Adapting the Proofs-as-Programs to Imperative SML

Iman Poernomo
School of Computer Science & Software Engineering
Monash University
Australia
Overview

- Constructive Program Synthesis
- The Curry-Howard protocol
- Proofs-as-imperative-programs
- Conclusions
Constructive program synthesis

- Proofs-as-programs
- 30 years of research
  - Martin-Lof’s type theory
  - Nuprl
  - Calculus of Constructions
  - Hayashi’s PX
  - Tyugu’s SSL
- Two approaches
  - basic
  - transformative
The general idea

- Build a constructive proof of a specification
  - During a constructive proof, the user must provide information to derive the specification
  - This information is used to derive a correct program that satisfies the specification
- Constructive information is encoded via a type theoretic representation of proofs, according to the Curry-Howard isomorphism
The Curry-Howard isomorphism

First described by Curry and extended to first-order logic by Howard

Constructive logic corresponds to a type theory
- Proofs are encoded as terms of a lambda calculus with disjoint unions, dependent products and sums
- Proven formulae can be considered to be types of proof terms
- Proof rules correspond to type inference rules
- Valid typing defines a realizability relation between formulae and terms
- Normalization is lambda reduction
Example: ND rules as type inference rules

Introduction rule for universal quantification corresponds to a type inference rule for lambda abstraction with dependent product type

\[
\frac{\Delta \vdash A[y/x]}{\Delta \vdash \forall x : s \cdot A} \quad (\forall\text{-I})
\]

\[
\frac{\Delta \vdash p^A[y/x]}{\Delta \vdash \lambda x : s. p^{\forall x : s \cdot A}} \quad (\forall\text{-I})
\]

Elimination rule for universal quantification corresponds to a type inference rule for lambda application with instantiation of dependent product type

\[
\frac{\Delta \vdash \forall x : s \cdot A}{\Delta \vdash A[c/x]} \quad (\forall\text{-E})
\]

\[
\frac{\Delta \vdash p^{\forall x : s \cdot A}}{\Delta \vdash app(p, c)^A[c/x]} \quad (\forall\text{-E})
\]
Example: Encoding a proof as a lambda term

\[
\begin{align*}
\forall x : \text{int}. x &= x & \text{Axiom} \\
\text{input} \ast 2 &= \text{input} \ast 2 & \forall - E \\
\exists \text{output} : \text{int}. \text{output} &= \text{input} \ast 2 & \exists - I \\
\forall \text{input} : \text{int}. \exists \text{output} : \text{int}. \text{output} &= \text{input} \ast 2 & \forall - I
\end{align*}
\]

\[
\lambda \text{input} : \text{int}. (\text{input} \ast 2, \text{app}(\text{Axiom}, \text{input} \ast 2)) \forall \text{input} : \text{int}. \exists \text{output} : \text{int}. \text{output} = \text{input} \ast 2
\]
Normalization

- Proof normalization corresponds to lambda reduction

- Example: Redundant introduction and elimination of an implication is normalized thus

\[
\frac{x}{A \vdash A} \quad \frac{a}{B \vdash A \rightarrow B} \quad \frac{(\Rightarrow-I) \quad b}{A \vdash B} \quad \frac{(\Rightarrow-E)}{B}
\]

which reduces to

\[
app(\lambda x. a^{(A\rightarrow B)}, b^A) \triangleright a[b/X]^B
\]

With corresponding lambda reduction rule
Basic proofs-as-programs

Based directly on the Curry-Howard isomorphism

The lambda calculus of the logical type theory is considered as a programming language

Formulae = types = specifications of required realizing programs

Normalization = lambda term reduction = program evaluation
Program specifications

E.g., Input/Output specification of a program

\[ \forall input : \text{int.} \exists output : \text{int.} \quad output = input \times 2 \]

Given any integer input

There is an integer output

Require a program whose output is equal to the input \( \times 2 \)
Realizability

*This approach to proofs-as-programs takes realizability to correspond to correct typing*

Well-typed terms, as realizers, do two things:
- Yield a correct program with respect to a specification
- Contain a proof of correctness of the program
Example: proof as a program

Given a constructive proof of

$$\forall x : t \bullet \exists y : s \bullet A(x, y)$$

we can form a corresponding term $p$ in the logical type theory of the form

$$\lambda x : t. (g_1(x) : s, g_2(x) A(x, y)) \forall x : t \bullet \exists y : s \bullet A(x, y)$$
Example: proof as realizing program

The proof-term $p$ is considered to be a program such that,

- on every input $x : t$, the evaluation of $app(p \ x)$ terminates (by strong normalization)
- the first projection $\pi_1 app(p \ x)$ of the output is such that
  \[ A(x, \pi_1 app(p \ x)) \]
  holds – satisfying the derived specification formula.
- the second projection $\pi_2 app(p \ x)$ is a proof of this.
Two problems with the basic approach (1)

Usability of proofs terms considered as programs:

- Commonly used programming languages (such as SML, Haskell, C++ with STL or Java) do not have dependent sum and product type constructors -- essential for defining types that correspond to AE specifications.
- Execution of realizing terms requires custom-built compiler or interpreter for the type theory.
- For larger scale practical programming problems, this can result in inefficient code that is not readily interoperable or maintainable.
Two problems with the basic approach (2)

Efficiency of proof terms considered as programs:

- A further problem is that lambda terms corresponding to constructive proofs often encode *computationally irrelevant*, non-constructive information.
- Such irrelevant information is introduced when proving Harrop formulae.
Transformative approaches

Based on the Curry-Howard isomorphism, but with additional program transformation

Two type theories involved:
- constructive logical type theory (LTT) for representing proofs
- a commonly used functional programming language (e.g., SML)

Specification of programs given by a different modified realizability relation between formulae and programs

Proof terms in the LTT are transformed into programs that satisfy the proved specification formulae (as modified realizers)
Modified realizability

Our modified realizers are simply typed SML programs – they are correct with respect to a specification, but do not carry proofs of correctness with them.

Formally, a program p is a modified realizer of a formula A when p can be used in place of the Skolem function for the Skolem form of A:

That is, when

$$\text{Sk}(A)[p/f_A]$$

is provable
Skolem form (1)

<table>
<thead>
<tr>
<th>$F$</th>
<th>$\text{xsort}(F')$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(\bar{a})$</td>
<td>$\text{Unit}$</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>$\begin{cases} \text{xsort}(A) &amp; \text{if not Harrop}(B) \ \text{xsort}(B) &amp; \text{if not Harrop}(A) \ \text{xsort}(A) \ast \text{xsort}(B) &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>$\text{xsort}(A) \mid \text{xsort}(B)$</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>$\begin{cases} \text{xsort}(B) &amp; \text{if not Harrop}(B) \ \text{xsort}(A) \rightarrow \text{xsort}(B) &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$\forall x : S.A$</td>
<td>$s \rightarrow \text{xsort}(A)$</td>
</tr>
<tr>
<td>$\exists x : S.A$</td>
<td>$\begin{cases} s &amp; \text{if Harrop}(A) \ s \ast \text{xsort}(A) &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\text{Unit}$</td>
</tr>
</tbody>
</table>
Skolem form (2)

(Skolem form and Skolem functions). Given a closed formula $A$, we define the Skolemization of $A$ to be the Harrop formula $Sk(A) = Sk'(A, \emptyset)$, where $Sk'(A, AV)$ is defined as follows. A unique function letter $f_A$ (of sort corresponding to SML type $\text{xsort}(A)$) called the Skolem function, is associated with each $A$. $AV$ represents a list of variables which will be arguments of $f_A$.

If $A$ is Harrop, then $Sk'(A, AV) \equiv A$.

If $A \equiv B \lor C$, then

$$Sk'(A, AV) = (\forall x : \text{xsort}(A). f_A(AV) = \text{Inl}(x) \rightarrow Sk'(B, AV)[x/f_B])$$

$$\land (\forall y : \text{xsort}(B). f_B(AV) = \text{Inr}(y) \rightarrow Sk(C, AV)[y/f_C])$$

If $A \equiv B \land C$, then

$$Sk'(A, AV) = Sk'(B, AV)[\text{fst}(f_A)/f_B] \land Sk'(C, AV)[\text{snd}(f_A)/f_C]$$

If $A \equiv B \rightarrow C$, then

1) if $B$ is Harrop, $Sk'(A, AV) = B \rightarrow Sk'(C, AV)[f_A/f_C]$.

2) if $B$ is not Harrop and $C$ is not Harrop,

$$Sk'(A, AV) = \forall x : s.(Sk'(B, AV)[x/f_B] \rightarrow Sk'(C, AV)[(f_Ax)/f_C])$$

If $A \equiv \exists y : s.P$, then

1) when $P$ is Harrop, $Sk'(A, AV) = Sk'(P, AV)[f_A(AV)/y]$

2) when $P$ is not Harrop, $Sk'(A, AV) = Sk'(P, AV)[\text{fst}(f_A(AV))/y][\text{snd}(f_A(AV))/f_P]$.

If $A \equiv \forall x : s.P$, then $Sk'(A, AV) = \forall x. Sk'(P, AV)[(f_Ax)/f_P]$. 
Skolem form

The Skolem form of
\[ \text{dbConnected} \lor \neg \text{dbConnected} \]
is equivalent to
\[ f = \text{inl}() \rightarrow \text{dbConnected} \land \]
\[ f = \text{inr}() \rightarrow \neg \text{dbConnected} \]

A modified realizer is a function \( f \) that says if there is a connection to the db or not

Modified realizer does not prove this fact – it exists in a separate language
Transforming proofs into programs

We use an *extraction map* to transform a proof-term into a program — the program is correct in the sense that it is a modified realizer of the formula.

The map does the following:

- Normalizes the proof, simplifying the proof-term by applying reduction rules of the lambda calculus.
- Removes “useless” non-constructive information from the proof term.

Normalization is *not* evaluation — the resulting program evaluates according to the semantics of the target programming language.
Extraction

\[ \lambda \text{input} : \text{int.} \ (\text{input} \times 2, \ \text{app}(\text{Axiom}, \ \text{input} \times 2)) \]

Constructive information

"Useless" non-constructive information

Normalize, then extract SML program from proof-term
This program is a modified realizer of the derived formula

fun input : int -> input * 2

In C/Java, this program might be written
function (int input) {return input*2;}
The Curry-Howard protocol
The problem

How should this transformative style of program synthesis be generalized over arbitrary logics and programming paradigms?

Logics: Linear logic, Modal logics, Hoare logic, Hennessy-Milner proof systems, etc.

Programming paradigms: Imperative, structured, object-oriented, parallelism, etc.
The protocol

- We identify a ontology of roles and relations that abstract the entities and relations of transformative proofs-as-programs.
- We claim that good, useful adaptations of proofs-as-programs are achieved by identifying these roles and relations in new contexts.
## Roles in the protocol

<table>
<thead>
<tr>
<th>Rôle</th>
<th>Domain in the Curry-Howard protocol</th>
<th>Properties of domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>Natural deduction system</td>
<td>A formal system that defines a logical calculus.</td>
</tr>
<tr>
<td>Proofs</td>
<td>Logical type theory (<em>LTT</em>)</td>
<td>A type theory that enables encoding of proofs in the logical calculus, according to the Curry-Howard isomorphism: types denote statements, terms denotes proofs and type inference corresponds to logical inference.</td>
</tr>
<tr>
<td>Programming language</td>
<td>Computational type theory (<em>CTT</em>)</td>
<td>A type theory for the programming language, equipped with an operational semantics.</td>
</tr>
</tbody>
</table>
The protocol requires

- a realizability relation realizes between formulae of programs of the $CTT$ and the $LTT$
- a program extraction and optimization map extract from proofs of the $LTT$ to programs of the $CTT$ such that, given a proof of a specification, the extracted program should realize the specification.
Relations between the roles (2)

The concepts of the protocol are related according to the following diagram (with $t^S$ denoting a proof-term $t$ of $S$ in the logical type theory):

$LTT$: $t_1^S$ \(\xrightarrow{\text{normalizes to}}\) $t_2^S$

extract

$CTT$: $p_1$ realizes $S$ \(\xrightarrow{\text{computationally equivalent}}\) $p_2$ realizes $S$
A useful generalizing framework?

It can be seen that functional/constructive logic transformative proofs-as-programs conforms to this framework.

The usefulness of the protocol as a generalization can only be shown through applications to new logics and programming languages.
Application: Proofs-as-imperative programs
Problem

- Proofs-as-programs approach generates *functional* programs – programs that do not use state

- Imperative programs use state: e.g.
  \[
  s := s + 10
  \]
  is an imperative program that increments the state \( s \) by 10

- Want to be able to adapt proofs-as-programs to *Hoare logic* for synthesis of correct *return values*
Hoare Logic

Hoare Logic allows us to build imperative programs that are correct for a specification.

Specifications take the form of pre- and post-conditions.

\[ \text{correct}(\text{pin}) = true \rightarrow \text{db.status'} = \text{connected} \]

Pre-condition: assume the \textit{state}_{\text{pin}} is initially acceptable by the program.

Post-condition: the final status of the database is that it must be connected.
Hoare logic

Rules of the logic allow us to (simultaneously) deduce new truths about programs from old and to form new programs from old.

If \( b \) is true, after running \( l_1 \), \( C \) is true.

If \( b \) is true, after running \( l_2 \), \( C \) is true.

If \( b \) is true, after running \( \text{if } b \text{ then } l_1 \text{ else } l_2 \), \( C \) is true.

\[
\begin{align*}
l_1 \cdot b &= \text{true} \rightarrow C \\
l_2 \cdot b &= \text{false} \rightarrow C \\
\text{if } b \text{ then } l_1 \text{ else } l_2 \cdot C
\end{align*}
\]

after running if \( b \) then \( l_1 \) else \( l_2 \), \( C \) is true.

(because when \( b \) is true, \( l_1 \) is executed and \( C \) is true – similarly, when \( b \) is false, \( l_2 \) is executed and \( C \) is still true)
Correct synthesis of return values

The presence of side-effects is what distinguishes the imperative paradigm from the functional one. However, side-effect-free return values are also important in imperative programs because they enable access to data, obtaining views of state. For example, the SML program

\[ s := 10; \ !s \times 2 \]

involves a side-effect producing assignment statement, \( s := 10 \) followed by a side-effect-free term \( !s \times 2 \), which will evaluate to a return value.

We want to adapt the constructive approaches to specify and extract imperative programs with return values.
How to adapt constructive results?

- Follow the Curry-Howard protocol
- Design a constructive version of Hoare logic (natural deduction system)
  1) Design augmented lambda terms to encode proofs in Hoare logic to form a Logical Type Theory
  2) Consider subset of imperative SML programs with return values as the Computational Type theory
  3) Define a notion of realizability between LTT and CTT
  4) Define extraction map to transform proofs of specifications into programs that realize the specifications
Constructive Hoare logic

We consider a constructive version of Hoare logic.

5 core rules parametrized over the usual intuitionistic rules: e.g.,

\[
\frac{\Delta \vdash_{\text{int}} P[a/y]}{\Delta \vdash_{\text{int}} \exists y : s \bullet P} \quad (\exists\text{-I})
\]

\[
\frac{\Delta_1 \vdash_{\text{int}} \exists y : s \bullet P \quad \Delta_2, P[x/y] \vdash_{\text{int}} C}{\Delta_1, \Delta_2 \vdash_{\text{int}} C} \quad (\exists\text{-E})
\]

where \( x \) not occur free in \( C \)

\[
\frac{\Delta \vdash_{\text{int}} A \quad \Delta' \vdash_{\text{int}} B}{\Delta, \Delta' \vdash_{\text{int}} (A \land B)} \quad (\land\text{-I})
\]
Core rules

\[ \vdash_{\text{IHL}} s := v \leadsto s_f = \text{tologic}_{i}(v) \]  
(assign)

where \( s \in \text{StateRef} \).

\[ \vdash_{\text{IHL}} p \leadsto (\text{tologic}_{i}(b) = \text{true} \rightarrow C) \quad \vdash_{\text{IHL}} q \leadsto (\text{tologic}_{i}(b) = \text{false} \rightarrow C) \]  
(ite)

\[ \vdash_{\text{IHL}} \quad \text{if } b \text{ then } p \text{ else } q \leadsto C \]

\[ \vdash_{\text{IHL}} p \leadsto (A[\bar{s}_i/\bar{v}] \Rightarrow B[\bar{s}_f/\bar{v}]) \quad \vdash_{\text{IHL}} q \leadsto (B[\bar{s}_i/\bar{v}] \Rightarrow C[\bar{s}_f/\bar{v}]) \]  
(seq)

\[ \vdash_{\text{IHL}} p; q \leadsto (A[\bar{s}_i/\bar{v}] \Rightarrow C[\bar{s}_f/\bar{v}]) \]

where \( A \) and \( B \) are free of state identifiers.

\[ \vdash_{\text{IHL}} q \leadsto (\text{tologic}_{i}(b) = \text{true} \land A[\bar{s}_i/\bar{v}] \Rightarrow A[\bar{s}_f/\bar{v}] \]  
(loop)

\[ \vdash_{\text{IHL}} \quad \text{while } b \text{ do } q \leadsto A[\bar{s}_i/\bar{v}] \Rightarrow (A[\bar{s}_f/\bar{v}] \land \text{tologic}_{f}(b) = \text{false}) \]

where \( A \) is free of state identifiers.

\[ \vdash_{\text{IHL}} p \leadsto P \quad \vdash_{\text{Int}} (P \Rightarrow A) \]  
(cons)

\[ \vdash_{\text{IHL}} p \leadsto A \]
Logical type theory

- Program/formulae pairs used in the Hoare logic are treated as types in a corresponding type theory.
- An augmented lambda calculus is used with extra constructors corresponding to applications of the core rules.
- A typing judgment corresponds to a proof derivation.
Core type inference rules

\( \vdash_{LTT(IHL)} \text{assign}(s, v)^{s:=v \diamond s_f \Rightarrow \text{tologic}_i(v)} \) (assign)

where \( s \in \text{StateRef} \)

\( \vdash_{LTT(IHL)} q_1^{\top_1 \diamond (\text{tologic}_i(b) = \text{true} \Rightarrow C)} \) \( \vdash_{LTT(IHL)} q_2^{\top_2 \diamond (\text{tologic}_i(b) = \text{false} \Rightarrow C)} \)

\( \vdash_{LTT(IHL)} \text{ite}(q_1, q_2) \) \text{if } b \text{ then } l_1 \text{ else } l_2 \diamond C \) (ite)

\( \vdash_{LTT(IHL)} p^{\top_1 \diamond (A[\bar{s}_i/\bar{v}] \Rightarrow B[\bar{s}_f/\bar{v}])} \) \( \vdash_{LTT(IHL)} q^{\top_2 \diamond (B[\bar{s}_i/\bar{v}] \Rightarrow C[\bar{s}_f/\bar{v}])} \)

\( \vdash_{LTT(IHL)} \text{seq}(p^{\top_1 \diamond (A[\bar{s}_i/\bar{v}] \Rightarrow B[\bar{s}_f/\bar{v}])}, q^{\top_2 \diamond (B[\bar{s}_i/\bar{v}] \Rightarrow C[\bar{s}_f/\bar{v}])})^{w_1;w_2 \diamond (A[\bar{s}_i/\bar{v}] \Rightarrow C[\bar{s}_f/\bar{v}])} \) (seq)

where \( A \) and \( B \) do not contain any state identifiers

\( \vdash_{LTT(IHL)} q^{\top \diamond (\text{tologic}_i(b) = \text{true} \wedge A[\bar{s}_i/\bar{v}] \Rightarrow A[\bar{s}_f/\bar{v}])} \) (loop)

\( \vdash_{LTT(IHL)} \text{wd}(q)^{\text{while } b \text{ do } w \diamond (A[\bar{s}_i/\bar{v}] \Rightarrow (A[\bar{s}_f/\bar{v}] \wedge \text{tologic}_f(b) = \text{false}))} \)

where \( A \) does not contain any state identifiers

\( \vdash_{LTT(IHL)} q_1^{p \diamond P} \) \( \vdash_{LTT(IHL)} q_2^{(P \Rightarrow A)} \)

\( \vdash_{LTT(IHL)} \text{cons}(q_1, q_2)^{p \diamond A} \) (cons)
A program $p$ is a *return value realizer* of formula $A$ when all return values of $p$ are modified realizers of $A$

- I.e., for each evaluation of the program $p$, $Sk(A)'[r/f_A]$ is true, where $r$ is a representation of the return value and $Sk(A)'$ is $Sk(A)$ with initial and final values of the run substituted for corresponding state references
Example

\[ s := s \times 3; !s \times 2 \]

is a return value realizer of

\[ s := s \times 3 \bullet s_f > s_i \land (\exists x : \text{int} \bullet Even(x) \land x > s_i) \]

because, for every evaluation

\[ \langle s := s \times 3; !s \times 2, \sigma \rangle \triangleright \langle \text{answer}, \sigma' \rangle \]

it is true that

\[ \sigma'(s) > \sigma(s) \land (Even(\text{answer}) \land answer > \sigma(s)) \]
Realizability

An SML program $p$ with return values realizes a specification in the Hoare logic $q \triangleleft A$ when

- $p$ has the required side-effects that are specified by $A$, just like $q$ does
- $p$ is a return value realizer of $A$ – all return values form modified realizers of $A$
Extraction

- We extend the extraction map of ordinary proofs-as-programs to our case, to extract return-value realizers from proofs in the Hoare logic.
- The map is from our LTT to our CTT (SML with return values).
Example cases:

<table>
<thead>
<tr>
<th>$\overrightarrow{wT}$</th>
<th>extract_{IHL}(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any proof-term $t$ with $H(T)$</td>
<td>$w$</td>
</tr>
<tr>
<td>$\text{wd}(u)\text{while } b \text{ do } l \circ P$ where $P$ is $A[\bar{s}_i / \bar{u}] \Rightarrow (A[\bar{s}_f / \bar{u}] \land \text{tologic}_f(b) = false)$</td>
<td>$rv_1 := \text{fn } x : \text{etype}(A) =&gt; x;$ while $b$ do $rv_2 := \text{extract}_{IHL}(q);$ $rv_1 := (\text{fn } x_2 :: x_1 =&gt; \text{fn } x : \text{etype}(A) =&gt; x_2(x_1 x))$ !$rv_2$ !$rv_1 ;$ !$rv_1 ;$</td>
</tr>
<tr>
<td>$\text{ite}(q_1, q_2) \text{if } b \text{ then } l_1 \text{ else } l_2 \circ C$</td>
<td>if $b$ then extract($q_1$) else extract($q_2$)</td>
</tr>
</tbody>
</table>

- Extraction over Harrop terms is trivial: the program of the program/formula pair is used (only side-effects are important)
- Extraction over (ite) case results in conditional statement with alternate return values
- Extraction over while-loop term is complicated in the non-Harrop case, due to iterated application of the functional return value for the looped term
Example cases

<table>
<thead>
<tr>
<th>$t^w{\Theta}$</th>
<th>$\text{extract}_{\text{IHL}}(t)$</th>
</tr>
</thead>
</table>
| seq($p^{w_1\circ P}, q^{w_2\circ Q})^{w_3\circ A[\bar{s}_I/\bar{v}]} \Rightarrow C[\bar{s}_f/\bar{v}]$) | \[
\begin{align*}
\text{rv}_p & := \text{extract}(p); \\
\text{rv}_q & := \text{extract}(q); \\
(\text{fn } x_p & => \text{fn } x_q => \\
\text{fn } x & : \text{etype}(A) => x_q (x_p x)) \\
!\text{rv}_p \! & \! \text{rv}_q
\end{align*}
\]
not $H(A)$
not $H(B)$
and
not $H(C)$ |
| $P$ is $A[\bar{s}_I/\bar{v}] \Rightarrow B[\bar{s}_f/\bar{v}]$ | \[
\begin{align*}
\text{rv}_p & := \text{extract}(p); \\
\text{rv}_q & := \text{extract}(q); \\
\text{rv}_q \! \text{rv}_p
\end{align*}
\]
$H(A)$
not $H(B)$
and not $H(C)$ |
| $Q$ is $B[\bar{s}_I/\bar{v}] \Rightarrow C[\bar{s}_f/\bar{v}]$ | \[
\begin{align*}
\text{rv}_q & := \text{extract}(q); \\
!\text{rv}_q
\end{align*}
\]
$H(A)$
$H(B)$
and
not $H(C)$ |
| $w;$ | \[
\begin{align*}
\text{rv}_q & := \text{extract}(q); \\
!\text{rv}_q
\end{align*}
\]
not $H(A)$
and $H(B)$
and
not $H(C)$ |
Example extraction case: (seq) rule

\[
\vdash_{LTT(IHL)} p_{w_1}^\diamond (A[s_i/v] \Rightarrow B[s_f/v]) \quad \vdash_{LTT(IHL)} q_{w_2}^\diamond (B[s_i/v] \Rightarrow C[s_f/v])
\]

\[
\vdash_{LTT(IHL)} \text{seq}(p_{w_1}^\diamond (A[s_i/v] \Rightarrow B[s_f/v]), q_{w_2}^\diamond (B[s_i/v] \Rightarrow C[s_f/v]))_{w_1; w_2}^\diamond (A[s_i/v] \Rightarrow C[s_f/v])
\]

If all formulae used are non-Harrop, then return values for sub-terms are functions – the output of the first is passed as input to the second to give the required return value overall.

\[
\begin{align*}
r_v_p & := \text{extract}(p); \\
r_v_q & := \text{extract}(q); \\
(\text{fn } x_p & \Rightarrow \text{fn } x_q \Rightarrow \\
\text{fn } x & : \text{etype}(A) & \Rightarrow x_q (x_p x)) \\
!r_v_p & !r_v_q
\end{align*}
\]
ATM Example: specification of return values

Example specification: ATM Bank machine
- user puts in card, enters PIN number (denoted by state pin)
- irrespective of whether the PIN is incorrect or not, we require an appropriate response send to the ATM screen

Return value specified as required return value realizer

\[ InMachine(pin) \rightarrow \exists x : \text{string.appropriateResponse}(x) \]

Pre-condition – PIN has been entered into machine

Require a return value x that is an appropriate response
ATM example

We use the following axioms that define the domain knowledge about the behaviour of the ATM machine.

This axiom says that, for any program, if the card is destroyed, it is appropriate to tell the user about it:

\[
\begin{align*}
\text{any} \cdot \text{card_destroyed}' &= \text{true} \rightarrow \text{appropriateResponse} \left( \text{"Your card has been destroyed"} \right) \\
\text{any} \cdot \text{db.status}' &= \text{connected} \rightarrow \text{appropriateResponse} \left( \begin{array}{l}
\text{Menu} \\
(1) \text{Withdraw} \\
(2) \text{Deposit} \\
(3) \text{Balance}
\end{array} \right) \\
\text{correct_pin_action} \cdot \left( \text{InMachine}(\text{pin}) \rightarrow \text{correct}(\text{pin}) = \text{true} \rightarrow \text{db.status}' = \text{connected} \right) \\
\text{incorrect_pin_action} \cdot \left( \text{InMachine}(\text{pin}) \rightarrow \text{correct}(\text{pin}) = \text{false} \rightarrow \text{card_destroyed}' = \text{true} \right)
\end{align*}
\]

This program destroys the card if the PIN is incorrect.
Example: Extended lambda terms

- We define new lambda terms to represent proofs in a constructive version of Hoare logic.
- We can use Hoare logic to prove specification

\[ \text{InMachine}(\text{pin}) \rightarrow \exists x : \text{string.appropriateResponse}(x) \]

we obtain a proof of

\[
\left( \begin{array}{l}
\text{if } \text{correct}(\text{pin}) \\
\text{then } \text{correct\_pin\_action} \\
\text{else } \text{incorrect\_pin\_action}
\end{array} \right) \quad \text{InMachine}(\text{pin}) \rightarrow \exists x : \text{string.appropriateResponse}(x)
\]

with proof-term …
\[ \text{itecase}(\lambda Y. Z. \text{app}(< (\text{Axiom}(\text{any } \bullet \text{db.status'} = \text{connected} \rightarrow \\
\text{appropriateResponse}(\text{"Menu (1) Withdraw (2) Deposit (3) Balance"}), \text{correct_pin_action}), \text{app}(\text{app}(\text{Axiom}(\text{correct_pin_action } \bullet \text{InMachine}(\text{pin}) \\
\rightarrow \text{correct}(\text{pin}) = \text{true} \rightarrow \text{db.status'} = \text{connected}), \\
Z^{\text{correct_pin_action } \bullet \text{InMachine}(\text{pin})}, \\
Y^{\text{correct_pin_action } \bullet \text{correct}(\text{pin}) = \text{true}))), \\
\lambda Y. Z. \text{app}(< (\text{Axiom}(\text{any } \bullet \text{card_destroyed'} = \text{true} \rightarrow \\
\text{appropriateResponse}(\text{"Your card has been destroyed"})), \text{incorrect_pin_action}), \text{app}(\text{app}(\text{Axiom}(\text{incorrect_pin_action } \bullet \\
\text{InMachine}(\text{pin}) \rightarrow \text{correct}(\text{pin}) = \text{false} \rightarrow \\
\text{card_destroyed'} = \text{true}), Z^{\text{incorrect_pin_action } \bullet \text{InMachine}(\text{pin})}, \\
Y^{\text{incorrect_pin_action } \bullet \text{correct}(\text{pin}) = \text{true}))))) \]
Extraction: Example

The extraction map removes non-constructive elements of proof-terms, to give imperative SML program.

The resulting SML programs have return values that are realizers of the specifications proved.

Applying the map to the lambda term for our proof gives the correct program:

```sml
if correct(pin) then
  correct_pin_action; "Menu (1) Withdraw (2) Deposit (3) Balance"
else
  incorrect_pin_action; "Your card has been destroyed"
```
Application: Design-by-contract

Design-by-contract is a well established method of software development (see Meyer)

- When a program is developed, it must be accompanied with two boolean-valued functions, called assertions. These form the so-called contract of the program.
- The boolean functions are called the pre- and post-condition assertions.
- Programs are tested at run-time by evaluating the values of the assertions in a dedicated test suite.
- If the pre-condition assertion evaluates to true before the program is executed, and the post-condition evaluates to false, then the program has an error and the designer is alerted by the test suite.

Complex code requires complex, sometimes functional assertions
Contracts as return values

Assertions are a special kind of return value

We can simulate assertions in SML as return values of disjoint union type

- The post-condition assertion for a program is taken to be true if it is of the form \( \text{Inl}(a) \) and false if it is of the form \( \text{Inr}(b) \).
- For example, assume a program that \( \text{s} := \text{s} \times 2; \text{Even}(\text{s}) \) consists of some imperative code \( \text{s} := \text{s} \times 2 \) with return values arising from \( \text{Even}(\text{s}) \), of type \( \text{Unit} | \text{Unit} \).
- If the state value \( \text{s} \) is even, then the return value of the program is \( \text{Inl}() \), and \( \text{Inr}() \) otherwise.
- Programs with assertions can then be evaluated within a testing tool that will generate appropriate error reports whenever a post-condition is violated.
Specification and Synthesis of contracts

- The specification of a required post-condition assertion is given by a disjunction of the form $(A \lor \neg A)$.
- The disjunction specifies the required post-condition as a return value realizer, of type $\text{xsort}(A \lor \neg A) = (\text{xsort}(A) | \text{xsort}(\neg A))$.
- So, given a proof of the form

$$\vdash \text{body} \cdot A \lor \neg A$$

we can extract a program that has equivalent side-effect to body, but with an accompanying post-condition assertion that correctly expresses the expectations of the designer, as stated as the Skolem function for the disjunction specification.
Improving programs built with faults

By utilizing our synthesis methods, we can analyze and improve programs that may be built from faulty sub-programs, complementing the ideas of design-by-contract.

The designer uses the rules of our Hoare logic to make proofs of the above contractual form.

As usual with the logic, only true, known properties about given programs are used.

Faulty programs may, however, be used and reasoned about.
Improving programs built with faults

- Instead of using formulae that assert the correctness of programs, we use disjunctive statements stating that the program may or may not be faulty.

- Because disjunctive statements correspond to post-condition assertions, our synthesis enables the automatic construction of a program with accompanying assertions, through the reasoning about the potential known faults of subprograms.
Example

- Consider a program designed to connect to a database, \texttt{connectDB}. The program is intended to always result in a successful connection, specified by formula $connected_f = true$.
- The program has a fault, and sometimes results in an unsuccessful connection.
- This situation is described truthfully by the disjunction

\[ \vdash \texttt{connectDB} \bullet connected_f = true \lor \neg connected_f = true \]

- This property is true of the program, and so the program may be used within Hoare logic to develop a larger program/formula theorem, without jeopardizing the truth of the final result.
Example

By our synthesis methods, proofs that involve this theorem can also be transformed into programs that use a version of `connectDB` with an accompanying assertion.

The assertion returns `inl()` when a connection has been made and `inr()` otherwise.

Larger programs built using our methods can make use of this assertion to deal with the possibility of a faulty connection.
Conclusions and Future Work

The Curry-Howard protocol is a very general semi-formal framework for adapting proofs-as-programs.

Applications show the framework can be used to produce useful adaptations.

Crossley, Poernomo and Wirsing applied the protocol to synthesis of structured SML code from proofs about structured CASL specifications.

Future work:
- Investigate further adaptations – currently Hennessy-Milner proof systems – synthesis of component data-views.
- Investigate ways of formalizing the protocol itself for the purposes of automation.
Further reading and Questions


Iman Poernomo, John N. Crossley, Martin Wirsing, *Programs, Proofs and Parameterized Specifications*, in Maura Cerioli, Gianna Reggio (Eds.), *WADT 2001*, LNCS 2267


All papers available on request from the author at email: ihp@csse.monash.edu.au