

A Fully Functional Description of Lucid

Stephen G. Matthews

Dept. of Computer Science
University of Warwick
Coventry, CV4 7AL, UK

email sgm@uk.ac.warwick.cs

ABSTRACT

The intensional language Lucid has a functional subset ISWIM(Lu(A)), however it cannot accommodate nested iteration. The two solutions proposed up to date are either a freezing declaration or a multidimensional algebra. The first is highly confusing and not functional, the second is cumbersome, and only "tantalizingly close" [W&A85] to functionality. This paper considers the reasons behind this problem. We argue that Lucid cannot be truly intensional unless ISWIM itself is made intensional. We propose such a system which avoids the necessity for either declarations or multidimensional algebras in a lucid style functional language.

1. Introduction

ISWIM (If you See What I Mean) [La64] [La66] is a very simple but elegant notation for describing computations. Starting with some algebra A , $\text{ISWIM}(A)$ is the set of all recursive definitions of first order recursive functions over A . Landin's contribution in ISWIM was to combine a syntax for such definitions with a semantics based upon evaluation, and a logic of programs based upon substitution in the λ -calculus. The proof of ISWIM's credentials are demonstrated by subsequent ideas on mainstream functional programming, in particular, graph reduction. What is not so impressive is that this technique seems to have a virtual stranglehold on the implementation of functional languages.

Lucid broke new ground by replacing the extensional algebra A by an intensional one, in this case a temporal one with domain ${}^{\omega}A$, denoted by $\text{Lu}(A)$. In effect this allowed ISWIM's extensional algebra to be interpreted in a context (i.e. time) and so allow for the alternative implementation technique of dataflow to be applied instead of the more usual graph reduction. The virtue of this will not be debated here, although we do take the position that the concept of intensionality in functional programming must be more fully developed before functional languages can be opened up to both wider use, and more efficient implementations.

Lucid's introduction of intensionality into ISWIM by means of streams (see below) introduced a curious dilemma. In order to demonstrate the credibility of intensionality it had to be shown that computational loops could be defined "iteratively" without recourse to the more usual recursive or higher order definitions of ISWIM. The archetypal (& much overworked) example is,

$$x = 1 \text{ fby } (x + 1)$$

This can be understood as an "iterative" definition in the sense of Kahn dataflow [Ka74], and should not be confused with an imperative iterative loop, e.g. a *while* loop. The dilemma was that such "iterative" loops could not be hierarchically composed (e.g. nested iteration) in $\text{ISWIM}(\text{Lu}(A))$, and so, surely, intensional algebras had little extra to offer above $\text{ISWIM}(A)$. The historical answer to this dilemma was to extend $\text{ISWIM}(\text{Lu}(A))$ in a non-functional manner, and so was born "freezing", instituted by the *is current* declaration. The experience of the author is that people find this notion highly confusing, and so cannot do any useful intensional programming in Lucid. We believe that Lucid has unintentionally misrepresented the potential for intensional functional programming by instituting freezing as a declaration. This hypothesis can be justified by considering the second suggested solution to the

above mentioned dilemma.

Freezing was introduced to overcome the unpleasant reality that nested time would not fit into the ISWIM(Lu(A)) framework. The second proposed solution forced nested time into this model by introducing multidimensional time, i.e. languages of the form ISWIM(Llu(A)), where Llu(A) has the domain,

$$\omega_{\omega} \rightarrow A$$

Mathematically this is much more acceptable. However, on programming grounds it is no more acceptable than freezing. We believe that time is inherently a one dimensional linear notion, and that it is unnatural to suggest that programming can take place in multidimensional time even if the mathematical framework is there. The mathematics works because the multidimensional time approach succumbs to a classic pitfall of functional languages. It is assumed that all transfer of information must be achieved via packaging and environments. In other words, the mathematics of multidimensional time works because it is a retreat to extensional ISWIM mathematics and as such is not intensional. Consequently we can hardly be surprised that it is no easier to programme with multidimensional algebras than it is with freezing.

The first lesson to be learned from Lucid is that intensional functional programming is possible by replacing the extensional algebra of an ISWIM language by an intensional one. The second is that intensional algebras such as Lu(A) are not enough, and thus we should not try to sidestep the problem by introducing declarations and multidimensional time. The final lesson is addressed in this paper. When designing an intensional functional language no extensional constructs such as those found in ISWIM can be regarded as sacred.

2. ISWIM⁺

This paper then is an exercise in meddling with ISWIM in order to obtain a notion of intensionality more extensive than an intensional algebra such as Lu(A). In this section we identify the features of ISWIM which may need tampering with.

ISWIM has two types of clause, the **where** and the **where-rec**. It would be nice to stick with the latter if we knew that all definitions in a body were to be regarded as recursive, however, the introduction of freezing into Lucid effectively mixes non-recursive definitions with recursive ones, e.g.


```

fac asa x eq n
  where-rec
    n is current n ;
    fac = 1 fby ( fac * next x ) ;
    x = 1 fby ( x + 1 ) ;
  end

```

(Note: Lucid uses **where**, when in Landin's notation it should be **where-rec**. To avoid confusion we stick with Landin's use of the words, and replace Lucid's **where** with **where-rec** in all subsequent Lucid examples.) The freezing declaration **is current** is non-recursive, but unfortunately not a definition in ISWIM(Lu(A)), thus ISWIM(Lu(A)) is a proper subset of Lucid. In this paper we will make freezing an ISWIM definition. The lesson from this example is that we need to mix recursion with non-recursion in the body of a clause. Thus we have two types of definition, the recursive "=", and the non-recursive "is" (to use Landin's notation), e.g.

```

inc( x )    =    x  where    x is x+1 ;  end

```

is a function to increment it's argument by 1. We can of course avoid the use of "is" and **where** in ISWIM by using both renaming of variables and the **where-rec** scope rules, e.g.

```

inc( x )    =    y  where-rec  y = x+1 ;  end

```

and so assume all definitions to be recursive. It is only Lucid's use of freezing which forces us to mix "is" with "=". To avoid confusion with ISWIM we call this new language of recursive and non-recursive definitions ISWIM⁺. It thus has programs such as,

```

e  where+
    x  is  f( x , y ) ;
    y  =  g( x , y ) ;
  end

```

(see below for the where⁺ notation).

The scope rules for definitions in ISWIM⁺ are the same as for ISWIM's two types of definition, except now we mix them. The scope rules for the ISWIM⁺ program,

```

e where defs end

```

are (crudely) summarised by,

1) x is *bound* in e iff it has a definition in the list of definitions *defs*.

2) x is *free* in e iff it is not bound.

- 3) all free variables on the right hand side of an "is" variable definition in a clause C are inherited from outside the innermost clause containing C .

4) all free variables on the right hand side of an "=" variable definition in a clause C take their scope from *defs* if they are defined there, otherwise from the innermost clause containing C .

5) all free variables on the right hand side of an "is" function definition in a clause C are inherited from the formal parameter list, otherwise from the innermost clause containing C .

6) all free variables on the right hand side of an "=" function definition in a clause C are inherited from the formal parameter list if mentioned there, otherwise from *defs* if mentioned there, otherwise from the innermost clause containing C .

Landin's **where** and **where-rec** clause notation makes the assumption that definitions in the accompanying body are either all non-recursive or all recursive respectively. This is unfortunate for ISWIM⁺ as such an assumption cannot be made. The recursion question is settled at the definition level in ISWIM⁺, as opposed to the clausal level in ISWIM. To emphasise this point we do not use either **where** or **where-rec** in ISWIM⁺, but instead introduce the new reserved word **where⁺**.

The treatment of substitution in ISWIM⁺ is as in ISWIM, and is taken directly from the λ -calculus.

$e[x / a]$: in expression e replace all free occurrences of x by a .

The above ISWIM⁺ example, can be described in λ -notation by,

$e[x / f(x, y), y / Y(\lambda y'. g(f(x, y), y'))]$

where Y is assumed to be some suitable fixed point combinator.

The question of extra intensionality above that of $Lu(A)$ thus reduces to an analysis of substitution and recursion (i.e. Y). As the latter may be regarded as a form of repeated substitution, it is fair to

assume that substitution should be the first one to have its intensionality examined.

3. Intensional Substitution

A truly intensional functional programming computational model based upon the ISWIM model must have a logic of programs as well as an algebraic semantics to be complete. It may well be difficult however to build the notion of intensionality into an ISWIM system whose logical foundation rests solely upon the λ -calculus. At the heart of this system are such notions as "abstraction" and "substitution". These are notions which have no concept of intension. Thus in Lucid we cannot abstract over time, and neither can we substitute only the "current" value of a variable. In the case of substitution we must substitute the entire history, e.g. consider the following LUSWIM(A) (i.e. ISWIM(Lu(A))) program.

```

next( x ) + x
  where-rec
    x      =  1 fby 2 ;
  end

```

It is important to realise that such substitutions are not made at each moment in time, as in this example it is not enough to pass the current value of x to the subject as we also need to know its next value as well. It appears to be this restriction which makes freezing difficult to formalise.

Intensional languages such as Lucid have a notion of substitution which is more akin to the operational ideas of lazy evaluation. The demand for x at time t can be expressed in λ -notation as the application x_t . The abstraction of all such demands is expressed by $\lambda t.x_t$. Using the standard η -conversion rule,

$$\lambda x . e(x) \quad \text{cnv}_{\eta} \quad e \quad \text{if } x \text{ is not free in } e$$

we can understand the abstraction of all such demands for x as just x . As Lucid has no explicit time parameters there is no problem in using such η -conversions to model the history of time, that is, until we have to nest times.

The abstraction $\lambda t.x_t$ can be given the following operational interpretation. *At the current moment in time demand the current value of x .* Similarly, $\lambda t.x_{t+1}$ can be read as, *at the current moment in time demand the next value of x .* In other words, the intensional abstraction operator λt can be read as *at the current moment in time*, while the intensional application x_t can be read as a demand for the current value

of x . Using these notations we can give the following intensional description of substitution.

$$e[x | A] = \lambda t . (e[x / \lambda t' . A_t])_t$$

This is logically equivalent to $e[x/A]$, and suggests two nested computations. This scenario of nested computation is, however, misleading. As time (i.e. ω) is infinite, at each moment in the outer computation we demand only a finite number of values from the inner one. In imperative languages this is equivalent to saying that the inner loop must terminate if the outer loop is to progress. The nature of this finite set is determined by the current value of the outer time. One rule for such a determination is Lucid's "when the inner time catches up with the outer time" rule, i.e. freezing. The Lucid clause,

e where-rec x is current A ; end

can be expressed by the λ -expression,

$$\lambda t . (e[x / \lambda t' . A_t])_t$$

We cannot now reduce this expression using η -conversion on the outer abstraction. In general, the current value of the inner computation is dependent upon the current outer time as well as the current inner time. Not only are such times connected (which is difficult enough), but the subject e of the clause must be used for two different computations, one over t , the other over t' . Non-intensional functional languages might overcome such problems by using explicit time parameters. We seek an algebra & logic of temporality for computations which avoids this.

4. Where-f

We suggest now a way of formalising computation rules such as Lucid's freezing rule by redefining substitution in ISWIM. Each form of substitution is, in effect, a computation rule. We consider in this paper only rules in which the outer computation demands a value from the inner computation determined by the two times. Thus we use a function,

$$f : \omega \times \omega \rightarrow \omega$$

A simple non-recursive substitution using this computation rule has the syntax,

e where-f x is A ; end

and with logical form,

$$\lambda t . (e[x / \lambda t' . A_{f(t,t')}])_t$$

Where-f reduces to Landin's **where** if we have,

$$f(t, t') = t'$$

This is because in this case the outer time does not affect the inner time, and so η -conversions can be used. If we have,

$$f(t, t') = t$$

then we have Lucid's freezing rule "when the inner time catches up with the outer time". As freezing is so important in Lucid we give this rule the special **when** name.

Lucid's "from now on" rule given by,

$$\text{remaining}(x)_{tt'} = x_{(t+t')t'}$$

can be described by a **where-f** in which,

$$f(t, t') = t + t'$$

5. Lucid⁺(A)

We now incorporate the above ideas into a new fully functional version of Lucid. Lucid⁺(A) has the following features. Firstly, the basic domain is the stream domain Lu(A). Secondly, we have both "is" and "=" definitions in ISWIM⁺ **where⁺** clauses. Thirdly, each such clause may have a **when** body as well as a **where⁺** body. For example, the factorial Lucid program given in Section 2 could be written in Lucid⁺(ω) as,

```

fac asa x eq n
  when
    n is n;
  and where+
    fac = 1 fby ( fac * next x );
    x   = 1 fby ( x + 1 );
  end

```

Where⁺ substitution is not dependent upon time, and so can be added (as above) to any other single form of intensional substitution such as **when**. Lucid⁺(A) is a fully functional extension of ISWIM(Lu(A)) equivalent to Lucid, but with no declarations, and with a formal semantics in the spirit of Landin, if you see what I mean.

6. Operational Semantics of Intensional Substitution

This will clearly depend upon the form of substitution chosen, however, there is one feature which all variations of $\text{Lucid}^+(A)$ will have in common. Consider the above factorial example, in particular it's subject,

`fac asa x eq n`

Operationally speaking we are demanding values for this expression at points in time. However, intensional substitution requires that we qualify the term "time". At the outer level time is an integer t , while at the inner level time is $f(t,t)$. Thus we not only demand a value at a given moment in time, but demand a value at a given moment in time under a given interpretation of time. This leads us to suggest that each particular form of intensional substitution generates a theory of polymorphic intensional types.

7. LUSWIM and Intensional Substitution

An interesting consequence of intensional substitution is that we can now define some of $\text{Lu}(A)$'s basic temporal operators purely in terms of substitution, e.g.,

`first(x). = $\lambda t . x_0$`

can be defined by,

`first(x) = x start x is x ; end`

where **start** is the **where-f** form of substitution in which,

`f(t , t') = 0`

This is an appealing alternative to the usual Lucid recursive form,

`first(x) = y where-rec y = x fby y ; end`

Similarly, the primitive $\text{Lu}(A)$ operator,

`next(x) = $\lambda t . x_{t+1}$`

can be described by,

`next(x) = x after x is x ; end`

where **after** is the **where-f** in which,

$$f(t, t') = t + 1$$

Note how in these two examples the inner time t' is not mentioned.

It would have been nice to claim that our version of intensional substitution was powerful enough to describe fby, however, this does not seem to be the case. We need an extended "conditional" notion of intensional substitution, e.g. in,

$$(x \text{ fby } y)_t$$

the value of t will determine whether we use values from x or from y .

8. Conclusions

This paper has succeeded in getting the mathematics of Lucid sorted out by "pushing" freezing into ISWIM itself. In doing this we have not had to corrupt the scope rules of ISWIM, and so, all clauses can still be effectively recursive merely by using **where**⁺. The validity of the intensional substitution approach is justified because it accommodates Landin's **where** & **where-rec**, and Lucid's **is current** & **remaining** with equal ease.

Intensional Substitution avoids the introduction of Lucid's "dimension shifting" operations such as,

$$\text{active}(x)_{tt'} = x_t$$

$$\text{contemp}(x)_t = x_{tt}$$

To make semantic sense of this requires the introduction of multidimensional algebras. This seems to be an example of a general problem in ISWIM languages where complex semantic objects are created by the pure necessity of having to conform to ISWIM's extensional rules on substitution. The intensional substitution approach appears to be (at least in the case of Lucid) a way of preventing this. Our only semantic objects in Lucid⁺(A) are streams over A.

9. Further Work

The intensional substitution approach still has a bug just like freezing, it is still very unnatural to program. However, from the programming perspective we have managed to bring one important feature of intensional languages to the fore. They require alternative computation rules such as freezing which ISWIM can only handle by recursion or higher order functions. Our approach allows such rules to be made more explicit by equating each rule with a form of substitution.

This should help, if only a little, with the programmer's intuition. We believe that the syntax of **where-f** and **when** have to be replaced by some "structured" form more suggestive of the nature of intensional substitution, e.g. to describe "when the inner time catches up with the outer time" in imperative language we have constructs of the form,

```
t := 0 ;
repeat
    t' := 0 ;
    while t' < t do t' := t'+1 ;
    t := t+1 ;
forever
```

however, this is only achieved using explicit time parameters.

A second possibility for further work on intensional substitution is to do with modelling real time, i.e. the non-negative real numbers. This is close to the multidimensional algebra approach.

Our main interest at this stage is to investigate the possibility of a theory of polymorphic intensional types, beginning with a type theoretic description of freezing in Lucid.

10. References

- [Ka74] *The Semantics of a Simple Language for Parallel Processing*, G.Kahn, Proc. IFIP Congress 74, pp. 471-475, Elsevier North-Holland, Amsterdam.
- [La64] *The Mechanical Evaluation of Expressions*, P.J.Landin, Computer Journal, Vol. 6, No. 4, pp. 308-320.
- [La66] *The Next 700 Programming Languages*, P.J.Landin, Comm. ACM Vol. 9, pp. 157-166.
- [W&A85] *Lucid, the Dataflow Programming Language*, W.W.Wadge & E.A.Ashcroft, APIC Studies in Data Processing No.22, Academic Press 1985.