Development task

Given precondition φ and postcondition ψ

develop a program $S^?$ such that

$$\boxed{[\varphi]\, S^?\, [\psi]}$$

Approach #1

Given precondition φ and postcondition ψ, develop a program $S^?$ such that

$$[\varphi]\, S^?\, [\psi]$$

- first try to have a good idea (or draw on what you learnt at the university) and

  write out a program S

- then verify that $\models [\varphi]\, S\, [\psi]$ — or rather, prove $\vdash [\varphi]\, S\, [\psi]$
- Once this is done, the task is completed, and you can cash your fees.

That is, if you succeed...

No hint what to do when one fails!

No hint even whether:

- the program developed is incorrect, or
- proving skills/tools employed are not sufficiently strong

Better approach

Develop the program gradually,

making sure at each step that

correctness is guaranteed if subsequent steps are correct

Example

Develop $S^?$ so that

$$[n \geq 0]\; S^?\; [rt^2 \leq n \wedge n < (rt+1)^2]$$

(and n is not modified in $S^?$)

Step 1

We can decide to proceed via:

$$
\begin{array}{l}
[n \geq 0] \\
\quad S_1^? \; [n \geq 0 \wedge rt = 0 \wedge sqr = 1] \; S_2^? \\
[rt^2 \leq n \wedge n < (rt+1)^2]
\end{array}
$$

That is, we want to:

- first, develop $S_1^?$ so that $[n \geq 0] \; S_1^? \; [n \geq 0 \wedge rt = 0 \wedge sqr = 1]$
- independently, develop $S_2^?$ so that
$$
\begin{array}{l}
[n \geq 0 \wedge rt = 0 \wedge sqr = 1] \\
\quad S_2^? \\
[rt^2 \leq n \wedge n < (rt+1)^2]
\end{array}
$$
- then, put $S^? \equiv S_1^?; S_2^?$

Correctness follows by the assertion $[n \geq 0 \wedge rt = 0 \wedge sqr = 1]$

Step 2

Develop $S_1^?$ so that

$$[n \geq 0]\ S_1^?\ [n \geq 0 \wedge rt = 0 \wedge sqr = 1]$$

(and n is not modified in $S_1^?$)

EASY!

Just put $S_1^?$ to be

$$rt := 0;\ sqr := 1$$

Verifies immediately!

Step 3

Develop $S_2^?$ so that

$$[n \geq 0 \wedge rt = 0 \wedge sqr = 1]\ S_2^?\ [rt^2 \leq n \wedge n < (rt+1)^2]$$

(and n is not modified in $S_2^?$)

Design decision: proceed via

$$
\begin{array}{l}
[n \geq 0 \wedge rt = 0 \wedge sqr = 1] \\
\quad \mathbf{while}\ [\varphi^?]\ b^?\ \mathbf{do}\ \mathbf{decr}\ e^?\ \mathbf{in}\ W^?\ \mathbf{wrt}\ \succ^? \\
\qquad\qquad S_3^? \\
[rt^2 \leq n \wedge n < (rt+1)^2]
\end{array}
$$

That is:

Choose $W^?$ and well-founded $\succ^? \subseteq W^? \times W^?$, as well as the invariant $\varphi^?$, boolean expression $b^?$, expression $e^?$, and develop $S_3^?$ so that:

- $\boxed{(n \geq 0 \wedge rt = 0 \wedge sqr = 1) \implies \varphi^?}$
- $\boxed{(\varphi^? \wedge \neg b^?) \implies (rt^2 \leq n \wedge n < (rt+1)^2)}$
- $\boxed{[\varphi^? \wedge b^?]\, S_3^?\, [\varphi^?]}$
- $\boxed{\mathcal{E}[\![e^?]\!]\, s \succ^? \mathcal{E}[\![e^?]\!]\, (\mathcal{S}[\![S_3^?]\!]\, s)}$ for all states $s \in \{\varphi^? \wedge b^?\}$

Eureka!

Choose:

$$\boxed{\varphi^? \equiv (sqr = (rt+1)^2 \wedge rt^2 \leq n)}$$

Then:

- The first requirement follows.
- Put $\boxed{b^? \equiv (sqr \leq n)}$ — and then the second requirement follows.
- Choose:
  - $\boxed{W^? = \mathbf{Nat}}$ with well-founded $\boxed{\succ^? = >}$
  - $\boxed{e^? = n - rt}$

Then proceed with further development...

Step 4

Develop $S_3^?$ so that

$$[sqr = (rt+1)^2 \wedge rt^2 \leq n \wedge sqr \leq n]$$
$$S_3^?$$
$$[sqr = (rt+1)^2 \wedge rt^2 \leq n]$$

(and n is not modified in $S_3^?$)

Design decision: proceed via

$$[sqr = (rt+1)^2 \leq n \wedge sqr \leq n]$$
$$S_4^?$$
$$[sqr = rt^2 \leq n]$$
$$S_5^?$$
$$[sqr = (rt+1)^2 \wedge rt^2 \leq n]$$

Termination

Let's not forget:

termination conditions are a part of the requirements

For $S_3^?$ we also require:

- $\boxed{\mathcal{E}[\![n - rt]\!]\, s > \mathcal{E}[\![n - rt]\!]\, (\mathcal{S}[\![S_3^?]\!]\, s)}$ for $s \in \{sqr = (rt+1)^2 \leq n \wedge rt^2 \leq n\}$

To ensure this, we choose to impose:

- $\boxed{\mathcal{E}[\![n - rt]\!]\, s > \mathcal{E}[\![n - rt]\!]\, (\mathcal{S}[\![S_4^?]\!]\, s)}$ for $s \in \{sqr = (rt+1)^2 \leq n \wedge rt^2 \leq n\}$

- $\boxed{\mathcal{E}[\![n - rt]\!]\, s \geq \mathcal{E}[\![n - rt]\!]\, (\mathcal{S}[\![S_5^?]\!]\, s)}$ for $s \in \{sqr = rt^2 \leq n\}$

Steps 5 & 6

Put $S_4^?$ to be

$$\boxed{rt := rt + 1}$$

and $S_5^?$ to be

$$\boxed{sqr := sqr + 2 * rt + 1}$$

Verifies immediately!

(including termination conditions)

EASY!

Putting all the steps together

$[n \geq 0]$

$\quad rt := 0;\ sqr := 1$

$\quad [n \geq 0 \wedge rt = 0 \wedge sqr = 1]$

$\quad \textbf{while}\ [sqr = (rt+1)^2 \wedge rt^2 \leq n]\ sqr \leq n\ \textbf{do}\ \textbf{decr}\ n - rt\ \textbf{in}\ \textbf{Nat}\ \textbf{wrt}\ >$

$\qquad\quad (rt := rt + 1\ [sqr = rt^2 \leq n]\ sqr := sqr + 2 * rt + 1)$

$[rt^2 \leq n \wedge n < (rt+1)^2]$

Correctness by construction!!!

. . . with proofs ready for use!

Specifications and formal program development "in-the-large"

What are specifications for?

For the system user: specification captures the properties of the system the user can rely on.

For the system developer: specification captures all the requirements the system must fulfil.

Specification engineering

Specification development: establishing desirable system properties and then designing a specification to capture them.

Specification validation: checking if the specification does indeed capture the expected system properties.

- prototyping and testing
- theorem proving

Formal specifications

Model-oriented approach: give a specific model — a system is *correct* if it displays the same behaviour.

Property-oriented approach: give a list of the properties required — a system is *correct* if it satisfies all of them.

In either case, start by determining the logical system to work with. . .

We will (pretend to) work in the standard algebraic framework

BUT: *everything carries over to more complex, and more realistic logical systems, capturing the semantics of more realistic programming paradigms.*

more about this elsewhere: **Institutions!**

Specification languages

Quite a few around. . . Choose one.

For instance: CASL :-)

Make even realistic large specification understandable!

Key idea: STRUCTURE

Use it to:

- build, understand and prove properties of specifications
- (though not necessarily to implement them)

Programmer's task

Given a requirements specification

produce a module that correctly implements it

Given a requirements specification SP

build a program P such that

$$SP \rightsquigarrow P$$

A formal definition of $\boxed{SP \rightsquigarrow P}$ is a given by the *semantics*

(of the specification formalism and of the programming language)

Recall the analogy:

module interface $\rightsquigarrow$ signature

module $\rightsquigarrow$ algebra

module specification $\rightsquigarrow$ class of algebras

Specification semantics

Given a specification SP:

- *signature* of SP: $Sig[SP]$
- *models* of SP: $Mod[SP] \subseteq \mathbf{Alg}(Sig[SP])$

We know what to start with:

Basic specifications: $\langle \Sigma, \Phi \rangle$

- $Sig[\langle \Sigma, \Phi \rangle] = \Sigma$
- $Mod[\langle \Sigma, \Phi \rangle] = Mod(\Phi)$

Keep them small...

Structured specifications

Built by combining, extending and modifying simpler specifications

Specification-building operations

For instance:

union: to combine constraints imposed by various specifications

translation: to rename and introduce new components

hiding: to hide interpretation of auxiliary components

Three typical, elementary, but quite flexible **sbo***'s*

Programmer's task

Informally:

Given a requirements specification

produce a module that correctly implements it

Semantically:

Given a requirements specification SP

build a model $M \in \mathbf{Alg}(Sig[SP])$ such that

$M \in Mod[SP]$

Development process:

$$\boxed{SP \rightsquigarrow M}$$

Never in a single jump!

Rather: proceed step by step, adding gradually more and more detail and incorporating more and more design and implementation decisions, until a specification is obtained that is easy to implement directly

$$\boxed{SP_0 \leadsto SP_1 \leadsto \cdots \leadsto SP_n}$$

ensuring:

$$\boxed{\frac{SP_0 \leadsto SP_1 \leadsto \cdots \leadsto SP_n \qquad SP_n \rightsquigarrow M}{SP_0 \rightsquigarrow M}}$$

Simple implementations

$$\boxed{SP \rightsquigarrow SP'}$$

Means:

$$Sig[SP'] = Sig[SP] \text{ and } Mod[SP'] \subseteq Mod[SP]$$

Proof obligation

linked with such implementations

- preserve the static interface (by preserving the signature)
- incorporate further details (by narrowing the class of models)

Composability follows:

$$\frac{SP \rightsquigarrow SP' \qquad SP' \rightsquigarrow SP''}{SP \rightsquigarrow SP''}$$

$$\frac{SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \qquad M \in Mod[SP_n]}{M \in Mod[SP_0]}$$

For instance

spec $\textsc{StringKey} = \textsc{String}$ **and** $\textsc{Nat}$
  then opn $hash\colon String \rightarrow Nat$

spec $\textsc{StringKey\_nil} = \textsc{String}$ **and** $\textsc{Nat}$
  then opn $hash\colon String \rightarrow Nat$
    axioms $hash(nil) = 0$

spec $\textsc{StringKey\_a\_z} = \textsc{String}$ **and** $\textsc{Nat}$
  then opn $hash\colon String \rightarrow Nat$
    axioms $hash(nil) = 0$
      $hash(a) = 1 \ldots hash(z) = 26$

THEN

$$\boxed{\textsc{StringKey} \rightsquigarrow \textsc{StringKey\_nil} \rightsquigarrow \textsc{StringKey\_a\_z}}$$

...and then, for instance

spec $\textsc{StringKeyCode} = \textsc{String}$ **and** $\textsc{Nat}$

then opns $hash\colon String \to Nat$

$str2nat\colon String \to Nat$

axioms $str2nat(nil) = 0$

$str2nat(a) = 1 \ldots str2nat(z) = 26$

$str2nat(str_1 \frown str_2) = str2nat(str_1) + str2nat(str_2)$

$hash(str) = str2nat(str)$ **mod** 15485857

hide $str2nat$

THEN

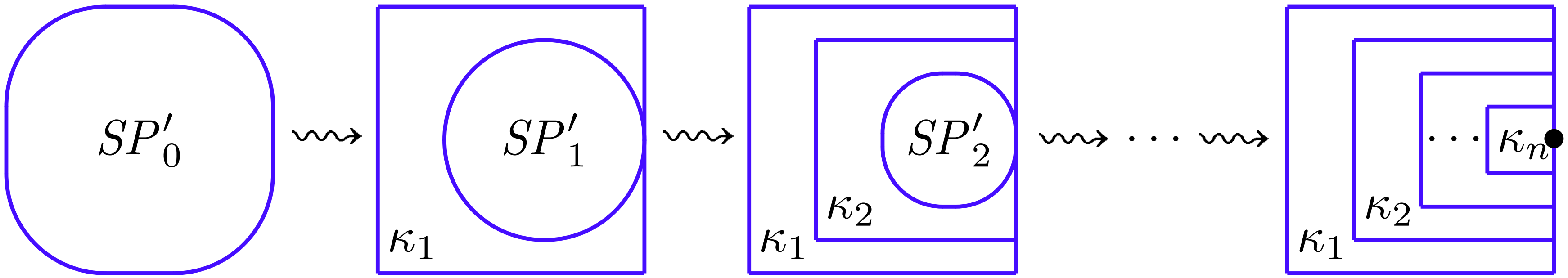
$\textsc{StringKey} \leadsto \textsc{StringKey\_nil} \leadsto \textsc{StringKey\_a\_z} \leadsto \textsc{StringKeyCode}$

...and the "code" in $\textsc{StringKeyCode}$

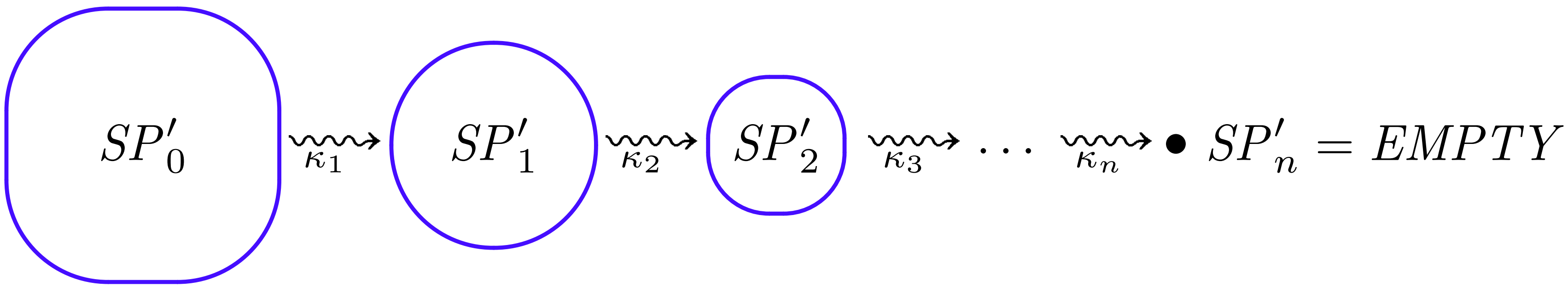
defines a program/model for $\textsc{StringKey}$

Extra twist

In practice, some parts will get fixed on the way:



Keep them apart from whatever is really left for implementation:



Constructor implementations

$$\boxed{SP \underset{\kappa}{\rightsquigarrow} SP'}$$

Proof obligation

linked with such implementations

Means:

$$\kappa(Mod[SP']) \subseteq Mod[SP]$$

where

$$\kappa\colon \mathbf{Alg}(Sig[SP']) \to \mathbf{Alg}(Sig[SP])$$

is a *constructor*:

Intuitively: *parameterised program* (*generic module*, SML *functor*)

Semantically: function between model classes

putting aside: *partiality*, *persistency*...

Composability revisited

$$\frac{SP \underset{\kappa}{\leadsto} SP' \qquad SP' \underset{\kappa'}{\leadsto} SP''}{SP \underset{\kappa';\kappa}{\leadsto} SP''}$$

$$\frac{SP_0 \underset{\kappa_1}{\leadsto} SP_1 \underset{\kappa_2}{\leadsto} \cdots \underset{\kappa_n}{\leadsto} SP_n = EMPTY}{\kappa_1(\kappa_2(\ldots\kappa_n(empty)\ldots)) \in Mod[SP_0]}$$

Methodological issues:

- *top-down* vs. *bottom-up* vs. *middle-out* development?
- *modular decomposition* (designing modular structure)

Warning: *Specification structure may change during the development!*

Separate means to design program modular structure

Branching implementation steps

$$SP \rightsquigarrow \boxed{\kappa \begin{cases} SP_1 \\ \vdots \\ SP_n \end{cases}}$$

This involves a "*linking procedure*" (n-argument constructor, parameterised program)

$$\kappa\colon \mathbf{Alg}(Sig[SP_1]) \times \cdots \times \mathbf{Alg}(Sig[SP_n]) \to \mathbf{Alg}(Sig[SP])$$

We require:

$$\frac{M_1 \in Mod[SP_1] \quad \cdots \quad M_n \in Mod[SP_n]}{\kappa(M_1, \ldots, M_n) \in Mod[SP]}$$

Proof obligation
linked with such design steps

CASL architectural specifications

CASL provides an explicit way to write down the *design specification* such a branching step amounts to:

$$\textbf{arch spec } ASP = \boxed{\begin{array}{ll} \textbf{units} & U_1 : SP_1 \\ & \ldots \\ & U_n : SP_n \\ \textbf{result} & \kappa(U_1, \ldots, U_n) \end{array}}$$

Moreover:

- units my be generic (parameterised programs, SML functors), but *always* are declared with their specifications
- CASL provides a rich collection of combinators to define κ and various additional ways to *define* units

Instead of conclusions

- Quite a lot of good theory around this;
- Even more bad practise . . .

Ever evading overall goal

Practical methods

for software specification and development

with solid foundations