# Functional Programming with Isabelle/HOL

Florian Haftmann
Technische Universität München

January 2009

## Overview

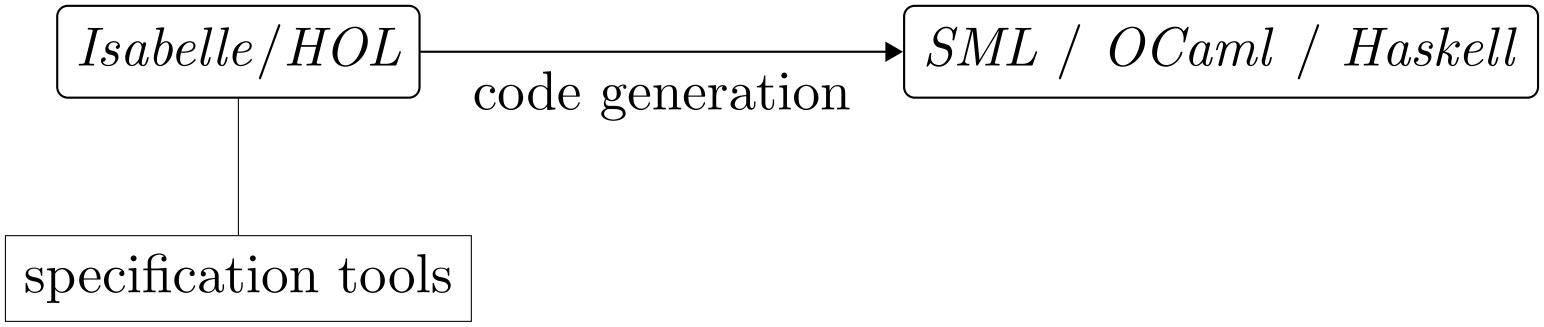
Viewing *Isabelle/HOL* as a functional programming language:

1. *Isabelle/HOL* Specification Tools.
2. Code Generation from *Isabelle/HOL*-Theories.
3. Behind the Scene.

## Overview

Viewing *Isabelle/HOL* as a functional programming language:

1. *Isabelle/HOL* Specification Tools.
2. Code Generation from *Isabelle/HOL*-Theories.
3. Behind the Scene.



## $Isabelle/HOL$ **specification tools**

## The definitional game

*Aim:* write "programs" in *Isabelle/HOL* as naturally as in, say, *SML* ...

## The definitional game

*Aim:* write “programs” in *Isabelle/HOL* as naturally as in, say, *SML* . . .
but ***it's not enough just to claim arbitrary things***:

**axiomatization** $nonsense :: nat \Rightarrow nat$ **where**
  $nonsense\text{-}def$: $nonsense\ n = Suc\ (nonsense\ n)$

## The definitional game

*Aim:* write "programs" in $\mathit{Isabelle/HOL}$ as naturally as in, say, $\mathit{SML}$ ...
but *it's not enough just to claim arbitrary things*:

**axiomatization** $\mathit{nonsense} :: \mathit{nat} \Rightarrow \mathit{nat}$ **where**
  $\mathit{nonsense\text{-}def}$: $\mathit{nonsense}\ n = \mathit{Suc}\ (\mathit{nonsense}\ n)$

**lemma** $0 = \mathit{Suc}\ 0$
**proof** $-$
  **from** $\mathit{nonsense\text{-}def}$
    **have** $\mathit{nonsense}\ 0 - \mathit{nonsense}\ 0 = \mathit{Suc}\ (\mathit{nonsense}\ 0) - \mathit{nonsense}\ 0$ **by** $\mathit{simp}$
  **then show** $0 = \mathit{Suc}\ 0$ **by** $\mathit{simp}$
**qed**

## The definitional game

*Aim:* write “programs” in *Isabelle/HOL* as naturally as in, say, *SML* ...
but *it's not enough just to claim arbitrary things*:

```
axiomatization nonsense :: nat ⇒ nat where
  nonsense-def: nonsense n = Suc (nonsense n)

lemma 0 = Suc 0
proof −
  from nonsense-def
    have nonsense 0 − nonsense 0 = Suc (nonsense 0) − nonsense 0 by simp
  then show 0 = Suc 0 by simp
qed
```

Things have to be properly *constructed*, that is:

- Find an appropriate *primitive definition*.
- *Derive* desired specification (*honest toil*).

*Specification tools* automate this.

## The *Isabelle/HOL* toolbox

$$\begin{tikzcd}
\boxed{\textit{Isabelle/HOL}} \arrow[r, "\text{code generation}"'] \arrow[d, dash] & \boxed{\textit{SML / OCaml / Haskell}} \\
\boxed{\text{specification tools}} &
\end{tikzcd}$$

***inductive predicates*** Knaster-Tarski fixed point theorem

***inductive datatypes*** inductive predicate *plus* *typedef*

***primitive recursion*** primitive recursion combinator

***terminating functions*** explicit function graph *plus* definite choice

## Type classes

Leightweight mechanism for *overloading* plus *abstract specification*.

Example: *algebra*

## Code generator basics

## Code generation paradigms

***proof extraction*** animates proof derivations in the spirit of the Curry-Howard isomorphism (cf. *Coq*)

## Code generation paradigms

***proof extraction*** animates proof derivations in the spirit of the Curry-Howard isomorphism (cf. *Coq*)

***shallow embedding*** identifies term language of logic with term language of target language

## Code generation paradigms

***proof extraction*** animates proof derivations in the spirit of the Curry-Howard isomorphism (cf. *Coq*)

***shallow embedding*** identifies term language of logic with term language of target language

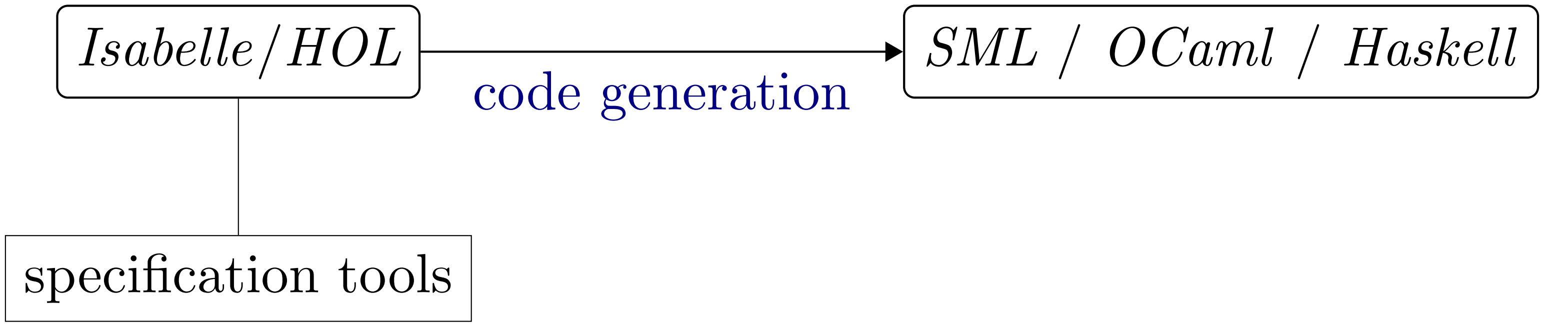
In the *HOL* tradition the second approach is favoured, *Isabelle/HOL* permits proof extraction, though.

## Code generation paradigms

***proof extraction*** animates proof derivations in the spirit of the Curry-Howard isomorphism (cf. *Coq*)

***shallow embedding*** identifies term language of logic with term language of target language

In the *HOL* tradition the second approach is favoured, *Isabelle/HOL* permits proof extraction, though.



## Code generation using shallow embedding

Correctness criterion: semantics of generated target language program $P$ describes a *term rewrite system* where each derivation can be *simulated* in the theory $\Theta$ of the logic:

```
sum [Suc Zero_nat, Suc Zero_nat]
```

```
datatype nat = Suc of nat | Zero_nat;

fun plus_nat (Suc m) n = plus_nat m (Suc n)
  | plus_nat Zero_nat n = n;

fun sum [] = Zero_nat
  | sum (m :: ms) = plus_nat m (sum ms);
```

```
Suc (Suc Zero_nat)
```

$$\begin{tikzcd}
\mathrm{t} \arrow[rr, Leftrightarrow, "\text{identification}"] \arrow[d, dash] & & \mathrm{t} \arrow[d, dash] \\
E_\Theta \arrow[rr, dashed, "\text{code generation}"] \arrow[d] & & E_P \arrow[d] \\
\mathrm{u} \arrow[rr, Leftrightarrow, "\text{identification}"] & & \mathrm{u}
\end{tikzcd}$$

## Code generation using shallow embedding

Correctness criterion: semantics of generated target language program $P$ describes a *term rewrite system* where each derivation can be *simulated* in the theory $\Theta$ of the logic:

`sum [Suc Zero_nat, Suc Zero_nat]`

```
datatype nat = Suc of nat | Zero_nat;

fun plus_nat (Suc m) n = plus_nat m (Suc n)
  | plus_nat Zero_nat n = n;

fun sum [] = Zero_nat
  | sum (m :: ms) = plus_nat m (sum ms);
```

`Suc (Suc Zero_nat)`

$$\begin{tikzcd}
\mathrm{t} \arrow[rr, Leftrightarrow, "\text{identification}"] \arrow[d, dash] & & \mathrm{t} \arrow[d, dash] \\
E_\Theta \arrow[d] \arrow[rr, dashed, "\text{code generation}"] & & E_P \arrow[d] \\
\mathrm{u} \arrow[rr, Leftrightarrow, "\text{identification}"] & & \mathrm{u}
\end{tikzcd}$$

(partial correctness)

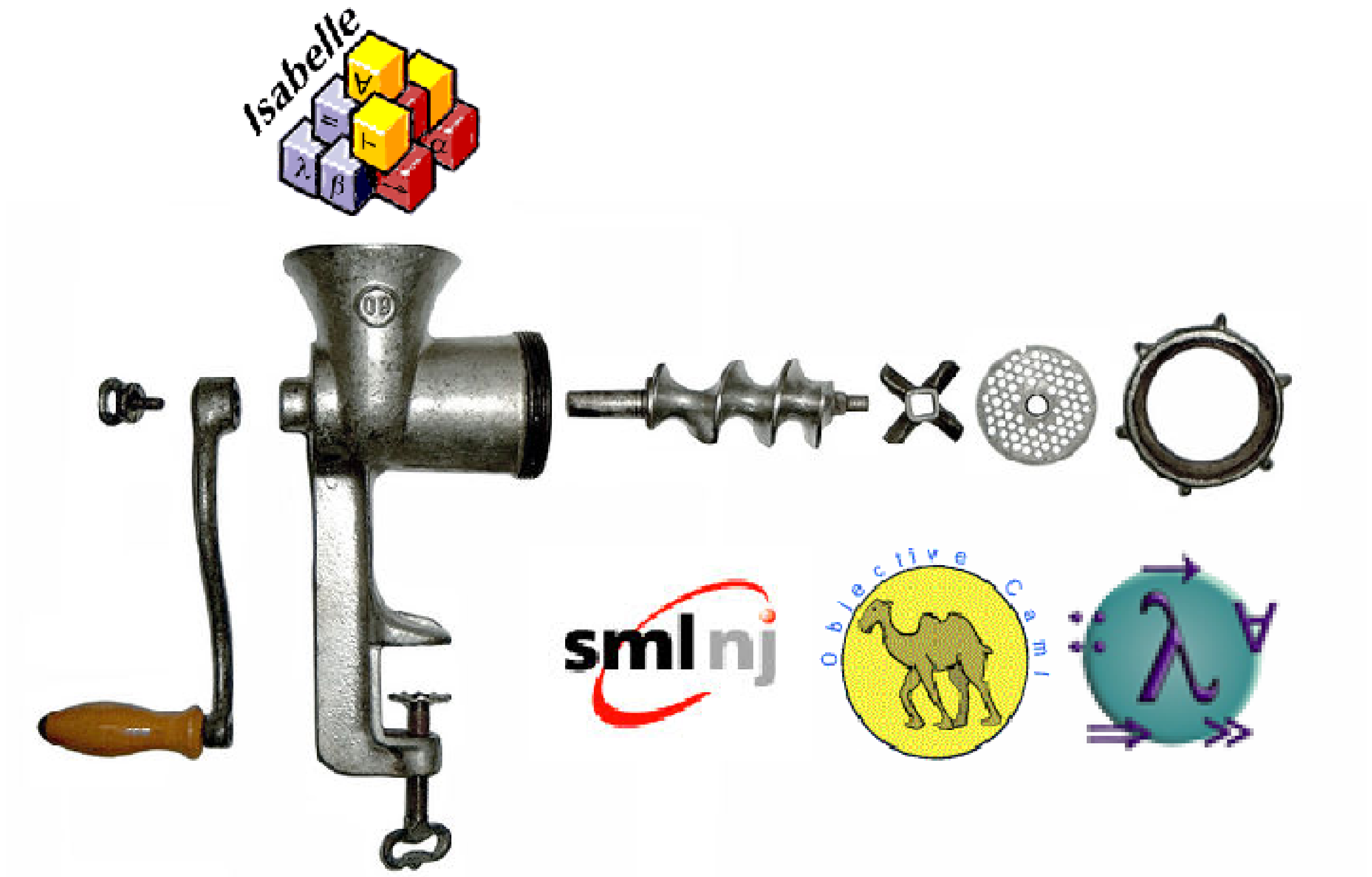
## Examples

- *amortised queues*
- *amortised queues* with poor man's datatype abstraction
- *algebra* with type classes

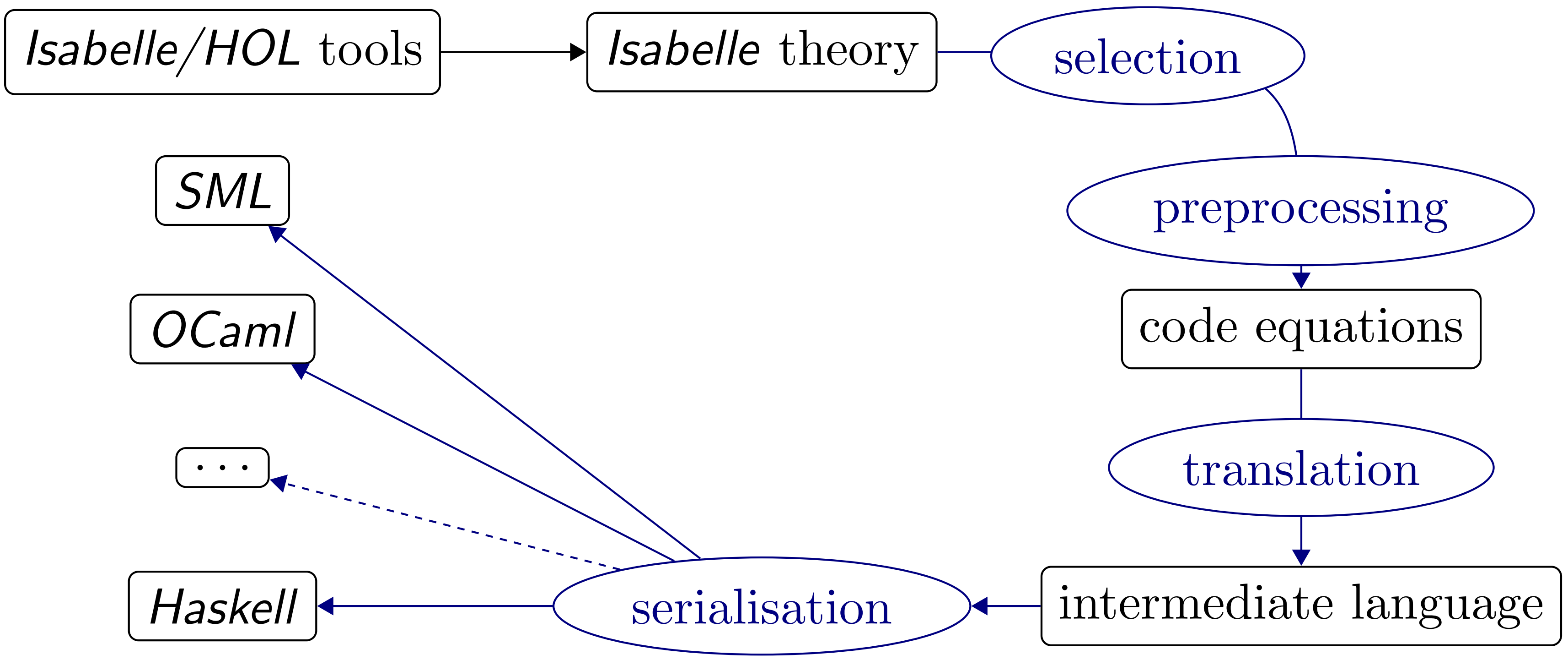
## A closer look at code generation

## How does a code generator look like?

## How does a code generator look like?



## Architecture



## Intermediate language

*purpose*: add “structure” to bare logical equations

## Intermediate language

*purpose*: add "structure" to bare logical equations

$$\mathit{data}\ \kappa\ \overline{\alpha}_k = f_1\ \mathit{of}\ \overline{\tau_1} \mid \ldots \mid f_n\ \mathit{of}\ \overline{\tau_n}$$

$$\begin{array}{l} \mathit{fun}\ f\ ::\ \forall\,\overline{\alpha{::}s_k}.\ \tau\ \ \mathit{where} \\ \quad f\ [\overline{\alpha{::}s_k}]\ \overline{t_1} = t_1 \\ \quad \mid \ldots \\ \quad \mid f\ [\overline{\alpha{::}s_k}]\ \overline{t_k} = t_k \end{array}$$

$$\begin{array}{l} \mathit{class}\ c \subseteq c_1 \cap \ldots \cap c_m\ \mathit{where} \\ \quad f_1 :: \forall\,\alpha.\ \tau_1,\ \ldots,\ f_n :: \forall\,\alpha.\ \tau_n \end{array}$$

$$\begin{array}{l} \mathit{inst}\ \kappa\ \overline{\alpha{::}s_k}\ ::\ c\ \mathit{where} \\ \quad f_1\ [\kappa\ \overline{\alpha{::}s_k}] = t_1,\ \ldots,\ f_n\ [\kappa\ \overline{\alpha{::}s_k}] = t_n \end{array}$$

. . . a kind of "Mini-Haskell"

## Intermediate language

*purpose*: add "structure" to bare logical equations

$$\mathit{data}\ \kappa\ \overline{\alpha}_k = f_1\ \mathit{of}\ \overline{\tau_1}\ |\ \ldots\ |\ f_n\ \mathit{of}\ \overline{\tau_n}$$

$$\begin{array}{l}
\mathit{fun}\ f\ ::\ \forall\,\overline{\alpha::s}_k.\ \tau\ \ \mathit{where} \\
\quad f\ [\overline{\alpha::s}_k]\ \overline{t_1} = t_1 \\
\quad |\ \ldots \\
\quad |\ f\ [\overline{\alpha::s}_k]\ \overline{t_k} = t_k
\end{array}$$

$$\begin{array}{l}
\mathit{class}\ c \subseteq c_1 \cap \ldots \cap c_m\ \ \mathit{where} \\
\quad f_1 :: \forall\,\alpha.\ \tau_1,\ \ldots,\ f_n :: \forall\,\alpha.\ \tau_n
\end{array}$$

$$\begin{array}{l}
\mathit{inst}\ \kappa\ \overline{\alpha::s}_k\ ::\ c\ \ \mathit{where} \\
\quad f_1\ [\kappa\ \overline{\alpha::s}_k] = t_1,\ \ldots,\ f_n\ [\kappa\ \overline{\alpha::s}_k] = t_n
\end{array}$$

…a kind of "Mini-Haskell"

…not "All-gol", but "Thin-gol"

## Selecting

Two degrees of freedom:

**code equations**
*by default:* **definition**, **primrec**, **fun**, **function**
*explicitly:* attribute [*code*]

**datatype constructors**
*by default:* **datatype**, **record**
*explicitly:* **code-datatype**

## Preprocessing

Interface to plugin arbitrary theorem transformations:

**rewrites**

```
simpset
```

**function transformators**

```
theory -> thm list -> thm list
```

## Serialising

Adaption to target-language specifics:

- improving readability and aesthetics of generated code (bools, tuples, lists, . . . )
- gaining efficiency (target-language integers)
- interface with language parts which have no direct counterpart in *HOL* (imperative data structures)

## Serialising

Adaption to target-language specifics:

- improving readability and aesthetics of generated code (bools, tuples, lists, . . . )
- gaining efficiency (target-language integers)
- interface with language parts which have no direct counterpart in *HOL* (imperative data structures)

. . . *but:* know what you are doing!

## Serialising

Adaption to target-language specifics:

- improving readability and aesthetics of generated code (bools, tuples, lists, . . . )
- gaining efficiency (target-language integers)
- interface with language parts which have no direct counterpart in $HOL$ (imperative data structures)

. . . *but:* know what you are doing!

Remember the fundamental rule of software engineering:

*Don't write your own $foo$; if you can, use somebody else's.*

## Serialising

Adaption to target-language specifics:

- improving readability and aesthetics of generated code (bools, tuples, lists, …)
- gaining efficiency (target-language integers)
- interface with language parts which have no direct counterpart in *HOL* (imperative data structures)

… *but:* know what you are doing!

Remember the fundamental rule of software engineering:

*Don't write your own foo; if you can, use somebody else's.*

*foo* $\in$ { operating system, garabage collector, cryptographic algorithm, concurrency framework, theorem prover, … }

## Serialising

Adaption to target-language specifics:

- improving readability and aesthetics of generated code (bools, tuples, lists, …)
- gaining efficiency (target-language integers)
- interface with language parts which have no direct counterpart in $HOL$ (imperative data structures)

… *but:* know what you are doing!

Remember the fundamental rule of software engineering:

*Don't write your own $foo$; if you can, use somebody else's.*

$foo$ $\in$ { operating system, garabage collector, cryptographic algorithm, concurrency framework, theorem prover, … }
$\cup$ {serialisation}

## What remains

**Not mentioned here**

- implementing equality
- code extraction from proofs

**Ongoing work and research**

- turning inductive predicates into equations
- *Haskabelle*: importing *Haskell* files
- Quickcheck
- concept for datatype abstraction

**Further reading**

- Tutorials in the *Isabelle* distribution for functions, code generation etc.
- PhD thesis on code generation (under heavy construction. . . )

. . .

**Happy proving, happy hacking**
**Thanks for your attention**