Code Generation via Higher-Order Rewrite Systems

Florian Haftmann[⋆] and Tobias Nipkow

Technische Universitt Mnchen, Institut fr Informatik
http://www.in.tum.de/~haftmann/
http://www.in.tum.de/~nipkow/

Abstract. We present the meta-theory behind the code generation facilities of Isabelle/HOL. To bridge the gap between the source (higher-order logic with type classes) and the many possible targets (functional programming languages), we introduce an intermediate language, Mini-Haskell. To relate the source and the intermediate language, both are given a semantics in terms of higher-order rewrite systems (HRSs). In a second step, type classes are removed from Mini-Haskell programs by means of a dictionary translation; we prove the correctness of this step. Building on equational logic also directly supports a simple but powerful algorithm and data refinement concept.

1 Introduction and related work

Like many theorem provers, Isabelle/HOL can generate functional programs from recursive functions specified in the logic. Many applications have taken advantage of this feature, e.g. the certified termination analysis tool CeTA [19] or the Quickcheck counterexample search [3]. The initial code generator [2] has since been replaced by a new design [6] that supports a) type classes and b) multiple target languages (currently: SML, OCaml and Haskell). This paper describes the meta-theory underlying this new design. The theoretical contributions can be summarized as follows:

– The formalization of the various stages of the translation between HOL and a functional programming language by means of an intermediate language, Mini-Haskell, with an equational semantics intermediate language, Mini-Haskell, with an equational semantics in terms of higher-order rewrite systems. The equational semantics has two advantages:
  - Correctness of the translation is established in a purely proof theoretic way by relating rewrite systems.
  - Instead of a fixed programming language we cover all functional languages where reduction of pure terms (no side effects, no exceptions, etc) can be viewed as equational deduction. This requirement is met

[⋆] Supported by DFG project NI 491/10-1.

by languages like SML, OCaml and Haskell, and we only generate pure programs. We are also largely independent of the precise nature of the source logic because we focus on its equational sublanguage.

- A non-trivial correctness proof for the replacement of type classes by dictionaries. In contrast to Haskell, where the meaning of type classes is *defined* by such a translation, our starting point is a language with type classes which already has a semantics. Thus we need to show that this translation preserves the semantics.

On a practical level we show how the code generator supports stepwise refinement of both algorithms and data by means of code lemmas that replace less efficient functions and data by more efficient ones in a uniform and logically sound way.

Related work. Many theorem provers support code generation by translating an internal functional language to an external one:

- Coq can generate OCaml, Haskell and Scheme both from constructive proofs and explicitly defined recursive functions [11].
- The language of the theorem prover ACL2 is (almost) a subset of Common Lisp, i.e. the translation is (almost) the identity function [5].
- PVS allows evaluation of ground terms by translation to Common Lisp [4].

The gap between the functional language of the theorem prover and the target programming language varies from system to system and needs to be bridged with care if it is less trivial than in the case of ACL2. We follow common practice (e.g. [10]) and show the correctness of the key part of our translation by a standard mathematical proof.

The outline of the paper is as follows: First we introduce the types and terms of Isabelle/HOL and describe its internal functional language (2). Then we describe how code generation works in principle and introduce the intermediate language to abstract from the details of specific target languages (3). The technical core of the paper is 4, where we prove correctness of a key component of our code generator, the dictionary translation that eliminates Isabelle's type classes from the intermediate language. Finally we describe how the code generator naturally supports algorithm and data refinement (5).

2 Isabelle/HOL

Isabelle/HOL [14] is an interactive proof assistant for higher-order logic (HOL). Isabelle's HOL is a typed λ-calculus with polymorphism and type classes. It is based on the following syntactic entities, where $\overline{e}_n$ denotes the tuple or sequence $e_1, \ldots, e_n$, where the index can be omitted for brevity.

- *classes*: c with a subclass relation $\subseteq$
- *sorts*: $s ::= c_1 \cap \ldots \cap c_n$
- *type constructors*: κ with fixed arities

– *types*: $\tau ::= \kappa\ \overline{\tau} \mid \alpha{::}s$
– *instances*: $\kappa :: \overline{s} \to c$
– *constants*: f with most general type scheme $\forall\, \overline{\alpha{::}s}.\ \tau$
– *terms*: $t ::= f\ [\overline{\tau}] \mid x{::}\tau \mid \lambda x{::}\tau.\ t \mid t_1\ t_2$

Classes correspond to Haskell type classes in their classical formulation [7]. Notationally we treat them as sets of types rather than predicates on types. Sorts are an auxiliary notion that describes (possibly empty) intersections of classes. Types are built up in the usual fashion from (sorted) type variables and type constructors. They form an order-sorted algebra [18]. The type-in-class and type-in-sort judgments $\tau :: c$ and $\tau :: s$ induced by subclasses and instances are defined in 4.

Terms are built up from polymorphic constants, variables, abstractions and applications. Constants are polymorphic and may appear at different types. If f has type scheme $\forall\, \overline{\alpha{::}s}_n.\ \tau$, where $\overline{\alpha}_n$ must be the set of all type variables in τ, any occurrence of f in a term must be of the form $f\ [\overline{\tau}_n]$, where type argument τ_i instantiates type parameter α_i. Well-typedness requires $\tau_i :: s_i$ ($i = 1, \ldots, n$), in which case $f\ [\overline{\tau}_n] :: \tau[\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n]$. The remaining typing rules for $t :: \tau$ are standard. We assume that type/term variables are consistently tagged with their sorts/types.

Isabelle/HOL identifies terms up to $\alpha\beta\eta$ conversion. Terms of the distinguished type *prop* are called propositions; the most interesting propositions in our case are equations built from equality $=$ with type scheme $\forall\,\alpha.\ \alpha \Rightarrow \alpha \Rightarrow prop$,[1] where $\Rightarrow$ is the function space type constructor.

It is important to realize that types are an integral part of the term language and that substitutions can affect both type and term variables. For example, we can have the equations $zero\ [nat] = 0$ and $zero\ [set\ \alpha] = \emptyset$. The presence of types ensures that at most one of these two equations is applicable to a given term: we have $zero\ [set\ nat] = \emptyset$ (by instantiation) but not $zero\ [set\ nat] = 0$.

Isabelle/HOL provides *theories* as containers of logical (and extra-logical) data. Internally, a theory is incrementally enriched with primitive definitions and theorems. Theorems can only be proved by a fixed set of inference rules. It is this notion of theorems as an abstract type that leads to a small trusted (and trustworthy) kernel. To make the kernel accessible to humans, high-level specification and automated proof tools are provided, to which Isabelle's specification and proof language *Isar* provides a coherent interface: Isar theory text consists of a series of *statements*, each of which produces new definitions and/or theorems. For example, this is a specification of queues in Isar: [2]

datatype α *queue* = *Queue* (α *list*)

definition *empty* :: α *queue* **where**
empty = *Queue* []

[1] For Isabelle experts: for our purpose we can and have identified $\equiv$ and $=$.
[2] In concrete Isabelle syntax, types are written postfix: $(\overline{\tau})\ \kappa$ rather than $\kappa\ \overline{\tau}$. Lists have explicit enumeration syntax [...]; cons is written as # and append as @.

fun *enqueue* :: $\alpha \Rightarrow \alpha$ *queue* $\Rightarrow \alpha$ *queue* **where**
  enqueue x (*Queue xs*) = *Queue* (*xs* @ [*x*])

fun *dequeue* :: α *queue* $\Rightarrow \alpha$ *option* $\times$ α *queue* **where**
    dequeue (*Queue* []) = (*None*, *Queue* [])
  | *dequeue* (*Queue* (*x* # *xs*)) = (*Some x*, *Queue xs*)

This illustrates datatype and function definitions. Statements for type class specification and instantiation complete Isabelle/HOL's functional programming language. Here is an example of a lemma with a simple proof (**by** ...):

lemma *dequeue-enqueue-empty*:
  dequeue (*enqueue x empty*) = (*Some x*, *empty*)
by (*simp add*: *empty-def*)

3 Code generation

The Haskell code generated from the *queue* specification contains no surprises:[3]

```
newtype Queue a = Queue [a];

empty :: forall a. Queue a;
empty = Queue [];

dequeue :: forall a. Queue a -> (Maybe a, Queue a);
dequeue (Queue []) = (Nothing, Queue []);
dequeue (Queue (x : xs)) = (Just x, Queue xs);

enqueue :: forall a. a -> Queue a -> Queue a;
enqueue x (Queue xs) = Queue (xs ++ [x]);
```

Superficially this appears like a trivial syntactic transformation of Isar text, but this is misleading: the source of code generation is not the Isar text as typed by the user, but equational theorems proved in the theory. Typically these result from the Isar statements above, but they may also have been proved by the user, which leads to a powerful method for program refinement (see 5). Thus code generation is the translation of a system of equations in the logic to a corresponding program text which implements the same system. A suitable abstract framework to describe these equations are higher-order rewrite systems (HRSs) [13], i.e. rewrite systems on typed λ-terms. Because types are really part of the term language (see the discussion above), we do not need to extend the HRS framework to cover our application. HRSs can serve as the uniform basis for both the source logic and the target programming language. If the code generator preserves the equations from the logic when turning them into programs, partial correctness of the generated programs w.r.t. the original equational theorems is guaranteed. No claims are stated for aspects which have no explicit representation in the logic, in particular termination or runtime complexity.

[3] Isabelle's type *option* is translated to Haskell's isomorphic type `Maybe`, and similarly for lists.

This scenario assumes that our target languages cover the simply-typed λ-calculus, and functions can be specified by equations with pattern matching, which is the case for our targets SML, OCaml and Haskell. Note that code generation addresses only the pure part of those languages: no side effects or exceptions. Hence an equational semantics is justified.

3.1 Intermediate language

There remains one substantial difference between equational theorems and a concrete target language program: a program cannot specify an arbitrary HRS, but imposes syntactic restrictions on the equations. Therefore one task of the code generator is to arrange equational theorems in a fashion such that translation to a target language becomes feasible. This is conveniently shared between all target languages by introducing an intermediate language "Mini-Haskell" with four kinds of statements:

$$\textsf{\textit{data}}\ \kappa\ \overline{\alpha}_k = f_1\ \textsf{\textit{of}}\ \overline{\tau_1} \mid \cdots \mid f_n\ \textsf{\textit{of}}\ \overline{\tau_n}$$

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

$$\begin{array}{l} \textsf{\textit{class}}\ c \subseteq c_1 \cap \cdots \cap c_m\ \textsf{\textit{where}} \\ \quad g_1 :: \forall\,\alpha{::}c.\ \tau_1,\ \ldots,\ g_n :: \forall\,\alpha{::}c.\ \tau_n \end{array}$$

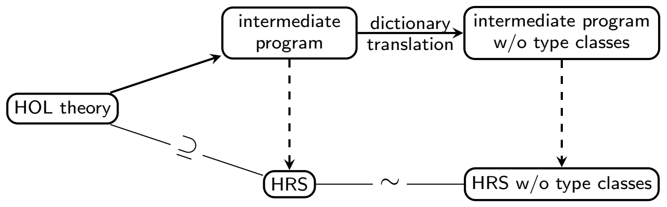
$$\begin{array}{l} \textsf{\textit{inst}}\ \kappa\ \overline{\alpha{::}s_k} :: c\ \textsf{\textit{where}} \\ \quad g_1\ [\kappa\ \overline{\alpha{::}s_k}] = t_1,\ \ldots,\ g_n\ [\kappa\ \overline{\alpha{::}s_k}] = t_n \end{array}$$

The *data* and *fun* statements should be clear. The *class* statement introduces a new class c with superclasses $c_1, \ldots, c_m$ and class methods $g_1, \ldots, g_n$. The *inst* statement instantiates class c with type constructor κ, assuming that the arguments of κ are of the sorts $\overline{s}_k$. Dropping the type variables we can write $\kappa :: \overline{s} \to c$ instead of $\kappa\ \overline{\alpha{::}s} :: c$.

Terms occurring as arguments on left-hand sides of equations in *fun* statements are required to be left-linear constructor patterns, where constructors are constants introduced by *data* statements. The class and instance hierarchy must be *coregular* [16]: for each instance $\kappa :: \overline{s}_i \to c$ and each superclass d of c, there must be exactly one instance $\kappa :: \overline{z}_i \to d$ and each s_j must be a subsort of z_j, i.e. each class in s_j must be a subclass (in the transitive reflexive sense) of some class in z_j. Among other things, this guarantees principal types. These and further standard well-formedness requirements are discussed elsewhere [6].

The equational semantics of a Mini-Haskell program is given by the set of equations in its *fun* and *inst* statements, restricted to well-typed terms. Therefore the translation from a HOL theory T to Mini-Haskell is straightforward: take some (user specified) subset of equational theorems from T, turn them into *fun* and *inst* statements, and enrich that with suitable *data* and *class* statements to form a type correct Mini-Haskell program. The semantic essence, the equations, are not modified, only the syntax is adjusted.

However, a translation to SML or OCaml requires a further step to eliminate type classes via dictionaries:



The upper level of the diagram is the actual translation process, the dashed arrows are the projections to the equations, the lower level are the resulting HRSs. The dictionary translation process is explained in 4. It alters the HRS considerably and we show that its semantics is preserved.

The transformation of an intermediate program to a program in a full-blown SML or Haskell-like target language is again a mere syntactic adjustment and does not change the equational semantics. Note that in this last step we restrict ourselves to partial correctness: if evaluation of a term t in the target language terminates with value v, then $t = v$ is derivable in the equational semantics of the intermediate program. Therefore we are independent of the evaluation strategy of the target language.

4 Dictionary translation

In Isabelle/HOL, types are part of the term language via $f\ [\overline{\tau}]$ and for class methods g these types help to determine if a particular equation for g applies or not. We remove these types and classes by the well-known *dictionary translation* (e.g. [7], which we loosely follow) and show that the semantics is preserved.

The dictionary translation is always applied to a whole program. In the following we avoid carrying around an explicit context but refer implicitly to the declarations in that program: typing of constants $f :: \forall\,\overline{\alpha{::}s}.\ \tau$ (in *fun*, *class* and *data* statements), instances $\kappa :: \overline{s} \to c$, and classes $c \subseteq c_1 \cap \cdots \cap c_m$.

Table 1 describes how dictionary translation operates on intermediate language statements. The central idea is that a statement *class* $c\ \ldots$ translates to a record-like datatype $\delta_c\ \alpha$, a *dictionary type*, which contains fields for all class methods of c. The class methods g_i are defined as projections of the appropriate fields from a dictionary of type $\delta_c\ \alpha$. Correspondingly a statement *inst* $\kappa\ \overline{\alpha{::}s_k} :: c\ \ldots$ translates to a dictionary of type $\delta_c\ (\kappa\ \overline{\alpha{::}s_k})$ containing methods defined in this instance. Superclasses are dealt with by extending dictionary types with additional fields for superclass dictionaries and by defining corresponding projections $\pi_{d \to c}$. Note that the *inst* translation only works because of coregularity (see above): otherwise the required dictionaries for the superclasses might not be well-defined.

statement	statement(s) with dictionaries
$\mathit{data}\ \kappa\ \overline{\alpha}_k\ =$	$\mathit{data}\ \kappa\ \overline{\alpha}_k\ =$
$\quad f_1\ \mathit{of}\ \overline{\tau_1} \mid \cdots \mid f_n\ \mathit{of}\ \overline{\tau_n}$	$\quad f_1\ \mathit{of}\ \overline{\tau_1} \mid \cdots \mid f_n\ \mathit{of}\ \overline{\tau_n}$
$\mathit{fun}\ f\ ::\ \forall\,\overline{\alpha{::}s}_k.\ \tau\ \mathit{where}$	$\mathit{fun}\ f\ ::\ \llparenthesis\forall\,\overline{\alpha{::}s}_k.\ \tau\rrparenthesis\ \mathit{where}$
$\quad f\ [\overline{\alpha{::}s}_k]\ \overline{t_1}\ =\ t_1$	$\quad \llparenthesis f\ [\overline{\alpha{::}s}_k]\ \overline{t_1}\rrparenthesis\ =\ \llparenthesis t_1\rrparenthesis$
$\mid \ldots$	$\mid \ldots$
$\mid f\ [\overline{\alpha{::}s}_k]\ \overline{t_n}\ =\ t_n$	$\mid \llparenthesis f\ [\overline{\alpha{::}s}_k]\ \overline{t_n}\rrparenthesis\ =\ \llparenthesis t_n\rrparenthesis$
$\mathit{class}\ c \subseteq c_1 \cap \cdots \cap c_m\ \mathit{where}$	$\mathit{data}\ \delta_c\ \alpha\ =$
$\quad g_1\ ::\ \forall\,\alpha{::}c.\ \tau_1,$	$\quad \Delta_c\ \mathit{of}\ (\delta_{c_1}\ \alpha)\ \cdots\ (\delta_{c_m}\ \alpha)\ \tau_1\ \cdots\ \tau_n$
$\quad \ldots,$	
$\quad g_n\ ::\ \forall\,\alpha{::}c.\ \tau_n$	$\mathit{fun}\ \pi_{c\to c_1}\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \delta_{c_1}\ \alpha\ \mathit{where}$
	$\quad \pi_{c\to c_1}\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{c_1}$
	$\ldots$
	$\mathit{fun}\ \pi_{c\to c_m}\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \delta_{c_m}\ \alpha\ \mathit{where}$
	$\quad \pi_{c\to c_m}\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{c_m}$
	$\mathit{fun}\ g_1\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \tau_1\ \mathit{where}$
	$\quad g_1\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{g_1}$
	$\ldots$
	$\mathit{fun}\ g_n\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \tau_n\ \mathit{where}$
	$\quad g_n\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{g_n}$
$\mathit{inst}\ \kappa\ \overline{\alpha{::}s}_k\ ::\ c\ \mathit{where}$	$\mathit{fun}\ c_\kappa\ ::\ \llparenthesis\forall\,\overline{\alpha{::}s}_k.\ \delta_c\ (\kappa\ \overline{\alpha{::}s}_k)\rrparenthesis\ \mathit{where}$
$\quad g_1\ [\kappa\ \overline{\alpha{::}s}_k] = t_1,$	$\quad \llparenthesis\kappa\ \overline{\alpha{::}s}_k\ ::\ c\rrparenthesis\ =$
$\quad \ldots,$	$\quad\quad \Delta_c\ \llparenthesis\kappa\ \overline{\alpha{::}s}_k\ ::\ c_1\rrparenthesis\ \cdots\ \llparenthesis\kappa\ \overline{\alpha{::}s}_k\ ::\ c_n\rrparenthesis$
$\quad g_n\ [\kappa\ \overline{\alpha{::}s}_k] = t_n$	$\quad\quad\quad \llparenthesis t_1\rrparenthesis\ \cdots\ \llparenthesis t_n\rrparenthesis$
	if $c \subseteq c_1 \cap \ldots \cap c_n$

Table 1. Dictionary translation for program statements

Both *fun* and *inst* statements are translated by means of three auxiliary functions $\llparenthesis\cdot\rrparenthesis$ on type schemes, terms and type-in-sort judgments:

Translation of type schemes: $\llparenthesis\forall\,\overline{\alpha{::}s}.\ \tau\rrparenthesis$ turns the sorts $\overline{s}$ into additional dictionary type parameters:

$$\begin{aligned}&\llparenthesis\forall\,\alpha_1\ ::\ (c_{1,1} \cap \cdots \cap c_{1,k_1})\ \cdots\ \alpha_n\ ::\ (c_{n,1} \cap \cdots \cap c_{n,k_n}).\ \tau\rrparenthesis\ =\\ &\quad \forall\,\alpha_1 \cdots \alpha_n.\ \delta_{c_{1,1}}\ \alpha_1 \Rightarrow \cdots \Rightarrow \delta_{c_{1,k_1}}\ \alpha_1 \Rightarrow\\ &\qquad \cdots \Rightarrow \delta_{c_{n,1}}\ \alpha_n \Rightarrow \cdots \Rightarrow \delta_{c_{n,k_n}}\ \alpha_n \Rightarrow \tau\end{aligned}$$

Translation of terms: $\llparenthesis t\rrparenthesis$ replaces type arguments by dictionaries:

$$\frac{f\ ::\ \forall\,\alpha_1{::}s_1\ \cdots\ \alpha_n{::}s_n.\ \tau}{\llparenthesis f\ [\tau_1, \ldots, \tau_n]\rrparenthesis = f\ \llparenthesis\tau_1\ ::\ s_1\rrparenthesis\ \cdots\ \llparenthesis\tau_n\ ::\ s_n\rrparenthesis}$$

$$\frac{}{\llparenthesis x{::}\tau\rrparenthesis = x{::}\tau} \qquad \frac{}{\llparenthesis\lambda x{::}\tau.\ t\rrparenthesis = \lambda x{::}\tau.\ \llparenthesis t\rrparenthesis} \qquad \frac{}{\llparenthesis t_1\ t_2\rrparenthesis = \llparenthesis t_1\rrparenthesis\ \llparenthesis t_2\rrparenthesis}$$

Translation of type-in-sort judgments: The translation of a type-in-class judgment $\tau :: c$ amounts to the construction of a dictionary D for type τ. We combine both into one judgment $\tau :: c \rightsquigarrow D$:

$$\frac{\kappa :: \overline{s}_n \to c \quad \tau_1 :: s_1 \rightsquigarrow D_1 \quad \dots \quad \tau_n :: s_n \rightsquigarrow D_n}{\kappa\ \tau_1 \cdots \tau_n :: c \rightsquigarrow c_\kappa\ D_1 \cdots D_n}$$

$$\frac{}{\alpha{::}(c_1 \cap \cdots \cap c_j \cap \cdots \cap c_n) :: c_j \rightsquigarrow \alpha_j}$$

$$\frac{\tau :: d \rightsquigarrow D \quad d \subseteq \ldots \cap c \cap \ldots}{\tau :: c \rightsquigarrow \pi_{d \to c}\ D}$$

$$\frac{\tau :: c_1 \rightsquigarrow D_1 \quad \dots \quad \tau :: c_n \rightsquigarrow D_n}{\tau :: c_1 \cap \cdots \cap c_n \rightsquigarrow D_1 \cdots D_n}$$

The first two rules create dictionaries from c_κs and dictionary variables. By convention we translate a type variable $\alpha{::}s$ where $s = c_1 \cap \cdots \cap c_n$ (and where the c_i are in some canonical order) into dictionary variables $\alpha_1, \ldots, \alpha_n$ such that each α_i represents a dictionary for class c_i. The third rule projects superclass dictionaries. The last rule reduces type-in-sort to type-in-class. It produces a sequence of dictionaries, one for each class c_i in the sort.

Now we define $(\!|\tau :: c|\!) = D$ if $\tau :: c \rightsquigarrow D$ is derivable (and similarly for $(\!|\tau :: s|\!) = \overline{D}$ and $\tau :: s \rightsquigarrow \overline{D}$). There can be multiple derivations of $\tau :: c$ with different Ds, in which case we pick an arbitrary canonical representative of the possible Ds when defining $(\!|\tau :: c|\!)$. Although our system is *coherent* in the sense of [9], a proof is beyond the scope of this paper.

For an example of the complete dictionary translation see Table 2.

An interesting alternative to the classic dictionary translation formalized above is Wehr's representation of dictionaries as ML modules [21]. This avoids polymorphic recursion which may otherwise arise in the translation (although this is rare in practice). Our intermediate language allows polymorphic recursion but the resulting ML code would be rejected by the compiler.

4.1 Correctness

Below we show that dictionary translation preserves reduction semantics. For reasons of space we do not argue preservation of well-typedness: in the worst case we end up with an ill-typed program that the target language compiler will reject. Well-typedness is frequently dealt with in the type class literature (e.g. [20]) and we concentrate on semantic arguments.

First some preliminaries:

Subclasses We follow [15] and eliminate subclasses: classes no longer inherit and each occurrence of a class c in a type or term is replaced by the intersection $c \cap c_1 \cap \cdots \cap c_n$ with all its (transitive) superclasses $c_1, \ldots, c_n$. To simplify the presentation below, we assume that subclassing has been eliminated.

statement	statement(s) with dictionaries
data $\mathbb{N} = \mathrm{Zero} \mid \mathrm{Suc}$ *of* $\mathbb{N}$ *data* $\mathrm{Inf}\ \alpha = \mathrm{Fin}$ *of* $\alpha \mid \infty$ *data* $\mathrm{List}\ \alpha = \mathrm{Nil} \mid \mathrm{Cons}$ *of* $\alpha\ (\mathrm{List}\ \alpha)$	*data* $\mathbb{N} = \mathrm{Zero} \mid \mathrm{Suc}$ *of* $\mathbb{N}$ *data* $\mathrm{Inf}\ \alpha = \mathrm{Fin}$ *of* $\alpha \mid \infty$ *data* $\mathrm{List}\ \alpha = \mathrm{Nil} \mid \mathrm{Cons}$ *of* $\alpha\ (\mathrm{List}\ \alpha)$
class monoid *where* $\mathrm{pls} :: \forall\,\alpha{::}\mathrm{monoid}.\ \alpha \Rightarrow \alpha \Rightarrow \alpha,$ $\mathrm{zero} :: \forall\,\alpha{::}\mathrm{monoid}.\ \alpha$	*data* $\mathrm{monoid}\ \alpha =$ Monoid *of* $(\alpha \Rightarrow \alpha \Rightarrow \alpha)\ \alpha$ *fun* $\mathrm{pls} :: \forall\,\alpha.\ \mathrm{monoid}\ \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha$ *where* $\mathrm{pls}\ (\mathrm{Monoid}\ x\ y) = x$ *fun* $\mathrm{zero} :: \forall\,\alpha.\ \mathrm{monoid}\ \alpha \Rightarrow \alpha$ *where* $\mathrm{zero}\ (\mathrm{Monoid}\ x\ y) = y$
fun $\mathrm{pls}_{\mathbb{N}} :: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$ *where* $\mathrm{pls}_{\mathbb{N}}\ \mathrm{Zero}\ n = n$ $\mid \mathrm{pls}_{\mathbb{N}}\ (\mathrm{Suc}\ m)\ n = \mathrm{Suc}\ (\mathrm{pls}_{\mathbb{N}}\ m\ n)$	*fun* $\mathrm{pls}_{\mathbb{N}} :: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$ *where* $\mathrm{pls}_{\mathbb{N}}\ \mathrm{Zero}\ n = n$ $\mid \mathrm{pls}_{\mathbb{N}}\ (\mathrm{Suc}\ m)\ n = \mathrm{Suc}\ (\mathrm{pls}_{\mathbb{N}}\ m\ n)$
fun $\mathrm{pls}_{\mathbf{Inf}} :: \forall\,\alpha{::}\mathrm{monoid}.$ $\mathrm{Inf}\ \alpha \Rightarrow \mathrm{Inf}\ \alpha \Rightarrow \mathrm{Inf}\ \alpha$ *where* $\mathrm{pls}_{\mathbf{Inf}}\ [\alpha{::}\mathrm{monoid}]\ (\mathrm{Fin}\ a)\ (\mathrm{Fin}\ b) =$ $\mathrm{Fin}\ (\mathrm{pls}\ [\alpha{::}\mathrm{monoid}]\ a\ b)$ $\mid \mathrm{pls}_{\mathbf{Inf}}\ [\alpha{::}\mathrm{monoid}]\ \infty\ b = \infty$ $\mid \mathrm{pls}_{\mathbf{Inf}}\ [\alpha{::}\mathrm{monoid}]\ a\ \infty = \infty$	*fun* $\mathrm{pls}_{\mathbf{Inf}} :: \forall\,\alpha.\ \mathrm{monoid}\ \alpha \Rightarrow$ $\mathrm{Inf}\ \alpha \Rightarrow \mathrm{Inf}\ \alpha \Rightarrow \mathrm{Inf}\ \alpha$ *where* $\mathrm{pls}_{\mathbf{Inf}}\ \alpha\ (\mathrm{Fin}\ a)\ (\mathrm{Fin}\ b) =$ $\mathrm{Fin}\ (\mathrm{pls}\ \alpha\ a\ b)$ $\mid \mathrm{pls}_{\mathbf{Inf}}\ \alpha\ \infty\ b = \infty$ $\mid \mathrm{pls}_{\mathbf{Inf}}\ \alpha\ a\ \infty = \infty$
inst $\mathbb{N} :: \mathrm{monoid}$ *where* $\mathrm{pls}\ [\mathbb{N}] = \mathrm{pls}_{\mathbb{N}},\ \mathrm{zero}\ [\mathbb{N}] = \mathrm{Zero}$	*fun* $\mathrm{monoid}_{\mathbb{N}} :: \mathrm{monoid}\ \mathbb{N}$ $\mathrm{monoid}_{\mathbb{N}} = \mathrm{Monoid}\ \mathrm{pls}_{\mathbb{N}}\ \mathrm{Zero}$
inst $\mathrm{Inf}\ (\alpha{::}\mathrm{monoid}) :: \mathrm{monoid}$ *where* $\mathrm{pls}\ [\mathrm{Inf}\ (\alpha{::}\mathrm{monoid})] = \mathrm{pls}_{\mathbf{Inf}}\ [\alpha{::}\mathrm{monoid}],$ $\mathrm{zero}\ [\mathrm{Inf}\ (\alpha{::}\mathrm{monoid})] =$ $\mathrm{Fin}\ (\mathrm{zero}\ [\alpha{::}\mathrm{monoid}])$	*fun* $\mathrm{monoid}_{\mathbf{Inf}} :: \forall\,\alpha.\ \mathrm{monoid}\ \alpha \Rightarrow$ $\mathrm{monoid}\ (\mathrm{Inf}\ \alpha)$ *where* $\mathrm{monoid}_{\mathbf{Inf}}\ \alpha =$ $\mathrm{Monoid}\ (\mathrm{pls}_{\mathbf{Inf}}\ \alpha)\ (\mathrm{Fin}\ (\mathrm{zero}\ \alpha))$
fun $\mathrm{sum} :: \forall\,\alpha{::}\mathrm{monoid}.\ \mathrm{List}\ \alpha \Rightarrow \alpha$ *where* $\mathrm{sum}\ [\alpha{::}\mathrm{monoid}]\ \mathrm{Nil} = \mathrm{zero}\ [\alpha{::}\mathrm{monoid}]$ $\mid \mathrm{sum}\ [\alpha{::}\mathrm{monoid}]\ (\mathrm{Cons}\ x\ xs) =$ $\mathrm{pls}\ [\alpha{::}\mathrm{monoid}]\ x\ (\mathrm{sum}\ [\alpha{::}\mathrm{monoid}]\ xs)$	*fun* $\mathrm{sum} :: \forall\,\alpha.\ \mathrm{monoid}\ \alpha \Rightarrow$ $\mathrm{List}\ \alpha \Rightarrow \alpha$ *where* $\mathrm{sum}\ \alpha\ \mathrm{Nil} = \mathrm{zero}\ \alpha$ $\mid \mathrm{sum}\ \alpha\ (\mathrm{Cons}\ x\ xs) =$ $\mathrm{pls}\ \alpha\ x\ (\mathrm{sum}\ \alpha\ xs)$
fun $\mathrm{example} :: \mathrm{Inf}\ \mathbb{N}$ *where* $\mathrm{example} = \mathrm{sum}\ [\mathrm{Inf}\ \mathbb{N}]$ $(\mathrm{Cons}\ (\mathrm{Fin}\ \mathrm{Zero})\ (\mathrm{Cons}\ \infty\ \mathrm{Nil}))$	*fun* $\mathrm{example} :: \mathrm{Inf}\ \mathbb{N}$ *where* $\mathrm{example} = \mathrm{sum}\ (\mathrm{monoid}_{\mathbf{Inf}}\ \mathrm{monoid}_{\mathbb{N}})$ $(\mathrm{Cons}\ (\mathrm{Fin}\ \mathrm{Zero})\ (\mathrm{Cons}\ \infty\ \mathrm{Nil}))$

Table 2. Dictionary translation example (for succintness some type arguments $[\overline{\tau}]$ are not printed explicitly)

Constructor terms We call a term r a *constructor term* if it only consists of fully applied constants introduced by *data* statements. Since *data* statements do not constrain the type variables (i.e. constrain them implicitly by the empty sort) we have $(\!|f|\!) = f$ for all data constructors, and hence $(\!|r|\!) = r$.

Terms and substitutions We make use of the notation $C[t]$ for terms where the context C is a term with a "hole" that is filled with a subterm t. Because $(\!|\cdot|\!)$ is a homomorphism on terms we have $(\!|C[t]|\!) = (\!|C|\!)[(\!|t|\!)]$.
Given a substitution σ we define $(\!|\sigma|\!)$ to be the substitution σ' such that $\sigma'(x) = (\!|\sigma(x)|\!)$ for all x. By induction on term t we obtain $(\!|\sigma(t)|\!) = (\!|\sigma|\!)((\!|t|\!))$.

Rewriting An HRS E is a set of rewrite rules $l = r$ where l and r are λ-terms of the same type. The rewrite relation $E \vdash t \longrightarrow t'$ holds iff $t = C[\sigma(l)]$ and $t' = C[\sigma(r)]$ for suitable C, σ and $l = r$ in E [13].

In the proof below we have to argue about the order in which equations are applied. These arguments become particularly transparent if we appeal to a well-known strategy, lazy evaluation as in Haskell. This is admissible for the following reasons. We focus our attention on the target languages SML, OCaml, Haskell. They impose a sequentialization of our rewrite systems at the end of the translation chain: overlapping equations are disambiguated by the order in which they occur. For example, $f(True) = e_1,\ f(x) = e_2$ is equivalent to $f(True) = e_1,\ f(False) = e_2$ in the target language. Thus we may as well assume that all function definitions in a program are non-overlapping to start with. Therefore the notion of lazy evaluation is well-defined, for example as given by the Haskell semantics. Now observe that in the theorem below we consider only reductions to normal forms. Hence Haskell subsumes SML or OCaml: if SML or OCaml evaluation finds a normal form, so does Haskell.

In the following we are given a fixed program P and its dictionary translation P_Δ. Let E and E_Δ be the the set of equations contained in *fun* and *inst* statements of P and P_Δ. We will now study the reduction behavior of E and E_Δ, i.e. view them as HRSs.

Theorem 1 (Correctness). *If all functions in P are defined by non-overlapping sets of equations, t is well-typed w.r.t. P, and r is a constructor term, then $E \vdash t \longrightarrow^* r$ iff $E_\Delta \vdash (\!|t|\!) \longrightarrow^* r$.*

Proof We start by comparing the structure of equations in both systems:

	equations E	equations E_Δ	
f	$f\ [\overline{\alpha{::}s}]\ \overline{t} = t$	$f\ \overline{\alpha}\ \overline{(\!|t|\!)} = (\!|t|\!)$	f_Δ
g	$g\ [\kappa\ \overline{\beta{::}s}] = t$	$c_\kappa\ \overline{\beta} = \Delta_c\ \ldots\ (\!|t|\!)\ \ldots$	Δ_I
		$g\ (\Delta_c\ \overline{x}) = x$	Δ_E

Throughout this proof f will always represent a constant introduced by a *fun* statement and g a class method. Equations in E can be partitioned into those defining fs and those defining gs. In E_Δ, equations of kind f_Δ correspond to equations of kind f; equations of kind g have no direct counterpart, but are split into equations of kind Δ_I producing a particular Δ_c and equations of kind Δ_E

consuming a particular Δ_c. Our proof will work in two steps: first, we establish an intermediate system which joins the Δ_E / Δ_I equations of E_Δ; then we show that this intermediate system behaves like E.

Because r is a constructor term, it is in normal form. Hence we may restrict our attention to reductions following a lazy evaluation strategy. First we show that in a lazy reduction sequence $E_\Delta \vdash (\!|t|\!) \longrightarrow^* r$, each Δ_I step is immediately followed by its corresponding Δ_E step. We note that in $(\!|t|\!)$, constants $c_\mathcal{K}$ can only occur in subterms of the form $h \ldots (c_\mathcal{K} \ldots)$, where h is a constant, and that this is preserved in each reduction step: the right-hand side of each reduction rule is either a single variable of non-dictionary type (Δ_E rules) or $(\!|t|\!)$ (f_Δ rules) or $\Delta_c\ (\!|t_1|\!) \ldots (\!|t_n|\!)$ (Δ_I rules, remember we have no superclasses). Looking at the rules of E_Δ we find that f_Δ and Δ_I rules do not require their dictionary arguments to be evaluated. Hence lazy evaluation will unfold f and $c_\mathcal{K}$ before unfolding their dictionary arguments. Finally we consider evaluation of a redex $c_\mathcal{K} \ldots$ inside $h \ldots (c_\mathcal{K} \ldots)$. As we just argued (by laziness) the h cannot be an f or another (not necessarily different) $c'_{\kappa'}$. Hence it must be a g, whose only dictionary parameter is the $c_\mathcal{K} \ldots$. Thus we now have a new redex $g\ (\Delta_c \ldots)$ which lazy evaluation will reduce by the corresponding Δ_E rule $g\ (\Delta_c\ \overline{x}) = x$.

We have shown that lazy evaluation automatically ensures that Δ_I and Δ_E steps always occur pairwise. Thus it is legitimate to treat those pairs as fixed singleton steps. Let $(\!|E|\!)$ (a suggestive name!) be the system which results from E_Δ by merging the corresponding Δ_I / Δ_E equations into equations of a new kind g_Δ (see below). By construction we have:

$$(\!|E|\!) \vdash (\!|t|\!) \longrightarrow^* r \ \text{ iff } \ E_\Delta \vdash (\!|t|\!) \longrightarrow^* r$$

The relationship between E and $(\!|E|\!)$ is very close and justifies the name $(\!|E|\!)$ because we have $(l = r) \in E$ iff $((\!|l|\!) = (\!|r|\!)) \in (\!|E|\!)$:

equations E		equations $(\!|E|\!)$	
f	$f\ [\overline{\alpha::s}]\ \overline{t} = t$	$f\ \overline{\alpha}\ \overline{(\!|t|\!)} = (\!|t|\!)$	f_Δ
g	$g\ [\kappa\ \overline{\beta::s}] = t$	$g\ (c_\mathcal{K}\ \overline{\beta}) = (\!|t|\!)$	g_Δ

The remainder of the proof shows

$$E \vdash t \longrightarrow^n r \ \text{ iff } \ (\!|E|\!) \vdash (\!|t|\!) \longrightarrow^n r$$

by induction on n. The case $n = 0$ is trivial. The induction step works according to the following picture:

$$\begin{tikzcd}
E \vdash & u \arrow[r, dashed] \arrow[d, leftrightarrow] & t \arrow[r, "n"] \arrow[d, leftrightarrow] & r \arrow[d, leftrightarrow] \\
(\!|E|\!) \vdash & (\!|u|\!) \arrow[r, dashed] & (\!|t|\!) \arrow[r, "n"] & r
\end{tikzcd}$$

The right part (solid lines) is the induction hypothesis. For the induction step it remains to prove the following implications:

1. $E \vdash u \longrightarrow t$ implies $(\!|E|\!) \vdash (\!|u|\!) \longrightarrow (\!|t|\!)$
2. $(\!|E|\!) \vdash (\!|u|\!) \longrightarrow v$ implies $\exists\, t.\ (\!|t|\!) = v \wedge E \vdash u \longrightarrow t$

Proof of 1. The rewrite step $E \vdash u \longrightarrow t$ takes place at a certain redex in u which is a substitution instance $\sigma(l)$ of the left-hand side l of an equation $l = r$ in E. Hence $u = C[\sigma(l)]$ and $t = C[\sigma(r)]$. Therefore

$$(\!|u|\!) = (\!|C[\sigma(l)]|\!) = (\!|C|\!)[(\!|\sigma(l)|\!)] = (\!|C|\!)[(\!|\sigma|\!)((\!|l|\!))] \text{ and}$$
$$(\!|t|\!) = (\!|C[\sigma(r)]|\!) = (\!|C|\!)[(\!|\sigma(r)|\!)] = (\!|C|\!)[(\!|\sigma|\!)((\!|r|\!))].$$

Thus $(\!|E|\!) \vdash (\!|u|\!) \longrightarrow (\!|t|\!)$ using equation $(\!|l|\!) = (\!|r|\!)$ in $(\!|E|\!)$.

Proof of 2. The rewrite step $(\!|E|\!) \vdash (\!|u|\!) \longrightarrow v$ implies $(\!|u|\!) = C'[\sigma'((\!|l|\!))]$ and $v = C'[\sigma'((\!|r|\!))]$ for suitable C', σ' and $(\!|l|\!) = (\!|r|\!)$ in $(\!|E|\!)$. From C' and σ' we obtain C and σ by reconstructing type arguments from dictionaries. This reconstruction is the inverse of function $(\!|\tau :: s|\!)$. Essentially it turns $c_{\mathcal{K}}$ back into κ and α_j into α. Then we have $u = C[\sigma(l)]$, $(\!|u|\!) = (\!|C|\!)[(\!|\sigma(l)|\!)]$ and $v = (\!|C|\!)[(\!|\sigma(r)|\!)]$. Defining $t = C[\sigma(r)]$ we obtain the desired $E \vdash u \longrightarrow t$ (using equation $l = r$ in E) and $(\!|t|\!) = (\!|C[\sigma(r)]|\!) = (\!|C|\!)[(\!|\sigma(r)|\!)] = v$. □

Although this proof restricts to non-overlapping equations, we believe that this theorem also holds without the restriction.

5 Program and data refinement

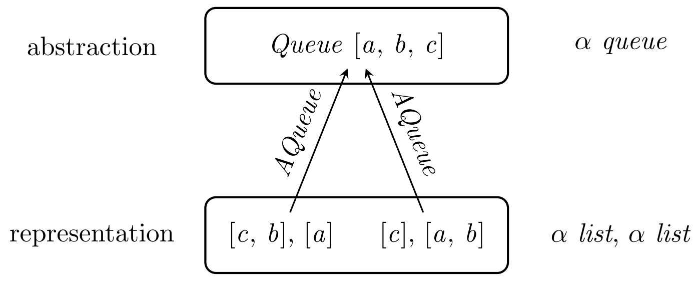
Program refinement is the replacement of less efficient algorithms and data structures by more efficient ones. We show how the code generator supports both activities with surprising ease because we can generate code from arbitrary equational theorems, not just definitions. Replacing one algorithm by another is in fact trivial. For example, implementing the standard recursive definition of list reversal *rev* (which takes quadratic time and space) by a linear, tail recursive one *itrev* of type α *list* $\Rightarrow$ α *list* $\Rightarrow$ α *list* simply requires a proof of the lemma *rev xs* = *itrev xs* []. Notifying the code generator of this lemma (which needs to be done explicitly) has the effect that from then on (for code generation) the original equations for *rev* are dropped and *rev xs* = *itrev xs* [] is used instead.

More interesting is a change of data structures, also known as *data refinement* [8]. The key is the insight that ***data*** statements of our intermediate language do not contribute to a program's equational semantics, by definition. Hence we can replace one datatype by another as long as we can still express our functions by pattern matching over the new rather than the old type.

Our approach to data refinement is best explained by an example. The queues presented in 2 are the natural abstract specifications that one can reason about in a straightforward manner. However, the generated code is suboptimal; a more efficient implementation would use amortized queues [17], which are pairs of lists. The queue corresponding to such a pair is obtained by reversing the first list and appending it to the second:

definition *AQueue* :: α *list* $\Rightarrow$ α *list* $\Rightarrow$ α *queue* **where**
  AQueue xs ys = *Queue* (*ys* @ *rev xs*)

This is a classic case of data refinement and $AQueue$ is the abstraction function:



For the primitive queue operations we can now prove alternative equations which perform pattern matching on $AQueue$ rather than $Queue$:

$$empty = AQueue\ []\ []$$

$$enqueue\ x\ (AQueue\ xs\ ys) = AQueue\ (x\ \#\ xs)\ ys$$

$$\begin{array}{l} dequeue\ (AQueue\ xs\ []) = \\ (\mathit{if}\ null\ xs\ \mathit{then}\ (None,\ AQueue\ []\ [])\ \mathit{else}\ dequeue\ (AQueue\ []\ (rev\ xs))) \end{array}$$

$$dequeue\ (AQueue\ xs\ (y\ \#\ ys)) = (Some\ y,\ AQueue\ xs\ ys)$$

We instruct the code generator to view $AQueue$ as a constructor. Now it produces the following Haskell program:

```
data Queue a = AQueue [a] [a];

empty :: forall a. Queue a;
empty = AQueue [] [];

dequeue :: forall a. Queue a -> (Maybe a, Queue a);
dequeue (AQueue xs (y : ys)) = (Just y, AQueue xs ys);
dequeue (AQueue xs []) =
  (if null xs then (Nothing, AQueue [] [])
    else dequeue (AQueue [] (reverse xs)));

enqueue :: forall a. a -> Queue a -> Queue a;
enqueue x (AQueue xs ys) = AQueue (x : xs) ys;
```

Clients of the abstract type α *queue* can continue to use the primitive operations *empty*, *enqueue* and *dequeue* and reason in terms of the abstract constructor *Queue*. Upon code generation, the primitive operations will now be implemented in terms of the concrete constructor *AQueue*. If a client has broken the abstraction and has used *Queue* for pattern matching in some function *f*, code generation for *f* will fail because *Queue* is no longer a constructor. Isabelle already objects, but even if it did not, Haskell would. For example, code generation for this perfectly good function definition fails:

$$\begin{array}{l} \textbf{fun}\ peek :: \alpha\ queue \Rightarrow \alpha\ option\ \textbf{where} \\ \quad peek\ (Queue\ []) = None \\ \mid peek\ (Queue\ (x\ \#\ xs)) = Some\ x \end{array}$$

Of course we can view *peek* as another primitive operation on queues and prove the following executable equation in terms of *AQueue*:

lemma *peek-AQueue* [*code*]:
peek (*AQueue xs ys*) = (*if null ys then*
(*if null xs then None else Some* (*last xs*)) *else Some* (*hd ys*))

A considerably larger example are Lochbihler's finite functions and their refinement to executable code [12].

Related work. ACL2 allows replacement of subterms at code generation time with other provably equal subterms [5]. Coq also allows replacement of one function by another at code generation time but this is completely unchecked. Neither system supports data refinement in the way we showed in our queue example.

6 Conclusion

We have presented the essentials behind Isabelle/HOL's code generator: it transforms a system of equations into a program in an intermediate language capturing the essence of functional programming languages. Type classes are supported and we proved that dictionary translation preserves their semantics. Program development in the form of algorithm and data refinement is supported by the underlying equational logic.

Recently the scope of the code generator has been extended towards logic programming [1]. Inductive predicates are translated to recursive functions and the equivalence is proved automatically within HOL. The code generator itself is left untouched.

Acknowledgement. We sincerely thank Alex Krauss and the referees for their many comments and suggestions.

References

1. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: TPHOLs '09: Proc. of the 22th International Conference on Theorem Proving in Higher Order Logics, *LNCS*, vol. 5674. Springer (2009)
2. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Types for Proofs and Programs (TYPES 2000), *LNCS*, vol. 2646. Springer (2002)
3. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: SEFM '04: Proc. of the Software Engineering and Formal Methods, Second International Conference. IEEE Computer Society (2004)
4. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating PVS specifications. Tech. rep., Computer Science Laboratory, SRI International (2001)

5. Greve, D.A., Kaufmann, M., Manolios, P., Moore, J.S., Ray, S., Ruiz-Reina, J.L., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. Journal of Functional Programming **18**(1), 15–46 (2007)
6. Haftmann, F.: Code generation from specifications in higher order logic. Ph.D. thesis, Technische Universität München (2009)
7. Hall, C., Hammond, K., Peyton Jones, S., Wadler, P.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems **18**(2) (1996)
8. Jones, C.B.: Systematic Software Development using VDM, second edn. Prentice Hall International (1990)
9. Jones, M.P.: Qualified types: Theory and practice. Ph.D. thesis, University of Nottingham (1994)
10. Letouzey, P.: Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq. Ph.D. thesis, Université Paris-Sud (2004)
11. Letouzey, P.: Coq Extraction, an Overview. In: Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008, *LNCS*, vol. 5028. Springer (2008)
12. Lochbihler, A.: Formalising FinFuns - generating code for functions as data from Isabelle/HOL. In: Proc. of the 22nd International Conference of Theorem Proving in Higher Order Logics, *LNCS*, vol. 5674. Springer (2009)
13. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theor. Comput. Sci. **192**, 3–29 (1998)
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
15. Nipkow, T., Prehofer, C.: Type checking type classes. In: Proc. 20th ACM Symp. Principles of Programming Languages. ACM Press (1993)
16. Nipkow, T., Prehofer, C.: Type reconstruction for type classes. J. Functional Programming **5**(2), 201–224 (1995)
17. Okasaki, C.: Catenable double-ended queues. In: Proc. Int. Conf. Functional Programming (ICFP '97). ACM Press (1997)
18. Schmidt-Schauß, M.: Computational aspects of an order-sorted logic with term declarations. LNAI 395. Springer (1989)
19. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: TPHOLs '09: Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics, *LNCS*, vol. 5674. Springer (2009)
20. Wehr, S.: ML modules and Haskell type classes: A constructive comparison. Master's thesis, Albert-Ludwigs-Universität, Freiburg (2005)
21. Wehr, S., Chakravarty, M.M.T.: ML modules and Haskell type classes: A constructive comparison. In: Proc. ASIAN Symposium on Programming Languages and Systems, *LNCS*, vol. 5356. Springer (2008)