Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 91 (2004) 195–211
www.elsevier.com/locate/entcs

The Polymorphic Imperative: a Generic Approach to In-place Update

C. B. Jay [1] H. Y. Lu [2] Q. T. Nguyen [3]

Faculty of Information Technology
University of Technology, Sydney
Sydney, Australia

Abstract

The constructor calculus supports generic operations defined over arbitrary data types including abstract data types. This paper extends the basic constructor calculus to handle constructed locations. The resulting calculus is able to define a generic assignment operation that performs in-place whenever appropriate and allocates fresh memory otherwise. This approach may eliminate many of the space overheads associated with higher-order polymorphic languages. In combination with existing generic programming techniques it can express some very powerful algorithms such as the visitor pattern.

Keywords: generic functions, constructor calculus, imperative programming, in-place update, location constructors

1 Introduction

One of the great strengths of functional programming is that it relieves the programmer of the need to manage memory, which helps to make programs shorter and easier to reason about. The price to be paid is that their compilers must take a conservative approach to memory allocation, often allocating new space in the heap and garbage collecting the old when in-place update would have been perfectly safe.

[1] Email: cbj@it.uts.edu.au
[2] Email: helenlu@it.uts.edu.au
[3] Email: qtnguyen@it.uts.edu.au

doi:10.1016/j.entcs.2003.12.013

A significant effort has been made to improve the efficiency of this process. For example, types in compilation [11] uses type information to ensure the safety of some in-place updating. This works well for integers, floats and tuples built of such simple types, but does not handle recursive types like lists where the type does not determine the shape of its values. Monitoring list lengths etc. may be attempted using *sized types* [4] or other dependently-typed systems, e.g. [13] with the aim of extracting compile-time information.

This paper provides an approach to assignment that is able to determine at run-time whether to assign in-place or not using existing techniques from the *constructor calculus* [5,6]. This is a variant of the lambda-calculus which supports powerful generic programming [2,7] by means of program *extensions* based on pattern-matching over constructors of arbitrary type. It is able to express generic programs for the usual second-order functions, like mapping and folding. It also can easily extend numerical operations, like equality and addition, from primitive datum types (such as the primitive integers or floats) to arbitrary data types.

The same approach is here used to extend primitive operations on locations to generic operations on *constructed locations*. When generic assignment is defined in this way then in-place update is the norm, based on matching the structure of the location with that of its new value. We can combine the primitive assignment with the generic to get the best of both worlds, complete safety and in-place update when possible.

Three examples are used to illustrate some of the benefits of this approach. The insertionsort program shows how a space efficient program can be written using higher-order functions and pattern-matching. The program converge shows how to iterate a function while using space efficiently. The visitor program captures the visitor pattern [3] as a generic function of type

$$\mathsf{visitor} : \mathsf{name}(X, Y) \to (\mathsf{loc}\ Y \to \mathsf{comm}) \to \mathsf{loc}\ Z \to \mathsf{comm}$$

which operates as follows: visitor $n\ f\ z$ traverses the structure of z applying f to any sub-structures named by n. For example, n may characterize employees and f may increment employee salaries inside an organisation.

The structure of the rest of the paper is as follows. Section 2 recalls key aspects of the constructor calculus. Section 3 adds types of locations to the basic calculus with some primitive operations for their creation, reading and writing. Section 4 introduces constructed locations and the corresponding operations. Section 5 provides examples. Section 6 draws conclusions and considers future work.

2 Review of the constructor calculus

The constructor calculus introduced in [5] was created to support the definition of generic functions like mapping and folding in which the types of parameters and return values can be instantiated for a variety of datatypes. The resultant definitions allow these functions to be able to act on a large class of datatypes. The work is further developed in [6] which explains any undefined notation used in this paper. The latter also showed how the key ideas could be demonstrated with respect to the Hindley-Milner type system, a simple variant of which has *types* T and *terms* t given by

$$T ::= X \mid D \mid 1 \mid T * T \mid T \to T$$

$$t ::= x \mid d \mid \mathsf{un} \mid \mathsf{pair} \mid t\ t \mid \lambda x.t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid \mathsf{fix}.$$

X is a *type variable*, D is a *type constant*, 1 is the *unit* type, $T_0 * T_1$ is the *product* of T_0 and T_1 and $T_0 \to T_1$ is the type of *functions* from T_0 to T_1. It would be a trivial matter to add *coproduct* (or *sum*) types to the calculus, but we shall see that they can be handled by allowing data type definitions. The term forms represent, respectively, variables, constants, the unique value of unit type, pairing, application, abstraction, conditional, let-declaration, and fixpoints.

The constructor calculus is created by replacing the conditional with a more powerful branching construct called an *extension*. Its syntax is

$$\mathsf{under}\ c\ \mathsf{apply}\ f\ \mathsf{else}\ g$$

where c is a *constructor* and f and g are terms, called the *specialisation* and *default* function respectively. The rewriting rules for extensions are

$$\begin{aligned}(\mathsf{under}\ c_n\ \mathsf{apply}\ f\ \mathsf{else}\ g)\ (c_n\ t_0\ \ldots\ t_{n-1}) &> f\ t_0\ \ldots\ t_{n-1}\\ (\mathsf{under}\ c_n\ \mathsf{apply}\ f\ \mathsf{else}\ g)\ t &> g\ t \quad \text{otherwise}\end{aligned}$$

where c_n represents a constructor that takes n arguments. From these rules it is clear that f may have a more specialised type than g since it is only required to act on the arguments of the constructor c. This is the key difference from conditionals (or case analyses) where the two branches must have exactly the same type. The type derivation rule for extensions is given by

$$\frac{\begin{array}{ll} c_n : \forall \Delta_c.T_0 \to \ldots \to T_n & \Gamma \vdash g : T \to T' \\ \upsilon = \mathcal{U}(T_n, T) & \upsilon\Gamma \vdash f : \upsilon(T_0 \to \ldots T_{n-1} \to T') \end{array}}{\Gamma \vdash \mathsf{under}\ c_n\ \mathsf{apply}\ f\ \mathsf{else}\ g : T \to T'}$$

where Γ is the term context and $\mathcal{U}$ is the most general unifier. It is important for the type safety of specialisation that the type scheme $\forall\Delta_c.T_0 \to \ldots \to T_n$ for c should be its principal type scheme, i.e. the one it is born with, since the specialisation function must be able to act on the sub-terms of any term constructed by c. That is, specialisation may assume the most general unification of T_n and T but no more.

Because of the constraints above, we limit ourselves to a finite set of constructor constants. Fundamental examples are un, pair and the polymorphic *exception*

$$\mathsf{exn} : X \to Y$$

Other examples will be added by datatype declarations.

It is useful to be able to extend numerical functions like addition and equality to arbitrary data types. To do this requires patterns that match with integers and floats. One approach would be to treat integers and floats as values constructed from some type of primitive integers and primitive floats, e.g. tuples of bits. Since it is unpleasant to expose these in the programming language we shall adopt a different approach, namely to introduce a new term form for each primitive data type, e.g. for the floating point numbers, one can have

$$\mathsf{underfloat\ apply}\ f\ \mathsf{else}\ g$$

with type derivation rule

$$\frac{\upsilon = \mathcal{U}(\mathsf{float}, T) \qquad \begin{array}{c}\Gamma \vdash g : T \to T' \\ \upsilon\Gamma \vdash f : \mathsf{float} \to \upsilon T'\end{array}}{\Gamma \vdash \mathsf{underfloat\ apply}\ f\ \mathsf{else}\ g : T \to T'}$$

and evaluation rules

$$(\mathsf{underfloat\ apply}\ f\ \mathsf{else}\ g)\ n > f\ n \quad \text{if } n \text{ is a floating point number}$$

$$(\mathsf{underfloat\ apply}\ f\ \mathsf{else}\ g)\ t > g\ t \quad \text{if } t \text{ cannot be a floating point number.}$$

We may write

$$\mathsf{under\ float\ apply}\ f\ \mathsf{else}\ g$$

for underfloat apply f else g. Similar rules and conventions will apply for a type int of integers and other datum types. As syntactic sugar, we may also write

$$\mathsf{match}\ t_1\ \mathsf{with}\ t$$

for $t\ t_1$ especially when t is given by pattern-matching.

We can use pattern-matching syntax to express extensions, where the pattern

$$\mid c\, x_0 \ \ldots \ x_{n-1} \rightarrow t$$

represents $\mathsf{under}\ c\ \mathsf{apply}\ \lambda x_0, \ldots x_{n-1}.t\ \mathsf{else}\ \ldots$. Similarly,

$$\mid \mathsf{float}\ x \rightarrow t$$

represents $\mathsf{underfloat}\ \mathsf{apply} \lambda x.t\ \mathsf{else}\ \ldots$. The ultimate default function in a pattern-match is given by the exception constant exn. For example, here is an equality function that acts on arbitrary tuples of floating point numbers

$$(\mathsf{tuplefloatequal} : X \rightarrow X \rightarrow \mathsf{bool}) =$$

$$\mid \mathsf{float}\ x \rightarrow (\ \mid \mathsf{float}\ y \rightarrow x = y)$$

$$\mid \mathsf{un} \rightarrow \lambda y.\mathsf{true}$$

$$\mid \mathsf{pair}\ x_0\ x_1 \rightarrow (\mid \mathsf{pair}\ y_0\ y_1 \rightarrow \mathsf{tuplefloatequal}\ x_0\ y_0\ \&\&\ \mathsf{tuplefloatequal}\ x_1\ y_1)$$

In practice, one wishes to create abstract data types and use their constructors, also known as *abstractors* when defining generic functions. For example,

$$\mathtt{datatype}\ \mathsf{complex} = \mathsf{complex\ of\ float\ and\ float}$$

introduces complex as a new abstractor which can be used to define new operations. If abstractors are treated as primitives then existing generic functions such as equality must be extended with cases to handle each new abstractor. Instead, terms built using abstractors are given a concrete representation (reflecting their deep structure) as a tuple of arguments (built using pair and un). This is then tagged with a *name* (representing the surface structure). Tagging is handled by the constructor tag of type

$$\mathsf{tag} : \forall X, Y\ \mathsf{name}(X, Y) \rightarrow X \rightarrow Y$$

where $\mathsf{name}(X, Y)$ is a type of names. For example, complex can be interpreted as

$$\lambda x, y.\mathsf{tag}\ \mathsf{complex\_name}\ (\mathsf{pair}\ x\ y)$$

where $\mathsf{complex\_name} : \mathsf{name}(\mathsf{float}{*}\mathsf{float}, \mathsf{complex})$. Comparison of arbitrary abstractors is achieved by pattern-matching against their names. To be precise, this requires an additional type derivation rule, namely

$$\frac{\begin{array}{ll} n : \mathsf{name}(X, Y) & g : T \rightarrow T' \\ \upsilon = \mathcal{U}(\mathsf{name}(X, Y), T) & f : \upsilon(T') \end{array}}{\vdash \mathsf{under}\ n\ \mathsf{apply}\ f\ \mathsf{else}\ g : T \rightarrow T'}$$

Type safety is maintained because name constants are required to have constant types.

So, a fully generic equality for data types is given by

```
(equal : $X \to X \to$ bool) =
| int $x \to$ ( | int $y \to$ primintequal $x$ $y$)
| float $x \to$ ( | float $y \to$ primfloatequal $x$ $y$)
| un $\to \lambda y$.true
| pair $x_0$ $x_1 \to$ ( | pair $y_0$ $y_1 \to$ equal $x_0$ $y_0$ && equal $x_1$ $y_1$)
|  tag $m$ $x \to$ (
   | tag $n$ $y \to$ (
     match $n$ with
      | $m \to$ equal $x$ $y$
      | _ $\to$ false))
| _ $\to \lambda y$.false
```

For example, equal (complex 3.3 4.4) (complex 3.3 5.5) reduces to applying

under complex_name apply equal (pair 3.3 4.4) (pair 3.3 5.5) else λy.false

to complex_name, which ultimately reduces to false.

The same approach can be used to generalize other numerical operations, e.g. addition, to arbitrary data structures. These generic operations can then be customized if desired. For example, multiplication of complex numbers can be given its own case.

3 Locations

This section adds to the constructor calculus some imperative features in a style similar to that of ML [10]. In this setting, the assignment can be treated as an atomic operation, which is simple to describe but extravagant with space.

The type comm of *commands* is equipped with two constants

$$\mathsf{skip} : \mathsf{comm}$$
$$\mathsf{seq} : \mathsf{comm} \to X \to X.$$

The command skip has no effect. Evaluation of a term of the form seq t_0 t_1 executes the command t_0 and then evaluates the term t_1. We may write $x; y$ for seq x y. While- and for-loops can be defined by fix-point construction, in the usual way.

Now let us consider assignable locations. Each type T has an associated type

$$\mathsf{loc}\ T$$

of *locations* that store values of type T. Locations support three polymorphic constants:

$$\mathsf{primloc} : X \to \mathsf{loc}\ X$$

$$\mathsf{primval} : \mathsf{loc}\ X \to X$$

$$\mathsf{primassign} : \mathsf{loc}\ X \to X \to \mathsf{comm}.$$

A term of the form $\mathsf{primloc}\ t$ creates a location whose initial value is that of t. A term of the form $\mathsf{primval}\ u$ represents the value stored at u. A term of the form $\mathsf{primassign}\ u\ t$ updates the location u with the value of t. Garbage collection is required to recover the redundant locations.

In general, one must restrict the quantification of type variables appearing in location types (see, e.g. [8,12] but this does not limit the polymorphism of the examples in this paper.

The *evaluation rules* in Figure 1 employ a big-step operational semantics. The *values* (meta-variable v) are given by the lambda-abstractions, extensions, constructors, constants, and terms of the form $d\ v_0\ \ldots\ v_k$ where d is a constructor or constant for which there is no explicit evaluation rule. Such rules are summarised by the rule for evaluating $d\ d_0\ \ldots d_{n-1}$ in the figure.

A *store* (meta-variable Σ) is a function from term variables of location type to values. These term variables must be of location type, and will be represented by the meta-variable u. An *evaluation context* is a pair (Σ, t) in which all free variables of the term t are in the domain of the store Σ. Evaluation is expressed using judgements of the form

$$(\Sigma, t) \Rightarrow (\Sigma', v)$$

where (Σ, t) is an evaluation context.

Most of the evaluation rules are standard. Note that beta-reduction is eager. For example, $(\Sigma, \mathsf{primassign}\ t_0\ t_1)$ is evaluated by first evaluating t_0. If its value is an identifier u then the value of u is updated in the store to that of the result of evaluating t_1. If, however, t_0 has some other value, e.g. an exception, then the result of evaluation is an exception.

Theorem 3.1 *For each evaluation context* (Σ, t) *there is an evaluation rule which can be applied. That is, evaluation is never stuck.*

Proof. The proof is by induction on the structure of t. $\square$

The evaluation rules give no indication of how store operations are to be implemented. One expects that assignment of datum values like integers or floats will be performed in-place and that assignment of functions will be by allocating fresh memory. The delicate case is an assignment of structured data, like a list.

Define list types by

$$\texttt{datatype}\ \mathsf{list}\ X = \mathsf{nil} \mid \mathsf{cons\ of}\ X : \mathsf{list}\ X$$

and adopt the usual functional syntax for representing them, e.g. $[1, 2, 3]$ represents $\mathsf{cons}\ 1\ (\mathsf{cons}\ 2\ (\mathsf{cons}\ 3\ \mathsf{nil}))$. Now consider the example

$$\begin{aligned}&\mathsf{let}\ x = \mathsf{primloc}\ [1]\ \mathsf{in}\\ &\mathsf{primassign}\ x\ [2]; \qquad (1)\\ &\mathsf{primassign}\ x\ [8, 9].\end{aligned}$$

Clearly, the first assignment could be in-place, but any simple implementation of primassign will miss this opportunity since it will not be able to distinguish this case from the second, shape-changing assignment.

4 Location Constructors

In-place update is possible when the structure of the location is matched by that of its new value. This matching can be checked by comparing constructors if locations are constructed in the same way that their values are. This section introduces a new class of constructors, the *location constructors*. They can be used to create generic functions for locating, valuing and assigning based on their primitive versions, just as equal is based on datum equality. In this setting, assignment can be treated as a generic operation, which can be performed in-place if possible, and create a new location otherwise

To each constructor $c : T_0 \to \ldots \to T_n$ associate a location constructor $\mathsf{conloc}\ c$ of type

$$\mathsf{conloc}\ c : \mathsf{loc}\ T_0 \to \ldots \to \mathsf{loc}\ T_n.$$

The two new evaluation rules associated with location constructors are given in Figure 2. They create identifiers for constructed locations and provide a specialisation rule for extensions that use them (the default rule being unchanged). Note that Theorem 3.1 still applies to this augmented system. Note too that if c is an abstractor then the techniques used to give c a concrete representation must be adapted for $\mathsf{conloc}\ c$.

The generic function loc creates constructed locations for constructed terms and primitive locations otherwise, e.g. for functions, commands and locations

$$\frac{(\Sigma, t[v_0/x]) \Rightarrow (\Sigma', v)}{(\Sigma, (\lambda x.t)\ v_0) \Rightarrow (\Sigma', v)}$$

$$\frac{(\Sigma, t\ (\mathsf{fix}\ t)) \Rightarrow (\Sigma', v)}{(\Sigma, \mathsf{fix}\ t) \Rightarrow (\Sigma', v)}$$

$$\frac{(\Sigma, t_0) \Rightarrow (\Sigma', v_0) \quad (\Sigma', t_1[v_0/x]) \Rightarrow (\Sigma'', v_1)}{(\Sigma, \mathsf{let}\ x = t_0\ \mathsf{in}\ t_1) \Rightarrow (\Sigma'', v_1)}$$

$$\frac{(\Sigma, t_0\ v_0\ \ldots v_{n-1}) \Rightarrow (\Sigma', v)}{(\Sigma, \mathsf{under}\ c_n\ \mathsf{apply}\ t_0\ \mathsf{else}\ t_1\ \mathsf{to}\ (c_n\ v_0\ \ldots v_{n-1})) \Rightarrow (\Sigma', v)}$$

$$\frac{(\Sigma, t_1\ v_2) \Rightarrow (\Sigma', v_3)}{(\Sigma, \mathsf{under}\ c_n\ \mathsf{apply}\ t_0\ \mathsf{else}\ t_1\ \mathsf{to}\ v_2) \Rightarrow (\Sigma', v_3)}$$

$$\frac{}{(\Sigma, \mathsf{exn}\ t_0\ t_1) \Rightarrow (\Sigma, \mathsf{exn}\ t_0)} \qquad \frac{}{(\Sigma, v\ (\mathsf{exn}\ t)) \Rightarrow (\Sigma, \mathsf{exn}\ t)}$$

$$\frac{}{(\Sigma, d\ d_0 \cdots d_{n-1}) \Rightarrow (\Sigma, d_n)}\ d\ d_0 \cdots\ d_{n-1} = d_n$$

$$\frac{}{(\Sigma, \mathsf{seq\ skip}\ v) \Rightarrow (\Sigma, v)}$$

$$\frac{}{(\Sigma, \mathsf{primloc}\ v) \Rightarrow (\Sigma, u \mapsto v, u)}\ u \text{ fresh}$$

$$\frac{}{(\Sigma, \mathsf{primval}\ u) \Rightarrow (\Sigma, \Sigma(u))} \qquad \frac{}{(\Sigma, \mathsf{primval}\ v) \Rightarrow (\Sigma, \mathsf{exn}\ v)}$$

$$\frac{}{(\Sigma, \mathsf{primassign}\ u\ v) \Rightarrow (\Sigma, u \mapsto v, \mathsf{skip})} \quad \frac{}{(\Sigma, \mathsf{primassign}\ v_0\ v_1) \Rightarrow (\Sigma, \mathsf{exn}\ v_0)}$$

$$\frac{}{(\Sigma, v) \Rightarrow (\Sigma, v)}$$

$$\frac{(\Sigma, t_0) \Rightarrow (\Sigma', v_0) \quad (\Sigma', t_1) \Rightarrow (\Sigma'', v_1) \quad (\Sigma'', v_0\ v_1) \Rightarrow (\Sigma''', v_2)}{(\Sigma, t_0\ t_1) \Rightarrow (\Sigma''', v_2)}$$

Fig. 1. Evaluation Rules

themselves. It is defined by

$$(\mathsf{loc} : X \to \mathsf{loc}\ X)\ =$$

$$\mid \mathsf{un} \to \mathsf{conloc\ un}$$

$$\mid \mathsf{pair}\ x0\ x1 \to \mathsf{conloc\ pair}\ (\mathsf{loc}\ x0)\ (\mathsf{loc}\ x1)$$

$$\mid \mathsf{tag}\ m\ x0 \to \mathsf{conloc\ tag}\ (\mathsf{primloc}\ m)\ (\mathsf{loc}\ x0)$$

$$\mid x \to \mathsf{primloc}\ x$$

$$\frac{}{(\Sigma, \mathsf{conloc}\ c_n\ u_0\ \ldots\ u_{n-1}) \Rightarrow (\Sigma, u \mapsto \mathsf{conloc}\ c_n\ u_0\ \ldots\ u_{n-1}, u)}\ u \text{ fresh}$$

$$\frac{(\Sigma, t_0\ u_0\ \ldots\ u_{n-1}) \Rightarrow (\Sigma', v)}{(\Sigma, \mathsf{under\ conloc}\ c_n\ \mathsf{apply}\ t_0\ \mathsf{else}\ t_1\ \mathsf{to}\ u) \Rightarrow (\Sigma', v)}\ \Sigma(u) = \mathsf{conloc}\, c_n\, u_0 \ldots u_{n-1}$$

Fig. 2. Evaluating constructed locations

Similarly, the generic valuation function is

```
(val : loc $X \to X$) =
    | conloc un $\to$ un
    | conloc pair $x0$ $x1$ $\to$ pair (val $x0$) (val $x1$)
    | conloc tag $m$ $x0$ $\to$ tag (primval $m$) (val $x0$)
    | $x \to$ primval $x$
```

The generic assignment function assign follows the same basic pattern as the others but takes two arguments. The definition of assign can be

```
(assign : loc $X \to X \to$ comm) =
    | conloc un $\to$ ( | un $\to$ skip)
    | conloc pair $x0$ $x1$ $\to$ ( | pair $y0$ $y1$ $\to$ assign $x0$ $y0$; assign $x1$ $y1$)
    | conloc tag $m$ $x$ $\to$ (
      | tag $n$ $y$ $\to$
        match primval $m$ with
        | $n \to$ assign $x$ $y$
        | _ $\to$ let $u$ = conloc tag $m$ $x$ in primassign $u$ (tag $n$ $y$))
    | $x \to$ primassign $x$
```

If the location was created by primloc then primassign will be invoked. Otherwise, assign will attempt to match the location constructor with that of the new value. The only case where the matching can fail is when the location and the term are tagged with different names, e.g. the assignment in

```
let $x$ = loc $[1, 3, 5]$ in
assign $x$ $[2, 4, 6]$;
```

will succeed in updating the whole structure in-place, but the assignment in

$$\mathsf{let}\ x = \mathsf{loc}\ [1,3]\ \mathsf{in}$$
$$\mathsf{assign}\ x\ [2,4,6];$$

will first update the locations for the first two list entries to 2 and 4, but then fail to match nil_name with cons_name.

It is convenient to have some sugar syntax here: let $!x$ denote val x and $x := y$ denote assign x y. Here is our earlier example (1) modified to use constructed locations:

$$\mathsf{let}\ x = \mathsf{loc}\ [1]\ \mathsf{in}$$
$$x := [2];$$
$$x := [8,9].$$

Now the first assignment is in-place and the second assignment succeeds too.

Thus assignment operation is now a generic function that can perform in-place whenever reasonable and allocate fresh memory otherwise.

5 Examples

This section uses three examples to illustrate how higher-order functions and pattern-matching can be combined with in-place update, user-control of memory and generic functions to produce short, expressive, efficient programs.

5.1 Insertion Sort

Insertion sort works by recursively inserting elements into a sorted list. Here is an implementation as a pair of purely functional programs. The insertion is performed by

```
(funinsertion : $(X \to X \to \mathsf{bool}) \to X \to \mathsf{list}\ X \to \mathsf{list}\ X)\ g\ x =$
  | nil $\to [x]$
  | cons $h\ t\ \to$
    if $g\ x\ h$
    then cons $h$ (funinsertion $g\ x\ t$)
    else cons $x$ (cons $h\ t$)
```

funinsertion is then used recursively to perform the sort in

```
(funinsertionsort : $(X \to X \to \mathsf{bool}) \to \mathsf{list}\ X \to \mathsf{list}\ X)$ $g$ =
    | nil → nil
    | cons $h\ t$ → funinsertion $g\ h$ (funinsertionsort $g\ t$)
```

This algorithm uses space proportional to the square of the list length. The following imperative algorithm insertionsort has a similar structure but only uses a constant amount of new memory (when performing swap).

```
(swap : loc $X$ → loc $X$ → comm) $x\ y$ =
  let $t$ = $!x$ in $x$ :=$!y$; $y$ := $t$


(insertion : $(X \to X \to \mathsf{bool})$ → loc $X$ → loc list $X$ → comm) $g\ x$ =
    | conloc nil → skip
    | conloc cons $h\ t$ →
      if $g\ !x\ !h$
      then swap $x\ h$; insertion $g\ h\ t$
      else skip


(insertionsort : $(X \to X \to \mathsf{bool})$ → loc list $X$ → comm) $g$ =
    | conloc nil → skip
    | conloc cons $h\ t$ → insertionsort $g\ t$; insertion $g\ h\ t$
```

The drawback of this program is that the assignments may be expensive to execute when the structures are large. The solution is to instantiate the polymorphic insertionsort to a type loc Y of locations to get a program of type

$$(\mathsf{loc}\ Y \to \mathsf{loc}\ Y \to \mathsf{bool}) \to \mathsf{loc\ list\ loc}\ Y \to \mathsf{comm}$$

which can easily be modified to produce a program

$$\mathsf{insertionsort} : (Y \to Y \to \mathsf{bool}) \to \mathsf{loc\ list\ loc}\ Y \to \mathsf{comm}$$

which will use a more efficient swapping.

5.2 Converge

The function converge defined below iterates a function $f : X \to X$ until the result stabilises, i.e. until some test $t : X \to X \to \mathsf{bool}$ applied to the old and new values becomes true. This captures a common situation when modelling the evolution of some system to a steady state, e.g. in the Barnes-Hut algorithm [1].

```
(converge : $(X \to X) \to (X \to X \to \mathsf{bool}) \to X \to X)$ $f\ t\ x =$
  let $y$ = loc $x$ in
  let $z$ = loc $(f\ x)$ in
  let $b$ = loc false in
  while not $(t\ !y\ !z)$ do
      $b :=$ (not $!b$);
      if $!b$
      then $y := (f\ !z)$
      else $z := f\ !y$ done;
  $!y$
```

The use of locations allows the programmer to indicate that exactly two locations of type X are required, rather than an unbounded number. Further, assignment will be done in-place if possible, with fresh memory allocated only when necessary. There are many examples where this will yield significant benefits. For example, it is common to represent complex dynamical systems using structures built of regions whose behaviours are of approximately equal complexity. If a region is quiet then its representation maintains it shape, and in-place update succeeds. Conversely, if a region is eventful then the shape of its representation is likely to change, and require fresh memory.

5.3 The Visitor

The visitor pattern [3] describes the process of traversing (and updating) a data structure. This can be done in Java by sub-classing from the Walkabout class [9] which uses reflection to determine the necessary structure. The constructor calculus supports a single generic visitor through its powerful pattern-matching approach. As with insertion sort, we will examine both a functional and imperative version of the algorithm.

```
(funvisitor : name $(X,Y) \rightarrow (Y \rightarrow Y) \rightarrow Z \rightarrow Z$) $m$ $f$ =
    | pair $z_0$ $z_1$ $\rightarrow$ pair (funvisitor $m$ $f$ $z_0$) (funvisitor $m$ $f$ $z_1$)
    | tag $n$ $z_0$ $\rightarrow$ (
      match $n$ with
       | $m \rightarrow f$ (tag $n$ $z_0$)
       | _ $\rightarrow$ tag $n$ (funvisitor $m$ $f$ $z_0$))
    | $z \rightarrow z$
```

funvisitor n f z looks for sub-structures of z named by n and applies f to them. It can be viewed as a form of mapping. As with insertion sort, this algorithm is quadratic in its use of space.

By contrast, the (imperative) visitor only requires constant space.

```
(visitor : name $(X,Y) \rightarrow$ (loc $Y \rightarrow$ comm) $\rightarrow$ loc $Z \rightarrow$ comm) $m$ $f$ $z$ =
match $z$ with
    | conloc pair $z_0$ $z_1$ $\rightarrow$ visitor $m$ $f$ $z_0$; visitor $m$ $f$ $z_1$
    | conloc tag $n$ $z_0$ $\rightarrow$ (
      match (primval $n$) with
       | $m \rightarrow f$ $z$
       | _ $\rightarrow$ visitor $m$ $f$ $z_0$)
    | _ $\rightarrow$ skip
```

Let us consider a particular situation, of updating the salaries of staff in an organisation. Given a type of salaries

```
datatype salary = salary of float
```

and a salary update function

```
(change_salary : float $\rightarrow$ salary $\rightarrow$ salary) $k$ = | salary $y$ $\rightarrow$ salary $(k * y)$
```

we can define

```
(change_salaries : float $\rightarrow Z \rightarrow Z$) $k$ =
funvisitor salary_name (change_salary $k$).
```

For example, if we define a type of universities by

$$\texttt{datatype}\ \mathsf{string} = \mathsf{string\ of\ list\ char}$$

$$\texttt{datatype}\ \mathsf{staff\_name} = \mathsf{staff\_name\ of\ string}$$

$$\texttt{datatype}\ \mathsf{staff} = \mathsf{staff\ of\ staff\_name\ and\ salary}$$

$$\texttt{datatype}\ \mathsf{department} = \mathsf{department\ of\ list\ staff}$$

$$\texttt{datatype}\ \mathsf{university} = \mathsf{university\ of\ list\ department}$$

then

$$\mathsf{let\ department1} = \mathsf{department}\ [\mathsf{staff}\ (\mathsf{staff\_name}\ \text{“Barry”})\ (\mathsf{salary}\ 12.0),$$

$$\mathsf{staff}\ (\mathsf{staff\_name}\ \text{“Helen”})\ (\mathsf{salary}\ 13.0),$$

$$\mathsf{staff}\ (\mathsf{staff\_name}\ \text{“Tony”})\ (\mathsf{salary}\ 14.0)]\ \mathsf{in}$$

$$\mathsf{change\_salaries}\ 2.0\ \mathsf{department1}$$

evaluates to

$$\mathsf{department}\ [\mathsf{staff}\ (\mathsf{staff\_name}\ \text{“Barry”})\ (\mathsf{salary}\ 24.0),$$

$$\mathsf{staff}\ (\mathsf{staff\_name}\ \text{“Helen”})\ (\mathsf{salary}\ 26.0),$$

$$\mathsf{staff}\ (\mathsf{staff\_name}\ \text{“Tony”})\ (\mathsf{salary}\ 28.0)].$$

The advantage of this approach is two-fold. First, one does not have to write nested patterns to represent departments, etc. Second, the visitor can be re-used after any change in the university structure, e.g. to create divisions, or on a totally different organisational structure.

Similarly, if we define

$$(\mathsf{change\_salary} : \mathsf{float} \rightarrow \mathsf{loc\ salary} \rightarrow \mathsf{comm})\ k =$$

$$\mid \mathsf{conloc\ salary}\ y \rightarrow \mathsf{primassign}\ y\ (k\ * (\mathsf{primval}\ y))$$

then

$$(\mathsf{update\_salaries} : \mathsf{float} \rightarrow \mathsf{loc}\ Z \rightarrow \mathsf{comm})\ k =$$

$$\mathsf{visitor\ salary\_name}\ (\mathsf{update\_salary}\ k)$$

will update salaries in place.

6 Conclusions

The constructor calculus provides a powerful technique for building generic functions for operations like mapping and addition in terms of some simple primitives. This paper shows that the same approach can be applied to imperative operations. That is, primitive operations for creating, reading from and writing to locations can underpin the definition of generic functions for these operations. The most striking advantage of this approach is that the generic assignment operation performs in-place update whenever reasonable, and allocates fresh memory otherwise. To our knowledge, this has not been achieved in other polymorphic languages.

The expressive power of the approach is shown through some representative examples. The insertionsort program shows how the functional programming style, with its pattern-matching and recursion can be used to define efficient imperative code. The converge program nicely illustrates the value of sharing control between the programmer and the system: the programmer specifies how many data structures are required while the system determines when fresh storage is required. The visitor program shows how the power of generic programming style combines naturally with the imperative features to provide flexible programming on large data structures.

The ideas and examples in this paper show that the constructor calculus is able to combine the functional and imperative programming styles within a single, simple calculus. We are investigating its relevance for other programming style such as object-orientation.

References

[1] Barnes, J. E. and P. Hut, A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(6270):446–449, 1986.

[2] Backhouse, R. and T. Sheard, editors, *Workshop on Generic Programming: Marstrand, Sweden, 18th June, 1998.* Chalmers University of Technology, 1998.

[3] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] Hughes, R. J. M., L. Pareto and A. Aiken, Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages.* ACM Press, 1996.

[5] Jay, C. B., Distinguishing data structures and functions: the constructor calculus and functorial types. In S. Abramsky, editor, *Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001, Kraków, Poland, May 2001 Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 217–239. Springer, 2001.

[6] Jay, C. B., The constructor calculus. www-staff.it.uts.edu.au/~cbj/Publications/constructors.ps, 2002.

[7] Jeuring, J. editor, *Proceedings: Workshop on Generic Programming (WGP 2000): July 6, 2000, Ponte de Lima, Portugal.* Utrecht University, UU-CS-2000-19, 2000.

[8] Paulson, L. C., *ML for the Working Programmer.* Cambridge University Press, 2nd edition, 1996.

[9] Palsberg, Jens and C. Barry Jay, The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

[10] Standard ML of New Jersey, cm.bell-labs.com/cm/cs/what /smlnj/.

[11] *The Third ACM SIGPLAN Workshop on Types in Compilation (TIC 2000) Montreal, Canada September 21, 2000*, 2000. www.cs.cmu.edu /~crary/tic00/.

[12] Wright, Andrew, Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Rice University, 1993.

[13] Xi, H. and F. Pfenning, Eliminating array bound checking through dependent types. In *Proceedings of Programming Language Design and Implementation(PLDI '98), Montreal, June 1998.*, pages 214–227, 1998.