A Code Generator Framework for Isabelle/HOL

Florian Haftmann* and Tobias Nipkow

Institut für Informatik, Technische Universität München
http://www.in.tum.de/{~haftmann,~nipkow}

Abstract. We present a code generator framework for Isabelle/HOL. It formalizes the intermediate stages between the purely logical description in terms of equational theorems and a programming language. Correctness of the translation is established by giving the intermediate languages (a subset of Haskell) an equational semantics and relating it back to the logical level. To allow code generation for SML, we present and prove correct a (dictionary-based) translation eliminating type classes. The design of our framework covers different functional target languages.

1 Introduction and related work

Executing formal specifications is a well-established topic and many theorem provers support this activity by generating code in a standard programming language from a logical description, typically by translating an internal functional language to an external one:

- Coq [15] can generate OCaml both from constructive proofs and explicitly defined recursive functions.
- Both Isabelle/HOL [1] and HOL4 generate SML code. In the case of Isabelle this code is also used for counter example search [2].
- The language of the theorem prover ACL2 is a subset of Common Lisp.
- PVS allows evaluation of ground terms by translation to Common Lisp [4].

Though code generation forms an increasingly vital part of many theorem provers, its functionality is often not formalized and must be trusted. In the case of ACL2 this is justified because its logic is a subset of Common Lisp, but the addition of single-threaded objects [3], which allow destructive updates, breaks this direct correspondence. The treatment of destructive updates in the Common Lisp code generated by PVS has been proved correct [14], although PVS code generation in general appears to have not been formalized. Code generation for Coq is studied in great detail, e.g. [7]. One of the key differences to our work is that Coq is already closer to a programming language than HOL, for example because it has inductive types built in. Code generation for Isabelle/HOL is described by Berghofer and Nipkow [1], who consider in particular the generation of Prolog-like code from inductive definitions, which we ignore, but who ignore

* Supported by DFG project NI 491/10-1

the correctness question, dismiss the purely functional part as straightforward, and do not cover type classes at all.

The key contributions of our paper can be summarized as follows:

- A framework that formalizes the intermediate stages between the purely logical description in terms of equational theorems and a programming language. Giving the intermediate languages (fragments of Haskell and SML) an equational semantics has two advantages:
  - Correctness of the translation is established in a purely proof theoretic way by relating equational theories.
  - 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: Given a pure program P and a pure term t, we assume that *if* t reduces to a value u, then t and u are equivalent modulo the equations of P. This requirement is met by languages like SML, OCaml and Haskell, and we only generate pure programs.
- A first treatment of code generation for type classes. Although we follow Haskell's dictionary passing style, the key difference is that in Haskell, type classes are defined by translation, whereas our starting point is a language with type classes which already has a meaning. Thus we need to show that the translation is correct.
- A constructive proof how to transform a set of equations conforming to some implementability restrictions into a program.

Note that throughout this paper type classes refer to their classical formulation (see, for example, [6]). Note further that we can lump Haskell and SML together because we will only guarantee partial correctness of the generated code.

Somewhat related but quite different is the work by Meyer and Wolff [9]. They translate between shallow embeddings of functional programs in HOL by means of tactics, whereas we justify the translation once and for all, but do this outside HOL. There are many further differences, like our thorough treatment of type classes.

After a sketch of the system architecture (§2), we introduce Isabelle's language of terms and types (§3), accompanied by an abstract Haskell-like programming language and its equational semantics (§4). Type classes are eliminated in favor of dictionaries and this translation into an SML-like sublanguage is proved correct (§5). We characterize implementable equation systems and give a correct translation (via a constructive proof) into the Haskell-like language (§6). A note on handling equality (§7) concludes our presentation.

2 System architecture

Conceptually, the process of code generation is split up in distinct steps:

1. Out of the vast collection of theorems proven in a formal *theory*, a reasonable subset modeling an equation system is *selected*.
2. The selected theorems are subjected to a deductive *preprocessing* step resulting in a structured collection of *defining equations*.
3. These are *translated* into a Haskell-like intermediate language.
4. From the intermediate language the final code in the *target language* is *serialized*.

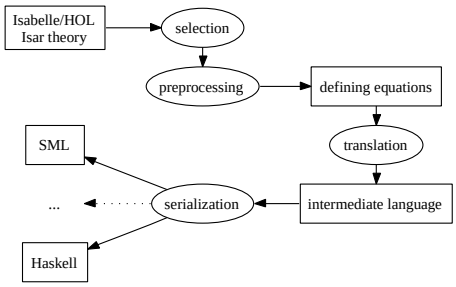

Fig. 1. code generation overview

Only the two last steps are carried out outside the logic; by making this layer as thin as possible, the amount of code to trust is kept minimal.

3 The Isabelle framework

The logic Isabelle/HOL [10] is an extension of Isabelle's *meta logic*, a minimal higher-order logic of simply typed lambda terms [12]. *Propositions* are terms of a distinguished type *prop*. *Theorems* are propositions constructed via some basic inference rules. Isabelle provides equality $\equiv :: \alpha \to \alpha \to \mathit{prop}$ and identifies terms up to $\alpha\beta\eta$ conversion. Isabelle's term language is an order-sorted [13] typed λ-calculus with schematic polymorphism[1]:

sorts $s ::= c_1 \cap \ldots \cap c_n$
types $\tau ::= \kappa\,\overline{\tau}_m \mid \tau_1 \to \tau_2 \mid \alpha :: s$ — [2]
terms $t ::= f\,[\overline{\tau}_n] \mid x :: \tau \mid \lambda x :: \tau.\, t \mid t_1\, t_2$

The notation $\overline{u}_n$ denotes the tuple or sequence $u_1, \ldots, u_n$; writing $\overline{u}$ means the length is irrelevant. The atomic symbols are *classes* c, *type constructors* κ, *type variables* α, *constants* f, and *variables* x. Classes classify types. Sorts are intersections of finitely many classes. Type variables are qualified by sorts, which restrict possible type instantiations. The System F-like notation $f\,[\overline{\tau}_n]$ is explained below.

Note that our terms and types have type and sort information attached to each occurrence of a variable. This simplifies the context in the well-formedness judgments to come but means that we tacitly assume that these attachments are consistent within a term or type. The empty intersection represents the universal sort which every type belongs to. Constants have *generic types* of the form $\forall \overline{\alpha :: s}_k.\, \tau$, where $\{\alpha_1 \ldots \alpha_k\}$ is the set of *all* type variables in τ.[3] If all s_i are empty, we write $\forall \overline{\alpha}_k.\, \tau$ instead.

[1] Hindley-Milner let-polymorphism without a local let
[2] The function arrow $\to$ is just a binary type constructor: $\text{`}\to\text{`}\,\tau_1\,\tau_2$
[3] Whenever a type scheme is given, this restriction is implicit.

A *context* Γ is a four-tuple $(TYP, SUP, \Sigma, \Omega)$ of (partial) functions: $TYP\,\kappa = k$ means that type constructor κ has arity k, $SUP\,c = \overline{c}_q$ that $\{c_1, \ldots, c_q\}$ is exactly the set of direct superclasses of class c, and $\Omega\,f = \forall\overline{\alpha :: s}_k.\,\tau$ that constant f has generic type $\forall\overline{\alpha :: s}_k.\,\tau$. The behavior of type constructors w.r.t. classes is expressed by Σ: $\Sigma(\kappa, c) = \overline{s}_k$ is called an *instance* and means that $\kappa\,\overline{\tau}_k$ is of class c if each τ_i is of sort s_i. Contexts must not contain SUP-cycles. *Notation*: Every Γ is implicitly of the form $(TYP, SUP, \Sigma, \Omega)$; we have the following judgments:

- Well-formed sorts: $\Gamma \vdash s$ means that $s = c_1 \cap \ldots \cap c_n$ is well-formed, i.e. $SUP\,c_i$ is defined for all i.
- Subsort relation: $\Gamma \vdash s \subseteq s'$ means that $s = c_1 \cap \ldots \cap c_n$ is a subsort of $s' = c'_1 \cap \ldots \cap c'_m$, i.e. for all i there is a j such that c'_i is a (not necessarily direct) superclass (w.r.t. SUP) of c_j.
- Well-formed types: $\Gamma \vdash \tau$ means that all type constructors κ in τ are applied to the required number of arguments $TYP\,\kappa$.
- Well-sorted types $\Gamma \vdash \tau :: s$ and well-typed terms $\Gamma \vdash t :: \tau$.

Precise definitions can be found elsewhere [11]. The definition of $\Gamma \vdash t :: \tau$ is standard except for constants: If $\Omega\,f = \forall\overline{\alpha :: s}_k.\,\tau$ and $\Gamma \vdash \tau_i :: s_i$ for all i then $\Gamma \vdash f\,[\overline{\tau}_k] :: \tau[\overline{\tau}_k/\overline{\alpha}_k]$. Each occurrence of a constant in a term carries the instantiation of its generic type to resolve ambiguities due to polymorphism and overloading (for brevity we will leave them out whenever they are obvious).

To guarantee principal types, Isabelle enforces coregularity of contexts:

Definition 1 *Γ is* coregular *if*
$\Sigma(\kappa, c) = \overline{s}_k$ *implies* $\forall d \in SUP\,c.\,\exists\overline{s'}_k.\,\Sigma(\kappa, d) = \overline{s'}_k \wedge \forall 1 \leq i \leq k.\,\Gamma \vdash s_i \subseteq s'_i$

It will turn out that coregularity is also essential for dictionary construction (§5).

4 From logic to programs

On the surface level, Isabelle/HOL offers all the constructs one finds in functional programming: data types, recursive functions, classes, and instances. But in the logical kernel, all we have are contexts and theorems. Clearly, equational theorems can represent programs. For example, a suitable context with natural numbers, pairs, class c and a constant $f :: \forall\alpha :: c.\,\alpha \to \alpha$, and two equations

$$f\,[nat]\,(n :: nat) \equiv n + 1$$
$$f\,[\alpha :: c \times \beta :: c]\,(x :: \alpha, y :: \beta) \equiv (f\,[\alpha]\,x, f\,[\beta]\,y)$$

have a straightforward interpretation as a Haskell program. But not every set of equations corresponds to a program, even if they superficially look like one:

$$f\,[bool \times bool]\,(x :: bool, y :: bool) \equiv (\neg x, \neg y)$$
$$f\,[nat \times nat]\,(x :: nat, y :: nat) \equiv (y, x)$$

is also definable in Isabelle, but realization of this overloaded system would demand a sophisticated type class system beyond Haskell 1.0. There are many further restrictions on what is executable. A major part of our task is to establish a precise link between a subset of our logic and an executable language.

Definition 2 *An equation of the form* $f\,[\overline{\tau}_k]\,\overline{t}_m \equiv t$ *is a* defining equation for f *with* type arguments $[\overline{\tau}_k]$, arguments $\overline{t}_m$ *and* right hand side t *iff*

1. *all variables of* t *occur in* $\overline{t}_m$,
2. *all type variables of* t *occur in* $[\overline{\tau}_k]$,
3. *no variable occurs more than once in* $\overline{t}_m$ *(left-linearity),*
4. *all* $\overline{t}_m$ *are pattern terms, where a pattern term is either a variable or a constant applied to a list of pattern terms.*

Due to these syntactic restrictions, defining equations resemble the kind of equations admissible in function definitions in functional programming languages; we will use defining equations as an abstract "executable" view on the logic:

Definition 3 *An* equation system *is a pair* (Γ, E) *where* E *is a set of defining equations which are well-typed in the context* Γ*. We write* $(\Gamma, E) \vdash s \equiv t$ *iff* $s \equiv t$ *follows by equational logic and* $\beta\eta$*-conversion.*

Note that Γ is necessary to restrict derivations to well-typed terms.

An abstract programming language

Our aim is to implement a given equation system as a program in a functional programming language. Therefore we introduce a Haskell-like intermediate language which captures the essence of target languages and give it a semantics as an equation system (equational reasoning is a common device in the Haskell community [5]). The language forms a bridge between logical and operational world. Its four *statements* are fun, data, class and inst. The semantics of statements is given by rules $\langle \Gamma, stmt \rangle \longrightarrow \langle \Gamma', E \rangle$ where Γ denotes an initial context, Γ' the resulting context (an extension of Γ) and E the set of defining equations.

The reading of the semantics can also be reversed. Then it describes what are the required equational theorems needed as witnesses to generate some statement (see §6). An example program is shown in the left column of Table 2.

Before we go into details we need to discuss the correctness issue. In the end, we want to ensure that if we translate an Isabelle/HOL theory via our intermediate language into SML or Haskell, and the compiler accepts it and some input term t, and reduces t to v, then $t \equiv v$ should be provable in Isabelle/HOL. This is partial correctness. Hence we only need to ensure that any equation used during reduction is contained in the semantics E we give to each statement. This is not hard to check by inspecting our semantics.

The semantics is expressed as a relation because it is partial due to context conditions. These context conditions are more liberal than in SML; this may lead to untypeable SML programs, for example due to polymorphic recursion, but is not a soundness problem.

Extending the various components of Γ is written $\Gamma[x := y]$ where the component in question is determined by the type of x and y. For example, $\Gamma[\kappa := k]$ extends the *TYP*-component. The extension $\Gamma[x := y]$ is not permitted if x is already defined in the corresponding component of Γ.

Function definitions Function definitions have the syntax

$$\mathsf{fun}\ f :: \forall \overline{\alpha :: s_k}.\ \tau\ \mathsf{where}\ f\,[\overline{\alpha :: s_k}]\,\overline{t_1} = t_1 \mid \ldots \mid f\,[\overline{\alpha :: s_k}]\,\overline{t_n} = t_n$$

where each equation $f\,\overline{t_i} = t_i$ must conform to the restrictions of Definition 2. The semantics is the obvious one:

$$\frac{\Gamma' = \Gamma[f := \forall \overline{\alpha :: s_k}.\ \tau] \quad \forall 1 \le i \le n.\ \Gamma' \vdash f\,[\overline{\alpha :: s_k}]\,\overline{t_i} \equiv t_i :: \mathit{prop}}{\langle \Gamma, \mathsf{fun}\ldots\rangle \longrightarrow \langle \Gamma', \{f\,[\overline{\alpha :: s_k}]\,\overline{t_1} \equiv t_1, \ldots\}\rangle}$$

Note that we allow polymorphic recursion.

Data types Recursive data types introduce a type constructor κ and *term constructors* f_i:

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

The $\overline{\alpha}_k$ must be distinct and no other type variable may occur in the $\overline{\tau_i}$:

$$\frac{\Gamma' = \Gamma[\kappa := k][f_1 := \forall \overline{\alpha}_k.\ \overline{\tau_1} \to \kappa\,\overline{\alpha}_k, \ldots] \quad \forall 1 \le i \le n.\ \Gamma' \vdash \overline{\tau_i} \to \kappa\,\overline{\alpha}_k}{\langle \Gamma, \mathsf{data}\ldots\rangle \longrightarrow \langle \Gamma', \{\}\rangle}$$

Note that our actual implementation allows fun to define mutually recursive functions and data mutually recursive data types. The corresponding modifications of our presentation are straightforward but tedious.

Type classes Overloading is covered by Haskell-style class and instance declarations [6]:

$$\mathsf{class}\ c \subseteq c_1 \cap \ldots \cap c_m\ \mathsf{where}\ f_1 :: \forall \alpha.\ \tau_1, \ldots, f_n :: \forall \alpha.\ \tau_n$$
$$\mathsf{inst}\ \kappa\,\overline{s}_k :: c\ \mathsf{where}\ f_1 = t_1, \ldots, f_n = t_n$$

Class declarations merely extend the context, provided they are well-formed:

$$\frac{\Gamma \vdash c_1 \cap \ldots \cap c_m \quad \forall 1 \le i \le n.\ \Gamma \vdash \tau_i}{\langle \Gamma, \mathsf{class}\ldots\rangle \longrightarrow \langle \Gamma[c := \overline{c}_m][f_1 := \forall \alpha :: c.\ \tau_1, \ldots], \{\}\rangle}$$

$$\frac{\Gamma' = \Gamma[(\kappa, c) := \overline{s}_k] \quad \forall 1 \le i \le n.\ \exists \tau.\ \Omega\, f_i = \forall \alpha :: c.\ \tau \wedge \Gamma' \vdash t_i :: \tau[\kappa\,\overline{\alpha :: s_k}/\alpha]}{\langle \Gamma, \mathsf{inst}\ldots\rangle \longrightarrow \langle \Gamma', \{f_1\,[\kappa\,\overline{\alpha :: s_k}] \equiv t_1, \ldots\}\rangle}$$

where Γ' must be coregular. Note that functions in a single inst may be mutually recursive.

Programs Now we lift $\longrightarrow$ to lists of statements:

$$\frac{}{\langle \Gamma, []\rangle \Longrightarrow \langle \Gamma, \{\}\rangle} \qquad \frac{\langle \Gamma, \mathit{stmt}\rangle \longrightarrow \langle \Gamma', E\rangle \quad \langle \Gamma', \mathit{stmts}\rangle \Longrightarrow \langle \Gamma'', E'\rangle}{\langle \Gamma, (\mathit{stmt}; \mathit{stmts})\rangle \Longrightarrow \langle \Gamma'', E \cup E'\rangle}$$

Statements are processed incrementally in an SML-like manner.

Definition 4 *A program S is a list of statements with a well-defined semantics, i.e. there exists a transition $\langle \Gamma_0, S\rangle \Longrightarrow \langle \Gamma, E\rangle$ where Γ_0 is the initial context containing only the type prop with polymorphic equality $\equiv$ of result type prop. We then write $S \rightsquigarrow (\Gamma, E)$.*

Note that if $S \rightsquigarrow (\Gamma, E)$, each equation in E is well-typed with respect to Γ due to monotonicity of context extensions and the construction of the rules for $\longrightarrow$. Context conditions imposed on inst statements ensure that Γ is coregular.

In §6 we show how to distill a program from a set of equations still involving classes. This requires some type class technicalities presented in §5 where we show how to replace classes by dictionaries.

5 Dictionary construction

We have given inst statements a semantics in terms of overloaded defining equations. In classical Haskell their semantics is given by a dictionary construction [16]. To justify this link, we formalize dictionary construction as a transformation of a program S within the intermediate language to a program in the same language but without any class or inst statements. To generate code for target languages lacking type classes (e.g. SML), this construction is carried out on the intermediate language (i.e. outside the logic).

Within our framework of order-sorted algebra, dictionary construction is described as a translation (relative to some Γ) of order-sorted types into dictionary terms [17], lifted to terms and statements:

- $(\!|\tau :: c|\!)$ maps a well-sortedness judgment to a corresponding dictionary,
- $(\!|t|\!)$ introduces dictionaries into a term t,
- $(\!|S|\!)$ transforms a program to its typeclass-free counterpart.

Dictionaries D are built from (global) dictionary constants c_κ and (local) dictionary variables α_n with explicit projections $\pi_{d\to c}$ by the following interpretation of rules for well-sortedness judgments. Dictionary construction relies on a unique representation of sorts. When writing $c_1 \cap \ldots \cap c_m$ we assume a total order of classes and that $c_1 \cap \ldots \cap c_m$ is minimal, i.e. no c_i is a subclass of any other c_j.

$$\frac{(\!|\tau :: c_1|\!) = D_1 \ldots (\!|\tau :: c_q|\!) = D_q}{(\!|\tau :: c_1 \cap \ldots \cap c_q|\!) = \overline{D}_q}\ (\mathrm{sort}_D)$$

$$\frac{(\!|\tau_1 :: s_1|\!) = D_1 \ldots (\!|\tau_n :: s_n|\!) = D_n \quad \Sigma\,(\kappa, c) = \overline{s}_n}{(\!|\kappa\,\tau_1 \ldots \tau_n :: c|\!) = c_\kappa\,\overline{D}_n}\ (\mathrm{constructor}_D)$$

$$\frac{}{(\!|(\alpha :: c_1 \cap \ldots \cap c_n \cap \ldots \cap c_q) :: c_n|\!) = \alpha_n}\ (\mathrm{variable}_D)$$

$$\frac{(\!|(\alpha :: s) :: d|\!) = D \quad c \in SUP\,d}{(\!|(\alpha :: s) :: c|\!) = \pi_{d\to c}\,D}\ (\mathrm{classrel}_D)$$

Rule classrel_D only works on judgments of the form $(\alpha :: s) :: c$, thus prohibiting pointless constructions followed by projections as in $\pi_{d\to c}\,(d_\kappa\ \ldots)$. There remains an ambiguity: there might be different paths $c_1 \in SUP\,c_2, c_2 \in SUP\,c_3, \ldots, c_{n-1} \in SUP\,c_n$ in the class hierarchy from a subclass c_n to a superclass c_1. To make the system deterministic we assume a canonical path is chosen. The correctness proof below (implicitly) shows that the exact choice is immaterial.

The $(\!|t|\!)$ function (relative to a context Γ) only affects type applications:

$$\begin{aligned}(\!|f\,[\tau_1 \ldots \tau_n]|\!) &= f\,(\!|\tau_1 :: s_1|\!) \ldots (\!|\tau_n :: s_n|\!) \text{ where } \Omega\,f = \forall\overline{\alpha :: s}_n.\,\tau\\ (\!|x :: \tau|\!) &= x :: \tau\\ (\!|\lambda x :: \tau.\,t|\!) &= \lambda x :: \tau.\,(\!|t|\!)\\ (\!|t_1\,t_2|\!) &= (\!|t_1|\!)\,(\!|t_2|\!)\end{aligned}$$

Note that $(\!|t|\!)$ is injective, i.e. $(\!|t_1|\!) = (\!|t_2|\!)$ implies $t_1 = t_2$. For succinctness we introduce two more abbreviations (where δ_c is a dictionary type corresponding to class c):

$$\begin{aligned}\{\!|\alpha :: c_1 \cap \ldots \cap c_n|\!\} &= (\delta_{c_1}\,\alpha) \ldots (\delta_{c_n}\,\alpha)\\ (\!|c_\kappa\,[\overline{\tau}_k]|\!) &= c_\kappa\,(\!|\tau_1 :: s_1|\!) \ldots (\!|\tau_k :: s_k|\!) \text{ where } \Sigma(\kappa, c) = \overline{s}_k\end{aligned}$$

$(\!|S|\!)$ is the concatenation of the transformation shown in Table 1 applied to each statement in S. Transformed statements stemming from class and inst statements introduce dictionary constants c_κ along with projections $\pi_{d\to c}$ for subclass relations and f for constants associated with classes (class operations). See Table 2 for an example.

The transformation of inst reveals the role of coregularity for dictionary construction: each $(\!|d_{i\kappa}\,[\overline{\alpha :: s}_k]|\!)$ on the right hand side requires that the corre-

statement	transformed statement(s)
fun $f :: \forall\overline{\alpha :: s}_k.\,\tau$ where $\quad f\,[\overline{\alpha :: s}_k]\,\overline{t_1} = t_1$ $\quad\mid \ldots$ $\quad\mid f\,[\overline{\alpha :: s}_k]\,\overline{t_n} = t_n$	fun $f :: \forall\overline{\alpha}_k.\,\overline{\{\!|\alpha :: s|\!\}}_k \to \tau$ where $\quad(\!|f\,[\overline{\alpha :: s}_k]\,\overline{t_1}|\!) = (\!|t_1|\!)$ $\quad\mid \ldots$ $\quad\mid (\!|f\,[\overline{\alpha :: s}_k]\,\overline{t_n}|\!) = (\!|t_n|\!)$
data $\kappa\,\overline{\alpha}_k =$ $\quad f_1$ of $\overline{\tau_1} \mid \ldots \mid f_n$ of $\overline{\tau_n}$	data $\kappa\,\overline{\alpha}_k =$ $\quad f_1$ of $\overline{\tau_1} \mid \ldots \mid f_n$ of $\overline{\tau_n}$
class $c \subseteq c_1 \cap \ldots \cap c_m$ where $\quad f_1 :: \forall\alpha.\tau_1, \ldots, f_n :: \forall\alpha.\tau_n$	data $\delta_c\,\alpha = \Delta_c$ of $\{\!|\alpha :: c_1 \cap \ldots \cap c_m|\!\}\,\overline{\tau}_n$ fun $\pi_{c\to c_i} :: \forall\alpha.\,\delta_c\,\alpha \to \delta_{c_i}\,\alpha$ where $\quad\pi_{c\to c_i}\,(\Delta_c\,c_1\ \ldots\ c_i\ \ldots\ c_m\,g_1\ \ldots) = c_i,\ 1 \le i \le m$ fun $f_i :: \forall\alpha.\,\delta_c\,\alpha \to \tau_i$ where $\quad f_i\,(\Delta_c\,c_1\ \ldots\ g_1\ \ldots\ g_i\ \ldots\ g_n) = g_i,\ 1 \le i \le n$
inst $\kappa\,\overline{s}_k :: c$ where $\quad f_1 = t_1, \ldots, f_n = t_n$	fun $c_\kappa :: \forall\overline{\alpha}_k.\,\overline{\{\!|\alpha :: s|\!\}}_k \to \delta_c\,(\kappa\,\overline{s}_k)$ where $\quad(\!|c_\kappa\,[\overline{\alpha :: s}_k]|\!) = \Delta_c\,(\!|d_{1\kappa}\,[\overline{\alpha :: s}_k]|\!)\ \ldots\ (\!|d_{m\kappa}\,[\overline{\alpha :: s}_k]|\!)\,\overline{(\!|t|\!)}_n$ $\qquad$ where $SUP\,c = d_1 \ldots d_m$

Table 1. Dictionary construction for statements.

sponding sort constraints stemming from the inst statement that gave rise to the definition of d_{i_κ} are as least as general as the $\overline{\alpha :: s}_k$.

We assume distinct name spaces to embed dictionary constant names c_κ and dictionary variable names α_n into the name space of constants and term variables respectively. Each dictionary term constructors Δ_c constructs a tuple containing the dictionaries of the direct superclasses of c together with the class operations declared in c.

Correctness

Given a program S with $S \rightsquigarrow (\Gamma, E)$ and its transformed counterpart $(\!|S|\!)$ with $(\!|S|\!) \rightsquigarrow (\Gamma_D, E_D)$, we show how E and E_D are related. The main problem is that derivations in E_D contain symbols not present in E. Thus intermediate terms in an E_D-derivation cannot always be viewed as Γ-terms. Hence we work with normal forms modulo certain projection rules, the rules stemming from class statements. For this fine-grained reasoning we move from equational logic to term rewriting. Instead of arbitrary equational proofs $(\Gamma_D, E_D) \vdash t_1 \equiv t_2$ we consider rewrite proofs $t_1 \xrightarrow{*}_{E_D} t_2$, see [8]. This models the evaluation of t_1. More precisely, we start with some $(\!|s|\!)$, reduce it to some t', and if t' is of the form

program	transformed program
data Nat = Zero | Suc of Nat	data Nat = Zero | Suc of Nat
data List α = Nil | Cons of α (List α)	data List α = Nil | Cons of α (List α)
class Pls where	data $\delta_{\mathrm{Pls}}\,\alpha = \Delta_{\mathrm{Pls}}$ of $\alpha \to \alpha \to \alpha$
pls :: $\forall\alpha.\ \alpha \to \alpha \to \alpha$	fun pls :: $\forall\alpha.\ \delta_{\mathrm{Pls}}\,\alpha \to \alpha \to \alpha \to \alpha$ where
	pls (Δ_{Pls} pls$'$) = pls$'$
class Neutr $\subseteq$ Pls where	data $\delta_{\mathrm{Neutr}}\,\alpha = \Delta_{\mathrm{Neutr}}$ of ($\delta_{\mathrm{Pls}}\,\alpha$) α
neutr :: $\forall\alpha.\ \alpha$	fun $\pi_{\mathrm{Neutr}\to\mathrm{Pls}}$ :: $\forall\alpha.\ \delta_{\mathrm{Neutr}}\,\alpha \to \delta_{\mathrm{Pls}}\,\alpha$ where
	$\pi_{\mathrm{Neutr}\to\mathrm{Pls}}$ (Δ_{Neutr} Pls neutr$'$) = Pls
	fun neutr :: $\forall\alpha.\ \delta_{\mathrm{Neutr}}\,\alpha \to \alpha$ where
	neutr (Δ_{Neutr} Pls neutr$'$) = neutr$'$
fun add :: Nat → Nat → Nat where	fun add :: Nat → Nat → Nat where
add Zero m = m	add Zero m = m
| add (Suc n) m = Suc (add n m)	| add (Suc n) m = Suc (add n m)
inst Nat :: Pls where	fun $\mathrm{Pls}_{\mathrm{Nat}}$ :: δ_{Pls} Nat where
pls = add	$\mathrm{Pls}_{\mathrm{Nat}}$ = Δ_{Pls} add
inst Nat :: Neutr where	fun $\mathrm{Neutr}_{\mathrm{Nat}}$ :: δ_{Neutr} Nat where
neutr = Zero	$\mathrm{Neutr}_{\mathrm{Nat}}$ = Δ_{Neutr} $\mathrm{Pls}_{\mathrm{Nat}}$ Zero
fun sum :: $\forall\alpha$ :: Neutr. List $\alpha \to \alpha$ where	fun sum :: $\forall\alpha.\ \delta_{\mathrm{Neutr}}\,\alpha \to$ List $\alpha \to \alpha$ where
sum [α] Nil = neutr [α]	sum α_1 Nil = neutr α_1
| sum [α] (Cons x xs) = pls [α] x (sum [α] xs)	| sum α_1 (Cons x xs) =
	pls ($\pi_{\mathrm{Neutr}\to\mathrm{Pls}}\,\alpha_1$) x (sum α_1 xs)
fun val :: Nat where	fun val :: Nat where
val = sum [Nat] (Cons (Suc Zero) Nil)	val = sum $\mathrm{Neutr}_{\mathrm{Nat}}$ (Cons (Suc Zero) Nil)

Table 2. Dictionary construction applied to example program.

$(\!|s'|\!)$, we want to conclude $s \to_E s'$. The remainder of this section is dedicated to that proof.

Observe that (by definition of $(\!|S|\!)$) E_D can be partitioned as $E_D = E_F \uplus E_I \uplus E_E$ where

E_F denotes all equations stemming from transformed fun statements.
E_I denotes all equations stemming from transformed inst statements that *introduce* a particular Δ_c: $c_\kappa\,\overline{\alpha} \equiv \Delta_c\,\overline{t}$.
E_E denotes all equations stemming from transformed class statements that *eliminate* a particular Δ_c: $p(\Delta_c\,\overline{x}) \equiv x_i$ where p is either a projection $\pi_{c\to d}$ or a class operation f.

Given an arbitrary reduction sequence $t_1 \stackrel{*}{\to}_{E_D} t_2$ where t_1 and t_2 are Δ-free, we can assume w.l.o.g. that no reduction takes place underneath a Δ — such reductions can always be postponed. This is because all arguments of Δ on the left-hand side of any equation in E_D are (distinct) variables. Similarly, we can postpone all E_I steps up to the point where they are needed, i.e. in front of a corresponding E_E step. This allows to view derivations in a normal form where each E_I step occurs immediately before a corresponding E_E step. Now we collapse these E_I / E_E pairs into single E_{IE} steps defined as follows:

$$E_{IE} = \{p(c_\kappa\,\overline{\alpha}) \equiv t_i \mid \langle c_\kappa\,\overline{\alpha} \equiv \Delta_c\,\overline{t}\rangle \in E_I \wedge \langle p(\Delta_c\,\overline{x}) \equiv x_i\rangle \in E_E\}$$

Clearly, the equations in E_{IE} are consequences of those in E_I and E_E. Neither E_F steps nor E_{IE} steps introduce Δs, so any intermediate term is Δ-free; hence no E_E steps remain. Thus we have transformed $t_1 \stackrel{*}{\to}_{E_D} t_2$ into $t_1 \stackrel{*}{\to}_{E_F \uplus E_{IE}} t_2$. Again we partition our rule set: $E_{IE} = E_{op} \uplus E_\pi$ where

E_{op} contains equations for class operations: $f\,(c_\kappa\,\overline{x}) \equiv \ldots$, and
E_π contains equations for superclass projections: $\pi_{c\to d}\,(c_\kappa\,\overline{x}) \equiv d_\kappa\ \ldots$.

This establishes a one-to-one correspondence between equations in E and equations in $E_F \uplus E_{op}$ such that $\langle t_1 \equiv t_2\rangle \in E$ implies $\langle (\!|t_1|\!) \equiv (\!|t_2|\!)\rangle \in E_F \uplus E_{op}$. For fun statements this holds by definition, for inst statements it holds by construction of E_{op}. Finally we transform our $E_F \uplus E_{IE}$ reduction sequence such that after each $E_F \uplus E_{op}$ step we normalize w.r.t. E_π. This transformation is accomplished by the following theorem:

Theorem 5 *Let R and P be two sets of defining equations in a context free of classes such that the left-hand sides of R and P do not overlap, P is confluent, terminating and right-linear, and the right-hand sides of P preserve β-normal forms (i.e. if a right-hand side of P is instantiated by β-normal forms, the result is in β-normal form.). Then the following commutation property holds:*

$$t_1 \leftarrow_P t \to_R t_2 \text{ implies } \exists u.\ t_1 \stackrel{=}{\to}_R u \stackrel{*}{\leftarrow}_P t_2$$

The proof is by a careful case distinction on the relative position of redexes.

By induction we obtain:

Corrolary 6 *If the assumptions of Theorem 5 hold, then* $t \xrightarrow{*}_{R \cup P} t'$ *implies* $t{\downarrow_P}\ (\rightarrow_R \circ \xrightarrow{!}_P)^*\ t'{\downarrow_P}$*, where* $x \xrightarrow{!} y$ *means* $x \xrightarrow{*} y$ *and* y *is in normal form.*

Setting $R = E_F \uplus E_{op}$ and $P = E_\pi$ we obtain: if $(\!|s|\!) \xrightarrow{*}_{E_F \uplus E_{op} \uplus E_\pi} (\!|s'|\!)$ then $(\!|s|\!)\ (\rightarrow_{E_F \uplus E_{op}} \circ \xrightarrow{!}_{E_\pi})^*\ (\!|s'|\!)$. Due to the form of the rules they satisfy the requirements of Theorem 5. In particular, $E_F \uplus E_{op}$ and E_π are orthogonal because no π occurs in any left-hand side of E_F or E_{op}. Also note that $(\!|s|\!){\downarrow_{E_\pi}} = (\!|s|\!)$ because $(\!|s|\!)$ contains no π-redexes.

Since each term t in a derivation $(\!|s|\!)\ (\rightarrow_{E_F \uplus E_{op}} \circ \xrightarrow{!}_{E_\pi})^*\ (\!|s'|\!)$ is Δ-free and E_π-normalized, it is the image of an s'' such that $(\!|s''|\!) = t$. Thus each $E_F \uplus E_{op}$ step using an equation $(\!|s_1|\!) \equiv (\!|s_2|\!)$ followed by E_π-normalization corresponds to an E-step using $s_1 \equiv s_2$. Normalization with E_π is necessary because substituting dictionaries into a translated term yields E_π redexes to access dictionaries of superclasses. In the original system E this is implicit due to subclassing.

Overall, we have now obtained the desired result:

Lemma 7 *If* $(\!|s|\!) \xrightarrow{*}_{E_D} (\!|s'|\!)$ *then* $s \xrightarrow{*}_E s'$.

6 Implementable systems

We will now discuss the step from a logical theory (Γ, E) to a program. It can be seen as the inverse of $\Longrightarrow$. More precisely:

Definition 8 *A program* S implements *an equation system* (Γ, E) *iff* $S \rightsquigarrow (\Gamma', E)$ *such that* Γ' *is* compatible *with* Γ*:*

- $TYP' \subseteq TYP$
- $SUP' \subseteq SUP$
- $\Sigma'(\kappa, c) = \overline{s'_k}$ *implies* $\exists \overline{s}_k.\ \Sigma(\kappa, c) = \overline{s}_k \wedge \forall 1 \leq i \leq k.\ \Gamma \vdash s'_i \subseteq s_i$
- $\Omega'\ c = \forall \overline{\alpha :: s'_k}.\ \tau$ *implies* $\exists \overline{s}_k.\ \Omega\ c = \forall \overline{\alpha :: s_k}.\ \tau \wedge \forall 1 \leq i \leq k.\ \Gamma \vdash s'_i \subseteq s_i$

Compatibility means that any expression which is valid with respect to Γ' is also valid with respect to Γ. Permitting more restrictive sort constraints may be necessary for implementation reasons. For example, equality $(=)$ is free of any class constraints in HOL but its implementation requires an equality class (§7).

Isabelle provides definition mechanisms (the details are immaterial) corresponding to our programming language statements data, fun, class and inst. Systems (Γ, E) defined purely in this manner are implementable. But there are other, more primitive ways to define functions in Isabelle which may not be directly implementable (see e.g. §4). To construct a program from (Γ, E) we need certain extra-logical information not directly contained in (Γ, E) anymore, e.g. what are the term constructors and the class operations. We will now isolate what is required and will then show that it enables us to assemble a program from a system (Γ, E). We do not explicitly discuss the preprocessor which selects and transforms an initial set of equations into the required form (if possible).

Term constructors determine which constants must be introduced by data statements:

Definition 9 *C is a set of* term constructors *for (Γ, E) iff for each constant f occurring in the arguments of a defining equation $f \in C$ holds, there are no defining equations in E for any $f \in C$, and for each $f \in C$ its generic type $\Omega\, f$ is of the form $\forall\overline{\alpha}.\, \overline{\tau}_n \to \kappa\, \overline{\alpha}$, and each occurrence of f in the arguments of a defining equation is fully applied with n arguments.*

Class and overloading discipline Defining equations may contain arbitrary type arguments whereas our programming language enforces a certain discipline. This is captured by an explicit association of constants to classes (corresponding to class statements):

Definition 10 *An n-to-1 relation $\in$ between constant symbols and class symbols is a* class membership relation *w.r.t. Γ iff for each $f \in c$ its generic type $\Omega\, f$ is of the form $\forall\alpha :: c.\, \tau$, i.e. is generic in only one type argument of class c. All constant symbols in the domain of $\in$ are named* class operations.

Given a set of defining equations E, $\langle f\, [\overline{\tau}_k]\ \ldots \equiv \ldots\rangle \in E$ is called

non-overloaded if $\overline{\tau}_m$ is of the form $[\overline{\alpha :: s}_m]$ and all equations for f in E have the same type arguments.

overloaded if $m = 1$ and $\overline{\tau}_m$ is of the form $\kappa\, \overline{\alpha :: s}_k$, and there is no other $\langle f\, [\kappa\ \ldots]\ \ldots \equiv \ldots\rangle \in E$

Non-overloaded definitions can be implemented by fun statements, overloaded definitions by inst statements. We now lift the definition of coregularity from contexts to systems of equations E in order to satisfy the coregularity requirement of the inst statement. E is *coregular* iff

$$\langle f\, [\kappa\, \overline{\alpha :: s'}_m] \ldots \equiv \ldots\rangle \in E, f \in c \,\text{implies}\, \forall d, \Gamma \vdash c \subseteq d, g \in d.$$
$$\exists \langle g\, [\kappa\, \overline{\alpha :: s''}_m] \ldots \equiv \ldots\rangle \in E.\, \forall 1 \leq n \leq m.\, \Gamma \vdash s'_n \subseteq s''_n$$

In case $c = d$, coregularity ensures that for a specific instance (κ, c) all class operations are instantiated and occur with the same sort constraints.

Definition 11 *A system (Γ, E) obeys* class discipline *w.r.t. a class membership relation $\in$ iff all defining equations in E are either overloaded or non-overloaded, and if for each $\langle f\, [\kappa\, \overline{\alpha :: s'}_k] \ldots \equiv \ldots\rangle \in E$ there are c and $\overline{s}_k$ such that $f \in c$, $\Sigma(\kappa, c) = \overline{s}_k$, and $\Gamma \vdash s'_i \subseteq s_i$ for $1 \leq i \leq k$. Furthermore E must be coregular.*

In a system with term constructors C which obeys class discipline w.r.t. $\in$, each constant symbol is either a term constructor (if $f \in C$), or a class operation (if $f \in c$ for some c), or simply a *function* (otherwise). With this partitioning, E induces the more specialized context components $\tilde{\Omega}$ and $\tilde{\Sigma}$:

Definition 12 *Let (Γ, E) be a system with term constructors C obeying class discipline w.r.t. $\in$. Then $\tilde{\Omega}\, f$ is identical to $\Omega\, f$ unless f is a function, in which case $\tilde{\Omega}\, f$ is derived from $\Omega\, f = \forall\overline{\alpha :: s}.\, \tau$ and any defining equation $f\, [\overline{\alpha :: s'}] \ldots \equiv \ldots$ as follows: $\tilde{\Omega}\, f = \forall\overline{\alpha :: s'}.\, \tau$. For each defining equation $f\, [\kappa\, \overline{\alpha :: s}_k] \ldots \equiv \ldots$ with $f \in c$ we define $\tilde{\Sigma}\, (\kappa, c) = \overline{s}_k$.*

Given a program S with $S \rightsquigarrow (\Gamma', E)$, by construction $\tilde{\Omega}$ and $\tilde{\Sigma}$ form the corresponding components of the context Γ'.

Definition 13 *An expression* $f\,[\overline{\tau}_n]$ occurs *in a defining equation iff* $f\,[\overline{\tau}_n]$ *occurs in either the right hand side* or *in any argument.*

Intuitively, this models a notion of "this function uses function f".

Because the sort constraints in $\tilde{\Omega}$ and $\tilde{\Sigma}$ may be more restrictive than those in Ω and Σ, it is necessary to require well-sortedness of E explicitly:

Definition 14 *A system* (Γ, E) *with term constructors* C *obeying class discipline w.r.t.* $\in$ *is* well-sorted *iff for any constant expression* $f\,[\overline{\tau}_n]$ *occurring in any defining equation,* $(SUP, TYP, \tilde{\Sigma}, \tilde{\Omega}) \vdash \tau_i :: s_i$ *for* $1 \leq i \leq n$*, where* $\tilde{\Omega}\, f = \forall \overline{\alpha :: s}_n.\, \tau$.

Note that well-sortedness can be achieved by the preprocessor which propagates the additional sort constraints through the system of equations.

Dependency closedness A requirement for programs S is that no statement relies on ingredients which have not been introduced so far; this induces a notion of *dependencies* of statements which is also reflected in the underlying system of defining equations. We describe this in terms of *dependency graphs* $\longrightarrow_E$ with directed edges where each node *either* refers to a term constructor, a generic class operation, a function (denoted by f) *or* refers to a particular instance of a class operation (denoted by f_κ). For brevity, we use the notation f_* referring uniformly to overloaded and non-overloaded constants, depending on which kind of defining equation f comes from.

Definition 15 *Given a well-sorted system* (Γ, E) *with term constructors* C *obeying class discipline w.r.t.* $\in$*, the* dependency graph $\longrightarrow_E$ *is defined by the following rules:*

$$\frac{\langle f\,[\overline{\tau}]\,\overline{t} \equiv t\rangle \in E \quad g\,[\overline{\tau'}] \text{ occurs in } \langle f\,[\overline{\tau}]\,\overline{t} \equiv t\rangle}{f_* \longrightarrow_E g} \;(dep)$$

$$\frac{\langle f\,[\overline{\tau}]\,\overline{t} \equiv t\rangle \in E \quad g\,[\overline{\tau'}] \text{ occurs in } \langle f\,[\overline{\tau}]\,\overline{t} \equiv t\rangle \quad \tilde{\Omega}\, g = \forall \overline{\alpha :: s}.\, \tau \quad c_\kappa \text{ occurs in } \overline{(\!|\tau' :: s|\!)_{\tilde{\Sigma}}} \quad h \in c}{f_* \longrightarrow_E h_\kappa} \;(dict)$$

$$\frac{f \in c \quad g \in c \quad (\kappa, c) \in dom\, \Sigma}{f_\kappa \longrightarrow_E g_\kappa} \;(instop)$$

dep models dependencies of defining equations on class, data or fun statements.

dict models dependencies of defining equations on inst statements; the notation $(\!|\tau' :: s|\!)_{\tilde{\Sigma}}$ means dictionary construction w.r.t. $\tilde{\Sigma}$.

instop models that class operations which are members of the same class have to be instantiated simultaneously (by means of a corresponding inst statement).

Definition 16 *A dependency graph* $\longrightarrow_E$ *is* closed *iff*

1. *for each edge* $f_* \longrightarrow_E g_*$, g_* *is either a term constructor in* C or *a generic class operation* or *a node with at least one outgoing edge;*
2. *the nodes of each strongly connected component of* $\longrightarrow_E$ *consist* either *only of non-overloaded constants* f or *of all overloaded constants* f_κ, $f \in c$, *for some fixed* κ *and* c.

The latter condition asserts that inst statements may not be mutually recursive, least of all in connection with fun statements.

Theorem 17 *Each well-sorted system* (Γ, E) *with term constructors* C *obeying class discipline w.r.t.* $\in$, *where* $\longrightarrow_E$ *is closed, is implementable.*

Proof sketch Let (Γ, E) have the required properties; a proof that (Γ, E) is implementable consists of the following parts:

- Show that C and $\in$ can be realized by data and class statements.
- Show that the defining equations E are mappable to fun and inst statements, i.e. that there exists a partitioning of E where each partition corresponds to a statement whose semantics is the original set of defining equations.
- Show that there exists an order of the above statements such that the corresponding list is a program.

The first is trivial by construction. For the second observe that class discipline yields partitions of non-overloaded defining equations corresponding to fun statements and overloaded defining equations corresponding to inst statements. The type and sort information needed for class, inst and fun statements comes from $\tilde{\Omega}$ and $\tilde{\Sigma}$. Well-sortedness of (Γ, E) in the sense of Definition 14 guarantees that the resulting statements are typeable. The third requires some thought how to find out an appropriate order for a given list of statements. Essentially, any symbol (c, κ or f) must be introduced *before* it occurs in a statement. Furthermore all inst statements must occur before they are required for typing type applications $f\ [\overline{\tau}]$. The following order guarantees this:

- data statements only depend on the existence of particular type constructors κ; so, data statements may be placed *before* any other kind of statements. data statements themselves are ordered such that any data statement depending on type constructor κ is preceeded by a data statement introducing κ; mutual dependencies result in mutual recursive data statements.
- class statements depend on the existence of type constructors κ and superclasses c; since all κ can be introduced by preceeding data statements, only the classes have to be considered. Ordering class statements in topological order with respect to the subclass relation (starting with the top classes) results in an order where all superclasses of a class statement are introduced by preceeding class statements.

– The issue of fun and inst statements is more complicated because a fun statement may depend on an inst statement and also vice versa. But dependency closedness implies that the strongly connected components of $\longrightarrow_E$ correspond exactly to (possibly mutually recursive) fun statements or inst statements; their order is determined by the graph. □

7 Executable equality

The constant denoting HOL equality $(=) :: \alpha \to \alpha \to \mathit{bool}$ is a purely logical construct. Its axiomatization is not in the form of defining equations. However, by providing an appropriate framework setup, we can derive defining equations for types which have an operational notion of equality (e.g. there is no operational equality on function types). Operational equality serves as example how the logic may be utilized to widen the possibilities for code generation without any need to extend the trusted code base of the framework.

When modeling operational equality we follow the Haskell approach: each equality type belongs to a type class eq: $(=) \in \mathit{eq}$. Isabelle/HOL proves for each recursive datatype a set of defining equations for equality on that type, similar to Haskell's "`deriving Eq`":

– Check whether all existing type constructors κ' which κ depends on are instances of eq ($\kappa' \, \overline{\mathit{eq}} :: \mathit{eq}$); if not, abort the whole procedure — then κ does not support operational equality.
– Declare κ an instance of eq, provided its type arguments are also instances of eq: $\kappa \, \overline{\mathit{eq}} :: \mathit{eq}$.
– Define a new constant $\mathit{eq}_\kappa \, [\overline{\alpha :: \mathit{eq}}] \equiv (=) \, [\kappa \, \overline{\alpha :: \mathit{eq}}]$.
– From this primitive definition prove injectiveness and distinctness equations as defining equations for eq_κ:

$$\mathit{eq}_\kappa \, (f_i \, \overline{x}_m) \, (f_i \, \overline{y}_m) \equiv (=) \, x_1 \, y_1 \wedge \ldots \wedge (=) \, x_m \, y_m$$
$$\mathit{eq}_\kappa \, (f_i \, \ldots) \, (f_j \, \ldots) \equiv \mathit{False}, \text{ for } i \neq j$$

where empty conjunctions collapse to True and recursive calls of equality $(=) \, [\kappa \, \overline{\alpha :: \mathit{eq}}]$ are replaced by eq_κ.
– Use the symmetric definition $(=) \, [\kappa \, \overline{\alpha :: \mathit{eq}}] \equiv \mathit{eq}_\kappa$ as defining equation for $(=) \, [\kappa \, \overline{\alpha :: \mathit{eq}}]$.

On code generation, the preprocessor (§2) propagates eq sort constraints through the system of defining equations, e.g. a defining equation $(\neq) \, [\alpha] \, x \, y \equiv \neg((=) \, [\alpha] \, x \, y)$ is constrained to $(\neq) \, [\alpha :: \mathit{eq}] \, x \, y \equiv \neg((=) \, [\alpha :: \mathit{eq}] \, x \, y)$. Because this is a purely logical inference, the translation process is completely unchanged. Thus, operational equality is treated inside the logic. In particular, this approach is completely independent from any target-language specific notion of equality (e.g. SML's polymorphic $(\mathit{op} =)$).

8 Conclusion

We have presented the design of Isabelle/HOL's latest code generator (available

in the development snapshot) which is, for the first time, able to deal with type classes. The correctness of code generation, in particular the relationship between type classes and dictionaries, is established by proof theoretic means. The trusted code base is minimized by working with a conceptually simple programming language with a straightforward equational semantics.

References

[1] S. Berghofer and T. Nipkow. Executing higher-order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *LNCS*, pages 24–40. Springer, 2002.

[2] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.

[3] R. S. Boyer and J. S. Moore. Single-threaded objects in ACL2. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 9–27, London, UK, 2002. Springer-Verlag.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Mar. 2001.

[5] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. *SIGPLAN Not.*, 41:206–217, Jan. 2006.

[6] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2), 1996.

[7] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*. Springer, 2003.

[8] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[9] T. Meyer and B. Wolff. Correct code-generation in a generic framework. In M. Aargaard, J. Harrison, and T. Schubert, editors, *TPHOLs 2000: Supplemental Proceedings*, OGI Technical Report CSE 00-009, pages 213–230. Oregon Graduate Institute, Portland, USA, July 2000.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[11] T. Nipkow and C. Prehofer. Type checking type classes. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 409–418, New York, NY, USA, 1993. ACM Press.

[12] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[13] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic With Term Declarations*, volume 395 of *LNCS*. Springer, 1989.

[14] N. Shankar. Static analysis for safe destructive updates in a functional language. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation (LOPSTR 2001)*, volume 2372 of *LNCS*, pages 1–24, 2001.

[15] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.1*, July 2006.

[16] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[17] M. Wenzel and F. Haftmann. Constructive type classes in Isabelle. To appear in proceedings of TYPES 2006.