The Epigram Prototype: a nod and two winks

Conor McBride

April 23, 2005

1 Introduction

I intend this document as a guide to the prototype EPIGRAM system. As things stand, there are still a great many things which need to be done, but the system can now be nursed through some useful small examples, so it seems appropriate to help people play with it.

1.1 What you need

To run EPIGRAM, you will need a compiled binary of the executable, and a suitable version of `xemacs`. I'm not an expert on `xemacs`, so I don't know which versions are suitable, but it's not hard to find out if yours is. I'm using 21.1 just now.

Epigram is developed under linux, and has been caught working under Windows XP (thanks Paul) and Mac OS X (thanks Michael). We now have a more stable web presence at

```
http://www.durham.ac.uk/CARG/epigram
```

Subscribers to

```
epigram@durham.ac.uk
```

get news of updates and things which do/don't/might one day work. Please do subscribe, and send `.epi` files, successful or otherwise.

To compile EPIGRAM, you will need a reasonably recent version of the Glasgow Haskell Compiler, `ghc`: I use lots of non-98 language extensions very heavily, so other systems probably won't play. Version 5.02 is certainly sufficiently recent—it's what I use at the moment.

2 The general picture

The EPIGRAM system is an interactive editor and interpreter for the EPIGRAM programming language, largely as defined in *The view from the left* [MM04], or *VFL*, for short. I'm very bad at writing editors, so I cheated—EPIGRAM runs as an inferior process under `xemacs`. You start it by running the script

```
epigram
```

which starts a new `xemacs`, with the EPIGRAM buffer in the foregroung, and the actual interaction happening in another buffer offstage. You can look at it if you like to watch computers working, but it may slow things down a touch. If you do look at it, you'll see that typing stuff in the EPIGRAM buffer causes event messages to be sent as textual input to the underlying process, which spits back much Elisp in response.

Mea culpa. This rather Heath-Robinson arrangement can be a little flaky at times, but that's how it is until some kind soul guis it up. It's not as dreadful as it was.

Important. The EPIGRAM buffer is read-only as far as `xemacs` is concerned. Keyboard input in that buffer is redirected. If you want to **quit** the EPIGRAM process, hit **Alt-Escape** in the EPIGRAM buffer. If you want to quit `emacs`, either use the menu, or switch to a buffer you can genuinely type in. Also useful is **Ctrl-Backspace**, which refreshes the display: use this if it garbles on you, or if you change the width of the frame.

2.1 What you see

EPIGRAM's syntax is two-dimensional. The EPIGRAM buffer contains a **document** with a rectangular region selected—highlighted with a bright background. A

document is a vertical sequence of lines; a

line is a horizontal sequence of boxes; a

box is either a character, or a bracket containing a document.

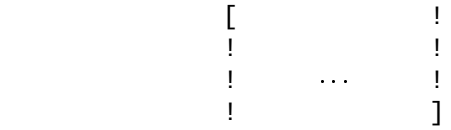
Hence, an EPIGRAM line can occupy more than one physical line in the buffer. You can combine two EPIGRAM lines on one by separating them with `;`, and split one into two by prefixing the second with `%`. A bracket is either a

group which has the usual functions of parenthesis

```
(               !
!               !
!       ...     !
!               )
```

or a

shed where you can tinker with text.



The EPIGRAM document is a **concrete syntax** tree, in which some leaves are sheds. The part of the document outside the sheds is managed, typechecked, typeset, coloured in and generally mucked about with by EPIGRAM; the content of a shed is monochrome, but it belongs to you and you may edit it freely.

At the top level, the document is a vertical sequence of **declarations** delineated by **rules**. A **rule** is a sequence of at least three `---`. The initial document has just one declaration, consisting of a shed, waiting for you to start work.

Important. If you want to open a shed for a new declaration, double-click on a rule, or select a rule and just start typing. Clicking below the buffer contents should select the final rule.

Of course, your program won't be happy if you keep it in the shed. You can let it escape to a more fulfilling existence by pressing the **Escape** key. If your document parses correctly, EPIGRAM will check it, and replace your lovely typesetting with something ghastly that it likes better. If your document does not parse, nothing happens. Note that the contents of *nested* sheds are still locked away. If you haven't finished your program, you can leave sheds (not necessarily empty) for the unfinished parts and check the rest anyway—EPIGRAM always tries to give as much feedback as possible, as early as possible. Nested sheds allow you to work in little steps or big steps as you see fit.

Mea culpa. I realize that some syntax error diagnostics might be nice.

Let's look at an example

```
------------------------------------------------------------------------------
                                        (   n : Nat   !
data (---------!  where (------------! ;  !-------------!
     ! Nat : * )        ! zero : Nat )    ! suc n : Nat )
------------------------------------------------------------------------------
     (   x, y : Nat   !
let  !---------------!
     ! plus x y : Nat )

     plus x y <= rec x
     { plus x y <= case x
       { plus zero y []
         plus (suc n) y => suc (plus n y)
       }
     }
------------------------------------------------------------------------------
inspect plus (suc (suc zero)) (suc (suc zero)) => suc (suc ?) : Nat
------------------------------------------------------------------------------
```

Here, you can see the three available kinds of declaration:

data declarations declare data structures by giving the

formation rule which declares a **type constructor** and the

introduction rules which declare the **data constructors** by which the data structure acquires data; **data constructors**

let declarations define programs by giving a

type signature which establishes the type of the intended program and a

decision tree which establishes the strategy by which the program will deliver output, given input;

inspect declarations allow you to inspect the value of an expression, so far as it has one.

So, what have we here? We have the natural numbers, declared inductively; we have begun to define addition by recursion and case analysis on its first argument, but we have not filled in the zero case—there's just a shed; we would like to know what $2 + 2$ is. You can select the shed by clicking on it. If you then type

```
=> y
```

and press Escape, you will complete both the program and the example.

Important. Alt-Backspace is 'undo last action'. **Ctrl-Alt-Return** is 'restart', and you can undo it.

Mea culpa. My Swedish friends will doubtless emphasize the importance of having the 'local undo' operation, which takes a selected region and pulls it back into a shed. I agree entirely: this is high on my hit-list, and when it is done, it morally ought to be bound to the Control-Escape key. It's quite tricky, because the deletion of information can have highly non-local effects, but there's an obvious naïve and inefficient way to do it, which I propose to start with...

Try undoing the 'return y' step and sending

```
<= case y
```

instead. This expands the decision tree to

```
plus x y <= rec x
{ plus x y <= case x
  { plus zero y <= case y
    { plus zero zero []
      plus zero (suc n) []
    }
    plus (suc n) y => suc (plus n y)
  }
}
```

That is, we have adjusted the zero-case strategy to require a case analysis on the second argument, and we now have two new subproblems to solve. If you send

```
=> suc (plus zero n)
```

for the zero-suc case, you will see that its background remains yellow, which is EPIGRAM's way of expressing doubt. Here, it does not know why the recursive call makes sense—if we want to do recursion on the second argument, we need to say so. If you undo again, you can build this program instead

```
plus x y <= rec x
{ plus x y <= case x
  { plus zero y <= rec y
    { plus zero x <= case x
      { plus zero zero => zero
        plus zero (suc n) => suc (plus zero n)
      }
    }
    plus (suc n) y => suc (plus n y)
  }
}
```

Mea culpa. EPIGRAM is very bad at choosing names. I plan both to make it much better at choosing names, and to allow you to do systematic α-conversion (outside sheds) by direct manipulation.

I hope this shows something of the flavour of EPIGRAM programm*ing*, not just programs. An EPIGRAM program is presented as a didactic dialogue between you and your mechanical student. The left-hand sides which the machine generates are questions in search of explanations which you provide on the right. I've been programming this way with human students for years, and it seems to do them some good. But I accept that it's a little prolix typing `<= case x`. Obviously `plus zero y`, `plus (suc n) y` is so much shorter.

2.2 Working in a shed

When a shed is selected, you can edit its contents by typing. As I said, I hate writing editors, so for now you get ordinary typing, plus arrow keys, backspace and delete. You'll find that if you type **(** or **[**, the corresponding **]** or **)** appears also, with the cursor between them—you're now editing an inner document. The **Return** key makes a new line in the local document; you can get out of the document by using the arrow keys, or directly, by typing a closing bracket. You can also jump around with the mouse.

You can't delete a bracket from inside it—you can backspace over it or delete it from in front. The editor maintains the width of a bracketed document as the maximum of the width of its lines, so you don't need to worry about aligning the `!` marks. Also, if you type a rule inside a bracket, the editor aligns it vertically with the text outside the bracket. Apart from that, things are quite normal, really.

Important. Some keystrokes you can use in another **xemacs** buffer:

Ctrl-c Ctrl-c sends the current buffer's contents to a selected empty shed in the EPIGRAM buffer—this enables you to load work.

Ctrl-c Ctrl-r sends the currently selected region in the same way.

Ctrl-c Ctrl-e replaces the current buffer's contents by those of the EPIGRAM buffer—this enables you to use 'ordinary' editing, and also to save your work.

Mea culpa. I know that it's not obvious whether the machine is rejecting its input or just thinking very hard. This is, however, discernable, either by looking in the Epigram-bin buffer, where you'll see if the escape key event has had its response yet, or by trying to type in the troublesome shed. Note that a buffer which isn't properly bracketed will be rejected by Ctrl-c Ctrl-c. Meanwhile, if your big shedload doesn't parse, do try sending it a bit at a time.

2.3 Working in `xemacs`

In the absence of local-undo, it's a reality that tinkering tends to happen offstage, in `xemacs`. In that respect, I've implemented a few low-budget gadgets to make life a little easier. Firstly, **Ctrl-Alt-Return** undoably returns EPIGRAM to the blissful state of ignorance it was born with. You can fire in a new version of your work when you're done tinkering. To help you tinker, I've added some `xemacs` keystrokes which exploit its support for *rectangles*, selected by dragging from top-left to bottom-right—the selection won't look but rectangular, but these will work all the same:

Ctrl-c r rectangle to selected empty shed

Ctrl-c Ctrl-s shed rectangle

Ctrl-c Ctrl-b bracket rectangle

3 Concrete syntax

This is the useful concrete syntax so far. It's mostly an ASCIIfication of that in *VFL*. The parser is slightly more liberal than this would suggest.

Mea culpa. Omitted here are some bits of concrete syntax which do not yet have elaborators, notably tuples. There are also some forms of term which do not yet have concrete syntax.

Important. The concrete syntax as presented here is not considered to be stable, although changes will be floated on the mailing list first and signalled in future versions of this document. Too many prototypes do suffer from bad syntax whose only justification is legacy. Not this one. We just wire up the old parser to the new renderer...

3.1 Terms

$term$	$=$	$head\ seq[head, \varepsilon]\ opt[\texttt{:}\ term]$	application
	$\mid$	`all` $sigs$ `=>` $term$	universal quantification
	$\mid$	`lam` $sigs$ `=>` $term$	lambda abstraction
	$\mid$	`ex` $sigs$ `=>` $term$	existential quantification
	$\mid$	$head\ seq[head, \varepsilon]$ `->` $term$	simple function space
	$\mid$	$head\ seq[head, \varepsilon]$ `=` $head\ seq[head, \varepsilon]$	proof irrelevant equality
	$\mid$	$gadget\ term$	data eliminator
$head$	$=$	`*`	type of types
	$\mid$	$identifier$	variable
	$\mid$	`(` $term$ `)`	grouped term
	$\mid$	`Zero`	proof irrelevant empty type
	$\mid$	`One`	proof irrelevant unit type
	$\mid$	`()`	the proof of `One`
	$\mid$	`refl`	constructor for equations
	$\mid$	`_`	'let me be explicit'
	$\mid$	`?`	'go figure'
	$\mid$	`[` $document[term]$ `]`	
$gadget$	$=$	`case` $\mid$ `rec` $\mid$ `view` $\mid$ `memo` $\mid$ `gen`	

3.2 Declarations

$source$	$=$	$maybe[\texttt{---}]$
		$seq[decl, \texttt{---}]$
		$opt[\texttt{---}]$
$decl$	$=$	`data` $sigs$ `where` $sigs$
	$\mid$	`let` $sigs$ `;` $program$
	$\mid$	`inspect` $term\ opt[\texttt{=>}\ term]$
	$\mid$	`[` $document[source]$ `]`

3.3 Signatures

$sigs$	$=$	$seq[sig, \texttt{;}]$	
sig	$=$	$sigvar$	untyped declaration
	$\mid$	$sigvar$ `:` $term$	typed declaration
	$\mid$	`(` $deduction$ `)`	natural deduction rule
	$\mid$	`[` $document[sigs]$ `]`	
$sigvar$	$=$	$identifier$	normal bound variable
	$\mid$	`_` $identifier$	implicitly bound variable
$deduction$	$=$	$\dfrac{sigs}{proforma : term}$	
$proforma$	$=$	$sigvar\ seq[sigvar, \varepsilon]$	typical application
	$\mid$	`[` $document[proforma]$ `]`	

3.4 Programs

$program$	$=$	$head\ seq[head, \varepsilon]\ rhs$	
	$\mid$	`[` $document[program]$ `]`	
rhs	$=$	`=>` $term$	return value
	$\mid$	`<=` $term\ programs$	by eliminator
	$\mid$	`[` $document[rhs]$ `]`	
$programs$	$=$	ε	no programs
	$\mid$	`{` $seq[programs, \texttt{;}]$ `}`	some programs

3.5 Utilities

$opt[cat]$	$=$	ε
	$\mid$	cat
$seq[cat, sep]$	$=$	ε
	$\mid$	$cat\ more[cat, sep]$
$more[cat, sep]$	$=$	$opt[sep\ cat\ more[cat, sep]]$

4 Declarations

What can you put in a declaration shed? A sequence of declarations (including declaration sheds), separated by `---`. Let's now look in a little more detail at the varieties of declaration currently supported.

4.1 `data`

Declarations of inductive families[Dyb91] take the form

`data` *formationSig* `where` *constructorSigs*

That is, you first declare the constructor for the family of types you are declaring, which appears in Blue, followed by the constructors of the data which populate it, which appear in red.[1]

An inductive family is an indexed collection of *mutually* defined inductive types. The type constructor must construct types, so its declaration typically takes the form of a **deduction**:

$$\left(\frac{\Xi}{\mathsf{D}\ \Xi\ :\ \star}\right)$$

Constructor declarations generally resemble

$$\left(\frac{\Delta}{\mathsf{c}\ \Delta\ :\ \mathsf{D}\ \vec{s}}\right)$$

As in *VFL*, Greek capitals stand both for a sequence of declarations (as above the line) and for the argument-sequence of variables so defined (as below it). We have already seen

$$\underline{\text{data}}\ \left(\frac{}{\mathsf{Nat}\ :\ \star}\right)\ \underline{\text{where}}\ \left(\frac{}{\mathsf{zero}\ :\ \mathsf{Nat}}\right)\ ;\ \left(\frac{n\ :\ \mathsf{Nat}}{\mathsf{suc}\ n\ :\ \mathsf{Nat}}\right)$$

This notation is a little more prolix than Haskell, say, for simple definitions, but with EPIGRAM's implicit syntax technology—more about this shortly—it pays off in spades the more dependent your types get. In any case, there is not yet any good reason why you should not write

$$\underline{\text{data}}\ \ \mathsf{Nat}\ :\ \star\ \ \underline{\text{where}}\ \ \mathsf{zero}\ :\ \mathsf{Nat}\ ;\ \ \mathsf{suc}\ :\ \mathsf{Nat} \to \mathsf{Nat}$$

By the way, if you just type `data` in a declaration shed, you will get a template for a data declaration.

EPIGRAM will only accept **strictly positive** families. That is, a constructor's argument types must either be **nonrecursive**, not mentioning D, or **recursive**—function types

[1] For those of you watching in black-and-white, blue is represented by sans-serif type and an initial capital letter, whilst red has the same typeface, but a lower case initial.

(possibly with no arguments) returning some $\mathsf{D}\ \vec{r}$, but with no other reference to D.

Note, these are 'old Swedish' families, rather than the more puritan 'new Swedish' families, which allow constructors only the 'unfocused' return type $\mathsf{D}\ \Xi$, rather than an arbitrarily 'focused' $\mathsf{D}\ \vec{t}$. I propose **focused** and **unfocused** as the terminology we have long been lacking in discussing this distinction, although Peter Hancock prefers 'Roman Catholic' and 'Presbyterian'. The fact remains that Catholicism is just more fun.

Mea culpa. EPIGRAM does not currently accept mutually defined type families, but the fact that an inductive family is a family of mutually defined types does at least provide a workaround.

When the formation signature and constructor signatures have been successfully elaborated, EPIGRAM generates the case, rec and view operators for the type family. As these are what make data data, data successfully elaborates too.

Remark. I have implemented (barring mutual definition) a fairly standard notion of inductive family, but I intend to extend this eventually to induction-recursion [DS01], and indeed to *dependent* mutual definition of families (my favourite example being $\mathsf{Ctxt} : \star;\ \mathsf{Type} : \mathsf{Ctxt} \to \star;\ \mathsf{Term} : \forall \Gamma : \mathsf{Ctxt} \Rightarrow \mathsf{Type}\ \Gamma \to \star$). And then there's **codata**... and *then* there's mixed data-codata families (modelled by alternating least and greatest fixpoints). A big zoo is a happy zoo.

4.2 Signatures

The basic form of a **signature**, whether in a data or let declaration, or just inside λ, $\forall$ or $\exists$, is

$$x\ :\ T$$

declaring a variable in a type. You may omit the $: T$ if you want the machine to attempt to infer the type—there is no guarantee that it will succeed. You may also prefix the variable $\_x$ to indicate that it is *implicit*. Moreover, you may group variables $x_1, x_2, \ldots, x_n$ if they are intended to share the *same* type. Signatures form sequences vertically (thus horizontally if separated by ;).

EPIGRAM also supports a first-order syntax for declaring higher-order objects—**deductions**, which generally resemble

$$\left(\frac{\Delta}{f\ \Delta\ :\ T}\right)$$

meaning

$$\forall \Delta \Rightarrow T$$

where Δ is a signature sequence, perhaps itself including deductions, and $f\ \Delta$ is a

proforma—a 'typical application' of the declared symbol to variables.

However, by a technological miracle, you get to omit 'sufficiently obvious' signatures from above the line—you may also omit variables from below the line if you wish them to be treated as implicit arguments; you may not permute the variables from the order in which you declare them. The technological miracle is, of course, Hindley-Milner-Damas type inference [DM82] over a Miller-style mixed quantifier prefix [Mil92], a combination which I gather Frank Pfenning came up with.

So, we get

$$\underline{\mathsf{data}}\ \left(\frac{X\ :\ \star}{\mathsf{List}\ X\ :\ \star}\right)\ \underline{\mathsf{where}}\ \left(\frac{}{\mathsf{nil}\ :\ \mathsf{List}\ X}\right)\ ;\ \left(\frac{x\ :\ X\ ;\ \ xs\ :\ \mathsf{List}\ X}{\mathsf{cons}\ x\ xs\ :\ \mathsf{List}\ X}\right)$$

and even

$$\underline{\mathsf{data}}\ \left(\frac{x\ :\ X\ ;\ \ xs\ :\ \mathsf{List}\ X}{\mathsf{In}\ x\ xs\ :\ \star}\right)$$
$$\underline{\mathsf{where}}\ \left(\frac{}{\mathsf{now}\ :\ \mathsf{In}\ x\ (\mathsf{cons}\ x\ xs)}\right)\ ;\ \left(\frac{tl\ :\ \mathsf{In}\ x\ xs}{\mathsf{later}\ tl\ :\ \mathsf{In}\ x\ (\mathsf{cons}\ y\ xs)}\right)$$

That's a whole bunch of symbols we didn't need to declare—and the element types didn't get mentioned in the constructor signatures at all. For the type of well-ordered trees, we get

$$\underline{\mathsf{data}}\ \left(\frac{B\ :\ A \to \star}{\mathsf{W}\ B\ :\ \star}\right)\ \underline{\mathsf{where}}\ \left(\frac{f\ :\ B\ a \to \mathsf{W}\ B}{\mathsf{sup}\ a\ f\ :\ \mathsf{W}\ B}\right)$$

which is just as well, as the standard $\mathsf{W}\ A\ B$ was a rude word when I was growing up. Note that although a is not declared, it's sensible to make it an explicit argument of sup as it cannot usually be inferred unless B is a partially applied type constructor.

This is just to emphasize that, whilst type inference as a global property of a dependently type language is a non-starter—Haskell is sufficiently dependently typed not to possess it—the technology underlying type inference still buys us a great deal. The idea is now to say enough to make the *plan* clear, then let the machine help you follow the plan.

Note that although one usually just writes the explicit arguments in the proforma, you can be clear about the order of the implicit arguments too, by prefixing them with $\_$. You could write $\mathsf{sup}\ \_A\ \_B\ a\ f$ to have the same effect more explicitly. Sometimes, if you leave this information out, there is more than one choice available to the machine for where and in which order the implicit arguments should be quantified. The current implementation keeps them as far to the right as possible; a previous version pushed them as far left as possible. I make no promises—if you want to rely on a particular choice and there is more than one, better to make the proforma explicit rather than trusting to luck.

One more note of caution: if you are using nested deductions, implicit variables are by

default quantified as locally as possible. For example, you could try declaring

$$\underline{\text{data}} \quad \left(\frac{\left(\frac{a \ : \ A}{B \ a \ : \ \star} \right)}{\mathsf{NotW} \ B \ : \ \star} \right) \quad \underline{\text{where}} \quad \left(\frac{\left(\frac{b \ : \ B \ a}{f \ b \ : \ \mathsf{NotW} \ B} \right)}{\mathsf{sup} \ a \ f \ : \ \mathsf{NotW} \ B} \right)$$

but the machine would try to quantify A locally to the first nested deduction, and everything but f in the second: you would not get what you want.

$$\underline{\text{data}} \quad \left(\frac{A; \ \left(\frac{a \ : \ A}{B \ a \ : \ \star} \right)}{\mathsf{W}' \ B \ : \ \star} \right) \quad \underline{\text{where}} \quad \left(\frac{B; \ a; \ \left(\frac{b \ : \ B \ a}{f \ b \ : \ \mathsf{W}' \ B} \right)}{\mathsf{sup} \ a \ f \ : \ \mathsf{W}' \ B} \right)$$

is the form equivalent to the original. One might argue that a and B are obviously meant to be quantified in the outer deduction as they are mentioned in its conclusion, but this requires extracting information from the conclusion before it is in scope.

Finally, on stepwise editing for signatures. Basically because I'm lazy, you can refine a signature shed to a deduction template by sending

$$\frac{\square}{\square \ : \ \square} \quad \text{but not} \quad \frac{\square}{\square}$$

I think it saved me one syntactic category, or something.

Note also that you can type a (vertical) sequence of signatures (or signature sheds) into a signature shed.

4.3 `let`

You can introduce a definition, and in particular, a program with a `let` declaration, of the form

$\underline{\text{let}}$ *functionSig*

program

If you omit the program, EPIGRAM will generate an opening programming problem from the signature as soon as it elaborates. You can get a template for the signature by sending just `let` to a declaration shed.

The initial programming problem is constructed directly from the proforma in a deduction signature; if you give a direct $f \ : \ T$ signature, EPIGRAM presumes that you intend to give a direct term for f.

You might send

$$\underline{\text{let}}\ \left(\frac{x, y\ :\ \mathsf{Nat}}{\mathbf{plus}\ x\ y\ :\ \mathsf{Nat}}\right)$$

and you will get the starting problem

$$\underline{\text{let}}\ \left(\frac{x, y\ :\ \mathsf{Nat}}{\mathbf{plus}\ x\ y\ :\ \mathsf{Nat}}\right)\ ;\ \mathbf{plus}\ x\ y\ []$$

ready to go!

More on programming in section 6.

EPIGRAM relies on and carefully maintains the well-foundedness of the context. Recursive programs aren't really recursive at all! The recursion in EPIGRAM comes ultimately from the rec gadget you get with a datatype, which introduces a memo-structure—recursive calls are just syntactic(?) sugar for the extraction of the appropriate value from a memo-structure. We shall extend this approach to explicitly grouped top-level mutually recursive definitions (by allowing let to take multiple signatures and programs) in due course. The main design choice required is in developing the syntax to explain how the recursive programs share a common recursive strategy, giving each program access to the appropriate memo-structure for the others.

We shall also be extending terms to support local let and programs to support local helper functions via some kind of 'where clause'.

4.4 `inspect`

The `inspect` declaration enables you to run programs you have written.

$$\underline{\text{inspect}}\ \ t$$

where t is a term, elaborates to

$$\underline{\text{inspect}}\ \ t\ \Rightarrow\ v\ :\ T$$

where v is t's value and T is its type.

As it stands, `inspect` can only be expected to behave sensibly when t elaborates fully in terms of objects which have been completely defined, but it's inevitable from the way EPIGRAM is implemented that `inspect` can also be used to run programs you *haven't* written.

Broadly speaking, if there are pieces missing from the program, you will get strange orange ?s in the value of your term (and possibly its type). As more information becomes available, you will see these being instantiated. What does not happen at the moment is that incomplete terms which become reducible on the instantiation of their holes will be replaced by their values—the holes will be instantiated but the reduction

will not visibly take place. I shall attend to this oversight in due course. Moreover, the mysterious orange ?s should be rendered as the lumps of unelaborated concrete syntax to which they correspond. This will allow to run programs which contain type errors and see where those errors bite *by example*. Years spent listening to undergraduates whingeing that 'the typechecker won't let my program work' have obviously had some impact.

5 Terms

This section examines a little more closely the syntax of EPIGRAM terms, and how the elaborator treats them. In particular, it outlines what happens with implicit syntax.

5.1 Basics

We start with a fairly basic type theory:

$$\frac{}{\vdash} \quad \frac{\Gamma \vdash S : \star}{\Gamma; x : S \vdash} \quad \frac{\Gamma; x : S; \Delta \vdash}{\Gamma; x : S; \Delta \vdash x : S}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash \beta S =_\star \beta T}{\Gamma \vdash s : T} \quad \frac{\Gamma \vdash}{\Gamma \vdash \star : \star} \quad \frac{\Gamma; x : S \vdash T[x] : \star}{\Gamma \vdash \forall x : S \Rightarrow T[x] : \star}$$

$$\frac{\Gamma; x : S \vdash t[x] : T[x]}{\Gamma \vdash \lambda x : S \Rightarrow t[x] : \forall x : S \Rightarrow T[x]} \quad \frac{\Gamma \vdash f : \forall x : S \Rightarrow T[x] \quad \Gamma \vdash s : S}{\Gamma \vdash f\ s : T[s]}$$

The simple function space $S \to T$ is just syntactic sugar for $\forall x : S \Rightarrow T$.

Yes, with $\star : \star$, this system is inconsistent, but it's a good place to start. EPIGRAM implements these rules: here β computes β-normal forms and $=_T$ is the η-equality for type T. In particular

$$\frac{\Gamma; x : S \vdash \beta(f\ x) =_{T[x]} \beta(g\ x)}{\Gamma \vdash f =_{\forall x:S \Rightarrow T[x]} g}$$

5.2 Datatypes

The effect of a data declaration is to add the type- and data-constructors to the context, generate the appropriate case and rec operators, and extend β appropriately.

If we have

$$\underline{\text{data}} \ \frac{\Xi}{\mathsf{D}\ \Xi : \star} \ \underline{\text{where}} \ \cdots \ \frac{\Delta_i}{\mathsf{c}\ \Delta_i : \mathsf{D}\ \vec{s}} \ \cdots$$

we get

$$\frac{x \,:\, \mathsf{D}\,\vec{t}}{\begin{array}{l}\underline{\text{case}}\ x \,:\ \ \forall\ P \,:\, \forall \Xi \Rightarrow \mathsf{D}\ \Xi \to \star \\ \qquad\vdots \\ \qquad m_i \,:\, \forall \Delta_i \Rightarrow P\ \vec{s}\ (\mathsf{c}\ \Delta_i) \\ \qquad\vdots \\ \qquad \Rightarrow P\ \vec{t}\ x\end{array}}$$

Note that $\underline{\text{case}}\ x$ is a term, but $\underline{\text{case}}$ is not. Its type actually explains *how* to do case analysis on x. The computational behaviour is as follows:

$$\frac{\Gamma \vdash \beta x \twoheadrightarrow \mathsf{c}_i\ \vec{a}}{\Gamma \vdash \beta(\underline{\text{case}}\ x) \mapsto \lambda P; \vec{m} \Rightarrow m_i\ \vec{a}}$$

Let us leave $\underline{\text{rec}}$ aside for the moment, but it too has a type which explains *how* to do structural recursion.

5.3 Implicit Syntax

Implicit syntax in EPIGRAM is inspired by Pollack's treatment in LEGO [Pol92], but differs in several ways. Firstly, EPIGRAM has a *separate* implicit function space:

$$\frac{\Gamma; x : S \vdash T[x] \,:\, \star}{\Gamma \vdash \forall \_ x : S \Rightarrow T[x] \,:\, \star}$$

$$\frac{\Gamma; x : S \vdash t[x] \,:\, T[x]}{\Gamma \vdash \lambda \_ x : S \Rightarrow t[x] \,:\, \forall \_ x : S \Rightarrow T[x]} \qquad \frac{\Gamma \vdash f \,:\, \forall \_ x : S \Rightarrow T[x]}{\Gamma \vdash f\ \_ \,:\, \forall x : S \Rightarrow T[x]}$$

$$\frac{\Gamma \vdash \beta S \mapsto S' \qquad \Gamma; x : S \vdash \beta(t[x]) \mapsto t'[x]}{\beta((\lambda \_ x : S \Rightarrow t[x])\ \_) \mapsto \lambda x : S' \Rightarrow t'[x]}$$

The postfix operator $\_$ makes an implicit function explicit. The elaborator silently **completes** any term whose type is implicitly quantified, unless it is of form $t\_$ and thus made explicit. Completion works by sticking in metavariables (or 'holes') for the implicit arguments and hoping they get solved by unification.

Other forms of implicit syntax—omitting types in quantifiers, etc, also work via metavariables. You can reliably leave off the quantifiers from λ unless it is being applied—this information is available in the type.

Mea culpa. The background colour of a completed term should indicate whether the machine has been successful at inferring the missing arguments, but I haven't wired that up yet. Also, you should ideally be able to expose implicit arguments—by double-clicking a completed term, say—in order to edit those which have not been successfully inferred and need to be supplied explicitly.

Note that, in an application, if an argument of implicit type is *required*, the argument you supply is elaborated under an implicit λ. The bound variable has no name, but it

can (and hopefully will) be used in the values inferred for implicit arguments.

Important. The implicit syntax mechanism makes a key simplifying assumption when it is inferring missing information: *A metavariable never stands for an implicit function type.* Without some such assumption, you get deadlock: if the type of an argument is a metavariable, you can't complete it to form the type which gets unified with the domain of the function to solve the metavariable. With the assumption, an argument of unknown type is complete already. It's common sense: the computer isn't as clever as you, so you should only expect it to solve easy problems by itself.

6 Programs

EPIGRAM programs are trees which explain how to solve a programming problem. Perhaps the easiest way to explain what is going on is to tell it pretty much like it is. If you supply the signature

$$\underline{\text{let}}\ \frac{\Delta}{\mathbf{f}\ \Delta\ :\ T}$$

then the initial programming problem is

$$\forall \Delta \Rightarrow \langle \mathbf{f}\ \Delta\ :\ T \rangle$$

This funny type in angle-brackets what *VFL* calls a **labelled type**, and it means 'the T that is $\mathbf{f}\ \Delta$'. These labels $\mathbf{f}\ \vec{p}$ are used to compute the left-hand sides of programs. The $\vec{p}$ are the **patterns**, and there is no a priori reason to presume that they are linear or consist solely of constructor forms. The right-hand side of a program line explains how to split a programming problem into zero or more subproblems: it elaborates to an LCF-style tactic, pairing a collection of subproblems with a combinator which computes the solution to the original problem from those of the subproblems.

One way to solve a programming problem

$$\forall \Delta \Rightarrow \langle \mathbf{f}\ \vec{p}\ :\ T \rangle$$

is to supply a **return**,

$$\Rightarrow t \qquad \text{where } \Delta \vdash t\ :\ T$$

so that your program looks like

$$\mathbf{f}\ \vec{p} \Rightarrow t$$

However, you can also split a programming problem by invoking an **eliminator**:

$$\Leftarrow e$$

where the eliminator e is any expression whose type looks like this

$$
\begin{array}{l}
\forall P \ : \ \forall \Xi \Rightarrow \star \\
\quad m_1 \ : \ \forall \Delta_1 \Rightarrow P \ \vec{s}_1 \\
\quad \vdots \\
\quad m_n \ : \ \forall \Delta_n \Rightarrow P \ \vec{s}_n \\
\Rightarrow P \ \vec{t}
\end{array}
$$

We call $\vec{t}$ the **targets** of the eliminator, because they are what it eliminates; P is called the **motive** because it stands for what we hope to achieve by the elimination; the m_i are the methods—they explain how to achieve the motive in each possible circumstance.

Now, EPIGRAM has a built-in type which captures the idea of **equality**. When you invoke $\Leftarrow \ e$ for our typical programming problem, EPIGRAM takes

$$P \mapsto \lambda \Xi \Rightarrow \forall \Delta \Rightarrow \Delta = \vec{t} \rightarrow \langle \mathbf{f} \ \vec{p} \ : \ T \rangle$$

This makes

$$m_i \ : \ \forall \Delta_i ; \Delta \Rightarrow \vec{s}_i = \vec{t} \rightarrow \langle \mathbf{f} \ \vec{p} \ : \ T \rangle$$

EPIGRAM then tries to solve $\vec{s}_i = \vec{t}$ by applying the substitutivity for equality and the disjointness and injectivity of equality, which is how the patterns in the subproblems become more instantiated. Moreover, in the case where the equations are clearly false, the subproblem disappears—this is why there is no 'nil' case when you implement 'tail' for nonempty length-indexed lists.

More details of this process can be found in *VFL*, or [McB00].

The point here is that the types of eliminators provide an *abstract* interface to the decomposition of programming problems. The eliminator itself is a higher-order function, computing the solution to the main problem from its subproblems. Constructor pattern matching is no longer hardwired into the compilation of functions: it is just one possible eliminator, generated for free with every datatype. The fun with EPIGRAM really starts when you roll your own.

VFL also gives a third way to solve programming problems: the 'with' rule, which allows you to abstract the value of an intermediate computation, adding it to the collection of values under scrutiny on the left. In first-order programming, this often eliminates the need for local case-expressions in right-hand-side terms. That's just as well, because local case-expressions don't really make sense in dependently typed programming—inspecting one value tells you more about other values too, and about the type of the thing you're trying to construct. Good! What's the point of testing something if it doesn't make any difference?

7 Things to do

This is a proportion of the current wish list. It gets longer more quickly than it shrinks.

7.1 Known bugs

Things which are supposed to work but are known not to. Don't hesitate to tell me if something goes wrong. I generally respond as quickly as humanly possible.

Currently, there's some funny business with the equation-solving in the <= rule. We'll see.

Important. The patterns you get when you apply an eliminator arise from equation solving. If the machine cannot solve the equations, then you will get less instantiation of patterns than you might wish for. This is *correct*. One benefit of Epigram's internal explanation of pattern matching by elimination with equational constraint is that you are not completely scuppered when this happens, as was the case in ALF. If the troublesome equation involves defined functions, it's not a big surprise that the machine is not clever enough—it's not as clever as you. If you can do another elimination which unblocks the function call, you may find the rest of your pattern appearing.

On the other hand, if the equation really ought to have a solution by standard first-order constructor methods, I'll put my hand up to a problem. If you suspect this to be the case, then it's the usual story: send me a bogey, and I'll Alt-Return it. If you feel intrepid, Alt-Return it yourself. No—I haven't implemented no-cycle yet, but I'm planning to cheat.

There's also a bit too much rigidity when it comes to implicit arguments in natural deduction rules. An explicitly mentioned but implicitly declared argument which first appears to the right of where it is first needed will not be shunted backwards as it should be.

7.2 Elaboration of terms

There are several things with no elaborators yet:

Tuples Existential quantification is already elaborated. The parser knows about tuples but the elaborator doesn't. A tuple will be a vertical group of terms. Implicit elements will be preceded by _. Right-nesting will be flattened. Tuples occurring where type information is being pushed inwards should not need type annotations. Others might, and at the moment, I'm trying to think of a neater way than slapping a huge cast on the whole thing. Elimination—see left-hand-sides. **Needs care.**

Anonymous lambda I'm sure I can do a better job of type inference for these. **Needs care.**

7.3 Elaboration of left-hand sides

More patterns I have only implemented first-order patterns with functions/constructors and pattern variables. Any term can potentially be a pattern. **Needs care.**

η-expansion in patterns You should be able to match existential types with tuples, universals with lambda, etc, without any explicit eliminator on the right. Interactively, it should just be a double-click. **Needs care.**

Unhiding A space which conceals an implicit argument should cough it up if you double-click on it. **Easy job.**

α-conversion Double-clicking on a pattern variable should replace it by a shed; sending an acceptable variable name should cause α-conversion. Otherwise, shed stays open. Trouble is, if you save the file in that state, it won't reload properly. I reckon we need to put the old name back first, or something. **Needs some rejigging.**

Better names I have a heuristic in mind for guessing better names during interactive programming, based on the ones you use in data declarations. **Needs some rejigging.**

7.4 Elaboration of right-hand sides

With rule Abstracting intermediate computations. I need to write a typchecking abstractor. **Needs much care, but no damage.**

From rule Bringing the next argument to the left. **Easy job.**

Multi-by Doing a bunch of eliminations on one line. Ideally, you should be able to see what subproblems you would get if you stopped adding more. **Needs much care and some rejigging.**

Helper functions You should be able to invoke a helper function on the right-hand side which should automatically generate some kind of 'where clause'. The machine should be able to figure out the type signature if the usage is a Miller-pattern $(help\ var_1 \ldots var_n)$. You should be able to lift the helper function out to be a mutually defined top-level function. **Dream on.**

7.5 Elaboration of recursion

Matching Constraint-solving for recursive calls extracted from memo-structure should be by matching—any match will do. I haven't written a matcher yet, so I'm using unification, which is much too paranoid. **Needs care.**

Nesting Double recursion now works by stopping the motive-generator quantifying over memo-structures. This gives lexicographic recursion. A better solution would be to teach the thing to find memo-structures. See 'resource-bounded proof search'.

7.6 Proof search for things implicit

At the moment, this applies η-rules, but apart from that, it just hopes unification does the business.

Equational simplification should work the same way as with programming problems; constructor equations to be proven should also be split; reflexivity should be an automatic win. **Needs care.**

Vacuity having the empty type as a hypothesis should be an instant win; having a refutation as a hypothesis should start a proof search. **Dream on.**

Resource-bounded proof search A reasonable strategy seems to me to limit backchaining to things you haven't used yet in that branch of the proof. The context is our initial set of gadgets. Of course, the context gets bigger in higher-order proof search: should we be worried? You can use local definition (when I've implemented it) to copy things which need to be used more than once. **PhD student?**

7.7 Syntactic mod cons

Infix operators Especially `->`. I think I know what I'm going to do...later. My inclination is to implement *postfix* operators which have a *pull* to the left and deliver a term of a given *weight*. Postfix operators apply to the largest term to the left whose weight is $\leq$ their pull. Variables and groups weigh 0. An application weighs what the function weighs. Operators are presumed prefix unless declared postfix. Exactly what a postfix declaration should look like, I'm not sure. I'd like to be able to use relative pulls and weights, rather than raw numbers. Scary idea: make postfix info live in function types—that way, you can make all sorts of daft things happen—no, maybe not.

Comments Just now, the best you can do is keep top-level comments in declaration sheds. I wonder how we could do better. Of course, comments should really be handled by the user interface (see below). Comments should also be able to quote terms which are subject to update/α-conversion.

7.8 User interface

I'm planning a bit of refactoring in order to facilitate the clean separation of the user interface. The interface will communicate with the elaborator using the concrete syntax

as an interchange language. The interface will be entirely responsible for both layout and parsing, and will keep the contents of sheds to itself. Concrete syntax trees will have nodes labelled with the names of elaboration problems/solutions. Node replacement will be the main sort of message that the elaborator sends the interface; node selection and shed instantiation will be the main messages the other way. This will facilitate:

Local undo Interface reverts a node to text in a shed and sends an undo event to the elaborator. The elaborator reverts everything else to concrete syntax and starts again. There are some obvious safe but not subtle ways to do a bit more information accountancy and ensure that only a few bits of the buffer get redone, but a really precise story here is future work.

Local information There should be another buffer/window/whatever which displays local information pertinent to the selected subobject: eg value (if known), (desired) type, what's in the context, what's in the context that might solve the goal, unsolved constraints, type errors for brown things, syntax errors for sheds which won't escape, goodness knows what else. In particular, when you sit at an empty shed, there should be a menu of candidate lumps of concrete syntax you might like to choose from.

Filing You should be able to select a contiguous bunch of declarations and send them to a file, causing them to be replaced with an import declaration. The elaborator should be none the wiser.

7.9 Exporting Epigram

It should be possible to persuade Epigram to export piles of stuff to:

LaTeX The interface should be able to generate LaTeX source from concrete syntax. **Easy to do badly.**

Coq It should not be too hard to make Epigram generate Coq .o files, although we have a few eta-rules they don't, which might put some sand in the vaseline. **Easy to get somewhere; hard to get everywhere.**

Edwin's compiler (I) For swift run-time-only code. **Should be easy.**

Edwin's compiler (II) For partial-evaluation-safe preloaded implementations of libraries. **Needs care.**

An independent typechecker Epigram's type theory isn't too outlandish. I already have an ugly-printer for raw terms which I use when I'm looking under the bonnet. Any self-respecting PhD student working in this general area should not find it too hard to build their own checker. **Easy job.**

A Summary of keystrokes and mouse actions

In the EPIGRAM buffer:

Escape elaborate a shed
Alt-Backspace undo last change
Ctrl-Backspace refresh display
Alt-Escape quit
Alt-Return dump elaborator state
Ctrl-Alt-Return return Epigram to initial state (but you can undo)

Double click on:

a rule to get a new declaration shed

In another **xemacs** buffer:

Ctrl-c Ctrl-c buffer to selected empty shed
Ctrl-c Ctrl-r region to selected empty shed
Ctrl-c r rectangle to selected empty shed
Ctrl-c Ctrl-e replace current buffer's contents by EPIGRAM buffer
Ctrl-c Ctrl-s shed rectangle
Ctrl-c Ctrl-b bracket rectangle

By 'rectangle', I mean a rectangular selection traced out by clicking on the top-left corner and dragging to the bottom-left corner. The highlighting won't look rectangular, but as xemacs has some gadgets for manipulating rectangular selections, these things were too good to miss.

B Recent changes

lexicographic recursion now works (see above)

typing on a dividing rule now opens up a shed

clicking below the buffer now selects the final rule

new keystrokes see above

Empty type It's currently called `Zero`. It was already present in the type of values, and it's proof irrelevant. It is eliminated with `case`.

Unit type It's currently called `One`, and it's constructed by `()`. It's already present in the type of values, and it's proof irrelevant, so why bother eliminating it?

References

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[DS01] P. Dybjer and A. Setzer. Indexed induction-recursion. In *Proof Theory in Computer Science International Seminar, PTCS*, pages 93–113, 2001.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[HP90] G. Huet and G. D. Plotkin, editors. *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks (Antibes, France)*, May 1990.

[McB00] C. McBride. Elimination with a Motive. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for proofs and programs*, volume 2277 of *LNCS*, pages 197–216. Springer-Verlag, 2000.

[Mil92] D. Miller. Unification under a mixed prefix. *J. Symbolic Computation*, 14(4):321–358, 1992.

[MM04] C. McBride and J. McKinna. The view from the left. *J. of Functional Programming*, 14(1), 2004.

[Pol92] Robert Pollack. Implicit syntax. An earlier version of this paper appeared in [HP90], May 1992.