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**

```agda
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 declaration of type variables)
Safe vector indexing

Implementation

```haskell
! : \{ n : \mathbb{N} \} \{ a : \text{Set} \} \rightarrow \text{Vec} a n \rightarrow \text{Fin} n \rightarrow a
[] ! ()
(x :: xs) ! zero = x
(x :: xs) ! (\text{suc} \ i) = xs ! i
```

Note

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

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**

<table>
<thead>
<tr>
<th>Direction</th>
<th>turn operation</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N = W</td>
<td>W</td>
</tr>
<tr>
<td>W</td>
<td>W = S</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>S = E</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>E = N</td>
<td>N</td>
</tr>
</tbody>
</table>

**Enumerating directions**

<table>
<thead>
<tr>
<th>Direction</th>
<th>dirNo operation</th>
<th>Fin 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N = # 0</td>
<td># 0</td>
</tr>
<tr>
<td>W</td>
<td>W = # 1</td>
<td># 1</td>
</tr>
<tr>
<td>S</td>
<td>S = # 2</td>
<td># 2</td>
</tr>
<tr>
<td>E</td>
<td>E = # 3</td>
<td># 3</td>
</tr>
</tbody>
</table>

**Repeated turn operation**

<table>
<thead>
<tr>
<th>n : N</th>
<th>turnN operation</th>
<th>Fin n</th>
<th>Direction</th>
<th>turnN operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>n = d</td>
<td>n</td>
<td>Direction</td>
<td>n = d</td>
</tr>
<tr>
<td>n</td>
<td>(suc n) d = turn (turnN n d)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can turn to each direction within 3 units

Calculation of the number of turns with a proof

\[
\text{turnProof} : \{ \text{from to} : \text{Direction} \} \\
\rightarrow \Sigma (\text{Fin 4}) (\lambda c \rightarrow \text{to} \equiv \text{turnN c from})
\]

\[
\text{turnProof} \{ N \} \{ N \} = (\text{turnCount} N N, \text{refl})
\]

\[
\text{turnProof} \{ N \} \{ W \} = (\text{turnCount} N W, \text{refl})
\]

... and 14 more combinations

Note

- \(\Sigma\) delivers a dependent pair of the number of turns \((\in \text{Fin 4})\) and a certificate that this number satisfies the requirement.

- \(\text{refl}\) constructs a trivial equation, tells us that Agda can prove the equation by evaluation of the two sides.

- The 16 \(\text{refl}\)'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**

\[
\text{turnAV} : \text{AV} \rightarrow \text{AV} \\
\text{turnAV} (\text{mkAV} \ t \ d \ x \ y) = \text{mkAV} (\text{suc} \ t) (\text{turn} \ d) \ x \ y
\]

The \text{turnAV} operation necessarily increments the time counter

**Iteration of turns**

\[
\text{turnAVI} : \mathbb{N} \rightarrow \text{AV} \rightarrow \text{AV} \\
\text{turnAVI} \ \text{zero} \ x = x \\
\text{turnAVI} (\text{suc} \ n) \ x = \text{turnAV} (\text{turnAVI} \ n \ x)
\]

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

Effect on location

\[
\text{step} : \text{Direction} \rightarrow (\mathbb{Z} \times \mathbb{Z}) \rightarrow (\mathbb{Z} \times \mathbb{Z})
\]

\[
\text{step } N (x, y) = (x, y + \mathbb{Z} (+ 1))
\]

\[
\text{step } W (x, y) = (x - \mathbb{Z} (+ 1), y)
\]

\[
\text{step } S (x, y) = (x, y - \mathbb{Z} (+ 1))
\]

\[
\text{step } E (x, y) = (x + \mathbb{Z} (+ 1), y)
\]

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

\[ \text{stepAV : AV} \rightarrow \text{AV} \]
\[ \text{stepAV (mkAV t d x y) with step } d \ (x, y) \]
\[ \ldots \mid (x1, y1) = \text{mkAV (suc t) d x}1 \ y1 \]

Note

- \textbf{with} permits pattern matching after evaluation
- Pattern matching is the principal way to gain type information

Repeated step operations

\[ \text{stepAVI : } \mathbb{N} \rightarrow \text{AV} \rightarrow \text{AV} \]
\[ \text{stepAVI zero } x = x \]
\[ \text{stepAVI (suc } n) \ x = \text{stepAV (stepAVI } n \ x) \]
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 + 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

interp : \{ s1 \ s2 : AV \} \rightarrow OpLang \ s1 \ s2 \rightarrow AV \rightarrow AV
interp \ Id \ x = x
interp (Wait \ _) \ x = x
interp \ Turn \ x = turnAV \ x
interp \ Step \ x = stepAV \ x
interp (r \ >>= \ f) \ x = interp \ f \ (interp \ r \ x)
interp (IfDir \ d \ x \ y) \ s \ with \ eqDir \ (AV.dir \ s) \ d
... | true = interp \ x \ s
... | false = interp \ y \ s

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

\[
\text{data ExLang : } AV \rightarrow AV \rightarrow \text{Set where}
\]

\[
\begin{align*}
\text{Emb} : \{ s1 \ s2 : AV \} & \rightarrow \text{OpLang } s1 \ s2 \rightarrow \text{ExLang } s1 \ s2 \\
\triangleright\triangleright\triangleright\triangleright\triangleright\triangleright : \{ s1 \ s2 \ s3 : AV \} & \rightarrow \text{ExLang } s1 \ s2 \rightarrow \text{ExLang } s2 \ s3 \rightarrow \text{ExLang } s1 \ s3 \\
\text{StepI} : \{ s : AV \} (n : \mathbb{N}) & \rightarrow \text{ExLang } s \ (\text{stepAVI } n \ s) \\
\text{TurnI} : \{ s : AV \} (k : \mathbb{N}) & \rightarrow \text{ExLang } s \ (\text{turnAVI } k \ s) \\
\text{TurnTo} : \{ t : \mathbb{N} \} \{ \text{from} : \text{Direction} \} (\text{to} : \text{Direction}) \{ x \ y : \mathbb{Z} \} & \rightarrow \text{ExLang } (\text{mkAV } t \ \text{from} \ x \ y) \ (\text{mkAV } (3 +^\mathbb{N} t) \ \text{to} \ x \ y)
\end{align*}
\]

Note

- **TurnTo** gives a static guarantee in presence of a dynamic argument (\text{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**

\[
\text{genNSteps} : \{ s : \text{AV} \} (n : \mathbb{N}) \rightarrow \text{OpLang} s \ (\text{stepAVI} \ n \ s) \\
\text{genNSteps} \ zero = \text{Id} \\
\text{genNSteps} \ (\text{suc} \ n) = \text{genNSteps} \ n >>= \text{Step} \\
\text{genNTurns} : (n : \mathbb{N}) \{ s : \text{AV} \} \rightarrow \text{OpLang} s \ (\text{turnAVI} \ n \ s) \\
\text{genNTurns} \ zero = \text{Id} \\
\text{genNTurns} \ (\text{suc} \ n) = \text{genNTurns} \ n >>= \text{Turn}
\]

**Translation**

\[
\text{translate1} : \{ s1 \ s2 : \text{AV} \} \rightarrow \text{ExLang} \ s1 \ s2 \rightarrow \text{OpLang} \ s1 \ s2 \\
\text{translate1} \ (\text{Emb} \ x) = x \\
\text{translate1} \ (x >>= y) = \text{translate1} \ x >>= \text{translate1} \ y \\
\text{translate1} \ (\text{StepI} \ n) = \text{genNSteps} \ n \\
\text{translate1} \ (\text{TurnI} \ n) = \text{genNTurns} \ n
\]
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

\[
\text{turnProof} : \{ \text{from to} : \text{Direction} \} \\
\rightarrow \Sigma (\text{Fin 4}) (\lambda c \rightarrow \text{to} \equiv \text{turnN c from})
\]

Translation  Note: case distinction remains in executable!

\[
\text{translate1} \ \{ \text{mkAV \_ from \_} \} \ (\text{TurnTo to}) \\
\hspace{1em} \text{with turnProof} \ \{ \text{from} \} \ \{ \text{to} \} \\
\hspace{2em} | \ (c, \text{proof}) \ \text{with} \ c \\
\hspace{3em} | \ (\text{zero rewrite proof} = \text{Wait 3}) \\
\hspace{3em} | \ (\text{suc zero) rewrite proof} = \text{Turn >>>= Wait 2} \\
\hspace{3em} | \ (\text{suc (suc zero))} \\
\hspace{4em} \text{rewrite proof} = (\text{Turn >>>= Turn) >>>= Wait 1} \\
\hspace{5em} | \ (\text{suc (suc (suc zero)))} \\
\hspace{6em} \text{rewrite proof} = (\text{Turn >>>= Turn) >>>= Turn} \\
\hspace{7em} | \ (\text{suc (suc (suc (suc ())))})
\]
Certificate for a complex AV task

To show

There is a program in OpLang (or ExLang) that for any given start location \((x_S, y_S)\) and final location \((x_F, y_F)\) it makes the vehicle move from start to final location within \(|x_F-x_S|+|y_F-y_S|+4\) 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 \(|x_F-x_S| \text{ or } |y_F-y_S|\) units
- Intermediate turn left takes one unit

Note

The idea makes the tight connection between the design of the algorithm and its proof obvious!
Proving resource property for walking towards North

Walking north \( m \) steps

\[
\text{walkNorth} : \{ t : \mathbb{N} \} \{ m : \mathbb{N} \} \{ x y : \mathbb{Z} \} \rightarrow \\
\text{ExLang} (\text{mkAV} t \mathbb{N} x y) (\text{mkAV} (m +_\mathbb{N} t) \mathbb{N} x (y +_{\mathbb{Z}} (+ m)))
\]

\[
\text{walkNorth} \{ t \} \boxed{m} \{ x \} \{ y \} \\
\text{rewrite} \text{lawNorth} \boxed{m} \{ t \} \{ x \} \{ y \} = \text{Stepl} \boxed{m}
\]

\text{rewrite lawNorth} makes the type of \text{walkNorth} fit \text{stepl}

Auxiliary law

\[
\text{lawNorth} : \{ m \ t : \mathbb{N} \} \{ x \ y : \mathbb{Z} \} \\
\rightarrow (\text{mkAV} (m +_\mathbb{N} t) \mathbb{N} x (y +_{\mathbb{Z}} (+ m))) \\
\equiv \text{stepAVI} m (\text{mkAV} t \mathbb{N} x y))
\]
Proof of auxiliary law (base case)

\[
\text{lawNorth} : \{ m \ t : \mathbb{N} \} \{ x \ y : \mathbb{Z} \} \\
\quad \rightarrow \ (\mkAV (m +^\mathbb{N} t) \mathbb{N} \times (y +^\mathbb{Z} (+ m))) \\
\quad \equiv \ \text{stepAVI} m (\mkAV t \mathbb{N} \times y))
\]

\[
\text{lawNorth} \{0\} \{t\} \{x\} \{y\} = \\
\begin{align*}
\mkAV (0 +^\mathbb{N} t) \mathbb{N} \times (y +^\mathbb{Z} (+ 0)) \\
\equiv \langle \text{refl} \rangle \\
\mkAV t \mathbb{N} \times (y +^\mathbb{Z} (+ 0)) \\
\equiv \langle \text{cong} (\lambda z \rightarrow \mkAV t \mathbb{N} \times z) (+^\mathbb{Z} - \text{identity} \{y\}) \rangle \\
\mkAV t \mathbb{N} \times y \\
\equiv \langle \text{refl} \rangle \\
\text{stepAVI} 0 (\mkAV t \mathbb{N} \times y)
\end{align*}
\]

Agda recognises inductive proofs

- Have shown above: \text{lawNorth} \{0\}
- Next we will show: \text{lawNorth} \{m\} \Rightarrow \text{lawNorth} \{\text{suc} \ m\}
Proof of auxiliary law (inductive case)

\[
\text{lawNorth} : \{ m \, t : \mathbb{N} \} \{ x \, y : \mathbb{Z} \} \\
\rightarrow (\text{mkAV} (\mathbb{N} m + t) \mathbb{N} x (y + Z (+ m))) \equiv \text{stepAVI} m (\text{mkAV} t \mathbb{N} x y)
\]

\[
\text{lawNorth} \{ \text{suc} \, m \} \{ t \} \{ x \} \{ y \} = \\
\text{begin} \\
\text{mkAV} ((\text{suc} \, m) + \mathbb{N} t) \mathbb{N} x (y + Z (+ (\text{suc} \, m))) \\
\equiv \langle \text{refl} \rangle \\
\text{mkAV} (\text{suc} \, (m + \mathbb{N} t)) \mathbb{N} x (y + Z (+ (\text{suc} \, m))) \\
\equiv \langle \text{cong} (\lambda z \rightarrow \text{mkAV} (\text{suc} \, (m + \mathbb{N} t)) \mathbb{N} x z) \\
(\text{z + sn = z + n + 1} \{ y \} \{ m \}) \rangle \\
\text{mkAV} (\text{suc} \, (m + \mathbb{N} t)) \mathbb{N} x ((y + Z (+ m)) + Z (+ 1)) \\
\equiv \langle \text{refl} \rangle \\
\text{stepAV} (\text{mkAV} (m + \mathbb{N} t) \mathbb{N} x (y + Z (+ m))) \\
\equiv \langle \text{cong} (\lambda x \rightarrow \text{stepAV} x) (\text{lawNorth} \{ m \} \{ t \} \{ x \} \{ y \}) \rangle \\
\text{stepAV} (\text{stepAVI} m (\text{mkAV} t \mathbb{N} x y)) \\
\equiv \langle \text{refl} \rangle \\
\text{stepAVI} (\text{suc} \, m) (\text{mkAV} t \mathbb{N} x y)
\]
Compositions of turns and walks

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

$$\text{turnWalkNorth} : \{ d : \text{Direction} \} (m : \mathbb{N}) \rightarrow \{ t : \mathbb{N} \} \{ x y : \mathbb{Z} \}$$
$$\rightarrow \text{ExLang} (\text{mkAV} \ t \ d \ x \ y)$$
$$\quad (\text{mkAV} (m +^N (3 +^N t)) N x (y +^Z (+ m))))$$
$$\text{turnWalkNorth} \ m = \text{TurnTo} \ N >>>= \text{walkNorth} \ m$$

Complete task (one of four cases)

$$\text{walkNorthWest} : \{ d : \text{Direction} \} (m \ n : \mathbb{N})$$
$$\rightarrow \{ t : \mathbb{N} \} \{ x y : \mathbb{Z} \}$$
$$\rightarrow \text{ExLang} (\text{mkAV} \ t \ d \ x \ y)$$
$$\quad (\text{mkAV} (t +^N m +^N n +^N 4) W (x -^Z (+ m)) (y +^Z (+ n))))$$
$$\text{walkNorthWest} \ {\_} \ m \ n \ {t} \ \text{rewrite LawMoveYX} \ {t} \ {m} \ {n}$$
$$= \text{turnWalkNorth} \ n >>>= \text{Turn} \ 1 >>>= \text{walkWest} \ m$$
Simple arithmetic equalities proven automatically

Example

\[
\text{LawMoveYX} : \{ t \, m \, n : \mathbb{N} \} \\
\quad \rightarrow (t +^\mathbb{N} m +^\mathbb{N} n +^\mathbb{N} 4 \equiv m +^\mathbb{N} (1 +^\mathbb{N} (n +^\mathbb{N} (3 +^\mathbb{N} t))))
\]

\[
\text{LawMoveYX} \{ t \} \{ m \} \{ n \}
\]

\[
= \text{solve} \ 3 \ (\lambda \ t \ m \ n \rightarrow t : + \ m : + \ n : + \ \text{con} \ 4 \\
\quad := m : + (\text{con} \ 1 : + (n : + (\text{con} \ 3 : + t)))) \ \text{refl} \ t \ m \ n
\]

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 $\times$. 
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

\[
\text{moveDelta} : (dx \ dy : \mathbb{Z}) \{ s : \text{AV} \} \\
\rightarrow \text{ExLang} \ s (\text{mkAV} (\text{AV.time} s +^N dx +^N dy +^N 4) \\
(\text{finalDir} \ dx \ dy) (\text{AV.px} s +^{\mathbb{Z}} dx) (\text{AV.py} s +^{\mathbb{Z}} dy))
\]

\[
\text{moveDelta} \ dx \ dy \ \{ \text{mkAV} \ t \ d \ xS \ yS \}
\]

\[
\text{with } dx \mid dy \\
... \mid + \ idx \mid + \ idiy \quad \text{rewrite } \text{abs1} \ \{xS\} \ \{idx\} \mid \text{abs1} \ \{yS\} \ \{idy\} \\
= \text{walkEastNorth} \ idx \ idiy
... \mid + \ idx \mid - [1 + idiy] \text{rewrite } \text{abs1} \ \{xS\} \ \{idx\} \mid \text{abs2} \ \{yS\} \ \{idy\} \\
= \text{walkSouthEast} \ idx \ (\text{suc idiy})
... \mid - [1 + idx] \mid + \ idiy \text{rewrite } \text{abs2} \ \{xS\} \ \{idx\} \mid \text{abs1} \ \{yS\} \ \{idy\} \\
= \text{walkNorthWest} \ (\text{suc idx}) \ idiy
... \mid - [1 + idx] \mid - [1 + idiy] \text{rewrite } \text{abs2} \ \{xS\} \ \{idx\} \mid \text{abs2} \ \{yS\} \ \{idy\} \\
= \text{walkWestSouth} \ (\text{suc idx}) \ (\text{suc idiy})
\]
Expressing implementation in terms of final coordinates

Mapping of the AV state, includes certificate

\[
\text{finalStateMoveTo} : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \text{AV} \rightarrow \text{AV}
\]

\[
\text{finalStateMoveTo } x\text{Fin } y\text{Fin } s
\]

with \((x\text{Fin} -^\mathbb{Z} \text{AV}.\text{px } s) \mid (y\text{Fin} -^\mathbb{Z} \text{AV}.\text{py } s)\)

\[
\ldots \mid dx \mid dy =
\]

\[
\text{mkAV} (\text{AV}.\text{time } s +^\mathbb{N} dx +^\mathbb{N} dy +^\mathbb{N} 4) (\text{finalDir } dx \text{ } dy) \]

\[
(\text{AV}.\text{px } s +^\mathbb{Z} dx) (\text{AV}.\text{py } s +^\mathbb{Z} dy)
\]

From distances to absolute coordinates

\[
\text{moveTo} : (x\text{Fin } y\text{Fin} : \mathbb{Z}) \{ s : \text{AV} \}
\rightarrow \text{ExLang } s (\text{finalStateMoveTo } x\text{Fin } y\text{Fin } s)
\]

\[
\text{moveTo } x\text{Fin } y\text{Fin} \{ \text{mkAV } t \text{ } d \text{Start } y\text{Start} \}
\]

\[
= \text{moveDelta} (x\text{Fin} -^\mathbb{Z} x\text{Start}) (y\text{Fin} -^\mathbb{Z} y\text{Start})
\]
StLang: embedded DSL at the strategic level

Language Definition

\[
\text{data StLang : AV } \rightarrow \ AV \rightarrow \ Set \ \text{where} \\
\text{MoveTo : } \{ s : AV \} (xFin \ yFin : \mathbb{Z}) \\
\rightarrow \ StLang s \ (\text{finalStateMoveTo xFin yFin s})
\]

Translation to ExLang

\[
\text{translate2 : } \{ s1 \ s2 : AV \} \rightarrow \ StLang s1 \ s2 \rightarrow \ ExLang s1 \ s2 \\
\text{translate2 (MoveTo } \{ s \} \ xFin \ yFin) = \ \text{moveTo xFin yFin } \{ s \}
\]

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.
**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