Multiple Values in Standard ML

Kevin Mitchell
LFCS, Department of Computer Science
University of Edinburgh

Abstract

ML has traditionally used pairs, tuples or currying to handle functions requiring more than one argument. Tuples are also used when a function needs to return multiple results. Compilers must then take care to optimise function calls if the resulting programs are to achieve a performance comparable to languages that support multiple arguments and results. In this paper we argue that it is difficult for a compiler to do this job entirely satisfactorily because the intended pattern of usage of a function is usually unknown at the time it is declared. We argue for the introduction of sequences into the language. Each expression evaluates to a sequence of values. Such sequences are not values and so cannot be bound to a variable. They can however be bound to a sequence of variables or, more generally, patterns of the same length. We demonstrate how this change can allow the programmer to help the compiler generate more efficient code. We also use this extension to justify an optimisation used by a number of ML compilers.

1 The Problem

We take as our starting point a simple stack based model for evaluating ML expressions. From this viewpoint, the primary purpose of evaluating an expression is to push a value onto the stack. To evaluate an expression such as exp_1 exp_2 eagerly we must first evaluate exp_1. This leaves (the representation of) a function on the stack. Evaluating exp_2 then pushes the value of the argument on the stack. Finally, after removing these two values, the body of the function is evaluated in the appropriate environment. The uniformity of function application, where a single value is always passed as argument and returned as the result, supports the smooth integration of higher-order functions and polymorphism. This simple model is at the heart of many ML implementations, although the details often vary for efficiency reasons. For example, registers may be used to cache the top few stack elements to reduce memory accesses.

Consider the expression f(exp_1, exp_2). Syntactically this looks similar to a function application in most imperative languages. However, as all functions in SML take a *single* argument, this should be read as the application of f to the pair resulting from the evaluation of (exp_1, exp_2). The equivalent expression in a language such as C would be viewed as the application of f to the *two* arguments resulting from

the evaluation of exp_1 and exp_2. This difference can be seen more clearly in the equivalent Standard ML expression

let val a = (exp_1, exp_2) in f a end

(assuming a is fresh). A naive implementation of this application would therefore build a pair as a result of evaluating (exp_1, exp_2) and pass this, via the stack, to the body of f. What happens if the definition of f is of the form

fun f (x,y) = ... ?

Suppose that exp_1 evaluates to v_1 and exp_2 to v_2, and so the argument passed to f consists of the pair (v_1,v_2). Matching the pattern (x,y) against the pair would result in a binding of x to v_1 and y to v_2. Clearly this is inefficient as we have constructed a pair and then immediately discarded it. A better solution would be to pass v_1 and v_2 separately on the stack. If at some other point in the program the function f was applied to a previously constructed pair, p say, then we can generate code to separate the components of p on the stack *before* the call.

Unfortunately the problem is more subtle than this. Consider a function of the form

fun g p = ...

having the same type as f, i.e. where p will be bound to a pair upon application. It may be inefficient to treat g in the same way as f. Suppose we choose to pass both components of the argument pair separately and consider the case where g is applied to a pair that already exists. We would take the pair apart, call g, and the body of g would then need to rebuild the pair in order to bind it to p. The function f naturally expects two arguments, whereas g expects one. However, the type system makes no distinction between them. A compiler, of course, is free to make more distinctions than a type system. It can call f and g using different calling conventions as long as it never gets confused.

Unfortunately, higher-order functions and functors complicate matters. For example, suppose we define a function

fun h a = a(1,2)

Both f and g can be passed as arguments to h, assuming that they take pairs of integers as arguments. When compiling the code for h, how can we know which convention to use when applying a to its argument? If h is to be compiled, rather than interpreted, it seems that we must fix the calling convention at compile time. This implies that all arguments to h must agree on how they receive their arguments. An obvious solution is to have *two* versions of the function f, one expecting the two arguments to be passed separately, and the other expecting a pair. This is not as extravagant as it might at first seem as the second function, the "slow entry point" version, can just split the pair and then 'tail-apply' to the first version. Whenever f is passed as an argument we use the slow entry point version for uniformity and when

it is applied directly to an argument we can use the fast entry point code. The net effect of this strategy is to remove the tuple creation overhead in the common case of applying a "visible" function to a locally constructed tuple argument. However, if higher-order functions or functors are used extensively then this optimisation is less effective. A simple example of where temporary tuples are still created is the zipmap function, the composition of a zip followed by a map:

```
fun zipmap(F, [], []) = []
  | zipmap(F, h1::t1, h2::t2) = F(h1,h2) :: (zipmap(F, t1, t2))
```

Each application of F to (h1,h2) is likely to involve the creation of an intermediate tuple. Dataflow analysis can be used to suggest that it would be more efficient if the functional argument to zipmap received its two arguments separately. But what if zipmap itself was passed as an argument to another function? Perhaps we should distinguish between functions that expect tuples and those that expect multiple values at the type level.

Temporary tuples are not just created when calling functions. Returning multiple results using tuples can create similar problems. Consider the function

$$\mathsf{fun\ f\ \ldots} = (exp_1, exp_2)$$

If the result of a call to this function is immediately bound to a pair pattern, as in val (x,y) = f ..., then it is likely that a temporary pair cell will be created. This could be avoided if f returned two values on the stack rather than a pair. However, if at some other point in the program the result of a call to the function was bound to a single variable then the caller would need to construct a pair from the result first. We can make a case for supporting functions with different return conventions just as we did for the calling conventions. The use of higher-order functions or functors complicates this case as well.

The problems discussed so far impose a time penalty if not optimised. Functions requiring multiple values can also lead to a space overhead in the particular case of constructor functions. Consider the datatype declaration

```
datatype ... = ...  |  C of int * bool
```

We assume that the other constructors are sufficiently complex that the C constructor must explicitly tag any values it is applied to. What does a value built using the C constructor look like? There are two obvious choices. The traditional approach is to treat C as taking a (pointer to) an object, in this case a pair. The result is a new cell containing a tag for C plus the pointer, as illustrated in Figure 1.[1] This representation is appropriate if we frequently apply C to pairs that are not created immediately prior to the application, and match the results against patterns where the enclosed pair is just bound to a variable. But what about the more usual case where the constructor application is immediately adjacent to the construction of

[1] We are ignoring any additional garbage collector overhead here.

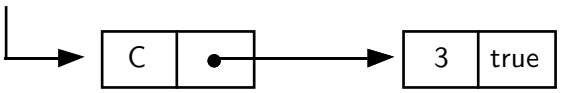

Figure 1: Indirect representation

the pair, and where pattern matching breaks the pair apart? In this case it would be more efficient to use the following representation, avoiding the construction of a pair.

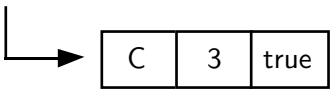

Figure 2: Embedded representation

In the (hopefully) rare cases where we apply C to an existing pair then space will be wasted due to a loss of sharing. Furthermore if we match the constructed value against a pattern such as C x then we will have to build a pair to bind to x. It is often hard for an ML compiler to employ dataflow analysis to choose between these two approaches due to the interactive nature of the language. Compilers usually just choose one or other representation and then stick to it. As we have seen, this can result in both space and time inefficiencies.

In all of these cases, if we could distinguish between multiple values and tuples in the type system then the user could indicate the intended usage patterns more precisely. The new types could also be hidden from the user, with heuristics being used by the compiler to determine when to treat tuples as multiple values and vice versa. In this case the extension to the language and type system could be used to prove that these optimisations at least preserve equivalence of results, even if they cannot guarantee to always improve performance in the general case.

We are certainly not the first to advocate making such distinctions. In the language Amber [2] tuples may not be bound to variables or embedded within data structures. In Zinc, a variant of ML [3], datatype constructors are defined to take multiple arguments, rather than tuples. In both cases records are used to build compound data values.

In the remainder of the paper we describe a simple language that contains both tuples and multiple values. We present a dynamic semantics for the language, in the style of the SML semantics [4], followed by a static semantics and consistency proof. We use the language to express and validate some of the optimisations hinted at in this introduction, and show how the system can be exploited by the programmer to optimise his programs. Finally we discuss a range of extensions that could be made to the language to support more advanced compiler heuristics.

2 The Language

Our case study will make heavy use of sequences, both at the semantic and object language levels, and so we start by defining some notation. We will use ε to indicate the empty sequence and the sequence resulting from adding the term t at the end of the sequence $\overrightarrow{t}$ will be represented by $\overrightarrow{t} \cdot t$. Sequence concatenation will be indicated by juxtaposition, as in $\overrightarrow{t_1}\overrightarrow{t_2}$. Finally, the term $\langle t_1, t_2, \ldots t_n \rangle$ will be used as an abbreviation for $\varepsilon \cdot t_1 \cdot t_2 \cdot \ldots \cdot t_n$.

The language chosen for this experiment is a simple functional language extended with pairs and sequences. The set of base constants is unimportant and is left undefined. The syntax of the language is defined by:

Constant	x		
Variable	x		
Expression	e	::=	$\mathrm{c} \mid x \mid \lambda \overrightarrow{p} \,.\, e \mid e_1\, e_2 \mid (e_1, e_2) \mid \varepsilon \mid e_1 \cdot e_2$
Pattern	p	::=	$x \mid (p_1, p_2)$
SPattern	$\overrightarrow{p}$	::=	$\varepsilon \mid \overrightarrow{p} \cdot p$

The intention is that the evaluation of an expression can yield a *sequence* of values. The expression ε will evaluate to the empty sequence. Furthermore, if the expression e_1 evaluates to $\overrightarrow{v}$ and e_2 to $\langle v \rangle$ then the compound expression $e_1 \cdot e_2$ will evaluate to the sequence $\overrightarrow{v} \cdot v$. Physically, these sequences will be represented by consecutive elements on the evaluation stack. A sequence of values is *not* a value. In particular, we will not be able to bind a sequence to a variable, or embed it in other data structures. To manipulate a sequence as a value would require moving the data from the stack to the heap (so that we can point to it). We will be able to do this, but the result will be a tuple of values, not a sequence. A function can be applied to a sequence of values. However, as we don't want to bind names to sequences, we must extend the notion of a pattern to a sequence of patterns in the obvious way.

2.1 The Dynamic Semantics

For simplicity, we present a semantics based on environments with judgments of the form $E \vdash e \Rightarrow \overrightarrow{v}$. This is to be read as "the expression e, when evaluated in the context of the environment E, can yield the sequence of values $\overrightarrow{v}$ ". Before presenting the rules that define this judgment, we must first define the terms *environment* and *value* in the usual way.

Value	v	::=	**Constant** $\cup$ $\{ [\overrightarrow{p}, e, E] \}$ $\cup$ $\{ (v_1, v_2) \}$
Value*	$\overrightarrow{v}$	::=	$\varepsilon \mid \overrightarrow{v} \cdot v$
Environment	E	::=	**Variable** $\xrightarrow{\text{fin}}$ **Value**

Note that an environment maps variables to values, not sequences of values. Environments can be combined, as in $E_1 E_2$. In the resulting environment the bindings in E_2 take priority whenever there is a conflict.

The rules defining the evaluation of an expression are presented in Figure 3. Some of the rules require an expression to evaluate to a single value, such as e_1 in the **fun** rule. We also demand that an argument produces as many values as the function expects, rather than at least as many. There is obvious scope for weakening some of these conditions and we will return to this point in Section 5.

const: $\dfrac{}{E \vdash c \Rightarrow \langle c \rangle}$ **var:** $\dfrac{}{E \vdash x \Rightarrow \langle E(x) \rangle}\ x \in \text{dom}\, E$

fun: $\dfrac{}{E \vdash \lambda \vec{p}\,.e \Rightarrow \langle [\vec{p}, e, E] \rangle}$ **pair:** $\dfrac{E \vdash e_1 \Rightarrow \langle v_1 \rangle \quad E \vdash e_2 \Rightarrow \langle v_2 \rangle}{E \vdash (e_1, e_2) \Rightarrow \langle (v_1, v_2) \rangle}$

appl(1): $\dfrac{E \vdash e_1 \Rightarrow \langle [\vec{p}, e, E'] \rangle \quad E \vdash e_2 \Rightarrow \vec{v_2} \qquad \vec{p}, \vec{v_2} \Rightarrow E'' \qquad E'E'' \vdash e \Rightarrow \vec{v}}{E \vdash e_1\ e_2 \Rightarrow \vec{v}}$

appl(2): $\dfrac{E \vdash e_1 \Rightarrow \langle c \rangle \qquad E \vdash e_2 \Rightarrow \vec{v_2} \qquad \text{APPLY}(c, \vec{v_2}) = \vec{v}}{E \vdash e_1\ e_2 \Rightarrow \vec{v}}$

seq(1): $\dfrac{}{E \vdash \varepsilon \Rightarrow \varepsilon}$ **seq(2):** $\dfrac{E \vdash e_1 \Rightarrow \vec{v_1} \quad E \vdash e_2 \Rightarrow \langle v_2 \rangle}{E \vdash e_1 \cdot e_2 \Rightarrow \vec{v_1} \cdot v_2}$

- - - - -

spat(1): $\dfrac{}{\varepsilon, \varepsilon \Rightarrow \{\}}$ **spat(2):** $\dfrac{\vec{p}, \vec{v} \Rightarrow E_1 \quad p, v \Rightarrow E_2}{\vec{p} \cdot p, \vec{v} \cdot v \Rightarrow E_1 E_2}$

- - - - -

pat(1): $\dfrac{}{x, v \Rightarrow \{(x, v)\}}$ **pat(2):** $\dfrac{p_1, v_1 \Rightarrow E_1 \quad p_2, v_2 \Rightarrow E_2}{(p_1, p_2), (v_1, v_2) \Rightarrow E_1 E_2}$

Figure 3: The dynamic semantics.

Notice how the result of e_2 in the expression $e_1 \cdot e_2$ is constrained to be a single value. Thus e and $\langle e \rangle$ are equivalent only if e evaluates to a single value.

2.2 The Static Semantics

For simplicity we restrict our attention to a monomorphic semantics. We assume a set of base types suitable for typing the constants of our language. A type is then either a base type, a function type or a pair type. An expression evaluates to a sequence of values, and so the static semantics will associate a sequence of types

with an expression.

Base Type	b		
Type	τ	::=	$b \mid \overrightarrow{\tau_1} \to \overrightarrow{\tau_2} \mid \tau_1 * \tau_2$
Type*	$\overrightarrow{\tau}$	::=	$\varepsilon \mid \overrightarrow{\tau} \cdot \tau$
Type Environment	T	::=	**Variable** $\xrightarrow{\text{fin}}$ **Type**

We associate with each type a set of values. We assume that $v \in \tau$ has already been defined for the base types, and will lift this to sequences in the obvious way.

Definition 2.1

$$\tau_1 * \tau_2 = \{(v_1, v_2) \mid v_1 \in \tau_1 \wedge v_2 \in \tau_2\}$$

$$\begin{aligned}\overrightarrow{\tau_1} \to \overrightarrow{\tau_2} &= \{c \mid \forall \overrightarrow{v_1} \in \overrightarrow{\tau_1} \cdot \forall \overrightarrow{v_2} \cdot (APPLY(c, \overrightarrow{v_1}) = \overrightarrow{v_2}) \supset \overrightarrow{v_2} \in \overrightarrow{\tau_2}\} \\ &\cup \{[\overrightarrow{p}, e, E] \mid \forall \overrightarrow{v_1} \in \overrightarrow{\tau_1} \cdot \exists E' \cdot \overrightarrow{p}, \overrightarrow{v_1} \Rightarrow E' \wedge \forall \overrightarrow{v_2} \cdot (EE' \vdash e \Rightarrow \overrightarrow{v_2}) \supset \overrightarrow{v_2} \in \overrightarrow{\tau_2}\}\end{aligned}$$

We will also need to lift the definition of $v \in \tau$ pointwise to environments, written $E \in T$. The static semantics is presented in Figure 4.

const: $\dfrac{}{T \vdash c : \langle \tau \rangle} \; c \in \tau$ **var:** $\dfrac{}{T \vdash x : \langle T(x) \rangle} \; x \in \text{dom } T$

fun: $$\frac{\overrightarrow{p}, \overrightarrow{\tau_1} : T' \qquad TT' \vdash e : \overrightarrow{\tau_2}}{T \vdash \lambda \overrightarrow{p} \,.\, e : \langle \overrightarrow{\tau_1} \to \overrightarrow{\tau_2} \rangle}$$

appl: $\dfrac{T \vdash e_1 : \langle \overrightarrow{\tau_2} \to \overrightarrow{\tau} \rangle \quad T \vdash e_2 : \overrightarrow{\tau_2}}{T \vdash e_1 \, e_2 : \overrightarrow{\tau}}$ **pair:** $\dfrac{T \vdash e_1 : \langle \tau_1 \rangle \quad T \vdash e_2 : \langle \tau_2 \rangle}{T \vdash (e_1, e_2) : \langle \tau_1 * \tau_2 \rangle}$

seq(1): $\dfrac{}{T \vdash \varepsilon : \varepsilon}$ **seq(2):** $\dfrac{T \vdash e_1 : \overrightarrow{\tau_1} \quad T \vdash e_2 : \langle \tau_2 \rangle}{T \vdash e_1 \cdot e_2 : \overrightarrow{\tau_1} \cdot \tau_2}$

- - - - -

spat(1): $\dfrac{}{\varepsilon, \varepsilon : \{\}}$ **spat(2):** $\dfrac{\overrightarrow{p}, \overrightarrow{\tau} : T_1 \quad p, \tau : T_2}{\overrightarrow{p} \cdot p, \overrightarrow{\tau} \cdot \tau : T_1 T_2}$

- - - - -

pat(1): $\dfrac{}{x, \tau : \{(x, \tau)\}}$ **pat(2):** $\dfrac{p_1, \tau_1 : T_1 \quad p_2, \tau_2 : T_2}{(p_1, p_2), \tau_1 * \tau_2 : T_1 T_2}$

Figure 4: The static semantics.

We now turn our attention to whether the static and dynamic semantics agree with each other. We start by stating a few simple lemmas connected with pattern matching.

Lemma 2.1 $p, \tau : T \wedge \nu \in \tau \supset \exists E \cdot p, \nu \Rightarrow E \wedge E \in T$

Lemma 2.2 $\vec{p}, \vec{\tau} : T \wedge \vec{\nu} \in \vec{\tau} \supset \exists E \cdot \vec{p}, \nu \Rightarrow E \wedge E \in T$

Lemma 2.3 $\vec{p}, \vec{\nu} \Rightarrow E_1 \wedge \vec{p}, \vec{\nu} \Rightarrow E_2 \supset E_1 = E_2$

The proofs of these lemmas are straightforward. We now state and prove the consistency theorem.

Theorem 2.4 (Consistency) $E \in T \ \wedge \ E \vdash e \Rightarrow \vec{\nu} \ \wedge \ T \vdash e : \vec{\tau} \ \supset \ \vec{\nu} \in \vec{\tau}$

Proof We use induction on the depth of inference of $T \vdash e : \vec{\tau}$. Consider the different possible cases for the last step in this proof:

1. The **const** rule was used, i.e. $e \equiv c$, $T \vdash c : \langle \tau \rangle$, where $c \in \tau$ and $\vec{\tau} = \langle \tau \rangle$. Now $E \vdash e \Rightarrow \langle c \rangle$ and $\langle c \rangle \in \langle \tau \rangle$.

2. The **var** rule was used, i.e. $e \equiv x$, $T \vdash x : \langle T(x) \rangle$ where $x \in \text{dom } T$ and $\vec{\tau} = \langle T(x) \rangle$. Now $E \in T$ and so $E(x) \in T(x)$, i.e. $\langle E(x) \rangle \in \langle T(x) \rangle$. But $E \vdash x \Rightarrow \langle E(x) \rangle$ as required.

3. The **fun** rule was used, i.e. $e \equiv \lambda \vec{p} \,.\, e'$. The type sequence $\vec{\tau}$ must be of the form $\langle \vec{\tau_1} \rightarrow \vec{\tau_2} \rangle$. Furthermore, we must have smaller proofs of $\vec{p}, \vec{\tau_1} : T'$ and $TT' \vdash e' : \vec{\tau_2}$ for some T'. The value $\vec{\nu}$ is constrained to be $\langle [\vec{p}, e', E] \rangle$ and so we must prove that $[\vec{p}, e', E] \in \vec{\tau_1} \rightarrow \vec{\tau_2}$. Suppose we have a sequence of values $\vec{\nu_1} \in \vec{\tau_1}$. By Lemma ?, there exists a unique E' such that $\vec{p}, \vec{\nu_1} \Rightarrow E'$ and $E' \in T'$. Suppose $EE' \vdash e' \Rightarrow \vec{\nu_2}$ for some $\vec{\nu_2}$. Now $EE' \in TT'$ and so by the induction hypothesis we have $\vec{\nu_2} \in \vec{\tau_2}$ as required.

4. The **appl** rule was used, i.e. $e \equiv e_1 \ e_2$ and we have smaller proofs of $T \vdash e_1 : \langle \vec{\tau_2} \rightarrow \vec{\tau} \rangle$ and $T \vdash e_2 : \vec{\tau_2}$ for some $\vec{\tau_2}$.

   There are two cases to consider:

   (a) The **appl(1)** rule was used in the dynamic semantics, i.e. $E \vdash e_1 \Rightarrow \langle [\vec{p}, e', E'] \rangle$, $E \vdash e_2 \Rightarrow \vec{\nu_2}$, $\vec{p}, \vec{\nu_2} \Rightarrow E''$ and $E'E'' \vdash e' \Rightarrow \vec{\nu}$. We may use the induction hypothesis to deduce that $[\vec{p}, e', E'] \in \vec{\tau_2} \rightarrow \vec{\tau}$ and $\vec{\nu_2} \in \vec{\tau_2}$. From Definition Definition 2.1 and Lemma 2.3 we may deduce that $\vec{\nu} \in \vec{\tau}$.

   (b) The **appl(2)** rule was used, i.e. $E \vdash e_1 \Rightarrow \langle c \rangle$, $E \vdash e_2 \Rightarrow \vec{\nu_2}$ and $\text{APPLY}(c, \vec{\nu_2}) = \vec{\nu}$. We may use the induction hypothesis to deduce that $c \in \vec{\tau_2} \rightarrow \vec{\tau}$ and $\vec{\nu_2} \in \vec{\tau_2}$. From Definition 2.1 we can then conclude that $\vec{\nu} \in \vec{\tau}$.

5. The **pair** rule was used, i.e. $e \equiv (e_1, e_2)$ and we have smaller proofs of $T \vdash e_1 : \langle \tau_1 \rangle$ and $T \vdash e_2 : \langle \tau_2 \rangle$, where $\vec{\tau} = \langle \tau_1 * \tau_2 \rangle$. Furthermore, we must have proofs that $E \vdash e_1 \Rightarrow \langle \nu_1 \rangle$ and $E \vdash e_2 \Rightarrow \langle \nu_2 \rangle$ where $\vec{\nu} = \langle (\nu_1, \nu_2) \rangle$. By

the induction hypothesis, we can conclude that $\langle \nu_1 \rangle \in \langle \tau_1 \rangle$, i.e. $\nu_1 \in \tau_1$, and $\langle \nu_2 \rangle \in \langle \tau_2 \rangle$, i.e. $\nu_2 \in \tau_2$. Therefore $(\nu_1, \nu_2) \in \tau_1 * \tau_2$ and so $\langle (\nu_1, \nu_2) \rangle \in \langle \tau_1 * \tau_2 \rangle$.

6. The **seq(1)** rule was used, i.e. $e \equiv \varepsilon$ and $\overrightarrow{\tau} = \varepsilon$. But then $E \vdash \varepsilon \Rightarrow \varepsilon$, i.e. $\overrightarrow{\nu} = \varepsilon$ and $\varepsilon \in \varepsilon$ as required.

7. The **seq(2)** rule was used, i.e. $e \equiv e_1 \cdot e_2$, $T \vdash e_1 : \overrightarrow{\tau_1}$, $T \vdash e_2 : \langle \tau_2 \rangle$ and $\overrightarrow{\tau} = \overrightarrow{\tau_1} \cdot \tau_2$. Furthermore, $E \vdash e_1 \Rightarrow \overrightarrow{\nu_1}$, $E \vdash e_2 \Rightarrow \langle \nu_2 \rangle$ and $\overrightarrow{\nu} = \overrightarrow{\nu_1} \cdot \nu_2$. By the induction hypothesis, $\overrightarrow{\nu_1} \in \overrightarrow{\tau_1}$ and $\langle \nu_2 \rangle \in \langle \tau_2 \rangle$, and hence $\overrightarrow{\nu_1} \cdot \nu_2 \in \overrightarrow{\tau_1} \cdot \tau_2$.

□

This theorem is perhaps not quite as strong as we would like. The type system should ensure that the evaluation of well-typed programs never gets stuck, meaning that the evaluation should either yield a sequence of values or fail to terminate. Our consistency theorem is weaker than this, only proving that *if* a value is produced then it is of the expected type. We could introduce 'wrong' values and add additional rules to produce one of these values whenever the evaluation can get stuck [5]. The modified consistency theorem would then claim that wrong values cannot be produced by well-typed expressions. Unfortunately this modification is still not entirely satisfactory as we may have missed some places where the evaluation can get stuck. A more comprehensive treatment requires the use of a transitional semantics where we can explicitly identify the intermediate configurations of an evaluation. It is a simple task to produce a transitional semantics for this language. Furthermore we can use evaluation contexts and/or a congruence relation to make the rules almost as concise as those presented in Figure 3. Unfortunately these techniques complicate the resulting consistency proof whilst shedding no additional light on the matters at hand. We will therefore stick with the relational approach for the remainder of the paper.

3 Some Examples

In this section we adopt a syntax slightly closer to SML to aid readability. Furthermore, we shall often write a type or a value where a singleton sequence is expected where this causes no confusion. We start by illustrating how sequences may be used *explicitly* by the programmer to increase program efficiency. The `zipmap` function presented in Section 1 was inefficient because it created an intermediate pair to hold `(h1,h2)`. If we promise to always provide `zipmap` with a function that expects a sequence rather than a pair then we can avoid this overhead. Furthermore, the type system will ensure that this promise is always kept. The modified version of `zipmap` is presented below

```
fun zipmap(F, [], []) = []
  | zipmap(F, h1::t1, h2::t2) = F<h1,h2> :: (zipmap(F, t1, t2))
```

The type of zipmap is now `((<'a,'b> → 'c) * 'a list * 'b list) -> 'c list`. Of course the downside of this alteration is that we can no longer apply zipmap to a function expecting a pair. This problem can be alleviated by the use of explicit or implicit coercions, as shown in Section 4.

Now consider an example where the inefficiency arises due to passing tuples as results. The function splitmap applies a function f to each element of a list, just as map does. However, the result of applying f is a pair, and the two components are accumulated in separate lists. The code looks like:

```
fun splitmap F [] = ([],[])                          (0)
  | splitmap F (h::t) =
      let val (t1,t2) = splitmap F t                 (1)
          val (h1,h2) = F h                          (2)
       in (h1::t1, h2::t2) end                       (3)
```

It is difficult for a compiler to avoid generating a pair for (h1,h2). We could always compile code for functions returning tuples so that they returned a sequence instead, requiring the caller to build the tuple if required. However, this would not interact well with polymorphic functions. Furthermore, in an expression such as I x, where x is bound to a tuple, we really don't want to break x apart unless we really have to. As with many of the examples in this paper, any fixed compilation strategy tends to optimise some cases only at the expense of others. Lifting sequences into the language avoids these tradeoffs, but at the expense of additional complexity in the language. Implicit coercions may help to hide some of the syntactic complexity however.

Returning to the example, we can make the code more efficient if we can assume that F returns two results on the stack. We replace line (2) by

```
          val <h1,h2> = F h                          (2)
```

The type of splitmap is now `('a → <'b,'c>) → 'a list -> ('b list * 'c list)`.

One further source of inefficiency is the pair constructed as the result of a call to splitmap. In all of the recursive calls this is immediately bound to (t1,t2) by the calling function. If we are allowed to define splitmap so that it returns a sequence then we can define a more efficient version by

```
fun splitmap F [] = <[],[]>                          (0)
  | splitmap F (h::t) =
      let val <t1,t2> = splitmap F t                 (1)
          val <h1,h2> = F h                          (2)
       in <h1::t1, h2::t2> end                       (3)
```

The type of splitmap is now `('a → <'b,'c>) → 'a list -> <'b list, 'c list>`.

Our simple language does not deal with datatypes, but they are simple enough to add. A constructor function would then take a sequence of values and return a singleton value, i.e. its type would be of the form $\overline{\tau} \to \langle \tau \rangle$ where τ is the type of the datatype. Consider the datatype

```
datatype D =
    C1 of <(int * bool)>
  | C2 of < int , bool >
```

The constructor C1 expects a single argument as value, a pair, and when used in pattern matching allows a variable to be bound to this pair. The constructor C2, in contrast, expects to be passed two arguments. It seems natural to expect that C2, when used in patterns, should return two values. If we follow this approach then we would be prevented from writing a pattern such as C2 x, but a pattern such as C2 <y,z> would be accepted. This suggests that a value built using the C1 constructor should be represented as in Figure 1, whereas a value built using C2 should be represented as in Figure 2. Rather than relying on a compiler making smart decisions, which is sometimes difficult in a language such as SML and leads to compromises, we can now indicate explicitly the intended usage patterns associated with each constructor.

In SML all functions must take an argument and return a result. An imperative version of a pseudo-random number function, with the seed embedded within a reference, does not need to be passed an argument. Similarly, many functions are executed only for their side-effects and return no useful result. The value () of type unit is usually used in these cases. However, the compiler then needs to treat this type specially so that such values are not actually passed and returned when this can be avoided. Using the empty sequence avoids such ad hoc treatment of the unit type.

4 Coercions and Optimisations

A consequence of introducing a distinction between tuples and sequences is the frequent need to switch from one view to the other. Coercions, either explicit or implicit, can be of help here. Generalising the language to n-tuples, for $n \geq 2$, we define

Definition 4.1

$$TS_n = \lambda\langle(x_1, \ldots, x_n)\rangle.\, \langle x_1, \ldots, x_n\rangle$$
$$ST_n = \lambda\langle x_1, \ldots, x_n\rangle.\, (x_1, \ldots, x_n).$$

Note that even if we restrict ourselves to nested pairs we cannot define ST_n inductively as we are not allowed to bind sequences to variables.

We can make explicit use of these coercion functions to deal with typing mismatches but they are also of use when dealing with a variety of compiler optimisations. Consider the evaluation of the expression (e_1, e_2). We must first evaluate the expressions e_1 and e_2 and then place the results in a pair. Initially one might try to create space for the tuple first and then evaluate each expression directly into the appropriate slot. However, this can cause garbage collection problems. A garbage

collection might occur before both slots had been filled and if not careful we may end up traversing an uninitialised pointer. A better strategy is usually to evaluate the expressions onto a stack (or registers) and to then create and initialise the pair in an atomic operation. This suggests that we should view the expression (e_1, e_2) as an abbreviation for

$$ST_2(\langle e_1, e_2\rangle)$$

We can generalise to n-tuples in the obvious way.

Now consider the function $\lambda\langle(p_1, p_2)\rangle.\, e$. When such a function is applied to an argument the first step is usually to break the argument into its two components and to then break the i^{th} value apart using pattern p_i.[2] We can therefore treat this function definition as if it had been written

$$\underbrace{\lambda x.\,(\quad \overbrace{\lambda\langle p_1, p_2\rangle.\, e}^{\text{“fast entry point”}} \quad)(TS_2\ x)}_{\text{“slow entry point”}}$$

where x is a fresh variable. Again the generalisation to n-tuples should be obvious.

What do we gain by such transformations, other than perhaps clarifying the operational steps involved in evaluation? Consider the expression

$$(\lambda\langle(p_1, p_2)\rangle.\, e)\,(e_1, e_2)$$

Using our transformations, this expression can be viewed as an abbreviation for

$$(\lambda x.\,(\lambda\langle p_1, p_2\rangle.\, e)(TS_2\ x))\,(ST_2\ \langle e_1, e_2\rangle)$$

From the rules of the dynamic semantics, it should be clear that the composition of TS_2 and ST_2 behaves like (a suitably restricted version of) the identity function, i.e.

Theorem 4.1 $TS_2 \circ ST_2 \;=\; I : \langle'a, 'b\rangle \to \langle'a, 'b\rangle$

The following theorem is a simple consequence of this result:

Theorem 4.2 $(\lambda x.\,(\lambda\langle p_1, p_2\rangle.\, e)(TS_2\ x))(ST_2(\langle e_1, e_2\rangle)) \;=\; (\lambda\langle p_1, p_2\rangle.\, e)\langle e_1, e_2\rangle$

This theorem justifies the optimisation mentioned in Section 1 and can be easily generalised to arbitrary n-tuples.

[2] This isn't necessarily true if we have wildcard patterns. In this case we might take the tuple apart more selectively.

Sequences can help the compiler to optimise the code. However, they may become tedious to use as we have to keep track of whether a function expects a tuple or a sequence, in addition to the usual problems of remembering the argument ordering and types. This might be a case where implicitly adding coercions such as TS_2 is relatively harmless as we are not discarding any information. For function application the situation is reasonably straightforward. Suppose we have an expression such as $(e_1\ e_2)$. If e_1 expects a sequence and e_2 returns a tuple then we must insert a TS coercion. Similarly if e_1 expects a tuple and e_2 returns a sequence then an ST coercion is inserted.

What about pattern matching coercions? There are various ways that pattern matching can fail due to tuple/sequence inconsistencies. Suppose we have a pattern p of the form $\mathsf{C}\ \overrightarrow{p}$, i.e. a constructor C applied to a sequence of patterns $\overrightarrow{p}$. If C expects a singleton ⟨n-tuple⟩ and $\overrightarrow{p}$ is a sequence of length n, $n \neq 1$, then we can just rewrite $\overrightarrow{p}$ as a tuple pattern. If C expects a sequence and $\overrightarrow{p}$ is a singleton tuple then there are two cases to consider:

1. $\overrightarrow{p}$ is of the form $\langle(p_1, \ldots, p_n)\rangle$, in which case we can just replace it by $\langle p_1, \ldots, p_n\rangle$.
2. It is of the form $\langle x\rangle$. In this case we have to introduce n fresh variables, $x_1, \ldots, x_n$, and replace x by $\langle x_1, \ldots, x_n\rangle$. We then need to bind x to $(x_1, \ldots, x_n)$ in the body of the match (e.g. using a let expression).

5 Generalisations

The expressions e_1 and e_2 are treated asymmetrically in $e_1 \cdot e_2$, as e_2 is constrained to evaluate to a singleton sequence. This is convenient as it allows us to indicate that an expression produces a single value by writing it as $\langle e\rangle$. However, it may also be useful to have a symmetrical version of this operator, written $e_1 \diamond e_2$. The rule for this operator is a variant of the **seq(2)** rule:

$$\frac{E \vdash e_1 \Rightarrow \overrightarrow{v_1} \quad E \vdash e_2 \Rightarrow \overrightarrow{v_2}}{E \vdash e_1 \diamond e_2 \Rightarrow \overrightarrow{v_1}\overrightarrow{v_2}}$$

Consider the application of e_1 to e_2. Suppose e_1 evaluates to a function expecting a sequence of arguments of length m and e_2 produces a sequence of values of length n, where $n > m$. The result would be undefined using our present rules as the pattern matching would fail. However, operationally this application might make sense. If the n values are pushed onto the stack then the function can just use the top m values, leaving $n - m$ values behind. Let's call the sequence of values left over $\overrightarrow{u}$. If the function produces the sequence of values $\overrightarrow{v}$ then the result of the whole application would be the sequence $\overrightarrow{u}\,\overrightarrow{v}$.

$$\frac{E \vdash e_1 \Rightarrow \langle[\,\overrightarrow{p}, e, E'\,]\rangle \quad E \vdash e_2 \Rightarrow \overrightarrow{v_2} \qquad \overrightarrow{v_2} = \overrightarrow{v_{21}}\overrightarrow{v_{22}} \qquad \overrightarrow{p}, \overrightarrow{v_{22}} \Rightarrow E'' \qquad E'E'' \vdash e \Rightarrow \overrightarrow{v}}{E \vdash e_1\ e_2 \Rightarrow \overrightarrow{v_{21}}\,\overrightarrow{v}}$$

Indeed one might go further, allowing e_1 to produce more than one value as well, as in

$$\frac{E \vdash e_1 \Rightarrow \overrightarrow{v_1} \cdot [\,\overrightarrow{p}, e, E'\,] \quad E \vdash e_2 \Rightarrow \overrightarrow{v_2} \qquad \overrightarrow{v_2} = \overrightarrow{v_{21}}\overrightarrow{v_{22}} \qquad \overrightarrow{p}, \overrightarrow{v_{22}} \Rightarrow E'' \quad E'E'' \vdash e \Rightarrow \overrightarrow{v}}{E \vdash e_1\ e_2 \Rightarrow \overrightarrow{v_1}\overrightarrow{v_{21}}\overrightarrow{v}}$$

Exploiting such generalisations in a user program would almost certainly be confusing. However, if their use arose naturally in the study of compiler optimisations then they might be worth investigating further.

6 Conclusions

We have demonstrated that the traditional SML approach, where tuples are used to model multiple values and results, may lead to implementation inefficiencies. Distinguishing these concepts at the language and type levels avoids many of these problems. Furthermore, in many cases coercion functions may be implicitly introduced to reduce the impact of these distinctions when efficiency is not such an important factor.

The paper has demonstrated how a small, albeit important, compiler optimisation can be formalised and verified using sequences. We need to investigate whether other optimisations can also be elucidated in this setting, or new optimisations suggested by this framework.

Our motivating examples were based on a stack view of program evaluation. We need to investigate the impact of our extensions on other implementation techniques, e.g. the continuation-passing style used in the NJ/SML implementation [1].

We cannot bind a sequence to a variable, for reasons discussed earlier. This has an impact on the types of the higher-order functions used extensively in ML programs. For example, we cannot write a general function composition operator that works over arbitrary sequences. However, for any fixed sequence lengths we can write such a function. Similarly, the identity function will only work over singleton sequences although versions for other lengths can be constructed. In general, we may have to construct a family of higher-order functions working on a range of sequence lengths. Alternatively we must use coercion functions to move to the tuple world before applying such higher-order functions. It would be interesting to investigate whether these cases can be handled in a more elegant fashion.

Introducing sequences into SML, as opposed to using them in an intermediate language within an ML compiler, would obviously involve some syntactic changes to the language. An alternative would be to leave the syntax unchanged but to alter the semantics so that tuples denoted multiple values. This would have the effect of invalidating any existing SML program where a tuple was bound to a variable, although a compiler may be able to add the appropriate coercions in many cases. It would be interesting to discover how often this situation occurred in practice. Binding tuples to variables may be rarer than we think because compound values are often packaged up using datatypes or records. It would be an interesting experiment

to instrument an SML compiler to gather statistics on how frequently tuples were used as compound data values in non-trivial programs.

References

[1] A. Appel. *Compiling with Continuations*, Cambridge University Press, 1992.

[2] L. Cardelli. *Amber*, In G. Cousineau, P.-L. Curien, and B. Robinet, eds. Combinators and Functional Programming Languages, LNCS 242, Springer, Berlin, 1986.

[3] X. Leroy. *The Zinc experiment: an economical implementation of the ML language*, INRIA Rocquencourt report, 1990.

[4] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*, MIT Press, 1990.

[5] M. Tofte. *Operational Semantics and Polymorphic Type Inference*, Computer Science Report CST-52-88, University of Edinburgh, 1988.