McCarthy-Painter Induction

James McKinna

Computer-Assisted Reasoning Group

Dept. Computer Science, University of Durham

J. H. McKinna@durham.ac.uk

STP seminar, St. Andrews, 10 July 2003.

Outline of talk

1. Some problems with function definition in a verification environment
2. Some example functions; some example properties
   - addition; associativity of addition
   - gcd; correctness of gcd
   - Burstall's (1969) induction principle for `lit`
   - McCarthy-Painter (1967) expression compiler; correctness
3. A proof principle for defined functions: "McCarthy-Painter induction"
4. EPIGRAM as a notation for function definition in type theory
5. 'Inductive' *vs.* 'Productive' specifications
6. Related work
7. Further work and Conclusions

Function definition in a verification environment

A simple minded taxonomy of the problems associated with definition of a function f in a formal logic:

- how to specify f;
- how to prove f exists/is well-defined/definition is legitimate;
- how to prove properties of f once you have it.

In this talk, I'll focus on the last point, bringing in the discussion of the first two where relevant or interesting.

The oldest problem in the literature McCarthy, Landin, Floyd, Hoare, Burstall, Scott, Milner, Weyrauch, Aubin, Boyer& Moore, Bundy, Morris, ...

Some example functions; some example properties

1. The addition function, $+$, specified by:
$$\begin{aligned} 0 + y &= y \\ \mathsf{s}x + y &= \mathsf{s}(x+y) \end{aligned}$$
satisfies

   $\forall x, y, z.\ (x+y)+z = x+(y+z)$

2. The greatest common divisor, $\mathbf{gcd}$, specified by:

$$\begin{aligned}
\mathbf{gcd}\ \ 0\ \ \mathsf{s}n &= \mathsf{s}n \\
\mathbf{gcd}\ \ \mathsf{s}m\ \ 0 &= \mathsf{s}m \\
\mathbf{gcd}\ \ n\ \ n &= n \\
\mathbf{gcd}\ (m+n)\ \ n &= \mathbf{gcd}\ m\ n \\
\mathbf{gcd}\ \ m\ \ (n+m) &= \mathbf{gcd}\ m\ n
\end{aligned}$$

   satisfies:

   $\forall m, n, z.(z = \mathbf{gcd}\ m\ n) \supset (z|m) \wedge (z|n) \wedge (\exists a, b.z = am+bn) \wedge \cdots$

3. (Burstall, 1969) The higher-order function $\texttt{lit}$ (ISWIM; Landin 1965) defined on lists by:

$$\begin{aligned} \texttt{lit}\, f\ a\quad (\texttt{nil}) &= a \\ \texttt{lit}\, f\ a\, (\texttt{cons}\ h\ t) &= f\ h\ (\texttt{lit}\, f\ a\ t) \end{aligned}$$

satisfies the following induction principle:

$$\forall \Phi.\, (\Phi\ \texttt{nil}\ a) \supset (\forall h, t, y.\, (\Phi\ t\ y) \supset \Phi\ (\texttt{cons}\ h\ t)\ (f\ h\ y)) \supset (\forall l.\, \Phi\ l\ (\texttt{lit}\, f\ a\ l))$$

4. (McCarthy-Painter, 1967) For a simple language of expressions, Expr, evaluation leaves the same result on top of the stack as executing the compiled instructions on that stack:

$$\forall e : \mathsf{Expr}. \forall s : \mathsf{Stk}. (\mathbf{eval}\ e) :: s = (\mathbf{exec}\ (\mathbf{compile}\ e)\ s)$$

An Induction Principle for $+$

If $\Phi \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ satisfies

- $\forall y.\, \Phi\ \mathsf{0}\ y\ y$ and
- $\forall x, y, z.\, \Phi\ x\ y\ z \supset \Phi\ (\mathsf{s}x)\ y\ (\mathsf{s}z)$

then

- $\forall x, y, z.\, z = x + y \supset \Phi\ x\ y\ z$
- *i.e.* $\forall x, y.\, \Phi\ x\ y\ (x + y)$

Proof Induction on x, following the recursion pattern in the definition/specification of $+$

Associativity of $+$

Two ways to establish $\forall x, y, z. x + (y + z) = (x + y) + z$

direct induction on x: • base case immediate by computation;

- step case: ripple the s wavefront (twice on each of LHS and RHS), then fertilize with IH;

eliminate innermost nested occurrence of $+$: prove, in effect,

$$\forall w, x, y. (w = x + y) \supset \forall z. x + (y + z) = (w) + z$$

using the induction principle for $+$:

- base case immediate by computation;
- in the step case, on the RHS the wavefront is *one level further out*, ripple on the LHS, then fertilize with IH.

Deriving the induction principle

The induction principle which we used for $+$ is **exactly** that for the inductively defined relation $P(w, x, y)$ which axiomatises the three-place relation $w = x + y$, *i.e.* the *inductively defined graph* of the function $+$:

$$\underline{\text{data}} \quad \frac{x, y, w \ : \ \mathbb{N}}{P\ x\ y\ w \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{P_{\mathsf{0}} y \ : \ P\ \mathsf{0}\ y\ y} \qquad \frac{p \ : \ P\ x\ y\ w}{P_{\mathsf{S}}\ p \ : \ P\ (\mathsf{s}x)\ y\ (\mathsf{s}w)}$$

We remark:

- The two-place function gives rise to a three-place relation (!);
- Each defining equation in the specification gives rise to a constructor (axiom, resp. inference rule);
- Each recursive call gives rise to an inductive premise.

An induction principle for **gcd**

Suppose $\Phi \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ satisfies

- $\forall m.\ \Phi\ (\mathsf{s}m)\ 0\ (\mathsf{s}m)$ and
- $\forall n.\ \Phi\ 0\ (\mathsf{s}n)\ (\mathsf{s}n)$ and
- $\forall x.\ \Phi\ x\ x\ x$ and
- $\forall m, n, d.\ \Phi\ (\mathsf{s}m)\ (\mathsf{s}n)\ d \supset \Phi\ (\mathsf{s}n{+}\mathsf{s}m)\ (\mathsf{s}n)\ d$
- $\forall m, n, d.\ \Phi\ (\mathsf{s}m)\ (\mathsf{s}n)\ d \supset \Phi\ (\mathsf{s}m)\ (\mathsf{s}m{+}\mathsf{s}n)\ d$

then

- $\forall x, y.\ \Phi\ x\ y\ (\mathbf{gcd}\ x\ y)$

Proof from the definition of the function . . . ???

Other examples and properties

- properties of **gcd** follow directly from the induction principle;
- proof of Burstall's `lit`-induction follows by simple list induction on l;
- statement and proof of induction principle for **compile** and/or **eval** omitted; syntax-directedness of compilation in fact means that structural induction on Expr suffices, at the cost of further analysing the **exec** function.

The general picture

To prove some property of (the values of) a recursive function f defined on inductively defined data D,

$$? \ : \ \Phi(f(e)) \ \ (\text{where } \ldots \vdash e \ : \ D)$$

- Instead of (structural) induction on *data* D (Burstall, 1969), do ...
  - "induction on the recursive calls of the function", "recursion-induction" (McCarthy & Painter: proofs of correctness of an expression compiler, 1967)
  - *cf.* "subgoal induction" (cf. Morris and Wegbreit, CACM, 1977)
  - *cf.* "the method of intermediate assertions" (Floyd: flowcharts, 1967)
- Induction involved is *rule* induction for $\mathtt{f}$-Ind, the *inductively defined graph* of the function f.

Defining $\mathbf{f}$-Ind: the case of EPIGRAM programs

EPIGRAM (McKinna & McBride, *The view from the left*, JFP, to appear 2003) consists, in essence, of syntax for declaring:

- inductive families of *dependently-typed data*:

$$\underline{\text{data}}\ \frac{\Phi}{\mathsf{D}\ \Phi\ :\ \star}\ \underline{\text{where}}\ \frac{\Phi_1}{\mathsf{c}_1\ \Phi_1\ :\ \mathsf{D}\ \vec{e}_1}\ \cdots\ \frac{\Phi_n}{\mathsf{c}_n\ \Phi_n\ :\ \mathsf{D}\ \vec{e}_n}$$

- defined function symbols $\mathbf{f}$ by writing *programs* p: $\underline{\text{let}}\ \frac{\Phi}{\mathbf{f}\ \Phi\ :\ R}\ p$

- where programs are written using
  - *user-defined* elimination principles, generalizing the use of D**-elim** from type theory;
  - (pattern matching on) subsidiary computations, akin to Haskell's pattern guards;
  - 'return's, *LHS* $\mapsto$ *RHS*, like functional equations

Example: $\mathbf{gcd}$ in EPIGRAM in its full glory

$$
\begin{array}{llll}
\mathbf{gcd} & m & n & \Leftarrow \underline{\textsf{view}}\ \mathbf{plusrec}\ m \\
\mathbf{gcd} & m & n & \Leftarrow \underline{\textsf{view}}\ \mathbf{plusrec}\ n \\
\mathbf{gcd} & m & n & \Leftarrow \underline{\textsf{view}}\ \mathbf{compare}\ m\ n \\
\quad \mathbf{gcd} & x & (x + \mathsf{s}y) & \Leftarrow \mathbb{N}\mathbf{-case}\ x \\
\qquad \mathbf{gcd} & \mathsf{0} & (\mathsf{s}y) & \mapsto \mathsf{s}y \\
\qquad \mathbf{gcd} & (\mathsf{s}x) & (\mathsf{s}x + \mathsf{s}y) & \mapsto \mathbf{gcd}\ (\mathsf{s}x)\ (\mathsf{s}y) \\
\quad \mathbf{gcd} & x & x & \mapsto x \\
\quad \mathbf{gcd} & (y + \mathsf{s}x) & y & \Leftarrow \mathbb{N}\mathbf{-case}\ y \\
\qquad \mathbf{gcd} & (\mathsf{s}x) & \mathsf{0} & \mapsto \mathsf{s}x \\
\qquad \mathbf{gcd} & (\mathsf{s}y + \mathsf{s}x) & (\mathsf{s}y) & \mapsto \mathbf{gcd}\ (\mathsf{s}x)\ (\mathsf{s}y)
\end{array}
$$

Remark five lines of ‘program’; five lines of ‘evidence’

Stripped-down **gcd** in EPIGRAM

$$
\begin{array}{llll}
\mathbf{gcd} & m & n & \Leftarrow \underline{\text{view}}\ \mathbf{compare}\ m\ n \\
\quad \mathbf{gcd} & 0 & (\mathsf{s}y) & \mapsto \mathsf{s}y \\
\quad \mathbf{gcd} & (\mathsf{s}x) & (\mathsf{s}x + \mathsf{s}y) & \mapsto \mathbf{gcd}\ (\mathsf{s}x)\ (\mathsf{s}y) \\
\ \ \mathbf{gcd} & x & x & \mapsto x \\
\quad \mathbf{gcd} & (\mathsf{s}x) & 0 & \mapsto \mathsf{s}x \\
\quad \mathbf{gcd} & (\mathsf{s}y + \mathsf{s}x) & (\mathsf{s}y) & \mapsto \mathbf{gcd}\ (\mathsf{s}x)\ (\mathsf{s}y)
\end{array}
$$

Remark removed ‘standard’ case analysis and recursion principles from definition; informative case analysis (trichotomy on $<_{\mathbb{N}}$) left in.

The graph $\mathbf{f}$-Ind *is* the graph of f, mostly

Given a defined function symbol $\mathbf{f}$, taking arguments $\vec{x}$, we introduce an inductively defined relation $\mathbf{f}$-Ind taking $\vec{x}, y$ whose *constructors* correspond precisely to the 'return' clauses in the definition of $\mathbf{f}$. Recursive calls on $\mathbf{f}\ \vec{a}$ yield inductive premises of the form $\mathbf{f}$-Ind $\vec{a}\ b$. McCarthy-Painter induction is then exactly the induction principle associated with $\mathbf{f}$-Ind.

We always have the following, by mimicking the definition of f:

$$\forall \vec{x}.\ \forall y.\ f(\vec{x}) = y \supset \mathbf{f}\text{-Ind}(\vec{x}, y)$$

Provided that the EPIGRAM definition is *unambiguous*, we may further prove that

$$\forall \vec{x}.\ \forall y.\ \mathbf{f}\text{-Ind}(\vec{x}, y) \supset f(\vec{x}) = y$$

and indeed that $\mathbf{f}$-Ind is *functional*:

$$\forall \vec{x}.\ \forall y, z.\ \mathbf{f}\text{-Ind}(\vec{x}, y) \supset \mathbf{f}\text{-Ind}(\vec{x}, z) \supset z = y$$

Advantages

- The “separation of concerns”: $\mathbf{f}$-Ind-induction is the ‘partial correctness’ assertion for f;
- Proving $\forall \vec{x}. \exists y. \mathbf{f}\text{-Ind}(\vec{x}, y)$ is the termination proof, ‘total correctness’ for f;
- Proving $\forall \vec{x}. \exists y. \mathbf{f}\text{-Ind}(\vec{x}, y)$ is *automatic*, given f; we look at the case of fs defined in the EPIGRAM notation;
- Uncouple from the computational behaviour of f, in favour of (a/the) *specification* of f;
- Works for non-deterministic specifications;
- Works for nested recursion, without tears; nested calls become chains of inductive premises;
- Works for McCarthy’s “91”-function, with some tears;

- Works for higher-order functions, with some tears; extensionality of the output function can become an issue;
- . . .

Disadvantages

- Uncoupled from the computational behaviour of f, in favour of (a/the) *specification* of f;
- Induction principle often no better than structural induction;
- Relies on having the result place in constructor form; syntax-directed rules
- ...

‘Inductive’ *vs.* ’Productive’ definitions

To show that $f \subseteq \Phi$, we go via the **inductive** McCarthy-Painter relation $\mathbf{f}$-Ind,

...but we might also require a **productive** relation, $\mathbf{f}$-Prod, which expresses the output(s) of f in *constructor form*, suitable for establishing Φ:

$$f \subseteq \mathbf{f}\text{-Ind} \subseteq \mathbf{f}\text{-Prod} \subseteq \Phi$$

We remark that

- *intensionally*, $\mathbf{f}$-Ind is a *least* fixed point (for its constructors), while $\mathbf{f}$-Prod is merely some fixed point, so $\mathbf{f}\text{-Ind} \subseteq \mathbf{f}\text{-Prod}$
- *extensionally*, we usually have that $\mathbf{f}\text{-Prod} \subseteq \mathbf{f}\text{-Ind}$, but as sets of derivation trees, they need not be isomorphic, *cf.* first-order *vs.* higher-order abstract syntax for binding (McKinna& Pollack 1993, 1995, 1999)
- the proof of $\mathbf{f}\text{-Prod} \subseteq \Phi$ is also by induction; show that Φ is a fixed point for the constructors of $\mathbf{f}$-Prod

Comparing Inductive and Productive reasoning about $=_{\mathbb{N}}$

Given the EPIGRAM definition of Boolean-valued equality on $\mathbb{N}$,

$$\underline{\text{let}} \quad \frac{m, n \; : \; \mathbb{N}}{\mathbb{N}\mathbf{eq}\ m\ n \; : \; \mathsf{Bool}} \qquad \begin{array}{lcl} \mathbb{N}\mathbf{eq}\ \mathsf{0}\ \mathsf{0} & \mapsto & \mathsf{true} \\ \mathbb{N}\mathbf{eq}\ \mathsf{0}\ \mathsf{s}n & \mapsto & \mathsf{false} \\ \mathbb{N}\mathbf{eq}\ \mathsf{s}m\ \mathsf{0} & \mapsto & \mathsf{false} \\ \mathbb{N}\mathbf{eq}\ \mathsf{s}m\ \mathsf{s}n & \mapsto & \mathbb{N}\mathbf{eq}\ m\ n \end{array}$$

we obtain the following (inductive) McCarthy-Painter definition:

$$\frac{}{\mathbb{N}\mathbf{eq}\text{-}\mathsf{Ind}\ \mathsf{0}\ \mathsf{0}\ \mathsf{true}} \qquad \frac{}{\mathbb{N}\mathbf{eq}\text{-}\mathsf{Ind}\ \mathsf{0}\ \mathsf{s}n\ \mathsf{false}} \qquad \frac{}{\mathbb{N}\mathbf{eq}\text{-}\mathsf{Ind}\ \mathsf{s}m\ \mathsf{0}\ \mathsf{false}}$$

$$\frac{\mathbb{N}\mathbf{eq}\text{-}\mathsf{Ind}\ m\ n\ b}{\mathbb{N}\mathbf{eq}\text{-}\mathsf{Ind}\ \mathsf{s}m\ \mathsf{s}n\ b}$$

Note, however, that we do not obtain the result in constructor form in the interesting (recursive) case, because of the tail call.

So to prove anything about this function, we need the following productive definition:

$$\frac{}{\mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ n\ n\ \mathsf{true}} \qquad \frac{m \neq n}{\mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ m\ n\ \mathsf{false}}$$

The proof that $\mathbb{N}\mathbf{eq}\text{-}\mathsf{Ind} \subseteq \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}$ looks roughly like this:

$? \ : \ \forall m, n.\ \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ m\ n\ (\mathbb{N}\mathbf{eq}\ m\ n)$

$\Leftarrow$ <u>induction</u> $\mathbb{N}\mathbf{eq}\text{-}\mathbf{ind}\ m\ n$

$? \ : \ \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ 0\ 0\ \mathsf{true}$

$? \ : \ \forall n.\ \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ 0\ \mathsf{s}n\ \mathsf{false}$

$? \ : \ \forall m.\ \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ \mathsf{s}m\ 0\ \mathsf{false}$

} *trivial*

$? \ : \ \forall m, n, b.\ \ \forall H : \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ m\ n\ b.\ \ \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ \mathsf{s}m\ \mathsf{s}n\ b$

$\Leftarrow$ <u>case</u> H

$? \ : \ \forall n.\ \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ \mathsf{s}n\ \mathsf{s}n\ \mathsf{true}$

$? \ : \ \forall m, n.\ m \neq n\ \rightarrow\ \mathbb{N}\mathbf{eq}\text{-}\mathsf{Prod}\ \mathsf{s}m\ \mathsf{s}n\ \mathsf{false}$

} *trivial*

Remark on recursion strategies

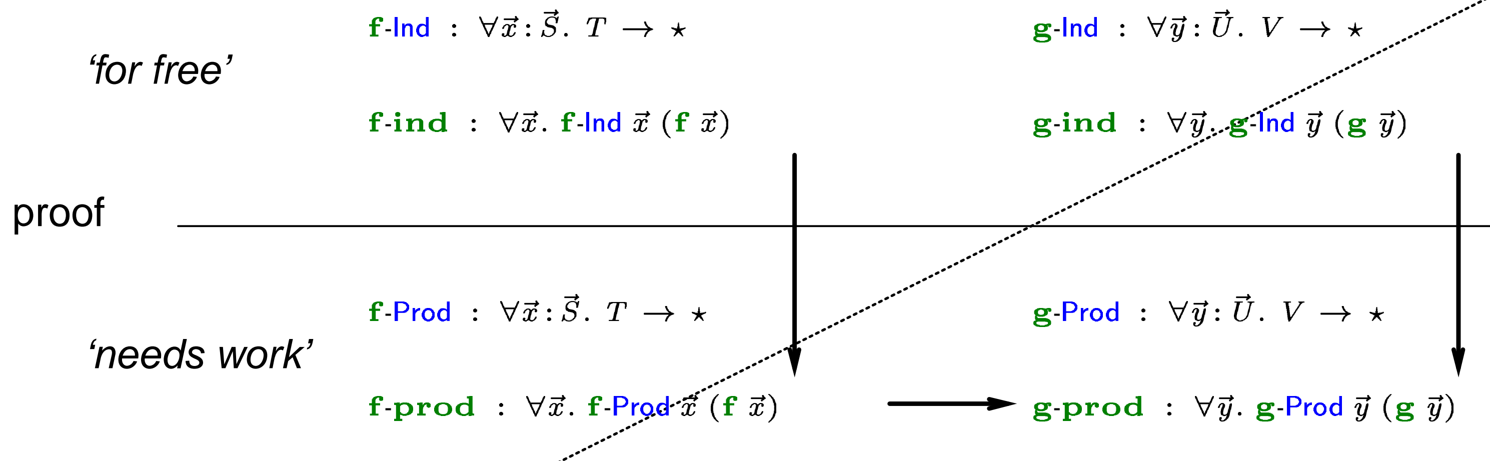
Informally (Boyer-Moore 1979), we know it's a good idea to use primitive recursive definitions for reasoning but bad for computation, while their extensionally equivalent tail-recursive formulations are good for computation, but bad for reasoning (need to generalize with accumulating parameters, . . .).

Remark here that it's not so clear cut: **gcd** makes tail calls, yet **gcd**-Ind is suficient to prove almost everything (once divisibility is closed under addition), whereas $\mathbb{N}$**eq**-Ind isn't good enough because of the tail call.

We don't have a formal definition of 'productive', merely observing that we should choose a suitable constructor-form analysis to allow the proof to progress by unfolding the property or some subcomputation. . . so we should try and relate this more precisely to so-called *flawed* inductions of Boyer-Moore and friends. . .

A compositional approach

When $\mathbf{g}$ is defined in terms of $\mathbf{f}$, we can (try to!) build up a compositional analysis of $\mathbf{g}$, via a productive relation $\mathbf{g}$-Prod: we prove $\mathbf{f}$-Ind $\subseteq$ $\mathbf{f}$-Prod$(\dagger)$ as usual, and prove $\mathbf{g}$-Ind $\subseteq$ $\mathbf{g}$-Prod$(\ddagger)$ by induction, using $(\dagger)$ to analyse calls to f:



Example: a *structural* first-order unification algorithm (McBride, 2000–2003)

$$\frac{x, y \;:\; \mathsf{Var}\ \mathsf{s}m}{\mathbf{thinnedVar}\ x\ y \;:\; \mathsf{Maybe}\ (\mathsf{Var}\ m)}$$

[recursion x, case y]

$$\frac{}{\mathbf{thinnedVar}\text{-}\mathsf{Prod}\ x\ (\mathbf{thin}\ x\ y')\ (\mathsf{yes}\ y')}$$

$$\frac{}{\mathbf{thinnedVar}\text{-}\mathsf{Prod}\ x\ x\ \mathsf{no}}$$

$$\frac{x \;:\; \mathsf{Var}\ \mathsf{s}m \qquad t \;:\; \mathsf{Term}\ \mathsf{s}m}{\mathbf{thinnedTerm}\ x\ y \;:\; \mathsf{Maybe}\ (\mathsf{Term}\ m)}$$

[recursion t]

$$\frac{}{\mathbf{thinnedTerm}\text{-}\mathsf{Prod}\ x\ (\mathbf{rename}\ (\mathbf{thin}\ x)\ t')\ (\mathsf{yes}\ t')}$$

$$\frac{}{\mathbf{thinnedTerm}\text{-}\mathsf{Prod}\ x\ (\mathbf{put}\ (\mathsf{var}\ x)\ pos)\ \mathsf{no}}$$

$\vdots$ [**flexFlex**, **flexRigid**, **mguAccum**] $\vdots$

$$\frac{s, t \;:\; \mathsf{Term}\ m}{\mathbf{mgu}\ s\ t \;:\; \mathsf{Maybe}\ \exists n.\mathsf{Subst}\ m\ n}$$

[recursion n, recursion s, case t]

$$\frac{\mathbf{Max}\ (\mathbf{Unifies}\ s\ t)\ \sigma}{\mathbf{mgu}\text{-}\mathsf{Prod}\ s\ t\ (\mathsf{yes}\ (n; \sigma))}$$

$$\frac{\mathbf{Nothing}\ (\mathbf{Unifies}\ s\ t)}{\mathbf{mgu}\text{-}\mathsf{Prod}\ s\ t\ \mathsf{no}}$$

Related work

Of course, almost everything is not new here, except:

- the EPIGRAM notation, and the uniformity which it brings to the preentation of functions, specifications and proofs;
- the precise way in which the construction of f-Ind and f-ind is automated;
- the identification (but not the automatic synthesis!) of the gap between f-Ind and f-Prod; heuristics to guide the choice of appropriate such relations, based on constructor-form analyses;
- the application to a compositional account of the correctness of some interesting programs, differing from published analyses, *e.g.* unification.

Nonetheless, it would be useful to relate what we are doing to

- Aubin, Boyer-Moore, Bundy 'recursion analysis';

- Bove-Capretta definition and correctness proofs of non-structural recursive functions in type theory;
- Bertot-Balaa fixpoint definitions in type theory.

Bove-Capretta domain predicates

- Recuperate Haskell programs/functions f, by
- Inductively defining the domain $\mathbf{Dom}(d)$ of definition of f,
- Giving rise to a *total* function on $d : D, p : \mathbf{Dom}(d)$
- Need a 'covering function' $\mathbf{allDom} : \forall d : D.\mathbf{Dom}(d)$

Now,

- $\mathbf{Dom}(d)$ is isomorphic (extensionally equivalent) to $\exists y.\mathbf{f}\text{-Ind}(d, y)$;
- The covering function is just the totality proof;
- The total function may be recovered by composing the covering function with the obvious projection;
- Rather than consider simply second projection, we may first do induction on $\mathbf{f}$-Ind, then second projection: this gives us a (possibly more informative) analysis of the form of the output of $\mathbf{f}$-Ind.

. . .

Further Work

This is still work in progress. . . 35 years after McCarthy-Painter! Other things we currently are considering

- re-analysis of compiler correctness proof in the presence of *exceptions* (Hutton& Wright, submitted as a Functional Pearl to JFP);
- functions returning higher-type objects, bringing extensionality into the picture; examples include various 'logical relations'-style constructions, as well as the work of Bertot& Balaa (2000, 2002);
- more detailed account of the relationship with existing related work as indicated;
- . . .

Conclusions

EPIGRAM was explicitly designed with ‘informative case analysis’ as the main programming methodology, and with the uniformity of induction over datatypes, function definitions and relational properties as the key theoretical device to support this.

McCarthy-Painter induction is available for free on top of function definition in EPIGRAM, and provides a zero’th-order approach to specification and verification of EPIGRAM programs, with clues as to how to proceed in a more sophisticated way.

The associated inductive graphs provide a way to legitimize and establish function definitions, even when no term with the corresponding *computational* behaviour is available.

Questions?