LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Deliverables: an approach to program development in the Calculus of Constructions

by

Rod Burstall and James McKinna

LFCS Report Series ECS-LFCS-91-133

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

January 1991

Deliverables: an approach to program development in the Calculus of Constructions[1]

Rod Burstall and James McKinna[2]
Laboratory for the Foundations of Computer Science
University of Edinburgh[3]

January 24, 1991

The problem: separating proofs and programs

There are two contrasting approaches to the formal development of correct programs:

Classical: We can write a program and produce a separate proof of its correctness

Synthetic: We prove the corresponding theorem and extract from it the program (its algorithmic content)

The classical approach seems unsatisfactory in that the proof is separate from the program. It is more acceptable in a formalism such as Floyd-Hoare assertions in which local correctness statements are attached to pieces of program.

The synthetic method is attractive at first sight but there is considerable difficulty in separating the algorithmic part of the proof from the correctness part. No automatic method for doing this is known. See for example Constable *et al* (1986), Nordström, Peterssen and Smith (1990) and Paulin-Mohring (1989).

Extracting programs from proofs

Classically we look for a constructive proof of a statement of the form

$$\forall x.(S(x) \rightarrow \exists y.R(x,y)),$$

in Constructions notation

```
{x:i} (S x -> exists [x:i] R x y).
```

[1]Draft

[2]The authors gratefully acknowledge the support of the EC BRA and the SERC

[3]J.C.M.B., King's Buildings, Mayfield Rd., Edinburgh EH9 3JZ, UK;
e-mail rb@lfcs.ed.ac.uk, jhm@lfcs.ed.ac.uk

But when we do a constructive proof we wind up with a function which, given an individual `x` and a proof of `S x`, produces a pair consisting of a an individual `y` and a proof that it satisfies `R x y`. In other words we cannot get the result `y` without providing the proof of `S x`. It seems that if we try to develop a substantial program this way we need to handle proofs all the time as we attempt to calculate values: the proof and the computation are inextricably mixed.

A better idea is to look for a pair (f, p) where f is a function from individuals to individuals and p is a proof that

$$\forall x.(S(x) \rightarrow R(x, f(x))),$$

in Constructions notation

```
{x:i} (S x -> R x (f x)).
```

Now this is not much good if we have to revert to the classical method of developing f, the program, and p, the proof, independently.

We want to take a more categorical view and build up such program-proof pairs by composition. We will call these pairs "deliverables"; they are what a Software House should deliver to its customers, a program plus a proof in a box with the specification printed on the cover. The customer can independently check the proof and then run the program. The point is that the program part can be trivially extracted from such a pair by taking the first projection and normalising. We have carried this out in Pollack's LEGO implementation of the Calculus of Constructions, see Luo, Pollack and Taylor (1989), Coquand and Huet (1988).

Later we will note that a specification given by an input property and an output property is insufficient, and show how the ideas can be extended to specify a relation between the input and the output.

The Cartesian Closed Category of Deliverables

We will define a cartesian closed category (ccc) of deliverables. That is it will have deliverables as morphisms. For an equational definition of a ccc see Lambek and Scott (1986), Section I.3; we will follow their treatment. The ccc is built up in three stages: first a category, then adding a terminal object and binary products, then adding exponentials. At each stage we will specify the equations which have to be verified. In order to satisfy these equations we will have to assume η-conversion and surjective pairing for products (non-dependent Σ); these are not in the normalisation algorithm for our LEGO implementation of Calculus of Constructions, but it is conjectured that they could be added without loss of consistency.

A category: composition and identity

Objects A type, s, together with a set over s, $S: s \to \mathrm{Prop}$.

Morphisms Morphisms from (s, S) to (t, T), say $\mathrm{del}_{s,t}\ S\ T$ are pairs

- $f: s \to t$
- p, a proof that for all $x: s$, $Sx \supset T(fx)$

$\mathrm{Del}_{s,t}\ S\ T\ f$ will be used to abbreviate

$$\forall x: s. Sx \supset T(fx).$$

Thus $\mathrm{Del}_{s,t}\ S\ T$ is a set over $s \to t$. Define $\mathrm{del}_{s,t}\ S\ T$ as the dependent sum type $\Sigma f: s \to t.\mathrm{Del}_{s,t}\ S\ T\ f$.

In general we will write types as subscripts and feel free to drop them where they can be understood from context. We will use F, G, H to denote morphisms in this category. Sometimes we will write the object (s, S) just as S, where the s may be inferred from the definition of S. In the LEGO code we are able to drop these type parameters, taking advantage of the type inference facility.

We define the composition thus

$$\frac{(s,S) \xrightarrow{(f,p)} (t,T) \quad (t,T) \xrightarrow{(g,q)} (u,U)}{(s,S) \xrightarrow{(f \circ g,\, p * q)} (u,U)}\ \mathrm{comp}_{s,t,u}\ S\ T\ U$$

where $f \circ g$ is the composition and $p * q$ is a proof of $(\mathrm{del}_{s,u}\ S\ U)(f \circ g)$. In fact $(p * q)x = q(fx)(px)$. We will not give such proofs explicitly from now on; they are easily derived in a top down manner in LEGO, and their explicit form is not particularly illuminating.

We define the identity thus

$$\frac{}{(s,S) \xrightarrow{(\mathrm{id}_s,\, p)} (s,S)}\ \mathrm{ident}_s\ S$$

where id_s is the identity function on s, and p is a (trivial) proof that it is in $\mathrm{Del}_{s,s}\ S\ S$. Thus using our convention of omitting types we may say that, given $F: S \to T$ and $G: T \to U$ we have defined comp $F\ G: S \to U$, and we have defined ident $S: S \to S$.

To show that this forms a category, we must check associativity and identity, and we have done so. Although the equation for associativity holds exactly, those for the identity arrows only hold up to η-conversion, so strictly speaking, we do not have a category. We will see later how to deal with this.

Cartesian category: terminal object and product

Our next step is to add the additional structure of a Cartesian category, a terminal object and binary products. The calculus of constructions provided by the LEGO system has binary products for types, `s#t`, but it does not have a terminal type with exactly one element, although ML for example provides such a type and it might reasonably be expected in LEGO. We hope that this minor omission may be repaired some time. There is also a difficulty with the products as follows. The pairing operation is written (x, y) and the projections $p.1$ and $p.2$; we have $(x, y).1$ reduces to x and $(x, y).2$ reduces to y, but not $(p.1, p.2)$ reduces to p. That is, the pairing is not surjective.

We introduce a unit type and a void element of it by declaration, rather than by definition as we would have preferred.

```
unit:Type
void:unit
Unit [x:unit] = Q x void
```

Here "`Q`" means the Leibniz equality, defined by `Q x y` iff any property holding for `x` holds for `y`. We define the unique morphism to the terminal object thus

$$\frac{}{(s,S) \xrightarrow{(u,p)} (\text{unit}, \text{Unit})}\ \text{term}_s\, S$$

where $u\ x = \text{void}$, and p is the (trivial) proof that $\text{Unit}(u\ x)$. We need to show that if $F\colon (s,S) \to (\text{unit}, \text{Unit})$ then $f = (u, p)$. This is only true if we add the extensionality axiom: $f = g$ if for all x, $f\ x = g\ x$, where $=$ refers to Leibniz equality.

We define the product of two objects (s, S) and (t, T) as $(s\sharp t, U)$ where $U\ z = S(z.1) \wedge T(z.2)$. We write $\text{prod}_{s,t}\ S\ T$. The projections are

$$\frac{}{\text{prod}_{s,t}\ S\ T \xrightarrow{(\pi_1, p_1)} S}\ \Pi^1_{s,t}\, S\ T \qquad \frac{}{\text{prod}_{s,t}\ S\ T \xrightarrow{(\pi_2, p_2)} T}\ \Pi^2_{s,t}\, S\ T$$

where $\pi_1\ z = z.1$ and p_1 proves that Del (prod $S\ T$) $S\ \pi_1$. Similarly for π_2.

We define pairing on morphisms

$$\frac{(s,S) \xrightarrow{(f,p)} (t,T) \quad (s,S) \xrightarrow{(g,q)} (u,U)}{(s,S) \xrightarrow{(\langle f,g\rangle, r)} \text{prod}_{t,u}\ T\ U}\ \text{pair}_{s,t,u} S\ T\ U$$

where $\langle f, g\rangle\ x = (f\ x, g\ x)$ and r is the associated proof.

We need to show

(a1) $\pi_1(\text{pair } S\ T) = S.$

(a2) $\pi_2(\text{pair } S\ T) = T.$

(b) $\text{pair } (\pi_1 S, \pi_2 S) = S.$

Equations (a1) and (a2) are easy but (b) does not hold for Leibniz equality (conversion) as we have remarked.

Cartesian closed category: exponentials

To form a cartesian closed category we need to add exponential objects, with two operations apply and curry. (These are respectively the co-unit and the bijection if we define the ccc by an adjunction). The exponential of (t, T) with respect to (s, S), written $\exp_{s,t}\ S\ T$ is $(s \to t, \text{Del}_{s,t}\ S\ T\ )$. We have

$$\frac{}{\text{prod}_{t\to s,t}(\exp_{t,s} T\ S)\ (T) \xrightarrow{(f,p)} (s, S)}\ \text{apply}_{s,t}\ S\ T$$

where $fz = (z.1)(z.2)$ and p is the associated proof.

$$\frac{\text{prod}_{u,t}\ U\ T \xrightarrow{(f,p)} (s, S)}{(u, U) \xrightarrow{(g,q)} \exp_{t,s} T\ S}\ \text{curry}_{s,t,u}\ S\ T\ U$$

where $g\ x\ y = f(x, y)$ and q is the associated proof.

These must satisfy

$$\text{apply}(\text{pair}(\text{comp}\ \pi_1(\text{curry}\ F))\ \pi_2) = F \quad \text{for all } F\text{: prod } S\ T \to U$$

$$\text{curry}(\text{apply}(\text{pair}(\text{comp}\ \pi_1\ G)\ \pi_2)) = G \quad \text{for all } G\text{: } U \to \exp\ T\ S$$

The necessary equations fail to hold for the same reasons as before (failure of η-conversion and surjective pairing). However we will remedy this below.

Natural number object

We now add a natural number object (indeed we will have to add an object for each inductively defined data type). This object comes with a family of morphisms embodying the induction principle for natural numbers.

```
nat:Type
zero:nat
succ:nat -> nat
Nat:nat -> Prop
Nat [n:nat] = true
```

As usual, we write "0" for "zero", and "+1" (postfix) for "succ". The natural number object is (nat, Nat).

We can define a function natrec: $t \to (t \to t) \to (\text{nat} \to t)$ polymorphic for any type t with axioms

$$\text{natrec } z\ s\ 0 = z$$

$$\text{natrec } z\ s\ (n+1) = s(\text{natrec } z\ s\ n)$$

$$\frac{(\text{unit}, \text{Unit}) \xrightarrow{(z,p)} (t,T) \quad (t,T) \xrightarrow{(s,q)} (t,T)}{(\text{nat}, \text{Nat}) \xrightarrow{(\text{natrec } z\ s\ r)} (t,T)}\ \text{Natrec}_t\ T$$

where r is the appropriate proof. In fact r uses the induction principle for nat to show that if z takes you into t and s preserves t then for any n, natrec z s n is in T.

Example: the function double

We need an example to see all this at work. We will define properties Odd and Even over the natural numbers and a function double; we then show that, for any natural n, double n is even.

```
Even:nat -> Prop
Odd [n:nat] = not (Even n)
```

These will have the axioms (for all n)

$$\text{Even } 0$$

$$\text{Even } n \supset \text{Odd } (n+1)$$

$$\text{Odd } n \supset \text{Even } (n+1)$$

with proofs n_0, n_1 and n_2 respectively. We define the deliverables

$$\frac{}{(\text{unit}, \text{Unit}) \xrightarrow{(\lambda u:\text{unit}.0,\, n_0)} (\text{nat}, \text{Even})}\ \text{zE}$$

$$\frac{}{(\text{nat}, \text{Even}) \xrightarrow{(\lambda n{:}\,\text{nat}.n+1, n_1)} (\text{nat}, \text{Odd})}\ \text{sEO}$$

$$\frac{}{(\text{nat}, \text{Odd}) \xrightarrow{(\lambda n{:}\,\text{nat}.n+1, n_2)} (\text{nat}, \text{Even})}\ \text{sOE}$$

In terms of these we define a deliverable for successor of successor

$$\text{ssEE} = \text{comp sEO sOE}$$

and a deliverable for double,

$$\text{doubleNE} = \text{Natrec Even zE ssEE}$$

. Now the typechecker can determine that doubleNE: $(\text{nat}, \text{Even}) \to (\text{nat}, \text{Even})$ as required.

Top down development

In LEGO one normally develops proofs by refinement, starting from the goal, selecting an inference rule to refine by and so producing subgoals. Although we have used the bottom-up approach in the last section, it is also possible to use the top-down refinement approach. We start with the goal

$(\text{nat}, \text{Even}) \to (\text{nat}, \text{Even})$

Then refine by the Natrec rule with the invariant (nat, Even) as parameter, getting as subgoals

$(\text{unit}, \text{Unit}) \to (\text{nat}, \text{Even})$

$(\text{nat}, \text{Even}) \to (\text{nat}, \text{Even})$

The first of these is solved by refining by zE. The second is refined by comp Even Odd Even giving as subgoals

$(\text{nat}, \text{Even}) \to (\text{nat}, \text{Odd})$

$(\text{nat}, \text{Odd}) \to (\text{nat}, \text{Even})$

These are solved by sEO and sOE respectively. Instead of refining by a deliverable we could just refine by an incomplete deliverable, leaving the proof part out but giving the function part. The proof part would appear as a pending subgoal. The LEGO system is quite flexible in the way it allows one to develop a proof, and this flexibility is also available for program development. It seems much better than being restricted to a rigid sequence of decisions, whether top down or bottom up.

Equivalence for deliverables

The previous account reveals a number of defects in the structure of our putative category caused by the lack of η-conversion and surjective pairing in LEGO. However, by explicitly considering types and predicates together with a notion of equality, we may define product, exponential etc., in such a way as to overcome these defects. So we really do have a category, and it is indeed cartesian closed.

To do this, we redefine an object to be a triple (s, S, p), consisting of

s, a type

S, a relation on s

p, a proof that S is a partial equivalence relation (i.e. symmetric and transitive, but not necessarily reflexive)

(This is the same as considering an equivalence relation $=$ and a distingushed property P: given such, we may define a per P^*xy iff $Px \wedge x = y \wedge Py$; conversely, given a per S we may define a property $S^\dagger x$ iff Sxx, with the equivalence relation $x = y$ iff $xSy \vee xQy$, where Q is the Leibniz equality, as usual.)

We then say an arrow from (s, S, p) to (t, T, q) is a pair (f, r), with $f : s \to t$ and r a proof of $f \equiv f$, where $\equiv$ is the partial equivalence relation defined by

$$f \equiv g \text{ iff } \forall x, y : s.xSy \Rightarrow (fx)T(fy).$$

Second order deliverables

The system which we have described above amounts to a functional version of the well known invariants used in proofs of imperative programs. Unfortunately there is no connection between the input and the output of the function. All we say is that if the input is in set S then the output is in set T, but there is no *relation* between them. For example we might specify that a sorting function takes lists to ordered lists, but we cannot specify that the output is a permutation of the input. The function might always produce the empty list, which is indeed sorted, but not very interesting. As a matter of fact the classical invariant proofs have the same weakness, masked by a tacit assumption that some variable which is carried through the computation does not change its value. To enforce the constraint that the output list be a permutation of the input list we need to resort to "second order" deliverables, where the deliverables defined above are "first order" ones.

So let us write Del^1 for Del above and del^1 for del. We now define Del^2 and del^2. Suppose $S\colon s \to \text{Prop}$, $P\colon s \to p \to \text{Prop}$ and $R\colon s \to r \to \text{Prop}$ for types s, p and r. Let

$$\text{Del}^2_{s,p,r}\ S\ P\ R\ f = \forall x.Sx \to (\forall y.P\ x\ y \to R\ x\ (f\ x\ y))$$

that is $\forall x.Sx \to \mathrm{Del}^1(P\,x)(R\,x)(f\,x)$. $\mathrm{del}^2_{s,p,r}\ S\ P\ R$ is the dependent sum type $\Sigma f{:}p \to r.\mathrm{Del}^2_{s,p,r}\ S\ P\ R\ f$. There is a category $\mathbf{C}_sS$ whose objects are (p,P), (r,R), ... for relations P, R, ... and whose morphisms from (p,P) to (r,R) are elements of $\mathrm{del}^2_{s,p,r}\ S\ P\ R$.

If $P_1{:}s \to p_1 \to \mathrm{Prop}$ and $P_2{:}s \to p_2 \to \mathrm{Prop}$, then they have a product $P_{12}{:}s \to (p_1 \sharp p_2) \to \mathrm{Prop}$ given by $P_{12}\ x\ y_{12} = S\ x \to (P_1\ x\ (\pi_1\ y_{12}) \wedge P_2\ x\ (\pi_2\ y_{12}))$, with projections $\Pi{:}P_{12} \to P_i$ given by $\Pi\ x\ y_{12} = (\pi_i\ y_{12}, \ldots)$ where ... is the (trivial) proof that for all $x{:}s$, $Sx \to (\forall y_{12}{:}p_1 \sharp p_2, P_{12}\ x\ y_{12} \to P_i\ x\ (\pi_i\ y_{12}))$.

We conjecture that $\mathbf{C}_sS$ is cartesian closed (modulo the equivalences mentioned earlier) although we haven't checked the details.

Given $F{:}\mathrm{del}^1\ T\ S$ and $G{:}\mathrm{del}^2\ S\ P\ R$ we can define objects (F^*P), (F^*R) and substitution $F^*G{:}\mathrm{del}^2\ T\ (F^*P)\ (F^*R)$. We conjecture that this substitution operation defines an indexing of del^2 over del^1, similar indeed to the general categorical framework for dependent types, based on our experience with a set-theoretic treatment of deliverables, in which we obtained a model of the Calculus of Constructions from a topos(a proof-irrelevant model of higher-order logic). This suggests a reflection in which a dependent product type may be used to name a hom-set of deliverables (compare Martin-Löf's "subset interpretation" of type theory). We are currently investigating all this.

One could proceed in analogous manner to third order deliverables and so on, but it is not clear how useful these are. Since we have Σ-types we can presumably code these as second order deliverables (thanks to Gordon Plotkin for this remark).

As an example, we have been experimenting with these second order deliverables for the proof of an insert sort. First we proved an induction principle over sorted lists, which enables us to establish a relation between an input list and the result of applying a recursive function to it:

$$\text{for } \phi{:}(\mathrm{list}\ \alpha) \to \beta \to \mathrm{Prop},\quad n{:}\beta,\quad c{:}\alpha \to (\mathrm{list}\ \alpha) \to \beta \to \beta$$

$$\frac{\begin{array}{c}\left[\mathrm{Sorted}\ (a::l),\ \ \phi\ l\ b\ \ [a{:}\alpha,\ l{:}\mathrm{list}\ \alpha,\ b{:}\beta]\right] \\ \vdots \\ \phi\ \mathrm{nil}\ n \qquad\qquad \phi\ (a::l)\ (c\ a\ l\ b)\end{array}}{\forall l{:}\mathrm{list}\ \alpha.\ \ \mathrm{Sorted}\ l \supset \phi\ l\ (\mathrm{listrec}\ n\ c\ l)}\ \text{Sorted List Induction}$$

From this principle we obtain the following operation on deliverables, which involves a dependent family, F, of second order deliverables:

$$\frac{N \in \mathrm{del}^1\ \mathrm{Unit}\ \mathrm{nil}^*\phi \quad F \in \Pi a{:}\alpha.\mathrm{del}^2\ ((\mathrm{cons}\ a)^*\ \mathrm{Sorted})\ \ \phi\ \ (\mathrm{cons}\ a)^*\phi}{(\mathrm{listrec}\ n\ c, \ldots) \in \mathrm{del}^2\ \mathrm{Sorted}\ \mathrm{True}\ \phi}$$

where $n = N.1$ and $c = \lambda a{:}\alpha.((F\,a).1)$. In the proof of insert sort, we have applied this construction to the invariant $\phi = \lambda l, m{:}\mathrm{list}\ \alpha.\mathrm{Sorted}\ m \wedge \mathrm{Perm}\ l\ m$. This enables us to prove that the output of sort is both sorted and a permutation of the input.

References

[Ben85] Bénabou, J. *Fibred categories and the foundations of näive category theory*, JSL, 1985.

[Bur89] Burstall, R.M. *An approach to Program Specification and Development in Constructions*, Talk given at the Workshop on Programming Logic, Båstad, Sweden, May 1989.

[BM90] Burstall, R.M. and McKinna, J.H. *A subset model of the theory of Constructions*, Edinburgh, 1990, in preparation.

[Gir86] Girard, J-Y. *Linear Logic*, in Theoretical Computer Science, Vol 50, North-Holland, The Netherlands, 1986.

[Gol84] Goldblatt, R. *Topoi: the categorical analysis of logic*, Studies in Logic and the Foundations Of Mathematics no. 98, revised 2nd edition, North-Holland, The Netherlands, 1984.

[Hyl82] Hyland, J.M.E. *The Effective Topos*, in: Proceedings of the Brouwer Centenary Symposium, North-Holland, The Netherlands, 1982.

[Hyl87] Hyland, J.M.E. *A small complete category*, in: Proceedings of the Conference on Church's Thesis: Fifty Years Later, 1987.

[HP86] Hyland, J.M.E., and Pitts, A.M. *Catgeorical Semantics for the theory of Constructions: Topos-theoretic models*, in: Proceedings of the AMS Conference on Categories in Compuetr Science, Boulder, Colorado, 1986

[Joh77] Johnstone, P.T. *Topos Theory*, Academic Press, London, 1977

[KW71] Kock, A. and Wraith, G. *Elementary Toposes*, Aarhus Lecture Notes no. 30, Aarhus Universitet, Denmark 1971.

[LS86] Lambek, J. and Scott, P.J. *An Introduction to Higher-Order Categorical Logic*, Cambridge Studies in Advanced Mathematics no. 7, Cambridge University Press, Cambridge, England, 1986.

[Luo90a] Luo, Z. *An Extended Calculus of Constructions*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh, June 1990.

[LPT89] Luo, Pollack, and Taylor, *How To Use LEGO (A Preliminary User's Manual)* LFCS Technical Note LFCS-TN-27, October 1989.

[Mog90] Moggi, E. *A category-theoretic account of program modules*, Draft, Edinburgh, August 1990.

[NPS90] Nordström, B., Peterssen, K. and Smith, J. *Programming in Martin-Löf's type theory*, Oxford University Press, 1990.

[Pit89] Pitts, A.M. *Categorical Semantics of Dependent Types*, Talk given at SRI International, Menlo Park, California, June 1989.

[PMW90] Paulin-Mohring, C. and Werner, B. *Extracting and Executing Programs developed in the Inductive Constructions System: a Progress Report*, in: Proceedings of the First Annual Workshop on Logical Frameworks, Antibes, May 1990, eds. Huet, G. and Plotkin, G.