sim}$ and $FR^{\Theta} = (FR)^{\Theta}$. The result below is that if F commutes with op-reciprocation then an initial algebra $\mathsf{in}\widetilde{B}$ for $\widetilde{B}$

gives a *final coalgebra* $(\mathsf{in}\widetilde{B})^{\ominus}$ for B. That is, for each A and $R : A \to F(B, A)$ there is a so-called *anamorphism* $[\![R]\!] : A \to (\mathsf{fix}\widetilde{B})^{\sim}$ with

$$S = [\![R]\!] \;\equiv\; S\,;(\mathsf{in}\widetilde{B})^{\ominus} = R\,;F(\preceq, S) \qquad \text{for all } S. \tag{4}$$

Note that the final coalgebra for B is the op-reciprocal of the initial algebra $\mathsf{in}\widetilde{B} : F(\widetilde{B}, \mathsf{fix}\widetilde{B}) \to \mathsf{fix}\widetilde{B}$ for $\widetilde{B}$. The type of $(\mathsf{in}\widetilde{B})^{\ominus}$ is $(\mathsf{fix}\widetilde{B})^{\sim} \to (F(\widetilde{B}, \mathsf{fix}\widetilde{B}))^{\sim}$, but if F commutes with op-reciprocation then we get the type shown in this diagram:

$$\begin{array}{ccc} A & \xrightarrow{\quad R \quad} & F(B, A) \\ {\scriptstyle [\![R]\!]}\downarrow & & \downarrow{\scriptstyle F(\preceq, [\![R]\!])} \\ (\mathsf{fix}\widetilde{B})^{\sim} & \xrightarrow[(\mathsf{in}\widetilde{B})^{\ominus}]{} & F(B, (\mathsf{fix}\widetilde{B})^{\sim}) \end{array}$$

The types can be further simplified by using initiality to show $(\mathsf{fix}\widetilde{B})^{\sim} = \mathsf{fix}B$.

Proposition 3 *The following functors on* **IRel** *commute with op-reciprocation:* $+$, $\times$, id, *the projections* $\mathbf{IRel}^2 \to \mathbf{IRel}$, *and those constant-valued functors whose value is an equivalence relation.*

Proof. For the functor constantly B, we have $x \preceq_{F\widetilde{A}} y \equiv x \preceq_B y$ and $x \preceq_{(FA)^{\ominus}} y \equiv y \preceq_B x$, hence $\preceq_B$ needs to be symmetric. The other cases are easy. □

Theorem 4 *Suppose* $F : \mathbf{IRel}^2 \to \mathbf{IRel}$ *has fixpoints and commutes with op-reciprocation. Then for each* B *the functor* $F(B, -)$ *has a final coalgebra.*

Proof. Given any A, B and $R : A \to F(B, A)$, define $[\![R]\!] = (\!|R^{\ominus}|\!)^{\ominus}$. If F commutes with op-reciprocation we have $R^{\ominus} : F(\widetilde{B}, \widetilde{A}) \to \widetilde{A}$, whence $(\!|R^{\ominus}|\!) : \mathsf{fix}\widetilde{B} \to \widetilde{A}$ and $[\![R]\!] : A \to \mathsf{fix}B$ as explained above. For any S,

$$\begin{array}{lll} & S = [\![R]\!] & \\ \equiv & S = (\!|R^{\ominus}|\!)^{\ominus} & \text{definition of } [\![-]\!] \\ \equiv & S^{\ominus} = (\!|R^{\ominus}|\!) & {}^{\ominus} \text{ involution} \\ \equiv & \mathsf{in}\widetilde{B}\,;S^{\ominus} = F(\preceq, S^{\ominus})\,;R^{\ominus} & \text{catamorphism (3) } B, S, R := \widetilde{B}, S^{\ominus}, R^{\ominus} \\ \equiv & S\,;(\mathsf{in}\widetilde{B})^{\ominus} = R\,;(F(\preceq, S^{\ominus}))^{\ominus} & {}^{\ominus} \text{ arrow-contravariant involution} \\ \equiv & S\,;(\mathsf{in}\widetilde{B})^{\ominus} = R\,;F(\preceq, S) & F \text{ commutes with } {}^{\ominus},\ {}^{\ominus} \text{ involution} \end{array}$$

which proves (4). □

4 WEAK PRODUCTS AND WEAK EXPONENTS IN Spec

Results in this section are proved in Gardiner et al. (1994) or Naumann (1996).

An *upfunctor* F on **Spec** is a monotonic graph morphism that preserves identities and satisfies the inequation $F(g \fatsemi h) \sqsubseteq Fg \fatsemi Fh$. Let $F : \mathbf{IRel} \to \mathbf{IRel}$ be a monotonic functor. There is a minimum upfunctor $\widehat{F}$ on **Spec** that agrees with F on ideals, given by $\widehat{F}A = FA$ and $\widehat{F}g = [FR] \fatsemi \langle FS \rangle$ where R, S factor g as $g = [R] \fatsemi \langle S \rangle$. (Similarly for bifunctors.) Agreeing with ideals means $\widehat{F}[R] = [FR]$. Moreover, $\widehat{F}$ satisfies

$$\widehat{F}(m \fatsemi g \fatsemi c) = \widehat{F}m \fatsemi \widehat{F}g \fatsemi \widehat{F}c \quad \text{if } m \text{ is a map and } c \text{ a comap.} \tag{5}$$

The lifted coproduct $\widehat{+}$ is monotonic and is a categorical coproduct, but the lifted product, which we write as $\otimes$, weakens as discussed in §1.

Define $A \otimes B$ to be the product $A \times B$ of prosets. The left projection is $[\pi_0] \in \mathbf{Spec}(A \otimes B, A)$ where π_0 is the projection function. For $f \in \mathbf{Spec}(D, A)$ and $g \in \mathbf{Spec}(D, B)$, the 'pairing' $(f \vartriangle g) \in \mathbf{Spec}(D, A \otimes B)$ generalizes the pairing of functions, but we do not need the explicit definition. As usual, we define $f \otimes g = [\pi_0] \fatsemi f \vartriangle [\pi_1] \fatsemi g$.

We say $f \in \mathbf{Spec}(A, B)$ is *strict* if $f\emptyset = \emptyset$ and *costrict* if $fB = A$. Comaps are strict.

Proposition 5 *For all transformers d, e, f, g, h of suitable types we have*

$$\begin{aligned}
(f \vartriangle g) \fatsemi [\pi_0] &\sqsubseteq f \quad \text{if } g \text{ is strict} && (6)\\
(f \vartriangle g) \fatsemi [\pi_0] &\sqsupseteq f \quad \text{if } g \text{ is costrict} &&\\
h \fatsemi [\pi_0] \vartriangle h \fatsemi [\pi_1] &\sqsupseteq h \quad \text{if } h \text{ is a comap (and } \sqsubseteq \text{ if } h \text{ is a map)} &&\\
h \fatsemi f \vartriangle h \fatsemi g &\sqsupseteq h \fatsemi (f \vartriangle g) \quad \text{if } h \text{ is a comap} && (7)\\
h \fatsemi f \vartriangle h \fatsemi g &\sqsubseteq h \fatsemi (f \vartriangle g) \quad \text{if } h \text{ is a map} &&\\
d \fatsemi f \vartriangle e \fatsemi g &\sqsubseteq (d \vartriangle e) \fatsemi (f \otimes g) && (8)\\
d \fatsemi f \vartriangle e \fatsemi g &= (d \vartriangle e) \fatsemi (f \otimes g) \quad \text{if } f, g \text{ are maps or all are comaps} &&
\end{aligned}$$

Moreover, $\vartriangle, \otimes$ preserve maps and comaps. Such a structure is is unique up to natural isomorphism.

Define the proset $B \rightsquigarrow C$ to be $\mathbf{Spec}(B, C)$ ordered by $\sqsubseteq$. This is not (in any direct sense) a lifting from **IRel**. 'Application' is an arrow $\mathsf{ap}_{B,C}$ in $\mathbf{Spec}((B \rightsquigarrow C) \otimes B\ ,\ C)$; operationally, it executes a stored program from a given initial state. We do not need the definition of ap, just its properties, and the same for 'Currying'. For f in $\mathbf{Spec}(A \otimes B, C)$, $\mathsf{cur} f$ is in $\mathbf{Spec}(A, B \rightsquigarrow C)$. For h in $\mathbf{Spec}(C, A)$, define $B \rightsquigarrow h = \mathsf{cur}(\mathsf{ap} \fatsemi h)$, so that $B \rightsquigarrow h$ is in $\mathbf{Spec}((B \rightsquigarrow C), (B \rightsquigarrow A))$.

Proposition 6 *For any f, $\mathsf{cur} f$ is a bimap. Also, $(B \rightsquigarrow) : \mathbf{Spec} \to \mathbf{Spec}$ is a*

monotonic functor and for all f, g of suitable types we have

$$
\begin{array}{rcll}
(\mathsf{cur} f \otimes \mathsf{id}) \fatsemi \mathsf{ap} & = & f & (9) \\
\mathsf{cur}((f \otimes \mathsf{id}) \fatsemi \mathsf{ap}) & \sqsubseteq & f \quad \textit{if } f \textit{ is a map} & (10) \\
\mathsf{cur}((f \otimes \mathsf{id}) \fatsemi \mathsf{ap}) & = & f \quad \textit{if } f \textit{ is a bimap} & (11) \\
\mathsf{cur}(g \fatsemi f) & = & g \fatsemi (B \rightsquigarrow f) & \\
\mathsf{cur}((f \otimes \mathsf{id}) \fatsemi g) & \sqsubseteq & f \fatsemi \mathsf{cur}\, g \quad \textit{if } f \textit{ is a map} & \\
\mathsf{cur}((f \otimes \mathsf{id}) \fatsemi g) & = & f \fatsemi \mathsf{cur}\, g \quad \textit{if } f \textit{ is a bimap} & (12)
\end{array}
$$

Such a structure is unique up to natural isomorphism.

Note that not even (10) holds for all transformers, but the evaluation rule (9) does: Currying a program and then executing it is the same as just executing it, even for nondeterministic and possibly-divergent programs.

5 INDUCTIVE DATA TYPES IN Spec

After the lifting of products and coproducts was well understood, the problem of lifting inductive data types to transformers remained open for some time. The solution found for powerset transformers by de Moor (1992) works here as well: Initial algebras in **Spec** are obtained from initial algebras in **MFun** by applying the inverse image operator $[-]$ and using the final coalgebra property in **IRel**. There are minor differences in the proof due to the absence of reciprocation and the presence of order. In this section we assume that $F : \mathbf{IRel}^2 \to \mathbf{IRel}$ is a monotonic functor that has fixpoints and commutes with op-reciprocation.

Theorem 7 *The upfunctor $\widehat{F} : \mathbf{Spec}^2 \to \mathbf{Spec}$ has fixpoints.*

Proof. The initial algebra for B is the inverse image $[\mathsf{in}\widetilde{B}]$ of the initial algebra for $\widetilde{B}$ in **IRel**. Being an initial algebra, $\mathsf{in}\widetilde{B}$ is an isomorphism, hence a map in **IRel**. Thus by Lemma 1(b) we have $[\mathsf{in}\widetilde{B}] = \langle(\mathsf{in}\widetilde{B})^o\rangle$, which exhibits the initial algebra for B in **Spec** as the direct image of the final coalgebra for B in **IRel** (by Theorem 4). Let g be in $\mathbf{Spec}(F(B, A), A)$. We will derive the definition of catamorphism $(\!|g|\!)$ in terms of an anamorphism in **IRel**. We need to show for any h in $\mathbf{Spec}(\mathsf{fix}B, A)$

$$
h = (\!|g|\!) \;\equiv\; [\mathsf{in}\widetilde{B}] \fatsemi h = \widehat{F}(\mathsf{id}, h) \fatsemi g \qquad
\begin{array}{ccc}
F(B, \mathsf{fix}B) & \xrightarrow{[\mathsf{in}\widetilde{B}]} & \mathsf{fix}B \\
{\scriptstyle \widehat{F}(\mathsf{id},h)}\downarrow & & \downarrow{\scriptstyle h} \\
F(B, A) & \xrightarrow[g]{} & A
\end{array}
\qquad (13)
$$

We calculate in **IRel**, so (13) takes the form

$$h = (\!|g|\!) \;\equiv\; h\,;[\mathsf{in}\widetilde{B}] = g\,;\widehat{F}(\mathsf{id},h) \qquad \begin{array}{ccc} \mathcal{U}(F(B,\mathsf{fix}B)) & \xleftarrow{[\mathsf{in}\widetilde{B}]} & \mathcal{U}(\mathsf{fix}B) \\ {\scriptstyle \widehat{F}(\mathsf{id},h)}\uparrow & & \uparrow{\scriptstyle h} \\ \mathcal{U}(F(B,A)) & \xleftarrow[g]{} & \mathcal{U}A \end{array} \tag{14}$$

Any h factors as $h = [\ni]\,;\langle h\,;\ni\rangle$, and id factors as $[\preceq]\,;\langle\preceq\rangle$. Thus we have

$$\begin{array}{lll} & g\,;\widehat{F}(\mathsf{id},h) & \\ = & g\,;[F(\preceq,\ni)]\,;\langle F(\preceq,(h\,;\ni))\rangle & \text{factorization, definition of } \widehat{F} \\ = & g\,;\Lambda(\ni/F(\preceq,\ni))\,;\langle F(\preceq,(h\,;\ni))\rangle & \text{definition of } [-] \\ = & \Lambda(g\,;\ni/F(\preceq,\ni)\,;F(\preceq,(h\,;\ni))) & g \text{ function, power calculus (2)} \\ = & \Lambda((g\,;\ni)/F(\preceq,\ni)\,;F(\preceq,(h\,;\ni))) & g \text{ function, quotient property} \end{array}$$

We also have

$$\begin{array}{lll} & h\,;[\mathsf{in}\widetilde{B}] & \\ = & h\,;\langle(\mathsf{in}\widetilde{B})^{o}\rangle & \mathsf{in}\widetilde{B} \text{ map in } \mathbf{IRel}, \text{ Lemma 1(b)} \\ = & h\,;\Lambda(\ni\,;(\mathsf{in}\widetilde{B})^{\ominus}) & \text{definitions of } \langle-\rangle \text{ and } {}^{\ominus} \\ = & \Lambda(h\,;\ni\,;(\mathsf{in}\widetilde{B})^{\ominus}) & h \text{ function, power calculus (2)} \end{array}$$

Now (14) is shown by

$$\begin{array}{lll} & h\,;[\mathsf{in}\widetilde{B}] = g\,;\widehat{F}(\mathsf{id},h) & \\ \equiv & \Lambda(h\,;\ni\,;(\mathsf{in}\widetilde{B})^{\ominus}) = \Lambda((g\,;\ni)/F(\preceq,\ni)\,;F(\preceq,(h\,;\ni))) & \text{above} \\ \equiv & h\,;\ni\,;(\mathsf{in}\widetilde{B})^{\ominus} = (g\,;\ni)/F(\preceq,\ni)\,;F(\preceq,(h\,;\ni)) & \text{power calc. (1)} \\ \equiv & h\,;\ni = [\![(g\,;\ni)/F(\preceq,\ni)]\!] & \text{anamorphism (4)} \\ \equiv & h = \Lambda[\![(g\,;\ni)/F(\preceq,\ni)]\!] & \text{power calc. (1)} \end{array}$$

and we have found the definition $(\!|g|\!) = \Lambda[\![(g\,;\ni)/F(\preceq,\ni)]\!]$. □

Lemma 8 *If g is a bimap then so is $(\!|g|\!)$.*

The proof, by tedious manipulations using Lemma 1(b), is omitted.

On the face of it, catamorphisms embody only a very simple form of structural recursion. In combination with exponents, however, catamorphisms give recursion with an extra parameter. A rather general formulation is needed for repmin.

Theorem 9 *Suppose* $F : \mathbf{IRel}^2 \to \mathbf{IRel}$ *is a monotonic functor with fixpoints and* $G : \mathbf{Spec}^2 \to \mathbf{Spec}$ *is an upfunctor such that*

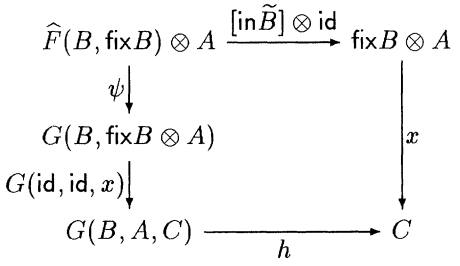
$$G(\mathsf{id}, \mathsf{id}, (f \fatsemi h)) = G(\mathsf{id}, \mathsf{id}, f) \fatsemi G(\mathsf{id}, \mathsf{id}, h) \quad \textit{for all } h \textit{ and all bimaps } f. \tag{15}$$

Suppose further that there is a family ψ *of arrows* $\psi_{A,B,C} : (\widehat{F}(A, B) \otimes C) \to G(A, C, B \otimes C)$ *with the following naturality property:*

$$\psi \fatsemi G(\mathsf{id}, \mathsf{id}, f \otimes \mathsf{id}) = (\widehat{F}(\mathsf{id}, f) \otimes \mathsf{id}) \fatsemi \psi \quad \textit{for all bimaps } f.$$

Then for any A, B, C, *and* h *in* $\mathbf{Spec}(G(B, A), C)$ *there is a unique* x *solving the equation* $([\mathsf{in}\widetilde{B}] \otimes \mathsf{id}) \fatsemi x = \psi \fatsemi G(\mathsf{id}, \mathsf{id}, x) \fatsemi h$.

The types in the following picture indicate how the result gives a parameterized form of the catamorphism recursion, and why ψ is needed to rearrange arguments.



The hypotheses for F, G and ψ are satisfied for functors lifted from **MFun** and **IRel** (recall (5)). For the repmin derivation the theorem is used twice, once with $G(B, A, C) = \widehat{F}(A, C)$ and once with $G(B, A, C) = \widehat{F}(B \otimes A, C)$. In both cases ψ is defined in terms of a so-called *distributor* $dist\ \widehat{F} : \widehat{F}B \times A \to \widehat{F}(B \times A)$, which determines a distributor $dist\ \mathsf{fix} : \mathsf{fix}B \times A \to \mathsf{fix}(B \times A)$ that pairs an element $a \in A$ with each node of a tree of Bs.

Proof. We calculate in **Spec**:

$$\begin{array}{lll}
 & ([\mathsf{in}\widetilde{B}] \otimes \mathsf{id}) \fatsemi x = \psi \fatsemi G(\mathsf{id}, \mathsf{id}, x) \fatsemi h & \\
\equiv & \mathsf{cur}(([\mathsf{in}\widetilde{B}] \otimes \mathsf{id}) \fatsemi x) = \mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, x) \fatsemi h) & \text{cur injective, by (9)} \\
\equiv & [\mathsf{in}\widetilde{B}] \fatsemi \mathsf{cur}\, x = \mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, x) \fatsemi h) & [\mathsf{in}\widetilde{B}] \text{ bimap, (12)} \\
\equiv & [\mathsf{in}\widetilde{B}] \fatsemi \mathsf{cur}\, x = \widehat{F}(\mathsf{id}, \mathsf{cur}\, x) \fatsemi k & \text{assumption for } k \text{ below} \\
\equiv & \mathsf{cur}\, x = (\!|k|\!) & \text{catamorphism (13)} \\
\Rightarrow & x = ((\!|k|\!) \otimes \mathsf{id}) \fatsemi \mathsf{ap} & \text{exponent (9)}
\end{array}$$

In the third step, we assume there is some $k : \widehat{F}(B, A \rightsquigarrow C) \to A \rightsquigarrow C$ such that $\mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, x) \fatsemi h) = \widehat{F}(\mathsf{id}, \mathsf{cur}\, x) \fatsemi k$. So it remains to discharge the assumption about k and show the reverse implication. We derive k as follows, using that $\widehat{F}$ and $\otimes$ preserve bimaps (from the theory of lifting).

$$
\begin{array}{lll}
 & \mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, x) \fatsemi h) & \\
= & \mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, (\mathsf{cur}\, x \otimes \mathsf{id}) \fatsemi \mathsf{ap}) \fatsemi h) & \text{exponent (9)} \\
= & \mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, \mathsf{cur}\, x \otimes \mathsf{id}) \fatsemi G(\mathsf{id}, \mathsf{id}, \mathsf{ap})) \fatsemi h) & \text{hypoth. (15), cur } x \text{ bimap} \\
= & \mathsf{cur}((\widehat{F}(\mathsf{id}, \mathsf{cur}\, x) \otimes \mathsf{id}) \fatsemi \psi \fatsemi G(\mathsf{id}, \mathsf{id}, \mathsf{ap}) \fatsemi h) & \text{cur } x \text{ bimap, } \psi \text{ natural} \\
= & \widehat{F}(\mathsf{id}, \mathsf{cur}\, x) \fatsemi \mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, \mathsf{ap}) \fatsemi h) & \text{exp. (12), cur } x \text{ bimap}
\end{array}
$$

We have derived $k = \mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, \mathsf{ap}) \fatsemi h)$. Because k is $\mathsf{cur}(\ldots)$, it is a bimap, hence so is $(\!|k|\!)$ (Lemma 8), which we use to show the reverse of the implication step in the first calculation.

$$
\begin{array}{lll}
 & x = ((\!|\mathsf{cur}(\psi \fatsemi G(\mathsf{id}, \mathsf{id}, \mathsf{ap}) \fatsemi h)|\!) \otimes \mathsf{id}) \fatsemi \mathsf{ap} & \\
\Rightarrow & \mathsf{cur}\, x = \mathsf{cur}((((\!|k|\!)) \otimes \mathsf{id}) \fatsemi \mathsf{ap}) & \text{Leibniz, definition of } k \\
\Rightarrow & \mathsf{cur}\, x = (\!|k|\!) & \text{exponent (11), } (\!|k|\!) \text{ bimap}
\end{array}
$$

□

For powerset transformers, the closed form derived above is only a least solution for x (Naumann 1994*b*). The difference is that k need not be a bimap in the powerset setting, so the proof does not go through as above.

6 APPLICATION TO THE REPMIN PROBLEM

The companion paper (Naumann 1998) develops problem-specific results and gives the repmin derivation in detail. Here we just sketch and assess the result.

Let us begin with an informal description in terms of functions. From the distributor $dist\,\mathsf{fix} : \mathsf{fix}B \times A \to \mathsf{fix}(B \times A)$ one obtains rep defined as the composite $(dist\,\mathsf{fix} \fatsemi \mathsf{fix}\,\pi_1) : \mathsf{fix}B \times A \to \mathsf{fix}A$ which replaces every node of a tree with a given value $a \in A$. The composite $(\mathsf{id} \vartriangle (\!|min|\!)) \fatsemi rep$ replaces every node of a tree with the minimum (according to min) of the tree. This definition for repmin involves two traversals because $dist\,\mathsf{fix}$ is defined from $dist\,F$ by catamorphism. But the recursion theorem can be used to derive a single-traversal program, which applies a Curried program. In terms of functional combinators, with F instantiated to the base for type $Ltree$, the result can be expressed in ML as follows.

```
datatype 'b tree = Leaf of 'b | Bin of ('b tree) * ('b tree);
fun repm (Leaf x)   = (x, Leaf)
 |  repm (Bin(u,v)) = let val (x,f) = repm u
                          val (y,g) = repm v
                      in ( min(x,y), fn w => Bin (f w, g w) )
                      end;
fun repmin t = let val (m,r) = repm t in r(m) end;
```

Function `repmin` works by constructing a pair consisting of the minimum `m` of the tree and a closure `r` which is then applied to `m`. Although the input is traversed just once, the closure is built from tree constructors in the same shape as the input.

The ML code is not polytypic, but polytypism has been implemented in functional languages (Jansson & Jeuring 1997). Polytypism aside, what is the imperative interpretation? Alas, the derived program is not directly expressible in conventional languages. It is the composite of transformers

$$Ltree\,\mathbb{N} \xrightarrow{(\!|h \vartriangle [min]|\!)} (\mathbb{N} \rightsquigarrow Ltree\,\mathbb{N}) \otimes \mathbb{N} \xrightarrow{\mathsf{ap}} Ltree\,\mathbb{N}$$

(where $h \vartriangle [min]$ corresponds to the body of `repm` above); but transformers model commands, and commands in conventional languages have the same final as initial state space.* Moreover, it is usually procedures rather than commands that can be stored or passed as parameters. Roughly speaking, however, the imperative interpretation is the same as the ML code above. The point is not that we derive an imperative program different from the functional one, but that it can be done in a setting where imperative constructs can be used in conjunction with $\rightsquigarrow$ and $(\!|-|\!)$.

As an aside, conventional imperative languages like C, C++, and Modula-3 do not allow closures to reference local variables like `f` and `g` above that are not in outermost scope, to simplify the runtime stack. In Naumann (1994*a*) this restriction helps avoid the much-studied problems of Algol, and constants (single-assignment variables) are used to express programs like that above. Variables in ML are single-assignment, but there are reference types for mutable cells.

It would be more striking to derive an essentially imperative repmin. For example, the following ML program constructs, in a single pass, a tree of references all to the same cell `m`. Program `repminr` initializes that cell, which is then updated to maintain the invariant that it is the minimum of the leaves that have been visited.

*In his talk at POPL 1998, John Reynolds remarked that it may be time to consider allowing commands that change the shape of their state space; at the 1996 MFPS workshop the author presented such a language, with stored commands modeled by the exponent of transformers.

```
datatype 'b tr = Leafr of 'b ref | Binr of ('b tr) * ('b tr);
val m = ref 0; (* arbitrary value *)
fun repmr (Leaf x) = ( m := min(x,!m); Leafr m  )
 |  repmr (Bin(u,v)) = let val t = repmr u
                           val s = repmr v
                       in Binr (t,s)
                       end;
fun repminr t = ( m := maxint; repmr t );
```

It is also possible to deal with trees of references without aliasing. The traversal can build a list of references to the nodes; that list is then traversed to set each leaf's cell to the minimum. The list could be stored as a singly linked list and accessed as a stack, resulting in better performance than the original program with two traversals. But such programs do not come from an interpretation of the exponent as a procedure type. Perhaps it is possible to treat heap storage and references as some form of exponent. Alternatively, one could use a data refinement transform the tree-structured closures into a concrete tree – but the functional program might be an adequate starting point for that.

Rather than explore either alternative, we turn to the more general question of whether the homset of transformers is a sensible model for stored procedures in program derivation.

7 DATA REFINEMENT AND THE CONCRETE EXPONENT

Let us assume there is some sub-procat **Prog** of **Spec** such that arrows in **Prog** correspond to concrete programs. For example, **Prog** could be the strict, positively conjunctive, continuous transformers. We assume **Prog** is closed under $\otimes$. So if f', g' are in **Prog** and $f \sqsubseteq f', g \sqsubseteq g'$, then $f' \Delta g'$ is a concrete refinement of $f \Delta g$. But for $f \in \mathbf{Spec}(A \otimes B, C)$ and f' in **Prog**, the refinement $f \sqsubseteq f'$ does not imply that $\mathsf{cur} f'$ is a concrete refinement of $\mathsf{cur} f$, because cur introduces the rather abstract type $B \rightsquigarrow C$. In this section we consider the exponent $B \rightsquigarrow' C = \mathbf{Prog}(B, C)$. We define the associated constructs ap' and cur' and we show how data refinement can be used to justify the following rule:

Suppose f is in $\mathbf{Prog}(A, B)$ for exponent-free A, B, and f' is f with all occurrences of $\mathsf{ap}, \mathsf{cur}, \rightsquigarrow$ replaced by their concrete counterparts $\mathsf{ap}', \mathsf{cur}', \rightsquigarrow'$. Then $f \sqsubseteq f'$. (16)

Section 8 gives the specific data refinement justifying the rule. The rule licenses us to ignore the distinction between $\rightsquigarrow$ and $\rightsquigarrow'$ in practice.

First, we need the definitions of ap and cur. For all $\psi \in \mathcal{U}C$, $b \in B$, and $g \in B \rightsquigarrow C$, define $(b, g) \in \mathsf{ap}\psi \equiv b \in g\psi$. For all $a \in A$ and $\varphi \in \mathcal{U}(B \rightsquigarrow C)$, define $a \in \mathsf{cur} f \varphi \equiv \mathsf{cu} f a \in \varphi$, where $\mathsf{cu} f a \in \mathbf{Spec}(B, C)$ is defined by $b \in \mathsf{cu} f a \psi \equiv (a, b) \in f\psi$. The definition of ap' in $\mathbf{Prog}((B \rightsquigarrow' C) \otimes B, C)$ is the same as for ap

but with $\rightsquigarrow$ replaced by $\rightsquigarrow'$. For any f in **Prog**, the definition of $\mathsf{cur}'f$ is the same as for $\mathsf{cur}\,f$ but with φ ranging over $\mathcal{U}(B \rightsquigarrow' C)$. For ap' and cur' to have reasonable properties, $\mathsf{cu}f a$ should be in **Prog** for every a, if $f \in \mathbf{Prog}$. This is true if **Prog** is defined by healthiness. Note that $\mathsf{cur}'f$ is not defined for f not in **Prog**; cur' can be extended to **Spec** but with poor properties.

For f in $\mathbf{Spec}(A, B)$ to be *data-refined* by f' in $\mathbf{Spec}(A', B')$ *via simulation* s means that $s_A \in \mathbf{Spec}(A', A)$ and $s_B \in \mathbf{Spec}(B', B)$ are comaps such that

$$s_A \fatsemi f \sqsubseteq f' \fatsemi s_B \qquad \begin{array}{ccc} A' & \xrightarrow{\;f'\;} & B' \\ {\scriptstyle s_A}\downarrow & \sqsubseteq & \downarrow{\scriptstyle s_B} \\ A & \xrightarrow[\;f\;]{} & B \end{array} \tag{17}$$

Any comap $A' \to A$ is $\langle R \rangle$ for some $R \in \mathbf{IRel}(A, A')$, and the ideal property makes sense if aRa' means that a is simulated by a' and $\preceq$ means refinement or approximation. The conjunction of (17) and the comap requirement has been called *total simulation*; note that it does not require the underlying relations to be total or functional.

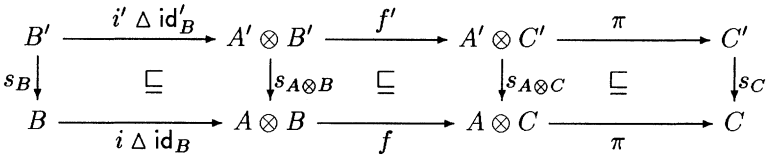
Data refinement is normally used as follows: for primitive types A, B and operations f, one explicitly defines s_A and s_B and proves (17). For constructed f, there should be a general result to the effect that constructed arrows are data refined via suitably constructed simulations; such constructs are said to *preserve data refinement*. For example, suppose (17) holds and also $g : A \to C$ is data-refined to $g' : A' \to C'$ by s_A, s_C. Then $f \vartriangle g$ should be data-refined by $f' \vartriangle g'$ via s_A and $s_{B \otimes C}$ where $s_{B \otimes C} = s_B \otimes s_C$.

The product and exponent in **Spec** preserve data refinement (Naumann 1995*a*); in particular, $s_{B \rightsquigarrow C}$ is defined as $s_B^* \rightsquigarrow s_C$ where s_B^* is the map determined by comap s_B. The usual imperative constructs preserve data refinement (Gardiner & Morgan 1991). The catamorphism construct preserves data refinement, because it can be expressed using general recursion which preserves data refinement. The local variable construct has been shown to preserve data refinement, but the proofs in the literature are at the level of predicates. We give a proof here because it demonstrates program-level calculation in **Spec** and because it provides a simple proof of soundness as explained below.

In the context of some 'global' state space(s), the local variable construct augments the state space with a new variable initialized in terms of the global state space, and then it executes the command and drops the new variable. Thus it takes an arrow $f : A \otimes B \to A \otimes C$ in **Spec** and an initialization $i : B \to A$ and yields the arrow $\mathsf{var}(i, f) : B \to C$ defined as $(i \vartriangle \mathsf{id}_B) \fatsemi f \fatsemi [\pi]$.

Theorem 10 *Local variables preserve data refinement. That is, if we have both* $s_{A \otimes B} \fatsemi f \sqsubseteq f' \fatsemi s_{A \otimes C}$ *and* $s_B \fatsemi i \sqsubseteq i' \fatsemi s_A$ *then* $\mathsf{var}(i, f)$ *is data-refined by* $\mathsf{var}(i', f')$ *via* s_B *and* s_C.

Proof. The result follows by pasting together the inequalities for the three components of $\mathsf{var}(i, f)$, depicted as follows.*



The middle inequality is by hypothesis, the right one follows from law (6), and

$$
\begin{array}{lll}
 & s_B \fatsemi (i \Delta \mathsf{id}) & \\
\sqsubseteq & s_B \fatsemi i \Delta s_B & s \text{ comap, law (7)} \\
\sqsubseteq & i' \fatsemi s_A \Delta s_B & \text{hypothesis (simulation for } i) \\
\sqsubseteq & (i' \Delta \mathsf{id}) \fatsemi s_{A,B} & \text{law (8), } s_{A\otimes B} = s_A \otimes s_B
\end{array}
$$

which proves the left inequality. □

Data refinement is used to prove ordinary refinement of programs in which the refined data is encapsulated as local variables, which is usually described as follows.

Corollary 11 (Soundness) *If $i' : B \to A'$ is data-refined by $i : B \to A$ via s_A and $f' : A' \otimes B \to A' \otimes C$ by $f : A \otimes B \to A \otimes C$ via s_A then* $\mathsf{var}(i, f) \sqsubseteq \mathsf{var}(i', f')$.

Proof. Take $R_A = \mathsf{id}_A$ and $R_C = \mathsf{id}_C$ in the theorem. □

A very similar notion of soundness justifies rule (16). If $s_A = \mathsf{id}$ and $s_B = \mathsf{id}$ then (17) is just $f \sqsubseteq f'$. So one can prove $f \sqsubseteq f'$ by data-refining constituents of f to corresponding constituents of f', and then conclude $f \sqsubseteq f'$ provided that the 'observable' types A, B are simulated by identities and the constructs preserve data refinement. We refrain from formalizing the result or its proof.

8 FROM Spec-EXPONENT TO Prog-EXPONENT

Typically, a simulation is the identity on observable types because it is built inductively from simulations on primitive types, with identity simulations for observable types and constructors that preserve identities. But here we need to refine a constructor itself (namely $\rightsquigarrow$). We shall give non-identity simulations for $\rightsquigarrow$-types, but take all simulations on primitives to be identity. Thus if $f : A \to B$ is simulated by

*Here we assume that $s_{A,B} = s_A \otimes s_C$ and $s_{A,C} = s_A \otimes s_C$ for some s_A, s_B, s_C. The assumption is reasonable: if $s_{A,C}$ involves A, C in combination then A will not be localized separately.

$f' : A' \to B'$ and A', B' are exponent-free then $A = A'$, $B = B'$, and $f \sqsubseteq f'$. That is why rule (16) needs the proviso that type are $\rightsquigarrow$-free.

Define $\mathsf{inc} : (B \rightsquigarrow' C) \to (B \rightsquigarrow C)$ to be the inclusion function. The simulation will be its inverse image $[\mathsf{inc}]$ in $\mathbf{Spec}((B \rightsquigarrow' C), (B \rightsquigarrow C))$, which is a bimap by Lemma 8(b). For any $g \in B \rightsquigarrow' C$ and $\varphi \in \mathcal{U}(B \rightsquigarrow C)$ we have $g \in [\mathsf{inc}]\varphi \equiv g \in \varphi$. Put differently, $[\mathsf{inc}]\varphi = \varphi \cap (B \rightsquigarrow' C)$. Informally, if φ is a predicate on programs then $[\mathsf{inc}]\varphi$ is the predicate 'satisfies φ and is in **Prog**'.

Formal proof of rule (16) is by induction on the structure of f, and simulations are defined by induction on the structure of types (needed because $\rightsquigarrow$-types can be nested). The interesting case is the simulation $s_{A \rightsquigarrow B}$ which needs to have type $(A' \rightsquigarrow' B') \to (A \rightsquigarrow B)$ using $s_A : A' \to A$ and $s_B : B' \to B$. One defines $s_{A \rightsquigarrow B}$ as

$$A' \rightsquigarrow' B' \xrightarrow{[\mathsf{inc}]} A' \rightsquigarrow B' \xrightarrow{s_A^* \rightsquigarrow s_B} A \rightsquigarrow B$$

But the second half, i.e. $s_A^* \rightsquigarrow s_B$, is treated in Naumann (1995*a*) to show that $\rightsquigarrow$ preserves data refinement. To focus on what is new, we treat only the special case $s_A = \mathsf{id}_A$ and $s_B = \mathsf{id}_B$ so that $s_{A \rightsquigarrow B}$ is just $[\mathsf{inc}]$ (because $\rightsquigarrow$ preserves identities).

Theorem 12 ap' *data-refines* ap *via* $[\mathsf{inc}]$. *And for any* f *in* $\mathbf{Prog}(A \otimes B, C)$, $\mathsf{cur} f$ *is data-refined by* $\mathsf{cur}' f$ *via* $[\mathsf{inc}]$.

The proof is a straightforward calculation at the level of predicates. The pictures are as follows (the righthand diagram is actually an equality).

$$\begin{array}{ccc} A \otimes (A \rightsquigarrow' B) & \xrightarrow{\mathsf{ap}'} & B \\ {\scriptstyle \mathsf{id} \otimes [\mathsf{inc}]}\downarrow & \sqsubseteq & \downarrow{\scriptstyle \mathsf{id}} \\ A \otimes (A \rightsquigarrow B) & \xrightarrow[\mathsf{ap}]{} & B \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{\mathsf{cur}' f} & B \rightsquigarrow' C \\ {\scriptstyle \mathsf{id}}\downarrow & \sqsubseteq & \downarrow{\scriptstyle [\mathsf{inc}]} \\ A & \xrightarrow[\mathsf{cur} f]{} & B \rightsquigarrow C \end{array}$$

9 DISCUSSION

Although some algebraic laws of imperative programming are weaker than those of functional and relational programming – e.g. for product and exponent – Theorem 9 for recursion on inductive data types is no weaker than for functions, despite applying to all transformers. Because the exponent includes specifications as well as concrete programs, it is not a concrete data type, and the exponent of concrete programs does not have adequate algebraic properties. Thanks to Theorem 12, it is always sound to derive a program using laws of the abstract exponent and to interpret the result in terms of the concrete one.

An application to the repmin problem was briefly sketched. It is encouraging that the laws are adequate for a nontrivial example. But the gap between implemented

imperative languages and the notations used in methodological studies remains larger than for functional languages, perhaps because command typing has been neglected (not to mention polytypism).

The repmin example does not make a strong case for the algebra of transformers, because with our interpretation of the exponent the derived program is essentially the same as its functional counterpart. Much more striking would be derivation of a first-order program using pointers. Perhaps a relational theory of pointers like that of Möller (1997) can be lifted. Is there a connection between pointers and exponents beyond the writing of arrows?

REFERENCES

Bird, R. & de Moor, O. (1996) *Algebra of Programming*. Prentice-Hall.

de Moor, O. (1992) Inductive data types for predicate transformers. *Information Processing Letters* **43**(3), 113–118.

de Moor, O. (1996) An exercise in polytypic programming: repmin. Typescript, www.comlab.ox.ac.uk/oucl/publications/books/algebra/papers/repmin.ps.gz.

Gardiner, P. H., Martin, C. E. & de Moor, O. (1994) An algebraic construction of predicate transformers. *Science of Computer Programming* **22**, 21–44.

Gardiner, P. & Morgan, C. (1991) Data refinement of predicate transformers. *Theoretical Computer Science* **87**, 143–162.

Jansson, P. & Jeuring, J. (1997) PolyP — a polytypic programming language extension. *In* Proceedings, POPL, ACM Press, pp. 470–82.

Martin, C. (1995) Towards a calculus of predicate transformers. *In* Proceedings, MFCS, Vol. 969 of *Springer LNCS*, pp. 489–49.

Möller, B. (1997) Calculating with pointer structures. *In* IFIP TC2/WG2.1 Working Conference on Algorithmic Languages and Calculi.

Morgan, C. (1994) *Programming from Specifications, second edition.* Prentice Hall.

Naumann, D. A. (1994*a*) Predicate transformer semantics of an Oberon-like language. *In* E.-R. Olderog, ed., *Programming Concepts, Methods and Calculi*, IFIP Transactions A-56, Elsevier.

Naumann, D. A. (1994*b*) A recursion theorem for predicate transformers on inductive data types. *Information Processing Letters* **50**, 329–336.

Naumann, D. A. (1995*a*) Data refinement, call by value, and higher order programs. *Formal Aspects of Computing* **7**, 652–662.

Naumann, D. A. (1995*b*) Predicate transformers and higher order programs. *Theoretical Computer Science* **150**, 111–159.

Naumann, D. A. (1996) A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science.* To appear.

Naumann, D. A. (1998) Beyond fun: Order and membership in polytypic imperative programming. *In* Mathematics of Program Construction.

10 BIOGRAPHY

Too little of software practice rests on a scientific foundation, as the author learned through experience as a professional programmer. His goal is to bridge the gap between methodological studies and the languages and tools used in practice. After completing his doctoral study at the University of Texas at Austin he joined the faculty of nearby Southwestern University, where formal methods were being integrated throughout the undergraduate curriculum. He is writing an undergraduate textbook on data structures and data refinement based on this experience. He is now Assistant Professor of Computer Science at Stevens Institute of Technology.