Mescal

Requirements and Architecture

Lambert Meertens

CWI
Utrecht University
Kestrel Institute

Why?

- A piece of a typical calculation:

  Can the Flipflop Lemma be applied to $\mathsf{hsh}(f, g)$? Try to express this in the form of $\xi((\xi^{-1} f) \odot (\xi^{-1} g))$ for some ξ:

$$
\begin{array}{ll}
 & \mathsf{hsh}(f, g) \\
= & \qquad \{ \text{ definition of } \mathsf{hsh} \} \\
 & (\mathsf{L} f \oplus g) \odot (f \oplus \mathsf{R} g) \\
= & \qquad \{ \text{ definition of } \odot \} \\
 & Cl\, \{ x \bowtie y \mid x \in \mathsf{L} f \oplus g,\ y \in f \oplus \mathsf{R} g \} \\
= & \qquad \{ \cdots \} \\
 & \ldots
\end{array}
$$

Program calculation

- This style of calculation is used to derive programs from specifications, typically by "massaging" them into a form so that some theorem applies
- Usually this involves "solving for unknowns" while checking applicability conditions
- Often this only succeeds by creating a local "minitheory" with its own definitions and lemmas

Problems ...

- Finding the minitheory that works may involve much trial and error
- With each revision many earlier steps must be rechecked for validity
- The expressions involved quickly become fairly large
- The resulting program is supposed to be "correct by construction", but trivial calculation errors easily sneak in, in particular when revising an earlier calculation

Using pencil and paper

- Pencil and paper is flexible: notations can be optimized for calculation; compare

  $$\frac{\mathsf{d}}{\mathsf{dx}} FG \quad = \quad \frac{\mathsf{d}F}{\mathsf{d}x} G + F \frac{\mathsf{d}G}{\mathsf{d}x}$$

  with

  ```
  diff(times(F,G),x) =
  plus(times(diff(F,x),G),
  times(F,diff(G,x)))
  ```

- No safeguards against errors
- Most revisions require tedious copying
- Easy to loose track of what still must be proved

Using a prover

- Notationally typically more rigid
- Creating theories is more work than you want to spend on disposable minitheories
- Not always easy to postpone proof obligations
- Easy to loose track of what you are doing
- Revision is still cumbersome

Primary aim of *Mescal*

- Reduce tedium and chance of clerical errors
- while retaining as much as possible of the flexibility and “lightness” of working with pencil and paper

Some non-requirements

- *Mescal* finds the proofs for you
- A *Mescal*- "verified" proof is correct
- *Mescal* compels its users to follow good mathematical standards

Primary requirements

- WYSIWYG editing*
- Users can define their own notations*
- Notation can be changed on the fly*
- Non-formal and formal text can be freely mixed, just as in a research paper*
- The formal parts may come from multiple formalisms, and may be heterogeneous
- Users can define their own formalisms
- Validity can be checked to the level desired by the user (from not-at-all to fully)
- Validity checking uses "spreadsheet evaluation": once turned on, it is automatically rechecked upon changes to the text

* Features of Mathspad

Some possible formalisms

- Allegorical calculus (*Algebra of Programming*)
- Category theory
- Relational calculus
- Lattice theory
- Polymorphic lambda calculus
- Haskell
- Java
- Analysis, Algebra, Geometry

Mescal as a kernel system

- *Mescal* has no built-in theories but a meta-formalism that allows the definition of formalisms

- Leverage will have to come from the accumulated creation of libraries of theories

- *Mescal* has only rudimentary theorem-proving capabilities, but will offer facilities for hooking up to "external engines" (provers, type-checkers, compilers, interpreters, computer-algebra systems, ...)

- *Mescal* has roughly the native proof-checking power of Automath

Formalisms

- *Forms* are generated by formation rules of a multi-sorted algebra
- Each form belongs to a *formalism*
- Forms appear in some *context*
- The context may impose additional requirements on the forms
- Forms may carry *certificates* issued by some formalism
- Certificates are again forms

Examples of certificates

FORM :: *CERTIFICATE*

- Expression E :: has type τ
- Proposition P :: holds
- Proof f :: is constructive
- Program p :: is type-correct
- Program p :: implements spec S
- Function f :: is uniformly continuous

Certificates

- are created by *certification rules* (which are like logic inference rules, but may involve arbitrary computations)
- identify "assumptions" used from the context
- usually identify a *witness* (or the information needed to reconstruct it)

Live constraints

$$X \quad \text{—(R)—} \quad Y$$

Objects X and Y are “linked” by constraint R:

- At all times X (R) Y holds
- When X changes, Y is made to change (if necessary) as well, so that the validity of X (R) Y is restored.
- Likewise when Y changes

Example: X —($\leq$)— Y

- ↯ indicates "spontaneous" change
- ⇣ indicates constraint-restoring change

X	Y
3	4
↯	
2	4
⇣	↯
1	1
	↯
1	4

Constraints may form a network

- Example: X —($\leq$)— Y —(SQ)— Z

X	Y	Z
3	4	16
5	5	25
4.69	4.69	22

Constraints may involve structure

- Example: X —(MAP(SQ))— Y

X	Y
[1,3]	[1,9]
[2,3]	[4,9]
[2,3,1]	[4,9,1]

Implementation of constraints

- Let $\mathsf{R} : A \sim B$ be a ditotal relation
- A maintainer of R is a pair of functions

  $$\begin{aligned} \lhd &: A \times B \to A \\ \rhd &: A \times B \to B \end{aligned}$$

  such that for all $x \in A$ and $y \in B$

  $$(x \lhd y)(\mathsf{R})y \quad \text{and} \quad x(\mathsf{R})(x \rhd y)$$

- After a change to y, $x := x \lhd y$ is executed, and likewise for x
- In addition, the change should be "as small as possible"

The certification rules

- are embodied in “edit steps” which may be performed on forms
- An edit step takes zero or more forms as parameters and then computes (if possible) a new form as result
- The edit step may use the parameters, as well as any certificates they carry, to compute a certificate for the new form
- The computation procedure is expressed as and recorded in the form of a constraint network

Example edit step in calculation

- Edit focus is on:

$$f(x) \leq f(y)$$

- Apply command "MONOTONICITY"
- Result:

$$\begin{array}{ll} & f(x) \leq f(y) \\ \Leftarrow & \{\ f \text{ is monotonic }\} \\ & x \leq y \end{array}$$

Edit step "MONOTONICITY"

- take term XRY where R is an order
- determine lsg $\langle C[\_], x, y\rangle$
  such that $X = C[x]$, $Y = C[y]$
- determine appropriate domain order r
- create proof obligation
  $\pi :=$ "is-monotonic(C)"
- produce new term $XRY \;\Leftarrow \{\pi\}\; x\,r\,y$
- set up the constraint network
- if OK, replace term by new term

Resulting term with constraints

- The term:

$$\begin{array}{ll} & X\,R\,Y \\ \Leftarrow & \qquad \{\ \pi\ \} \\ & x\,r\,y \end{array}$$

- The constraints:

$$\begin{array}{rcl} \langle X, Y\rangle & \text{—(LSG)—} & \langle C[\_], x, y\rangle \\ \langle C[\_], x, y\rangle & \text{—(ADO)—} & r \\ \langle C[\_], R, r\rangle & \text{—(PrObl}_{\mathsf{M}}\text{)—} & \pi \end{array}$$

Specifying the edit step

- Can fully be done by supplying
  - *template* terms for source/result
  - the constraint network in symbolic form
  - constraint definitions
- ADO = Appropriate Domain Order
  - use knowledge about $C[\_]$
    and/or type of x and y
  - obtain from prover or use heuristic

Discharging proof obligations

- In principle the task of the user
- Dispatch lazily to some prover
  (represented as a constraint)
  - "internal" prover
  - external prover(s)
  - the user
- Internal prover for trivial cases:
  (if f has attribute "is-monotonic"
  this counts as a proof)
  and maybe less trivial ones:
  (if f and g are monotonic,
  so is $f(g(\_))$)
- Edit step in theorems/lemmas:
  add obligation to the assumptions

Other views

- The approach is not specific
  to the calculational proof style:
  the term:

$$
\begin{array}{ll}
 & X\,R\,Y \\
\Leftarrow & \qquad \{\ \pi\ \} \\
 & x\,r\,y
\end{array}
$$

  may also be presented thus:

$$
\frac{\pi \qquad\qquad x\,r\,y}{X\,R\,Y}
$$

Major open issues

- A convenient “scripting language” for giving constraint definitions
- A convenient “scripting language” for specifying hook-up to external engines (protocol!)
- Facilities for formal diagrams