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. Genercity over data types has been achieved with polymorphism. Genercity over type constructors, often referred to as polytypism, is an area of active research. However, genercity over the length of tuples has not been achieved in a typed language. This paper shows the usefulness of such genercity and presents the zip calculus, an extension of a typed lambda calculus that gives genercity over the length of tuples.

1 Introduction

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

In program transformation the need for genercity 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 genercity provided by polymorphism and polytypism is still not adequate for certain programs: another form of genercity is often needed—genercity over the length of tuples. This paper shows the usefulness of "n-tes" (tuples whose lengths are unknown) and proposes a method to extend a programming language with n-tes.

Section 2 gives examples of the usefulness of n-tes. 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

\[
\begin{align*}
\text{zip} & : [a] \rightarrow [b] \rightarrow [(a,b)] \\
\text{zip3} & : [a] \rightarrow [b] \rightarrow [c] \rightarrow [(a,b,c)] \\
\text{zip4} & : [a] \rightarrow [b] \rightarrow [c] \rightarrow [d] \rightarrow [(a,b,c,d)] \\
& \ldots
\end{align*}
\]

whose functionality can be guessed from their types. Also, there are the family of functions unzip, unzip3, unzip4, \ldots and the family of functions zipWith, zipWith3, zipWith4, \ldots To write the zip3, zip4, \ldots 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)\(^2\):

\[
\begin{align*}
\text{map(cross}(f,g)) \cdot \text{zip} &= \text{zip} \cdot \text{cross(map } f, \text{map } g) \\
\text{where} \\
\text{cross } (f,g) (x,y) &= (f \cdot x, g \cdot y)
\end{align*}
\]

The comparable theorem for zip3 is

\[
\begin{align*}
\text{map(cross3}(f,g,h)) \cdot \text{zip3} &= \text{zip3} \cdot \text{cross3(map } f, \text{map } g, \text{map } h) \\
\text{where} \\
\text{cross3 } (f,g,h) (x,y,z) &= (f \cdot x, g \cdot y, h \cdot z)
\end{align*}
\]

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, \ldots And note the following laws for 2-tuples and 3-tuples\(^3\):

\[
\begin{align*}
(fst \cdot x, snd \cdot x) &= x \\
(fst3 \cdot x, snd3 \cdot x, thd3 \cdot x) &= x
\end{align*}
\]

(For which one needs another set of families of functions: fst, fst3, fst4, \ldots and snd, snd3, snd4, \ldots.) 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).

\(^2\) \text{"\cdot" is function composition; "map } f \text{" applies } f \text{ 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:

\[
\begin{align*}
\text{fst } e &= \ldots \hspace{1cm} \text{Prove the case for the} \\
&= e1 \hspace{1cm} \text{"fst" of the tuple.} \\
\text{snd } e &= e2 \hspace{1cm} \text{Wave hands.}
\end{align*}
\]

Thus,

\[
\begin{align*}
\text{e} &= (\text{fst } e, \text{snd } e) = (e1, e2) \hspace{1cm} \text{Make a conclusion about} \\
& \hspace{1cm} \text{the tuple as a whole.}
\end{align*}
\]

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 metalinguage 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 \( e1 = e2 \Leftarrow e3 = e4 \) \cite{HL78} \( (e1, e2, e3, e4) \) 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 \text{fst} and \text{snd} cases. Is it possible to avoid this duplication of derivations? Note that, in general, the form of \((e1, e2)\) is \((C[fst], C[snd])\) \footnote{Where \(C[e] \) represents a program context \(C[]\) with its holes filled by expression \(e\).} (or can be transformed into such a form). So, one would like to merge the two similar derivations

\[
\begin{align*}
\text{fst } e &= \ldots \hspace{1cm} = C[fst] \\
\text{snd } e &= \ldots \hspace{1cm} = C[snd]
\end{align*}
\]

into a single derivation

\[
\begin{align*}
#i \ e &= \ldots \hspace{1cm} = C[#i]
\end{align*}
\]

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 \(F_\omega\), encoded as a Pure Type System (PTS) \cite{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.
\[
e ::= v \\
| \lambda v.t.e \quad \text{variables} \\
| e_1 e_2 \quad \text{abstraction} \\
| \Pi v.t_1.t_2 \quad \text{application} \\
| t \quad \text{type of abstractions} \\
| \ast \quad \text{type of types} \\
| (e_1, e_2, \ldots) t \quad \text{tuple (having type } t \text{)} \\
| m_n \quad \text{projection (} 1 \leq m \leq n \text{)} \\
| n^d \quad \text{dimension} \\
| D \quad \text{type of dimensions} \\
| +_d t \quad \text{sum type} \\
| \text{In}_d \quad \text{constructors for } +_d \{(\text{In}_d, 1, \text{In}_d, 2, \ldots)\} \\
| \text{case}_d \quad \text{destructor for } +_d \\
\]

\[
i ::= e \quad \text{projections (of type } n^d) \\
\]

\[
d ::= e \quad \text{dimensions (of type } D) \\
\]

\[
t ::= e \quad \text{types and kinds (of type } \ast \text{ or } \square) \\
\]

\[
m, n ::= \{\text{natural numbers}\} \\
\]

**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 \(i, d, \text{ and } t\) are used to provide intuition for what is enforced by the type system (but not by the syntax). \(F_{\omega}\) 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 \((m_n : n^d)\) - get the \(m\)-th element of an \(n\)-tuple), their types \((n^d - \text{dimensions}, \text{where } m_n : n^d)\), and "\(D\)" the type of these \(n^d\). And (3) \(n\)-sums made via \(n\)-tuples: for \(n\)-sums \((+_d)(t_1, \ldots, t_n)u\) the constructor family, \(\text{In}_{(n^d)}\), is an \(n\)-tuple of constructors and the destructor \(\text{case}_{(n^d)}\) takes an \(n\)-tuple of functions.

To get the second element of a 3-tuple, a projection is applied to it \((e_1, e_2, e_3)t_2\) 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 \(\Pi\) term): for instance, one can write \((e_1, e_2, e_3)(\Pi i : 3^d. (E_1, E_2, E_3)(\Pi \ell : 3^d. \ast) i)\). Genericity over tuple length is achieved because we can write functions such as "\(\lambda d : D. \lambda i : d. c\)" in which \(d\) can be any dimension \((1^d, 2^d, \ldots)\).

\[5\] Using "\(\_\)" for an unused variable. Also, \(a \rightarrow b\) is used as syntactic sugar for \(\Pi \ell : a. b\)
Reduction Rules:

\[(\lambda v : t. e) e_2 = e_1 \{e_2/v\}\]  \((\beta \text{ reduce})\)

\[(e_1, \ldots, e_n) t_{i_n} = e_i\]  \((\times \text{ reduce})\)

\[\text{case}_d e ((\text{in}_d), e') = e, e'\]  \((+ \text{ reduce})\)

Eta laws:

\[\lambda x : a. e x = e\] if \(e \equiv a \to b, x \notin \text{fv}(e)\)  \((\to \text{ eta})\)

\[\langle e_1, \ldots, e_n \rangle (\Pi i : n^d. A) = e\] if \(e \equiv (\Pi i : n^d. A)\)  \((\times \text{ eta})\)

\[\text{case}_d \text{in}_d e = e\] if \(e \equiv +_d A\)  \((+ \text{ eta})\)

Instantiation:

\[h \cdot \text{case}_d f = \text{case}_d \{^{i^d}h \cdot f, i\}\] if \(h\) strict  \((\text{inst})\)

\[C[\text{case}_d \{^{i^d}\lambda v : t. e\}, x] = \text{case}_d \{^{i^d}\lambda v : t. C[e]\}, x\] if \(C[\cdot\] strict  \((\text{inst})\)

Fig. 2. Laws

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

\[\langle^{i^d}e, i \rangle \equiv \lambda i : d. e\]

\[e, i \equiv e i\]

\[\times_d t \equiv \Pi i : d. t_i\]

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

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

\[\langle^{i^d}e, j \rangle = e \{j/i\}\]  \((\text{n-tuple reduce} i)\)

\[\langle^{i^d}e, i \rangle = e\] if \(e \equiv \times_d A, i \notin \text{fv}(e)\)  \((\text{n-tuple eta})\)

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

\[\langle^{i^d}f, g\rangle (2d \to a \to b), i \langle x, y\rangle (2d \to a), i\]

\[= \langle^{i^d}f, g\rangle (2d \to a \to b), i \langle x, y\rangle (2d \to a), i\] \((\times \text{ eta})\)

\[\langle^{i^d}f, g\rangle (2d \to a \to b), i \langle x, y\rangle (2d \to a), i\] \(2d \to b\)

\[= \langle^{i^d}f, g\rangle (2d \to a \to b), 1_2 \langle x, y\rangle (2d \to a), 1_2\] \((\text{n-tuple reduce}, 2x)\)

\[\langle^{i^d}f, g\rangle (2d \to a \to b), 1_2 \langle x, y\rangle (2d \to a), 1_2\]

\[= \langle f, g\rangle (2d \to a \to b), 1_2 \langle x, y\rangle (2d \to a), 1_2\]

\[= \langle f, g\rangle (2d \to a \to b), 2_2 \langle x, y\rangle (2d \to a), 2_2\] \((\times \text{ reduce}, 4x)\)

\[= \langle f, g\rangle (2d \to b)\]

The tuples \(f, g\) and \(x, y\) are “zipped” together, this is why it is called the zip calculus.
\[\begin{align*}
\Gamma \vdash a : A, \quad &\Gamma \vdash B : s, \quad A \equiv B \\
\Gamma \vdash a : B \quad &\text{(conv)} \\
\Gamma \vdash A : s \\
\Gamma, x : A \vdash x : A \quad &\text{(var)} \\
\Gamma \vdash f : (\Pi x : A.B), \quad &\Gamma \vdash a : A \quad \text{(app)} \\
\Gamma \vdash f a : B(a/x) \\
\Gamma \vdash b : B, \quad &\Gamma \vdash (\Pi x : A.B) : t \quad \text{(lam)} \\
\Gamma, x : A \vdash b : B \\
\Gamma, x : A.B : t, \quad (s,t,u) \in \mathcal{R} \\
\Gamma \vdash (\Pi x : A.B) : u \quad &\text{(pi)} \\
\end{align*}\]

Fig. 3. Type Judgments for a Pure Type System

\[\forall j \in \{1..n\}, \quad \Gamma \vdash a_j : A(j/i), \quad \Gamma \vdash (\Pi i : n^d.A) : t \quad \text{(tuple)}\]

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 \Pi 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 \(\equiv_{\beta}\) in the judgment (conv) is beta-equivalence (alpha-equivalent terms are identified).

In the case of the zip calculus, the set of sorts is \(\mathcal{S} = \{1^d, 2^d, \ldots\} \cup \{*, \square, D\}\), the set of constants is \(\mathcal{C} = \mathcal{S} \cup \{m_n | 1 \leq m \leq n\}\), and the axioms \(\mathcal{A}\) and rules \(\mathcal{R}\) are as follows:

\[
\begin{array}{c|c|c}
\mathcal{A} & \mathcal{R} \\
\hline
\star : \square & \lambda v_t : t.e \\
\ h_n : n^d & (\square, *, *) \\
n^d : D & \lambda v_i : T . e \\
D : \square & (\square, \square, \square) \\
\end{array}
\]

The \(\mathcal{R}\) rules indicate what lambda abstractions are allowed (which is the same as saying which \(\Pi\) 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 (\(e - \text{terms}, t - \text{types}, T - \text{kinds}, i - \text{projections}, d - \text{dimensions}, v_x - \text{variable in class } x\)).

In the zip calculus there is an additional term, \(\langle e_1, e_2, \ldots \rangle t\), 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 \(\equiv_{\beta}\) relation in the (conv) judgment must be extended to include not just (\(\beta\) reduce) but also (\(\times\) reduce) and (\(\times\) eta).

To get generic sums, one needs only add + as a constant and the following two primitives:
\[ \begin{align*} 
\Gamma \vdash f : \rightarrow (\Pi x : A . B), \quad & \quad \Gamma \vdash a : A', \quad A =_{\beta} A' \quad \text{(app)} \\
\Gamma \vdash a : B(a/x) \quad & \quad x : A \in \Gamma \quad \text{(var)} \\
\Gamma, x : A \vdash b : B, \quad & \quad \Gamma \vdash (\Pi x : A . B) : t \quad \text{(lam)} \\
\Gamma \vdash (\lambda x : A . b) : (\Pi x : A . B) \quad & \quad c : s \in \mathcal{A} \quad \text{(axiom)} \\
\Gamma \vdash A \rightarrow s, \quad & \quad \Gamma, x : A \vdash B : \rightarrow t, \quad (s, t, u) \in \mathcal{R} \quad \text{(pi)} \\
\Gamma \vdash (\Pi x : A . B) : u \quad & \quad \Gamma \vdash a : A, \quad A \rightarrow_{\beta} B \quad \text{(red)} \\
\end{align*} \]

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

\[ \forall j \in \{1..n\}, \quad \Gamma \vdash a_2 : A[j_2/i], \quad \Gamma \vdash (\Pi i : n^2 . A) : t \quad \text{(tuple)} \]

\[ \Gamma \vdash (a_1, \ldots, a_n) : (\Pi i : n^2 . A) : (\Pi i : n^2 . A) \]

\[ \Gamma \vdash a : A, \quad A \rightarrow_{\beta} B \quad \text{(red') } \]

\[ \Gamma \vdash f : \rightarrow C, \quad \Gamma \vdash a : \rightarrow \rightarrow B \]

\[ \Gamma \vdash f a : B(a/x) \quad \text{(app') } \]

**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 \( A \) 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, \( A \) 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 “\( \Gamma \vdash x :\rightarrow X \)” and “\( \rightarrow_{\beta} \)” 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 \( \rightarrow_{\beta} \) is \( \rightarrow_{\beta} \) extended with \( (\times \text{ reduce}) \) and \( =_{\eta} \) is equality up to \( (\times \text{ eta}) \) convertibility. The intuition for the change of (app) is that \( f \) may evaluate to

\[ (\Pi x : a_1.b_1, \ldots, \Pi x : a_n.b_n)t \]

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

\[ \Pi x : (a_1, \ldots, a_n)(n^d \rightarrow \times) i . (b_1, \ldots, b_n)(n^d \rightarrow \times) i \]

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 \((x_1, \ldots, x_n)t\); \(m\) is put for the projection \(m_n\); the dimension \(d\) is dropped from \(x_d\); and when applying dimensions and types, "\(f_{d_1,t_1,t_2}\)" is put for "\(f_{d_1}l_1t_2\)". 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 \(*\)), \(i, j, k, l\) for projections (terms of type \(n^d\)), and \(d, I, J, K, L\) for dimension variables (terms of type \(D\)). (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:

\[
\begin{align*}
\text{zip3} \ &: \ [(a), [b], [c]] \to [(a, b, c)] \\
\text{zip3} \ (a:as, b:bs, c:cs) &= (a, b, c) \ : \ \text{zip3} \ (as, bs, cs) \\
\text{zip3} \ _\ &= []
\end{align*}
\]

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

\[
\begin{align*}
\text{zip} \ &: \ \times(\times[a]) \to \times[a] \\
\text{zip} \ (\langle x, i : xs, i \rangle) &= x : \text{zip} \ xs \\
\text{zip} \ _\ &= []
\end{align*}
\]

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 \texttt{seqTupleMaybe} is added:

\[
\text{seqTupleMaybe} \ &: \ \times(\times[a, i] \to \text{Maybe} \ b, i) \to \times[a] \to \text{Maybe}(\times[b])
\]

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

\[
\begin{align*}
\text{map}(\text{cross3}(f, g, h)) \cdot \text{zip3} &= \text{zip3} \cdot \text{cross3}(\text{map} f, \text{map} g, \text{map} h) \\
\text{where} \\
\text{cross3} \ (f, g, h) \ (x, y, z) &= (f x, g y, h z)
\end{align*}
\]

can be generalized in the zip calculus to this:

\[
\begin{align*}
\text{map}(\text{cross}_d f) \cdot \text{zip} &= \text{zip} \cdot \text{cross}_d (\times_i^d \text{map} f, i) \\
\text{where} \\
\text{cross}_d f \ x &= (\times_i^d f, i, x)
\end{align*}
\]

And this law

\[
\langle x_1, x_2, x_3 \rangle = x
\]

can be generalized to the \((n\text{-tuple eta})\) law:

\[
(\times_i^d x, i) = x
\]
4.3 More General Derivations

Remember this example from section 2.3?

\[ \text{fst } e \]
\[ = \ldots \]
\[ = C[\text{fst}] \]

Similarly, \( \text{snd } e = C[\text{snd}] \)

Thus,
\[ e = (\text{fst } e, \text{snd } e) = (C[\text{fst}], C[\text{snd}]) \]

Now a generic transformation can be done handily:

\[ e \]
\[ = \langle \ldots \rangle \]
\[ = \langle C[.] \rangle \]

The \textit{fst} and \textit{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 \( f \bar{x} \) for the “vector” \( \langle f \bar{x}_1, \ldots, f \bar{x}_n \rangle \) but now \( g(f \bar{x}) \) could signify either \( \langle g(f \bar{x}_1), \ldots, g(f \bar{x}_n) \rangle \) or \( g(f \bar{x}_1, \ldots, f \bar{x}_n) \). 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 \( \langle -^d e \rangle \) is a tuple of identical elements):

\[
\text{map3matrix}_{a, b, i, j, K} :: (a \rightarrow b) \rightarrow \times_{i \leq j} \langle \times_{i \leq j} \langle \times_{i \leq K} a \rangle \rangle \rightarrow \times_{i \leq j} \langle \times_{i \leq K} b \rangle \\
\text{map3matrix}_{a, b, i, j, K} = \lambda f. \lambda m. \langle i : j \{ k : K \ f \ m_{i,j,k} \} \rangle
\]

The expression \( \langle i : j \{ k : K \ e \} \rangle \) 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 “\( m_{i,j,k} \)”. Matrix transposition is straightforward:

\[
\text{transpose}_{i, j, a} :: \times_{i \leq j} \langle \times_{i \leq j} a_{i,j} \rangle \rightarrow \times_{i \leq j} \langle \times_{j \leq i} a_{i,j} \rangle \\
\text{transpose}_{i, j, a} = \lambda x. \langle i : j \{ i : i \ x_{i,j} \} \rangle
\]

The transpose is done by “reversing” the subscripts of \( x \). Note that the type variable \( a \) above is a \textit{matrix} of types and, for any \( n \), \text{transpose} could be applied to a tuple of n-tuples. An application of \text{transpose} is reduced as follows:

\[
\text{(transpose}_{3d, 2d, a} \ ((x_1, x_2), (y_1, y_2), (z_1, z_2))) \rightarrow (\langle i : i \ (\langle (x_1, x_2), (y_1, y_2), (z_1, z_2) \rangle_{i,j}), \ (z_1, z_2)) \rightarrow (\langle (x_1, x_2), (y_1, y_2), (z_1, z_2) \rangle_{i,j}, \ z_2)
\]

\[ \text{z2} \]

9
Note the various ways one can transform a two-dimensional matrix:

\[
\begin{align*}
\langle \langle m \rangle_{i,j} \rangle & \quad \text{m itself} \\
\langle \langle \langle m \rangle_{i,j} \rangle \rangle & \quad \text{the transpose of m} \\
\langle \langle \langle f \ m \rangle_{i,j} \rangle \rangle & \quad \text{f applied to each element of m} \\
\langle \langle \langle f \ \langle m \rangle_{i,j} \rangle \rangle \rangle & \quad \text{f applied to each “row” of m} \\
\langle \langle \langle f \ \langle m \rangle_{i,j} \rangle \rangle \rangle & \quad \text{f applied to each “column” of m}
\end{align*}
\]

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):

\[
\text{transpose}_{j,i,b}(\text{transpose}_{i,j,a} m) \\
= \langle \langle k \ \langle \langle m \rangle_{i,j} \rangle_{k,l} \rangle \rangle \\
= \langle \langle k \ \langle \langle m \rangle_{i,k} \rangle_{l,l} \rangle \rangle \\
= \langle \langle k \ \langle m \rangle_{i,k} \rangle \rangle \\
= \langle \langle m \rangle \rangle \\
= m
\]

A law, called Abides, is

\[
\text{case } \langle \lambda x. (a1, b1), \lambda y. (a2, b2) \rangle \ x \\
= \langle \text{case } \lambda x. a1, \lambda y. a2 \rangle \ x \ , \ \langle \text{case } \lambda x. b1, \lambda y. b2 \rangle \ x
\]

and its derivation is

\[
\text{case } \langle \lambda x. (a1, b1), \lambda y. (a2, b2) \rangle \ x \\
= \langle \langle \text{case } \lambda x. (a1, b1), \lambda y. (a2, b2) \rangle \ x \rangle \ _1 \\
\quad \langle \text{case } \lambda x. (a1, b1), \lambda y. (a2, b2) \rangle \ x \rangle \ _2 \\
= \langle \langle \text{case } \lambda x. (a1, b1) \rangle \ _1, \lambda y. (a2, b2) \rangle \ x \rangle \\
\quad \langle \text{case } \lambda x. (a1, b1) \rangle \ _2, \lambda y. (a2, b2) \rangle \ x \rangle \\
= \langle \langle \text{case } \lambda x. a1, \lambda y. a2 \rangle \ x \rangle \\
\quad \langle \text{case } \lambda x. b1, \lambda y. b2 \rangle \ x \rangle
\]
Here is a generic version of Abides

\[
\text{case } (\langle \lambda y. (\langle m, i, j \rangle y) \rangle x = (\langle \text{case } (\langle \lambda y. m, i, j \rangle y) \rangle x)
\]

and its derivation is

\[
\begin{align*}
\text{case } & (\langle \lambda y. (\langle m, i, j \rangle y) \rangle x = (\langle \text{case } (\langle \lambda y. (\langle m, i, j \rangle y) \rangle x) \rangle) \\
& = (\langle \text{case } (\langle \lambda y. (\langle m, i, j \rangle y) \rangle x) \rangle_{i}) \\
& = (\langle \text{case } (\langle \lambda y. (\langle m, i, j \rangle y) \rangle x) \rangle_{i}) \\
& = (\langle \text{case } (\langle \lambda y. m, i, j \rangle y \rangle x) \rangle \\
\text{\{n-tuple \eta\}} \\
\text{\{inst\}} \\
\text{\{n-tuple reduce\}}
\end{align*}
\]

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 `μ_{(n)}` is \((\times (\langle n \rangle^d) \to (\times (\langle n \rangle^d))) \to (\times (\langle n \rangle^d))\) (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:

\[
\begin{align*}
\text{in}_F & : \quad F(\mu F) \to \mu F & \text{original} \\
\text{in}_{F,F} & : \quad \times (\langle i_1 \rangle (F(\mu F)),i) \to (\mu F)_i & \text{generic} \\
\text{out}_F & : \quad \mu F \to F(\mu F) & \text{original} \\
\text{out}_{F,F} & : \quad \times (\langle i_1 \rangle (\mu F),i) \to (F(\mu F),i) & \text{generic}
\end{align*}
\]

From these a more generic `cata` can be defined:\footnote{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.}

\[
\begin{align*}
\text{cata}_{F,a} & : (F \to a) \to (\mu F \to a) & \text{original} \\
\text{cata}_{F,F,a} & : (\times (\langle i_1 \rangle (F(a)),i) \to a,i) \to (\times (\langle i_1 \rangle (\mu F),i) \to a,i) & \text{generic} \\
\text{cata}_{F,a} \varphi & = \text{fix } \lambda f. F \cdot f \cdot \text{out}_F \\
\text{cata}_{F,F,a} \varphi & = \text{fix } \lambda f. \langle i_1 \rangle \varphi_1 \cdot (\times (\langle i_1 \rangle f),i) \cdot (\text{out}_{F,F},i) & \text{generic}
\end{align*}
\]
So, \( \text{cata}_{(n^a), F, a} \) 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 \( \star \to \star \):

\[
\begin{align*}
\text{id} &= F \text{id} \\
F \, f \cdot F \, g &= F \, (f \, g)
\end{align*}
\]

can be generalized to functors of kind \( x^{(\neg^I \star)} \to x^{(\neg^I \star)} \):

\[
\begin{align*}
\neg^I \text{id} &= F \, \neg^I \text{id} \\
\langle x^I \rangle (F \, f, j, \cdot (F \, g, j)) &= F \, \langle x^I \rangle \, f, i, \cdot g, i
\end{align*}
\]

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 \( x^{(a)} \approx a \) (the bijections being \( \lambda x. \, x \cdot 1 \) and \( \lambda x. \, x \cdot (x) \)).

## 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:

\[
\begin{align*}
\text{tupleToList} & : \ x^{(\neg^I a)} \to \text{list} \, a \\
\text{seqTupleL, seqTupleR} & : \text{Monad} \, m \Rightarrow x^{((a, i \to m \, b, i))} \to x \to m \,(x b)
\end{align*}
\]

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

\[
\begin{align*}
\text{curry2} & : (a \to b \to c) \to (a, b) \to c \\
\text{curry3} & : (a \to b \to c \to d) \to (a, b, c) \to d \\
\ldots
\end{align*}
\]

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


