Given a state consisting of one highly classified and one lowly classified object:

\[ \text{STATE}[p] \]
\[ \text{high, low : bool} \]

can we specify loosely an operation on the state which does not result in any information transfer from ‘high’ to ‘low’?

\[ \text{STATE, STATE'} \]
\[ \Rightarrow \]
\[ \text{f: STATE } \rightarrow \text{ STATE} \]
\[ \forall s^1 s^2: \text{STATE} \cdot (s^1.\text{low} = s^2.\text{low}) \]
\[ \Rightarrow ((f s^1).\text{low} = (f s^2).\text{low}) \]

We could then write our schema:

\[ \text{STATE, STATE'} = f \theta \text{STATE} \]

but since all the work has been done in the specification of ‘f’ the use of the schema appears superfluous.
Note that in the axiomatic definition of f, the requirement is expressed as a property of f, but this property has not itself been given a name.

It is therefore not possible to express in the object language the claim that some other explicitly defined function has this property.

For example the following function has the required property:

\[
g: \text{STATE} \rightarrow \text{STATE}
\]

\[
\forall s : \text{STATE} \cdot g \ s = s
\]

but we cannot state this in Z without restating the original property (though it can be said in the metalanguage).

To enable such correctness propositions to be expressed we must give a name to the property itself as follows:

\[
\text{secure} : \mathcal{P} (\text{STATE} \rightarrow \text{STATE})
\]

\[
f \in \text{secure} \iff \\
\forall s^1, s^2 : \text{STATE} \cdot (s^1.\text{low} = s^2.\text{low}) \\
\Rightarrow (f \ s^1).\text{low} = (f \ s^2).\text{low}
\]

The conjecture that ‘g’ satisfies this specification can now be expressed:

\[
\vdash ? \ g \in \text{secure}
\]
If we define a further requirement:

\[
\text{safe : } \mathbb{P} (\text{STATE} \rightarrow \text{STATE})
\]

\[
f \in \text{safe} \iff \\
\forall s^1, s^2 : \text{STATE} \cdot (s^1.\text{high} = s^2.\text{high}) \\
\Rightarrow (f s^1).\text{high} = (f s^2).\text{high}
\]

Then the combination of these two requirements:

\[
\text{no_flow : } \mathbb{P} (\text{STATE} \rightarrow \text{STATE})
\]

\[
\text{no_flow} = \text{secure} \cap \text{safe}
\]

may be regarded as a REFINEMENT of the original specification "secure".
That it is a refinement can be expressed in the object language as the conjecture:

\[
\vdash \text{no_flow} \subseteq \text{secure}
\]

Note that here refinement is defined as a relationship between specifications which is distinct from the relationship between a specification and an implementation.

**SPECIFYING OPERATIONS AS FUNCTIONS**

Type of \textit{Object}

\[
\text{AUTO}
\]

Type of \textit{Specification}

\[
\mathbb{P} \text{ AUTO}
\]

Type of \textit{Operation}

\[
\text{IN} \times \text{STATE} \rightarrow \text{STATE} \times \text{OUT} \\
\subseteq \mathbb{P}(\text{IN} \times \text{STATE} \times \text{STATE} \times \text{OUT})
\]

Type of \textit{Specification of Operation}

\[
\mathbb{P} (\text{IN} \times \text{STATE} \rightarrow \text{STATE} \times \text{OUT})
\]

Type of \textit{Non-Deterministic Operation}

\[
\text{IN} \times \text{STATE} \rightarrow \mathbb{P}^1 (\text{STATE} \times \text{OUT})
\]

Type of \textit{Specification of Non-Deterministic Operation}

\[
\mathbb{P} (\text{IN} \times \text{STATE} \rightarrow \mathbb{P}^1 (\text{STATE} \times \text{OUT}))
\]

Type of \textit{Partial Operation}

\[
\text{IN} \times \text{STATE} \Rightarrow \text{STATE} \times \text{OUT}
\]

Type of \textit{Specification of Partial Operation}

\[
\mathbb{P} (\text{IN} \times \text{STATE} \Rightarrow \text{STATE} \times \text{OUT})
\]

Type of \textit{Partial Non-Deterministic Operation}

\[
\text{IN} \times \text{STATE} \Rightarrow \mathbb{P}^1 (\text{STATE} \times \text{OUT})
\]

Type of \textit{Specification of Partial Non-Deterministic Operation}

\[
\mathbb{P} (\text{IN} \times \text{STATE} \Rightarrow \mathbb{P}^1 (\text{STATE} \times \text{OUT}))
\]
Z SCHEMAS INTERPRETED AS OPERATIONS

Until the publication of Spivey’s book "understanding Z" no account was available of how schemas are to be interpreted as specifications of operations.

Spivey gives an account of a satisfaction relationship between schemas and implementations which can be formalised within Z as follows.

Let us consider this with reference to schemas describing the secure operations discussed above.

The type of a schema describing an operation over \textsc{state} is:

\[
\text{SOPTYPE} \equiv \text{P} \land \text{STATE}
\]

According to Spivey this is a loose specification of a non-deterministic partial operation (in the general case). It may therefore be interpreted as an entity of type:

\[
\text{FTYPE} \equiv \text{STATE} \rightarrow \text{P} \land \text{STATE}
\]

\[
\text{INTTYPE} \equiv \text{P} \left(\text{STATE} \rightarrow \text{P} \land \text{STATE}\right)
\]

A formal account of this interpretation would therefore be a map from \text{OPTYPE} to \text{INTTYPE}:

\[
\text{MAPTYPE} \equiv \text{OPTYPE} \rightarrow \text{INTTYPE}
\]

\[
\text{\_satisfies\_ : FTYPE} \leftrightarrow \text{SOPTYPE}
\]

\[
\forall \text{SOPTYPE} : S, \text{FTYPE} : f \bullet
\]

\[
f \text{ satisfies } S \iff
\forall s : \text{STATE} \bullet
f s \subseteq \{S \mid \theta \text{STATE} = s \bullet \theta \text{STATE}'\}
\]

note that:

\[
f \text{ satisfies } S \land g \text{ satisfies } S
\]

\[
\Rightarrow (f \text{ merge } g) \text{ satisfies } S
\]

where \((f \text{ merge } g) x = f x \cup g x\)

\[
\text{ML}
\]

\text{new\_theory} 'form167'';
\text{new\_parent}'pf'';
\text{loadf}'infra'';
\texttt{high:bool, low:bool} \\
\texttt{T} \\

\texttt{secure : (STATE \rightarrow STATE) \rightarrow bool} \\
\begin{align*}
  f \in \text{secure} & \iff \\
  \forall s^1 \ s^2 : \text{STATE} \bullet (s^1 \cdot \text{S\_low} = s^2 \cdot \text{S\_low}) \\
  \Rightarrow ((f \ s^1) \cdot \text{S\_low} = (f \ s^2) \cdot \text{S\_low})
\end{align*}

\texttt{g: STATE \rightarrow STATE} \\
\begin{align*}
  \forall s : \text{STATE} \bullet g \ s = s
\end{align*}

\texttt{safe : (STATE \rightarrow STATE) \rightarrow bool} \\
\begin{align*}
  f \in \text{safe} & \iff \\
  \forall s^1 \ s^2 : \text{STATE} \bullet (s^1 \cdot \text{S\_high} = s^2 \cdot \text{S\_high}) \\
  \Rightarrow ((f \ s^1) \cdot \text{S\_high} = (f \ s^2) \cdot \text{S\_high})
\end{align*}

\texttt{set\_goal([],"g \in \text{secure}");} \\
\begin{align*}
  & \epsilon (\text{REWRITE\_TAC[secure\_THM;X\_2e\_DEF;g\_THM]}); \\
  & \text{let g\_secure\_THM = save\_top\_thm 'g\_secure\_THM'};;
\end{align*}

\texttt{set\_goal([],"g \in \text{safe}");} \\
\begin{align*}
  & \epsilon (\text{REWRITE\_TAC[safe\_THM;X\_2e\_DEF;g\_THM]}); \\
  & \text{let g\_safe\_THM = save\_top\_thm 'g\_safe\_THM'};;
\end{align*}

\texttt{no\_flow : (STATE \rightarrow STATE) \rightarrow bool} \\
\begin{align*}
  \text{no\_flow} & = \text{secure} \cap \text{safe}
\end{align*}
1. SECURE SYSTEM DESIGN USING SECURITY KERNEL

The secure system is just a function over the state which has the property ‘secure’. We show that such a system can be constructed from two subsystems, one of which is a trusted security kernel, and the other of which is an untrusted user program. The trustedness of the kernel is represented by a specification for the kernel which it must satisfy if the total system is to be secure. The untrustedness of the user program is reflected in its trivial specification.

The design consists of a set of subcomponent specifications and a construction showing how the system is built from the subcomponents.

1.1. The user program specification

The user program is any function which operates on the state.

```
USER_PROGRAM_TYPE == STATE → STATE
```

```
let USER_PROGRAM_TYPE = "STATE → STATE";;
```

```
user_program_specification : P USER_PROGRAM_TYPE

user_program_specification = {f:USER_PROGRAM_TYPE}
```
1.2. The Kernel Specification
The kernel mediates between the user program and the secure data store.

\[ \text{KERNEL\_TYPE} = \text{USER\_PROGRAM\_TYPE} \times \text{bool} \rightarrow (\text{STATE} \rightarrow \text{STATE}) \]

\[ \text{kernel\_specification} : \mathbb{P} \text{KERNEL\_TYPE} \]

\[ k \in \text{kernel\_specification} \iff \forall f: \text{USER\_PROGRAM\_TYPE}, b: \text{bool} \cdot k(f, b) \in \text{secure} \]

let KERNEL\_TYPE = "\text{USER\_PROGRAM\_TYPE} \times \text{bool} \rightarrow (\text{STATE} \rightarrow \text{STATE})";;

\[ \text{kernel\_specification} : \wedge \text{KERNEL\_TYPE} \rightarrow \text{bool} \]

\[ k \in \text{kernel\_specification} \iff \forall (f:\text{USER\_PROGRAM\_TYPE}) (b: \text{bool}) \cdot k(f, b) \in \text{secure} \]

i.e. a kernel is something which can run an untrusted program at any classification without permitting a breach of security.

1.3. The Construction
The system as a whole consists of a security kernel running some application at some clearance. Let us leave the classification loosely undefined.

user\_clearance:bool

T

The construction is a function which takes a security kernel and a user program and yields a secure system:
secure_system_construction : (KERNEL_TYPE × (STATE → STATE)) → (STATE → STATE)

∀k:KERNEL_TYPE, u:STATE → STATE•
    secure_system_construction (k,u) = k(u,user_clearance)

secure_system_construction : (KERNEL_TYPE × USER_PROGRAM_TYPE) → (STATE → STATE)

∀(k:KERNEL_TYPE) (u:USER_PROGRAM_TYPE)•
    secure_system_construction (k,u) = k(u,user_clearance)

The correctness of this design is the conjecture that this construction will always yield secure system when applied to subcomponents which meet their specifications.

set_goal([],"∀(k:KERNEL_TYPE) (u:USER_PROGRAM_TYPE)•
    u ∈ user_program_specification
    ∧ k ∈ kernel_specification
    ⇒ secure_system_construction(k,u) ∈ secure");

ML

1.4. An Implementation of the Kernel

The design correctness was a very trivial result, because of the way the kernel specification was written. This may look like cheating, but we now show an example of a security kernel and prove that it meets the specification.

The kernel filters the current state before passing it on to the user program, and then filters the state computed by the application before using it to update the real state. These two filtering operations represent the placement of constraints on read and on write which are dependent on the clearance of the user program.

Filtering isn't strictly possible because of the primitive structure of the state.

We define a function which takes two STATES and a classification and merges the two states taking values dominated by the classification from the first state and other values from the second.
_dominates_ : bool ↔ bool

∀x:bool• x dominates x ∧ T dominates F

dominates: bool → bool → bool

∀(x:bool)(y:bool)• dominates x y ⇔ y ⇒ x

ML

print_theory‘form167‘;;
The Theory form 167
Parents -- HOL pf
Types -- "STATE"
Constants --
IS_STATE ":bool × bool → bool"
REP_STATE ":STATE → bool × bool"
ABS_STATE ":bool × bool → STATE" S_high ":STATE → bool"
S_low ":STATE → bool" secure ":(STATE → STATE) → bool"
g ":STATE → STATE" secure ":(STATE → STATE) → bool"
no_flow ":(STATE → STATE) → bool"
user_program_specification ":(STATE → STATE) → bool"
kernel_specification
":(STATE → STATE) × bool → (STATE → STATE)) → bool"
user_clearance ":bool"
secure_system_construction
":(STATE → STATE) × bool → (STATE → STATE)) × (STATE → STATE) → (STATE → STATE)"
Curried Infixes -- dominates ":bool → (bool → bool)"
Axioms --
STATE_AXIOM
| ≃rep•
| (∀x’ x”• (rep x’ = rep x”) ⇒ (x’ = x”)) ∧
| (∀x• p_or_choice IS_STATE x = (∃x’• x = rep x’))
Definitions --
REP_STATE
| ≃REP_STATE =
| (µrep•
| (∀x’ x”• (rep x’ = rep x”) ⇒ (x’ = x”)) ∧
| (∀x• p_or_choice IS_STATE x = (∃x’• x = rep x’))
ABS_STATE
| ≃∀x• ABS_STATE x = (µx′• x = REP_STATE x’)
Theorems --
IS_STATE_THM
| ∀high low• IS_STATE(high,low)
STATE_assorted_THMS
| (∃a a’• (REP_STATE a = REP_STATE a’) = (a = a’)) ∧
| (∀r• p_or_choice IS_STATE r = (∃a• t = REP_STATE a)) ∧
| (∀r t’•
| p_or_choice IS_STATE r ⇒
| p_or_choice IS_STATE r’ ⇒
| ((ABS_STATE r = ABS_STATE r’) = (r = r’)) ∧
| (∀a• ∃r• (a = ABS_STATE r) ∧ p_or_choice IS_STATE r) ∧
| (∀a• ABS_STATE(REP_STATE a) = a) ∧
| (∀r• p_or_choice IS_STATE r = (REP_STATE(ABS_STATE r) = r))
STATE_IS_REP_THM
| ∀x• IS_STATE(S_high x,S_low x)
ABS_REP_STATE_THM
| ∀x• ABS_STATE(S_high x,S_low x) = x
S_high_THM
| ∀high low• S_high(ABS_STATE(high,low)) = high
\begin{align*}
S_{low} \text{ |} & \forall \text{high low} \cdot S_{low}(\text{ABS\_STATE}(\text{high}, \text{low})) = \text{low} \\
\text{secure} \text{ |} & \forall f \cdot \\
& f \in \text{secure} \iff \\
& (\forall s^1 s^2 \cdot \\
& (s^1 \cdot S_{low} = s^2 \cdot S_{low}) \Rightarrow \\
& ((f \cdot s^1) \cdot S_{low} = (f \cdot s^2) \cdot S_{low})) \\
g_{\text{THM}} \text{ |} & \forall s \cdot g s = s \\
g_{\text{secure} \text{ |}} & g \in \text{secure} \\
\text{safe} \text{ |} & \forall f \cdot \\
& f \in \text{safe} \iff \\
& (\forall s^1 s^2 \cdot \\
& (s^1 \cdot S_{high} = s^2 \cdot S_{high}) \Rightarrow \\
& ((f \cdot s^1) \cdot S_{high} = (f \cdot s^2) \cdot S_{high})) \\
g_{\text{safe} \text{ |}} & g \in \text{safe} \\
\text{no\_flow} \text{ |} & \forall f \cdot \text{no\_flow} \Rightarrow f \in \text{secure} \\
\text{no\_flow} \text{ |} & \forall f \cdot \text{no\_flow} \Rightarrow f \in \text{safe} \\
g_{\text{no\_flow} \text{ |}} & g \in \text{no\_flow} \\
\text{secure} \text{ |} & \text{no\_flow} \subseteq \text{secure} \\
\text{user\_program\_specification} \text{ |} & \forall f \cdot \text{user\_program\_specification} = (\lambda f \cdot T) \\
\text{kernel\_specification} \text{ |} & \forall k \cdot k \in \text{kernel\_specification} \iff (\forall b \cdot (k(f,b) \in \text{secure})) \\
\text{user\_clearance} \text{ |} & T \\
\text{secure\_system\_construction} \text{ |} & \forall k \cdot \text{secure\_system\_construction}(k,u) = k(u, \text{user\_clearance}) \\
\text{design\_correct} \text{ |} & \forall k \cdot \\
& u \in \text{user\_program\_specification} \land k \in \text{kernel\_specification} \Rightarrow \\
& (\text{secure\_system\_construction}(k,u)) \in \text{secure} \\
\text{dominates} \text{ |} & \forall x \cdot x \text{ dominates } y \iff y \Rightarrow x
\end{align*}