New features of the Isabelle theorem prover

—

proof terms and code generation

Stefan Berghofer[1]

Institut für Informatik
TU München

[1]supported by DFG PhD programme "Logic in Computer Science"

Proof terms

The LCF-Approach

- Abstract datatype of theorems
- Theorems can only be constructed by using a fixed set of primitive operations
- In principle, no recording of proofs necessary

But how can we ...

- ... exchange theorems between different theorem provers?
- ... check proofs of theorems by a small and independent checker?
- ... do things such as proof-carrying code?

$\Longrightarrow$ Proof terms for Isabelle

Isabelle — a logical framework

Meta logic

Intuitionistic higher order logic

- Terms: $t = x \mid c \mid \lambda x :: \tau.\ t \mid t\ t$
- Types: $\tau = \alpha \mid (\tau_1, \ldots, \tau_n) tc$ where $tc \in \{\rightarrow,\ \mathsf{prop},\ \ldots\}$
- Logical connectives:

universal quantification	$\bigwedge$	::	$(\alpha \rightarrow \mathsf{prop}) \rightarrow \mathsf{prop}$
implication	$\Longrightarrow$	::	$\mathsf{prop} \rightarrow \mathsf{prop} \rightarrow \mathsf{prop}$

Object logics are formalized using meta logic

Tr	::	$\mathsf{bool} \rightarrow \mathsf{prop}$
$\exists$	::	$(\alpha \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}$
...		
exI	:	$\bigwedge P\ x. \mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ (\exists x.\ P\ x)$
exE	:	$\bigwedge P\ Q. \mathsf{Tr}\ (\exists x.\ P\ x) \Longrightarrow (\bigwedge x.\ \mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ Q) \Longrightarrow \mathsf{Tr}\ Q$
...		

Proof terms – basic concepts

$$p \;=\; h \mid c_{[\overline{\tau_n/\alpha_n}]} \mid \lambda h : \varphi.\ p \mid \lambda x :: \tau.\ p \mid p\ p \mid p\ t$$

Provability

$$\frac{}{\Gamma, h : \varphi, \Gamma' \vdash h : \varphi} \qquad \frac{\Sigma(c) = \varphi}{\Gamma \vdash c_{[\overline{\tau_n/\alpha_n}]} : \varphi[\overline{\tau_n/\alpha_n}]}$$

$$\frac{\Gamma, h : \varphi \vdash p : \psi \quad \Gamma \vdash \varphi :: \mathsf{prop}}{\Gamma \vdash (\lambda h : \varphi.\ p) : \varphi \Longrightarrow \psi} \qquad \frac{\Gamma, x :: \tau \vdash p : \varphi}{\Gamma \vdash (\lambda x :: \tau.\ p) : \bigwedge x :: \tau.\ \varphi} \quad x \notin \mathsf{FV}(\Gamma)$$

$$\frac{\Gamma \vdash p : \varphi \Longrightarrow \psi \quad \Gamma \vdash q : \varphi}{\Gamma \vdash (p\ q) : \psi} \qquad \frac{\Gamma \vdash p : \bigwedge x :: \tau.\ \varphi \quad \Gamma \vdash t :: \tau}{\Gamma \vdash (p\ t) : \varphi[t/x]}$$

Proof states

- Proof states are represented as theorems

$$\varphi_1 \Longrightarrow \cdots \Longrightarrow \varphi_n \Longrightarrow \varphi \qquad\qquad \text{where } \varphi_i = \bigwedge \overline{x}.\ \overline{A} \Longrightarrow P$$

- Proof term corresponding to proof state:

$$\lambda(g_1 : \varphi_1) \ldots (g_n : \varphi_n).\ \ \ldots\ (g_i\ \overline{x^i}\ \overline{h^i})\ \ldots$$

Reconstruction of partial proof terms

- What can be omitted?

$$p \;=\; h \mid c_{[\overline{\tau_n/\alpha_n}]} \mid \lambda h : \varphi.\ p \mid \lambda x :: \tau.\ p \mid p\ p \mid p\ t$$

- Model missing information by unification variables:

$$\lambda x{::}\tau.\ (p\, t) \qquad \rightsquigarrow \qquad \lambda x{::}?\alpha.\ (p\,(?f\ x))$$

Note: "lifting" of variables!

- Proof reconstruction judgement: $\Gamma \vdash p \rhd (\varphi,\ C)$

"p proves φ (in context Γ), provided that the constraints C are solvable"

- C contains equality constraints between terms ($t_1 =^? t_2$) and types ($\tau_1 =^? \tau_2$)

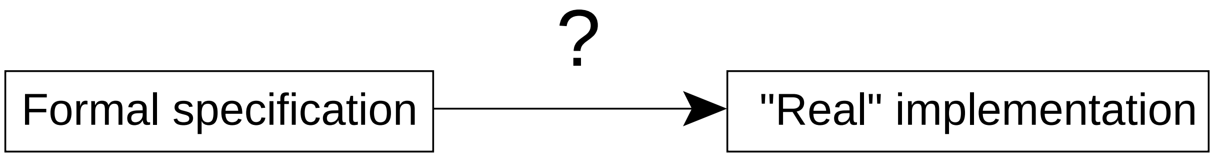
- What can be reconstructed by pattern unification?

$$\begin{array}{lll} \mathsf{exI} & : & \mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ (\exists x.\ P\ x) \\ \mathsf{exE} & : & \mathsf{Tr}\ (\exists x.\ P\ x) \Longrightarrow (\bigwedge x.\ \mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ Q) \Longrightarrow \mathsf{Tr}\ Q \end{array}$$

Code Generation

Why code generation?

- Validation problem: Have we specified the right thing?
- Bridge the gap between specification and implementation



Design goals

- Isabelle: generic theorem prover $\Longrightarrow$ Generic code generator required!
- Target language: Functional programming language (currently ML)

Specifications in Isabelle/HOL

A surprisingly large part of Isabelle/HOL specifications is executable:

- Inductive Datatypes

  datatype term = Var nat | App term term | Abs term

- Primitive recursive functions

  primrec

  subst (Var i) s k = (if $k < i$ then Var $(i-1)$ else if $i = k$ then s else Var i)
  subst (App t u) s k = App (subst t s k) (subst u s k)
  subst (Abs t) s k = Abs (subst t (lift s 0) $(k+1)$)

- Inductive definitions ($\approx$ PROLOG-style Horn Clauses)

  inductive

  App (Abs s) $t \rightarrow_\beta$ subst s t 0
  $s \rightarrow_\beta t \Longrightarrow$ App s $u \rightarrow_\beta$ App t u
  $s \rightarrow_\beta t \Longrightarrow$ App u $s \rightarrow_\beta$ App u t
  $s \rightarrow_\beta t \Longrightarrow$ Abs $s \rightarrow_\beta$ Abs t

Code generation — Summary

Applications

- Project Bali: Formal specification of Java programming language semantics
- Generate executable code (i.e. interpreters) from specification of operational semantics and JVM
- Execute sample programs with generated interpreters

Interesting issues

- HOL does not distinguish between computable and non-computable objects.
- What to do with non-executable things such as unbounded quantification, or the selection operator (ε)?