Programming of Autonomous Vehicles with Static Guarantees

Christoph Herrmann
University of St Andrews
http://www.cs.st-andrews.ac.uk/~ch

June 17, 2011
Scottish Theorem Proving Meeting
Dundee

Motivation

Autonomous systems must have a very high reliability

Important: guarantees on maximum time / resource consumption

Solution: Programming in a DSL with dependent types

- Time, fuel, location etc. become part of the operational semantics of the programming language
- User-defined types specify important system properties
  - programs which do not meet these properties simply do not compile and are never executed
- Advantages to a separate analysis
  - certificates come automatically with the program
  - compositionality of the design: trust that available components are sufficient to build a system which meets the requirements
  - properties to verify might be so sophisticated that the design of the program needs to support their proofs

Suggestions for autonomous vehicle programming

Layers

1. Strategic layer: instructions part of a global plan, interpreted by each AV and translated to programs at the execution layer
2. Execution layer: instructions interpreted by controllers of the AV, e.g. for motors and sensors
3. Operational layer: instructions are atomic device operations with a clear operational semantics: e.g. make a step or a turn

Usage

- Mission programs
  - written or generated at the strategic layer
  - high-level view of properties, no details
- Translation between the layers certified to preserve properties

Research contributions

Methodology and prototype implementation

- Domain-specific language (layers) embedded into Agda
- Bottom-up construction of operational behaviour with the according proofs of state changes (time, location)
- Translation between the layers

Technical challenges

- Verification of properties
  - with non-polynomial expressions
  - in several unbounded variables
  - for systems with an infinite state space
- Compile-time guarantees for choices depending on run-time values

Simple Agda example with static certificates: vectors

Vector definition

```
data Vec (a : Set) : ℕ → Set where
  [] : Vec a zero
  _::_ : {n : ℕ} → a → Vec a n → Vec a (suc n)
```

Note

- Arguments of Vec *a* *n*
  - *a* is a polymorphic type variable, same for all elements of the data structure
  - *n* is an index, can vary from element to element
- Underscore denotes argument position of infix/mixfix operators
- Curly braces embrace implicit arguments (e.g. for delaration of type variables)

Safe vector indexing

Implementation

```
_!_ : {n : ℕ} {a : Set} → Vec a n → Fin n → a
[] ! ()
(x :: xs) ! zero    = x
(x :: xs) ! (suc i) = xs ! i
```

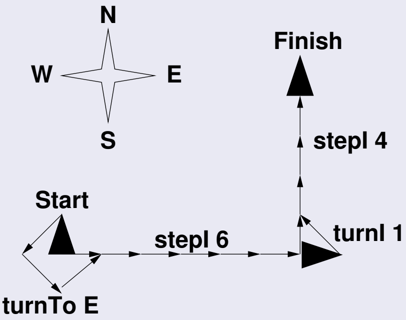
Note

- *Index out of bounds* not possible, because index must be in Fin n : numbers $\{0, .., n{-}1\}$
- () : absurd pattern (roughly: no inhabitant for Fin n)

Setting

Vehicles can only be located at points of a 2D integer grid

Example tour



Operations

- Initialisation: the vehicle is dropped at a particular location
- Step: the vehicle can move to the next grid point
- Turn: the vehicle can change orientation by turning left, i.e. (N → W, W → S, S → E, E → N).
- Wait: the vehicle can wait for a given number of time units

AV data type

Enumeration type for directions

```
data Direction : Set where
  N W S E : Direction
```

Record type for AV state

```
record AV : Set where
  constructor mkAV
  field
    time : ℕ
    dir  : Direction
    px   : ℤ
    py   : ℤ
```

Turn operations

Turn operation

$$\begin{aligned} &\text{turn} : \text{Direction} \\ &\quad \to \text{Direction} \\ &\text{turn N} = \text{W} \\ &\text{turn W} = \text{S} \\ &\text{turn S} = \text{E} \\ &\text{turn E} = \text{N} \end{aligned}$$

Enumerating directions

$$\begin{aligned} &\text{dirNo} : \text{Direction} \\ &\quad \to \text{Fin } 4 \\ &\text{dirNo N} = \sharp\, 0 \\ &\text{dirNo W} = \sharp\, 1 \\ &\text{dirNo S} = \sharp\, 2 \\ &\text{dirNo E} = \sharp\, 3 \end{aligned}$$

Repeated turn operation

$$\begin{aligned} &\text{turnN} : \{n : \mathbb{N}\} \to \text{Fin } n \to \text{Direction} \to \text{Direction} \\ &\text{turnN zero } d = d \\ &\text{turnN (suc } n)\ d = \text{turn (turnN } n\ d) \end{aligned}$$

We can turn to each direction within 3 units

Calculation of the number of turns with a proof

$$\text{turnProof} : \{\textit{from to} : \text{Direction}\} \to \Sigma\,(\text{Fin } 4)\,(\lambda\, c \to \textit{to} \equiv \text{turnN}\ c\ \textit{from})$$

$$\text{turnProof}\ \{\text{N}\}\ \{\text{N}\} = (\textit{turnCount}\ \text{N N}, \text{refl})$$
$$\text{turnProof}\ \{\text{N}\}\ \{\text{W}\} = (\textit{turnCount}\ \text{N W}, \text{refl})$$

... and 14 more combinations

Note

- Σ delivers a dependent pair of the number of turns ($\in$ Fin 4) and a certificate that this number satisfies the requirement
- refl constructs a trivial equation, tells us that Agda can prove the equation by evaluation of the two sides
- The 16 refl's are internally different and must be listed separately even if the cases are enumerated automatically.

Lifting turn operations to the AV state

Single turn

$$\mathsf{turnAV} : \mathsf{AV} \rightarrow \mathsf{AV}$$
$$\mathsf{turnAV}\ (\mathsf{mkAV}\ t\ d\ x\ y) = \mathsf{mkAV}\ (\mathsf{suc}\ t)\ (\mathsf{turn}\ d)\ x\ y$$

The turnAV operation necessarily increments the time counter

Iteration of turns

$$\mathsf{turnAVI} : \mathbb{N} \rightarrow \mathsf{AV} \rightarrow \mathsf{AV}$$
$$\mathsf{turnAVI}\ \mathsf{zero}\ x = x$$
$$\mathsf{turnAVI}\ (\mathsf{suc}\ n)\ x = \mathsf{turnAV}\ (\mathsf{turnAVI}\ n\ x)$$

These functions lift the semantics between the layers, they are not (necessarily) evaluated during the operation of the system.

Step operation

Effect on location

$$
\begin{aligned}
\text{step} &: \text{Direction} \to (\mathbb{Z} \times \mathbb{Z}) \to (\mathbb{Z} \times \mathbb{Z}) \\
\text{step N } (x, y) &= (x, y +^{\mathbb{Z}} (+\,1)) \\
\text{step W } (x, y) &= (x -^{\mathbb{Z}} (+\,1), y) \\
\text{step S } (x, y) &= (x, y -^{\mathbb{Z}} (+\,1)) \\
\text{step E } (x, y) &= (x +^{\mathbb{Z}} (+\,1), y)
\end{aligned}
$$

Note: representation of integers

- let $n \in \mathbb{N}$
- $(+\,n)$ lifts n
- $(-[1+\,n\,])$ represents $-(\text{suc } n)$

Lifting step operation to AV state

Lifting it to AV state

```
stepAV : AV → AV
stepAV (mkAV t d x y) with step d (x, y)
... | (x1, y1) = mkAV (suc t) d x1 y1
```

Note

- **with** permits pattern matching after evaluation
- Pattern matching is the principal way to gain type information

Repeated step operations

```
stepAVI : ℕ → AV → AV
stepAVI zero x = x
stepAVI (suc n) x = stepAV (stepAVI n x)
```

OpLang: embedded DSL for atomic AV operations

Encapsulation to achieve safety

- So far there was no protection against arbitrary changes of state entries
- Encapsulation into a data type with state changes associated with constructor application achieves consistency

```
data OpLang : AV → AV → Set where
   Id    : {s : AV} → OpLang s s
   Wait  : {t : ℕ} {d : Direction} {x y : ℤ} (k : ℕ)
         → OpLang (mkAV t d x y) (mkAV (k +ℕ t) d x y)
   Turn  : {s : AV} → OpLang s (turnAV s)
   Step  : {s : AV} → OpLang s (stepAV s)
   _>>=_ : {s1 s2 s3 : AV}
         → OpLang s1 s2 → OpLang s2 s3 → OpLang s1 s3
   IfDir : {s1 s2 : AV} (d : Direction) → OpLang s1 s2
         → OpLang s1 s2 → OpLang s1 s2
```

Operational semantics of OpLang

$$
\begin{array}{l}
\mathsf{interp} : \{s1\ s2 : \mathsf{AV}\} \to \mathsf{OpLang}\ s1\ s2 \to \mathsf{AV} \to \mathsf{AV} \\
\mathsf{interp\ Id}\ x = x \\
\mathsf{interp\ (Wait\ \_)}\ x = x \\
\mathsf{interp\ Turn}\ x = \mathsf{turnAV}\ x \\
\mathsf{interp\ Step}\ x = \mathsf{stepAV}\ x \\
\mathsf{interp}\ (r >>= f)\ x = \mathsf{interp}\ f\ (\mathsf{interp}\ r\ x) \\
\mathsf{interp}\ (\mathsf{IfDir}\ d\ x\ y)\ s\ \textbf{with}\ \mathsf{eqDir}\ (AV.dir\ s)\ d \\
\ldots \mid true = \mathsf{interp}\ x\ s \\
\ldots \mid false = \mathsf{interp}\ y\ s
\end{array}
$$

Note

- Here all information is already available in the state changes, but one could imagine that extended behaviour is implemented which does not need static guarantees
- Likewise code (e.g. in C) could be generated for OpLang programs

ExLang: embedded DSL at the execution layer

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{ExLang} : \mathsf{AV} \to \mathsf{AV} \to \mathsf{Set}\ \textbf{where} & & \\
\quad \mathsf{Emb} & : & \{s1\ s2 : \mathsf{AV}\} \to \mathsf{OpLang}\ s1\ s2 \to \mathsf{ExLang}\ s1\ s2 \\
\quad \_{>}{>}{>}{=}\_ & : & \{s1\ s2\ s3 : \mathsf{AV}\} \\
 & & \to \mathsf{ExLang}\ s1\ s2 \to \mathsf{ExLang}\ s2\ s3 \to \mathsf{ExLang}\ s1\ s3 \\
\quad \mathsf{StepI} & : & \{s : \mathsf{AV}\}\ (n : \mathbb{N}) \to \mathsf{ExLang}\ s\ (\mathsf{stepAVI}\ n\ s) \\
\quad \mathsf{TurnI} & : & \{s : \mathsf{AV}\}\ (k : \mathbb{N}) \to \mathsf{ExLang}\ s\ (\mathsf{turnAVI}\ k\ s) \\
\quad \mathsf{TurnTo} & : & \{t : \mathbb{N}\}\ \{from : \mathsf{Direction}\}\ (to : \mathsf{Direction})\ \{x\ y : \mathbb{Z}\} \\
 & & \quad \to \mathsf{ExLang}\ (\mathsf{mkAV}\ t\ from\ x\ y)\ (\mathsf{mkAV}\ (3 +^{\mathbb{N}} t)\ to\ x\ y)
\end{array}
$$

Note

- TurnTo gives a static guarantee in presence of a dynamic argument (*to*)
- The semantics will be defined by a translation to OpLang
- It makes sense to generate code from this language directly, provided the specified behaviour (which is still simple) is established

Translation from ExLang to OpLang

Auxiliary functions

$$
\begin{aligned}
&\mathsf{genNSteps} : \{s : \mathsf{AV}\}\ (n : \mathbb{N}) \to \mathsf{OpLang}\ s\ (\mathsf{stepAVI}\ n\ s)\\
&\mathsf{genNSteps}\ \mathsf{zero} = \mathsf{Id}\\
&\mathsf{genNSteps}\ (\mathsf{suc}\ n) = \mathsf{genNSteps}\ n >\!>\!= \mathsf{Step}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{genNTurns} : (n : \mathbb{N})\ \{s : \mathsf{AV}\} \to \mathsf{OpLang}\ s\ (\mathsf{turnAVI}\ n\ s)\\
&\mathsf{genNTurns}\ \mathsf{zero} = \mathsf{Id}\\
&\mathsf{genNTurns}\ (\mathsf{suc}\ n) = \mathsf{genNTurns}\ n >\!>\!= \mathsf{Turn}
\end{aligned}
$$

Translation

$$
\begin{aligned}
&\mathsf{translate1} : \{s1\ s2 : \mathsf{AV}\} \to \mathsf{ExLang}\ s1\ s2 \to \mathsf{OpLang}\ s1\ s2\\
&\mathsf{translate1}\ (\mathsf{Emb}\ x) = x\\
&\mathsf{translate1}\ (x >\!>\!>\!= y) = \mathsf{translate1}\ x >\!>\!= \mathsf{translate1}\ y\\
&\mathsf{translate1}\ (\mathsf{StepI}\ n) = \mathsf{genNSteps}\ n\\
&\mathsf{translate1}\ (\mathsf{TurnI}\ n) = \mathsf{genNTurns}\ n
\end{aligned}
$$

Bounded static variation (the "trick" in partial evaluation)

Problem

- Turning into a certain direction without assumption about previous direction
- Comparison with a non-static value yields a non-static value

Solution

Jones, Gomard, Sestoft [1993]: Partial Evaluation and Automatic Program Generation, Section 4.8.3: Variables of bounded static variation:
"It often happens in partial evaluation that a variable seems dynamic since it depends on dynamic input, but only takes on finitely many values. In such cases a bit of reprogramming can yield much better results from partial evaluation. This kind of reprogramming, or program transformation, which does not alter the standard meaning of the program but leads to better residual programs is called a *binding-time improvement*."

Translation of TurnTo operations

Function turnProof revisited

```
turnProof : {from to : Direction}
  → Σ (Fin 4) (λ c → to ≡ turnN c from)
```

Translation Note: case distinction remains in executable!

```
translate1 {mkAV _ from _ _} (TurnTo to)
  with turnProof {from} {to}
... | (c, proof) with c
... | zero rewrite proof = Wait 3
... | (suc zero) rewrite proof = Turn >>= Wait 2
... | (suc (suc zero))
  rewrite proof = (Turn >>= Turn) >>= Wait 1
... | (suc (suc (suc zero)))
  rewrite proof = (Turn >>= Turn) >>= Turn
... | (suc (suc (suc (suc ()))))
```

Certificate for a complex AV task

To show

There is a program in OpLang (or ExLang) that for any given start location (xS, yS) and final location (xF, yF) it makes the vehicle move from start to final location within $|xF - xS| + |yF - yS| + 4$ units of time.

Idea for implementation and proof

- Walk in one direction, turn left, walk in the other direction
- Initial turn takes at most three units
- Walk into one direction takes $|xF - xS|$ or $|yF - yS|$ units
- Intermediate turn left takes one unit

Note

The idea makes the tight connection between the design of the algorithm and its proof obvious!

Walking north m steps

$$\mathsf{walkNorth} : \{t : \mathbb{N}\}\ (\boxed{m} : \mathbb{N})\ \{x\ y : \mathbb{Z}\} \to$$
$$\mathsf{ExLang}\ (\mathsf{mkAV}\ t\ \mathsf{N}\ x\ y)\ (\mathsf{mkAV}\ (\boxed{m} +^{\mathbb{N}} t)\ \mathsf{N}\ x\ (y +^{\mathbb{Z}} (+\ \boxed{m})))$$

$$\mathsf{walkNorth}\ \{t\}\ \boxed{m}\ \{x\}\ \{y\}$$
$$\textbf{rewrite}\ \mathsf{lawNorth}\ \{\boxed{m}\}\ \{t\}\ \{x\}\ \{y\} = \mathsf{StepI}\ \boxed{m}$$

rewrite lawNorth makes the type of walkNorth fit stepI

Auxiliary law

$$\mathsf{lawNorth} : \{m\ t : \mathbb{N}\}\ \{x\ y : \mathbb{Z}\}$$
$$\to (\mathsf{mkAV}\ (m +^{\mathbb{N}} t)\ \mathsf{N}\ x\ (y +^{\mathbb{Z}} (+\ m))$$
$$\equiv \mathsf{stepAVI}\ m\ (\mathsf{mkAV}\ t\ \mathsf{N}\ x\ y))$$

Proof of auxiliary law (base case)

```
lawNorth : {m t : ℕ} {x y : ℤ}
  → (mkAV (m +ℕ t) N x (y +ℤ (+ m))
    ≡ stepAVI m (mkAV t N x y))
lawNorth {0} {t} {x} {y} =
begin
  mkAV (0 +ℕ t) N x (y +ℤ (+ 0))
 ≡⟨ refl ⟩
  mkAV t N x (y +ℤ (+ 0))
 ≡⟨ cong (λ z → mkAV t N x z) (+ℤ − identity {y}) ⟩
  mkAV t N x y
 ≡⟨ refl ⟩
  stepAVI 0 (mkAV t N x y)
∎
```

Agda recognises inductive proofs

- Have shown above: lawNorth {0}
- Next we will show: lawNorth {m} ⇒ lawNorth {suc m}

Proof of auxiliary law (inductive case)

```
lawNorth : {m t : ℕ} {x y : ℤ}
   → (mkAV (m +ℕ t) N x (y +ℤ (+ m)) ≡ stepAVI m (mkAV t N x y))


lawNorth {suc m} {t} {x} {y} =
begin
  mkAV ((suc m) +ℕ t) N x (y +ℤ (+ (suc m)))
 ≡⟨ refl ⟩
  mkAV (suc (m +ℕ t)) N x (y +ℤ (+ (suc m)))
 ≡⟨ cong (λ z → mkAV (suc (m +ℕ t)) N x z)
  (z+sn=z+n+1 {y} {m}) ⟩
  mkAV (suc (m +ℕ t)) N x ((y +ℤ (+ m)) +ℤ (+ 1))
 ≡⟨ refl ⟩
  stepAV (mkAV (m +ℕ t) N x (y +ℤ (+ m)))
 ≡⟨ cong (λ x → stepAV x) (lawNorth {m} {t} {x} {y}) ⟩    -- IND ASS
  stepAV (stepAVI m (mkAV t N x y))
 ≡⟨ refl ⟩
  stepAVI (suc m) (mkAV t N x y)
∎
```

Compositions of turns and walks

Initial turn and single dimension walk (one of four cases)

$$
\begin{array}{l}
\mathsf{turnWalkNorth} \;:\; \{d : \mathsf{Direction}\}\,(m : \mathbb{N}) \to \{t : \mathbb{N}\}\,\{x\;y : \mathbb{Z}\} \\
\quad \to \mathsf{ExLang}\,(\mathsf{mkAV}\;t\;d\;x\;y) \\
\qquad (\mathsf{mkAV}\;(m +^{\mathbb{N}} (3 +^{\mathbb{N}} t))\;\mathsf{N}\;x\;(y +^{\mathbb{Z}} (+\,m))) \\
\mathsf{turnWalkNorth}\;m \;=\; \mathsf{TurnTo}\;\mathsf{N} >\!\!>\!\!>\!\!= \mathsf{walkNorth}\;m
\end{array}
$$

Complete task (one of four cases)

$$
\begin{array}{l}
\mathsf{walkNorthWest} \;:\; \{d : \mathsf{Direction}\}\,(m\;n : \mathbb{N}) \\
\quad \to \{t : \mathbb{N}\}\,\{x\;y : \mathbb{Z}\} \\
\quad \to \mathsf{ExLang}\,(\mathsf{mkAV}\;t\;d\;x\;y) \\
\qquad (\mathsf{mkAV}\;(t +^{\mathbb{N}} m +^{\mathbb{N}} n +^{\mathbb{N}} 4)\;\mathsf{W}\;(x -^{\mathbb{Z}} (+\,m))\;(y +^{\mathbb{Z}} (+\,n))) \\
\mathsf{walkNorthWest}\;\{\_\}\;m\;n\;\{t\}\;\mathbf{rewrite}\;\mathsf{LawMoveYX}\;\{t\}\;\{m\}\;\{n\} \\
\quad = \mathsf{turnWalkNorth}\;n >\!\!>\!\!>\!\!= \mathsf{TurnI}\;1 >\!\!>\!\!>\!\!= \mathit{walkWest}\;m
\end{array}
$$

Simple arithmetic equalities proven automatically

Example

$$\mathrm{LawMoveYX} : \{t\ m\ n : \mathbb{N}\} \rightarrow (t +^{\mathbb{N}} m +^{\mathbb{N}} n +^{\mathbb{N}} 4 \equiv m +^{\mathbb{N}} (1 +^{\mathbb{N}} (n +^{\mathbb{N}} (3 +^{\mathbb{N}} t))))$$

$$\mathrm{LawMoveYX}\ \{t\}\ \{m\}\ \{n\} = \mathrm{solve}\ 3\ (\lambda\ t\ m\ n \rightarrow t :+ m :+ n :+ \mathrm{con}\ 4 := m :+ (\mathrm{con}\ 1 :+ (n :+ (\mathrm{con}\ 3 :+ t))))\ \mathrm{refl}\ t\ m\ n$$

Note

Agda can prove the equivalence of two (equivalent) expressions for instances of commutative rings automatically, involving only variables, the ring constants and ring operations + and *.

Proof almost finished, case distinction based on directions

Bounded static variation again!

- Unbounded number of coordinate distances
- But a bounded number of signs for their differences

```
moveDelta : (dx dy : ℤ) {s : AV}
  → ExLang s (mkAV (AV.time s +ᴺ dx +ᴺ dy +ᴺ 4)
    (finalDir dx dy) (AV.px s +ᶻ dx) (AV.py s +ᶻ dy))
moveDelta dx dy {mkAV t d xS yS}
  with dx | dy
... | + idx | + idy          rewrite abs1 {xS} {idx} | abs1 {yS} {idy}
  = walkEastNorth idx idy
... | + idx | − [1 + idy] rewrite abs1 {xS} {idx} | abs2 {yS} {idy}
  = walkSouthEast idx (suc idy)
... | − [1 + idx] | + idy rewrite abs2 {xS} {idx} | abs1 {yS} {idy}
  = walkNorthWest (suc idx) idy
... | − [1 + idx] | − [1 + idy] rewrite abs2 {xS} {idx} | abs2 {yS} {idy}
  = walkWestSouth (suc idx) (suc idy)
```

Expressing implementation in terms of final coordinates

Mapping of the AV state, includes certificate

$$
\begin{array}{l}
\mathsf{finalStateMoveTo} : \mathbb{Z} \to \mathbb{Z} \to \mathsf{AV} \to \mathsf{AV} \\
\mathsf{finalStateMoveTo}\ \mathit{xFin}\ \mathit{yFin}\ s \\
\quad \textbf{with}\ (\mathit{xFin} -^{\mathbb{Z}} \mathit{AV.px}\ s) \mid (\mathit{yFin} -^{\mathbb{Z}} \mathit{AV.py}\ s) \\
\ldots \mid \mathit{dx} \mid \mathit{dy} = \\
\quad \mathsf{mkAV}\ (\mathit{AV.time}\ s +^{\mathbb{N}} \mathit{dx} +^{\mathbb{N}} \mathit{dy} +^{\mathbb{N}} 4)\ (\mathsf{finalDir}\ \mathit{dx}\ \mathit{dy}) \\
\qquad (\mathit{AV.px}\ s +^{\mathbb{Z}} \mathit{dx})\ (\mathit{AV.py}\ s +^{\mathbb{Z}} \mathit{dy})
\end{array}
$$

From distances to absolute coordinates

$$
\begin{array}{l}
\mathsf{moveTo} : (\mathit{xFin}\ \mathit{yFin} : \mathbb{Z})\ \{s : \mathsf{AV}\} \\
\quad \to \mathsf{ExLang}\ s\ (\mathsf{finalStateMoveTo}\ \mathit{xFin}\ \mathit{yFin}\ s) \\
\mathsf{moveTo}\ \mathit{xFin}\ \mathit{yFin}\ \{\mathsf{mkAV}\ t\ d\ \mathit{xStart}\ \mathit{yStart}\} \\
\quad = \mathsf{moveDelta}\ (\mathit{xFin} -^{\mathbb{Z}} \mathit{xStart})\ (\mathit{yFin} -^{\mathbb{Z}} \mathit{yStart})
\end{array}
$$

StLang: embedded DSL at the strategic level

Language Definition

$$\textbf{data}\ \mathsf{StLang} : \mathsf{AV} \to \mathsf{AV} \to \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{MoveTo} : \{s : \mathsf{AV}\}\ (xFin\ yFin : \mathbb{Z})$$
$$\qquad \to \mathsf{StLang}\ s\ (\mathsf{finalStateMoveTo}\ xFin\ yFin\ s)$$

Translation to ExLang

$$\mathsf{translate2} : \{s1\ s2 : \mathsf{AV}\} \to \mathsf{StLang}\ s1\ s2 \to \mathsf{ExLang}\ s1\ s2$$
$$\mathsf{translate2}\ (\mathsf{MoveTo}\ \{s\}\ xFin\ yFin) = \mathsf{moveTo}\ xFin\ yFin\ \{s\}$$

Note

- The operations and certificates are expressed concisely, although the implementation and proofs are quite complex
- The translation preserves the mapping between start state *s1* and final state *s2*, so the properties are actually implemented

Conclusions

Main contributions

- Methodology: certified translation between layers of a domain-specific language, bridging the gap between low-level operations and user interface
- Generation of a non-trivial AV program with unbounded parameters that comes with static guarantees

General experiences gained

- Simpler to prove simple functions than larger ones, so construct your program using many simple functions.
- Proof and implementation should go along with each other.
- Use data types to encapsulate certificates in indices and intermediate representations (here DSLs) to match against.
- Pattern matching can exploit bounded static variation.
- Automatic solvers beyond the ring solver are desirable.

Work in progress

Challenge

- What can be done if the run-time choices are unbounded, e.g. by using an external optimisation function?
- It is not useful if external functions need to anticipate the required certificates, but they must implement the desired behaviour, including resource consumption.

Our ongoing approach

- Provide a simple but safe default strategy with a certificate
- The external function is applied when desired
  - its result is tested algorithmically trying to build up the representation of a certificate
  - if test successful then certificate permits use of the result, otherwise default strategy is used