The Zip Calculus*

Mark Tullsen

Department of Computer Science
Yale University
New Haven CT 06520-8285
tullsen@cs.yale.edu

Research Report YALEU/DCS/RR-1191
February 2000

Abstract. Many have recognized the need for genericity in programming and in program transformation. Genericity over data types has been achieved with polymorphism. Genericity over type constructors, often referred to as polytypism, is an area of active research. However, genericity over the length of tuples has not been achieved in a typed language. This paper shows the usefulness of such genericity and presents the zip calculus, an extension of a typed lambda calculus that gives genericity over the length of tuples.

1 Introduction

The key to writing robust software is abstraction, but genericity is needed to use abstraction: to write a generic sort routine, genericity over types is needed (i.e., polymorphism); to write a generic fold[1], genericity over type *constructors* (e.g., `List` and `Tree` where `List a` and `Tree a` are types) is needed—this is often called polytypism.

In program transformation the need for genericity is amplified: for example, in a monomorphic language, if we write `sortInt` and `sortFloat` we will have laws about `sortInt` and `sortFloat` instead of just one law about a generic `sort`; also we have to transform `sortInt` and `sortFloat` separately, even if we can "cut-and-paste" the program derivation. So, generic programs reduce not only the size of programs, but also the number of laws and the length of program derivations.

For this reason, the program transformation community, notably the Bird-Meertens Formalism (or Squiggol) community [BdM97,Mee86,MFP91], has been working to make programs more generic—not just polymorphic, but polytypic [Mal90b,Mal90a,JJ97,JBM98]. However, the genericity provided by polymorphism and polytypism is still not adequate for certain programs: another form of genericity is often needed—genericity over the length of tuples. This paper shows the usefulness of "n-tuples" (tuples whose lengths are unknown) and proposes a method to extend a programming language with n-tuples.

Section 2 gives examples of the usefulness of n-tuples. Section 3 describes the zip calculus, its syntax, semantics and type system. Section 4 returns to the examples and shows what programs, laws, and program derivations look like using the zip calculus; other applications are also presented, including how to generalize catamorphisms to mutually recursive data types. Finally, section 5 discusses some limitations and compares this work to related work.

* This research was supported in part by NSF under Grant Number CCR-9706747.

[1] Otherwise known as a catamorphism, a function inductively defined over an inductive data structure.

2 Why Are N-Tuples Needed?

An n-tuple is a tuple whose length is unknown. Is such genericity useful? Definitely! Just like the genericity provided by polymorphism and polytypism, n-tuples give (1) more general programs, (2) more general laws about those programs, and (3) more general program derivations.

2.1 More General Programs

The following functions are defined in the Haskell [HJW92] Prelude

```
zip  :: [a]→[b]→[(a,b)]
zip3 :: [a]→[b]→[c]→[(a,b,c)]
zip4 :: [a]→[b]→[c]→[d]→[(a,b,c,d)]
...
```

whose functionality can be guessed from their types. Also, there are the family of functions `unzip`, `unzip3`, `unzip4`, ... and the family of functions `zipWith`, `zipWith3`, `zipWith4`, ... To write the `zip3`, `zip4`, ... functions is not hard but tedious. It is clearly desirable to abstract over these and write one generic `zip`, one generic `zipWith`, and one generic `unzip`.

2.2 More General Laws

Note the free theorem [Wad89] for `zip` (an uncurried version)[2]:

```
map(cross(f,g)) · zip  =  zip · cross(map f,map g)
where
cross (f,g) (x,y) = (f x,g y)
```

The comparable theorem for `zip3` is

```
map(cross3(f,g,h)) · zip3 = zip3 · cross3(map f,map g,map h)
where
cross3 (f,g,h) (x,y,z) = (f x, g y, h z)
```

To generate these laws is not hard but tedious (and error-prone). To formulate this *family* of laws yet another family of functions is needed: `cross`, `cross3`, `cross4`, ... And note the following laws for 2-tuples and 3-tuples[3]:

```
(fst x, snd x) = x
(fst3 x, snd3 x, thd3 x) = x
```

(For which one needs *another* set of families of functions: `fst`, `fst3`, `fst4`, ... and `snd`, `snd3`, `snd4`, ...) One would wish to generalize over these *families* of laws. Having fewer, but more generic, laws is very desirable in a program transformation system: one has fewer laws to learn, fewer laws to search, and more robust program derivations (i.e., program derivations are more likely to remain valid when applied to a modified input program).

[2] "·" is function composition; "`map f`" applies `f` to each element of a list.

[3] Ignoring the complication that these laws are not valid in Haskell, which has lifted tuples; these same laws are valid in the Zip Calculus which has unlifted tuples.

2.3 More General Program Derivations

It is common to have program derivations of the following form:

`fst e`
`= ...`
`= e1`
	Prove the case for the "fst" of the tuple.

Similarly, `snd e = e2`	*Wave hands.*
Thus, `e = (fst e,snd e) = (e1,e2)`	*Make a conclusion about the tuple as a whole.*

When arguing informally, this works well and of course scales easily to 3-tuples and up. However, in a practical program transformation system this "similarly" step must be done without "hand waving" and hopefully without duplicating the derivation. One way to do this is to express the above law in some meta-language or meta-logic where one could say something like $\forall n.\forall i < n.P($`#i`$)$ (using ML syntax for projections where `#1` = `fst`, `#2` = `snd`).

However, a meta-language is now needed to express program laws. A simpler approach to transformation, the schematic approach, avoids the use of a meta-language: program laws are of the form "$e_1 = e_2 \Leftarrow e_3 = e_4$" [HL78] ($e_1, e_2, e_3, e_4$ are programs in the language, all free variables are implicitly universally quantified, and the premise is optional); program derivations are developed by successively applying program laws: the law is instantiated, the premise is satisfied, then the conclusion is used to replace equals for equals. However, using this approach for the above derivation requires one to duplicate the derivation for the `fst` and `snd` cases. Is it possible to avoid this duplication of derivations? Note that, in general, the form of `(e1,e2)` is `(C[fst],C[snd])`[4] (or can be transformed into such a form). So, one would like to merge the two similar derivations

```
fst e = ... = C[fst]
snd e = ... = C[snd]
```

into a single derivation

```
#i e  = ... = C[#i]
```

However, this still does not work because the "`i`" in `#i` must be an integer and cannot be a variable. But if "`i`" could be a variable, then simple equational reasoning can be used—as in the schematic approach—without the need to add a meta-language. The zip calculus allows one to do this.

3 The Zip Calculus

The zip calculus is a typed lambda calculus extended with n-tuples and sums. In particular, it starts as F_ω, encoded as a Pure Type System (PTS) [Bar92,PM97], a construct for n-tuples is added, and then n-sums are added (very simply using n-tuples). In a PTS, terms, types, and kinds are all written in the same syntax.

[4] Where `C[e]` represents a program context `C[]` with its holes filled by expression `e`.

e	$::= v$	variables
	$\mid \lambda v\!:\!t.\, e$	abstraction
	$\mid e_1 e_2$	application
	$\mid \Pi v\!:\!t_1.\, t_2$	type of abstractions
	$\mid \star$	type of types
	$\mid \langle e_1, e_2, ...\rangle t$	tuple (having type t)
	$\mid m_n$	projection ($1 \leq m \leq n$)
	$\mid n^{\mathrm{d}}$	dimension
	$\mid \mathrm{D}$	type of dimensions
	$\mid +_d\, t$	sum type
	$\mid In_d$	constructors for $+_d$ ($\langle In_{d.1}, In_{d.2}, ...\rangle$)
	$\mid \mathrm{case}_d$	destructor for $+_d$
i	$::= e$	projections (of type n^{d})
d	$::= e$	dimensions (of type D)
t	$::= e$	types and kinds (of type $\star$ or $\square$)
m, n	$::=$ {natural numbers}	

Fig. 1. Syntax

As the syntax of terms and types was becoming nearly identical (because tuples exist at the type level), the choice of a PTS seemed natural. Also, the generality of a PTS makes for fewer typing rules. However, the generality of a PTS can make a type system harder to understand: it is difficult to know what is a valid term, type, and kind without understanding the type checking rules.

3.1 Syntax and Semantics

The syntax of the terms of the zip calculus is in Fig. 1. The pseudo syntactic classes i, d, and t are used to provide intuition for what is enforced by the type system (but not by the syntax). F_ω encoded as a PTS would have the first five terms in Fig. 1. The following are added: (1) Tuples which are no longer restricted to the term level but exist at the type level. (2) Projection constants (m_n - get the m-th element of an n-tuple), their types (n^{d} - dimensions, where $m_n : n^{\mathrm{d}}$), and "D" the type of these n^{d}. And (3) n-sums made via n-tuples: for n-sums ($+_{(n^{\mathrm{d}})}\langle t_1, ..., t_n\rangle u$) the constructor family, $In_{(n^{\mathrm{d}})}$, is an n-tuple of constructors and the destructor $\mathtt{case}_{(n^{\mathrm{d}})}$ takes an n-tuple of functions.

To get the second element of a 3-tuple, a projection is applied to it $\langle e_1, e_2, e_3\rangle t\ 2_3$ giving e_2; a 3-tuple is a function whose range is $\{1_3, 2_3, 3_3\}$ (the projections with type 3^d). In this example e_1, e_2, e_3 can each have a different type because t, the type of the tuple, can be a dependent type (a Π term): for instance, one can write $\langle e_1, e_2, e_3\rangle(\Pi i : 3^d.\, \langle E_1, E_2, E_3\rangle(\Pi\_ \!:\! 3^d.\, \star)\, i)$[5]. Genericity over tuple length is achieved because we can write functions such as "$\lambda d\!:\!D.\lambda i\!:\!d.e$" in which d can be any dimension (1^d,2^d,...).

[5] Using "_" for an unused variable. Also, $a \to b$ is used as syntactic sugar for $\Pi\_ \!:\! a.\, b$)

Reduction Rules:

$$
\begin{array}{rll}
(\lambda v\!:\!t.e_1)\, e_2 = e_1\{e_2/v\} & & (\beta \text{ reduce}) \\
\langle e_1, \ldots, e_n\rangle t\, i_n = e_i & & (\times \text{ reduce}) \\
\mathrm{case}_d\ e\,((In_d)_{.i}\, e') = e_{.i}\, e' & & (+ \text{ reduce})
\end{array}
$$

Eta laws:

$$
\begin{array}{rll}
\lambda x\!:\!a.\, e\, x = e & \text{if } e :: a \to b,\ x \notin \mathrm{fv}(e) & (\to \text{eta}) \\
\langle e\, 1_n, \ldots, e\, n_n\rangle(\Pi i\!:\!n^{\mathrm{d}}.A) = e & \text{if } e :: (\Pi i\!:\!n^{\mathrm{d}}.A) & (\times \text{ eta}) \\
\mathrm{case}_d\ In_d\ e = e & \text{if } e :: +_d A & (+ \text{ eta})
\end{array}
$$

Instantiation:

$$
\begin{array}{rll}
h \cdot \mathrm{case}_d\, f = \mathrm{case}_d\, \langle^{i:d}\, h \cdot f_{.i}\rangle & \text{if } h \text{ strict} & (\text{inst}) \\
\mathrm{C}[\mathrm{case}_d\, \langle^{i:d} \lambda v\!:\!t.\, e\rangle\, x] = \mathrm{case}_d\, \langle^{i:d} \lambda v\!:\!t.\, \mathrm{C}[e]\rangle\, x & \text{if } \mathrm{C}[\,] \text{ strict} & (\text{inst})
\end{array}
$$

Fig. 2. Laws

Although tuples are another form of function, the following syntactic sugar is used to syntactically distinguish tuple functions from standard functions:

$$
\begin{aligned}
\langle^{i:d} e\rangle &\equiv \lambda i\!:\!d\,.\,e \\
e_{.i} &\equiv e\, i \\
\times_d\, t &\equiv \Pi i\!:\!d\,.\,t\, i
\end{aligned}
$$

Since one can write tuples of types, one must distinguish between $\langle t_1, t_2\rangle(2d \to \star)$ (a tuple of types, having kind $2d \to \star$) and $\times_{(2^{\mathrm{d}})}\langle t_1, t_2\rangle(2d \to \star)$ (a product of types, having kind $\star$).

The semantics is given operationally: the three reduction rules of Fig. 2 are applied left to right with a leftmost outermost reduction strategy. Translating the (β reduce) and ($\to$ eta) laws into the above syntactic sugar gives these laws:

$$
\begin{array}{lll}
\langle^{i:d} e\rangle_{.j} = e\{j/i\} & & (\text{n-tuple reduce}) \\
\langle^{i:d} e_{.i}\rangle = e & \text{if } e :: \times_d A,\ i \notin \mathrm{fv}(e) & (\text{n-tuple eta})
\end{array}
$$

To give some intuition regarding the semantics of n-tuples, note this equivalence:

$$
\begin{array}{ll}
\quad \langle^{i:2^{\mathrm{d}}} \langle \mathtt{f},\mathtt{g}\rangle(\mathtt{2d{\to}a{\to}b})_{.i}\ \langle \mathtt{x},\mathtt{y}\rangle(\mathtt{2d{\to}a})_{.i}\rangle & \\
= & \{\times\ \mathit{eta}\} \\
\quad \langle\langle^{i:2^{\mathrm{d}}} \langle \mathtt{f},\mathtt{g}\rangle(\mathtt{2d{\to}a{\to}b})_{.i}\ \langle \mathtt{x},\mathtt{y}\rangle(\mathtt{2d{\to}a})_{.i}\rangle_{.1_2}, & \\
\quad\ \langle^{i:2^{\mathrm{d}}} \langle \mathtt{f},\mathtt{g}\rangle(\mathtt{2d{\to}a{\to}b})_{.i}\ \langle \mathtt{x},\mathtt{y}\rangle(\mathtt{2d{\to}a})_{.i}\rangle_{.2_2}\rangle(\mathtt{2d{\to}b}) & \\
= & \{\mathit{n\text{-}tuple\ reduce},\ 2\mathrm{x}\} \\
\quad \langle\langle \mathtt{f},\mathtt{g}\rangle(\mathtt{2d{\to}a{\to}b})_{.1_2}\ \langle \mathtt{x},\mathtt{y}\rangle(\mathtt{2d{\to}a})_{.1_2}, & \\
\quad\ \langle \mathtt{f},\mathtt{g}\rangle(\mathtt{2d{\to}a{\to}b})_{.2_2}\ \langle \mathtt{x},\mathtt{y}\rangle(\mathtt{2d{\to}a})_{.2_2}\rangle\rangle(\mathtt{2d{\to}b}) & \\
= & \{\times\ \mathit{reduce},\ 4\mathrm{x}\} \\
\quad \langle \mathtt{f\ x,\ g\ y}\rangle(\mathtt{2d{\to}b}) &
\end{array}
$$

The tuples $\langle \mathtt{f},\mathtt{g}\rangle$ and $\langle \mathtt{x},\mathtt{y}\rangle$ are "zipped" together, this is why it is called the zip calculus.

$$\frac{\Gamma \vdash a : A, \quad \Gamma \vdash B : s, \quad A =_\beta B}{\Gamma \vdash a : B} \text{(conv)} \qquad \frac{c : s \in \mathcal{A}}{\vdash c : s} \text{(axiom)}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \text{(var)} \qquad \frac{\Gamma \vdash b : B \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B} \text{(weak)}$$

$$\frac{\Gamma \vdash f : (\Pi x : A.B), \quad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B\{a/x\}} \text{(app)} \qquad \frac{\Gamma, x : A \vdash b : B, \quad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \text{(lam)}$$

$$\frac{\Gamma \vdash A : s, \quad \Gamma, x : A \vdash B : t, \quad (s,t,u) \in \mathcal{R}}{\Gamma \vdash (\Pi x : A.B) : u} \text{(pi)}$$

Fig. 3. Type Judgments for a Pure Type System

$$\frac{\forall j \in \{1..n\}.\ \Gamma \vdash a_j : A\{j_n/i\}, \quad \Gamma \vdash (\Pi i : n^{\mathrm{d}}.A) : t}{\Gamma \vdash \langle a_1, ..., a_n \rangle\, (\Pi i : n^{\mathrm{d}}.A) : (\Pi i : n^{\mathrm{d}}.A)} \text{(tuple)}$$

Fig. 4. Additional Type Judgments for the Zip Calculus

3.2 The Type System

The terms of a PTS consist of the first four terms of Fig. 1 (variables, lambda abstractions, applications, and Π terms) plus a set of constants, $\mathcal{C}$. The specification of a PTS is given by a triple ($\mathcal{S}$,$\mathcal{A}$,$\mathcal{R}$) where $\mathcal{S}$ is a subset of $\mathcal{C}$ called the sorts, $\mathcal{A}$ is a set of axioms of the form "$c : s$" where $c \in \mathcal{C}$, $s \in \mathcal{S}$, and $\mathcal{R}$ is a set of rules of the form $(s1, s2, s3)$ where $s1, s2, s3 \in \mathcal{S}$. The typing judgments for a PTS are given in Fig. 3. In a PTS, the definition of $=_\beta$ in the judgment (conv) is beta-equivalence (alpha-equivalent terms are identified).

In the case of the zip calculus, the set of sorts is $\mathcal{S} = \{1^{\mathrm{d}}, 2^{\mathrm{d}}, ...\} \cup \{\star, \Box, D\}$, the set of constants is $\mathcal{C} = \mathcal{S} \cup \{m_n | 1 \le m \le n\}$, and the axioms $\mathcal{A}$ and rules $\mathcal{R}$ are as follows:

$\mathcal{A}$ axioms	$\mathcal{R}$ rules	
$\star : \Box$	$(\star, \star, \star)$	$\lambda v_e : t\,.\,e$
$m_n : n^{\mathrm{d}}$	$(\Box, \star, \star)$	$\lambda v_t : T\,.\,e$
$n^{\mathrm{d}} : \mathrm{D}$	$(\Box, \Box, \Box)$	$\lambda v_t : T\,.\,t$
$\mathrm{D} : \Box$	$(\mathrm{D}, \mathrm{D}, \star)$	$\lambda v_i : d\,.\,i$
	$(\mathrm{D}, \star, \star)$	$\lambda v_i : d\,.\,e$
	$(\mathrm{D}, \Box, \Box)$	$\lambda v_i : d\,.\,t$

The $\mathcal{R}$ rules indicate what lambda abstractions are allowed (which is the same as saying which Π terms are well-typed). Here there are six $\mathcal{R}$ rules which correspond to the six allowed forms of lambda abstraction. The expression to the right of each rule is an intuitive representation of the type of lambda abstraction which the rule represents (e - terms, t - types, T - kinds, i - projections, d - dimensions, v_x - variable in class x).

In the zip calculus there is an additional term, $\langle e_1, e_2, ... \rangle t$, which cannot be treated as a constant in a PTS (ignoring sums for the moment). The addition of this term requires two extensions to the PTS: one, an additional typing judgment (Fig. 4) and two, the $=_\beta$ relation in the (conv) judgment must be extended to include not just (β reduce) but also ($\times$ reduce) and ($\times$ eta).

To get generic sums, one needs only add $+$ as a constant and the following two primitives:

$$\begin{array}{lll} \texttt{In} & :: \Pi I{:}\mathrm{D}.\ \Pi a : \times_I \langle -^{:I} \star \rangle. & \times_I \langle^{i:I}\, a_{.i} \to +_I\, a \rangle \\ \texttt{case} & :: \Pi I{:}\mathrm{D}.\ \Pi a : \times_I \langle -^{:I} \star \rangle . \Pi b : \star. & \times_I \langle^{i:I}\, a_{.i} \to b \rangle \to (+_I\, a \to b) \end{array}$$

$$\frac{\Gamma \vdash f :\twoheadrightarrow (\Pi x : A.B), \quad \Gamma \vdash a : A',\ A =_\beta A'}{\Gamma \vdash f\,a : B\{a/x\}}\ \text{(app)} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{(var)}$$

$$\frac{\Gamma, x : A \vdash b : B, \quad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}\ \text{(lam)} \qquad \frac{c : s \in \mathcal{A}}{\vdash c : s}\ \text{(axiom)}$$

$$\frac{\Gamma \vdash A :\twoheadrightarrow s, \quad \Gamma, x : A \vdash B :\twoheadrightarrow t, \quad (s,t,u) \in \mathcal{R}}{\Gamma \vdash (\Pi x : A.B) : u}\ \text{(pi)} \qquad \frac{\Gamma \vdash a : A, \quad A \twoheadrightarrow_\beta B}{\Gamma \vdash a :\twoheadrightarrow B}\ \text{(red)}$$

Fig. 5. Syntax Directed Type Judgments for a Functional PTS

$$\frac{\forall j \in \{1..n\}.\ \Gamma \vdash a_j : A\{j_n/i\}, \quad \Gamma \vdash (\Pi i : n^{\mathrm{d}}.A) : t}{\Gamma \vdash \langle a_1, ..., a_n \rangle\, (\Pi i : n^{\mathrm{d}}.A) : (\Pi i : n^{\mathrm{d}}.A)}\ \text{(tuple)}$$

$$\frac{\Gamma \vdash a : A, \quad A \twoheadrightarrow_{\beta\delta} B}{\Gamma \vdash a :\twoheadrightarrow B}\ \text{(red}')$$

$$\frac{\Gamma \vdash f :\twoheadrightarrow C, \quad \Gamma \vdash a :\twoheadrightarrow A,\ C =_\eta \Pi x : A.B}{\Gamma \vdash f\,a : B\{a/x\}}\ \text{(app}')$$

Fig. 6. Syntax Directed Type Judgments for the Zip Calculus

3.3 Type Checking

There are numerous properties, such as subject reduction, which are true of Pure Type Systems in general [Bar92]. There are also known type checking algorithms for certain subclasses of PTSs. Although the zip calculus is not a PTS, it is hoped that most results for PTSs will carry over to the "almost PTS" zip calculus.

A PTS is functional when the relations $\mathcal{A}$ and $\mathcal{R}$ are functions ($c : s_1 \in \mathcal{A}$ and $c : s_2 \in \mathcal{A}$ imply $s_1 = s_2$; $(s,t,u_1) \in \mathcal{R}$ and $(s,t,u_2) \in \mathcal{R}$ imply $u_1 = u_2$). In the case of the zip calculus, $\mathcal{A}$ and $\mathcal{R}$ are functions. If a PTS is functional there is an efficient type-checking algorithm as given in Fig. 5 (cf. [PM97] and [VMP94]), where the type judgments of Fig. 3 have been restructured to make them syntax-directed. The judgment (red) just defines the shortcut "$\Gamma \vdash x :\twoheadrightarrow X$" and $\twoheadrightarrow_\beta$ is beta-reduction.

This algorithm can be modified as in Fig. 6. The rule (tuple) is as before (Fig. 4) but (app′) and (red′) replace the (app) and (red) judgments of Fig. 5. Here $\twoheadrightarrow_{\beta\delta}$ is $\twoheadrightarrow_\beta$ extended with ($\times$ reduce) and $=_\eta$ is equality up to ($\times$ eta) convertibility. The intuition for the change of (app) is that f may evaluate to

$$\langle \Pi x : a_1.b_1, ..., \Pi x : a_n.b_n \rangle t\; i$$

and application should be valid when, for instance, this is equivalent to a type of the form

$$\Pi x : \langle a_1, ..., a_n \rangle (n^{\mathrm{d}} \to \star)\, i\, .\, \langle b_1, ..., b_n \rangle (n^{\mathrm{d}} \to \star)\, i$$

A proof of the soundness and completeness of this algorithm should be similar to that in [VMP94].

4 Examples

Writing programs in an explicitly typed calculus can be onerous; to alleviate this, a number of shortcuts are often used in the following: the "$:t$" is dropped in lambdas (and in the n-tuple syntactic sugar); the t is dropped from $\langle x_1, ..., x_n\rangle t$; m is put for the projection m_n; the dimension d is dropped from $\times_d$; and when applying dimensions and types, "f_{d,t_1,t_2}" is put for "$f\, d\, t_1\, t_2$". Also, $f\,x\ = e$ is syntactic sugar for $f\ = \lambda x.e$. The following conventions are used for meta-variables: t, a, b, c, A, B, C for types (terms of type $\star$), i, j, k, l for projections (terms of type n^{d}), and d, I, J, K, L for dimension variables (terms of type $\mathtt{D}$). (Distinguish the variable d from a dimension written as 2^{d} or n^{d}.)

So, armed with the zip calculus, what kind of programs and laws can be written?

4.1 More General Programs

An uncurried `zip3` is as follows in Haskell:

```
zip3 :: ([a],[b],[c]) → [(a,b,c)]
zip3 (a:as,b:bs,c:cs) = (a,b,c) : zip3 (as,bs,cs)
zip3 _                = []
```

If Haskell had n-tuples, one could write a generic `zip` as follows:

```
zip :: ×⟨ⁱ[a.ᵢ]⟩ → [×a]
zip   ⟨ⁱx.ᵢ:xs.ᵢ⟩ = x : zip xs
zip   _         = []
```

Note that patterns are extended with n-tuples. Unfortunately, this zip cannot be written in the zip calculus (extended with recursive data types and a fix point operator) unless a primitive such as `seqTupleMaybe` is added:

```
seqTupleMaybe :: ×⟨ⁱ a.ᵢ →Maybe b.ᵢ⟩ → ×a→Maybe(×b)
```

However, once this primitive is added, n-tuple patterns can be defined. It is a trivial extension of the transformation given in [Tul00]. Section 5.1 returns to this problem of functions that must be primitives.

4.2 More General Laws

The parametricity theorem for an uncurried zip3

```
map(cross3⟨f,g,h⟩) · zip3 = zip3 · cross3⟨map f,map g,map h⟩
where
cross3 ⟨f,g,h⟩ ⟨x,y,z⟩ = ⟨f x, g y, h z⟩
```

can be generalized in the zip calculus to this:

```
map(cross_d f) · zip = zip · cross_d ⟨ⁱ:ᵈmap f.ᵢ⟩
where
cross_d f x = ⟨ⁱ:ᵈf.ᵢ x.ᵢ⟩
```

And this law

```
⟨x.₁, x.₂, x.₃⟩ = x
```

can be generalized to the (n-tuple eta) law:

```
⟨ⁱ:ᵈx.ᵢ⟩ = x
```

4.3 More General Derivations

Remember this example from section 2.3?

```
  fst e
= ...
= C[fst]

Similarly, snd e = C[snd]

Thus,
e = (fst e,snd e) = (C[fst],C[snd])
```

Now a generic transformation can be done handily:

$$
\begin{array}{ll}
 & \texttt{e} \\
= & \langle^{i}\ \ \texttt{e}_{.i}\ \ \rangle \\
= & \langle^{i}\ \ \ldots\ \ \rangle \\
= & \langle^{i}\ \ \texttt{C[}_{.i}\texttt{]}\ \rangle
\end{array}
$$

The `fst` and `snd` cases are transformed simultaneously by transforming the body of the n-tuple, Thus, a meta-language is no longer needed to express such generic transformations.

4.4 Nested N-Tuples

Typical informal notations for representing n-tuples are ambiguous: e.g., one writes $f\,\overline{x}$ for the "vector" $\langle f\,x_1, ..., f\,x_n\rangle$ but now $g\,(f\,\overline{x})$ could signify either $\langle g(f\,x_1), ..., g(f\,x_n)\rangle$ or $g\,\langle f\,x_1, ..., f\,x_n\rangle$. These notations do not extend to nested n-tuples. But in the zip calculus, one can easily manipulate nested n-tuples ("matrices"). For example, to apply a function to every element of a three-dimensional matrix is coded as follows (note that $\langle\text{-}^{:d}\texttt{e}\rangle$ is a tuple of identical elements):

$$
\begin{array}{lll}
\texttt{map3Dmatrix}_{a,b,I,J,K} & :: & (a \to b) \to \times\langle\text{-}^{:I}\times\langle\text{-}^{:J}\times\langle\text{-}^{:K}a\rangle\rangle\rangle \to \times\langle\text{-}^{:I}\times\langle\text{-}^{:J}\times\langle\text{-}^{:K}b\rangle\rangle\rangle \\
\texttt{map3Dmatrix}_{a,b,I,J,K} & = & \lambda\texttt{f}.\lambda\texttt{m}.\langle^{i:I}\langle^{j:J}\langle^{k:K}\texttt{f m}_{.i.j.k}\rangle\rangle\rangle
\end{array}
$$

The expression $\langle^{i:I}\langle^{j:J}\langle^{k:K}\texttt{e}\ \rangle\rangle\rangle$ is a 3-dimensional matrix where `e` is the value of the elements; here the value of the elements is "`f`" applied to the corresponding value of the original matrix "$\mathrm{m}_{.i.j.k}$". Matrix transposition is straightforward:

$$
\begin{array}{lll}
\texttt{transpose}_{I,J,a} & :: & \times\langle^{i:I}\times\langle^{j:J}a_{.i.j}\rangle\rangle \to \times\langle^{j:J}\times\langle^{i:I}a_{.i.j}\rangle\rangle \\
\texttt{transpose}_{I,J,a} & = & \lambda\texttt{x}.\langle^{j:J}\langle^{i:I}\ \ \texttt{x}_{.i.j}\ \rangle\rangle
\end{array}
$$

The transpose is done by "reversing" the subscripts of `x`. Note that the type variable a above is a *matrix* of types and, for any n, `transpose` could be applied to a tuple of n-tuples. An application of `transpose` is reduced as follows:

$$
\begin{array}{ll}
 & (\texttt{transpose}_{3^{d},2^{d},a}\ \langle\langle\texttt{x1},\texttt{x2}\rangle,\langle\texttt{y1},\texttt{y2}\rangle,\langle\texttt{z1},\texttt{z2}\rangle\rangle)_{.2.3} \\
\to & \langle^{j}\langle^{i}\ \langle\langle\texttt{x1},\texttt{x2}\rangle,\langle\texttt{y1},\texttt{y2}\rangle,\langle\texttt{z1},\texttt{z2}\rangle\rangle_{.i.j}\ \ \rangle\rangle_{.2.3} \\
\to & \ \langle^{i}\ \langle\langle\texttt{x1},\texttt{x2}\rangle,\langle\texttt{y1},\texttt{y2}\rangle,\langle\texttt{z1},\texttt{z2}\rangle\rangle_{.i.2}\ \ \rangle_{.3} \\
\to & \quad \langle\langle\texttt{x1},\texttt{x2}\rangle,\langle\texttt{y1},\texttt{y2}\rangle,\langle\texttt{z1},\texttt{z2}\rangle\rangle_{.3.2} \\
\to & \quad \langle\texttt{z1},\texttt{z2}\rangle_{.2} \\
\to & \quad \texttt{z2}
\end{array}
$$

Note the various ways one can transform a two dimensional matrix:

$\langle^i\langle^j \mathtt{m}_{.i.j}\rangle\rangle$	m itself
$\langle^j\langle^i \mathtt{m}_{.i.j}\rangle\rangle$	the transpose of m
$\langle^i\langle^j \mathtt{f}\ \mathtt{m}_{.i.j}\rangle\rangle$	f applied to each element of m
$\langle^i \mathtt{f}\ \langle^j \mathtt{m}_{.i.j}\rangle\rangle$	f applied to each "row" of m
$\langle^j \mathtt{f}\ \langle^i \mathtt{m}_{.i.j}\rangle\rangle$	f applied to each "column" of m

It should be clear that this notation extends to matrices of higher dimensions.

4.5 Program Transformation

The original motivation for the zip calculus was to create a language adapted to program transformation. This section shows how the zip calculus can simplify program transformation.

Here are two laws about the transpose function defined above:

$$m_{.i.j} = (\text{transpose}_{I,J,a}\ m)_{.j.i}$$
$$m = \text{transpose}_{J,I,b}(\text{transpose}_{I,J,a}\ m)$$

Here is a proof of the second (a proof of the first is part of the derivation):

$$
\begin{array}{lll}
 & \mathtt{transpose}_{J,I,b}\ (\mathtt{transpose}_{I,J,a}\ \mathtt{m}) & \\
= & \langle^l\langle^k\ \langle^j\langle^i \mathtt{m}_{.i.j}\rangle\rangle_{.k.l}\ \rangle\rangle & \{\textit{unfold transpose twice}\} \\
= & \langle^l\langle^k\ \langle^i \mathtt{m}_{.i.k}\rangle_{.l}\ \rangle\rangle & \{\textit{n-tuple reduce}\} \\
= & \langle^l\langle^k\ \mathtt{m}_{.l.k}\ \rangle\rangle & \{\textit{n-tuple reduce}\} \\
= & \langle^l\ \mathtt{m}_{.l}\ \rangle & \{\textit{n-tuple eta}\} \\
= & \mathtt{m} & \{\textit{n-tuple eta}\}
\end{array}
$$

A law, called Abides, is

$$
\begin{array}{c}
\mathtt{case}\ \langle\lambda\mathtt{x}.\langle\mathtt{a1},\mathtt{b1}\rangle\ ,\ \lambda\mathtt{y}.\langle\mathtt{a2},\mathtt{b2}\rangle\rangle\ \mathtt{x} \\
= \\
\langle\mathtt{case}\ \langle\lambda\mathtt{x}.\mathtt{a1},\lambda\mathtt{y}.\mathtt{a2}\rangle\ \mathtt{x}\ ,\ \mathtt{case}\ \langle\lambda\mathtt{x}.\mathtt{b1},\lambda\mathtt{y}.\mathtt{b2}\rangle\ \mathtt{x}\rangle
\end{array}
$$

and its derivation is

$$
\begin{array}{ll}
\mathtt{case}\ \langle\lambda\mathtt{x}.\langle\mathtt{a1},\mathtt{b1}\rangle\ ,\ \lambda\mathtt{y}.\langle\mathtt{a2},\mathtt{b2}\rangle\rangle\ \mathtt{x} & \\
= & \{\times\ \textit{eta}\} \\
\langle(\mathtt{case}\ \langle\lambda\mathtt{x}.\langle\mathtt{a1},\mathtt{b1}\rangle\ ,\ \lambda\mathtt{y}.\langle\mathtt{a2},\mathtt{b2}\rangle\rangle\ \mathtt{x})_{.1}\ , & \\
\quad(\mathtt{case}\ \langle\lambda\mathtt{x}.\langle\mathtt{a1},\mathtt{b1}\rangle\ ,\ \lambda\mathtt{y}.\langle\mathtt{a2},\mathtt{b2}\rangle\rangle\ \mathtt{x})_{.2}\ \rangle & \\
= & \{\textit{inst, 2x}\} \\
\langle(\mathtt{case}\ \langle\lambda\mathtt{x}.\langle\mathtt{a1},\mathtt{b1}\rangle_{.1},\ \lambda\mathtt{y}.\langle\mathtt{a2},\mathtt{b2}\rangle_{.1}\rangle\ \mathtt{x})\ , & \\
\quad(\mathtt{case}\ \langle\lambda\mathtt{x}.\langle\mathtt{a1},\mathtt{b1}\rangle_{.2},\ \lambda\mathtt{y}.\langle\mathtt{a2},\mathtt{b2}\rangle_{.2}\rangle\ \mathtt{x})\ \rangle & \\
= & \{\times\ \textit{reduce, 4x}\} \\
\langle\mathtt{case}\ \langle\lambda\mathtt{x}.\mathtt{a1},\ \lambda\mathtt{y}.\mathtt{a2}\rangle\ \mathtt{x}\ , & \\
\quad\mathtt{case}\ \langle\lambda\mathtt{x}.\mathtt{b1},\ \lambda\mathtt{y}.\mathtt{b2}\rangle\ \mathtt{x}\ \rangle &
\end{array}
$$

Here is a generic version of Abides

$$\texttt{case}\ \langle^{i} \lambda \texttt{y}.\langle^{j} \texttt{m}_{.i.j}\ \texttt{y}\rangle\rangle\ \texttt{x} = \langle^{j} \texttt{case}\ \langle^{i} \lambda \texttt{y}.\texttt{m}_{.i.j}\ \texttt{y}\rangle\ \texttt{x}\rangle$$

and its derivation is

$$\begin{array}{ll}
\quad \texttt{case}\ \langle^{i} \lambda \texttt{y}.\langle^{j} \texttt{m}_{.i.j}\ \texttt{y}\rangle\rangle\ \texttt{x} & \\
= & \{\textit{n-tuple eta}\} \\
\langle^{j} (\texttt{case}\ \langle^{i} \lambda \texttt{y}.\langle^{j} \texttt{m}_{.i.j}\ \texttt{y}\rangle\rangle\ \texttt{x})_{.j}\rangle & \\
= & \{\textit{inst}\} \\
\langle^{j}\ \texttt{case}\ \langle^{i} \lambda \texttt{y}.\langle^{j} \texttt{m}_{.i.j}\ \texttt{y}\rangle_{.j}\rangle\ \texttt{x}\rangle & \\
= & \{\textit{n-tuple reduce}\} \\
\langle^{j}\ \texttt{case}\ \langle^{i} \lambda \texttt{y}.\ \texttt{m}_{.i.j}\ \texttt{y}\quad \rangle\ \texttt{x}\rangle &
\end{array}$$

which corresponds directly to the non-generic derivation above. Note that instantiation is only applied once (not twice) and reduction once (not four times), and this law is generic over sums of any length and products of any length!

4.6 Generic Catamorphisms

It was obvious that Haskell's `zip` family of functions could benefit from n-tuples, but it is interesting that catamorphisms [MFP91] can benefit from n-tuples, resulting in catamorphisms over mutually recursive data structures.

First, a fix point operator for terms, **`fix`**, and a fix point operator at the type level, μ, must be added to the calculus. Normally, the kind of μ is $(\star \to \star) \to \star$ (i.e., it takes a functor of kind $\star \to \star$ and returns a type), but here the kind of $\mu_{(n^{\mathrm{d}})}$ is $(\times\langle^{-:n^{\mathrm{d}}} \star\rangle \to \times\langle^{-:n^{\mathrm{d}}} \star\rangle) \to \times\langle^{-:n^{\mathrm{d}}} \star\rangle$ (i.e., it takes a functor transforming n-tuples of types and returns an n-tuple of types). The subscript of μ is dropped when clear from the context. The primitives `in` and `out` now work on tuples of functions. Note how their types have been extended:

$$\begin{array}{lll}
\texttt{in}_{F} & :: \ F(\mu F) \to \mu F & \text{original} \\
\texttt{in}_{I,F} & :: \ \times\langle^{i:I} (F(\mu F))_{.i} \to (\mu F)_{.i}\rangle & \text{generic}
\end{array}$$

$$\begin{array}{lll}
\texttt{out}_{F} & :: \ \mu F \to F(\mu F) & \text{original} \\
\texttt{out}_{I,F} & :: \ \times\langle^{i:I} (\mu F)_{.i} \to (F(\mu F))_{.i}\rangle & \text{generic}
\end{array}$$

From these a more generic `cata` can be defined[6]:

$$\begin{array}{lll}
\texttt{cata}_{F,a} & :: \ (F\,a \to a) \to (\mu F \to a) & \text{original} \\
\texttt{cata}_{I,F,a} & :: \ \times\langle^{i:I} (F\,a)_{.i} \to a_{.i}\rangle \to \times\langle^{i:I} (\mu F)_{.i} \to a_{.i}\rangle & \text{generic}
\end{array}$$

$$\begin{array}{lll}
\texttt{cata}_{F,a}\ \phi & = \mathrm{fix}\ \lambda \texttt{f}.\phi \cdot F\ \ \texttt{f} \cdot \texttt{out}_{F} & \text{original} \\
\texttt{cata}_{I,F,a}\ \phi & = \mathrm{fix}\ \lambda \texttt{f}.\langle^{i:I} \phi_{.i} \cdot (F\ \ \texttt{f})_{.i} \cdot (\texttt{out}_{I,F})_{.i}\rangle & \text{generic}
\end{array}$$

---

[6] Of course, since the definition of `cata` is polytypic in the first place, this assumes that there is some form of polytypism (note the application of the functor `F` to a term), though type classes would suffice here.

So, $\mathtt{cata}_{(n^d),F,a}$ takes and returns an n-tuple of functions. All laws (such as cata-fusion) can now be generalized. Also, the standard functor laws for a functor F of kind $\star \to \star$

```
           id = F id
    F f · F g = F (f·g)
```

can be generalized to functors of kind $\times\langle -^{:I}\star\rangle \to \times\langle -^{:J}\star\rangle$:

$$\begin{aligned} \langle -^{:J}\mathtt{id}\rangle &= \mathtt{F}\ \langle -^{:I}\mathtt{id}\rangle \\ \langle^{j:J}(\mathtt{F\ f})_{.j}\ \cdot\ (\mathtt{F\ g})_{.j}\rangle &= \mathtt{F}\ \langle^{i:I}\ \mathtt{f}_{.i}\ \cdot\ \mathtt{g}_{.i}\ \rangle \end{aligned}$$

The original cata and functor laws can be derived from these by instantiating the n-tuples to 1-tuples and then making use of the isomorphism $\times\langle a\rangle \approx a$ (the bijections being $\lambda x.\, x_{.1_1}$ and $\lambda x.\langle x\rangle$).

5 Conclusion

5.1 Limitations

The zip calculus does not give polytypism (nor does polytypism give n-tuples); these are orthogonal language extensions:

- Polytypism: generalizes `zipList`, `zipMaybe`, `zipTree`, ...
- N-tuples: generalizes `zip`, `zip3`, `zip4`, ...

An n-tuple is similar to a heterogeneous array (or heterogenous finite list); but although one can map over n-tuples, zip n-tuples together, and transpose nested n-tuples, one *cannot induct* over n-tuples! So, n-tuples are clearly limited in what they can express. As a result, one cannot define the following functions in the zip calculus

$$\begin{aligned} &\mathtt{tupleToList} && ::\ \times\langle -^{:d}\mathtt{a}\rangle \to \mathtt{list\ a} \\ &\mathtt{seqTupleL,seqTupleR} && ::\ \mathtt{Monad\ m\ =>}\ \times\langle^{i}\ \mathtt{a}_{.i} \to \mathtt{m\ b}_{.i}\rangle \to \times\mathtt{a} \to \mathtt{m}(\times\mathtt{b}) \end{aligned}$$

However, if we provide `seqTupleL` and `seqTupleR` as primitives, then

- Each of these families of Haskell functions can be generalized to one generic function: `zip`..., `zipWith`..., `unzip`..., and `liftM1`...
- The function `seqTupleMaybe` from section 4.1 can be defined.
- A number of Haskell's list functions could also be defined for n-tuples: `zip`, `zip3`, ..., `zipWith`, `zipWith3`, ..., `unzip`, `unzip3`, ..., `map`, `sequence`, `mapM`, `transpose`, `mapAccumL`, `mapAccumR`. (These functions all act "uniformly" on lists—they act on lists without permuting the elements or changing their length.)

Other functions cannot even be given a type in the zip calculus. For instance, there is the `curry` family of functions

```
curry2 :: (a->b->c)    -> (a,b)->c
curry3 :: (a->b->c->d) -> (a,b,c)->d
...
```

but there is no way to give a type to a generic `curry`. Extending the zip calculus to type this generic `curry` is an area for future research.

5.2 Relation to Other Work

Polytypic programming [Mal90b,Mal90a,MFP91] has similar goals to this work (e.g., PolyP [JJ97] and Functorial ML [JBM98]). However, as just noted, the genericity of polytypism and n-tuples appear orthogonal. As seen in section 4.6, with *both* polytypism and n-tuples some very generic programs and laws can be written.

Two approaches that achieve the same genericity as n-tuples are the following: (1) One can forgo typed languages and use an *untyped language* to achieve this level of genericity: e.g., in Lisp a list can be used as an n-tuple. (2) A language with *dependent types* [Aug99] could encode n-tuples (and much more); though the disadvantages are that type checking is undecidable (not to mention the lack of type inference) and the types are more complex.

Related also is Hoogendijk's thesis [Hoo97] in which is developed a notation to generalize binary products of categories to n-products of categories; his notation is variable free, categorical, and heavily overloaded.

5.3 Summary

Implementation has not been addressed. One method is to simply inline all n-tuples, although this could lead to code explosion and does not support separate compilation. Another method is to implement n-tuples as functions (as they are just another form of function); just as there are a range of implementation techniques for polymorphic functions, there are analogous choices for implementing functions generic over dimensions.

Future work is (1) to extend the zip calculus to be polytypic, (2) to increase the expressiveness of the zip calculus (so `seqTupleL`, `seqTupleR`, and `tupleToList` can be defined in the language and `curry` could be given a type), and (3) to implement a type inference algorithm for the zip calculus.

I hope to have shown that the genericity provided by n-tuples is useful in a programming language and particularly useful in program transformation. Although there are other solutions, the calculus presented here is a simple solution to getting n-tuples in a typed language. One of the notable benefits of n-tuples in a transformation system is that they allow one to do many program transformations by simple equational reasoning which otherwise would require a meta-language.

Acknowledgements. I would like to thank Valery Trifonov for many helpful discussions.

References

[Aug99] Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 239–250. ACM, June 1999.

[Bar92] H. P. Barendregt. Lambda calculi with types. In D. M. Gabbai Samson Abramski and T. S. E. Maiboum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[HJW92] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.

[Hoo97] Paul Ferenc Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Dept. of Math. and Computing Science, Eindhoven Univ. of Technology, 1997.

[JBM98] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[Mal90a] G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[Mal90b] Grant Malcolm. *Algebraic Types and Program Transformation.* PhD thesis, University of Gronigen, 1990.

[Mee86] L. Meertens. Algorithmics - towards programming as a mathematical activity. In J. W. de Bakker, E. M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986. CWI Monographs, volume 1.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Verlag, June 1991. LNCS 523.

[PM97] Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[Tul00] Mark Tullsen. First class patterns. In E. Pontelli and V. Santos Costa, editors, *Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, volume 1753 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[VMP94] L. S. Van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. *Lecture Notes in Computer Science*, 806:19–61, 1994.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*. Springer Verlag, 1989.