Formalising FinFuns – Generating Code for Functions as Data from Isabelle/HOL

Andreas Lochbihler

Universität Karlsruhe (TH)

17.8.2009, TPHOLs 2009

Motivation I — Quickcheck

```
lemma "{a,b} = {x,y} ⟷ a=x∧b=y ∨ a=y∧b=x"
by(blast elim: equalityE)

lemma
  "{a,b,c} = {x,y,z} ⟷ a=x∧b=y∧c=z ∨ a=x∧b=z∧c=y ∨
                        a=y∧b=x∧c=z ∨ a=y∧b=z∧c=x ∨
                        a=z∧b=x∧c=y ∨ a=z∧b=y∧c=x"
quickcheck
```

Motivation I — Quickcheck

```
lemma "{a,b} = {x,y} ⟷ a=x∧b=y ∨ a=y∧b=x"
by(blast elim: equalityE)

lemma
  "{a,b,c} = {x,y,z} ⟷ a=x∧b=y∧c=z ∨ a=x∧b=z∧c=y ∨
                        a=y∧b=x∧c=z ∨ a=y∧b=z∧c=x ∨
                        a=z∧b=x∧c=y ∨ a=z∧b=y∧c=x"
quickcheck
```

```
*** Unable to generate code for term:
*** {a, b, c} = {x, y, z}
*** required by:
*** <Top>
*** At command "quickcheck".
```

Motivation I — Quickcheck

```
lemma "{a,b} = {x,y} ⟷ a=x∧b=y ∨ a=y∧b=x"
by(blast elim: equalityE)

lemma
  "{a,b,c} = {x,y,z} ⟷ a=x∧b=y∧c=z ∨ a=x∧b=z∧c=y ∨
                        a=y∧b=x∧c=z ∨ a=y∧b=z∧c=x ∨
                        a=z∧b=x∧c=y ∨ a=z∧b=y∧c=x"
quickcheck
```

```
*** Unable to generate code for term:
*** {a, b, c} = {x, y, z}
*** required by:
*** <Top>
*** At command "quickcheck".
```

```
Counterexample found:

a = int (Suc 0)
b = int 0
c = int 0
x = int (Suc 0)
y = int (Suc 0)
z = int 0
```

Motivation II — Code generation

```
fun conf :: "val ⇒ ty ⇒ bool" ("_ :≤ _" [80, 80] 100)
where "Intg _ :≤ Integer = True"
    | "Bool _ :≤ Boolean = True"
    | "_ :≤ _             = False"

types state = "var ⇒ val"
types env = "var ⇒ ty"

definition conf_state :: "env ⇒ state ⇒ bool" ("_ ⊢ _ [ok]" [80, 0] 100)
where "E ⊢ σ [ok] ⟷ (∀V. σ V :≤ E V)"

export_code conf_state in Haskell file -
```

Motivation II — Code generation

```
fun conf :: "val ⇒ ty ⇒ bool" ("_ :≤ _" [80, 80] 100)
where "Intg _ :≤ Integer = True"
    | "Bool _ :≤ Boolean = True"
    | "_ :≤ _            = False"

types state = "var ⇒ val"
types env = "var ⇒ ty"

definition conf_state :: "env ⇒ state ⇒ bool" ("_ ⊢ _ [ok]" [80, 0] 100)
where "E ⊢ σ [ok] ⟷ (∀V. σ V :≤ E V)"

export_code conf_state in Haskell file -
```

```
*** Wellsortedness error
*** (in code equation "?e ⊢ ?sigma [ok] ≡ ∀v. ?sigma v :≤ ?e v"):
*** Type FinFunExamples.var not of sort enum
*** No type arity FinFunExamples.var :: enum
*** At command "export_code".
```

Shortcomings of finite maps

Finite maps in Isabelle

1. Model as function: $'a \Rightarrow {}'b$ *option*
   $\Rightarrow$ No support for code generation
2. Use associative lists: $('a \times {}'b)$ *list*

   Clutter proofs with implementation details

   Suffer from multiple representations:
   $\mathit{map\text{-}of}\ [(2,32),(3,100)] = \mathit{map\text{-}of}\ [(3,100),(2,32)]$

- *None* is always the "default value"
- Only useful for functions $'a \Rightarrow {}'b$ *option*
  What about other types?

Use FinFuns to push the code generator's boundaries

Function equality and quantifiers are a major limitation for Isabelle's code generator

Solution: Represent functions explicitly as graph

- Default value for almost all points
- List of point-value pairs for changed points

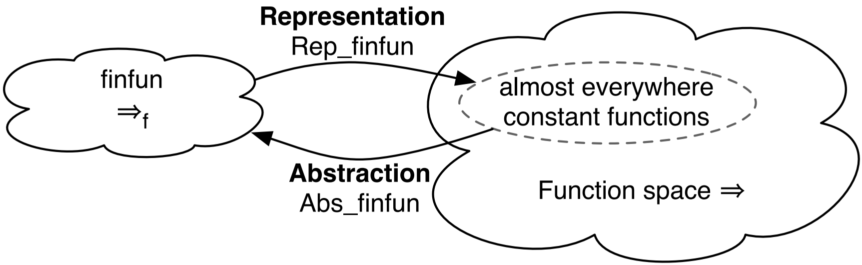
$(3 :=_f 100)$
$(2 :=_f 32)$
$K^f 0$

Goals:

- easy to use:
  - Logically like functions (extensionality, application, composition, ...)
  - Syntax similar to ordinary functions
- executability (equality, quantifiers, composition)
- conservative extension of Isabelle and the code generator

The new FinFun type $\Rightarrow_f$

Logic: subset of function space



Implementation: datatype

```
data Finfun a b = Finfun_const b
                | Finfun_update_code (Finfun a b) a b
```

Operators for FinFuns

FinFun	ordinary function	operation	complexity
$K^f\ c$	$\lambda x.\ c$	constant function	$\mathcal{O}(1)$
$\hat{f}(a :=_f b)$	$f(a := b)$	pointwise update	$\mathcal{O}(\#\hat{f})$
$\hat{f}_f\ x$	$f\ x$	application	$\mathcal{O}(\#\hat{f})$
$g \circ_f \hat{f}$	$g \circ f$	composition	$\mathcal{O}(\#\hat{f} \cdot (\#\hat{f} + \|g\|))$
$(\hat{f},\ \hat{g})^f$	$\lambda x.\ (f\ x,\ g\ x)$	parallel evaluation	$\mathcal{O}(\#\hat{f} \cdot (\#\hat{f} + \#\hat{g}))$
finfun-All $\hat{P}$	$\forall x.\ P\ x$	universal quantifier	$\mathcal{O}((\#\hat{P})^2)$
$\hat{f} = \hat{g}$	$f = g$	equality test	$\mathcal{O}((\#\hat{f} + \#\hat{g})^2)$

where $\#\hat{f}$ = number of updated points in $\hat{f}$

Equality test via universal quantification:

$$f = g \ \leftrightarrow\ \forall x.\ f\ x = g\ x \ \leftrightarrow\ \forall x.\ ((\lambda(y,\ z).\ y = z) \circ (\lambda x.\ f\ x,\ g\ x))\ x$$
$$\hat{f} = \hat{g} \ \leftrightarrow\ \textit{finfun-All}\ ((\lambda(y,\ z).\ y = z) \circ_f (\hat{f},\ \hat{g})^f)$$

Deleting redundant updates

1. Constructor *finfun-update-code* $\hat{f}\ a\ b$ (written $\hat{f}(\!|a :=_f b|\!)$) instead of $\hat{f}(a :=_f b)$.
2. Implement $\hat{f}(a :=_f b)$ recursively:

$$(K^f\ c)(a :=_f b) = \mathit{if}\ b = c\ \mathit{then}\ K^f\ c\ \mathit{else}\ (K^f\ c)(\!|a :=_f b|\!)$$

$$(\hat{f}(\!|a' :=_f b'|\!))(a :=_f b) = \mathit{if}\ a = a'\ \mathit{then}\ \hat{f}(a :=_f b)\ \mathit{else}\ (\hat{f}(a :=_f b))(\!|a' :=_f b'|\!)$$

$\hat{f}$

$(\!|3 :=_f \mathit{False}|\!)$
$(\!|2 :=_f \mathit{False}|\!)$
$(\!|1 :=_f \mathit{False}|\!)$
$K^f\ \mathit{True}$

$\hat{f}(2 :=_f \mathit{True})$

Deleting redundant updates

1. Constructor *finfun-update-code* $\hat{f}$ a b (written $\hat{f}(\!|a :=_f b|\!)$) instead of $\hat{f}(a :=_f b)$.
2. Implement $\hat{f}(a :=_f b)$ recursively:

$$(K^f\ c)(a :=_f b) = \mathit{if}\ b = c\ \mathit{then}\ K^f\ c\ \mathit{else}\ (K^f\ c)(\!|a :=_f b|\!)$$

$$(\hat{f}(\!|a' :=_f b'|\!))(a :=_f b) = \mathit{if}\ a = a'\ \mathit{then}\ \hat{f}(a :=_f b)\ \mathit{else}\ (\hat{f}(a :=_f b))(\!|a' :=_f b'|\!)$$

$\hat{f}$		$\hat{f}(2 :=_f \mathit{True})$
$(\!|3 :=_f \mathit{False}|\!)$	$(2 :=_f \mathit{True})$	$(\!|3 :=_f \mathit{False}|\!)$
$(\!|2 :=_f \mathit{False}|\!)$		
$(\!|1 :=_f \mathit{False}|\!)$		
$K^f\ \mathit{True}$		

Deleting redundant updates

1. Constructor *finfun-update-code* $\hat{f}\ a\ b$ (written $\hat{f}\llparenthesis a :=_f b \rrparenthesis$) instead of $\hat{f}(a :=_f b)$.
2. Implement $\hat{f}(a :=_f b)$ recursively:

$$(K^f\ c)(a :=_f b) = \mathit{if}\ b = c\ \mathit{then}\ K^f\ c\ \mathit{else}\ (K^f\ c)\llparenthesis a :=_f b \rrparenthesis$$

$$(\hat{f}\llparenthesis a' :=_f b' \rrparenthesis)(a :=_f b) = \mathit{if}\ a = a'\ \mathit{then}\ \hat{f}(a :=_f b)\ \mathit{else}\ (\hat{f}(a :=_f b))\llparenthesis a' :=_f b' \rrparenthesis$$

$\hat{f}$

$\llparenthesis 3 :=_f \mathit{False} \rrparenthesis$
$\llparenthesis 2 :=_f \mathit{False} \rrparenthesis$
$\llparenthesis 1 :=_f \mathit{False} \rrparenthesis$
$K^f\ \mathit{True}$

$(2 :=_f \mathit{True})$

$\hat{f}(2 :=_f \mathit{True})$

$\llparenthesis 3 :=_f \mathit{False} \rrparenthesis$

Deleting redundant updates

1. Constructor *finfun-update-code* $\hat{f}\ a\ b$ (written $\hat{f}\llparenthesis a :=_f b \rrparenthesis$) instead of $\hat{f}(a :=_f b)$.
2. Implement $\hat{f}(a :=_f b)$ recursively:

$$(K^f\ c)(a :=_f b) = \mathit{if}\ b = c\ \mathit{then}\ K^f\ c\ \mathit{else}\ (K^f\ c)\llparenthesis a :=_f b \rrparenthesis$$

$$(\hat{f}\llparenthesis a' :=_f b' \rrparenthesis)(a :=_f b) = \mathit{if}\ a = a'\ \mathit{then}\ \hat{f}(a :=_f b)\ \mathit{else}\ (\hat{f}(a :=_f b))\llparenthesis a' :=_f b' \rrparenthesis$$

$\hat{f}$
$\llparenthesis 3 :=_f \mathit{False} \rrparenthesis$
$\llparenthesis 2 :=_f \mathit{False} \rrparenthesis$
$\llparenthesis 1 :=_f \mathit{False} \rrparenthesis$
$K^f\ \mathit{True}$

$(2 :=_f \mathit{True})$

$\hat{f}(2 :=_f \mathit{True})$
$\llparenthesis 3 :=_f \mathit{False} \rrparenthesis$
$\llparenthesis 1 :=_f \mathit{False} \rrparenthesis$

Deleting redundant updates

1. Constructor *finfun-update-code* $\hat{f}\ a\ b$ (written $\hat{f}(\!|a :=_f b|\!)$) instead of $\hat{f}(a :=_f b)$.
2. Implement $\hat{f}(a :=_f b)$ recursively:

$$(K^f\ c)(a :=_f b) = \textit{if}\ b = c\ \textit{then}\ K^f\ c\ \textit{else}\ (K^f\ c)(\!|a :=_f b|\!)$$

$$(\hat{f}(\!|a' :=_f b'|\!))(a :=_f b) = \textit{if}\ a = a'\ \textit{then}\ \hat{f}(a :=_f b)\ \textit{else}\ (\hat{f}(a :=_f b))(\!|a' :=_f b'|\!)$$

$\hat{f}$
$(\!|3 :=_f \textit{False}|\!)$
$(\!|2 :=_f \textit{False}|\!)$
$(\!|1 :=_f \textit{False}|\!)$
$K^f\ \textit{True}$

$(2 :=_f \textit{True})$

$\hat{f}(2 :=_f \textit{True})$
$(\!|3 :=_f \textit{False}|\!)$
$(\!|1 :=_f \textit{False}|\!)$
$K^f\ \textit{True}$

Deleting redundant updates

1. Constructor *finfun-update-code* $\hat{f}\ a\ b$ (written $\hat{f}(\!| a :=_f b |\!)$) instead of $\hat{f}(a :=_f b)$.
2. Implement $\hat{f}(a :=_f b)$ recursively:

$$(K^f\ c)(a :=_f b) = \mathit{if}\ b = c\ \mathit{then}\ K^f\ c\ \mathit{else}\ (K^f\ c)(\!| a :=_f b |\!)$$

$$(\hat{f}(\!| a' :=_f b' |\!))(a :=_f b) = \mathit{if}\ a = a'\ \mathit{then}\ \hat{f}(a :=_f b)\ \mathit{else}\ (\hat{f}(a :=_f b))(\!| a' :=_f b' |\!)$$

$\hat{f}$
$(\!| 3 :=_f \mathit{False} |\!)$
$(\!| 2 :=_f \mathit{False} |\!)$
$(\!| 1 :=_f \mathit{False} |\!)$
$K^f\ \mathit{True}$

$\hat{f}(2 :=_f \mathit{True})$
$(\!| 3 :=_f \mathit{False} |\!)$
$(\!| 1 :=_f \mathit{False} |\!)$
$K^f\ \mathit{True}$

Universal quantifier *finfun-All*

$$\textit{finfun-All}\ \hat{P} \leftrightarrow \forall x.\ \hat{P}_f\ x$$

no recursive implementation

Generalise: *ff-All xs* $\hat{P}$ holds if $\hat{P}$ holds everywhere except at points in *xs*:

$$\textit{ff-All}\ xs\ \hat{P} \leftrightarrow \forall x.\ x \in \textit{set}\ xs \vee \hat{P}_f\ x$$
$$\textit{finfun-All}\ \hat{P} \leftrightarrow \textit{ff-All}\ []\ \hat{P}$$

Recursive implementation:

$$\textit{ff-All}\ xs\ (K^f\ c) \leftrightarrow c \vee \textit{set}\ xs = \textit{UNIV}$$
$$\textit{ff-All}\ xs\ (\hat{P}(\!|x :=_f y|\!)) \leftrightarrow (y \vee x \in \textit{set}\ xs) \wedge \textit{ff-All}\ (x \cdot xs)\ \hat{P}$$

set xs = *UNIV*: use type information (implemented via type classes)

infinite type: always false

finite type: count distinct elements in *xs* and compare to *card UNIV*

Example revisited: executable state conformance

```
types state = "var ⇒ val"
types env = "var ⇒ ty"

definition conf_state :: "env ⇒ state ⇒ bool" ("_ ⊢ _ [ok]" [80, 0] 100)
where "E ⊢ σ [ok] ⟷ (∀V. σ V :≤ E V)"

export_code conf_state in Haskell file -
```

```
types state = "var ⇒f val"
types env = "var ⇒f ty"

definition conf_state :: "env ⇒ state ⇒ bool" ("_ ⊢ _ [ok]" [80, 0] 100)
where "E ⊢ σ [ok] ⟷ (∀V. σf V :≤ Ef V)"

lemma conf_state_code [code]:
  "E ⊢ σ [ok] ⟷ finfun_All ((λ(V, T). V :≤ T) ∘f (σ, E)^f)"
by(auto simp add: conf_state_def finfun_All_All)

export_code conf_state in Haskell file -
```

WORKS!

Much more ...

"Primitive" recursion operator *finfun-rec*:

$$\mathit{finfun\text{-}rec}\ c\ u\ (K^f\ b) = c\ b$$

$$\mathit{finfun\text{-}rec}\ c\ u\ (\hat{f}(a :=_f b)) = u\ a\ b\ (\mathit{finfun\text{-}rec}\ c\ u\ \hat{f})$$

c and u must satisfy identities between K^f _ and _(_ $:=_f$ _)

Executable sets:

Subset: $\hat{f} \subseteq_f \hat{g} \leftrightarrow \mathit{finfun\text{-}All}\ ((\lambda(x, y).\ x \longrightarrow y) \circ_f (\hat{f}, \hat{g})^f)$

Complement: $-\hat{f} = (\lambda b.\ \neg b) \circ_f \hat{f}$

...

Code generator automatically replaces set operations with FinFun operations where possible.

Conclusion

Future work:

- Binary search trees for totally ordered domains
- Better integration with Isabelle packages

Summary:

+ FinFuns generalise finite maps
+ Executable equality and quantifiers for almost everywhere constant functions
+ Syntax similar to ordinary functions
+ Conservative extension of Isabelle and the code generator

– No λ-abstraction, higher-order unification

- Available in the AFP and Isabelle Development Snapshot