Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Code Generation from Specifications in Higher-Order Logic

Florian Haftmann

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Florian Matthes
Prüfer der Dissertation:		
	1.	Univ.-Prof. Tobias Nipkow, Ph.D.
	2.	Univ.-Prof. Dr. Helmut Seidl

Die Dissertation wurde am 27. Mai 2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 9. November 2009 angenommen.

Zusammenfassung

Ein sehr rigoroser Ansatz zur Vermeidung fehlerhaft implementierter Software ist formale Verifikation: sowohl Verhaltensbeschreibung (abstrakte Spezifikation) als auch Implementierung (ausführbare Spezifikation) werden in einem geeigneten logischen Kalkül beschrieben, und es wird gezeigt, dass beide sich gleich verhalten. Schon aufgrund der zahlreichen technischen Details liegt es nahe, das Überprüfen der einzelnen Beweisschritte innerhalb eines Beweisassistenten vorzunehmen. Dieser mechanische Ansatz ermöglicht es auch, in gewissen Fällen eine ausführbare Spezifikation automatisch in ein Programm in einer geeigneten Programmiersprache zu überführen. Dieses etablierte Verfahren ist als Code-Generierung bekannt.

Ziel dieser Arbeit ist die Darstellung eines Codegenerator-Frameworks für den interaktiven Theorembeweiser *Isabelle/HOL*, einer Implementierung höherstufiger Logik. Gegenüber existierenden Ansätzen weist das Framework zwei substantielle Neuerungen auf: ein sehr allgemeines, aber leichtgewichtiges Konzept für Datentypabstraktion und die Unterstützung von *Isabelle*-Typklassen im Sinne von *Haskell*-Typklassen. Konzeptionell möglich ist Generierung von Code für funktionale Sprachen mit Pattern Matching; konkrete Instantiierungen des Frameworks liegen vor für die Zielsprachen *SML*, *OCaml* und *Haskell*.

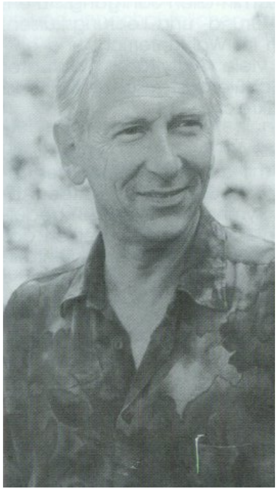
Die praktische Verwendbarkeit des Codegenerator-Frameworks wird mit exemplarischen Anwendungen demonstriert.

Abstract

A very rigorous weapon against implementation errors in software systems is formal verification: both the desired behaviour (abstract specification) and the implementation (executable specification) are formalised in a suitable logical calculus, and the equivalence of both is proved. The numerous technical details involved in such a procedure suggest to let a proof assistant check all proof steps. This mechanical approach in certain cases enables an automatic translation from an executable specification to a program in a suitable programming language: code generation.

The aim of this thesis is to present a code generator framework for the interactive proof assistant *Isabelle/HOL*, an implementation of higher-order logic. The framework includes two substantial novelties: a general but lightweight concept for datatype abstraction and support for *Isabelle* type classes in the manner of *Haskell* type classes. Code can be generated for functional programming languages supporting pattern matching; concrete instances for *SML*, *OCaml* and *Haskell* are presented.

The practical usability of the code generator framework is demonstrated with example applications.



In memoriam Werner Krehbiel (1941–2004)

Acknowledgements

The accomplishment of this thesis has been a fulfilling and absorbing task in all its facets: acquisition of knowledge, system development, elaboration. This would not have been possible without the constant support and feedback from the Isabelle group in Munich, whose (former and current) members I am deeply indebted to: Tobias Nipkow gave me the opportunity to work in his research group and supervised this thesis; Alexander Krauss was a travel mate on my journeys in both the figurative and literal sense; Stefan Berghofer and Makarius Wenzel supported and helped me patiently in my starting time; further Clemens Ballarin, Gertrud Bauer, Jasmin Blanchette, Sascha Böhme, Lukas Bulwahn, Amine Chaieb, Johannes Hölzl, Julien Narboux, Steven Obua, Norbert Schirmer, Christian Urban, Tjark Weber and Martin Wildmoser — may future generations of PhD students enjoy the same enlightenment, pleasure and friendly working atmosphere with and around Isabelle as I had the opportunity to experience.

Further thanks I owe to Helmut Seidl for acting as referee.

Among the many people involved with Isabelle whose centre of life is not in Munich I would like to mention Larry Paulson, John Matthews, Brian Huffman and Gerwin Klein, who gave important inspiration and feedback for my work.

Preliminary parts of this thesis have been read and commented by Alexander Krauss, Sascha Böhme, Makarius Wenzel and Jasmin Blanchette, whom I would like to thank in particular for his language expertise — remaining deficiencies still fall under my responsibility.

This research was financially supported by the DFG project NI 491/10-1.

Contents

CHAPTER 1

Introduction

Cuiusvis hominis est errare,
nullius nisi insipientis in errore perseverare.
Marcus Tullius Cicero, Roman orator,
from: Oratio Philippica Duodecima

1.1 Scenario

The practical motivation of this work is software development. More than thirty years ago, F. L. Bauer introduced an essay with the following paragraph [3][1]:

> *Programmieren ist die Erfüllung eines Kontrakts: Das Problem wird vereinbart, das lösende Programm wird abgeliefert. Die meisten Programme sind heute, zumindest auf den ersten Anhieb,* nicht *korrekt (manche bleiben auch ewig falsch und werden doch verkauft): sie erfüllen den Kontrakt nicht. Dieser ist häufig auch gar nicht ganz eindeutig formuliert. Darin ist aber der Grund für die vielen „Programmierfehler“ nicht oder nur teilweise zu suchen. Er liegt vielmehr hauptsächlich in der undisziplinierten Art, mit der von jung und alt landauf, landab das Programmieren betrieben wird.*

Nowadays it is widely acknowledged that programming is no ad hoc task combined with some kind of black art, but requires care and diligence; software development has become an established engineering discipline, namely *software engineering.* A typical computer science curriculum contains at least one lecture [11] discussing issues like software development workflow methodologies, project communication, visual modelling languages etc. — the human-oriented side of software development, so to speak.

[1] Translation by the author: Programming is the fulfilment of a contract: The problem is agreed on, the solving program is delivered. Most programs today are, at least in the first attempt, *not* correct (some remain wrong eternally and sell nonetheless): they do not fulfil the contract. Often it is not formulated precisely either. This however should not be taken as the cause for the many "programming mistakes", at least not in general. The reason is the undisciplined habit in which programming is carried out by young and old all over the country.

Similarly, approaches for a rigorous treatment of programming languages have been developed: the key idea is to describe their semantics in a precise mathematical framework [64]. This provides a general framework to think and reason about programs — the machine-oriented side of software development, so to speak.

These techniques allow to develop formal methods to avoid or detect implementation errors in software systems; the most rigorous of those is *formal verification*: both the desired behaviour (abstract specification) and the implementation (executable specification) are formalised in a suitable logical calculus, and the equivalence of both is proved.

Applying this procedure on paper only is unsatisfactory: numerous technical details are a drudgery to deal with, and worse, subtle details could escape the attention of a human reviewer. This suggests to use a proof assistant for proof checking and automation.

A consequent next step is to mechanise the transition from logic to executable code: code generation. This allows to translate a certain class of specifications directly to corresponding code respecting the original specification.

The component which carries out this translation, the *code generator*, is critical: an erroneously implemented code generator could produce code which does *not* respect the original specification. Thus the code generator must be developed with enough diligence in order to be trusted and reliable.

The development and presentation of such a code generator is the purpose of this thesis. Beside the purely scientific results we also hopefully provide a tool which helps bridge the gap between logic and programming and thus opens up new applications for formal techniques in software engineering.

1.2 Contributions

Our overall aim is to bring the worlds of theorem proving and functional programming closer together:

Generic principles. We want to give a precise foundation of code generation in terms of ingredients of the underlying logical calculus without dependencies on a particular implementation.

Concrete system. We are not interested in developing a sophisticated new calculus on paper only but to pragmatically extend an existing environment to obtain a practically usable system.

Similarly, contributions are both conceptual and technical:

- We give a precise characterisation of code generation by shallow embedding. This characterisation, the *foundation* of code generation, is kept as simple as possible and as rich as necessary. It is elaborated on the meta-theoretic level once and for all and not extended later. Previously code generation has usually been considered a “trivial syntactic transformation”; consequently, existing code generator approaches base to a large extent on *intuition* and *folklore*. Our investigation will show that shallow embedding is a very generic approach to code generation which even provides a simple concept of datatype abstraction.

- The logical calculus we will use provides type classes in the manner of Haskell 1.0 [26], which we consider for code generation to support overloading; this clarifies the relationship between the operational and logical aspect of type classes.
- As proof assistant we choose *Isabelle/HOL*, an implementation of Church's higher-order logic [16]. We provide a code generator which translates logical descriptions in higher-order logic to a state-of-the-art functional programming language containing the typed λ-calculus as subset (see §2.4.3); concrete instances of such languages are *SML*, *OCaml* and *Haskell*.
- The code generator interacts closely with other parts of the system, in particular existing deductive infrastructure. This allows to extend the range of executable constructs dramatically while leaving the foundation of the code generator unchanged.

Our choice for *Isabelle/HOL* is motivated by the following observations:

- Higher-order logic is quite near to functional programming, following the equation "higher-order logic = functional programming plus logic". Programmers familiar with functional programming get acquainted with *Isabelle/HOL* rather fast.
- Higher-order logic allows to express many programming paradigms inside the same framework and to establish different views on the same logical concept (see §4.1.3).
- *Isabelle/HOL* includes a user interface which facilitates interaction with the proof assistant, thus relieving the user from many technical details.
- Abundant successful projects close to real-world programs have been carried out with higher-order logic over the years (e.g. semantics of Java-like languages [32], calculations forming part of the proof of the Kepler conjecture [43], a compiler for a subset of C [35]).

Although restricting to functional programming languages might be considered too limited, we argue that especially *Haskell* has proved able to absorb low-level issues seamlessly into a purely functional world (notably imperative data structures, I/O [30], concurrency, transactions [27]).

The thesis is structured as follows:

- This introduction continues which a presentation of related work.
- The relevant foundation are set out in §2.
- §3 is dedicated to the principles of the code generator, its architecture and its foundations.
- The usability of the system is bolstered by various examples in §4.
- A conclusion §5 sketches future extensions.

1.3 Related work

Most state-of-the-art theorem proving systems support some form of code generation. There are two fundamental code generation principles:

Shallow embedding: Types in the programming language are identified with types in the logic, functions in the programming language with constants in the logic; code generation is nothing else than the inverse image of that identification. This works best for logics which are already close to functional programming languages in structure and expressiveness . The translation by the code generator is usually conceptually simple.

Proof extraction: Proof terms are animated in the spirit of the Curry-Howard isomorphism [19]. That is, proofs are interpreted constructively. Traditionally this approach is applied in logics with a rich (dependent) type system. Thus the translation is more involved, since the type system of the logic is much more expressive than that of a functional programming language.

Let us illustrate these two principles by examining three typical representatives.

1.3.1 Calculus of inductive constructions — *Coq*

The *Coq* [57] proof assistant is based on the calculus of inductive constructions [9], a dependent type theory where types, terms and proof terms are syntactically represented uniformly.

Due to its logic, *Coq* is a natural candidate for proof extraction. Here an example how subtraction of natural numbers can be specified:

```
(* A Lemma *)

Theorem le_Sm_n_pred:
  forall m n: nat, S m <= n -> { q : nat | S q = n }.
proof.
  let m: nat, n: nat.
  per cases on n.
    suppose it is 0.
      assume (S m <= 0).
      hence thesis by (le_Sn_0 m).
    suppose it is (S q).
      thus thesis using exists q; reflexivity.
  end cases.
end proof. Qed.

(* The Main Theorem *)

Theorem exists_minus:
  forall m n: nat, m <= n -> { q : nat | m + q = n }.
proof.
  let m: nat.
  per induction on m.
    suppose it is 0.
      let n: nat.
      thus thesis using simpl; exists n; reflexivity.
    suppose it is (S q) and hyp: thesis for q.
      let n: nat.
      assume Sq_n: (S q <= n).
      then ex_r: {r : nat | S r = n} by (le_Sm_n_pred q n).
      consider r such that (S r = n) from ex_r.
```

```
        then n_Sr: (n = S r).
        then (S q <= S r) by Sq_n.
        then (q <= r).
        then ex_s: {s : nat | q + s = r} by hyp.
        consider s such that qsr: (q + s = r) from ex_s.
        thus (S q + s = n) by n_Sr, plus_Sn_m, qsr.
      end induction.
    end proof. Qed.
```

The specification is given as a dependent type which for each pair of natural numbers m and n with $m \leq n$ yields a natural number q such that the $m + q = n$. The existence of a member of this type is witnessed by a proof which essentially is an induction on an existential proposition. From this the following *Haskell* code can be extracted:

```
module Coq_nat_extraction where

import qualified Prelude

__ = Prelude.error "Logical or arity value used"

false_rect :: () -> a1
false_rect _ =
  Prelude.error "absurd case"

false_rec :: () -> a1
false_rec _ =
  false_rect __

data Nat = O
         | S Nat

type Sig a = a
  -- singleton inductive, whose constructor was exist

le_Sm_n_pred :: Nat -> Nat -> Nat
le_Sm_n_pred m n =
  case n of
    O -> false_rec __
    S h -> h

exists_minus :: Nat -> Nat -> Nat
exists_minus _main_arg n =
  case _main_arg of
    O -> n
    S h -> exists_minus h (le_Sm_n_pred h n)
```

Notice how the proof of an existential proposition is turned into the computation of the corresponding witness, while an induction is turned into a recursion.

Curry-Howard notwithstanding, extraction of code from proofs is delicate since it has to re-separate the categories types, terms (with computational content) and proofs (without computational content) which coincide in the logical calculus but not in executable code [57]. In particular, the user must decide before starting a formal development which parts of it should to be executable and must take this into account both during definitions and proofs.

Code generation in *Coq* also works using shallow embedding, e.g. by primitive recursion which is denoted in *Coq* by the `Fixpoint` statement.

```
Fixpoint minus (n : nat) (m : nat) {struct m} : nat :=
  match n, m with
    | n, O => n
```

```
      | O, m => O
      | S n, S m => minus n m
    end.
```

Here is the corresponding code:

```
module Coq_nat_translation where

import qualified Prelude

data Nat = O
         | S Nat

minus :: Nat -> Nat -> Nat
minus n m =
  case n of
    O -> (case m of
            O -> n
            S n0 -> O)
    S n0 -> (case m of
               O -> n
               S m0 -> minus n0 m0)
```

1.3.2 *ACL2*

ACL2 is, in its own words, "both a programming language in which you can model computer systems and a tool to help you prove properties of those models" [31]. It appears to the user as a purely functional fragment of *Common Lisp* [39], which is also its implementation language. Also the user interface is modelled in the manner of a *Lisp* toplevel, allowing direct evaluation of functions.

Due to its very nature, a distinguished code generator functionality is not necessary for *ACL2*: specified programs can be run directly on *Common Lisp* systems. This can be seen as a shallow embedding drawn to its ultimate consequence. The absence of explicit proofs rules out proof extraction.

As an example we give here functions for appending and concatenating lists, which bear no surprise for programmers familiar with the typical *Lisp*-style *nil*/*cons* lists, together with a theorem stating that the reverse of the reverse of a list is the list itself:

```
(defun app (xs ys)
  (if (endp xs) ys (cons (car xs) (app (cdr xs) ys))))

(defthm app_xs_nil
  (implies (true-listp xs) (equal (app xs nil) xs)))

(defthm app-assoc
  (equal (app (app xs ys) zs)
    (app xs (app ys zs))))

(defun rev (xs)
  (if (endp xs) nil (app (rev (cdr xs)) (cons (car xs) nil))))

(defthm rev-true_listp
  (implies (true-listp xs) (true-listp (rev xs))))

(defthm rev-app-commute
  (implies (true-listp ys)
    (equal (rev (app xs ys)) (app (rev ys) (rev xs)))))
```

```
(defthm rev-involutary
  (implies (true-listp xs) (equal (rev (rev xs)) xs)))
```

This examples exhibits key concepts of *ACL2*:

- There is no static type system, and *ACL2* is total. Practically, this means that `(rev (cons 42 1705))` is a valid expression which can be evaluated according to the semantics in the logic to `(cons 42 nil)`. This requires certain type-like assumptions such as "is a proper list" to be encoded explicitly into propositions, e.g. in the formulation of theorem `rev-involutary`. *ACL2* has a concept named *guards* which has no logical relevance but reconciliates the totality of *ACL2* with the semantics of *Common Lisp*: guards are assertions on function arguments and results which can be checked statically for consistency among a set of functions. Consistent guard annotations guarantee that these functions behave the same way in the logic as under evaluation in *Common Lisp*.
- *ACL2* features a different interaction paradigm than other provers: sophisticated automation and proof planning facilities allow the user to "define" theorems, and the system gives a detailed complaint when it is not able to prove them on its own.

1.3.3 Higher-order logic

Higher-order logic (HOL) is based on works by Church [16] and Gordon [22]. It combines a simply-typed λ-calculus with logical connectives such as implication and quantifiers. There are many implementations of *HOL* available. Classical *HOL* proof assistants (e.g. *HOL4* [53], *HOL Light* [28]) expose their implementation language (*SML* or *OCaml*) as a *meta-language* to the user which is used for combining proof tactics etc. The HOL implementation of our particular interest, *Isabelle/HOL* [42], deviates from that tradition by providing a distinguished specification and proof language, *Isar*, restricting the use of *SML* to system development proper [15].

HOL is already quite near to a functional programming language. Thus code generation via shallow embedding is a natural procedure. Previously to the work presented here, a code generator for *SML* did already exist [7]: given the *Isabelle/HOL* specification

datatype $nat = Zero \mid Succ\ nat$

fun $minus :: nat \Rightarrow nat \Rightarrow nat$ **where**
$\quad minus\ n\ Zero = n$
$\mid minus\ Zero\ m = Zero$
$\mid minus\ (Succ\ n)\ (Succ\ m) = minus\ n\ m$

Isabelle/HOL generated the following *SML* program:

```
datatype nat = Succ of nat | Zero;

fun minus (Succ n) (Succ m) = minus n m
  | minus Zero (Succ v) = Zero
  | minus n Zero = n;
```

In this simple example code generation superficially appears as a naive syntactic transformation. Our work supersedes the existing code generator by a far more general approach.
Another particularity of *Isabelle/HOL* is that it provides optional proof terms which allow also for proof extraction (see further §4.6.1).

1.4 A note on notation

The concepts presented in this thesis range over different layers; to provide a minimum of orientation, we distinguish two of them with different notational conventions:

The abstract layer: This consists of plain text like the sentence you are currently reading, containing semi-formal statements like ***Definition*** or ***Synopsis***. It includes logical concepts without reference to a particular implementation; typically this is typeset in *italics* or sanserif.

The concrete layer: This covers *Isar* theory text and *SML*, *OCaml* or *Haskell* source text; in the case of *Isar* we adopt the best-style typesetting of theories; for the programming languages we write `typewriter`. In some cases we do not give explicit *Isar* theory text which shows how to accomplish a formal development in detail but merely quote results (types, terms, theorems, ...) in *italics*.

This thesis is written using the typesetting facilities of *Isabelle*. In particular this means that the ingredients of the concrete layer have been formally checked by the system itself, thus reducing the risk of typos.

Further conventions:

- We abbreviate vectors of symbols $s_1, s_2, \ldots, s_n$ by $\overline{s}_n$; if n is irrelevant we write just $\overline{s}$. Such vectors are neither tuples nor lists but *shallow* in the sense that they disappear in any context which allows for a “flattening”, e.g. $f\ \overline{x}_n$ abbreviates $f\, x_1 \ \ldots\ x_n$, not $f\ (x_1, \ldots, x_n)$. Also zip comprehensions are used: $\overline{x} \otimes \overline{y}$ implies that both $\overline{x}$ and $\overline{y}$ have same length and denotes the vector $(x_1 \otimes y_1) \ \ldots\ (x_n \otimes y_n)$, where $\otimes$ is an arbitrary infix operator.

- A wildcard pattern _ denotes an anonymous variable occurring only once in an expression.

- Angle brackets $\langle \ldots \rangle$ help to separate formal notations belonging to different levels, e.g. $\langle f\ x = x\rangle \in A$ denotes that the proposition $\langle f\ x = x\rangle$, *not* the boolean value $f\ x = x$, is contained in set A.

CHAPTER **2**

Foundations

Do you pine for the days when men were men
and wrote their own device drivers?
Linus Torvalds, operating system architect, from:
Just for Fun: The Story
of an Accidental Revolutionary.

We give an overview over relevant characteristics of the *Isabelle/HOL* system. Equipped with this logical foundations we introduce an equational logics framework which provides the formal base for a treatment of code generation by shallow embedding.

Contents

2.1 The logical framework *Isabelle/Pure*

Isabelle [46] is a generic proof assistant designed for interactive reasoning in a variety of logics, notably higher-order logic and set theory. All these are implemented on top of the *logical framework Isabelle/Pure* (for short, *Pure*). This architecture allows to reuse infrastructure applicable to different calculi (*object logics* in *Isabelle* terminology). Indeed, *Pure* is a versatile framework for applications involving formal methods and logic.

In this chapter we give a synopsis of *Pure*'s characteristics that are relevant in our context. This involves a considerable amount of formal notation; for better orientation §A gives a short reference on this.

2.1.1 Logical expressions

Synopsis 1 (the* Pure *logic)

The logical calculus of *Pure* is a minimalistic higher-order logic of simply-typed schematically polymorphic λ-terms; in other words, there are three categories of logical expressions:

types τ consist of *type constructors* κ with a fixed arity and type variables α:

$$\tau ::= \kappa\ \tau_1 \cdots \tau_k \mid \alpha$$

Function space $\alpha \Rightarrow \beta$ is simply a binary type constructor with right-associative infix syntax.

terms t include application, abstraction, (local) variables of a particular type, and *constants*:

$$t ::= t_1\ t_2 \mid \lambda x{::}\tau.\ t \mid x{::}\tau \mid f$$

proofs are abstract derivations; resulting propositions are identified with terms of a distinguished type *prop*, containing

implication $P \Longrightarrow Q$
and universal *quantification* $\bigwedge x{::}\tau.\ P\ x$, where by convention outermost quantifiers can be omitted.

$\alpha\beta\eta$-equivalence is implicit. Concluded proofs are *theorems*. In derivations *axioms* and *theorems* are represented uniformly as *proof constants* [5]. We do not give much attention to proof terms or proof text, denoting their presence simply by $\langle proof \rangle$.

Constants are schematically polymorphic, meaning that each constant is assigned a most general type scheme $f :: \forall \alpha_1 \ldots \alpha_n.\ \tau$. This schematic polymorphism carries over to proof constants (though we do not use any explicit notation for this).

Notationally, we adhere to the following conventions:

- Typing contexts are avoided by assuming consistent type annotations for local variables x; for conciseness they can be omitted.

- Type schemes are closed, i.e. in $f :: \forall\, \alpha_1 \ldots \alpha_n.\ \tau$, the set $\{\alpha_1, \ldots, \alpha_n\}$ is *exactly* the set of type variables in τ listed in a canonical order.
- If necessary, constant types are clarified either by an explicit type annotation $f{::}\tau$, or by System-F-style type instantiations $f\ [\tau_1, \tau_2, \ldots, \tau_n]$ with respect to f's most general type scheme $f :: \forall\, \alpha_1\ \alpha_2 \ldots \alpha_n.\ \tau$.
- The notation $\tau\ [\tau_1, \tau_2, \ldots, \tau_n]$ denotes a substitution on the type level where τ contains exactly n (distinct) type variables which according to a canonical order are replaced by the type arguments $\tau_1, \tau_2, \ldots, \tau_n$.
- All presented formal entities are well-typed (with respect to an implicit context).

Conceptually types are implicit due to Hindley-Milner type inference; proofs are irrelevant (§2.1.3). Thus, in type theoretic parlance, the structure of the logic is determined by terms depending on terms $\lambda x{::}\tau.\ t$, proofs depending on terms $\bigwedge x{::}\tau.\ P\ x$, and proofs depending on proofs $P \Longrightarrow Q$.

2.1.2 Theory extensions

In informal mathematics, theories are developed *incrementally*, by enriching an implicit global context with definitions and theorems (where either may depend on previous definitions and theorems). For formal reasoning, this must be made explicit.

Pure provides a notion of hierarchical *theories* Θ. The type of theories is an extensible sum type, containing logical and extra-logical data [63]. Thus Θ can be thought of as a tuple consisting of an unbounded but finite number of components $\Theta = (C_1, C_2, C_3, \ldots, C_n)$. If in a particular theory Θ a particular judgement A holds, we write $\Theta \vdash A$. Since any such judgement A is induced by a finite number of components $C_l1, C_l2, \ldots, C_lm$ in Θ, it is also legitimate to write $(C_l1, C_l2, \ldots, C_lm) \vdash A$, ignoring the other components in the theory. The same situation can be denoted by $\Theta = (C_l1, C_l2, \ldots, C_lm, \ldots) \vdash A$.

In the implementation of *Pure*, the different components C_k are just data in the *SML* environment; if necessary we will represent them here by partial functions.

Recall that the logical expressions may contain three kinds of global named entities: type constructors κ, constants f and theorems. We represent each of these by the following judgements with respect to Θ:

Synopsis 2 (basic logical propositions)

type arities $\Theta = (\Upsilon, \ldots) \vdash \kappa :: * \to \cdots \to *$ map a type constructor κ to its arity, which we give here as flat kind. Type arities are managed as function Υ which maps a type constructor to its arity.

constant types $\Theta = (\Omega, \ldots) \vdash f :: \forall\, \alpha_1 \ldots \alpha_n.\ \tau$ associate a constant f with its most general type scheme $\forall\, \alpha_1 \ldots \alpha_n.\ \tau$. Constant types are managed as a function Ω which maps a constant to its type scheme.

theorem propositions $\Theta \vdash a : P$ associate a proof constant a with its corresponding proposition P; i.e. the name of a proof constant also serves as name of the corresponding proved proposition.

The initial *Pure* theory Θ_0 which is the base of all theory developments already contains some ingredients, notably

type of propositions
$\Theta_0 \vdash \mathit{prop} :: *$
function space
$\Theta_0 \vdash (\Rightarrow) :: * \to * \to *$
equality
$\Theta_0 \vdash (\equiv) :: \ \forall\alpha.\ \alpha \Rightarrow \alpha \Rightarrow \mathit{prop}$
rule of reflexivity
$\Theta_0 \vdash \mathit{reflexive} : \bigwedge x.\ x \equiv x$
rule of symmetry
$\Theta_0 \vdash \mathit{symmetric} : \bigwedge x\ y.\ x \equiv y \Longrightarrow y \equiv x$
rule of transitivity
$\Theta_0 \vdash \mathit{transitive} : \bigwedge x\ y\ z.\ x \equiv y \Longrightarrow y \equiv z \Longrightarrow x \equiv z$
combination rule
$\Theta_0 \vdash \mathit{combination} : \bigwedge f\ g\ x\ y.\ f \equiv g \Longrightarrow x \equiv y \Longrightarrow f\ x \equiv g\ y$
abstraction rule
$\Theta_0 \vdash \mathit{abstraction} : \bigwedge x\ y.\ x \equiv y \Longrightarrow \lambda z.\ x \equiv \lambda z.\ y$

Others may be introduced by means of *theory extensions*, particular schemes of adding new data to Θ. In our theory context model, theory extensions manifest as monotonic extension of one or more underlying components, i.e. component functions are assigned values at formerly undefined input values. This monotonicity guarantees that theory extensions themselves are conservative, i.e. if a judgement holds ($\Theta \vdash A$), it also holds after a theory extension ($\Theta' \vdash A$).

Synopsis 3 (basic theory extensions)

constant definition constdef f_def: $(f :: \tau\ [\overline{\alpha}]) :\equiv t$

adds a new constant with $f :: \forall\overline{\alpha}.\ \tau$ with a corresponding theorem $f\_def : f \equiv t$, given that t does not contain free variables, the set of type variables in t is exactly $\{\overline{\alpha}\}$, and f does not occur in t. Monotonicity implies that f has not yet been introduced.

theorem definition theorem a: $P\ \langle \mathit{proof} \rangle$

adds a new theorem with $a : P$, where $\langle \mathit{proof} \rangle$ does not refer to a.

Both schemes are easy to justify: theorem definitions can be inlined by replacing each reference to a by P, similarly constant definitions can be inlined by replacing each f in a term to t and each f_def in a proof to reflexivity $\bigwedge x.\ x \equiv x$. In other words, both kinds of extension can be eliminated in an extra-logical step; therefore they are *definitional* theory extensions.

Further important extension schemes:

Synopsis 4 (further theory extensions)

type declaration typedecl $\kappa :: * \to \cdots \to *$

adds a new type constructor κ with a fixed arity.

constant declaration constdecl $f :: \forall \overline{\alpha}.\ \tau$

adds a new constant with $f :: \forall \overline{\alpha}.\ \tau$ which is not specified further.

overloaded definitions overload f_κ_def: $(f :: \tau\ [\kappa\ \overline{\alpha}]) :\equiv t$

adds a new definition for an existing constant, given that t does not contain free variables, the set of type variables in t is exactly $\overline{\alpha}$, there has been no previous definition for f with the same or a more general type, and occurrences of f in t refer either to any α or to other instances $\kappa'\ \overline{\tau}$ such that corresponding definitions do not refer to κ .... Together with type classes, these definitions allow for overloading in the manner of *Haskell* 1.0 (see [24] for details).

These theory extensions can be proved consistency-preserving on the meta-theoretic level. Other extension schemes put on the user the burden to ensure that no nonsense is introduced; these are called *axiomatic* and meant to be used rarely, mainly for defining object logics. As the most generic axiomatic scheme:

Synopsis 5 (axiom declaration)

axiom declaration axiom a: P

adds a new theorem $a : P$.

2.1.3 Putting on the *LCF* glasses

Isabelle is an *LCF*-style proof assistant [21]. In traditional *LCF*-style systems, proofs are not recorded explicitly to save memory. Only the propositions are kept as values of an abstract datatype `thm`. Primitive inferences are implemented as *ML* functions operating on the concrete representation of values of type `thm`; the corresponding program module is referred to as the *LCF kernel*.

This has two consequences: From an implementation point of view, each sophisticated deduction is composed of primitive inferences; indeed, one typical discipline in *LCF*-style proof assistant is to implement advanced deductions by breaking them down to primitive inferences, leaving the logical foundations untouched. From a conceptual point of view, this requires *proof irrelevance*: the properties of every theorem are completely specified by means of its proposition. In particular, whether a proof constant $foo : f \equiv t$ is an axiom, a definition or a derived theorem does not matter. Similarly, there is no distinction between primitive rules of the framework and derived ones.

Isabelle deviates from the classical *LCF* style by optionally providing explicit proof terms. Thus, though most parts of the system follow the principle of proof irrelevance, there are some proof-dependent applications where the *construction* of definitions and theorems actually matters, notably extraction of programs from proofs (§4.6.1) and elimination of overloading (§4.6.2); but this is outside the core calculus.

2.1.4 A glimpse at the *Isar* language

The interaction between the *Isabelle* system and the user happens through the *Isar* language. An *Isar* text is structured as a series of *Isar commands*, each consisting of a particular keyword followed by text denoting logical and non-logical entities (types, terms, names, ...). When processed incrementally, each command performs a corresponding *transaction* on an underlying state.

A major class of commands issues theory updates; for example, there is a command **definition** providing a user-view to primitive definitions:

definition $K :: \alpha \Rightarrow \beta \Rightarrow \alpha$ **where**
$K\_def\colon K\ x\ y \equiv x$

Concrete *Isar* syntax deviates from the abstract notation we have used so far, notably the use of postfix notation for type application. This **definition** produces the following theory updates internally:

constdef $K\_def\_raw\colon (K :: \alpha \Rightarrow \beta \Rightarrow \alpha) :\equiv \lambda x\ y.\ x$
theorem $K\_def\colon \bigwedge x\ y.\ K\ x\ y \equiv x\ \ \langle proof\rangle$

Note that the free variables in the specification as given by the user are implicitly generalised.

Similarly, theorems can be added using the **lemma** command:

lemma K_equals:
$x \equiv y \Longrightarrow K\ x\ z \equiv K\ y\ w$
unfolding K_def **by** (*rule reflexive*)

The theorem is stated as a proposition, accompanied by *proof text* that builds the corresponding derivation. That proof text typically contains references to *proof commands* and *proof methods* which perform certain reasoning steps. Here, we do **unfolding** of the theorem $\bigwedge x\ y.\ K\ x\ y \equiv x$ and conclude the proof by using the *rule* of reflexivity $\bigwedge x.\ x \equiv x$; we do not take further heed of the proof text here, but see §2.2.3. The resulting theory update follows:

theorem $K\_equals\colon \bigwedge x\ y\ z\ w.\ x \equiv y \Longrightarrow K\ x\ z \equiv K\ y\ w\ \ \langle proof\rangle$

Again, free variables are implicitly generalised.

The commands **theorem** and **corollary** are synonyms for **lemma** but allow to emphasise different levels of relevance for theorems as a formal comment.

2.2 The *Isabelle/HOL* system

2.2.1 *Isabelle/HOL* as extension of *Isabelle/Pure*

Isabelle/HOL is an implementation of higher-order logic as an *extension* of *Pure*: types of *Pure* are identified with types in *Isabelle/HOL* (for short, *HOL*); *HOL* introduces a distinguished type *bool* with constants *True* and *False* denoting truth values in connection with typical logical connectives like implication ($\longrightarrow$), quantification $\forall\, x.\ P\ x$, $\exists\, x.\ P\ x$ and equality ($=$), of which logical equivalence ($\longleftrightarrow$) is a special case. Terms of type *bool* are embedded into *prop* by a judgement *Trueprop* :: *bool* $\Rightarrow$ *prop* which is usually inserted automatically and not printed by the syntax layer. Sets on type α by convention are represented as predicates of type $\alpha \Rightarrow$ *bool* with comprehension syntax $\{x.\ P\ x\}$ and membership operator ($\in$). This object logic also provides a consistency-preserving type introduction primitive:[1]

Synopsis 6 (*HOL *type definition)

type definition typedef $\kappa\ \overline{\alpha} = \{x{::}\tau.\ P\ x\}$ $\langle proof\rangle$

adds a new type constructor κ as a copy of an existing type τ (with the same parameters $\overline{\alpha}$), restricting the representable values x to those satisfying the predicate P. Since the meta-theory of higher-order logic demands types to be non-empty, a witness that κ is inhabited must be provided.

Built on top of these basic ingredients, *HOL* provides further notions: products $\alpha \times \beta$ and pairs $(x,\ y)$; natural numbers *nat* with Peano-style constructors 0 and *Suc* n; lists α *list* with constructors $[\![]\!]$ and $x : xs$, together with list enumeration syntax $[\![x_1,\ x_2,\ x_3,\ \ldots]\!]$. These and corresponding operations are fairly standard and will be used without detailed explanations; but see §B for a quick overview.

2.2.2 The *Isabelle/HOL* toolbox

Using the type definition and constant definition primitives, *HOL* provides *derived* definition schemes which internally are mapped down to primitive ones but provide a more comfortable interface to the user. We give the most important ones here by example with concrete *Isar* syntax.

Inductive definitions (**inductive**) allow to specify a predicate with introduction rules:

inductive *partition* :: $(\alpha \Rightarrow bool) \Rightarrow \alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list \Rightarrow bool$ **where**
$partition\ f\ [\![]\!]\ [\![]\!]\ [\![]\!]$
$\mid f\ x \Longrightarrow partition\ f\ xs\ ys\ zs \Longrightarrow partition\ f\ (x : xs)\ (x : ys)\ zs$
$\mid \neg\ f\ x \Longrightarrow partition\ f\ xs\ ys\ zs \Longrightarrow partition\ f\ (x : xs)\ ys\ (x : zs)$

Internally, they are based on fixed point constructions using the Knaster-Tarski fixed point theorem [47]. Corresponding elimination and induction rules are derived.

Inductive datatypes (**datatype**) correspond to datatype declarations in functional programming language:

datatype $expr = Number\ nat \mid Var\ string$
$\mid Plus\ expr\ expr$ (**infixl** $\oplus$ 65) $\mid Times\ expr\ expr$ (**infixl** $\otimes$ 65)

Constructors satisfy the logical characterisations of injectivity and distinctness, e.g.

[1] Consistency-preserving relative to the *HOL* standard models which are only a subset of all *Pure* models; there are *Pure* models which are incompatible with typedef!

$\bigwedge expr1\ expr2\ expr1'\ expr2'.\ expr1 \oplus expr2 = expr1' \oplus expr2' \longleftrightarrow expr1 = expr1' \wedge expr2 = expr2'$

$\bigwedge expr1'\ expr2'\ list.\ expr1' \otimes expr2' \neq Var\ list$

Datatypes are internally constructed by an appropriate inductive predicate together with a type definition.

For each datatype, a corresponding recursion combinator is constructed which allows for *primitive recursive functions* (**primrec**) on the structure of datatypes:

$$\begin{array}{l}\textbf{primrec}\ reverse :: \alpha\ list \Rightarrow \alpha\ list\ \textbf{where} \\ \quad reverse\ [\![]\!] = [\![]\!] \\ \mid reverse\ (x : xs) = reverse\ xs\ @\ [\![x]\!]\end{array}$$

A huge class of terminating *recursive functions with pattern matching* (**fun**) are definable using the construction of explicit function graphs [33]:

$$\begin{array}{l}\textbf{fun}\ map2 :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha\ list \Rightarrow \beta\ list \Rightarrow \gamma\ list\ \textbf{where} \\ \quad map2\ f\ [\![]\!]\ \_ = [\![]\!] \\ \mid map2\ f\ \_\ [\![]\!] = [\![]\!] \\ \mid map2\ f\ (x : xs)\ (y : ys) = f\ x\ y : map2\ f\ xs\ ys\end{array}$$

The attentive reader may note that **datatype** and **primrec**/**fun** form the nucleus of a functional programming language embedded into *HOL*. Although it will become apparent in §3.1.3 that code generation itself by no means depends on these specification mechanisms, they are indispensable tools for anybody who wants to *write* functional programs in *HOL* without tinkering with low-level constructions such as typedef, constdef etc.

In §4.2.3 we also explain how to turn inductive definitions into executable specifications, thus extending the programming language to a functional-logic language.

2.2.3 Example proof: The natural numbers are well-founded

We give a small, non-trivial example of a *Isar* theory development; its primary purpose is to give a taste of how *Isar* developments work out in practice.

$$\textbf{datatype}\ nat = Zero \mid Succ\ nat$$

A copy of the natural numbers is specified as datatype: each natural number is either *Zero* or successor (*Succ*) of another natural number.[2]

$$\begin{array}{l}\textbf{primrec}\ less\ (\textbf{infix} \prec 50)\ \textbf{where} \\ \quad m \prec Zero \longleftrightarrow False \\ \mid m \prec Succ\ n \longleftrightarrow (case\ m\ of\ Zero \Rightarrow True \mid Succ\ m \Rightarrow m \prec n)\end{array}$$

[2]Natural numbers are an integral part of the *HOL* distribution, as is also everything shown in this section.

We specify the "is less than" relation by means of primitive recursion; the operation is given infix syntax ($\prec$).

lemma *less_self*:
$n \prec Succ\ n$
by ($induct\ n$) *simp_all*

A lemma: each natural number is less than its successor. The primary proof device is natural induction via method *induct* which fits nicely with the primitive recursion scheme. The *simp_all* method invokes the *simplifier* which performs automated equational reasoning.

The next proof is already a bit more involved:

lemma *less_SuccE*:
$m \prec Succ\ n \Longrightarrow m \prec n \vee m = n$

We state: If a natural number m is less than the successor of another number n, then either m is less than n or m is equal n.

proof ($induct\ m\ arbitrary{:}\ n$)

The proof again opens by induction on m with the additional requirement to generalise over n. This leaves us with two sub-propositions to prove:

1. $\bigwedge n.\ Zero \prec Succ\ n \Longrightarrow Zero \prec n \vee Zero = n$
2. $\bigwedge m\ n.\ (\bigwedge n.\ m \prec Succ\ n \Longrightarrow m \prec n \vee m = n) \Longrightarrow$
   $Succ\ m \prec Succ\ n \Longrightarrow Succ\ m \prec n \vee Succ\ m = n$

On the proof text level, the hypothetical parts of those emerge as **fix** and **assume**, while the conclusions emerge as **show**.

fix n
show $Zero \prec n \vee Zero = n$ **by** ($cases\ n$) *simp_all*
next
fix $m\ n$
assume $\bigwedge n.\ m \prec Succ\ n \Longrightarrow m \prec n \vee m = n$
and $Succ\ m \prec Succ\ n$
then show $Succ\ m \prec n \vee Succ\ m = n$ **by** ($cases\ n$) *simp_all*
qed

The proof of the desired induction theorem is given here in full:

lemma *less_induct*:
$(\bigwedge n.\ (\bigwedge m.\ m \prec n \Longrightarrow P\ m) \Longrightarrow P\ n) \Longrightarrow P\ n$
proof −
assume *wellfounded*: $\bigwedge n.\ (\bigwedge m.\ m \prec n \Longrightarrow P\ m) \Longrightarrow P\ n$
have $\bigwedge q.\ q \prec Succ\ n \Longrightarrow P\ q$
proof ($induct\ n$)
fix q
have R: $P\ Zero$ **by** (*rule wellfounded*) *simp*
assume $q \prec Succ\ Zero$
then have $q = Zero$ **by** ($cases\ q$) *simp_all*

```
    with R show P q by simp
  next
    fix n q
    assume step: ⋀q. q ≺ Succ n ⟹ P q
    assume q ≺ Succ (Succ n)
    then have q ≺ Succ n ∨ q = Succ n using less_SuccE by blast
    then show P q
    proof
      assume q ≺ Succ n then show P q by (rule step)
    next
      assume q = Succ n
      then show P q by (auto intro: wellfounded step)
    qed
  qed
  with less_self show P n by auto
qed
```

Without going into detail, the primary ingredients of the *Isar* proof language can be glimpsed at:

- Pending subgoals of the form $\bigwedge \overline{x}.\ \overline{P\,\overline{x}} \Longrightarrow Q\,\overline{x}$ correspond to **fix** $\overline{x}$ **assume** $\overline{P\,\overline{x}}$ **show** $Q\,\overline{x}$; the choice of names is up to the writer.
- Proofs are conducted stepwise, accumulating facts ("**have**") where each deduction is carried out by a subproof.
- Full-blown sub-proofs are bracketed by **proof** and **qed**, where both may apply a *method*, e.g. *induct*.
- **by** denotes a degenerate subproof: **by** *method*1 *method*2 is short for **proof** *method*1 **qed** *method*2.
- The block-structure of subproofs is implicit.

For more details see [8].

2.3 Type classes

2.3.1 Syntactic properties

A characteristic property of the *Pure* logic are *type classes* [24]. These correspond to type classes in their classical formulation in Haskell 1.0 [60]. *Pure* type classes can also be interpreted as an instance of order-sorted algebra [41]. The admissible extension of the calculus is accomplished as follows:

- *Sorts* are added as a further level of logical expressions; sorts s are (possibly empty) intersections of finitely many *classes* c: $s ::= c_1 \cap \ldots \cap c_l$.
- Sorts are represented canonically as minimal intersections of finitely many classes, ordered according to a total order of classes.
- $\top$ denotes the empty class intersection, the *top sort*.

- Type variables are decorated by consistent sort annotations: $\tau ::= \kappa\ \tau_1 \ldots \tau_k \mid \alpha{::}s$; sort constraints are sometimes omitted in the text, especially if the sort is $\top$.

This induces the following judgements:

Synopsis 7 (rules for order-sorted algebra)

subclass relation $\mho$ maps a class c to the set of its direct superclasses $\{c_1, \ldots, c_k\}$.

arity signature Σ maps a pair of type-constructor κ and class c to their corresponding sort arguments $\overline{s}_n$. Such type-constructor/class pairs are called *instances* and written as c_κ.

subsort

$$\frac{c \in s}{\mho \vdash c \subseteq s} \qquad \frac{c' \in s \qquad c' \in \mho\ c}{\mho \vdash c \subseteq s} \qquad \frac{\mho \vdash c_1 \subseteq s \qquad \cdots \qquad \mho \vdash c_n \subseteq s}{\mho \vdash c_1 \cap \cdots \cap c_n \subseteq s}$$

well-sorted

$$\frac{(\mho,\, \Sigma) \vdash \tau_1 :: s_1 \qquad \cdots \qquad (\mho,\, \Sigma) \vdash \tau_n :: s_n \qquad \Sigma\ c_\kappa = \overline{s}_n}{(\mho,\, \Sigma) \vdash \kappa\ \overline{\tau}_n :: c}\ \text{constr}$$

$$\frac{}{(\mho,\, \Sigma) \vdash (\alpha{::}(\cdots \cap c \cap \cdots)) :: c}\ \text{var} \qquad \frac{(\mho,\, \Sigma) \vdash \tau :: c' \qquad c \in \mho\ c'}{(\mho,\, \Sigma) \vdash \tau :: c}\ \text{classrel}$$

$$\frac{(\mho,\, \Sigma) \vdash \tau :: c_1 \qquad \cdots \qquad (\mho,\, \Sigma) \vdash \tau :: c_n}{(\mho,\, \Sigma) \vdash \tau :: c_1 \cap \cdots \cap c_n}\ \text{sort}$$

The *subsort* relation $s_1 \subseteq s_2$ lifts the primitive subclass relation induced by $\mho$ to sorts according to the rules of transitivity and intersection. Primitive instance relations (*arities*) induced by Σ are lifted to well-sortedness judgements $\tau :: s$ in virtue of the rules of *well-sortedness.*

Both $\mho$ and Σ are components of the theory context $\langle \Theta = (\ldots,\, \mho,\, \Sigma,\, \ldots) \rangle$ and have to obey some well-formedness conditions:

- $\mho$ is *acyclic*, i.e. the transitive closure of the primitive subclass relations induced by $\mho$ is cycle-free; in consequence $\subseteq$ is antisymmetric.

- Σ is *coregular*: if $\Sigma\ c_\kappa = \overline{s}_n$, then for all superclasses c' of c (i.e. $c \subseteq c'$) holds that $\Sigma\ {c'}_\kappa = \overline{s'}_n$, where each ${s'}_k$ in $\overline{s'}_n$ is a supersort of the corresponding s_k in $\overline{s}_n$ (i.e. $s_k \subseteq {s'}_k$). Coregularity guarantees important meta-theoretic properties such as most general unifiers [41] as well as the possibility of dictionary construction (see §3.2.7).

- $\mho$ is *minimal* in the sense that it does not contain transitive edges, e.g. if $c_1 \in \mho\ c_2$ and $c_2 \in \mho\ c_3$, then $c_1 \notin \mho\ c_3$ since this edge is subsumed. In other words, $\mho$ forms a Hasse diagram.

Note that subclassing itself is no essential property of order-sorted algebra: by *expanding* a sort s to the complete sort $\bigcap c.\ c \subseteq s$, the subclass relation becomes irrelevant since the sort representation subsumes all potential superclasses.

2.3.2 Logical interpretation

The type class properties presented so far allow to use type classes for syntactic restriction of type instantiation. However such purely syntactic type classes are rarely used in practice. Type classes gain their usefulness when we assign a logical meaning to them:

- Types τ are embedded as terms by means of a type constructor $itself :: * \to *$ and an unspecified constant $TYPE$ of type $\forall\,\alpha.\ \alpha\ itself$. By convention we write the term $TYPE$ of type $\tau\ itself$ as $TYPE\ \tau$.
- Each type class c is logically interpreted by attaching a constant C with type $\forall\,\alpha.\ \alpha\ itself \Rightarrow prop$ which acts as a predicate for the judgement $\tau :: c$. As suggestive notation we write the proposition $C\ (TYPE\ \tau)$ as $(\!|\tau :: c|\!)$.
- This notation lifts to sorts naturally: $(\!|\tau :: c_1 \cap \cdots \cap c_n|\!) \equiv (\!|\tau :: c_1|\!) \wedge \cdots \wedge (\!|\tau :: c_n|\!)$.

What we want to achieve is implicit reasoning with type classes: if $\tau :: s$ is derivable then also $(\!|\tau :: s|\!)$ holds. For this sake we ensure that for each $c_1 \in \mho\ c_2$ and $\Sigma\ c_\kappa = \overline{s}_n$ respectively, the corresponding logical witnesses

$$(\!|\alpha :: c_2|\!) \Longrightarrow (\!|\alpha :: c_1|\!)$$

$$(\!|\beta_1 :: s_1|\!) \Longrightarrow \ldots \Longrightarrow (\!|\beta_n :: s_n|\!) \Longrightarrow (\!|\kappa\ \overline{\beta}_n :: c|\!)$$

are provided by means of the following definitional theory extension schemes for logically interpreted type classes:

Synopsis 8 (theory extensions for order-sorted algebra)

class definition classdef $c \subseteq c_1 \cap \ldots \cap c_n$: $P\ [\alpha]$

adds a new class c as subclass of $c_1 \cap \ldots \cap c_n$ (by updating the underlying $\mho$ at point c with $\{c_1, \ldots, c_n\}$), together with the following logical steps:

constdef c_def: $(\!|\alpha :: c|\!) :\equiv P\ [\alpha] \wedge (\!|\alpha :: c_1|\!) \wedge \ldots \wedge (\!|\alpha :: c_n|\!)$
theorem c_axiom: $P\ [\alpha :: c]$ $\langle proof\rangle$
theorem c_c_1: $(\!|\alpha :: c|\!) \Longrightarrow (\!|\alpha :: c_1|\!)$ $\langle proof\rangle$
theorem ...
theorem c_c_n: $(\!|\alpha :: c|\!) \Longrightarrow (\!|\alpha :: c_n|\!)$ $\langle proof\rangle$

The proofs for the logical witnesses follow easily from the definition of $(\!|\alpha :: c|\!)$.

instance definition instance $\kappa :: (\overline{s})\ c$ $\langle proof\rangle$

proves the theorem

theorem c_κ: $(\!|\kappa\ \overline{\beta{::}s}_n :: c|\!)$ $\langle proof\rangle$

using the proof given for instance, from where follows the witness $(\!|\beta_1 :: s_1|\!) \Longrightarrow \ldots \Longrightarrow (\!|\beta_n :: s_n|\!) \Longrightarrow (\!|\kappa\ \overline{\beta}_n :: c|\!)$ easily. Then Σ at point c_κ is updated to $\overline{s}$ (given that Σ remains coregular).

Since the inference rules for well-sortedness judgements $\tau :: s$ mimic the corresponding deductions on predicate judgements $(\!|\tau :: s|\!)$, these extension schemes guarantee that $\tau :: s$ and $(\!|\tau :: s|\!)$ are interchangeable; in other word, the following inferences are admissible [62]:

$$\frac{}{(\!|(\alpha{::}c) :: c|\!)}$$

$$\frac{P\ [\alpha :: s]}{(\!|\alpha :: s|\!) \Longrightarrow P\ [\alpha :: \top]}$$

Note that we have not introduced sort constraints on the type parameters of type schemes. The reason is that, given the logical interpretation of type classes above, these bear no logical relevance: it is always safe, given a constant $f :: \forall\,\alpha.\ \tau$, to write down terms containing f in an unconstrained manner. In contrast, for theorems there is a difference: $P\ (f\ [\alpha{::}\top])$ *cannot* be derived from $P\ (f\ [\alpha{::}c])$. Though sort constraints on type schemes play no role in the foundation of the calculus, for the benefit of the user the type checker allows to declare sort constraints for constants.

Observe that on the foundational level the typical association of classes and constants (class parameters) known from Haskell is not present; it emerges in the user-view (see §2.3.3).

The matter of type classes exhibits an inherent oddity of *Pure*: though instance judgements explicitly refer to type constructors κ, the logic itself does not provide an extension scheme to introduce new κs with specific properties, only unspecified κ by means of type declarations. Nonetheless, derived object logics may provide mechanisms to introduce semantically meaningful type constructors, like *HOL* typedef (see §2.2.1).

2.3.3 End-user view

The user is seldom exposed to the foundation of type classes sketched so far. The user interface for type classes treats them as a special case of abstract algebraic specifications (*locales* in *Isar* terminology [2, 25, 24]). We need not go into details here but merely present the look-and-feel to the end-user, using the following example taken from algebra:

class *semigroup* =
  fixes *mult* :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$ (**infixl** $\otimes$ 70)
  assumes *assoc*: $(x \otimes y) \otimes z = x \otimes (y \otimes z)$

Classes are introduces using the **class** statement. The class is specified given hypothetical operations (**fixes**, class parameters) together with hypothetical properties (**assumes**, class axioms). These are immediately lifted to a global constant $(\otimes) :: \alpha \Rightarrow \alpha \Rightarrow \alpha$ with constraint $\alpha{::}\mathit{semigroup}$ together with a corresponding theorem $\bigwedge(x{::}\alpha{::}\mathit{semigroup})\ (y{::}\alpha{::}\mathit{semigroup})\ z :: \alpha{::}\mathit{semigroup}.\ (x \otimes y) \otimes z = x \otimes y \otimes z.$

Here, the typical association of class parameters $(\otimes)$ to a class (*semigroup*) emerges. It is managed by a context component function ω mapping classes to constant names, in our case: $\omega\ \mathit{semigroup} = \{(\otimes)\}$.

Instantiation of a type class consists of two parts: giving appropriate specifications for the class parameters and proving that these satisfy the class axioms:

instantiation *nat* :: *semigroup*
begin

In this example the specification of the class parameter specialised on type *nat* is given by **primrec**.

primrec *mult_nat* :: $\mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$ **where**
  $(0{::}\mathit{nat}) \otimes n = n$
  $\mid \mathit{Suc}\ m \otimes n = \mathit{Suc}\ (m \otimes n)$

Internally **instantiation** takes care that this specification is realised by an overloaded definition (in this case, overload *mult_nat_def*: $(\otimes)\ [\mathit{nat}] :\equiv \ldots$).

Before concluding the target, a proof is carried out that the given specifications respects the class specification (command **instance**):

instance proof
  fix $m\ n\ q :: \mathit{nat}$
  show $m \otimes n \otimes q = m \otimes (n \otimes q)$
    by (*induct m*) *auto*
qed

end

Subclassing is specified by giving a list of superclasses (here, *semigroup*), together with further specification elements:

class *monoid* = *semigroup* +
  fixes *neutral* :: α (**1**)
  assumes *neutl*: $\mathbf{1} \otimes x = x$
    and *neutr*: $x \otimes \mathbf{1} = x$

2.4 A framework for describing code generation

Now we turn our attention to a formal description of code generation by shallow embedding. For this purpose we use equational logic in higher-order rewrite systems (HRS) as an abstract unified view on both logic and programming language.

In the following direct references to the languages *ML* and *Haskell* are made. These are mainly for illustration; the described principles could easily transferred to other languages like *Scala* [44], *Python* [58], *Scheme* [56] or F# [49], although no implementation effort has been done so far in that direction; see Definition 10 for a precise characterisation of the necessary language properties.

2.4.1 Higher-order rewrite systems

Synopsis 9 (higher-order rewrite systems)

HRSs [38] describe an equivalence relation $\approx$ on λ-terms induced by a set of equations E modulo $\alpha\beta\eta$-conversion; each equation $lhs \equiv rhs$ in E may contain free variables such that free variables in lhs are a superset of the free variables in rhs. Due to $\beta\eta$-conversion we can assume that both lhs and rhs are in η-long normal-form.

Particular instances of HRSs are referred to by their set of equations E.

HRSs provide two interchangeable views:

Reduction systems: A term t is rewritten by applying an equation $lhs \equiv rhs$ from E to a subterm s in t, which means to instantiate $lhs \equiv rhs$ to $lhs' \equiv rhs'$ such that lhs' is s and then replace s by rhs', resulting in u. This is written as $E \Vdash t \longrightarrow u$.

Equational logic: The equivalence relation $\approx$ is defined as follows:

$$\frac{\langle t \equiv u\rangle \in E}{E \Vdash \sigma\ t \approx \sigma\ u}\ \text{axiom}\ \ \text{(where } \sigma \text{ is a term substitution)}$$

$$\frac{}{E \Vdash t \approx t}\ \text{refl} \qquad \frac{E \Vdash u \approx t}{E \Vdash t \approx u}\ \text{sym} \qquad \frac{E \Vdash t \approx v \qquad E \Vdash v \approx u}{E \Vdash t \approx u}\ \text{trans}$$

$$\frac{E \Vdash t \approx u \qquad E \Vdash v \approx w}{E \Vdash t\ v \approx u\ w}\ \text{comb} \qquad \frac{E \Vdash v \approx w}{E \Vdash \lambda x.\ v \approx \lambda x.\ w}\ \text{abs}$$

In HRSs α-equivalent terms are identified. We may assume w.l.o.g. that after each reduction step $\langle E \Vdash t \longrightarrow u\rangle$, u is in long $\beta\eta$-normal-form, which is reached by β-normalising and then η-expanding; then the following holds: $E \Vdash t \approx u$ is equivalent to $E \Vdash t \longleftrightarrow^* u$, where $\longleftrightarrow^*$ is the symmetric transitive reflexive closure of $\longrightarrow$. This justifies to use both the single-step operational view $\longrightarrow$ and the abstract view $\approx$ interchangeably, depending which is more appropriate in a particular context.

For terms in our HRSs, we choose the *Pure* term language; terms are well-formed with respect to an implicit typing context $(\Upsilon,\ \Omega,\ \mho,\ \Sigma)$. For succinctness we omit outermost quantifiers in equations, in other words, free variables in equations are implicitly bound.

Observe the strict separation of the two symbols $\equiv$ and $\approx$: $\equiv$ serves as separator between the left-hand side and the right-hand side of directed equations, whereas $\approx$ denotes the equivalence relation induced by stepwise application of those rewrite rules; in a certain sense, the equations in E denote the *static* or *compile-time* view on the HRS, whereas the equivalence $t \approx u$ represents a particular *dynamic* or *runtime* instance of that HRS.

2.4.2 *Pure* as a HRS

Pure is trivially suitable to simulate a HRS:

- Take equational theorems $\bigwedge \overline{x}.\ t \equiv u$ as equations $t \equiv u$ in E.
- Using *Pure* equality $\equiv$ as the equivalence relation $\approx$, the inference rules for equational logic are all primitive *Pure* inferences as listed in §2.1.2; the *axiom* rule is trivial since equations and equivalences coincide and substitutions of rules are admissible.

2.4.3 HRSs as model for target languages

We now use HRSs to develop a model for target languages which abstracts from enough details to provide a firm base for code generation.

So far we have used the term "target language" in a loose fashion; the following definition characterises which properties a programming language has to supply to serve as target language:

Definition 10 (target language)

A (fragment of a) programming language is a *target language* if the following properties hold:

1. Expressions of the language are expressive enough to embed the typed λ-calculus and its semantics.
2. Evaluating terms representable in the typed λ-calculus leads again to terms representable in the typed λ-calculus.
3. A facility for syntactic pattern matching is provided.
4. Syntactically, a program can be seen as a list of *statements*.
5. Semantically, a (well-formed) program P can be described reasonably as a HRS with equations E_P, where each statement contributes a (finite) number of equations to E_P.

For illustration of this definition, have a look a the following diagram depicting an *SML* program:

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

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

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

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

```
Suc (Suc Zero_nat)
```

This (well-formed) program consists of three statements `datatype nat`, `fun plus_nat` and `fun sum`; the latter two induce equations which form a HRS which, in the example, normalises `sum [Suc Zero_nat, Suc Zero_nat]` to `Suc (Suc Zero_nat)`.

Definition 10 refrains from using parts of the language which have no straightforward interpretation in the typed λ-calculus; this rules out catchable exceptions, imperative data structures, etc. Also any kind of structuring devices like name spaces, abstraction principles, visibility rules, module systems, etc., are not considered.

For *ML*-like languages, the requirements of Definition 10 are surely reasonable: points 1, 2 and 3 are satisfied, and programs in essence consist of `val`/`fun`/function definitions which are equations (points 4, 5).

Definition 11 (target language semantics)

Given a target language as characterised by Definition 10, the semantics of any well-formed program P in that language is given by a HRS by means of point 5 in Definition 10.

This definition already establishes an abstraction level: a HRS does not define any notion of rule precedence, evaluation order etc., whereas a target language is likely to do so. So, while a particular HRS might permit a couple of reduction sequences for a given term, it is not stated which of these the corresponding program will perform. Since we will only guarantee partial correctness for generated programs, this is admissible.

2.4.4 Code generation using shallow embedding

Considering code generation using shallow embedding, the HRS abstraction level from the previous section yields the following conclusions:

- When translating *Pure* to a target language, point 1 in Definition 10 allows us to silently identify terms. Formally this identification can be described by a suitable implicit morphism on terms. For the moment we ignore type classes; later dictionary construction will allow to eliminate type classes explicitly (see §3.2.7).
- Point 2 guarantees that each term stemming from an evaluation in the target language has an inverse image in *Pure*.
- HRSs may serve as a common abstract view on both *Pure* and target languages.

Before we establish this common abstract view, an auxiliary definition classifying a relationship between two HRSs:

Definition 12 (compatibility)

A HRS E_1 is *compatible* with another HRS E_2 if for each derivation $E_1 \Vdash t \longrightarrow^* u$ also $E_2 \Vdash t \longrightarrow^* u$ holds, and vice versa.

Obviously, the relation "compatible with" is an equivalence relation.

Definition 13 (code generation)

Given a *Pure* theory Θ with a set of equations E_Θ, we call a target language program P the program *generated from* E_Θ if P is well-formed and E_P is compatible with E_Θ.

In other words, a generated program can be seen as the *implementation* of a *Pure* system of equations, as visualised in the following picture:

$$
\begin{tikzcd}
 & E_\Theta \Vdash \mathrm{t} \arrow[r, "*"] \arrow[d, leftrightarrow] \arrow[dl, dashed, "\text{code generation}"'] & \mathrm{u} \arrow[d, leftrightarrow] \\
P \arrow[r, "\text{semantics}"] & E_P \Vdash \mathrm{t} \arrow[r, "*"] & \mathrm{u}
\end{tikzcd}
$$

Compatibility guarantees that each equivalence $E_P \Vdash t \approx u$ stemming from a run of $E_P \Vdash t \longrightarrow^* u$ can be simulated by E_Θ; thus partial correctness of generated programs is guaranteed. If E_Θ and E_P are equal, compatibility holds trivially; however the definition of compatibility provides enough freedom to cope with slightly different but appropriately related E_Θ and E_P, which is a necessity if the term language of E_P is richer than that of E_Θ (see §3.2.6 and §3.2.7).

Definition §13 formulates requirements for a program P generated from E_Θ but does not state how such a P shall be constructed. This will be the focus of the next chapter.

Code generation via shallow embedding only employs notions which have a direct representation in the logic, e.g. terms, types, equations or patterns; it does not state anything about concepts which are not represented *within* the logic but appear by inspecting the logic from *outside*: evaluation order, termination, complexity. Our humble restriction to partial correctness relieves us from dealing with those non-logical aspects. Obviously, it would be desirable to have a smaller semantic gap, but practically there are two obstacles:

- There might be no formal specification at all, e.g. for *Haskell*.
- Even the existence of a rigorous standard does not promise that it is implemented in any real system, as is the case for *SML*.

For this reasons we are content with the abstraction we have gained using the HRS model and concentrate on the conclusions and possibilities following from that; indeed, our equational logic view is widely accepted as "morally correct" e.g. in the *Haskell* community [18].

CHAPTER **3**

Code generation

We do not retreat from reality, we rediscover it.
C. S. Lewis, British author, from:
On Stories, and Other Essays on Literature

Starting with the equational logics framework from the the last chapter, we introduce the code generator's architecture. The transition from *Pure* logic to an abstract program is given in detail, with special diligence dedicated to the treatment of type classes. We illustrate how this abstract system is put to work in collaboration with the *HOL* system.

Contents

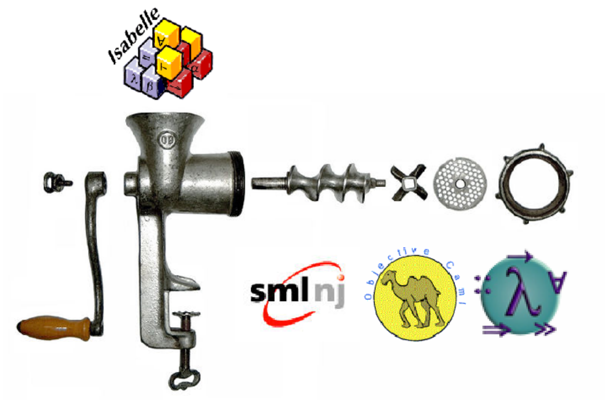
3.1 Towards a concrete code generator

As stated in the last chapter, the essential problem of code generation is to turn a system of equations E_Θ into a corresponding program P satisfying definition §13. In the following sections we introduce a suitable framework in a top-down manner, starting with the big picture and ending with details; this streamlined presentation sometimes comes with the necessity to leave certain issues to the intuition of the reader at a certain stage, postponing the discussion of subtleties to a later point.

We only guarantee partial correctness. On the one hand, this relieves us from lots of technical problems; on the other hand, in that sense a program P yielding *no* equations is always correct. It is at the discretion of the user to make something "meaningful" or "practically applicable" out of code generation. This issue will be taken up in §3.5, leading to a couple of example applications in §4.

The deliberate simplicity of most aspects in the following sections has not been established *a priori* but rather evolved from a continuous process of reconsideration and elimination of superfluous concepts. Arguably, the main achievement is not what is required but what has been left out while retaining a practically usable system.

3.1.1 *Pure* and *HOL*



More has to be said about the relationship of *Pure* and *HOL* concerning code generation. The *foundation* of the code generator, i.e. the relationship between abstract logic and executable code, is completely explained inside *Pure*. Since *HOL* is an *extension* of *Pure*, each *HOL* theory is also a *Pure* theory and thus accessible to code generation directly. Even more, in most practical applications the *user* of the code generator is not expected to juggle with raw *Pure* ingredients in order to derive code from them, although this is possible of course; instead, *HOL* is likely to be used, including its powerful specification and automation tool box. For this sake different scattered components of the *HOL* system have been tailored to work smoothly together to accomplish a practically usable system.

The superficial ambivalence *Pure* vs. *HOL* has also an impact in presentation: abstract generic considerations will use *Pure* equality ($\equiv$), while most concrete examples will use *HOL* equality ($=$). This lifting is admissible since $t = t'$ implies $t \equiv t'$.

3.1.2 Patterns and code equations

We will syntactically restrict the kind of equations E_Θ in the HRS we examine; as motivation, have a look at these equational theorems:

$$\begin{aligned}&\mathit{inv} :: (\mathit{int} \Rightarrow \mathit{int}) \Rightarrow \mathit{int} \Rightarrow \mathit{int}\\ &\mathit{inv}\ (\lambda k.\ k + 1) = (\lambda k.\ k - 1)\end{aligned}$$

$$\begin{aligned}&(+) :: \mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{int}\\ &\bigwedge k\ l\ r :: \mathit{int}.\ k + (l + r) = (k + l) + r\end{aligned}$$

The following "*Haskell*" fragments correspond with those:

```
inv :: (Integer -> Integer) -> Integer -> Integer
inv (\k -> k + 1) = (\k -> k - 1)

plus :: Integer -> Integer -> Integer
plus k (plus l r) = plus (plus k l) r
```

Although syntactically these equations are perfect translations, they are not valid *Haskell*: the left-hand side of the equations contain abstractions and non-constructor constants, which is not allowed. Therefore E_Θ shall only contain equations of a particular syntactic shape; to describe these, we need an auxiliary definition:

Definition 14 (pattern)

A *pattern* is a term which is either a variable or a constant from a set of constructors Ξ which is fully applied to arguments which themselves are patterns. To distinguish patterns from general terms, we write p instead of t, and C instead of f. More explicitly, $(\Omega, \Xi) \vdash \mathit{pattern}\ p$ denotes that p is a pattern relative to Ω and Ξ.

Whenever patterns are involved, the set of constructors Ξ is treated as implicit part of the typing context $(\Upsilon, \Omega, \mho, \Sigma, \Xi)$. The choice of the constructor set Ξ has no logical relevance; its purpose is to guarantee that all constants showing up in terms which are restricted to patterns are constructors and will therefore not come into conflict with the strict requirements of patterns in target languages.

Definition 15 (code equation)

An equation $f\ [\overline{\tau}]\ \overline{p} \equiv t$ is called a *code equation* with *head f*, *arguments* $\overline{p}$ and *right-hand side t* if the following syntactic properties hold:

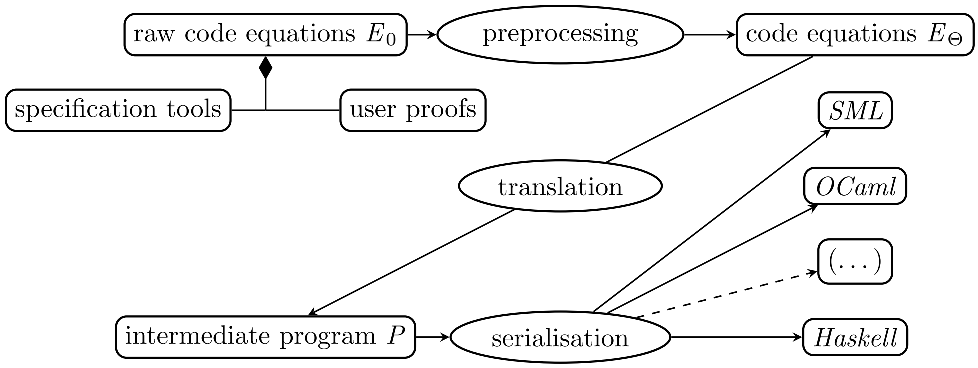
1. the $\overline{p}$ are patterns,
2. all free variables of t occur in $\overline{p}$,
3. all free type variables of t occur in $\overline{\tau}$,
4. no free variable occurs more than once in $\overline{p}$ (left-linearity),
5. if there exists a class c such that $f \in \omega\ c$, then $[\overline{\tau}]$ is a singleton $[\kappa\ \overline{\alpha{::}s}]$; otherwise the $[\overline{\tau}]$ are all distinct type variables $[\overline{\alpha{::}s}]$.

This syntactic restrictions are the same as found in function definitions of typical functional programming languages; the different treatment of code equations referring to class parameters resembles overloading as accomplished by *Isabelle* type classes (cf. §2.3.3).

3.1.3 Architecture overview

Synopsis 16 (code generator architecture)

The code generator itself consists of three major components carrying out three steps sequentially as follows:



- Starting point is a collection of raw code equations E_0 in a *Pure* theory Θ; due to proof irrelevance their origin does not matter, but typically this will be either a specification tool or an explicit proof by the user.
- Before these raw code equations E_0 are continued with, they can be subjected to theorem transformations. This *preprocessor* is an interface which allows to apply the full expressiveness of ML-based theorem transformations to code generation; motivating examples are shown in §3.3.3. The result of the preprocessing step is a structured collection of *code equations* E_Θ.
- These code equations are *translated* to a program P in an abstract intermediate language. Conceptually this covers the whole transition from logic to code and satisfies the requirements of Definition 13, as will be elaborated in §3.2.
- Finally, the abstract program P is *serialised* into concrete source code of a target language. This step only produces concrete syntax but does not change the program in essence; all conceptual transformations occur in the translation step. Therefore we will not consider concrete target languages at all but discuss any meta-theory at the level of the abstract intermediate language.

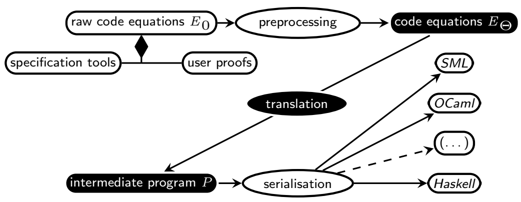
There are two conflicting requirements for a code generator: *trustability* and *flexibility*. Simplicity increases the first but prevents the latter; additional functionality endangers the first. The key to reconcile both can be found in the architecture:

- The architecture clearly separates translation, which is conceptually involved but technically simple, from serialisation, which is technically involved but conceptually simple. These two steps are kept to the essential minimum.
- The preprocessing step allows to retain flexibility and to obtain a practically

usable system: arbitrary transformations can be carried out on raw code equations, allowing for optimisation etc. All these transformations are guarded by *LCF* inferences and do not endanger trustability; this can be seen as an adaptation of the traditional *LCF* paradigm to code generation (cf. §2.1.3).

3.2 An abstract intermediate language

3.2.1 Motivation



The abstract intermediate language plays a central role in the whole process since it marks the transition between (formally checked) logical entities and target language source code. Its purpose is to capture the essence of target languages while still abstracting from technical, target-language specific details, motivated by three observations:

- By capturing the essence of target languages once and for all, code infrastructure is shared conveniently among different target languages.
- An intermediate language facilitates examination of properties of the translation since it provides a rest point before the "dirty" and diverse world of a target language is entered. One could also think about a formalisation of its properties.
- A key difference between logic and target languages is the following: logical entities from a theory Θ like equations and declaration are constructed and derived in a certain way, but after this has been accomplished, the construction is not relevant any longer (cf. §2.1.3). On the opposite, target languages naturally require an explicit construction in a program.

To illustrate the last issue, two small counterexamples for *SML* "programs" which would not compile, but their fictive inverse images are easily constructible:

```
fun g x y = y;
fun f x = f (g (h x) x);
```

does not compile since the constant `h` mentioned in the function definition of `f` is not present. A series of declarations yielding a suitable theory Θ could look like this:

constdecl $h :: \forall\,\alpha.\ \alpha \Rightarrow \alpha$
constdef g_def: $(g :: \alpha \Rightarrow \beta \Rightarrow \beta) :\equiv \lambda x\ y.\ y$
constdef f_def: $(f :: \alpha \Rightarrow \alpha) :\equiv \lambda x.\ x$
theorem g_code: $g\ x\ y \equiv y\ \ \langle proof\rangle$
theorem f_code: $f\ x \equiv f\ (g\ (h\ x)\ x)\ \ \langle proof\rangle$

using the theorems *g_code* and *f_code* as equations for E_Θ.

The second "program"

```
fun f (x, y) = (f x, f y);
```

would not typecheck in *SML* due to type circularity. With overloading this can be easily accomplished in *HOL*:

constdecl $f :: \forall\,\alpha.\ \alpha \Rightarrow \alpha$
overload $f\_{\times}\_def$: $(f :: \alpha \times \beta \Rightarrow \alpha \times \beta) :\equiv \lambda p.\ (f\ (fst\ p),\ f\ (snd\ p))$
theorem $f\_{\times}\_code$: $f\ (x,\ y) \equiv (f\,x,\ f\,y)\ \ \langle proof\rangle$

So, the translation from logic to intermediate language is a process of "coagulation" which groups free-floating equations and logical entities from Θ to an intermediate program P. Thereby enough "structure" is added so that the final transition step to a target language program does not produce "garbage" as in the examples above, but well-formed programs. This step is common to all target languages, thus it is reasonable to place it at this stage.

3.2.2 Definition

We define the intermediate language as a kind of "Mini-Haskell" which coagulates free-floating logical entities onto four statements: *data* for datatypes, *fun* for functions, *class* and *inst* for type classes and overloading.

Definition 17 (statements of the intermediate language)

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

fun $f :: \forall\,\overline{\alpha{::}s}_k.\ \tau$ *where*
$f\ [\overline{\alpha{::}s}_k]\ \overline{p_1} = t_1$
$\mid \ldots$
$\mid f\ [\overline{\alpha{::}s}_k]\ \overline{p_n} = t_n$

class $c \subseteq c_1 \cap \cdots \cap c_m$ *where*
$f_1 :: \forall\,\alpha.\ \tau_1,\ \ldots,\ f_n :: \forall\,\alpha.\ \tau_n$

inst $\kappa\ \overline{\alpha{::}s}_k :: c$ *where*
$f_1\ [\kappa\ \overline{\alpha{::}s}_k] = t_1,\ \ldots,\ f_n\ [\kappa\ \overline{\alpha{::}s}_k] = t_n$

Terms are *Pure* terms; later we will extend the term language with local pattern matching (§3.2.6).

A notable deviation to *Haskell* is that *inst* only permits one single equation per class parameter; this is merely for technical reasons since it facilitates dictionary construction (§3.2.7). In practice this is not really a restriction, see §3.3.2.

In the next section we will equip the intermediate language with

- a definition of well-formedness,
- an equational semantics describing how an intermediate program P yields a HRS E_P.

These are chosen in a way that they mirror the given well-formedness requirements and equational semantics of target languages; since serialisation does not involve any conceptual transformations but only produces concrete syntax, each well-formed intermediate program can be serialised to a well-formed target-language program with the *same* equational semantics.

3.2.3 Well-formed programs and their semantics

Informally, *well-formedness* demands that in a program everything "fits together". It is a *forward-link* to concrete target languages which guarantees that the result of serialisation is a compilable target language program.

For this reason it is sufficient to understand the intermediate language intuitively as a fragment of *Haskell* where *data*, *class* and *inst* correspond to `data`, `class` and `instance` respectively and `fun` is syntactic sugar for function bindings. Explicit dictionary construction (cf. §3.2.7) will explain how *class* and *inst* statements can be eliminated by intermediate programs; the remaining *data* and *fun* statements map to *SML* and *OCaml* programs in a straightforward manner.

We define well-formedness by associating each statement with *prerequisites* and *results*; *prerequisites* of a statement are judgements which must be well-formed for a statement to be well-formed; *results* are typing declarations which can be assumed for the whole program to hold if the statement is well-formed. These declarations are similar to theory declarations as introduced in §2.4 and §2.3. Contrary to a theory Θ whose number of components is extensible, we are now in a fixed setting and thus the typing context is not an extensible sum but a fixed tuple $\Gamma_P = (\Upsilon_P, \Omega_P, \Xi, \mho_P, \Sigma_P, \omega_P)$ whose components roughly correspond to those of theories Θ with the following functions as components:

Υ_P maps type constructors κ to their arity $* \rightarrow \ldots \rightarrow *$.

Ω_P maps constants f to their most general type $\forall\, \overline{\alpha{::}s}_k.\ \tau$; unlike as in *Pure*, sort constraints are an integral part of the type since they have operational relevance.

Ξ denotes the set of all constants f that are constructors; recall that the choice of Ξ does not contribute to the semantics, but is relevant for well-formedness since in target languages the distinction between constructors and non-constructors is essential.

$\mho_P$, Σ_P, ω_P are the subclass relation, the arity signature and the constant-to-class association as described in §2.3.

Prerequisites and results for the four kinds of statements are as follows:

Definition 18 (prerequisites and results)

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

prerequisites $\overline{\tau_1}, \ldots, \overline{\tau_n}$

results $\Upsilon_P\ \kappa = \overline{*}_k \rightarrow *$,
$\Omega_P\ f_1 = \forall\, \overline{\alpha}_k.\ \overline{\tau_1} \Rightarrow \kappa\ \overline{\alpha}_k, \ldots, \Omega_P\ f_n = \forall\, \overline{\alpha}_k.\ \overline{\tau_n} \Rightarrow \kappa\ \overline{\alpha}_k$,
$f_1 \in \Xi, \ldots, f_n \in \Xi$

fun $f :: \forall\,\overline{\alpha{::}s}_k.\ \overline{\tau} \Rightarrow \tau$ *where*
$f\ [\overline{\alpha{::}s}_k]\ \overline{p_1} = t_1$
$|\ \ldots$
$|\ f\ [\overline{\alpha{::}s}_k]\ \overline{p_n} = t_n$

prerequisites $\overline{p_1} :: \overline{\tau},\ \ldots,\ \overline{p_n} :: \overline{\tau},\ t_1 :: \tau,\ \ldots,\ t_n :: \tau,$
$\mathit{pattern}\ \overline{p_1},\ \ldots,\ \mathit{pattern}\ \overline{p_n}$

results $\Omega_P\ f = \forall\,\overline{\alpha{::}s}_k.\ \overline{\tau} \Rightarrow \tau$

class $c \subseteq c_1 \cap \cdots \cap c_m$ *where*
$f_1 :: \forall\,\alpha.\ \tau_1,\ \ldots,\ f_n :: \forall\,\alpha.\ \tau_n$

prerequisites $c_1,\ \ldots,\ c_m,\ \tau_1,\ \ldots,\ \tau_n$

results $\mho_P\ c = \{c_1,\ \ldots,\ c_m\},\ \omega_P\ c = \{f_1,\ \ldots,\ f_n\},$
$\Omega_P\ f_1 = \forall\,\alpha{::}c.\ \tau_1,\ \ldots,\ \Omega_P\ f_n = \forall\,\alpha{::}c.\ \tau_n$

inst $\kappa\ \overline{\alpha{::}s}_k :: c$ *where*
$f_1\ [\kappa\ \overline{\alpha{::}s}_k] = t_1,\ \ldots,\ f_n\ [\kappa\ \overline{\alpha{::}s}_k] = t_n$

prerequisites $\kappa :: \overline{*}_k \to *,\ s_1,\ \ldots,\ s_k,\ c,$
$\omega_P\ c = \{f_1,\ \ldots,\ f_n\},$
$f_1 :: \forall\,\alpha{::}c.\ \tau_1,\ \ldots,\ f_n :: \forall\,\alpha{::}c.\ \tau_n,$
$t_1 :: \tau_1\ [\kappa\ \overline{\alpha{::}s}_k],\ \ldots,\ t_n :: \tau_n\ [\kappa\ \overline{\alpha{::}s}_k]$

results $\Sigma_P\ c_\kappa = \overline{\alpha{::}s}_k$

Results of the *data* statement effectively restrict the type signatures of constructors. Prerequisites and results of the *inst* statement couple together arities $\kappa\ \overline{\alpha{::}s}_k :: c$ and type parameters $[\kappa\ \overline{\alpha{::}s}_k]$ of overloaded code equations.

Definition 19 (programs and well-formedness)

A set P of statements is called a *program.* A program is *well-formed* if *all* prerequisites of *all* statements in P are typeable in the typing context Γ_P induced by *all* results stemming from *all* statements in P *plus* the initial context declaration of the function space $\Upsilon_P\ (\Rightarrow) = * \to * \to *$.

This description may seem more complicated then necessary but it accomplishes recursion and mutual mutual recursion (between *fun*s, *data*s, *inst*s) quite easily. The initial presence of function space $(\Rightarrow)$ means that the function spaces of *Pure*, the intermediate language and (by way of consequence) the target language are identified.

The equational semantics of a program is now easily captured:

Definition 20 (semantics of a program)

Let P be a well-formed program. Then its semantics is given as a pair $(\Gamma_P,\ E_P)$ where Γ_P is the typing context induced by *all* results stemming from *all* statements in P and E_P is the set of all equations contained in *fun* and *inst* statements.

3.2.4 A correct translation

A translation can be modelled by a partial function transl from a set of code equations E_Θ to an intermediate program P; the definition of correctness follows directly from Definition 13:

Definition 21 (correct translation)

A translation transl is *correct* if for any input argument E_Θ it either fails or yields a well-formed P such that E_P is compatible with E_Θ.

In the context of the whole code generation procedure, compatibility is a *backlink* from intermediate language to logic. Compatibility is implied by the following two properties:

- E_P is syntactically the same as E_Θ.
- Each expression occurring in E_P is also typeable in Θ.

The first is easily accomplished. Concerning the second, according to the definition of E_P, each expression in E_P also occurs in P; well-formedness of P implies that each expression in E_P is well-typed with respect to Γ_P. Therefore it is sufficient to guarantee that each type judgement derivable wrt. Γ_P is also derivable wrt. Θ.

For this purpose, each statement in a program P is associated with a *certificate* in the logic; such a certificate consists of a bunch of equations and statements about the context:

Definition 22 (certificates for intermediate statements)

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

$$\begin{aligned}
\textbf{context}\ \ & \Upsilon\ \kappa = \overline{*}_k \to * \\
& \Omega\ f_1 = \forall\,\overline{\alpha}_k.\ \overline{\tau_1} \Rightarrow \kappa\ \overline{\alpha}_k \\
& \ldots \\
& \Omega\ f_n = \forall\,\overline{\alpha}_k.\ \overline{\tau_n} \Rightarrow \kappa\ \overline{\alpha}_k
\end{aligned}$$

$\textsf{\textit{fun}}\ f :: \forall\,\overline{\alpha{::}s}_k.\ \tau\ \textsf{\textit{where}}$
$\quad f\ [\overline{\alpha{::}s}_k]\ \overline{p_1} = t_1$
$\mid \ldots$
$\mid f\ [\overline{\alpha{::}s}_k]\ \overline{p_n} = t_n$

$$\begin{aligned}
\textbf{equations}\ \ & f\ [\overline{\alpha{::}s}_k]\ \overline{p_1} \equiv t_1 \\
& \ldots \\
& f\ [\overline{\alpha{::}s}_k]\ \overline{p_n} \equiv t_n \\
\textbf{context}\ \ & \Omega\ f = \forall\,\overline{\alpha}_k.\ \tau
\end{aligned}$$

$\textsf{\textit{class}}\ c \subseteq c_1 \cap \cdots \cap c_m\ \textsf{\textit{where}}$
$\quad f_1 :: \forall\,\alpha.\ \tau_1,\ \ldots,\ f_n :: \forall\,\alpha.\ \tau_n$

$$\begin{aligned}
\textbf{context}\ \ & \mho\ c = \{c_1,\ \ldots,\ c_m\} \\
& \omega\ c = \{f_1,\ \ldots\ f_n\} \\
& \Omega\ f_1 = \forall\,\alpha.\ \tau_1 \\
& \ldots \\
& \Omega\ f_n = \forall\,\alpha.\ \tau_n
\end{aligned}$$

inst $\kappa\ \overline{\alpha{::}s_k}$ :: c *where*
$f_1\ [\kappa\ \overline{\alpha{::}s_k}] = t_1,\ \ldots,\ f_n\ [\kappa\ \overline{\alpha{::}s_k}] = t_n$

equations $f_1\ [\kappa\ \overline{\alpha{::}s_k}] \equiv t_1$
$\ldots$
$f_n\ [\kappa\ \overline{\alpha{::}s_k}] \equiv t_n$
context $(\mho,\ \Sigma) \vdash \kappa\ \overline{\alpha{::}s_k}\ ::\ c$

The translation is guided by these certificates according to

Definition 23 (translation respecting certificates)

A translation transl from a system of code equations E_Θ to a program P respects the certificates in Definition 22 if each context judgement induced by P holds in Θ and E_P is exactly E_Θ.

We discuss the certificates briefly:

data What seems astonishing at first sight is that datatypes are characterised purely syntactically; the typical injectivity and distinctness properties of constructors known from *HOL* datatypes are absent. The reason is that in our HRS model an equations holds regardless of which logical interpretation the constants in its term patterns obey.

Here again the non-logical role of constructors Ξ comes to light: Ξ is not referred to in any certificate at all since the requirement to classify constants into constructors and non-constructors is a requirement of the target language, not the logic.

In the extreme case, two logically equal constants $f \equiv g$, each of a type *foo* could serve as datatype constructors for type *foo*; of course both would be distinguishable by their different *representation* in the target language (e.g. by means of target-language built-in equality), but this does not damage compatibility.

The absence of logical properties of datatypes gives some freedom in choosing constructors and allows for simple datatype abstraction, which we examine further in §4.1.

fun The logical equations match the semantics of the corresponding *fun* exactly, as to be expected. The sort constraints of the constant's type are not determined by the theory context Θ, where they are not represented anyway, but by the code equations.

class This merely echos the logical subclass structure and class parameter association.

inst The overloaded code equations match the semantics of the corresponding *inst* exactly. The context statement concerning the instance c_κ is formulated such that the sort arguments need not be the same as in the underlying theory but may be more special; this freedom allows an appropriate treatment of equality to be discussed in §3.3.4.

To show how the certificates for intermediate statements guarantee compatibility, we examine the relationship between $\Gamma_P = (\Upsilon_P, \Omega_P, \Xi, \mho_P, \Sigma_P, \omega_P)$ and $\Theta = (\Upsilon, \Omega, \mho, \Sigma, \omega, \ldots)$:

- Υ_P, $\mho_P$ and ω_P are projections of Υ, $\mho$ and ω.
- Ξ has no relation to the theory context Θ.
- Stripping the sorts from any type scheme of Ω_P yields the corresponding type scheme of Ω.
- Sort arguments in Σ_P are not more general than in Σ.

Therefore, Γ_P does not permit any typing judgements more general than Θ. In addition, certificates guarantee that E_P and E_Θ are the same. In conclusion, the following lemma characterises the translation sufficiently:

Lemma 24

Let transl be a translation which respects the certificates in Definition 22 and produces only wellformed programs P; then the resulting program P yields a HRS E_P which is *compatible* with E_Θ.

3.2.5 Well-sorted systems

Definitions 19 and 22 of well-sorted programs and certificates also characterise systems of code equations E_Θ which can actually be translated to a well-formed programs P; necessary properties are e.g. a suitable choice of Ξ such that constants are either constructors or non-constructors consistently across the whole system E_Θ. Further, given a constant f which is not a class parameter, all equations in E_Θ containing f as head must have the same sort arguments — this corresponds to the equations of a *fun* statement. Most of these properties are self-evident, with one exception which deserves closer attention: the role of Σ_P.

Sort constraints in $\Omega_P\ f$ are determined by the sort constraints of the corresponding code equations for constant f; sort arguments in $\Sigma_P\ c_\kappa$ are determined by Σ *and* corresponding code equations $g\ [\kappa\ \overline{\alpha{::}s}] \equiv t$ for all $g \in \omega\ c$. Thus Ω_P and Σ_P are mutually dependent: sort constraints in Ω_P may require a specialisation of Σ_P, which in turn can induce more special sort constraints in Ω_P etc. If the translation of a system of code equations E_Θ shall yield a well-formed program, it demands that sort arguments in Σ_P are chosen appropriately; this motivates the following definition:

Definition 25 (well-sorted systems)

A tuple $(\mho, \Sigma_P, \omega, E_\Theta)$ is called *well-sorted* if each equation in E_Θ is well-sorted wrt. $(\mho, \Sigma_P)$ and

- all code equations $f\ [\overline{\alpha{::}s_k}]\ \overline{p} \equiv t$ headed by a constant f which is not a class parameter have the same sort constraints $\overline{s}_k$ on their type arguments $\overline{\alpha}_k$,
- there is at most one code equation $f\ [\kappa\ \overline{\alpha{::}s_k}]\ \overline{p} \equiv t$ for each pair of a class parameter f and a type constructor κ,

- for each code equation $f\ [\kappa\ \overline{\alpha{::}s_k}]\ \overline{p} \equiv t$ headed by a class parameter f of class c (i.e. $f \in \omega\ c$) and instance c_κ with sort arguments $\Sigma_P\ c_\kappa = \overline{s'}_k$ holds $\forall\, 1 \leq i \leq k.\ (\mho,\, \Sigma_P) \vdash s'_i \subseteq s_i$.

In other words, a well-sorted system guarantees that sort constraints stemming from equations relevant for *fun* and *inst* statements do not break well-formedness (§3.2.3). Σ_P is then also the arity signature of the resulting program.

How Σ_P is determined practically will be sketched in §3.3.4 together with motivating examples.

3.2.6 Local pattern matching

A common idiom in target languages is local pattern matching:

$$\mathit{case}\ t\ \mathit{of}\ p_1 \Rightarrow t_1 \mid \cdots \mid p_n \Rightarrow t_n$$

where t, t_1, ..., t_n are terms and p_1, ..., p_n are patterns. Pattern matching occurs in further variants:

$\lambda(C\ y).\ t \mid (D\ y).\ u$ for $(\lambda x.\ \mathit{case}\ x\ \mathit{of}\ C\ y \Rightarrow t \mid D\ y \Rightarrow u)$
$\mathit{let}\ C\ y = x\ \mathit{in}\ t$ for $\mathit{case}\ x\ \mathit{of}\ C\ y \Rightarrow t$
...

It is advantageous to have an explicit representation of pattern matching in the intermediate language. First, an extension of HRSs with pattern matching:

Definition 26 (HRS with local pattern matching)

A HRS with *local pattern matching* extends the term language by explicit *case* expressions $\mathit{case}\ t\ \mathit{of}\ p_1 \Rightarrow t_1 \mid \cdots \mid p_n \Rightarrow t_n$ where t is a term of type τ_p, t_1, ..., t_n are terms of type τ_t and p_1, ..., p_n are patterns of type τ_p.

The semantics of such a *case* expression is given by a decomposition into an additional *fun* statement in the underlying program

$$\begin{array}{l} \mathit{fun}\ g :: \overline{\tau} \Rightarrow \tau_p \Rightarrow \tau_t\ \mathit{where} \\ \quad\ \ g\ \overline{x}\ p_1 = t_1 \\ \quad \mid \ldots \\ \quad \mid g\ \overline{x}\ p_n = t_n \end{array}$$

and a corresponding *case*-free expression $g\ \overline{x}\ t$, where g is a fresh constant symbol in the program and $\overline{x{::}\tau}$ are all free variables in $\overline{t}_n$.

Simultaneously we extend the term language of the intermediate language with explicit pattern matching.

But how do *case* expressions emerge from *Pure* terms? *HOL* creates the illusion of *case* expressions by providing for each datatype $\kappa\ \overline{\alpha}$ with constructors $C_1\ \mathit{of}\ \overline{\tau_1}$ $\cdots$ $C_n\ \mathit{of}\ \overline{\tau_n}$ a *case* combinator

$$\mathit{case}_\kappa :: (\overline{\tau_1} \Rightarrow \beta) \Rightarrow \cdots \Rightarrow (\overline{\tau_n} \Rightarrow \beta) \Rightarrow \kappa\ \overline{\alpha} \Rightarrow \beta$$

where the special syntax

$$\textsf{case}\ t\ \textsf{of}\ C_1\ \overline{x_1} \Rightarrow t_1 \mid \cdots \mid C_n\ \overline{x_n} \Rightarrow t_n$$

is represented by

$$case_\kappa\ (\lambda \overline{x_1}.\ t_1)\ \cdots\ (\lambda \overline{x_n}.\ t_n)\ t$$

Similarly we enrich the translation from *Pure* to the intermediate language by an explicit concept of case combinators:

Definition 27 (case combinators and case certificates)

A constant $case_\kappa$ is named *case combinator* if it is accompanied with a *case certificate* of the form

$$\bigwedge \overline{w_n}\ \overline{x_1}.\ case_\kappa\ w_1\ \cdots\ w_n\ (C_1\ \overline{x_1}) \equiv w_1\ \overline{x_1}$$
$$\ldots$$
$$\bigwedge \overline{w_n}\ \overline{x_n}.\ case_\kappa\ w_1\ \cdots\ w_n\ (C_n\ \overline{x_n}) \equiv w_n\ \overline{x_n}$$

Until now, we have identified the term language of *Pure* and the intermediate language; so the transformation of *Pure* terms to terms in the intermediate language has been identity. With the introduction of *case* expressions the transformation gets more involved: If $case_\kappa$ is a case combinator, then the *Pure* expression

$$case_\kappa\ w_1\ \cdots\ w_n\ t$$

is mapped to

$$\textsf{case}\ t\ \textsf{of}\ C_1\ \overline{x_1} \Rightarrow w_1\ \overline{x_1} \mid \cdots \mid C_n\ \overline{x_n} \Rightarrow w_n\ \overline{x_n}$$

with cases $w_k\ \overline{x_k}$ normalised modulo $\beta\eta$.

This modification is admissible: the mapping of *Pure* $case_\kappa$ expressions is revertible; thus it is still possible to identify terms in E_Θ and E_P. Further observe that case certificates for a case combinator $case_\kappa$ are also code equations for $case_\kappa$. These simulate the equational semantics of the corresponding *case* expression; in other words, the two systems

$$f\ \cdots \equiv \cdots\ \textsf{case}\ t\ \textsf{of}\ C_1\ \overline{x_1} \Rightarrow w_1\ \overline{x_1} \mid \cdots \mid C_n\ \overline{x_n} \Rightarrow w_n\ \overline{x_n}$$

and

$$f\ \cdots \equiv \cdots\ case_\kappa\ w_1\ \cdots\ w_n\ t$$
$$case_\kappa\ w_1\ \cdots\ w_n\ (C_1\ \overline{x_1}) \equiv w_1\ \overline{x_1}$$
$$\ldots$$
$$case_\kappa\ w_1\ \cdots\ w_n\ (C_n\ \overline{x_n}) \equiv w_n\ \overline{x_n}$$

have the same equational semantics. Thus *case* expressions can be seen as mere syntactic sugar which inlines a set of code equations representing the corresponding case certificate:

Lemma 28

Let E_Θ be a system of code equations with a subset $E_{case} \subseteq E_\Theta$ of case certificates.

Then the translation from $E_\Theta \setminus E_{case}$ to a program P with local pattern matching by means of case certificates E_{case} yields a HRS E_P which is compatible to E_Θ.

The modified translation can be slightly extended to accomplish further *HOL* syntax facilities: beyond simple patterns, *HOL* syntax also supports *nested* and *partial* patterns; nesting is achieved by a suitable combination of case combinators, partial patterns map each non-present pattern to the unspecified constant *undefined* in *HOL*. The code generator takes this into account when translating case expressions by leaving clauses with *undefined* out, thus yielding a pattern match failure if the corresponding pattern would occur, which is legitimate since we only demand partial correctness. Indeed, arbitrary constants can be treated like *undefined*.

Another syntactic *HOL* device is a simple *let* for local bindings without polymorphism. This can be seen as a degenerate case with the *case certificate*

$$\bigwedge w\ x.\ \mathit{Let}\ x\ w \equiv w\ x$$

All these minor extensions are admissible.

3.2.7 Dictionary construction

The underlying idea. For target languages with type classes like *Haskell*, the serialisation of *class* and *inst* statements is straightforward. Otherwise, type classes and overloading are eliminated by means of dictionary construction. The idea behind is simple: eliminate overloading by abstraction. E.g. if in a function

$\mathsf{fun}\ \mathit{greater} :: \forall\,\alpha.\ \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}\ \mathsf{where}$
$\quad \mathit{greater}\ [\alpha]\ x\ y = \mathit{not}\ (\mathit{less\_eq}\ x\ y)$

less_eq refers to a class parameter, this overloading can be eliminated by abstracting *greater* over *less_eq*:

$\mathsf{fun}\ \mathit{greater} :: \forall\,\alpha.\ (\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}\ \mathsf{where}$
$\quad \mathit{greater}\ [\alpha]\ \mathit{less\_eq}\ x\ y = \mathit{not}\ (\mathit{less\_eq}\ x\ y)$

Each call to *greater* is then augmented by passing an appropriate *less_eq*, which is either also an abstracted additional parameter or an appropriate instance of $\mathit{less\_eq}_\kappa$ on a particular type, as follows:

$\mathsf{fun}\ \mathit{between} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}\ \mathsf{where}$
$\quad \mathit{between}\ m\ n\ q = \mathit{less\_eq}_{nat}\ m\ n \wedge \mathit{greater}\ \mathit{less\_eq}_{nat}\ n\ q$

Intuitively, the choice of the overloaded definition to take in a particular place is propagated through a system of equations until typing allows for a decision.

Type classes describe a discipline of how to domesticate this idea. A *class* statement for class c defines a record-like data type $\delta_c\ \alpha$ (dictionary type) which contains fields for all class parameters of c. The class parameters themselves are defined as *fun*s which project the appropriate field from a value of type $\delta_c\ \alpha$. An *inst* statement for an instance c_κ provides a concrete record-like value of type $\delta_c\ (\kappa\ \overline{\alpha{::}s_k})$ containing the concrete instances for class operations of class c on type constructor κ, which is given as dictionary term c_κ.

Superclasses are accomplished by extending dictionary types with additional fields for superclass dictionaries and corresponding projections $\pi_{c' \to c}$. By construction, different projection paths $\pi_{c_1 \to c} \circ \pi_{c' \to c_1}$ and $\pi_{c_2 \to c} \circ \pi_{c' \to c_2}$ in a diamond diagram yield the same result, making the exact choice of a projection path irrelevant.

Dictionary construction on intermediate programs. Definition 29 gives the rules for dictionary construction on intermediate programs in detail.

The transformation of a program P into a program P_Δ where type classes and overloading are eliminated by means of dictionary construction is shown in tabular form; beside the transformation of *class* and *inst* statements sketched above, also *fun*s must be translated to abstract over and insert dictionaries appropriately; for this purpose two notations are used:

- $\{\!|\forall\, \overline{\alpha::s}.\ \tau|\!\}$ adds dictionary type parameters to a type scheme $\forall\, \overline{\alpha::s}.\ \tau$.
- $\{\!|t|\!\}$ inserts dictionaries into a term t; this is almost identity, except for constants $f\ [\tau_1,\ \ldots,\ \tau_n]$.

Dictionary construction proper for a constant $f\ [\tau_1,\ \ldots,\ \tau_n]$ is accomplished by a constructive interpretation of the underlying well-sortedness judgements $\tau_1 :: s_1$ $\ldots$ $\tau_n :: s_n$, where $\{\!|\tau :: s|\!\}$ maps a well-sortedness judgement to the corresponding dictionaries. This inserts occurrences of concrete dictionary terms c_κ as well as dictionary variables; by convention dictionary variables are named after the corresponding type variable with an additional index α_j, or α_Δ if the index does not matter.

Type expressions τ themselves are invariant under dictionary construction — sort constraints on type variables are only an annotation convention and no syntactic part of the type itself.

Definition 29 (dictionary construction)

relative to a fixed context $(\Omega_P,\ \mho_P,\ \Sigma_P)$:

for well-sortedness judgements

$$\frac{\Sigma_P\ c_\kappa = \overline{s}_n}{\{\!|\kappa\ \tau_1\ \cdots\ \tau_n :: c|\!\} = c_\kappa\ \{\!|\tau_1 :: s_1|\!\}\ \cdots\ \{\!|\tau_n :: s_n|\!\}}\ \{\!|\text{constr}|\!\}$$

$$\frac{}{\{\!|(\alpha::(c_1 \cap \cdots \cap c_j \cap \cdots \cap c_n)) :: c_j|\!\} = \alpha_j}\ \{\!|\text{var}|\!\}$$

$$\frac{c \in \mho_P\ c'}{\{\!|\tau :: c|\!\} = \pi_{c' \to c}\ \{\!|\tau :: c'|\!\}}\ \{\!|\text{classrel}|\!\}$$

$$\frac{}{\{\!|\tau :: c_1 \cap \cdots \cap c_n|\!\} = \{\!|\tau :: c_1|\!\}\ \cdots\ \{\!|\tau :: c_n|\!\}}\ \{\!|\text{sort}|\!\}$$

for type schemes

$$\begin{gathered}\{\!|\forall\, \alpha_1 :: (c_{1;1} \cap \cdots \cap c_{1;k_1})\ \cdots\ \alpha_n :: (c_{n;1} \cap \cdots \cap c_{n;k_n}).\ \tau|\!\} = \\ \forall\, \alpha_1 \cdots \alpha_n.\ \delta_{c_{1;1}}\ \alpha_1 \Rightarrow \cdots \Rightarrow \delta_{c_{1;k_1}}\ \alpha_1 \\ \Rightarrow \cdots \Rightarrow \delta_{c_{n;1}}\ \alpha_n \Rightarrow \cdots \Rightarrow \delta_{c_{n;k_n}}\ \alpha_n \Rightarrow \tau\end{gathered}$$

for terms

$$\frac{\Omega_P\ f = \forall\, \alpha_1{::}s_1\ \cdots\ \alpha_n{::}s_n.\ \tau}{\{\!\!\{f\ [\tau_1,\ \ldots,\ \tau_n]\}\!\!\} = f\ \{\!\!\{\tau_1\ ::\ s_1\}\!\!\}\ \cdots\ \{\!\!\{\tau_n\ ::\ s_n\}\!\!\}}\ \{\!\!\{\mathrm{const}\}\!\!\}$$

$$\frac{}{\{\!\!\{x{::}\tau\}\!\!\} = x{::}\tau}$$

$$\frac{}{\{\!\!\{\lambda x{::}\tau.\ t\}\!\!\} = \lambda x{::}\tau.\ \{\!\!\{t\}\!\!\}}$$

$$\frac{}{\{\!\!\{t_1\ t_2\}\!\!\} = \{\!\!\{t_1\}\!\!\}\ \{\!\!\{t_2\}\!\!\}}$$

for programs

statement	transformed statement(s)
data $\kappa\ \overline{\alpha}_k =$ f_1 *of* $\overline{\tau_1}$ $\mid \cdots \mid$ f_n *of* $\overline{\tau_n}$	*data* $\kappa\ \overline{\alpha}_k =$ f_1 *of* $\overline{\tau_1}$ $\mid \cdots \mid$ f_n *of* $\overline{\tau_n}$
fun $f\ ::\ \forall\,\overline{\alpha{::}s}_k.\ \tau$ *where* $f\ [\overline{\alpha{::}s}_k]\ \overline{p_1} = t_1$ $\mid \ldots$ $\mid f\ [\overline{\alpha{::}s}_k]\ \overline{p_n} = t_n$	*fun* $f\ ::\ \{\!\!\{\forall\,\overline{\alpha{::}s}_k.\ \tau\}\!\!\}$ *where* $\{\!\!\{f\ [\overline{\alpha{::}s}_k]\ \overline{p_1}\}\!\!\} = \{\!\!\{t_1\}\!\!\}$ $\mid \ldots$ $\mid \{\!\!\{f\ [\overline{\alpha{::}s}_k]\ \overline{p_n}\}\!\!\} = \{\!\!\{t_n\}\!\!\}$
class $c \subseteq c_1 \cap \cdots \cap c_m$ *where* $g_1\ ::\ \forall\,\alpha.\ \tau_1,\ \ldots,$ $g_n\ ::\ \forall\,\alpha.\ \tau_n$	*data* $\delta_c\ \alpha =$ Δ_c *of* $(\delta_{c_1}\ \alpha)\ \cdots\ (\delta_{c_m}\ \alpha)\ \tau_1\ \cdots\ \tau_n$ *fun* $\pi_{c\to c_1}\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \delta_{c_1}\ \alpha$ *where* $\pi_{c\to c_1}\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{c_1}$ $\ldots$ *fun* $\pi_{c\to c_m}\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \delta_{c_m}\ \alpha$ *where* $\pi_{c\to c_m}\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{c_m}$ *fun* $g_1\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \tau_1$ *where* $g_1\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{g_1}$ $\ldots$ *fun* $g_n\ ::\ \forall\,\alpha.\ \delta_c\ \alpha \Rightarrow \tau_n$ *where* $g_n\ (\Delta_c\ x_{c_1}\ \cdots\ x_{c_m}\ x_{g_1}\ \cdots\ x_{g_n}) = x_{g_n}$
inst $\kappa\ \overline{\alpha{::}s}_k\ ::\ c$ *where* $g_1\ [\kappa\ \overline{\alpha{::}s}_k] = t_1,\ \ldots,$ $g_n\ [\kappa\ \overline{\alpha{::}s}_k] = t_n$	*fun* $c_\kappa\ ::\ \{\!\!\{\forall\,\overline{\alpha{::}s}_k.\ \delta_c\ (\kappa\ \overline{\alpha{::}s}_k)\}\!\!\}$ *where* $\{\!\!\{\kappa\ \overline{\alpha{::}s}_k\ ::\ c\}\!\!\} = \Delta_c$ $\{\!\!\{\kappa\ \overline{\alpha{::}s}_k\ ::\ c_1\}\!\!\}\ \cdots\ \{\!\!\{\kappa\ \overline{\alpha{::}s}_k\ ::\ c_n\}\!\!\}$ $\{\!\!\{t_1\}\!\!\}\ \cdots\ \{\!\!\{t_n\}\!\!\}$ with $\mho_P\ c = \{c_1,\ \ldots,\ c_n\}$

The transformation of *inst* statements also reveals the role of coregularity (cf. §2.3.1) for dictionary construction: syntactically, the presence of the instance $\kappa\ \overline{\alpha{::}s}_k\ ::\ c$ also demands the presence of all instances $\kappa\ \overline{\alpha{::}s}_k\ ::\ c_i$ for all c_i in $\mho_P\ c$.

Next we discuss why correctness is maintained under dictionary construction.

Well-formedness. Following Definition 19 we have to set *prerequisites* and *results* of P and P_Δ in relation to each other. Results induce typing contexts $\Gamma_P = (\Upsilon_P, \Omega_P, \Xi, \mho_P, \Sigma_P, \omega_P)$ and $\Gamma_\Delta = (\Upsilon_\Delta, \Omega_\Delta, \Xi_\Delta, \mho_\Delta, \Sigma_\Delta, \omega_\Delta)$, which are related as follows:

- By construction $\mho_\Delta$, Σ_Δ and ω_Δ are empty.
- $\Xi_\Delta = \Xi \cup \{\Delta_c.\ c \in \mathit{dom}\ \mho_P\}$ and $\Upsilon_\Delta = \Upsilon_P \cup \{(\delta_c, *).\ c \in \mathit{dom}\ \mho_P\}$.
- $\Omega_\Delta = \{(f, \langle \{\!\!\{\forall \overline{\alpha{::}s}_k.\ \tau\}\!\!\}\rangle).\ \Omega_P\ f = \langle \forall \overline{\alpha{::}s}_k.\ \tau\rangle \wedge \nexists c.\ f \in \omega_P\ c\}$
$\cup \{(\pi_{c \to c'}, \langle \forall \alpha.\ \delta_c\ \alpha \Rightarrow \delta_{c'}\ \alpha\rangle).\ c' \in \mho_P\ c\}$
$\cup \{(g, \langle \forall \alpha.\ \delta_c\ \alpha \Rightarrow \tau\rangle).\ g \in \omega_P\ c \wedge \Omega_P\ g = \langle \forall \alpha.\ \tau\rangle\}$
$\cup \{(c_\kappa, \langle \{\!\!\{\forall \overline{\alpha{::}s}_k.\ \delta_c\ (\kappa\ \overline{\alpha{::}s}_k)\}\!\!\}\rangle).\ \Sigma_P\ c_\kappa = \overline{s}_k\}$

From this the prerequisites of *data* statements in P_Δ follow easily; concerning *fun*s, we first prove:

Lemma 30 (types of dictionary values)

$$\Gamma_\Delta \vdash \{\!\!\{\tau :: c_1 \cap \ldots \cap c_n\}\!\!\} :: (\delta_{c_1}\ \tau) \cdots (\delta_{c_n}\ \tau)$$

The proof follows by induction over the dictionary construction rules for well-sortedness judgements. Next follows:

Lemma 31 (type preservation under dictionary construction)

$$\frac{\Gamma_P \vdash t :: \tau}{\Gamma_\Delta \vdash \{\!\!\{t\}\!\!\} :: \tau}$$

Proof by induction over the dictionary construction rules for terms: except $\{\!\!\{\text{const}\}\!\!\}$ all rules in the translations of terms are trivial to prove; the additional arguments added to a particular constant f by $\{\!\!\{\text{const}\}\!\!\}$ have exactly the same type as the arguments added to the type scheme of the *fun* statement which introduces *f*.

From this follows well-typedness of equations in *fun* statements. What remains to be shown is that the transformed equations meet the syntactic requirements of code equations (cf. Definition 15):

- Equations are headed by function symbols: f in equations stemming from *fun* statements in P, c_κ in equations stemming from *inst*.
- On the left hand side, distinct fresh dictionary variables α_Δ are added, which are trivially patterns and left-linear.
- By hypothesis, every free type variable on the right hand side occurs on the left hand side; thus all dictionary variables added on the right hand side also occur on the left hand side.
- Since constructors' type schemes have empty sort constraints, constructors do not gain dictionary parameters; thus patterns are left unchanged by dictionary construction.

Thus P_Δ is well-formed.

Towards compatibility. Next we turn our attention to the rewrite systems induced by P and P_Δ, named E_P and E_{P_Δ}. The major difference between both systems is that E_{P_Δ} contains additional constants Δ_c which complicate the analysis. To cope with these we first state a lemma which allows to view reduction sequences $E_{P_\Delta} \Vdash t \longrightarrow \cdots \longrightarrow u$ in a normal form where reduction steps involving Δ_c only occur at certain places.

Lemma 32 (normalising of tuple projections)

Let Δ, $\pi_1, \ldots, \pi_k$ be dedicated constants in a HRS whose set of equations decomposes into three partitions:

- $E_\Delta = \{\langle f\ \overline{x} \equiv \Delta\ \overline{t}\rangle\}$ where only $\overline{x}$ occur as free variables in $\overline{t}$;
- $E_\pi = \{\langle \pi_1\ (\Delta\ \overline{x}_k) \equiv x_1\rangle, \ldots, \langle \pi_k\ (\Delta\ \overline{x}_k) \equiv x_k\rangle\}$;
- E whose equations are left-linear and do not contain Δ.

Then each reduction sequence

$$E \uplus E_\Delta \uplus E_\pi \Vdash t \longrightarrow \cdots \longrightarrow u$$

where t does not contain Δ has a normal form where each reduction step using E_Δ occurs in only two kinds of positions:

- directly in front of a corresponding E_π step;
- in a (possibly empty) trailing reduction sequence consisting only of E_Δ steps and consequent rewrites below the corresponding Δ constants (subsequently referred to as "tail").

Metaphorically speaking, we have one rule E_Δ introducing tuples, corresponding tuple eliminations E_π and generic rules E that do not influence the structure of tuples. Then w.l.o.g. E_Δ steps occur lazily.

The proof works by shifting each E_Δ step (denoted by $\xrightarrow{\Delta}$) as far to the right as possible. A step following a $\xrightarrow{\Delta}$ falls into one of the following categories:

- $\xrightarrow{\pi}$: a corresponding E_π step;
- $\xrightarrow{\parallel}$: an orthogonal non-$\xrightarrow{\Delta}$ step;
- $\xrightarrow{a}$: a step *above* the Δ introduced by $\xrightarrow{\Delta}$, but no $\xrightarrow{\Delta}$ itself;
- $\xrightarrow{b}$: a step *below* the Δ introduced by $\xrightarrow{\Delta}$, but no $\xrightarrow{\Delta}$ itself.

A $\xrightarrow{\Delta}$ step directly followed by a corresponding $\xrightarrow{\pi}$ is already normalised. To account for this by convention we treat such pairs as a monolithic singleton step $\xrightarrow{\Delta\pi}$. These steps are not treated specially: in the classification scheme above a $\xrightarrow{\Delta\pi}$ can be classified e.g. as a $\xrightarrow{\parallel}$ relative to some $\xrightarrow{\Delta}$ step.

A critical observation is that in the reduction sequence $\xrightarrow{\Delta} \cdot \xrightarrow{b}$ the order cannot be swapped. Thus we cannot move a singleton $\xrightarrow{\Delta}$ rightward but have to consider a compound $\xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{*}$ instead. Their occurrences are shift right as follows:

1. $t \xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{k} \cdot \xrightarrow{\pi} u \ \rightsquigarrow\ t \xrightarrow{\Delta\pi} \cdot \xrightarrow{b}{}^{k-l}\ u$

2. $t \xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{k} \cdot \xrightarrow{\parallel} u \ \rightsquigarrow\ t \xrightarrow{\parallel} \cdot \xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{k}\ u$

3. $t \xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{k} \cdot \xrightarrow{a} u \ \rightsquigarrow\ t \xrightarrow{a} \cdot \left(\xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{k} \right)^{*}\ u$

Each of these shifts is valid since the initial and resulting term remain the same:

- In case 1, all steps in $\xrightarrow{b}{}^{k}$ which do not affect the part of the redex $(\Delta \ \ldots)$ projected by $\xrightarrow{\pi}$ can be stripped; the remaining steps $\xrightarrow{b}{}^{k-l}$ can take place after the projection step $\xrightarrow{\pi}$.
- In case 2, the swapped steps do not interfere at all.
- In case 3, the redex $(\Delta \ \ldots)$ may be dropped or replicated; the compound steps $\xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{k}$ must be iterated accordingly.

If applied iteratively, these shifts produce the desired normal form by pushing each $\xrightarrow{\Delta}$

- either to its corresponding $\xrightarrow{\pi}$ step, resulting in a normalised $\xrightarrow{\Delta\pi}$ step
- or into a trailing series of compounds $\xrightarrow{\Delta} \cdot \xrightarrow{b}{}^{*}$, which is the "tail" from the lemma proposition.

The procedure terminates according to the following argument: any reduction sequence matches the pattern

$$\cdot \left(\left(\xrightarrow{\Delta} \mid \xrightarrow{\Delta\pi} \right) \cdot \xrightarrow{b}{}^{*} \right)^{r_1} \cdot \longrightarrow \cdot \left(\left(\xrightarrow{\Delta} \mid \xrightarrow{\Delta\pi} \right) \cdot \xrightarrow{b}{}^{*} \right)^{r_2} \cdot$$
$$\longrightarrow \cdot \cdots \cdot \longrightarrow \cdot \left(\left(\xrightarrow{\Delta} \mid \xrightarrow{\Delta\pi} \right) \cdot \xrightarrow{b}{}^{*} \right)^{r_n} \cdot$$

where $\longrightarrow$ denotes an arbitrary step not being of class $\xrightarrow{b}$ wrt. to the precedent $\xrightarrow{\Delta}$ or $\xrightarrow{\Delta\pi}$ step. This pattern induces a tuple $(r_1, r_2, \ldots, r_n)$; the corresponding lexicographic order can be employed as termination measure:

- In case 1: $(\overline{v}, r, s, \overline{w}) \ \rightsquigarrow\ (\overline{v}, r + s, \overline{w})$
- In case 2: $(\overline{v}, r, s, \overline{w}) \ \rightsquigarrow\ (\overline{v}, r - 1, s + 1, \overline{w})$
- In case 3: $(\overline{v}, r, s, \overline{w}) \ \rightsquigarrow\ (\overline{v}, r - 1, s + *, \overline{w})$

Compatibility. Going back to Definition 12, the property that two HRSs are compatible refers to an implicit morphism which translates terms in one HRS into the other, and back. So far, this morphism has always been identity or a simple one-to-one translation in the case of *case* expressions (cf. §3.2.6). With dictionary construction, the matter gets more complicated and deserves our special attention. Unsurprisingly, the morphism from E_P and E_{P_Δ} is the (injective) dictionary construction function $\{\!|\cdot|\!\}$. The morphism back from E_{P_Δ} to E_P strips all terms occurring as dictionary arguments; we denote this (surjective) function by $|\!\}\cdot\{\!|$. These morphisms are *no* bijection: it holds $|\!\}(\{\!|t|\!\})\{\!| = t$ but *not* $\{\!|(|\!\}w\{\!|)|\!\} = w$. The reason is that E_{P_Δ} has richer term expressions, mainly due to superclass projections $\pi_{c \to c'}$. Having set out these prerequisites, the propositions to prove in the first place are

1. $E_P \Vdash t \longrightarrow^* u$ implies $E_{P_\Delta} \Vdash \{\!|t|\!\} \longrightarrow^* \{\!|u|\!\}$
2. $E_{P_\Delta} \Vdash w \longrightarrow^* u$ implies $E_P \Vdash |\!\}w\{\!| \longrightarrow^* |\!\}u\{\!|$

For case 2, we can assume w.l.o.g. that w is the image of a term t under dictionary construction; thus $w = \{\!|t|\!\}$ and $|\!\}w\{\!| = |\!\}(\{\!|t|\!\})\{\!| = t$, which simplifies the proposition to

2. $E_{P_\Delta} \Vdash \{\!|t|\!\} \longrightarrow^* u$ implies $E_P \Vdash t \longrightarrow^* |\!\}u\{\!|$

Intuitively, dictionary construction separates the *decision* which equation to take for a particular overloaded constant g from its actual *application*. Lemma 32 allows us to rejoin both: We apply it for each class c occurring in P; the tuple constructor is Δ_c, the corresponding introduction equation is $\langle c_\kappa\ \overline{\alpha_\Delta} = \Delta_c\ \ldots\rangle$ the projections are equations with overloaded constants $\langle g\ (\Delta_c\ \ldots\ x\ \ldots) = x\rangle$ and superclass projections $\langle \pi_{c \to c'}\ (\Delta_c\ \ldots\ x\ \ldots) = x\rangle$. Thus we can examine reduction sequences in E_{P_Δ} in normal form such that each application of an equation $\langle c_\kappa\ \overline{\alpha_\Delta} = \Delta_c\ \ldots\rangle$

- either is immediately followed by an application of a corresponding equation $\langle g\ (\Delta_c\ \ldots\ x\ \ldots) = x\rangle$ or $\langle \pi_{c \to c'}\ (\Delta_c\ \ldots\ x\ \ldots) = x\rangle$, forming compounds,
- or occurs in the "tail" of the reduction sequence.

We can ignore the "tail" entirely: given such a "tail" $u \longrightarrow^* u'$, all reduction steps in $\longrightarrow^*$ occur at redex positions which are stripped away by $|\!\}\cdot\{\!|$, so $|\!\}u\{\!| = |\!\}u'\{\!|$.

Further we can now merge the c_κ / g / $\pi_{c \to c'}$ equations in E_{P_Δ}, resulting in a system $\{\!|E_P|\!\}$ which in structure is quite similar to E_P:

equations in E_P	equations in $\{\!|E_P|\!\}$
$\langle f\ [\overline{\alpha{::}s}]\ \overline{p} \equiv t\rangle$	$\langle\{\!|f\ [\overline{\alpha{::}s}]\ \overline{p}|\!\} \equiv \{\!|t|\!\}\rangle \rightleftharpoons \langle f\ \overline{\alpha_\Delta}\ \overline{p} \equiv \{\!|t|\!\}\rangle$
$\langle g\ [\kappa\ \overline{\beta{::}s}] \equiv u\rangle$	$\langle\{\!|g\ [\kappa\ \overline{\beta{::}s}]|\!\} \equiv \{\!|u|\!\}\rangle \rightleftharpoons \langle g\ (c_\kappa\ \overline{\beta_\Delta}) \equiv \{\!|u|\!\}\rangle$
	$\langle \pi_{c \to c'}\ (c_\kappa\ \overline{\beta_\Delta}) \equiv c'_\kappa\ \ldots\rangle$

For each equation $\langle lhs = rhs\rangle$ in E_P there is a corresponding equation $\langle\{\!|lhs|\!\} = \{\!|rhs|\!\}\rangle$ in $\{\!|E_P|\!\}$. Let us denote these two classes of equations as $\mathcal{E}$ and $\{\!|\mathcal{E}|\!\}$, respectively. As a distinguished property $\{\!|E_P|\!\}$ contains equations of class $\{\!|\pi|\!\}$ that normalise superclass projections $\pi_{c \to c'}$. Dictionary construction on the left hand side of an equation for a non-overloaded constant f produces additional left-linear arguments $\overline{\alpha_\Delta}$. In case of an equation for an overloaded constant g, it produces as argument a constant c_κ applied to additional left-linear arguments $\overline{\beta_\Delta}$; here c_κ serves as syntactic discriminator which overloaded equation for g to use exactly: different instances $\kappa\ \overline{\tau}$ and $\kappa'\ \overline{\tau}'$ produce different discriminators c_κ and $c_{\kappa'}$. Note further that

- if a $\mathcal{E}$ equation can be applied to a particular redex in a term t, the corresponding $\{\!\!\{\mathcal{E}\}\!\!\}$ equation can be applied to the corresponding redex in term $\{\!\!\{t\}\!\!\}$
- and correspondingly, if a $\{\!\!\{\mathcal{E}\}\!\!\}$ equation can be applied to a particular redex in a term t, the corresponding $\mathcal{E}$ equation can be applied to the corresponding redex in term $\}\!\!\}t\{\!\!\{$.

This holds due to the structure of left hand sides in $\{\!\!\{\mathcal{E}\}\!\!\}$ equations: dictionary construction only inserts new free variables but does not restrict or widen the patterns themselves.

Applying Lemma 32, the propositions to prove are

1. $E_P \Vdash t \longrightarrow^* u$ implies $\{\!\!\{E_P\}\!\!\} \Vdash \{\!\!\{t\}\!\!\} \longrightarrow^* \{\!\!\{u\}\!\!\}$
2. $\{\!\!\{E_P\}\!\!\} \Vdash \{\!\!\{t\}\!\!\} \longrightarrow^* u$ implies $E_P \Vdash t \longrightarrow^* \}\!\!\}u\{\!\!\{$

Proof of (1). The reduction sequence $E_P \Vdash t \longrightarrow^* u$ consists of a series of $\mathcal{E}$ steps; each of these we can simulate by a corresponding $\{\!\!\{\mathcal{E}\}\!\!\}$ step and a subsequent normalising of superclass projections, according to the following picture:

$$\begin{tikzcd}
E_P \Vdash & t \arrow[rr, "\mathcal{E}"] \arrow[d, "\{\!\!\{\cdot\}\!\!\}"] & & w \arrow[d, "\{\!\!\{\cdot\}\!\!\}"] \\
\{\!\!\{E_P\}\!\!\} \Vdash & \{\!\!\{t\}\!\!\} \arrow[r, dashed, "\{\!\!\{\mathcal{E}\}\!\!\}"] & w' \arrow[r, dashed, "\{\!\!\{\pi\}\!\!\}^*"] & \{\!\!\{w\}\!\!\}
\end{tikzcd}$$

Proof of (2). In the opposite direction, the situation is more delicate since the mere existence of dictionary values $c_\kappa\ \ldots$ breaks the direct correspondence: Reductions in a term t which take place under a c_κ have no effect in $\}\!\!\}t\{\!\!\{$! The solution is to use an appropriate amortisation when constructing the reduction sequence in E_P; whenever a step in $\{\!\!\{E_P\}\!\!\}$ is to be simulated in E_P, the following rules apply:

- If the step occurs below any c_κ, this reduction is "memorised" at the next c_κ directly above in the term structure.
- Otherwise, the corresponding step in E_P is applied directly.
- If the step strips a c_κ by projecting it, all memorised steps at this c_κ which remain valid after projection are simulated right after; this can happen recursively, e.g. the stripping of a c_κ underneath a $c_{\kappa'}$ can lead to new steps memorised at $c_{\kappa'}$.
- If the steps drops a c_κ by other means, the memorised steps are ignored — they have no relevance for construction of the reduction sequence in E_P at all.

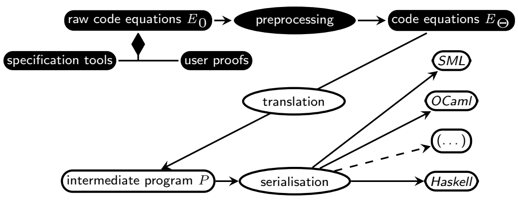
Both proof directions together show that E_P is compatible with $\{\!\!\{E_P\}\!\!\}$. By means of Lemma 32, E_P then is compatible with E_{P_Δ}. With transitivity follows that E_Θ is compatible with E_{P_Δ}.

Finally with well-formedness of P_Δ follows:

Lemma 33

Let P a well-formed program; let P_Δ be the program constructed by applying dictionary construction to P. Then P_Δ is well-formed and E_{P_Δ} is compatible with E_P.

3.3 Code generation in practice using *Isabelle/HOL*



With the properties discussed so far, code generation is explained thoroughly. In the following, we illustrate how this manifests in practice — from the user perspective, *HOL* offers everything needed for writing down reasonable executable specifications without much need to think what happens behind the scene. The key technique to achieve this is the provision of a suitable set of code equations E_Θ. Here the *preprocessor* component plays a central role.

The requirements put on the translation in §3.2.3 and §3.2.4 give three degrees of freedom:

- Code equations E_Θ can be chosen freely from the theorems of the theory Θ. Internally, the code generator does bookkeeping on a pool of *raw code equations* E_0 selected by the user implicitly or explicitly, from which a subset is chosen and transformed by the preprocessor to derive E_Θ. Case certificates $E_{case} \subseteq E_0$ are a special instance of code equations.
- Constructor constants Ξ can be chosen freely, as long as they conform to the syntactic restrictions imposed by Definition 22 and the patterns in code equations and case patterns contain only constructor constants. In theory it is possible to infer Ξ from the code equations, but it is more robust from the user perspective to be explicit here.
- Sort constraints in Ω_P and Σ_P are not fixed; they are implicitly relative to Σ and E_Θ.

3.3.1 Code generator default setup

As noted in §2.2.2, **datatype** and **definition**/**primrec**/**fun** form the core of a functional programming language inside *HOL*. To accommodate for this, these statements are instrumentalised the following way:

- A **datatype** registers the given constructors in Ξ and registers the corresponding *case* combinator certificate in E_{case}.
- **definition**/**primrec**/**fun** registers the resulting equations in E_0, internally lifting every *HOL* equation $(=)$ to a *Pure* equation $(\equiv)$.

This means that "naive" code generation can proceed without further ado. For example, here a simple "implementation" of amortised queues [14]:

datatype α *queue* $=$ *Queue* $(\alpha\ list)$ $(\alpha\ list)$

definition *empty* $::$ α *queue* **where**
$empty = Queue\ [\!]\ [\!]$

primrec *enqueue* $:: \alpha \Rightarrow \alpha\ queue \Rightarrow \alpha\ queue$ **where**
$enqueue\ x\ (Queue\ xs\ ys) = Queue\ (x : xs)\ ys$

fun *dequeue* $:: \alpha\ queue \Rightarrow \alpha\ option \times \alpha\ queue$ **where**
$dequeue\ (Queue\ [\!]\ [\!]) = (None,\ Queue\ [\!]\ [\!])$
$|\ dequeue\ (Queue\ xs\ (y : ys)) = (Some\ y,\ Queue\ xs\ ys)$
$|\ dequeue\ (Queue\ xs\ [\!]) =$
$(case\ rev\ xs\ of\ y : ys \Rightarrow (Some\ y,\ Queue\ [\!]\ ys))$

Then the corresponding *Haskell* code looks as follows:

```
module Example where {


fold :: forall a b. (a -> b -> b) -> [a] -> b -> b;
fold f [] s = s;
fold f (x : xs) s = fold f xs (f x s);

rev :: forall a. [a] -> [a];
rev xs = fold (\ a b -> a : b) xs [];

list_case :: forall t a. t -> (a -> [a] -> t) -> [a] -> t;
list_case f1 f2 (a : list) = f2 a list;
list_case f1 f2 [] = f1;

data Queue a = Queue [a] [a];

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

dequeue :: forall a. Queue a -> (Maybe a, Queue a);
dequeue (Queue [] []) = (Nothing, Queue [] []);
dequeue (Queue xs (y : ys)) = (Just y, Queue xs ys);
dequeue (Queue (v : va) []) =
  let {
    (y : ys) = rev (v : va);
  } in (Just y, Queue [] ys);

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

}
```

Some observations and remarks:

- Code generation performs a dependency analysis to generate all statements necessary for a consistent program: if a constant f occurs on the right hand side of a code equations, all code equations headed by f are included in E_Θ.
- The translations for type α *queue* and functions *empty* and *enqueue* bear no surprise.
- The generated code for lists and option values uses the *Haskell* built-in lists and `Maybe` values due to a default serialiser setup whose discussion we postpone for a moment (see §3.4.1).
- In the last clause of the *dequeue* function, observe that compared to the original **fun** specification, the first list argument for the *Queue* constructor in the left-hand side pattern is split into a (:) expression. This is *not* due to the code generator but due to **fun** which disambiguates overlapping patterns by splitting them sequentially [33]:

  $$\begin{array}{l} \mathit{dequeue}\ (\mathit{Queue}\ [\![]\!]\ [\![]\!]) = (\mathit{None},\ \mathit{Queue}\ [\![]\!]\ [\![]\!]) \\ \mathit{dequeue}\ (\mathit{Queue}\ \mathit{xs}\ (y : \mathit{ys})) = (\mathit{Some}\ y,\ \mathit{Queue}\ \mathit{xs}\ \mathit{ys}) \\ \mathit{dequeue}\ (\mathit{Queue}\ (v : \mathit{va})\ [\![]\!]) = \\ (\mathit{case}\ \mathit{rev}\ (v : \mathit{va})\ \mathit{of}\ y : \mathit{ys} \Rightarrow (\mathit{Some}\ y,\ \mathit{Queue}\ [\![]\!]\ \mathit{ys})) \end{array}$$

- The "partial" *case* (whose remaining clauses internally are *undefined*, cf. §3.2.6) is mapped to a `case` with only one clause, which by convention is printed as a `let`.

It is emphasised that, though in this example the *Isar* source text and the resulting *Haskell* text appears quite similar, what happens is *not* a translation of the source text; instead, the source text produces a theory Θ from which the resulting program text is ultimately generated. Θ can also be enriched manually to produce different code — e.g. we could provide an alternative *fun* for *dequeue* proving the following equations explicitly:

lemma [*code*]:
$$\begin{array}{l} \mathit{dequeue}\ (\mathit{Queue}\ \mathit{xs}\ [\![]\!]) = \\ \quad (\mathit{if}\ \mathit{xs} = [\![]\!]\ \mathit{then}\ (\mathit{None},\ \mathit{Queue}\ [\![]\!]\ [\![]\!])\ \mathit{else}\ \mathit{dequeue}\ (\mathit{Queue}\ [\![]\!]\ (\mathit{rev}\ \mathit{xs}))) \\ \mathit{dequeue}\ (\mathit{Queue}\ \mathit{xs}\ (y : \mathit{ys})) = (\mathit{Some}\ y,\ \mathit{Queue}\ \mathit{xs}\ \mathit{ys}) \end{array}$$
by (*cases xs*, *simp_all*) (*cases rev xs*, *simp_all*)

The annotation *code* is an *Isar attribute* which states that the given theorems should be considered as code equations for a *fun* statement — the corresponding constant is determined syntactically. The resulting code:

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

Note that the equality test $xs = [\![]\!]$ has been replaced by the predicate `null xs`. This is due to a default setup in the *preprocessor* to be discussed further in §3.3.3.

For examples for user-specified constructors for datatypes see §4.1, §4.2.2 and §4.2.3.

3.3.2 *class* and *instantiation*

Concerning type classes and code generation, let us again examine an example from abstract algebra (cf. §2.3.3):

class *semigroup* =
  fixes *mult* :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$ (**infixl** $\otimes$ 70)
  assumes *assoc*: $(x \otimes y) \otimes z = x \otimes (y \otimes z)$

class *monoid* = *semigroup* +
  fixes *neutral* :: α (**1**)
  assumes *neutl*: $\mathbf{1} \otimes x = x$
    and *neutr*: $x \otimes \mathbf{1} = x$

instantiation *nat* :: *monoid*
begin

primrec *mult_nat* **where**
    $0 \otimes n = (0{::}nat)$
  | $Suc\ m \otimes n = n + m \otimes n$

definition *neutral_nat* **where**
  $\mathbf{1} = Suc\ 0$

lemma *add_mult_distrib*:
  fixes *n m q* :: *nat*
  shows $(n + m) \otimes q = n \otimes q + m \otimes q$
  by (*induct n*) *simp_all*

instance proof
  fix *m n q* :: *nat*
  show $m \otimes n \otimes q = m \otimes (n \otimes q)$
    by (*induct m*) (*simp_all add*: *add_mult_distrib*)
  show $\mathbf{1} \otimes n = n$
    by (*simp add*: *neutral_nat_def*)
  show $m \otimes \mathbf{1} = m$
    by (*induct m*) (*simp_all add*: *neutral_nat_def*)
qed

end

We define the natural operation of the natural numbers on monoids:

primrec *pow* :: $nat \Rightarrow \alpha{::}monoid \Rightarrow \alpha{::}monoid$ **where**
    $pow\ 0\ a = \mathbf{1}$
  | $pow\ (Suc\ n)\ a = a \otimes pow\ n\ a$

This we use to define the discrete exponentiation function:

definition *bexp* :: $nat \Rightarrow nat$ **where**
  $bexp\ n = pow\ n\ (Suc\ (Suc\ 0))$

The corresponding code:

```
module Example where {


data Nat = Zero_nat | Suc Nat;

plus_nat :: Nat -> Nat -> Nat;
plus_nat (Suc m) n = plus_nat m (Suc n);
plus_nat Zero_nat n = n;

class Semigroup a where {
  mult :: a -> a -> a;
};

class (Semigroup a) => Monoid a where {
  neutral :: a;
};

pow :: forall a. (Monoid a) => Nat -> a -> a;
pow Zero_nat a = neutral;
pow (Suc n) a = mult a (pow n a);

neutral_nat :: Nat;
neutral_nat = Suc Zero_nat;

mult_nat :: Nat -> Nat -> Nat;
mult_nat Zero_nat n = Zero_nat;
mult_nat (Suc m) n = plus_nat n (mult_nat m n);

instance Semigroup Nat where {
  mult = mult_nat;
};

instance Monoid Nat where {
  neutral = neutral_nat;
};

bexp :: Nat -> Nat;
bexp n = pow n (Suc (Suc Zero_nat));

}
```

An inspection reveals that the equations stemming from the **primrec** statement within instantiation of class *semigroup* with type *nat* are mapped to a separate function declaration `mult_nat` which in turn is used to provide the right-hand side for the `instance Semigroup Nat`. This perfectly agrees with the restriction that *inst* statements may only contain one single equation for each class class parameter (see §3.2.2). The **instantiation** mechanism manages that for each class parameter $f\ [\kappa\ \overline{\alpha{::}s_k}]$ to be defined by term t a *shadow constant* $f_\kappa\ [\overline{\alpha{::}s_k}]$ is defined as follows:

constdef $f_\kappa\_primitive\_def$: $f_\kappa\ [\overline{\alpha{::}s_k}] :\equiv t$
overload $f_\kappa\_overload\_def$: $f\ [\kappa\ \overline{\alpha{::}s_k}] :\equiv f_\kappa\ [\overline{\alpha{::}s_k}]$

From this the proper definition follows by transitivity:

theorem f_κ_def: $f\ [\kappa\ \overline{\alpha{::}s_k}] \equiv t\ \langle proof\rangle$

Equation $f_\kappa\_overload\_def$ is used for the *inst* statement; using it as a rewrite rule, code equations for $f\ [\kappa\ \overline{\alpha{::}s_k}]$ can be turned into code equations for $f_\kappa\ [\overline{\alpha{::}s_k}]$. This happens transparently, providing the illusion that class parameters can be instantiated with more than one equation.

This is a convenient place to show how explicit dictionary construction manifests in generated code. Here, the same example in *OCaml*:

```
module Example =
struct

type nat = Zero_nat | Suc of nat;;

let rec plus_nat
  x0 n = match x0, n with Suc m, n -> plus_nat m (Suc n)
    | Zero_nat, n -> n;;

type 'a semigroup = {mult : 'a -> 'a -> 'a};;
let mult _A = _A.mult;;

type 'a monoid = {semigroup_monoid : 'a semigroup; neutral : 'a};;
let neutral _A = _A.neutral;;

let rec pow _A
  x0 a = match x0, a with Zero_nat, a -> neutral _A
    | Suc n, a -> mult _A.semigroup_monoid a (pow _A n a);;

let neutral_nat : nat = Suc Zero_nat

let rec mult_nat
  x0 n = match x0, n with Zero_nat, n -> Zero_nat
    | Suc m, n -> plus_nat n (mult_nat m n);;

let semigroup_nat = ({mult = mult_nat} : nat semigroup);;

let monoid_nat =
  ({semigroup_monoid = semigroup_nat; neutral = neutral_nat} :
    nat monoid);;

let rec bexp n = pow monoid_nat n (Suc (Suc Zero_nat));;

end;; (*struct Example*)
```

The translation follows the abstract rules given in §3.2.7 but uses a little bit syntactic sugar: instead of datatypes, it uses record types for dictionary types.

3.3.3 The preprocessor

As introduced in §3.1.3, the *preprocessor* allows to employ arbitrary transformations on the initial raw code equations E_0. The preprocessor does *not* interfere with the meta-theory behind the code generator since all the steps it is able to do are carried out through the *LCF* inference kernel — it just provides means of implicit automation. Typical tasks of the preprocessor include:

- dependency analysis: if a constant f occurs on the right hand side of a code equation in E_Θ, all code equations in E_0 headed by f are added to E_Θ.
- normalising type arguments such that they are the same across all code equations for a particular constant *f*.
- specialising sort constraints in E_Θ to achieve a well-sorted system ($\mho$, Σ_P, ω, E_Θ) (see further §3.3.5).

Beyond that, arbitrary LCF-style rewrite transformations can be configured, which typically include:

- replacing non-executable constructs by executable ones;
  e.g. rewrite rule $\bigwedge x\ xs.\ x \in \mathit{set}\ xs \longleftrightarrow \mathit{member}\ xs\ x$
- replacing executable but inconvenient constructs;
  e.g. rewrite rule $\bigwedge xs.\ xs = [\,] \longleftrightarrow \mathit{null}\ xs$

Various more ambitious applications will be presented in §4.2.2 and §4.2.3.

3.3.4 Equality

An interesting question is how *HOL* equality $(=)$ is handled by the code generator. Constant $(=)$ is characterised in *HOL* by the following axiomatisation:

$$\begin{array}{l}
\mathit{refl} : \bigwedge t.\ t = t \\
\mathit{subst} : \bigwedge s\ t\ P.\ s = t \Longrightarrow P\ s \Longrightarrow P\ t \\
\mathit{ext} : \bigwedge f\ g.\ (\bigwedge x.\ f\ x = g\ x) \Longrightarrow f = g
\end{array}$$

It is anything but obvious how this shall yield something executable. One Haskellish option could be to ignore $(=)$ entirely but instead provide a different equality operation *eq* belonging to a type class *eq* which is then supposed to implement the desired notion of “equality”. But this is not feasible in practice — people tend to use $(=)$ quite often, and this would burden the user to provide two versions of theorems, one with $(=)$ and the other with *eq*. However the use of a type class shows the way. The key addition is to ensure that $(=)$ and *eq* behave the same:

class *eq* =
  fixes *eq* :: $\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$
  assumes *eq*: $\mathit{eq}\ x\ y \longleftrightarrow x = y$

This allows

- to implement $(=)$ by *eq* using the equation $x = y \longleftrightarrow \mathit{eq}\ x\ y$
- to provide suitable code equations for *eq* on a particular type – which for datatypes can be automated easily.

What remains is to propagate the constraint *eq* through the whole system of code equations, to achieve a well-sorted system. For example, the equations for the list membership test:

$$\begin{array}{l}
\mathit{member}\ [\alpha{::}\mathit{type}] :: \alpha\ \mathit{list} \Rightarrow \alpha \Rightarrow \mathit{bool} \\
\mathit{member}\ [\alpha{::}\mathit{type}]\ [\,]\ y \longleftrightarrow \mathit{False} \\
\mathit{member}\ [\alpha{::}\mathit{type}]\ (x : xs)\ y \longleftrightarrow x = y \vee \mathit{member}\ [\alpha{::}\mathit{type}]\ xs\ y
\end{array}$$

are given an additional class constraint *eq* on the list type in order to fit together with the code equation $x = y \longleftrightarrow \mathit{eq}\ x\ y$ of $(=)$:

$$\begin{array}{l}
\mathit{member}\ [\alpha{::}\mathit{eq}] :: \alpha{::}\mathit{eq}\ \mathit{list} \Rightarrow \alpha{::}\mathit{eq} \Rightarrow \mathit{bool} \\
\mathit{member}\ [\alpha{::}\mathit{eq}]\ [\,]\ y \longleftrightarrow \mathit{False} \\
\mathit{member}\ [\alpha{::}\mathit{eq}]\ (x : xs)\ y \longleftrightarrow x = y \vee \mathit{member}\ [\alpha{::}\mathit{eq}]\ xs\ y
\end{array}$$

This propagation of sort constraints need not be done manually by the user since the preprocessor does this automatically.

The sketched approach towards implementing equality has two major advantages: it does not touch the foundation of the code generator since only inner-logical devices are applied, and it does not depend on any notion of equality in target languages which can be quite different from *HOL* equality.

3.3.5 Producing well-sorted systems

The *eq* class yields an example demonstrating how sort constraints in $\mho_P$ may influence sort arguments in Σ_P:

class $\mathit{total\_order} =$
  fixes $\mathit{less\_eq} :: \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$ (**infixl** $\preceq$ 50)
  assumes $\mathit{order\_refl}$: $x \preceq x$
  and $\mathit{order\_trans}$: $x \preceq y \Longrightarrow y \preceq z \Longrightarrow x \preceq z$
  and $\mathit{antisym}$: $x \preceq y \Longrightarrow y \preceq x \Longrightarrow x = y$
  and linear: $x \preceq y \vee y \preceq x$

lemma $\mathit{not\_less\_eq}$:
  $\neg\, x \preceq y \longleftrightarrow y \preceq x \wedge x \neq y$
  using *linear* **by** (*auto simp*: *order_refl antisym*)

instantiation $* :: (\mathit{total\_order}, \mathit{total\_order})\ \mathit{total\_order}$
begin

definition $\mathit{less\_eq\_prod} :: \alpha \times \beta \Rightarrow \alpha \times \beta \Rightarrow \mathit{bool}$ **where**
  $x \preceq y \longleftrightarrow \mathit{fst}\ x \preceq \mathit{fst}\ y \wedge \mathit{fst}\ x \neq \mathit{fst}\ y$
   $\vee\ \mathit{fst}\ x = \mathit{fst}\ y \wedge \mathit{snd}\ x \preceq \mathit{snd}\ y$

instance proof
qed (*auto simp*: *less_eq_prod_def order_refl not_less_eq*
  intro: *order_trans dest*: *antisym*)

end

definition $\mathit{between} :: \alpha{::}\mathit{total\_order} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$ **where**
  $\mathit{between}\ x\ y\ z \longleftrightarrow x \preceq y \wedge y \preceq z$

definition $\mathit{framed} :: \alpha{::}\mathit{total\_order} \Rightarrow \alpha \times \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$ **where**
  $\mathit{framed}\ x\ p\ y \longleftrightarrow \mathit{between}\ (x,\ x)\ p\ (y,\ y)$

The following generated *Haskell* code uses the built-in `Eq` class due to a default adaptation setup (see §3.4.1), a fact that need not bother here:

```
module Example where {


class Total_order a where {
  less_eq :: a -> a -> Bool;
};
```

```
less_eqa ::
  forall a b.
    (Eq a, Total_order a, Total_order b) => (a, b) -> (a, b) -> Bool;
less_eqa x y =
  less_eq (fst x :: a) (fst y :: a) && not (fst x == fst y) ||
    fst x == fst y && less_eq (snd x :: b) (snd y :: b);

instance (Eq a, Total_order a,
           Total_order b) => Total_order (a, b) where {
  less_eq = less_eqa;
};

between :: forall a. (Total_order a) => a -> a -> a -> Bool;
between x y z = less_eq x y && less_eq y z;

framed :: forall a. (Eq a, Total_order a) => a -> (a, a) -> a -> Bool;
framed x p y = between (x, x) p (y, y);

}
```

What is essential is that the instance $eq_\times$ which in the logic has arity

$$\times :: (\mathit{total\_order}, \mathit{total\_order})\ \mathit{total\_order}$$

under code generation maps to

$$\textsf{inst}\ \times\ (\alpha :: \mathit{total\_order} \cap \mathit{eq})\ (\beta :: \mathit{total\_order}) :: \mathit{total\_order}$$

How does the additional eq constraint on α enter the stage? Observe the code equation for $(\preceq)$ $[\alpha{::}\mathit{total\_order} \times \beta{::}\mathit{total\_order}]$ on the product type:

$$\begin{array}{l}(\preceq)\ [\alpha{::}\mathit{total\_order} \times \beta{::}\mathit{total\_order}]\ x\ y \longleftrightarrow (\preceq)\ [\alpha{::}\mathit{total\_order}]\ (\mathit{fst}\ x)\ (\mathit{fst}\ y) \\ \wedge\ \mathit{fst}\ x \neq \mathit{fst}\ y \vee \mathit{fst}\ x = \mathit{fst}\ y \wedge (\preceq)\ [\beta{::}\mathit{total\_order}]\ (\mathit{snd}\ x)\ (\mathit{snd}\ y)\end{array}$$

The occurrence of $(=)$ on type α issues the preprocessor to add an eq constraint on α:

$$\begin{array}{l}(\preceq)\ [\alpha{::}(\mathit{eq} \cap \mathit{total\_order}) \times \beta{::}\mathit{total\_order}]\ x\ y \longleftrightarrow \\ (\preceq)\ [\alpha{::}(\mathit{eq} \cap \mathit{total\_order})]\ (\mathit{fst}\ x)\ (\mathit{fst}\ y) \wedge \mathit{fst}\ x \neq \mathit{fst}\ y\ \vee \\ \mathit{fst}\ x = \mathit{fst}\ y \wedge (\preceq)\ [\beta{::}\mathit{total\_order}]\ (\mathit{snd}\ x)\ (\mathit{snd}\ y)\end{array}$$

Recalling §3.3.2, internally the class parameter $(\preceq)$ $[\alpha{::}\mathit{total\_order} \times \beta{::}\mathit{total\_order}]$ on products is replaced by $(\preceq_\times)$ $[\alpha{::}\mathit{type}]$ $[\beta{::}\mathit{type}]$. The equation underlying the *inst* for $\mathit{total\_order}_\times$ then is

$$\begin{array}{l}(\preceq)\ [\alpha{::}\mathit{total\_order} \times \beta{::}\mathit{total\_order}] = \\ (\preceq_\times)\ [\alpha{::}\mathit{total\_order}]\ [\beta{::}\mathit{total\_order}]\end{array}$$

Since the $(\preceq_\times)$ $[\alpha{::}\mathit{type}]$ $[\beta{::}\mathit{type}]$ on the right hand side again enforces an eq constraint on α, this requires to specialise the equation:

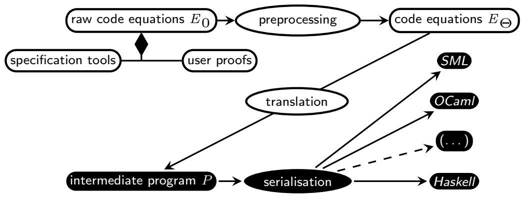
$$\begin{array}{l}(\preceq)\ [\alpha{::}(\mathit{eq} \cap \mathit{total\_order}) \times \beta{::}\mathit{total\_order}] = \\ (\preceq_\times)\ [\alpha{::}(\mathit{eq} \cap \mathit{total\_order})]\ [\beta{::}\mathit{total\_order}]\end{array}$$

Consequently the whole *inst* is specialised.

This propagating of sort constraints through a system of code equations is performed by the preprocessor by means of a fixpoint algorithm. Equality using class eq is a canonical example how the preprocessor propagates sort constraints through a system of code equations; a further instance of this problem can be found in §4.2.1.

3.4 Concerning serialisation

Serialisation prints an intermediate program piecewise into concrete source code of a target language program; since the structure of the intermediate program already is close to the target program, this conceptually involves little further transformations. Accomplishing concrete target language source code is rather technical; we will only touch the subject here and point to the documentation for further reading [23].



3.4.1 Adaptation

Technically, each serialiser consists of a generic part providing specific printing rules for statements, terms, types etc., and an *adaptation* layer allowing for special printing of particular constants, type constructors and classes. Typically applications of adaptation include:

- readability and aesthetics; e.g. for fundamental types like tuples, lists and options, the default setup is to use the corresponding target-language counterparts, including pretty syntax.
- efficiency; e.g. the possibility to implement *HOL ints* by target-language built-in integers as described in §4.2.2.
- interaction with predefined target-language ingredients; e.g. mastering imperative data structures as described in §4.3.

3.4.2 Subtle situations and borderline cases

In practically rarely occurring situations, serialisation is not straightforward. We give a cursory glance of some:

***data* without constructors.** The certificates for *data* statements permit degenerated types without any constructor. Though this is of little practical use, it is not rejected by the translation. Serialisation for *Haskell* bears no problem; concerning *ML*, such an empty datatype `foo` is serialised as `datatype foo = Foo`, where `Foo` is an identifier not used elsewhere in the program.

***fun* without equations.** A further degenerate case are *fun* statements without any equations. These can be seen as functions which always fail. For this reason it is legitimate to translate such empty *fun* statements into functions raising an exception or error.

Mutual recursion between *fun* statements. Traditionally, *ML* languages distinguish between value bindings `val` (without function arguments) and function bindings `fun` (with function arguments). Value bindings do not permit mutual recursion; however in some higher-order situations (e.g. §4.2.3), *fun* statements with *no* arguments can be mutually recursive. This is accomplished by adding unit values `()` as pseudo-arguments to the function bindings and invocations; after this mutually recursive block, simple value bindings `val foo = foo ()` allow to use `foo` in the further run naively without an additional `()`. The same mechanisms allows to accomplish mututal recursion between *fun* and *inst* statements in *ML*.

Haskell has just one function declaration concept and therefore does not need this workaround.

Polymorphic recursion. Polymorphic recursion occurs when a constant in a recursive specification occurs with different type instances. In traditional Hindley-Milner type inference as implemented in *ML* and *Isabelle*, this is not possible since the inference cannot cope with this. However there can still be a valid Hindley-Milner type: *Haskell* masters this problem by requiring an explicit type constraint which then has just to be checked rather than inferred.

Although the *HOL* specification tools themselves do not allow for polymorphic recursion, nonetheless code equations can be constructed which contain polymorphic recursion, e.g. the following equations for list reversal:

$$\begin{aligned}&\mathit{rev}\ (x : \mathit{xs}) = \mathit{flat}\ (\mathit{rev}\ [\![[\![x]\!],\ \mathit{rev}\ \mathit{xs}]\!])\\ &\mathit{rev}\ [\![x,\ y]\!] = [\![y,\ x]\!]\\ &\mathit{rev}\ [\![x]\!] = [\![x]\!]\\ &\mathit{rev}\ [\![\,]\!] = [\![\,]\!]\end{aligned}$$

Interesting examples of polymorphic recursion [45] require advanced recursive datatypes which are beyond the capabilities of the **datatype** command, but this does not imply that these types cannot be constructed some other way. So, although this example is pathological, it demonstrates that code equations with polymorphic recursion can occur in principle.

The *ML* serialiser does not provide a workaround for this; the code is naively generated:

```
fun rev [] = []
  | rev [x] = [x]
  | rev [x, y] = [y, x]
  | rev (x :: xs) = flat (rev [[x], rev xs]);
```

and then rejected by the *ML* compiler.

Dictionaries in contravariant position. Another issue affects type classes whose parameters have a type of a particular form:

class *typerank* =
  fixes *typerank* :: α *itself* $\Rightarrow$ *nat*

Here the type variable α occurs only in the input arguments, not in the output value; let us call this *contravariant position*.[1] The class *typerank* allows to encode the rank

[1]The type α *itself* is the phantom type used in the logical interpretation of type classes (see §2.3.2); the corresponding intermediate statement is *data* α *itself* = *TYPE*.

of a type in the logic: the rank of an atomic type is 0, for a parametrised type the rank is the successor of the maximum of the ranks of all its arguments. E.g. for *nat* and $\alpha \times \beta$, the instances look this:

instantiation *nat* :: *typerank*
begin

definition
  typerank (_ :: *nat itself*) = 0

instance ..

end

instantiation $*$ :: (*typerank*, *typerank*) *typerank*
begin

definition
  typerank (_ :: ($\alpha \times \beta$) *itself*) =
    Suc (*max* (*typerank* (*TYPE* α)) (*typerank* (*TYPE* β)))

instance ..

end

Let us inspect the generated code for *typerank* on products in *Haskell*:

```
data Itself a = Type;

class Typerank a where {
  typeranka :: Itself a -> Nat;
};

typerank ::
  forall a b. (Typerank a, Typerank b) => Itself (a, b) -> Nat;
typerank Type =
  Suc (maxa (typeranka (Type :: Itself a))
        (typeranka (Type :: Itself b)));
```

The occurrences of `Type` on the right hand side are decorated with explicit type constraints; otherwise the context would provide too little information to infer the type of each `Type`. The *Haskell* serialiser uses an heuristic to determine whether such type annotations are necessary.

In *ML* dictionaries are represented explicitly, so this problem does not occur there.

3.5 What is “executable”?

Until this moment we have refrained to state explicitly which fragment of *HOL* is “executable”. On the one side, in §3.2.5 some properties were given which a system of code equations E_Θ must obey to be translatable to an intermediate program P: most notably a suitable choice of constructors Ξ, and well-sortedness.

On the other side, practically this answer is not very helpful: First, it does not state anything whether E_Θ is actually “meaningful” — the empty system $E_\Theta = \{\}$ is well-sorted and yields a partially correct program by definition! Second, even a system violating those criteria need not to be inherently non-executable because it could just result from an inappropriate choice of code equations — look ahead to §4.1.2 to see an example of something that looks non-executable at first sight but in fact is.

For this reason we do not use the attribute “executable” in a formal way but pragmatically: specifications are executable if they can be turned into executable programs in a target language using the code generator with support of the preprocessor, still leaving open the question whether the generated program is able to perform something “useful”. This leads to the following classification of executable specifications: executable *Isar* specifications include

- **datatype** and **class** statements;
- **definition**, **primrec** and **fun** statements with only executable constants on the right hand side;
- **instantiation**s using executable means of specification;
- types and constants with an explicit executable *implementation* consisting of appropriate constructors and code equations;
- constants which are turned executable by a suitable preprocessor setup (e.g. equality on proper datatypes, cf. §3.3.4).

Note that in equations stemming from **primrec** and **fun** only patterns occur on the left hand side (cf. §3.2.3). For generic **function**s with explicit construction proofs this does not hold necessarily:

function $\mathit{even} :: \mathit{nat} \Rightarrow \mathit{bool}$ **where**
  $\mathit{even}\ (2 * n) \longleftrightarrow \mathit{True}$
| $\mathit{even}\ (2 * n + 1) \longleftrightarrow \mathit{False}$
proof $-$
 fix $P :: \mathit{bool}$
 fix $m :: \mathit{nat}$
 assume 0: $\bigwedge n.\ m = 2 * n \Longrightarrow P$
 and 1: $\bigwedge n.\ m = 2 * n + 1 \Longrightarrow P$
 show P **proof** $(\mathit{cases}\ m\ \mathit{mod}\ 2 = 0)$
  case True **then have** $m = 2 * (m\ \mathit{div}\ 2)$ **by** *arith*
  with 0 **show** P **.**
 next
  case False **then have** $m = 2 * (m\ \mathit{div}\ 2) + 1$ **by** *arith*
  with 1 **show** P **.**
 qed
qed (*simp_all*, *arith*)

termination *even* **by** *rule+*

This is a perfect logically consistent function specification, but the left hand sides of the resulting equations contain non-constructors!

The following chapters contain many fragments of executable specifications, either built with the *HOL* specification tool box or manually using explicit proofs of equations. They illustrate various practically reasonable ways to develop executable specifications using *HOL* and the code generator together and thus are example applications supporting the aim of this thesis: bringing the worlds of theorem proving and functional programming closer together.

CHAPTER **4**

Turning specifications into programs

> *My thesis is that programming is not at the bottom of the intellectual pyramid, but at the top. It's creative design of the highest order. It isn't monkey or donkey work; rather, as Edsger Dijkstra famously claimed, it's amongst the hardest intellectual tasks ever attempted.*
> Richard Bornat, British computer scientist,
> from: In defence of programming

So far we have presented the basic ingredients of the code generator. We now turn the focus to the question how we can actually make *use* of it. To this end we discuss various specification examples which illustrate datatype abstraction in *HOL*, a fundamental principle to turn *abstract* specifications into *executable* ones. We sketch some applications combining the code generator with the existing expressiveness of the deductive system: executing inductive predicates, propositions on finite types, binary representation of natural numbers. As an example for an adaptation of the serialiser, we discuss possibilities to operate with destructive data structures within the pure logic of *HOL*. Further sections demonstrate how the code generator collaborates with other parts of the system: evaluation, counterexample generation, proof terms, and proof extraction.

Contents

4.1 Datatype abstraction

In the examples shown so far (e.g in §3.3.1), the constructors stem directly from *HOL* **datatype** declarations. This is just a convenience since in many cases this will be the desired thing. Nonetheless the choice of constructors for datatypes in generated code is free as long as the types of constructors conform to some syntactic restrictions (cf. §3.2.3). This freedom establishes a simple concept for datatype abstraction, which we will examine with some examples.

In contrast to the previous chapter which describes how the code generator *works*, this one focusses on how it can be *applied*. So all specification and proof developements are carried out by the user of the system, unless it is explicitly indicated that particular steps occur automatically.

4.1.1 Amortised queues revisited

From a practical point of view, the amortised queues presented in §3.3.1 are not wholly convincing: the amortised representation is convenient for execution but clutters proofs involving queues considerably.

One improvement could be to establish enough abstract properties of amortised queues once and for all which can be used in further proofs and hide the primitive details of the specification.

Here we give a different, more direct approach. Let us start with a logical specification of queues in a straightforward manner:

datatype α *queue* $=$ *Queue* $(\alpha\ list)$

$empty :: \alpha\ queue$
$empty = Queue\ [\![\,]\!]$

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

$dequeue :: \alpha\ queue \Rightarrow \alpha\ option \times \alpha\ queue$
$dequeue\ (Queue\ [\![\,]\!]) = (None,\ Queue\ [\![\,]\!])$
$dequeue\ (Queue\ (x : xs)) = (Some\ x,\ Queue\ xs)$

On top of this, we are able to provide an alternative amortised characterisation of queues as follows: first, we specify:

$$AQueue :: \alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ queue$$
$$AQueue\ xs\ ys = Queue\ (ys\ @\ rev\ xs)$$

Thus $AQueue$ logically describes the embedding of a pair of lists representing an amortised queue into the corresponding value of type $\alpha\ queue$. With this definition the following equations are easily proved:

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

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

$$dequeue\ (AQueue\ xs\ [\![]\!]) =$$
$$(if\ xs = [\![]\!]\ then\ (None,\ AQueue\ [\![]\!]\ [\![]\!])\ else\ dequeue\ (AQueue\ [\![]\!]\ (rev\ xs)))$$
$$dequeue\ (AQueue\ xs\ (y : ys)) = (Some\ y,\ AQueue\ xs\ ys)$$

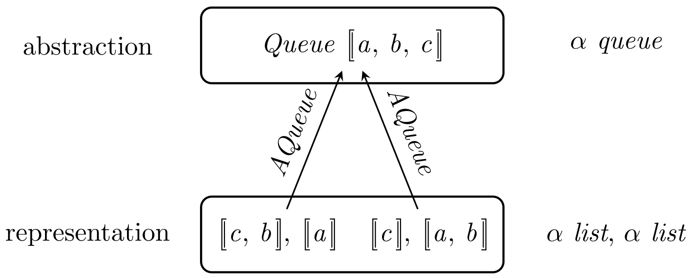
We can use these equations as code equations for queues immediately:

```
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 [] (rev xs)));

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

In contrast to the amortised queues from §3.3.1, these proof steps for setting up code generation need only be made once; all other proofs about queues just refer to the plain, direct specification from above.

This approach mirrors an established methodology in software engineering: *datatype abstraction* [29]. The idea is to *encapsulate* the concrete representation of an abstract type (in our example, $\alpha\ queue$) such that only primitive operations (here, *empty*, *enqueue* and *dequeue*) are allowed to operate directly on it; the primitive operations then form an interface to the outside world through which operations on values of the abstract type take place.

What does access to the concrete representation mean in our setting? The embedding of the concrete representation $\alpha\ list$, $\alpha\ list$ to the abstract type $\alpha\ queue$ is mediated by the constant $AQueue :: \alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ queue$. Accessing the representation of a value of type $\alpha\ queue$ means to inverse the function application of $AQueue$. This inversion is *not* expressed by an explicit constant; rather, the value is matched against a pattern with $AQueue$ as constructor, e.g. in the equations for *enqueue* and *dequeue*. The scenario is depicted in the following picture:



It is obvious that the abstraction function (in our case $AQueue$) does not need to be injective, e.g. $AQueue\ [\![c,\ b]\!]\ [\![a]\!]$ and $AQueue\ [\![c]\!]\ [\![a,\ b]\!]$ represent the same value. In other words, when matching a term against $AQueue$ there is no guarantee that the arguments are in a canonical representation. It does not matter in this queue example but will rear its head in the next section: we are not able to specify *invariants* on representations.

This datatype abstraction concept is *implicit* since it naturally stems from the meta-theory of code generation without any additional checks necessary: violations of encapsulation are impossible *by construction*. E.g. if we would specify a further operation

$$
\begin{array}{l}
tap :: \alpha\ queue \Rightarrow \alpha\ option \\
tap\ (Queue\ [\![\,]\!]) = None \\
tap\ (Queue\ (x : xs)) = Some\ x
\end{array}
$$

which makes use of the logical datatype constructor $Queue$, this would be rejected since under code generation $Queue$ is *not* the datatype constructor for type $\alpha\ queue$. Nonetheless we can introduce tap as another primitive operation proving the following equation:

$$
\begin{array}{l}
tap\ (AQueue\ xs\ ys) = \\
(case\ ys\ of\ [\![\,]\!] \Rightarrow if\ xs = [\![\,]\!]\ then\ None\ else\ Some\ (last\ xs) \\
\ \mid\ y : x \Rightarrow Some\ y)
\end{array}
$$

which again respects the rules of encapsulation.

4.1.2 Implementing rational numbers

Not every HOL type is a **datatype** in the logical sense; some rather represent abstract logical concepts. A prominent example are rational numbers, type rat. Internally, they are constructed as a quotient of pairs of integer numbers of type int [48] and a corresponding typedef. For concrete rat values the following constant is provided:

$$Fract :: int \Rightarrow int \Rightarrow rat$$

$Fract\ p\ q$ is the value $\frac{p}{q}$; in the case $q = 0$, the value is 0.[1]

$Fract$ is not injective, e.g.

$$Fract\ 35\ 42 = Fract\ 5\ 6$$

[1] It would be possible the leave $Fract\ p\ q$ underspecified for $q = 0$; HOL is a total logic, so the *totalisation* by defining $Fract\ k\ 0 = 0$ does not harm but sometimes avoids dull case distinctions.

and thus *Fract* is no **datatype** constructor in the strict *HOL* sense. But *Fract* can serve as a constructor in a *data* statement for the *rat* type:[2]

```
data Rat = Fract Integer Integer;
```

Appropriate equations, e.g. for multiplication, are easily proved:

$$\mathit{Fract}\ a\ b * \mathit{Fract}\ c\ d = \mathit{Fract}\ (a * c)\ (b * d)$$

resulting in

```
times_rat :: Rat -> Rat -> Rat;
times_rat (Fract a b) (Fract c d) = Fract (a * c) (b * d);
```

For addition, case distinctions are needed:

$$\begin{array}{l}\mathit{Fract}\ a\ b + \mathit{Fract}\ c\ d = \\ (\mathit{if}\ b = 0\ \mathit{then}\ \mathit{Fract}\ c\ d \\ \ \mathit{else}\ \mathit{if}\ d = 0\ \mathit{then}\ \mathit{Fract}\ a\ b\ \mathit{else}\ \mathit{Fract}\ (a * d + c * b)\ (b * d))\end{array}$$

resulting in

```
plus_rat :: Rat -> Rat -> Rat;
plus_rat (Fract a b) (Fract c d) =
  (if b == 0 then Fract c d
    else (if d == 0 then Fract a b else Fract (a * d + c * b) (b * d)));
```

Is there a possibility to avoid the case distinctions on the denominators? The established methodology in software engineering for dealing with such pathological cases are *invariants*: the primitive operations which are permitted access to the *representation* of values of an *abstract datatype* have to respect an appropriate *invariant*. In our case, an appropriate invariant would be that the denominator never equals 0. This is typically expressed using equations with premises, e.g.

$$b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow \mathit{Fract}\ a\ b + \mathit{Fract}\ c\ d = \mathit{Fract}\ (a * d + c * b)\ (b * d)$$

But these we cannot use within our code generator framework, so we are not able to employ invariants. This turns out even more unsatisfactory when we consider the code equation for equality

$$\begin{array}{l}\mathit{eq}\ (\mathit{Fract}\ a\ b)\ (\mathit{Fract}\ c\ d) \longleftrightarrow \\ (\mathit{if}\ b = 0\ \mathit{then}\ c = 0 \vee d = 0 \\ \ \mathit{else}\ \mathit{if}\ d = 0\ \mathit{then}\ a = 0 \vee b = 0\ \mathit{else}\ a * d = b * c)\end{array}$$

with the corresponding code

```
eq_rat :: Rat -> Rat -> Bool;
eq_rat (Fract a b) (Fract c d) =
  (if b == 0 then c == 0 || d == 0
    else (if d == 0 then a == 0 || b == 0 else a * d == b * c));
```

[2]For convenience all *rat* examples use a setup of the serialiser which maps *HOL int* to *Haskell* `Integer`; see §3.4.1.

A convenient invariant would be that the denominator is strictly positive and enumerator and denominator are coprime; then the equality check could simply check enumerators and denominators for equality. In lack of this, we must compute the cross product explicitly, and check the denominators for zero.

Despite this deficiency, it can be reasonable to normalise enumerator and denominator. We can make use of the fact that in the logic *rat* is an abstract type without any notion of concrete representation. So we define a normaliser operation $\mathit{Fract}_{\downarrow}$ as equivalent to *Fract*:

$$\mathit{Fract}_{\downarrow}\ a\ b = \mathit{Fract}\ a\ b$$

Next we prove a suitable code equation for $\mathit{Fract}_{\downarrow}$ which normalises its arguments:

$$\begin{array}{l}\mathit{Fract}_{\downarrow}\ a\ b = \\ (\mathit{if}\ a = 0 \vee b = 0\ \mathit{then}\ \mathit{Fract}\ 0\ 1 \\ \ \mathit{else}\ \mathit{let}\ c = \mathit{gcd}\ a\ b \\ \qquad \mathit{in}\ \mathit{if}\ 0 < b\ \mathit{then}\ \mathit{Fract}\ (a\ \mathit{div}\ c)\ (b\ \mathit{div}\ c) \\ \qquad\quad \mathit{else}\ \mathit{Fract}\ (-\ (a\ \mathit{div}\ c))\ (-\ (b\ \mathit{div}\ c)))\end{array}$$

Finally we replace each *Fract* by $\mathit{Fract}_{\downarrow}$, which is trivial since both are equal.

The resulting code equations for multiplication and addition thus look as follows:[3]

$$\mathit{Fract}\ a\ b * \mathit{Fract}\ c\ d = \mathit{Fract}_{\downarrow}\ (a * c)\ (b * d)$$

$$\begin{array}{l}\mathit{Fract}\ a\ b + \mathit{Fract}\ c\ d = \\ (\mathit{if}\ b = 0\ \mathit{then}\ \mathit{Fract}\ c\ d \\ \ \mathit{else}\ \mathit{if}\ d = 0\ \mathit{then}\ \mathit{Fract}\ a\ b\ \mathit{else}\ \mathit{Fract}_{\downarrow}\ (a * d + c * b)\ (b * d))\end{array}$$

In the case of rational numbers, the lack of a concept for invariants leads to a loss of efficiency; in the examples in the next section more fundamental problems arise.

4.1.3 Mappings

A common device in higher-order settings are *mappings*, associations from *keys* α to *values* β, which are logically represented as functions $\alpha \Rightarrow \beta\ \mathit{option}$. This lightweight approach is very convenient for reasoning, but code generated *directly* from that is rarely feasible: the naive encoding results in inconvenient towers of λ-abstractions.

Our datatype abstraction concept turns out helpful here. As a prerequisite, we wrap up the type $\alpha \Rightarrow \beta\ \mathit{option}$ into a trivial datatype:

$$\mathbf{datatype}\ (\alpha, \beta)\ \mathit{map} = \mathit{Map}\ (\alpha \Rightarrow \beta\ \mathit{option})$$

This wrapping is necessary since we need an explicit type constructor *map* to provide constructors for. On top of this we define primitive operations:

$$\mathit{Mapping.empty} :: (\alpha, \beta)\ \mathit{map}$$

$$\mathit{Mapping.lookup} :: (\alpha, \beta)\ \mathit{map} \Rightarrow \alpha \Rightarrow \beta\ \mathit{option}$$

$$\mathit{Mapping.update} :: \alpha \Rightarrow \beta \Rightarrow (\alpha, \beta)\ \mathit{map} \Rightarrow (\alpha, \beta)\ \mathit{map}$$

$$\mathit{Mapping.delete} :: \alpha \Rightarrow (\alpha, \beta)\ \mathit{map} \Rightarrow (\alpha, \beta)\ \mathit{map}$$

$$\mathit{Mapping.size} :: (\alpha, \beta)\ \mathit{map} \Rightarrow \mathit{nat}$$

[3] See §C.1 for the corresponding *Haskell* code.

Naive code equations using cascaded λ-abstractions look as follows, where the function $point :: \alpha \Rightarrow (\beta \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$ represents a point-wise update on a function value.[4]

$$Mapping.empty = Map\ (\lambda x.\ None)$$

$$Mapping.lookup\ (Map\ f) = f$$

$$Mapping.update\ k\ v\ (Map\ f) = Map\ (point\ k\ (\lambda\_.\ Some\ v)\ f)$$

$$Mapping.delete\ k\ (Map\ f) = Map\ (point\ k\ (\lambda\_.\ None)\ f)$$

These equations exhibit the deficiencies of this approach: encoding each mapping update into a function update is inefficient, and the mapping values are not inspectible, e.g. there is no way to determine the number of keys in a mapping.

Association lists. A step to a more execution-oriented implementation are association lists, i.e. explicit mappings of $\alpha \times \beta$ values:

$$\begin{aligned}
&AList.lookup :: (\alpha \times \beta)\ list \Rightarrow \alpha \Rightarrow \beta\ option\\
&AList.lookup\ [\![]\!]\ k = None\\
&AList.lookup\ (x : xs)\ k =\\
&(if\ k = fst\ x\ then\ Some\ (snd\ x)\ else\ AList.lookup\ xs\ k)
\end{aligned}$$

$$\begin{aligned}
&AList.update :: \alpha \Rightarrow \beta \Rightarrow (\alpha \times \beta)\ list \Rightarrow (\alpha \times \beta)\ list\\
&AList.update\ k\ v\ [\![]\!] = [\![(k,\ v)]\!]\\
&AList.update\ k\ v\ (x : xs) =\\
&(if\ k = fst\ x\ then\ (k,\ v) : xs\ else\ x : AList.update\ k\ v\ xs)
\end{aligned}$$

$$\begin{aligned}
&AList.delete :: \alpha \Rightarrow (\alpha \times \beta)\ list \Rightarrow (\alpha \times \beta)\ list\\
&AList.delete\ k\ [\![]\!] = [\![]\!]\\
&AList.delete\ k\ (x : xs) =\\
&(if\ k = fst\ x\ then\ AList.delete\ k\ xs\ else\ x : AList.delete\ k\ xs)
\end{aligned}$$

We specify the corresponding constructor $AList :: (\alpha \times \beta)\ list \Rightarrow (\alpha,\ \beta)\ map$ for mappings as

$$\begin{aligned}
&AList\ [\![]\!] = Mapping.empty\\
&AList\ (x : xs) = Mapping.update\ (fst\ x)\ (snd\ x)\ (AList\ xs)
\end{aligned}$$

which enables us to prove:[5]

$$Mapping.empty = AList\ [\![]\!]$$

$$Mapping.lookup\ (AList\ xs) = AList.lookup\ xs$$

$$Mapping.update\ k\ v\ (AList\ xs) = AList\ (AList.update\ k\ v\ xs)$$

$$Mapping.delete\ k\ (AList\ xs) = AList\ (AList.delete\ k\ xs)$$

$$Mapping.size\ (AList\ xs) = length\ (distinct\ (map\ fst\ xs))$$

In a conventional implementation the invariant would hold that no key occurs more than once in an association list. Since we cannot express invariants, the code equations for *Mapping.delete* has to run through whole the association list since keys

[4] See §C.2 for the corresponding *Haskell* code.
[5] See §C.3 for the corresponding *Haskell* code.

could occur duplicated; likewise *Mapping.size* must ignore duplicate keys “manually”.

Binary search trees. Another suitable implementation of mappings are binary search trees:

$$\begin{array}{l}\textbf{datatype}\ (\alpha,\ \beta)\ \mathit{tree} = \mathit{Empty}\\ \quad |\ \mathit{Branch}\ \beta\ \alpha\ ((\alpha,\ \beta)\ \mathit{tree})\ ((\alpha,\ \beta)\ \mathit{tree})\end{array}$$

$$\begin{array}{l}\mathit{Tree.lookup} :: (\alpha{::}\mathit{linorder},\ \beta)\ \mathit{tree} \Rightarrow \alpha \Rightarrow \beta\ \mathit{option}\\ \mathit{Tree.lookup}\ \mathit{Empty} = \mathit{Map.empty}\\ \mathit{Tree.lookup}\ (\mathit{Branch}\ v\ k\ l\ r) =\\ (\lambda k'.\ \mathit{if}\ k' = k\ \mathit{then}\ \mathit{Some}\ v\\ \qquad \mathit{else}\ \mathit{if}\ k' \le k\ \mathit{then}\ \mathit{Tree.lookup}\ l\ k'\ \mathit{else}\ \mathit{Tree.lookup}\ r\ k')\end{array}$$

$$\begin{array}{l}\mathit{Tree.update} :: \alpha \Rightarrow \beta \Rightarrow (\alpha{::}\mathit{linorder},\ \beta)\ \mathit{tree} \Rightarrow (\alpha{::}\mathit{linorder},\ \beta)\ \mathit{tree}\\ \mathit{Tree.update}\ k\ v\ \mathit{Empty} = \mathit{Branch}\ v\ k\ \mathit{Empty}\ \mathit{Empty}\\ \mathit{Tree.update}\ k'\ v'\ (\mathit{Branch}\ v\ k\ l\ r) =\\ (\mathit{if}\ k' = k\ \mathit{then}\ \mathit{Branch}\ v'\ k\ l\ r\\ \ \mathit{else}\ \mathit{if}\ k' \le k\ \mathit{then}\ \mathit{Branch}\ v\ k\ (\mathit{Tree.update}\ k'\ v'\ l)\ r\\ \qquad \mathit{else}\ \mathit{Branch}\ v\ k\ l\ (\mathit{Tree.update}\ k'\ v'\ r))\end{array}$$

$$\begin{array}{l}\mathit{Tree.keys} :: (\alpha,\ \beta)\ \mathit{tree} \Rightarrow \alpha\ \mathit{list}\\ \mathit{Tree.keys}\ \mathit{Empty} = [\![]\!]\\ \mathit{Tree.keys}\ (\mathit{Branch}\ \mathit{uu}\ k\ l\ r) = k : \mathit{Tree.keys}\ l\ @\ \mathit{Tree.keys}\ r\end{array}$$

$$\begin{array}{l}\mathit{Tree.size} :: (\alpha,\ \beta)\ \mathit{tree} \Rightarrow \mathit{nat}\\ \mathit{Tree.size}\ t =\\ \mathit{length}\ (\mathit{map\_filter}\ (\mathit{Tree.lookup}\ t)\ (\mathit{distinct}\ (\mathit{Tree.keys}\ t)))\end{array}$$

Then we define a constructor $\mathit{Tree} :: (\alpha,\ \beta)\ \mathit{tree} \Rightarrow (\alpha,\ \beta)\ \mathit{map}$ for mappings as

$$\mathit{Tree}\ t = \mathit{Map}\ (\mathit{Tree.lookup}\ t)$$

with the following implementation of the mapping operations:[6]

$$\mathit{Mapping.empty} = \mathit{Tree}\ \mathit{Empty}$$

$$\mathit{Mapping.lookup}\ (\mathit{Tree}\ t) = \mathit{Tree.lookup}\ t$$

$$\mathit{Mapping.update}\ k\ v\ (\mathit{Tree}\ t) = \mathit{Tree}\ (\mathit{Tree.update}\ k\ v\ t)$$

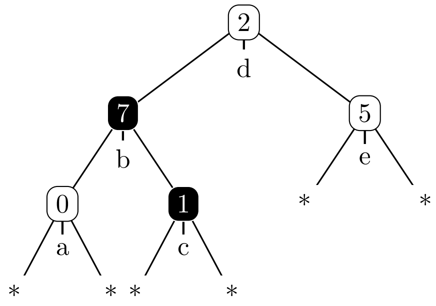
$$\mathit{Mapping.size}\ (\mathit{Tree}\ t) = \mathit{Tree.size}\ t$$

How does the absence of invariants affect the implementation? A search tree invariant in our case would express that all nodes in the left branch are less than the current key and all nodes in the right branch are strictly greater than the current key:

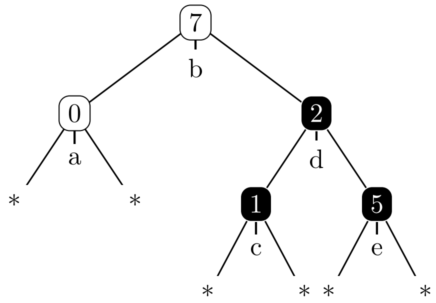
$$\begin{array}{l}\mathit{invariant}\ \mathit{Empty} \longleftrightarrow \mathit{True}\\ \mathit{invariant}\ (\mathit{Branch}\ v\ k\ l\ r) \longleftrightarrow\\ (\forall\, k'{\in}\mathit{set}\ (\mathit{Tree.keys}\ l).\ k' \le k)\ \wedge\\ (\forall\, k'{\in}\mathit{set}\ (\mathit{Tree.keys}\ r).\ k < k')\ \wedge\ \mathit{invariant}\ l\ \wedge\ \mathit{invariant}\ r\end{array}$$

[6] See §C.4 for corresponding *Haskell* code

Since we cannot express invariants, one might ask why our implementation works anyway. Have a look at the following example of a tree violating *invariant*:



The node with key 7 violates the invariant and renders itself and its right branch *dead*: none of the values of these nodes is ever considered by *Tree.lookup*. Similarly *Tree.update* does just pass through dead nodes while searching for a place to insert or replace a node; thus *Tree.lookup* and *Tree.update* also works on non-invariant trees. Not every operation is as liberal, especially when the tree structure is re-arranged. Consider a rotation to right in the tree root; in trees satisfying *invariant* rotation does not change the corresponding mapping; but in a tree containing dead nodes this does not hold in general:



Here tree node 7 has become alive whereas nodes 2 and 5 have become dead. So all operations involving a *restructuring* of the tree cannot ignore dead nodes, which makes it difficult to implement operations like *Mapping.delete* or balanced trees.

It would be possible to specify an operation

$$cut :: (\alpha,\ \beta)\ tree \Rightarrow (\alpha,\ \beta)\ tree$$

satisfying the properties

$$\bigwedge t.\ Tree.lookup\ (cut\ t) = Tree.lookup\ t$$
$$\bigwedge t.\ invariant\ (cut\ t)$$

I.e. *cut* eliminates all dead nodes from a tree. Then the implementation of all operations involving tree restructuring would have to *cut* their input argument tree first. Ironically, the operations to build trees inductively (*Empty*, *Tree.update*) never produce trees with dead nodes, but due to the lack of an invariant concept there is no way to utilise this. This makes any workaround like *cut* appear more like a parody than a solution.

4.1.4 Stocktaking

This survey yields two central results:

- The meta-theory of code generation yields an implicit concept for datatype abstraction. Indeed, the construction of appropriate representations for abstract types is the central proficiency in applying code generation. The used principles follow established techniques in software development, e.g. gradual improvement. This seems to indicate that code generation using shallow embedding is quite “natural” and intuitive.

- Datatype abstraction is useful but restricted due to the deficiency to express invariants. There is no simple way to circumvent this, but we will discuss possible solutions in §5.3.

4.2 Combining code generation and deductions

The preprocessor (c.f. §3.3.3) plays an essential role to obtain a practically usable system. We will illustrate this statement by a couple of examples whose main focus is to provide a suitable preprocessor setup to accomplish particular solutions. Some of them are used in the applications shown in §4.3 and later.

4.2.1 Enumerating finite types

§3.3.4 has shown how type classes accomplish executable equality; it has been demonstrated in §3.3.5 that this demands a sort inference algorithm to obtain a practically usable system. This sort inference mechanism is also useful in other applications than equality, one of which we discuss in this section.

As a motivating example, consider a specification involving character encodings. Encodings themselves are directly represented by functions $char \Rightarrow nat$, where *char* is a type consisting of 256 character symbols.[7]

Let us assume that the hypothetical specification which uses such encodings would involve the following things:

- Check whether a given encoding is injective.

- Check whether two given encodings are distinct.

Both concepts involve equality on functions, which is not executable in general. But in this example the domain of the underlying function type is *char* which is finite. Intuitively this allows to decide equality since we just have to enumerate the elements of the domain.

We develop a generic abstract specification which employs finiteness of types to provide an executable characterisation of equality and related concepts on functions. Finiteness of types is expressed using a type class:

[7] Logically type *char* is a datatype; for convenience we use here a code generator setup mapping values of type *char* on target language characters.

```
class enum =
  fixes enum :: α list
  assumes in_enum: x ∈ set enum
```

Class *enum* provides an explicit enumeration of all elements of that type. Instances for finite base types (*unit*, *bool*, *char*) are provided in a straightforward manner. Finiteness maps over product and sum types as expected:

```
instantiation ∗ :: (enum, enum) enum
begin

definition
  enum = flat (map (λx. map (Pair x) enum) enum)

instance proof
qed (auto intro: in_enum simp add: enum_prod_def)

end

instantiation + :: (enum, enum) enum
begin

definition
  enum = map Inl enum @ map Inr enum

instance proof
  fix x :: α + β
  show x ∈ set enum
    by (cases x) (simp_all add: enum_sum_def in_enum)
qed

end
```

These instances can serve as patterns how to lift finiteness over an arbitrary non-recursive datatype.

Class *enum* will enable us to provide executable injectivity test and executable equality for functions with a finite domain; as a preliminary, a universal quantifier for lists:

```
primrec every :: (α ⇒ bool) ⇒ α list ⇒ bool where
    every f [] ⟷ True
  | every f (x : xs) ⟷ f x ∧ every f xs
```

Equipped with this we can provide a code equation for universal quantification over finite types:

```
lemma all_code [code]: (∀x :: α::enum. P x) ⟷ every P enum
proof −
```

have $\bigwedge xs.\ every\ P\ xs \longleftrightarrow (\forall x \in set\ xs.\ P\ x)$
proof $-$
  fix xs
  show $every\ P\ xs \longleftrightarrow (\forall x \in set\ xs.\ P\ x)$
    by $(induct\ xs)\ simp\_all$
qed
then show $?thesis$ **by** $(auto\ intro{:}\ in\_enum)$
qed

For injectivity, the primitive definition may serve as code equation:

$inj\ f \longleftrightarrow (\forall x\ y.\ f\ x = f\ y \longrightarrow x = y)$

The last step is then to define executable equality on functions by means of extensionality:

instantiation $fun :: (type,\ eq)\ eq$
begin

definition
  $eq\_class.eq\ f\ g \longleftrightarrow (\forall x.\ f\ x = g\ x)$

instance proof
qed $(simp\_all\ add{:}\ eq\_fun\_def\ expand\_fun\_eq)$

end

Equipped with this we return to our two desired checks from above, using the following contrived definition:

definition
  $example :: ((char \Rightarrow nat) \Rightarrow bool) \times ((char \Rightarrow nat) \Rightarrow (char \Rightarrow nat) \Rightarrow bool)$
  where $example = (inj,\ \lambda e1\ e2.\ e1 = e2)$

Let us examine how the corresponding generated code looks like:

```
class Enuma a where {
  enum :: [a];
};

every :: forall a. (a -> Bool) -> [a] -> Bool;
every f [] = True;
every f (x : xs) = f x && every f xs;

alla :: forall a. (Enuma a) => (a -> Bool) -> Bool;
alla p = every p enum;

implies :: Bool -> Bool -> Bool;
implies p q = not p || q;

inj :: forall a b. (Enuma a, Eq a, Eq b) => (a -> b) -> Bool;
inj f = alla (\ x -> alla (\ y -> implies (f x == f y) (x == y)));

eq_fun :: forall a b. (Enuma a, Eq b) => (a -> b) -> (a -> b) -> Bool;
eq_fun f g = alla (\ x -> f x == g x);
```

```
instance (Enuma a, Eq b) => Eq (a -> b) where {
  a == b = eq_fun a b;
};

instance Enuma Char where {
  enum = ['\0'..'\255'];
};

example ::
  ((Char -> Nat) -> Bool, (Char -> Nat) -> (Char -> Nat) -> Bool);
example = (inj, (\ a b -> a == b));
```

The code equations for *inj* and *eq* on functions as well as the instance $eq_{\Rightarrow}$ have received an additional *enum* constraint. This has not been added by the user but by the sort inference algorithm during preprocessing; the additional constraint is induced by the code equation $All\ p \longleftrightarrow every\ p\ enum$.

4.2.2 Binary representation of natural numbers

Motivation and principle. In this section we introduce a preprocessor setup which deals with natural numbers *nat*. Natural numbers in their application typically show two different facets:

- as an inductive datatype with constructors $0 :: nat$ and $Suc :: nat \Rightarrow nat$, typically used in connection with operations on other inductive datatypes, e.g. length of lists, height of trees.
- as an algebraic and numeric type providing basic operations like $(+) :: nat \Rightarrow nat \Rightarrow nat$, $(\leq) :: nat \Rightarrow nat \Rightarrow bool$, etc.

The explicit $0/Suc$ representation grows linearly with the size of represented numbers. Considering efficiency this is unsatisfactory; therefore in computing usually logarithmic radix representations are used, typically base 2. The same approach also works in *HOL*, where a binary representation of *nat*s can be accomplished as follows:

$Dig\_zero :: nat \Rightarrow nat\ (\_/\bullet/0)$
$n\bullet 0 = n + n$

$Dig\_one :: nat \Rightarrow nat\ (\_/\bullet/1)$
$n\bullet 1 = n + n + 1$

These operations represent appending a zero or one bit respectively to a binary numeral. The annotated syntax allows for a suggestive notation of binary numerals, e.g. $1\bullet 0\bullet 1\bullet 0\bullet 1\bullet 0$ is syntax for the decimal number 42; the leading digit 1 is the conventional constant 1 for natural numbers. Equipped with this, the basic operations addition and multiplication can be expressed in a straightforward manner using the following equations:

$0 + n = n$
$n + 0 = n$
$1 + 1 = 1\bullet 0$
$1 + m\bullet 0 = m\bullet 1$

$1 + m\bullet 1 = m + 1\bullet 0$
$n\bullet 0 + 1 = n\bullet 1$
$n\bullet 0 + m\bullet 0 = n + m\bullet 0$
$n\bullet 0 + m\bullet 1 = n + m\bullet 1$
$n\bullet 1 + 1 = n + 1\bullet 0$
$n\bullet 1 + m\bullet 0 = n + m\bullet 1$
$n\bullet 1 + m\bullet 1 = n + m + 1\bullet 0$

$0 * n = 0$
$n * 0 = 0$
$1 * n = n$
$n * 1 = n$
$n\bullet 0 * m = n * m\bullet 0$
$n\bullet 1 * m\bullet 0 = n * m\bullet 0 + m\bullet 0$
$n\bullet 1 * m\bullet 1 = n * m\bullet 1 + m\bullet 1$

Binary numerals in *HOL* — Embedding the integers into natural numbers. For historical reasons, *HOL*'s binary numerals are slightly different: they carry a sign and therefore are equivalent to integers, with concrete values are built from the four constants $Int.Pls = 0$, $Int.Min = -1$, $Int.Bit0\ k = k + k$ and $Int.Bit1\ k = 1 + k + k$.[8] Numerals on *nat*s are expressed using those integer numerals and re-embedding them into the natural numbers using a conversion logically equivalent to $nat :: int \Rightarrow nat$ which maps a non-negative *int* value to its corresponding *nat* value and a negative *int* value to 0:

lemma *plus_nat_int*:
  $n + m = nat\ (int\ n + int\ m)$
  by *simp*

lemma *times_nat_int*:
  $n * m = nat\ (int\ n * int\ m)$
  unfolding *of_nat_mult* [*symmetric*] **by** *simp*

Here $int :: nat \Rightarrow int$ is the coercion from *nat*s to *int*s.

Using binary numerals for code generation. For the sake of efficiency, it is desirable to represent natural numbers in target languages in binary form. We can use the existing numeral infrastructure in *HOL* to accomplish this by choosing $nat :: int \Rightarrow nat$ as datatype constructor for *nat*. This pragmatic choice has another technical advantage: if target language integers are used for *HOL* *int*s, also *nat*s directly inherit the increased performance.

This alone however is not enough in practice: the inductive representation of *nat*s is so fundamental and occurs so often that the user would need to eliminate any $0/Suc$ pattern matching manually in order to gain an executable specification which would not break the abstraction over the representation of *nat*.

[8] This also answers the question of how *int* values are represented in generated code when *no* target language integer values are used as in §4.1.2: these four constants serve as datatype constructors.

Eliminating pattern matching on natural numbers. To cope with this, we provide a suitable preprocessor setup which eliminates $0/Suc$ pattern matching automatically in most cases.

For case distinction on *nat* a rewrite using an explicit *if* expression is sufficient:

lemma *nat_case_if* [*code*, *code unfold*]:
  $nat\_case = (\lambda f\ g\ n.\ if\ n = 0\ then\ f\ else\ g\ (n - 1))$
  by (*auto simp add*: *expand_fun_eq dest*!: *gr0_implies_Suc*)

A pair of code equations matching on $0/Suc$ must be merged to one equation using an explicit *if* expression:

lemma *Suc_if_eq*:
  assumes $f\ 0 = g$ **and** $(\bigwedge n.\ f\ (Suc\ n) = h\ n)$
  shows $f\ n = (if\ n = 0\ then\ g\ else\ h\ (n - 1))$
  using *assms* **by** (*cases n*) *simp_all*

This rule must be applied to a set of code equations repeatedly until every occurrence of $0/Suc$ pattern matching has vanished. Example:

fun *is_even* :: *nat* ⇒ *bool* **where**
  $is\_even\ 0 \longleftrightarrow True$
  $|\ is\_even\ (Suc\ 0) \longleftrightarrow False$
  $|\ is\_even\ (Suc\ (Suc\ n)) \longleftrightarrow is\_even\ n$

In this function specification the second and third equation match the premises of *Suc_if_eq* and can be merged, resulting in

$is\_even\ 0 \longleftrightarrow True$
$is\_even\ (Suc\ n) \longleftrightarrow (if\ n = 0\ then\ False\ else\ is\_even\ (n - 1))$

The next iteration merges these remaining equations:

$is\_even\ n \longleftrightarrow$
$(if\ n = 0\ then\ True$
$else\ if\ n - 1 = 0\ then\ False\ else\ is\_even\ (n - 1 - 1))$

There are examples which this merging scheme cannot cope with:

function *take* :: *nat* ⇒ *α list* ⇒ *α list* **where**
  $take\ n\ [] = []$
  $|\ take\ 0\ xs = []$
  $|\ take\ (Suc\ n)\ (x : xs) = x : take\ n\ xs$
  by *pat_completeness auto*

termination *take* **by** *lexicographic_order*

Here the 0 counterpart for the *Suc* clause is missing. In such situations it is still up to the user to provide pattern-match free code equations, which is straightforward in this case:

lemma *take_code* [*code*]:
$take\ n\ xs =$
$(if\ n = 0 \vee xs = [\![]\!]\ then\ [\![]\!]\ else\ head\ xs : take\ (n - 1)\ (tail\ xs))$
by (*cases n*, *simp_all*) (*cases xs*, *simp_all*)

Examples of such code equations however seem to occur rarely in practice; the whole machinery for implementing *nat*s in binary representation has proved to run smoothly in large-sized applications (e.g. calculations in the proof of the Kepler conjecture [43]).

4.2.3 Inductive predicates

In §2.2.2 **inductive** was introduced as a fundamental *HOL* specification tool in connection with the promise that even a certain class of inductively defined predicates shall be accessible for code generation. Here we focus on the principles to turn **inductive** specifications into executable programs; [6] gives a detailled description how this is automated.

As a running example we use a formalisation of λ-terms with de-Bruijn indices modelled by an inductive datatype:

datatype *lambda* = *Var nat* | *App lambda lambda* (**infixl** · 200) | *Abs lambda*

Application is expressed using pretty infix notation $t \cdot u$. Next index-lifting *lift* and variable substitution *subst* are specified:

primrec $lift :: nat \Rightarrow lambda \Rightarrow lambda$ **where**
$lift\ k\ (Var\ i) = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + 1))$
$|\ lift\ k\ (s \cdot t) = lift\ k\ s \cdot lift\ k\ t$
$|\ lift\ k\ (Abs\ s) = Abs\ (lift\ (k + 1)\ s)$

primrec $subst :: nat \Rightarrow lambda \Rightarrow lambda \Rightarrow lambda$ **where**
$subst\ k\ s\ (Var\ i) =$
$(if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ s\ else\ Var\ i)$
$|\ subst\ k\ s\ (t \cdot u) = subst\ k\ s\ t \cdot subst\ k\ s\ u$
$|\ subst\ k\ s\ (Abs\ t) = Abs\ (subst\ (k + 1)\ (lift\ 0\ s)\ t)$

Using this, beta-reduction is defined inductively:

inductive $beta :: lambda \Rightarrow lambda \Rightarrow bool$ (**infixl** $\rightarrow_\beta$ 50) **where**
$Abs\ s \cdot t \rightarrow_\beta subst\ 0\ t\ s$
$|\ s \rightarrow_\beta t \Longrightarrow s \cdot u \rightarrow_\beta t \cdot u$
$|\ s \rightarrow_\beta t \Longrightarrow u \cdot s \rightarrow_\beta u \cdot t$
$|\ s \rightarrow_\beta t \Longrightarrow Abs\ s \rightarrow_\beta Abs\ t$

Intuitively, we would expect beta-reduction ($\rightarrow_\beta$) to be executable. To be more precise, inductive predicates describe *enumerations* of possible arguments such that the predicate expression evaluates to *True*. The key technique to distill such enumerations from a given inductive specification are *mode assignments*: an analysis of dataflow tells us which arguments are at least required for a predicate in order that the remaining arguments can be computed using the underlying inductive specification [54]. In the ($\rightarrow_\beta$) example, one possible mode classifies the first argument as *input* and the second one as *output*, thus enumerating all normal forms of a given input term.

How are these enumerations expressed inside the logic? The central idea is to provide a dedicated type to represent enumerations isomorphic to sets

$$\textbf{datatype}\ \alpha\ \mathit{pred} = \mathit{Pred}\ (\alpha \Rightarrow \mathit{bool})$$

together with a complementary projection

$$\begin{array}{l} \mathit{eval} :: \alpha\ \mathit{pred} \Rightarrow \alpha \Rightarrow \mathit{bool} \\ \mathit{eval}\ (\mathit{Pred}\ f) = f \end{array}$$

This allows to characterise a particular mode assignment of a predicate to be expressed as an enumeration, in our example:

$$\begin{array}{l} \mathit{beta}_{io} :: \mathit{lambda} \Rightarrow \mathit{lambda}\ \mathit{pred} \\ \mathit{beta}_{io}\ t = \mathit{Pred}\ (\lambda u.\ t \rightarrow_\beta u) \end{array}$$

Enumerations form a plus monad with the following basic operations:

$\bot :: \alpha\ \mathit{pred}$ is the empty enumeration:
$\bot = \mathit{Pred}\ (\lambda x.\ \mathit{False})$

$\mathit{single} :: \alpha \Rightarrow \alpha\ \mathit{pred}$ is the singleton enumeration:
$\mathit{single}\ x = \mathit{Pred}\ (\lambda y.\ y = x)$

$(\gg\!=) :: \alpha\ \mathit{pred} \Rightarrow (\alpha \Rightarrow \beta\ \mathit{pred}) \Rightarrow \beta\ \mathit{pred}$ takes a function which returns an enumeration applies it to every element of an enumeration and flattens the resulting enumerations:
$P \gg\!= f = \mathit{Pred}\ (\lambda x.\ \exists y.\ \mathit{eval}\ P\ y \wedge \mathit{eval}\ (f\ y)\ x)$

$(\sqcup) :: \alpha\ \mathit{pred} \Rightarrow \alpha\ \mathit{pred} \Rightarrow \alpha\ \mathit{pred}$ forms the union of two enumerations:
$P \sqcup Q = \mathit{Pred}\ (\mathit{eval}\ P \sqcup \mathit{eval}\ Q)$

To illustrate how enumerations are constructed using these operations and the introduction rules of a predicate, we give here the equation constructed for beta_{io}:

$$\begin{array}{l}
\mathit{beta}_{io}\ t = \\
\mathit{single} \\
\quad t \gg\!= (\lambda x.\ \mathit{case}\ x\ \mathit{of}\ \mathit{Abs}\ s \cdot t \Rightarrow \mathit{single}\ (\mathit{subst}\ 0\ t\ s) \mid _ \cdot t \Rightarrow \bot \\
\qquad\qquad \mid _ \Rightarrow \bot) \sqcup \mathit{single} \\
\qquad\qquad\qquad\qquad t \gg\!= (\lambda x.\ \mathit{case}\ x\ \mathit{of} \\
\qquad\qquad\qquad\qquad\qquad s \cdot u \Rightarrow \mathit{beta}_{io}\ s \gg\!= (\lambda x.\ \mathit{single}\ (x \cdot u)) \\
\qquad\qquad\qquad\qquad\qquad \mid _ \Rightarrow \bot) \sqcup \mathit{single} \\
\qquad t \gg\!= (\lambda x.\ \mathit{case}\ x\ \mathit{of}\ u \cdot s \Rightarrow \mathit{beta}_{io}\ s \gg\!= (\lambda x.\ \mathit{single}\ (u \cdot x)) \\
\qquad\qquad\quad \mid _ \Rightarrow \bot) \sqcup \mathit{single} \\
\qquad\qquad\qquad\qquad\quad t \gg\!= (\lambda x. \\
\mathit{case}\ x\ \mathit{of}\ \mathit{Abs}\ s \Rightarrow \mathit{beta}_{io}\ s \gg\!= (\lambda x.\ \mathit{single}\ (\mathit{Abs}\ x)) \mid _ \Rightarrow \bot)
\end{array}$$

It is not our main focus here to explain how this equation is proved using the definition of $beta_{io}$ and the basic enumeration operations; instead we concentrate on the question of how to make the enumerations executable. Unsurprisingly, the key to a solution is to choose an appropriate set of datatype constructors. As a prerequisite, here is an auxiliary type:

$$\begin{array}{l}\textbf{datatype}\ \alpha\ seq = seq.Empty \mid seq.Insert\ \alpha\ (\alpha\ pred) \\ \quad \mid seq.Join\ (\alpha\ pred)\ (\alpha\ seq)\end{array}$$

Values of type $\alpha\ seq$ are embedded into type $\alpha\ pred$ by defining:

$$\begin{array}{l}pred\_of\_seq :: \alpha\ seq \Rightarrow \alpha\ pred \\ pred\_of\_seq\ seq.Empty = \bot \\ pred\_of\_seq\ (seq.Insert\ x\ P) = single\ x \sqcup P \\ pred\_of\_seq\ (seq.Join\ P\ xq) = P \sqcup pred\_of\_seq\ xq\end{array}$$

This we use to provide a constant *Seq* which will serve as datatype constructor for type $\alpha\ pred$.

$$\begin{array}{l}Seq :: (unit \Rightarrow \alpha\ seq) \Rightarrow \alpha\ pred \\ Seq\ f = pred\_of\_seq\ (f\ ())\end{array}$$

Two further auxiliary constants mediate between $\alpha\ pred$ and $\alpha\ seq$:

$$\begin{array}{l}apply :: (\alpha \Rightarrow \beta\ pred) \Rightarrow \alpha\ seq \Rightarrow \beta\ seq \\ apply\ f\ seq.Empty = seq.Empty \\ apply\ f\ (seq.Insert\ x\ P) = seq.Join\ (f\ x)\ (seq.Join\ (P \ggg f)\ seq.Empty) \\ apply\ f\ (seq.Join\ P\ xq) = seq.Join\ (P \ggg f)\ (apply\ f\ xq)\end{array}$$

$$\begin{array}{l}adjunct :: \alpha\ pred \Rightarrow \alpha\ seq \Rightarrow \alpha\ seq \\ adjunct\ P\ seq.Empty = seq.Join\ P\ seq.Empty \\ adjunct\ P\ (seq.Insert\ x\ Q) = seq.Insert\ x\ (Q \sqcup P) \\ adjunct\ P\ (seq.Join\ Q\ xq) = seq.Join\ Q\ (adjunct\ P\ xq)\end{array}$$

On top of this, we prove the following code equations for our $\alpha\ pred$ operations:

$$\bot = Seq\ (\lambda u.\ seq.Empty)$$

$$single\ x = Seq\ (\lambda u.\ seq.Insert\ x\ \bot)$$

$$Seq\ g \ggg f = Seq\ (\lambda u.\ apply\ f\ (g\ ()))$$

$$\begin{array}{l}Seq\ f \sqcup Seq\ g = \\ Seq\ (\lambda u.\ case\ f\ ()\ of\ seq.Empty \Rightarrow g\ () \\ \qquad\qquad \mid seq.Insert\ x\ P \Rightarrow seq.Insert\ x\ (P \sqcup Seq\ g) \\ \qquad\qquad \mid seq.Join\ P\ xq \Rightarrow adjunct\ (Seq\ g)\ (seq.Join\ P\ xq))\end{array}$$

We give the corresponding *SML* code in full:

```
structure Example =
struct

datatype 'a seq = Empty | Insert of 'a * 'a pred |
  Join of 'a pred * 'a seq
and 'a pred = Seq of (unit -> 'a seq);
```

```
fun bind (Seq g) f = Seq (fn u => apply f (g ()))
and apply f Empty = Empty
  | apply f (Insert (x, p)) = Join (f x, Join (bind p f, Empty))
  | apply f (Join (p, xq)) = Join (bind p f, apply f xq);

val bot_pred : 'a pred = Seq (fn u => Empty)

fun single x = Seq (fn u => Insert (x, bot_pred));

fun seq_case f1 f2 f3 (Join (pred, seq)) = f3 pred seq
  | seq_case f1 f2 f3 (Insert (a, pred)) = f2 a pred
  | seq_case f1 f2 f3 Empty = f1;

fun adjunct p Empty = Join (p, Empty)
  | adjunct p (Insert (x, q)) = Insert (x, sup_pred q p)
  | adjunct p (Join (q, xq)) = Join (q, adjunct p xq)
and sup_pred (Seq f) (Seq g) =
  Seq (fn u =>
        (case f () of Empty => g ()
           | Insert (x, p) => Insert (x, sup_pred p (Seq g))
           | Join (p, xq) => adjunct (Seq g) (Join (p, xq))));

end; (*struct Example*)
```

In shape this follows a well-known ML technique for lazy lists: each inspection of a lazy list by means of an application `f ()` is protected by a constructor `Seq`. Thus we enforce a lazy evaluation strategy for predicate enumerations even for eager languages.[9]

Enumerations also demonstrate that datatypes in target languages need not satisfy the usual logical characteristics of inductive *HOL* **datatype**s (cf. §3.2.4): the generated types α *pred* and α *seq* describe potentially infinite data structures; a literal construction of both types by means of **datatype** would only allow finite data structures.

4.3 Mastering destructive data structures

In this section we sketch an adaptation of the code generator to interact with destructive data structures (references, arrays) in *SML*, *OCaml* and *Haskell*.

Typically, functional programming languages distinguish *pure* and *impure* expressions: *pure* expressions are plain λ-terms, whereas *impure* expressions may issue *side effects* on an underlying state.

4.3.1 Side effects, linear type systems and state monads

The logic *HOL* is pure. How to model impure expressions then? One reasoning device for side effects is a *denotational semantics* [64] where side effects are modelled explicitly as transformations of an underlying state σ:

$$\sigma \Rightarrow \alpha \times \sigma$$

The choice of σ depends on which kind of side effects are to be modelled; for our purpose we will consider σ as a structure representing a *heap*, a mapping from addresses to (typed) data.

[9] See §C.5 for the corresponding *SML* code in full.

An explicit state σ allows to describe impure effects in a pure setting. This, however, results in a discrepancy to the "real world": nothing prevents to write down a term $\lambda s{::}\sigma.\ (s,\ s)$ which *forks* a state — this is not what we expect from a real program where the state is threaded through, i.e. each particular state value is used exactly once since after an update the former state value has been destroyed (single-threadedness).

To escape this deficiency, linear type system have been proposed [59]. The idea is to define a type system which allows to encode notions like "this value is used exactly once" (*linear typing*). Thus if state values of type σ can be typed linear, they are used single-threadedly by construction.

The *ACL2* logic mentioned in §1.3.2 uses this approach: it incorporates a simplistic linearity check which allows to implement certain values destructively, which is useful to implement highly efficient simulators [10].

We will follow a different path: single-threadedness can also be accomplished using a *state monad*, where the state is lifted into a higher-order datatype:

$$\mathit{datatype}\ \alpha\ \mathit{Heap} = \mathit{Heap}\ (\sigma \Rightarrow \alpha \times \sigma)$$

Thus a value of type α *Heap* represents a *computation* which may affect the underlying state and returns a *result* of type α. To access the encapsulated state transformations, two combinators $(\gg\!=)$ and *return* are provided:

$$(\gg\!=) :: \alpha\ \mathit{Heap} \Rightarrow (\alpha \Rightarrow \beta\ \mathit{Heap}) \Rightarrow \beta\ \mathit{Heap}$$
$$\mathit{Heap}\ f \gg\!= g = \mathit{Heap}\ (\lambda s.\ \mathit{let}\ (x,\ s') = f\ s\ \mathit{in}\ \mathit{case}\ g\ x\ \mathit{of}\ \mathit{Heap}\ h \Rightarrow h\ s')$$

$$\mathsf{return} :: \alpha \Rightarrow \alpha\ \mathit{Heap}$$
$$\mathsf{return}\ x = \mathit{Heap}\ (\lambda s.\ (x,\ s))$$

$(\gg\!=)$ composes two computations such that the result of the first is the argument to the second and the state is threaded through both, resulting in a new computation; *return* lifts a pure value to a computation which has no effect on the underlying state. α *Heap* is an abstract type, there are no other means to access σ.

A monadic program then consists of an entry point $\mathit{main} :: \alpha\ \mathit{Heap}$ which may contain impure parts composed by $(\gg\!=)$ and pure parts lifted by *return*. Single-threadedness guarantees that the underlying state can be implemented destructively.

The monadic approach to destructive data structures is canonical in Haskell [30], where the heap monad type is `ST s a`; hereby `a` corresponds to α from above, the role of `s` will be clarified below. In what follows we describe how the key techniques of the monadic approach are transferred to *HOL*; for details see [12].

4.3.2 A polymorphic heap in *HOL*

A crucial question is how to model a polymorphic heap in *HOL*. The technique we present here uses type classes to encode values of countable types into a model of an unbounded memory of natural numbers. Here is a type class of countable types with explicit mappings from and to the natural numbers:

class *countable* =
  fixes $\mathit{nat\_of} :: \alpha \Rightarrow \mathit{nat}$
  fixes $\mathit{of\_nat} :: \mathit{nat} \Rightarrow \alpha$
  assumes *nat_of_inject*: $\mathit{nat\_of}\ x = \mathit{nat\_of}\ y \Longrightarrow x = y$
  assumes *of_nat_of*: $\mathit{of\_nat}\ (\mathit{nat\_of}\ x) = x$

For arbitrary first-order datatypes over countable types, canonical instances for class *countable* can be provided (see §D for details). It is important to note that these encodings only have a logical purpose – they are *not* used for execution.

A heap suitable for first-order types can thus be encoded as

datatype $heap = Heap\ (nat \Rightarrow nat\ list)\ nat$

where the first field is mapping from addresses to encoded values and the second field describes the lowest "unallocated" memory position.

Then arrays are just functions from references to addresses:

datatype $\alpha\ array = Array\ nat$

Though addresses themselves are untyped, the array type carries its type parameter as a phantom type, which enables us to combine the primitive untyped world of *heap* and *array* with the typed surrounding world by three fundamental operations: *alloc* allocates an array using an initial value, *peek* reads from an array and *poke* changes an array.

primrec $alloc :: \alpha{::}countable\ list \Rightarrow heap \Rightarrow \alpha\ array \times heap$ **where**
$alloc\ xs\ (Heap\ memory\ limit) =$
$(Array\ limit,\ Heap\ (point\ limit\ (\lambda ms.\ map\ nat\_of\ xs)\ memory)\ (Suc\ limit))$

fun $peek :: heap \Rightarrow \alpha\ array \Rightarrow \alpha{::}countable\ list$ **where**
$peek\ (Heap\ memory\ limit)\ (Array\ addr) = map\ of\_nat\ (memory\ addr)$

fun $poke :: \alpha\ array \Rightarrow (\alpha{::}countable\ list \Rightarrow \alpha\ list) \Rightarrow heap \Rightarrow heap$ **where**
$poke\ (Array\ addr)\ f\ (Heap\ memory\ limit)$
$= Heap\ (point\ addr\ (map\ nat\_of \circ f \circ map\ of\_nat)\ memory)\ limit$

Mediation between typed values (α) and their untyped encodings (nat) happens through the operations $nat\_of :: \alpha \Rightarrow nat$ and $of\_nat :: nat \Rightarrow \alpha$ of the *countable* class, where the type α is determined by the phantom type parameter of the corresponding *array* reference. Logically, a value of type α *array* relative to a *heap* corresponds to a list of type α *list*, the *array value*.

The injectivity of the underlying *countable* mapping allows to establish basic properties of those operations, e.g.

$$\bigwedge heap\ heap'\ xs\ a.\ (a,\ heap') = alloc\ xs\ heap \Longrightarrow peek\ heap'\ a = xs$$

For simplicity we restrict the presentation here to arrays; references can be handled as arrays of length one.

4.3.3 Putting the heap into a monad

The next step is to wrap up the *heap* into a monad:

datatype $\alpha\ Heap = churning\ (heap \Rightarrow \alpha \times heap)$

For simplified usage of *Heap* operations various combinators are provided:

primrec $execute :: \alpha\ Heap \Rightarrow heap \Rightarrow \alpha \times heap$ **where**
$execute\ (churning\ f) = f$

definition $peeking :: (heap \Rightarrow \alpha) \Rightarrow \alpha\ Heap$ **where**
$peeking\ f = churning\ (\lambda heap.\ (f\ heap,\ heap))$

definition $poking :: (heap \Rightarrow heap) \Rightarrow unit\ Heap$ **where**
$poking\ f = churning\ (\lambda heap.\ ((),\ f\ heap))$

Thus we have three combinators to lift operations on bare heaps into the monad type $\alpha\ Heap$

$peeking :: (heap \Rightarrow \alpha) \Rightarrow \alpha\ Heap$ for read access
$poking :: (heap \Rightarrow heap) \Rightarrow unit\ Heap$ for write access
$churning :: (heap \Rightarrow \alpha \times heap) \Rightarrow \alpha\ Heap$ for combined read/write access

and one unpacking operation

$execute :: \alpha\ Heap \Rightarrow heap \Rightarrow \alpha \times heap$

These primitives allow us to define the basic monad combinators *return* and ($\gg=$):

definition $return :: \alpha \Rightarrow \alpha\ Heap$ **where**
$return\ x = peeking\ (\lambda\_.\ x)$

definition $bind :: \alpha\ Heap \Rightarrow (\alpha \Rightarrow \beta\ Heap) \Rightarrow \beta\ Heap$ (**infixl** $\gg=$ 54) **where**
$f \gg= g = churning\ (\lambda heap.\ let\ (x,\ heap') = execute\ f\ heap$
$in\ execute\ (g\ x)\ heap')$

On top of this we provide fundamental array operations:

definition $array :: \alpha{::}countable\ list \Rightarrow \alpha\ array\ Heap$ **where**
$array\ xs = churning\ (alloc\ xs)$ — array allocation

definition $index\_error :: nat \Rightarrow nat \Rightarrow \alpha$ **where**
$index\_error\ i\ n = undefined$

definition $nth :: \alpha{::}countable\ array \Rightarrow nat \Rightarrow \alpha\ Heap$ **where**
$nth\ a\ i = peeking\ (\lambda heap.\ let\ xs = peek\ heap\ a;$
$n = length\ xs$
$in\ if\ i < n\ then\ xs\ !\ i\ else\ index\_error\ i\ n)$ — array read access

definition $upd :: nat \Rightarrow (\alpha{::}countable \Rightarrow \alpha) \Rightarrow \alpha\ array \Rightarrow unit\ Heap$ **where**
$upd\ i\ f\ a = poking\ (poke\ a\ (\lambda xs.\ let\ n = length\ xs$
$in\ if\ i < n\ then\ map\_nth\ i\ f\ xs\ else\ index\_error\ i\ n))$ — array write access

definition $len :: \alpha{::}countable\ array \Rightarrow nat\ Heap$ **where**
$len\ a = peeking\ (\lambda heap.\ length\ (peek\ heap\ a))$ — array length determination

The operations *upd* and *len* involve a subtlety: if the index exceeds the length of the underlying array, a dedicated constant *index_error* is used. This is just to be understood as a symbolic marker — it well never show up in generated code (see below).

An alternative for *upd* would be to ignore updates on non-existing positions entirely, but we want to model the primitive array operations as closely to operations on arrays in target languages as possible.

To actually reason about monadic programs involving such *array* operations, additional reasoning infrastructure is needed, which is presented thoroughly in [12].

4.3.4 Interfacing with destructive code

We aim now to identify the *array*, *nth*, *upd* and *len* with corresponding array operations in *Haskell*. Before we attempt this, we have to overcome a fundamental discrepancy: in the logic modelling we use (naturally) natural numbers of type *nat* for indexing arrays, whereas for *Haskell* a built-in numeral type shall be used. How to mediate between these two? The solution is to provide a *HOL* type *index* which is logically isomorphic to *nat* but is mapped to target language built-in integers by convention. Type *index* is also an instance of *countable*, so we can just use *nat_of* and *of_nat* to coerce.

This allows us to introduce variant operations for *nth*, *upd* and *len* that use *index* instead of *nat*; we also introduce a further operation corresponding to *array* since the *Haskell* array operation we want to use later has a slightly different type signature.

primrec $array_i :: index \times index \Rightarrow \alpha{::}countable\ list \Rightarrow \alpha\ array\ Heap$ **where**
$array_i\ (k,\ l)\ xs = array\ ((take\ (nat\_of\ l - nat\_of\ k) \circ drop\ (nat\_of\ k))\ xs)$

lemma *array_code* [*code*]:
$array\ xs = array_i\ (0,\ of\_nat\ (length\ xs))\ xs$
by *simp*

definition $nth_i :: \alpha{::}countable\ array \Rightarrow index \Rightarrow \alpha\ Heap$ **where**
$nth_i\ a\ k = nth\ a\ (nat\_of\ k)$

lemma *nth_code* [*code*]:
$nth\ a\ n = nth_i\ a\ (of\_nat\ n)$
unfolding nth_i_def **by** *simp*

definition $upd_i :: \alpha\ array \Rightarrow index \Rightarrow \alpha{::}countable \Rightarrow unit\ Heap$ **where**
$upd_i\ a\ k\ x = upd\ (nat\_of\ k)\ (\lambda\_.\ x)\ a$

lemma *upd_code* [*code*]:
$upd\ n\ f\ a = (let\ k = of\_nat\ n\ in$
$nth_i\ a\ k \gg\!= (\lambda x.\ upd_i\ a\ k\ (f\ x)))$
unfolding $nth_i\_def\ upd_i\_def$ **by** (*simp add*: *Let_def*)

definition $len_i :: \alpha{::}countable\ array \Rightarrow index\ Heap$ **where**
$len_i\ a = len\ a \gg\!= (\lambda n.\ return\ (of\_nat\ n))$

lemma *len_code* [*code*]:
$len\ a = len_i\ a \gg\!= (\lambda k.\ return\ (nat\_of\ k))$

unfolding len_i_def **by** *simp*

A suitable adaptation setup (cf. §3.4.1) maps the pseudo-destructive primitives onto corresponding *Haskell* counterparts as follows:

HOL	*Haskell*
α *Heap*	`Control.Monad.ST.ST Control.Monad.ST.RealWorld` α
α *array*	`Data.Array.ST.STArray Control.Monad.ST.RealWorld` α
$f \ggg g$	f `>>=` g
return x	`return` x
$array_i\ (k,\ l)\ xs$	`Data.Array.ST.newListArray (`k`, `l`)` xs
$len_i\ a$	`Control.Monad.liftM snd (Data.Array.ST.getBounds` a`)`
$nth_i\ a\ n$	`Data.Array.ST.readArray` a n
$upd_i\ a\ n\ x$	`Data.Array.ST.writeArray` a n x

The *index_error* constant in the definition of *nth* and *upd* does not occur in the code equations for *nth* and *upd* at all — its only purpose is to pragmatically guarantee that *nth* and *upd* behave "underspecified" for index out of bounds. Otherwise, nth_i and upd_i would yield "reasonable" results while `Data.Array.ST.readArray` and `Data.Array.ST.writeArray` would break with an exception. Formally, this behaviour would be correct since we only guarantee partial correctness for generated code. However it would be counter-intuitive.

The explicit conversions between *nat* values and *index* values are formally correct but are inefficient. Fortunately, everything needed to avoid these conversions has already been presented in §4.2.2: implementing natural numbers by target language integers. The only thing which has to be added is the mapping of the conversions $of\_nat :: nat \Rightarrow index$ and $nat\_of :: index \Rightarrow nat$ to identity; thus there occurs no conversion between values at all at runtime.

Something has to be said about the `s` argument in the `Control.Monad.ST s a` type. Its purpose is to permit computations involving destructive data structures to be embedded in pure ones by means of a combinator

```
runST :: (forall s. Control.Monad.ST s a) -> a
```

Its type is of higher-rank polymorphism and in conventional notation would be written as $(\forall\, \sigma.\ ST\ \sigma\ \alpha) \Rightarrow \alpha$. The bound σ is a phantom type whose purpose is to prevent any value depending on the heap from "escaping" the `Control.Monad.ST` monad: each value depending on the heap has some σ in its type, which prevents that α contains σ since it is locally bound. We do not attempt to transfer this to *HOL*: in the mapping above `s` is instantiated to be the fixed type value `Control.Monad.ST.RealWorld`.

Let us conclude with a short evaluation. We have presented a lightweight approach to model destructive data structures in *HOL*, where we use only existing infrastructure: logical specifications and a simple adaptation of the *Haskell* serialiser. What remains unsatisfactory is that the identification of *HOL* operations and *Haskell* operations is only based on intuition; a more ambitious justification would have to set an explicit model for *Haskell* arrays in relation to the existing *HOL* model.

4.4 A quickcheck implementation in *Isar*

Quickcheck [17] is a *Haskell* library which allows a developer to specify simple propositions about functions and to implement generators for random values of particular datatypes. Both in combination then are applied to search for counterexamples, values for which the functions violate the specified propositions. This allows to discover erroneous function implementations quite quickly and thus *quickcheck* has become a standard tool for program development in *Haskell*.

For interactive theorem proving, unsuccessful attempts to prove theorems about erroneous specifications are even more annoying and cost time. One approach is to search for finite counter models using SAT-solving [61]. Another possibility is *quickcheck*, which has been adopted successfully to *HOL* using the previously existing code generator tailored towards *SML* [55].

What makes *quickcheck* so attractive is its simplicity: it builds directly on the bare bones of the underlying *Haskell* language without much additional infrastructure. In this section we will show how to transfer this principle to *HOL*, using as much existing infrastructure as possible. Beside the practical benefit, our *quickcheck* implementation also illustrates the relevance of overloading and type classes, and is an elegant example how *HOL* can be used as a programming language.

4.4.1 Evaluation and reconstruction

One prerequisite for *quickcheck* is the *evaluation* of a given term t in the logic:

1. t is compiled to its corresponding term t' in the system language *SML*.

2. t' is evaluated to its normal form u' in the system runtime.

3. This result u' is then reconstructed into its corresponding term u in the logic again.

In this situation we employ Definition 13 from where follows that it is admissible to assume $t \equiv u$. The underlying system of equations E contains all code equations referring to constants in t and their transitive dependencies.

The open question is how to reconstruct a term u' in the system language back to a term in the logic u. Technically speaking, an *SML* value u' of an *SML* type τ must be translated into an *SML* value u'' of *SML* type `term`, where `term` implements the term representation of *HOL* such that its logical interpretation is u. Fortunately this can be easily accomplished using existing *HOL* facilities, roughly as follows:

- Provide a logical type *term* which corresponds to the *SML* implementation of term representations.

- Provide a type class *term_of* with a class parameter $\mathit{term\_of} :: \alpha \Rightarrow \mathit{term}$.

- Provide suitable instances of class *term_of* for all types contained in an evaluated term.

- To evaluate and reconstruct a term t, just evaluate $\mathit{term\_of}\ t$.

Before we come to the technical details, there is a further prerequisite: generating code referring to internal representation of terms inadvertently contains string literals, e.g. for constant names. To achieve this, the *HOL string* type could be used and mapped to *SML* target language strings. But this would impose the decision to translate *HOL string*s to *SML* strings for *every* occurrence of *string*s. Instead, we provide a dedicated type *message_string* which logically is a copy of *string* but is always mapped to *SML* strings, thus not interfering with *string* at all.

Since the term representation *term* necessarily also involves type representations, we start with a datatype representing (monomorphic) types:

$$\textbf{datatype}\ \mathit{type} = \mathit{Tyco}\ \mathit{message\_string}\ (\mathit{type}\ \mathit{list})$$

The convention is that

$$\begin{array}{l}\text{concrete term } \mathit{Tyco}\ \kappa\ [\![\tau_1, \ldots, \tau_k]\!] \\ \quad \text{represents abstract type } \kappa\ \tau_1 \cdots \tau_k\end{array}$$

Concerning term representation, there is a fundamental restriction: reconstruction of an *SML* term u' of type τ requires its representation to be inspectible on the *SML* level. For datatypes this is possible using pattern matching, given that the types appearing in the corresponding constructors are themselves inspectible. So u' may not contain any function type. Due to the evaluation semantics of *SML*, u' then is a closed term containing only constructors fully applied to arguments. This enables us to keep the term representation to the essential minimum:

$$\textbf{datatype}\ \mathit{term} = \mathit{Const}\ \mathit{message\_string}\ (\mathit{type}\ \mathit{list})\ (\mathit{term}\ \mathit{list})$$

where

$$\begin{array}{l}\text{concrete term } \mathit{Const}\ f\ [\![\tau_1, \ldots, \tau_k]\!]\ [\![t_1, \ldots, t_n]\!] \\ \quad \text{represents abstract term } f\ [\tau_1] \cdots [\tau_k]\ t_1 \cdots t_n\end{array}$$

Concrete values of types *type* and *term* can easily be re-transferred to the internal *SML* representation of types and terms in *Isabelle*. Logically they are constructed using an appropriate class specification:

$$\begin{array}{l}\textbf{class}\ \mathit{term\_of} = \\ \quad \textbf{fixes}\ \mathit{type\_of} :: \alpha\ \mathit{itself} \Rightarrow \mathit{type} \\ \quad\quad \textbf{and}\ \mathit{term\_of} :: \alpha \Rightarrow \mathit{term}\end{array}$$

The α *itself* phantom type has been introduced in §2.3.2. Given a datatype $\kappa\ \alpha_1 \ldots \alpha_k$ with constructors $f_1, \ldots, f_l$, the construction of instances for *term_of* is canonical and happens automatically:

$$\begin{array}{l}
\mathit{type_of}\ [\kappa\ \alpha_1 \ldots \alpha_k]\ (\mathit{TYPE}\ \kappa\ \alpha_1 \ldots \alpha_k) \\
\quad = \mathit{Tyco}\ \text{"}\kappa\text{"}\ [\![\mathit{type_of}\ [\alpha_1]\ (\mathit{TYPE}\ \alpha_1), \ldots, \mathit{type_of}\ [\alpha_k]\ (\mathit{TYPE}\ \alpha_k)]\!] \\[1ex]
\mathit{term_of}\ (f_1\ [\tau_1] \ldots [\tau_{m1}]\ x_1 \ldots x_{n1}) \\
\quad = \mathit{Const}\ \text{"}f_1\text{"}\ [\![\mathit{type_of}\ [\tau_1]\ (\mathit{TYPE}\ \tau_1), \ldots, \mathit{type_of}\ [\tau_{m1}]\ (\mathit{TYPE}\ \tau_{m1})]\!] \\
\quad\quad [\![\mathit{term_of}\ x_1, \ldots, \mathit{term_of}\ x_{n1}]\!] \\
\ldots \\
\mathit{term_of}\ (f_l\ [\tau_1] \ldots [\tau_{ml}]\ x_1 \ldots x_{nl}) \\
\quad = \mathit{Const}\ \text{"}f_l\text{"}\ [\![\mathit{type_of}\ [\tau_1]\ (\mathit{TYPE}\ \tau_1), \ldots, \mathit{type_of}\ [\tau_{ml}]\ (\mathit{TYPE}\ \tau_{ml})]\!] \\
\quad\quad [\![\mathit{term_of}\ x_1, \ldots, \mathit{term_of}\ x_{nl}]\!]
\end{array}$$

As example we present here the corresponding instances for the binary trees from §4.1.3:

$$
\begin{aligned}
&\mathit{type\_of}\ [\mathit{tree}\ \alpha{::}\mathit{term\_of}\ \beta{::}\mathit{term\_of}]\ (\mathit{TYPE}\ (\alpha,\ \beta)\ \mathit{tree}) = \\
&\mathit{Tyco}\ \text{"}\mathit{tree}\text{"} \\
&\ \llbracket \mathit{type\_of}\ [\alpha{::}\mathit{term\_of}]\ (\mathit{TYPE}\ \alpha),\ \mathit{type\_of}\ [\beta{::}\mathit{term\_of}]\ (\mathit{TYPE}\ \beta)\rrbracket
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{term\_of}\ [\mathit{tree}\ \alpha{::}\mathit{term\_of}\ \beta{::}\mathit{term\_of}]\ \mathit{Empty} = \\
&\mathit{Const}\ \text{"}\mathit{Empty}\text{"} \\
&\ \llbracket \mathit{type\_of}\ [\alpha{::}\mathit{term\_of}]\ (\mathit{TYPE}\ \alpha),\ \mathit{type\_of}\ [\beta{::}\mathit{term\_of}]\ (\mathit{TYPE}\ \beta)\rrbracket\ \llbracket\rrbracket \\
&\mathit{term\_of}\ [\mathit{tree}\ \alpha{::}\mathit{term\_of}\ \beta{::}\mathit{term\_of}]\ (\mathit{Branch}\ v\ k\ l\ r) = \\
&\mathit{Const}\ \text{"}\mathit{Branch}\text{"} \\
&\ \llbracket \mathit{type\_of}\ [\alpha{::}\mathit{term\_of}]\ (\mathit{TYPE}\ \alpha),\ \mathit{type\_of}\ [\beta{::}\mathit{term\_of}]\ (\mathit{TYPE}\ \beta)\rrbracket \\
&\ \llbracket \mathit{term\_of}\ [\beta{::}\mathit{term\_of}]\ v,\ \mathit{term\_of}\ [\alpha{::}\mathit{term\_of}]\ k, \\
&\quad \mathit{term\_of}\ [\mathit{tree}\ \alpha{::}\mathit{term\_of}\ \beta{::}\mathit{term\_of}]\ l, \\
&\quad \mathit{term\_of}\ [\mathit{tree}\ \alpha{::}\mathit{term\_of}\ \beta{::}\mathit{term\_of}]\ r\rrbracket
\end{aligned}
$$

Then evaluation of a term $t :: \tau$ proceeds by generating code for $\mathit{term\_of}\ t :: \mathit{term}$, resulting in an *SML* term $u' :: \mathit{term}$, whose re-translation into the internal *SML* representation of terms is straightforward.

4.4.2 A random engine in *HOL*

To obtain random values, we implement a simple random engine following [34].

Random values are generated from random seeds, pairs of natural numbers. We use the *index* type introduced in §4.3.4 to represent the natural number values. The reason is that we want to map random seeds to target language numerals; logically, *index* is a plain copy of *nat*. The fundamental operation is *next* which computes a "random" value of type *index* from a random seed which is updated in the course of the computation:

$$
\begin{aligned}
&\textbf{types}\ \mathit{seed} = \mathit{index} \times \mathit{index} \\[1ex]
&\textbf{definition}\ \mathit{minus\_shift} :: \mathit{index} \Rightarrow \mathit{index} \Rightarrow \mathit{index} \Rightarrow \mathit{index}\ \textbf{where} \\
&\ \ \mathit{minus\_shift}\ r\ k\ l = (\mathit{if}\ k < l\ \mathit{then}\ r + k - l\ \mathit{else}\ k - l) \\[1ex]
&\textbf{primrec}\ \mathit{next} :: \mathit{seed} \Rightarrow \mathit{index} \times \mathit{seed}\ \textbf{where} \\
&\ \ \mathit{next}\ (v,\ w) = (\mathit{let} \\
&\quad\ k = \ v\ \mathit{div}\ 53668; \\
&\quad\ v' = \mathit{minus\_shift}\ 2147483563\ (40014 * (v\ \mathit{mod}\ 53668))\ (k * 12211); \\
&\quad\ l = \ w\ \mathit{div}\ 52774; \\
&\quad\ w' = \mathit{minus\_shift}\ 2147483399\ (40692 * (w\ \mathit{mod}\ 52774))\ (l * 3791); \\
&\quad\ z = \ \mathit{minus\_shift}\ 2147483562\ v'\ (w' + 1) + 1 \\
&\quad \mathit{in}\ (z,\ (v',\ w')))
\end{aligned}
$$

Typically, computations involving random seeds have the type signature $\overline{\tau} \Rightarrow \mathit{seed} \Rightarrow \tau' \times \mathit{seed}$. To compose such computations we will use two infix combinators:

$$
\begin{aligned}
&(\circ\!\!>) :: (\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \gamma) \Rightarrow \alpha \Rightarrow \gamma \\
&f\ \circ\!\!>\ g = (\lambda s.\ g\ (f\ s))
\end{aligned}
$$

$$
\begin{aligned}
&(\circ\!\!\gg) :: (\alpha \Rightarrow \beta \times \gamma) \Rightarrow (\beta \Rightarrow \gamma \Rightarrow \delta) \Rightarrow \alpha \Rightarrow \delta \\
&f\ \circ\!\!\gg\ g = (\lambda s.\ \mathit{let}\ (x,\ s') = f\ s\ \mathit{in}\ g\ x\ s')
\end{aligned}
$$

These combinators form an "open" state monad where the state is not wrapped up in a type constructor (cf. §4.3.1).

The core function which all complex computations depending on random values will use is *range* which computes a "random" value of type *index* within a given range $[0 \ldots k[$:

fun *log* :: *index* ⇒ *index* ⇒ *index* **where**
  log b i = (*if b* ≤ 1 ∨ *i* < *b then* 1 *else* 1 + *log b* (*i div b*))

fun *iterate* :: *index* ⇒ (β ⇒ α ⇒ β × α) ⇒ β ⇒ α ⇒ β × α **where**
  iterate k f x = (*if k* = 0 *then Pair x else f x* ∘≫ *iterate* (*k* − 1) *f*)

definition *range* :: *index* ⇒ *seed* ⇒ *index* × *seed* **where**
  range k = (*if k* = 0 *then Pair* 0
    else iterate (*log* 2147483561 *k*)
     (λ*l*. *next* ∘≫ (λ*v*. *Pair* (*v* + *l* ∗ 2147483561))) 1
    ∘≫ (λ*v*. *Pair* (*v mod k*)))

Where does the initial value for the random seed in a random computation stem from? The scenario is that a random computation in the logic of type *seed* ⇒ τ × *seed* is subject to code generation to the system language *SML*, resulting in an *SML* term `f` of type `seed -> T * seed`. From this the random-value dependent result `x` of type `T` is extracted by applying `f` to an extralogically supplied random seed.

4.4.3 Generating random values of datatypes

Generators for random values of arbitrary types are accomplished using another type class:

class *random* =
  fixes *random* :: *index* ⇒ *seed* ⇒ α × *seed*

Constant *random* :: *index* ⇒ *seed* ⇒ τ × *seed* computes a random value of type τ relative to a random seed; the first argument specifies a *size* of the result, where the exact interpretation of this size parameter is not relevant.

Particular instances of *random* can be supplied by the user. For convenience this is automated for datatypes, following [55]. For example, here is a possible instantiation for *lists*:

instantiation *list* :: (*random*) *random*
begin

fun *list_random* :: *index* ⇒ *index* ⇒ *seed* ⇒ α *list* × *seed* **where**
 list_random i j =
 range i ∘≫
 (λ*l*. *if i* ≤ *l* + 1 *then Pair* []
    else random j ∘≫ (λ*x*. *list_random* (*i* − 1) *j* ∘≫ (λ*xs*. *Pair* (*x* : *xs*))))
definition
  random i = *list_random i i*

instance ..

end

4.4.4 Checking a proposition

We conclude with a description of how the key requirements from the previous sections are used for searching counterexamples of propositions.

Suppose a proposition[10] $\bigwedge\overline{x{::}\tau}_n.\ \mathit{Trueprop}\ (P\ \overline{x}_n)$ will be checked for counterexamples, given that $\tau :: (\mathit{random} \cap \mathit{term\_of})$ for all $\overline{\tau}_n$. The proposition is turned into an abstraction $\lambda\overline{x{::}\tau}_n.\ P\ \overline{x}_n$. This is then wrapped in a series of computations of random values for each parameter:

$$
\begin{array}{l}
\lambda \mathit{size}.\ \mathit{random\ size} \\
\circ\!\!\gg (\lambda x_1{::}\tau_1.\ \mathit{random\ size} \\
\circ\!\!\gg (\lambda x_2{::}\tau_2.\ \mathit{random\ size} \\
\circ\!\!\gg (\lambda x_3{::}\tau_3.\ \ldots \\
\circ\!\!\gg (\lambda x_n{::}\tau_n.\ \mathit{if}\ (P\ x_1\ x_2\ x_3\ \cdots\ x_n)\ \mathit{then\ None} \\
\mathit{else\ Some}\ [\mathit{term\_of}\ x_1,\ \mathit{term\_of}\ x_2,\ \mathit{term\_of}\ x_3,\ \ldots,\ \mathit{term\_of}\ x_n])\ \ \ldots\ )))
\end{array}
$$

which is of type $\mathit{index} \Rightarrow \mathit{seed} \Rightarrow \mathit{term\ list\ option} \times \mathit{seed}$. This term t then is subjected to code generation, yielding an *SML* term which checks the underlying proposition using random value assignments, relative to a given size; when a random value assignment refutes the proposition, the assigned values are termified and returned as a list of terms. This is finally used by the user interface to repeatedly search for counterexamples, displaying the first counterexample found.

The use of type classes is the key advantage for an implementation of quickcheck in *HOL*: the implementation benefits from dictionary construction directly and does not need to produce this by hand.

4.5 Normalisation by evaluation

The code generator infrastructure also opens a possibility for a light-weight term evaluation machinery known as *normalisation by evaluation* (NBE) [4]. The underlying idea is to delegate β-reduction and pattern matching to the runtime environment of a functional programming language but still to maintain an embedded symbolic representation of terms which allows normalised terms to be properly reconstructed and to contain uninterpreted symbols, e.g. free variables. Compared with fully symbolic evaluation this yields a considerable speedup.

We sketch briefly how the existing implementation of NBE in *Isabelle* uses the existing code generator infrastructure; for details see [1]. First, the embedded representation of terms in the implementation language *SML*:

```
datatype nterm = Symbol of name * nterm list | Abs of (nterm -> nterm);

fun apply (Symbol (name, ts)) t = Symbol (name, append ts [t])
  | apply (Abs f) t = f t;
```

[10] *Trueprop* is the embedding of *HOL* boolean values of type *bool* into the propositional type *prop* of the framework (cf. §2.2.1), which for clarity is printed here explicitly.

Terms may contain uninterpreted symbols: constants, (free) variables, etc. We represent them uniformly by `Symbol`, assuming an appropriate naming scheme to distinguish the different categories. The exact representation of type `name` does not matter; we will use strings here. Uninterpreted symbols may be applied to arguments. Abstractions are represented as functions in *SML*.

Application is implemented by a function `apply` which for uninterpreted symbols just appends its argument to the list of applied arguments, while for abstraction the argument is applied to the underlying *SML* function — this in essence delegates β-reduction to *SML*.

Code equations can be transformed to *SML* functions of type `nterm list -> nterm`, e.g. the *map* function

$$\begin{aligned}&\mathit{map} :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ \mathit{list} \Rightarrow \beta\ \mathit{list}\\ &\mathit{map}\ f\ [\![\,]\!] = [\![\,]\!]\\ &\mathit{map}\ f\ (x : \mathit{xs}) = f\ x : \mathit{map}\ f\ \mathit{xs}\end{aligned}$$

is represented on the *nterm* level as

```
fun map [f, Symbol ("Nil", [])] = Symbol ("Nil", [])
  | map [f, Symbol ("Cons", [x, xs])] =
    Symbol ("Cons", [apply f x, map [f, xs]])
  | map [f, xs] = Symbol ("map", [f, xs]);
```

The first two equations are an exact translation of the given code equations; the third is a default equation: if no previous equation matches, the whole *nterm* remains as it is.

The key observation is that *SML* functions of type `nterm list -> nterm` can be embedded into type `nterm` using the combinator:

```
fun function Zero_nat f xs = f xs
  | function (Suc n) f xs = Abs (fn x => function n f (append xs [x]));
```

In non-recursive representation, `function` is

```
function n f [] =
  Abs (fn x₁ => Abs (fn x₂ => ... Abs (fn xₙ => f [x₁, x₂, ..., xₙ]) ...))
```

Applied to the `map` example, we get:

```
function 2 map [] = Abs (fn f => Abs (fn xs => map [f, xs]))
```

Here the connection to the code generator emerges: equations in *fun* statements can be compiled seamlessly into *nterm* expressions which can themselves be used in compilations of other equations. Thus the code generator provides the necessary infrastructure for implementing a fast evaluator using NBE.

Also *case* expressions can make use of pattern matching in *SML*. The idea is to compile *case* expressions in the intermediate language to `case` expressions in *SML*; if no pattern matches, a last default clause falls back to the naive translation of the whole combinator expression (cf. §3.2.6).

class and *inst* statements can be compiled away using an appropriate dictionary construction (cf. §3.2.7).

Of what relevance are *data* statements for NBE? None. Recall (§3.2.4) that *data* statements do not contribute to the equational semantics of a program anyway. Their only purpose is to achieve the classification of some constants as constructors since typical target languages have to know about this. However NBE does not need this. So for NBE the restrictions on code equations (cf. §3.1.2) can be weakened: constants appearing in arguments on the left hand side need not be constructors. So also equations like

$$(p \wedge q) \wedge r \longleftrightarrow p \wedge (q \wedge r)$$

are usable for NBE; the practical gain however is marginal: terms occurring in realistic evaluations seldom match such patterns.

The embedded term representation also allows to lift another restriction of code equations: left-linearity. Equivalence of *nterm* values can be underapproximated using the following check:

```
fun surely_same (Symbol (name1, ts1)) (Symbol (name2, ts2)) =
  name1 = name2 andalso
    (length ts1 = length ts2 andalso surely_sames ts1 ts2)
  | surely_same (Abs f) (Abs g) = false
and surely_sames [] [] = true
  | surely_sames (t1 :: ts1) (t2 :: ts2) =
    surely_same t1 t2 andalso surely_sames ts1 ts2;
```

If `surely_same` returns `true` for two *nterms*, they are equivalent, otherwise no statement is made. This allows us to consider equations with duplicated variables on the left hand side. Each duplicated variable x is made distinct by replacing it with new variables $x_1, \ldots, x_n$. During runtime the expressions bound to those variables are checked: `surely_same` x_1 x_2 `andalso` `surely_same` x_2 x_3 `andalso` $\cdots$ `andalso` `surely_same` x_{n-1} x_n. If this check succeeds, the expressions are definitely equal and the proper equation can be applied; otherwise, the next equation is considered.

An instance of this problem is reflexivity of equality. In evaluations containing uninterpreted variables a term like *if* $x = x$ *then* A *else* B may occur. Using the above technique for lifting left-linearity, reflexivity

$$x = x \longleftrightarrow \mathit{True}$$

can be used as a code equation which reduces the above term to A.

4.6 Applications of proof terms for code generation

Isabelle per se is an LCF-style prover where proofs are irrelevant (c.f. §2.1.3), but also provides optional proof terms which can be animated in extra-logical applications. Two such applications are *proof extraction* and *elimination of overloading*; we discuss their relation to code generation briefly.

4.6.1 Extraction from constructive proofs

Extraction from constructive *HOL* proofs to executable programs has already been extensively studied [5]: when extracting from a proof *prf*, the result is formally defined in *HOL* by a constdef f_{prf} *_def*: $f_{prf} :\equiv \ldots$ and it is proved that the defined constant satisfies the property specified in *prf*. Code generation itself then proceeds using f_{prf} *_def* as an equation. Due to this architecture our code generator is accessible for proof extraction directly without any additional effort.

There remains one particular benefit to mention which stems from type classes; examine the following constructive proof:

lemma *split_last*:
  fixes xs :: α *list*
  assumes $xs \neq [\![]\!]$
  shows $\exists\, y\ ys.\ xs = ys\ @\ [\![y]\!]$
using *assms* **proof** (*induct xs*)
  case *Nil* **then have** *False* **by** *simp*
  then show *?case* **..**
next
  case (*Cons x xs*) **show** *?case* **proof** (*cases xs*)
    case *Nil* **then have** $x : xs = [\![]\!]\ @\ [\![x]\!]$ **by** *simp*
    then show *?thesis* **by** *iprover*
  next
    case *Cons* **then have** $xs \neq [\![]\!]$ **by** *simp*
    with *Cons.hyps* **have** $\exists\, y\ ys.\ xs = ys\ @\ [\![y]\!]$ **.**
    then obtain *y ys* **where** $xs = ys\ @\ [\![y]\!]$ **by** *iprover*
    then have $x : xs = (x : ys)\ @\ [\![y]\!]$ **by** *simp*
    then show *?thesis* **by** *iprover*
  qed
qed

The extracted definition looks as follows:

$$
\begin{array}{l}
\mathit{split\_last} \equiv \\
\lambda x.\ \mathit{list\_rec}\ \mathit{default} \\
\quad (\lambda x\ xa\ H2. \\
\qquad \mathit{case}\ xa\ \mathit{of}\ [\![]\!] \Rightarrow (x,\ [\![]\!]) \\
\qquad \mid\ a : \mathit{list} \Rightarrow \mathit{let}\ (xa,\ y) = H2\ \mathit{in}\ (xa,\ x : y)) \\
\quad x
\end{array}
$$

Since *HOL* is a total logic, the constant *split_last* must return a value of type α even if it is given the empty list, which is not supposed to happen in the context of an extracted program since the premise does prevent this. Thus an arbitrary value of α can serve as a formal placeholder. Following *Coq*, the standard approach is to choose an unspecified constant (here, *default*). Then the canonical translation of *default* is an exception (cf. §3.4.2):

```
split_last :: forall a. [a] -> (a, [a]);
split_last a =
  list_rec (error "default")
    (\ x xa h2 ->
      (case xa of {
        [] -> (x, []);
        aa : list -> let {
                        ab = h2;
                        (xaa, y) = ab;
                      } in (xaa, x : y);
      }))
    a;
```

Presuming that the actual value of the first argument to *list_rec* is never used, this fits nicely to a language with a lazy semantics (e.g. *Haskell*), but is problematic for eager languages: not being used does not necessarily prevent the placeholder to be evaluated. Since the actual choice of the placeholder value does not matter, this problem can be circumvented by mechanisms which substitute a user-supplied value for *default*. Beside some brittleness, this cannot deal with polymorphism properly either.

Type classes offer a natural and elegant solution to this problem: *default* is specified as parameter of a class *default*. This makes it possible to instantiate *default* as follows:[11]

instantiation $*$ :: ($default$, $default$) $default$
begin

definition
  $default = (default,\ default)$

instance ..

end

instantiation *list* :: (*type*) *default*
begin

definition
  $default = [\![]\!]$

instance ..

end

How *default* is defined on particular instances is not relevant since the actual choice of placeholder values has no impact on the correctness of the extracted code. With these instantiations, code generation can proceed canonically:

```
class Default a where {
  defaultb :: a;
};

defaulta :: forall a b. (Default a, Default b) => (a, b);
defaulta = (defaultb, defaultb);

split_last :: forall a. (Default a) => [a] -> (a, [a]);
split_last a =
  list_rec defaulta
    (\ x xa h2 ->
      (case xa of {
        [] -> (x, []);
        aa : list -> let {
                       ab = h2;
                       (xaa, y) = ab;
                     } in (xaa, x : y);
      }))
    a;
```

[11] This could be easily automated for arbitrary datatypes, but the system relieves the decision how *default* is actually defined to the user.

4.6.2 Definitional eliminating of overloading

The rules behind dictionary construction presented in §3.2.7 can also be applied to proof terms [24]. Thus it is possible to transform a system of definitions and proofs involving overloading and logically interpreted type classes such that both are eliminated by dictionaries, where class parameters f in terms $t\ [f]$ become term parameters $\lambda x.\ t\ [x]$ and type class judgements $(\!|\alpha :: c|\!)$ in proofs $P\ [(\!|\alpha :: c|\!)]$ become hypotheses $H \Longrightarrow P\ [H]$.

This might suggest that it is possible to compile away type classes from code equations to a system of code equations not referring to type classes at all. That however is not the case: when compiling away type classes from a code equation, the result may contain logical parts of a type class as a premise and hence is no code equation any more.

We illustrate this with an example:

class *decr* = *wellorder* + *bot* +
  fixes $decr :: \alpha \Rightarrow \alpha$
  assumes *decr_bot*: $decr\ \bot = \bot$
  assumes *decr_less*: $x \neq \bot \Longrightarrow decr\ x < x$

The class *decr* enriches a well-founded order (class *wellorder*) with a universal least element $\bot$ (class *bot*) by an explicit decrement operation *decr* which respects the underlying order.

Next function *distance* specifies an explicit counting of the number of *decr* steps until $\bot$ is reached:

function $distance :: \alpha{::}decr \Rightarrow nat$ **where**
  $distance\ x = (if\ x = \bot\ then\ 0\ else\ Suc\ (distance\ (decr\ x)))$
by *pat_completeness auto*

termination *distance*
proof
  from *wf* **show** $wf\ \{(y{::}\alpha{::}decr,\ x).\ y < x\}$ **.**
next
  fix $x :: \alpha{::}decr$
  assume $x \neq \bot$
  with *decr_less* **have** $decr\ x < x$ **.**
  then show $(decr\ x,\ x) \in \{(y,\ x).\ y < x\}$
    by *simp*
qed

The termination proof brings the problem to the surface: the proof inadvertently depends on the well-foundedness (theorem *wf*) and the strictness of *decr* (theorem $\bigwedge x.\ x \neq \bot \Longrightarrow decr\ x < x$); this has the consequence that the proof of the equation $distance\ x = (if\ x = \bot\ then\ 0\ else\ Suc\ (distance\ (decr\ x)))$ depends on both. After extralogical dictionary construction these dependencies would persist as explicit premises for this theorem, turning the former equation to an implication with an equation as conclusion; this obviously violates the syntactic requirements of code equations.

CHAPTER **5**

Conclusion

5.1 Stocktaking and evaluation

In the introduction we stated that our aim is the close integration of *logic* and *programming*, both in theory and in practice. We admit that this aim on its own is far too ambitious to be covered exhaustively within the range of a single PhD thesis. Nonetheless our investigation has yielded results and experiences which provide a firm base for further work in this area:

- Numerous example applications (see §4) suggest that code generation using shallow embedding matches the intuition behind the equation "higher-order logic = functional programming plus logic" quite well; the code generator is simple to use and practically applicable.
- Though the details of the code generation meta-theory (see §2.4) seem a little daunting, equational logic is appropriate to guarantee partial correctness. Restricting the executable content of a logical theory to equations is simplistic but honest: code generation only relies on properties which have a direct representation inside the logic. We refrain from stating anything about generated code concerning evaluation order, termination etc. since these issues have no logical representation in a shallow embedding. Similarly we do not use operational models for our target languages. Instead we view programs as equational rewrite systems; interaction with target-language specifics is possible but requires diligence and careful thinking (cf. §3.4.1 and §4.3).
- The principle of proof irrelevance is applied consequently, i.e. all equational theorems have the same status, regardless of their origin. This gives great flexibility in weaving implementations (see §3.2.4) and an incomplete but nevertheless useful concept for datatype abstraction (see §4.1).
- *Isabelle*'s type classes are essential; they have a straightforward interpretation as an instance of order-sorted algebra which is both the foundation for their logical content (see §2.3.2) as also for their operational elimination using dictionary construction (see §3.2.7). Though *Isabelle*'s type classes are quite restrictive from the perspective of the current type class facilities in Haskell (e.g. multi-parameter type classes and type constructor classes), they open up a number of applications, notably equality (§3.3.4), whose treatment was the original motivation for type classes anyway.

- Deductive transformations are valuable tools for transforming "raw" specifications given by the user into something accessible for code generation (e.g. equality in §3.3.4, or examples in §4.2); they leave the foundation of the code generator untouched but increase the practically executable concepts in *HOL* dramatically.
- Infrastructure shared between target languages saves a lot of duplicated work; the interfaces of the code generator allow for derived applications (e.g. §4.4) without the need to re-code the same tasks over and over.
- Formal program specification, gradual improvement, datatype refinement, etc., are well-established methodologies in software development, but practically their application is still largely restricted to "preliminary thoughts" and (probably erroneous) paper proofs. *HOL* and the code generator together provide an environment in which those techniques can be applied thoroughly in a checked and smooth manner.

Using this as a starting point, further questions and applications can be dealt with, which we will sketch in the next sections.

5.2 Bolstering the foundation of the code generator

The meta-theory of code generation could be further studied and strengthened:

5.2.1 Formalised meta-theory of the intermediate language

When presenting the intermediate language in §3.2 we gave only rough proof sketches of its properties; a rigorous treatment would require a formalisation of the intermediate language itself, especially the issue of order-sorted algebra and dictionary construction.

5.2.2 Operational semantics of target languages

The last phase of code generation, the serialisation, has been dealt with only cursory in §3.4. A first step towards a formal treatment would be to give a precise justification for the equational semantics model (Definition 10) for selected target languages, presuming a suitable operational semantics exists. This could open a perspective for a precise treatment of specific adaptations, e.g. the interaction with imperative data structures (see §4.3).

5.2.3 Evaluation strategies and termination

The equational logic model is honest in the sense that it uses only notions which are explicit in the logic. It does not cover notions like evaluation strategies and termination which have no correspondence in the logic. Therefore, no device is provided to guarantee total correctness of generated code, or, more generally, to distill the preconditions under which generated code is totally correct.

A possibility to reason about termination would be to embed the termination behaviour deeply into the logic: relations model input arguments and result values of function calls; well-founded relations certify termination (e.g. [13]). A trusted checker has to ensure that a termination certificate matches the structure of corresponding code equations.

5.3 Extending the foundation of the code generator

Some shortcomings in the code generator can only be amended by extending its foundation:

5.3.1 Invariants

The inability to specify invariants (see §4.1) turned out to be tiresome. The question is how to encode them into the code equations. As an example, imagine we would choose $set :: \alpha\ list \Rightarrow \alpha\ set$ as constructor for implementing sets. Removal of a single element from such a set can be described by a guarded equation together with an invariant preservation statement:

$$\bigwedge xs\ x.\ no\_duplicates\ xs \Longrightarrow set\ xs - \{x\} = set\ (remove\ x\ xs)$$
$$\bigwedge xs\ x.\ no\_duplicates\ xs \Longrightarrow no\_duplicates\ (remove\ x\ xs)$$

which reads: under the assumption that the elements in the representing list are distinct, we can remove an element from the set by removing its first occurrence in the representing list; the elements in the resulting representing list are also distinct. Here, *no_duplicates* describes the invariant which arguments to *set* must obey. Permitting guarded code equations would demand a policy like the following: be C a constructor whose arguments $\bar{t}$ in at least one code equation are guarded by a predicate P, then

- each occurrence of some $C\ \bar{p}$ on the left hand side of a code equation can be guarded by $P\ \bar{p}$;
- each occurrence of some $C\ \bar{t}$ on the right hand side of a code equation must be guarded by $P\ \bar{t}$.

Informally that means that any application of C must be guaranteed to respect P, while any pattern matching against C can assume that its arguments respect P.

5.3.2 Predicate subtyping

The *PVS* proof system provides a logic similar to *HOL* with one outstanding facility: *predicate subtypes* [50]. The set of values of a predicate subtype is described as the set of values of a type satisfying a predicate. This allows to operate with notions like "the type of all natural numbers which are multiples of three" directly.

An adaptation of this concept could offer a perspective to deal with partial functions. Assume we have the following guarded equation specifying the half of even natural numbers:

$$\bigwedge n.\ even\ n \Longrightarrow half\ n = (if\ n = 0\ then\ 0\ else\ Suc\ (half\ (n - 2)))$$

Code generation currently cannot cope with such "partial" functions. A solution could be to view such guards as predicate subtype specifications. In the example this would mean that each occurrence of *half* on the right hand side must by guarded by *even*:

$$\bigwedge n.\ even\ n \Longrightarrow even\ (n - 2)$$

This goes in a similar direction as datatype invariants.

5.3.3 Logics other than *HOL*

Code generation currently is based directly on *Isabelle/Pure* and its extension *Isabelle/HOL*. However there are other object logics which could likewise serve as a framework for development of executable specifications. One natural candidate for this is *Isabelle/HOLCF*, a formalisation of domain theory on top of *Isabelle/HOL* [40]. Continuous functions in *HOLCF* are modelled by a separate continuous function space $\alpha \rightarrow \beta$ equipped with an embedded continuous application $(\cdot) :: (\alpha \rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$. Therefore, specifications involving the continuous function space do not yield code equations, e.g. the characteristic equation of the continuous identity function

$$id_{\rightarrow} \cdot x = x$$

is *not* a code equation since the continuous application on the left hand side violates the syntactic requirements (cf. Definition 15).

One possibility to accommodate for this could be to put an object-logic specific *foundation layer* between translation and logic; this layer would provide an abstract view on logical ingredients and explain how "operational" entities (equations, types, classes, ...) are retrieved from logical ones (theorems, context, ...). In the *Pure*/*HOL* case, the layer would just hand things through, whereas for *HOLCF* continuous application would be mapped onto *Pure* application etc. Of course each foundation layer would need to be justified according to the specifics of the corresponding object logic.

5.4 Extending the code generator infrastructure

Within the fixed meta-theoretical framework of the code generator, the following additions can be thought of:

5.4.1 Further target languages

In §2.4 we gave a characterisation of the requirements a language has to fulfil to serve as target language within our framework. Beside the existing serialisers for *SML*, *OCaml* and *Haskell*, two further promising candidates are *Scheme* [56] and *Scala* [44]. *Scheme* itself has no notion of patterns, but these can be simulated using a combinator, as the code extraction of *Coq* does [36].

5.4.2 Managing scope and accessibility

Currently, code is generated as a bulk of statements; when this is supposed to be used in a bigger program, we silently assume that the programmer knows how to use this code and where its entry points are supposed to be. Maybe this is unsatisfactory for larger developments; the code generator could be enhanced such that the user can specify which functions are to be exported and which not.

5.5 Deductive tools and advanced applications

The extensions discussed below do not affect the code generator at all: they provide richer automation and expressiveness to the user for specifications and increase the domain of generated code.

5.5.1 Packing machinery

When discussing datatype abstraction in §4.1.3, we had to introduce the trivial datatype **datatype** (α, β) *map* $=$ *Map* $(\alpha \Rightarrow \beta$ *option*$)$ in order to establish an abstraction over its concrete representation. A similar situation occurs with sets in *HOL* which are represented as predicates $\alpha \Rightarrow$ *bool*. Developing an implementation for finite sets is possible using constructors $\{\}$ :: α *set* and *insert* :: $\alpha \Rightarrow \alpha$ *set* $\Rightarrow$ α *set*, but this also requires that values of sets are *packed* into a proper type (say, α *fset*).

The unpacked types $\alpha \Rightarrow \beta$ *option* and $\alpha \Rightarrow$ *bool* have the advantage that proofs operate directly on generic concepts which profit from existing rules and automation without the need to pack and unpack values explicitly. For code generation, the situation is the other way round: explicit type constructors are necessary to establish a proper abstraction and perhaps distinguish different kinds of mappings or sets. So, when starting a formal development, the user has to choose whether

- to use (duplicated) packed types (*map*, *fset*), thus complicating the proofs
- or to use the unpacked types ($\alpha \Rightarrow \beta$ *option*, $\alpha \Rightarrow$ *bool*) and deriving a (duplicated) executable version in parallel which uses packed types (*map*, *fset*).

This situation is unsatisfactory. A possible solution would be an automated *packing* machinery which could work roughly as follows:

- For a given type $\tau\ [\overline{\alpha}]$, define a packed type $\kappa\ \overline{\alpha}$ as datatype with a constructor $C_\kappa :: \tau\ [\overline{\alpha}] \Rightarrow \kappa\ \overline{\alpha}$ and a projection $d_\kappa :: \kappa\ \overline{\alpha} \Rightarrow \tau\ [\overline{\alpha}]$ such that the inversions $\bigwedge x.\ C_\kappa\ (d_\kappa\ x) \equiv x$ and $\bigwedge x.\ d_\kappa\ (C_\kappa\ x) \equiv x$ hold.
- Let the user supply one or more constants whose type signature and code equations shall be packed (e.g. $f :: nat \Rightarrow \tau\ [\overline{\alpha}] \Rightarrow \tau\ [\overline{\alpha}]$); for each of these f a packed constant f' is defined which a packed type (here, $f' :: nat \Rightarrow \kappa\ \overline{\alpha} \Rightarrow \kappa\ \overline{\alpha}$), using f, C_κ and d_κ (here constdef $f' :\equiv \lambda x.\ (C_\kappa\ (f\ (d_\kappa\ x)))$). With the inversions from above substitution rules for each f (here, $f \equiv \lambda x.\ (d_\kappa\ (f'\ (C_\kappa\ x))))$ are proved.

- Using the substitution rules and the inversions, the code equations of each f are packed into code equations for f' (e.g., $\bigwedge x.\ f\ x \equiv \dots\ x\ \dots$ is packed into $\bigwedge x.\ f'\ x \equiv C_\kappa\ (\dots\ d_\kappa\ x\ \dots)$); hereby the inversions eliminate every occurrence of $C_\kappa\ (d_\kappa\ \dots)$ and $d_\kappa\ (C_\kappa\ \dots)$.
- In consequence, all code equations which depend on f are packed.
- The resulting set of code equations is the same as if type $\kappa\ \overline{\alpha}$ would have been present in the underlying specification from the beginning. Code equations not containing C_κ or d_κ use $\kappa\ \overline{\alpha}$ abstract, while others access its concrete representation $\tau\ [\overline{\alpha}]$; for these an alternative implementation can be provided using the existing concepts for datatype abstraction.

A prerequisite for this machinery are *functorial lifters* for all types which may contain packed types as parameters; each such type $\kappa'\ \overline{\alpha}_k$ must provide such a lifter $map_{\kappa;i} :: (\alpha_i \Rightarrow \beta) \Rightarrow \kappa'\ \alpha_1\ \dots\ \alpha_{i-1}\ \alpha_i\ \alpha_{i+1}\ \dots\ \alpha_k \Rightarrow \kappa'\ \alpha_1\ \dots\ \alpha_{i-1}\ \beta\ \alpha_{i+1}\ \dots\ \alpha_k$ for each type parameter α_i such that $\bigwedge f\ g\ y.\ (\bigwedge x.\ f\ (g\ x) = x) \Longrightarrow map_{\kappa;i}\ f\ (map_{\kappa;i}\ g\ y) \equiv y$.

5.5.2 Infinite data structures

In lazy languages like *Haskell*, infinite data structures are a common modelling device, e.g. the list of all even natural numbers:

```
even :: [Integer]
even = even’ 0 where even’ n = n : even’ (n + 2)
```

For such lazy types currently no tool support is provided in *HOL*. One approach could be coinductive datatypes which are defined as greatest rather than least fixpoints [37, 20]. Another possibility is use a domain-theoretic approach using *HOLCF* (see above §5.3.3); this would also demand to extend the foundation of the code generator to cover *HOLCF* as well as sketched above.

5.5.3 Parallelism

With the advent of multi-core processors, the importance of parallelism has increased dramatically. To adopt code generation for parallelism, suitable concepts from *Parallel Haskell* can be borrowed:

- A pure functional language permits parallelism inherently, e.g. the well-known combinator `map :: (a -> b) -> [a] -> [b]` can apply its function argument to all elements of its list argument in parallel without changing its equational semantics. To apply this for code generation no further infrastructure is needed: parallel combinators can be specified conventionally in *HOL* and mapped to parallel counterparts using the existing adaptation mechanism (see §3.4.1).
- The currently rising star on the sky of concurrent programming is *Software Transactional Memory* (STM) [27]. It provides a framework to implement reentrant transactions using shared transactional variables; access conflicts are resolved by the framework by rolling back transactions. This transactional approach to synchronisation is well-established in relational databases; it avoids

the typical problems of primitive synchronisation mechanisms like deadlocks, starvation, etc.

The challenge here is how to adopt STM to *HOL*. The transaction-based concurrency framework has to be embedded into the purely functional logic. Also means to reason about STM-based programs must be provided.

For me the chief thing is that I feel
that the whole matter is now "exorcised",
and rides me no more.
I can turn now to other things.
John Ronald Reuel Tolkien,
author of the century [52],
from a letter to Stanley Unwin

APPENDIX **A**

Notions of the *Pure* logic and their notations

Isopropyl-propenyl-barbitursaures-phenyl-
dimethyl-dimethyl-amino-pyrazolon. ...
So einfach, und man kann sich's doch nicht merken.
Karl Valentin, from: In der Apotheke

A.1 Expressions

(c.f. Synopsis 1)

sorts $s ::= c_1 \cap \ldots \cap c_l$
top sort $\top$

types $\tau ::= \kappa\ \tau_1\ \ldots\ \tau_k \mid \alpha{::}s$
type variables α, β, γ, ...

terms $t ::= t_1\ t_2 \mid \lambda x{::}\tau.\ t \mid x{::}\tau \mid f$
terms t, u, v, w, ...
variables x, y, z, ...
constants f, g, ...

proofs containing implication $P \Longrightarrow Q$
and universal quantification $\bigwedge x{::}\tau.\ P\ x$

A.2 Theory context

(c.f. Synopsis 2 and 7)

$\Theta = (\mho, \Sigma, \Upsilon, \Omega, \omega, \ldots)$ with

subclass relation $\mho\ c = \{c_1, \ldots, c_k\}$

type arities $\Upsilon\ \kappa = * \to \cdots \to *$

arity signature $\Sigma\ c_\kappa = \overline{s}$

constant types $\Omega\ f = \forall\, \alpha_1\ \ldots\ \alpha_n.\ \tau$

constant-to-class association $\omega\ f = c$

A.3 Theory extensions

(c.f. Synopsis 3, 4, 5 and 8, also 6)

class definition

$\mathsf{classdef}\ c \subseteq c_1 \cap \ldots \cap c_n\colon\ P\ [\alpha]$

type definition (no *Pure* theory extension, only *HOL*!)

$\mathsf{typedef}\ \kappa\ \overline{\alpha} = \{x{::}\tau.\ P\ x\}\ \ \langle \mathit{proof} \rangle$

type declaration

$\mathsf{typedecl}\ \kappa :: * \to \cdots \to *$

instance definition

$\mathsf{instance}\ \kappa :: (\overline{s})\ c\ \ \langle \mathit{proof} \rangle$

constant definition

$\mathsf{constdef}\ \mathit{f\_def}\colon (f :: \tau\ [\overline{\alpha}]) :\equiv t$

constant declaration

$\mathsf{constdecl}\ f :: \forall\, \overline{\alpha}.\ \tau$

overloaded definition

$\mathsf{overload}\ \mathit{f\_\kappa\_def}\colon (f :: \tau\ [\kappa\ \overline{\alpha}]) :\equiv t$

theorem definition

$\mathsf{theorem}\ a\colon\ P\ \ \langle \mathit{proof} \rangle$

axiom declaration

$\mathsf{axiom}\ a\colon\ P$

APPENDIX **B**

Selected ingredients of *Isabelle/HOL*

In his thinking, things had to be done.
And if no one else would be hacking them, he would.
Steven Levy, American journalist,
from: Hackers

booleans	**datatype** $bool = True \mid False$
base connectives	$P \wedge Q$, $P \vee Q$, $P \longrightarrow Q$, $P \longleftrightarrow Q$
sets	$\alpha\ set \rightleftharpoons (\alpha \Rightarrow bool)$
intersection	$A \cap B$
union	$A \cup B$
difference	$A - B$
membership	$x \in A$
empty set	$\{\}$
set literals	$\{a,\ b,\ c\}$
singleton insertion	$insert\ a\ A = \{a\} \cup A$
lattices	
infimum	$x \sqcap y$
supremum	$x \sqcup y$
set infimum	$\bigsqcap\{x,\ y,\ z,\ \ldots\} = x \sqcap y \sqcap z \sqcap \ldots$
set supremum	$\bigsqcup\{x,\ y,\ z,\ \ldots\} = x \sqcup y \sqcup z \sqcup \ldots$
bottom element	$\bot$
top element	$\top$
arithmetic	
natural numbers	**datatype** $nat = 0 \mid Suc\ nat$
integer numbers	int
number literals	42, 1705
basic arithmetic	$x + y$, $x - y$, $x * y$
comparisons	$x \leq y$, $x < y$, $min\ x\ y$, $max\ x\ y$
divisibility	$x\ div\ y$, $x\ mod\ y$, $gcd\ x\ y$
conversions	$nat :: int \Rightarrow nat$, $int :: nat \Rightarrow int$

products	**datatype** $\alpha \times \beta = Pair\ \alpha\ \beta$
tuples	$(a,\ b) \rightleftharpoons Pair\ a\ b$ – $(a,\ b,\ c) \rightleftharpoons (a,\ (b,\ c))$
projections	$fst :: \alpha \times \beta \Rightarrow \alpha$, $snd :: \alpha \times \beta \Rightarrow \beta$

sums	**datatype** $\alpha + \beta = Inl\ \alpha \mid Inr\ \beta$

options	**datatype** $\alpha\ option = None \mid Some\ \alpha$

lists	**datatype** $\alpha\ list = [\![]\!] \mid (:)\ \alpha\ (\alpha\ list)$
destructors	$head\ (x : xs) = x$ $tail\ [\![]\!] = [\![]\!]$ $tail\ (x : xs) = xs$
functorials	$fold\ f\ [\![]\!]\ s = s$ $fold\ f\ (x : xs)\ s = fold\ f\ xs\ (f\ x\ s)$ $map\ f\ [\![]\!] = [\![]\!]$ $map\ f\ (x : xs) = f\ x : map\ f\ xs$ $filter\ P\ [\![]\!] = [\![]\!]$ $filter\ P\ (x : xs) = (if\ P\ x\ then\ x : filter\ P\ xs\ else\ filter\ P\ xs)$ $map\_filter\ f\ [\![]\!] = [\![]\!]$ $map\_filter\ f\ (x : xs) =$ $(case\ f\ x\ of\ None \Rightarrow map\_filter\ f\ xs \mid Some\ y \Rightarrow y : map\_filter\ f\ xs)$
concatenation	$[\![]\!]\ @\ ys = ys$ $(x : xs)\ @\ ys = x : xs\ @\ ys$ $flat\ [\![]\!] = [\![]\!]$ $flat\ (x : xs) = x\ @\ flat\ xs$
reversal	$rev\ [\![]\!] = [\![]\!]$ $rev\ (x : xs) = rev\ xs\ @\ [\![x]\!]$
indexing	$length\ [\![]\!] = 0$ $length\ (x : xs) = Suc\ (length\ xs)$ $(x : xs)\ !\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ k \Rightarrow xs\ !\ k)$ $map\_nth\ n\ f\ [\![]\!] = [\![]\!]$ $map\_nth\ 0\ f\ (x : xs) = f\ x : xs$ $map\_nth\ (Suc\ n)\ f\ (x : xs) = x : map\_nth\ n\ f\ xs$
set conversion	$set\ [\![]\!] = \{\}$ $set\ (x : xs) = insert\ x\ (set\ xs)$
singleton removal	$remove\ x\ [\![]\!] = [\![]\!]$ $remove\ x\ (y : xs) = (if\ x = y\ then\ xs\ else\ y : remove\ x\ xs)$
duplicates	$no\_duplicates\ [\![]\!] \longleftrightarrow True$ $no\_duplicates\ (x : xs) \longleftrightarrow x \notin set\ xs \wedge no\_duplicates\ xs$ $distinct\ [\![]\!] = [\![]\!]$ $distinct\ (x : xs) =$ $(if\ x \in set\ xs\ then\ distinct\ xs\ else\ x : distinct\ xs)$

APPENDIX C

Code examples

```
/* You are not expected to understand this */

if (rp -> p_flag & SSWAP) {
  rp -> p_flag =& ~SSWAP;
  aretu(u.u_ssav);
}
```

Sixth Edition Unix, lines 2240ff.

C.1 Rational numbers

(see §4.1.2)

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module Rat where {


leta :: forall b a. b -> (b -> a) -> a;
leta s f = f s;

abs_int :: Integer -> Integer;
abs_int i = (if i < 0 then negate i else i);

split :: forall b c a. (b -> c -> a) -> (b, c) -> a;
split f (a, b) = f a b;

sgn_int :: Integer -> Integer;
sgn_int i = (if i == 0 then 0 else (if 0 < i then 1 else negate 1));

apsnd :: forall c b a. (c -> b) -> (a, c) -> (a, b);
apsnd f (x, y) = (x, f y);

divmod :: Integer -> Integer -> (Integer, Integer);
divmod k l =
  (if k == 0 then (0, 0)
    else (if l == 0 then (0, k)
           else apsnd (\ a -> sgn_int l * a)
                  (if sgn_int k == sgn_int l
                    then (\k l -> divMod (abs k) (abs l)) k l
```

```
                    else let {
                           a = (\k l -> divMod (abs k) (abs l)) k l;
                           (r, s) = a;
                         } in (if s == 0 then (negate r, 0)
                                else (negate r - 1, abs_int l - s)))));

mod_int :: Integer -> Integer -> Integer;
mod_int a b = snd (divmod a b);

zgcd :: Integer -> Integer -> Integer;
zgcd k l =
  abs_int
    (if l == 0 then k else zgcd l (mod_int (abs_int k) (abs_int l)));

data Rat = Fract Integer Integer;

div_int :: Integer -> Integer -> Integer;
div_int a b = fst (divmod a b);

fract_norm :: Integer -> Integer -> Rat;
fract_norm a b =
  (if a == 0 || b == 0 then Fract 0 1
    else let {
           c = zgcd a b;
         } in (if 0 < b then Fract (div_int a c) (div_int b c)
                else Fract (negate (div_int a c))
                       (negate (div_int b c))));

plus_rat :: Rat -> Rat -> Rat;
plus_rat (Fract a b) (Fract c d) =
  (if b == 0 then Fract c d
    else (if d == 0 then Fract a b
           else fract_norm (a * d + c * b) (b * d)));

minus_rat :: Rat -> Rat -> Rat;
minus_rat (Fract a b) (Fract c d) =
  (if b == 0 then Fract (negate c) d
    else (if d == 0 then Fract a b
           else fract_norm (a * d - c * b) (b * d)));

times_rat :: Rat -> Rat -> Rat;
times_rat (Fract a b) (Fract c d) = fract_norm (a * c) (b * d);

divide_rat :: Rat -> Rat -> Rat;
divide_rat (Fract a b) (Fract c d) = fract_norm (a * d) (b * c);

}
```

C.2 Mappings — naive implementation

(see §4.1.3)

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module Mapping_Naive where {


newtype Map a b = Map (a -> Maybe b);

empty :: forall a b. Map a b;
empty = Map (\ uu -> Nothing);
```

```
point :: forall a b. (Eq a) => a -> (b -> b) -> (a -> b) -> a -> b;
point x g f z = (if z == x then g (f z) else f z);

delete :: forall a b. (Eq a) => a -> Map a b -> Map a b;
delete k (Map f) = Map (point k (\ uu -> Nothing) f);

lookupa :: forall a b. Map a b -> a -> Maybe b;
lookupa (Map f) = f;

update :: forall a b. (Eq a) => a -> b -> Map a b -> Map a b;
update k v (Map f) = Map (point k (\ uu -> Just v) f);

}
```

C.3 Mappings — implementation by association lists

(see §4.1.3)

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module Mapping_AList where {


data Nat = Zero_nat | Suc Nat;

mapa :: forall b a. (b -> a) -> [b] -> [a];
mapa f [] = [];
mapa f (x : xs) = f x : mapa f xs;

newtype Map a b = AList [(a, b)];

deletea :: forall a b. (Eq a) => a -> [(a, b)] -> [(a, b)];
deletea k [] = [];
deletea k (x : xs) =
  (if k == fst x then deletea k xs else x : deletea k xs);

lookupb :: forall a b. (Eq a) => [(a, b)] -> a -> Maybe b;
lookupb [] k = Nothing;
lookupb (x : xs) k =
  (if k == fst x then Just (snd x) else lookupb xs k);

updatea :: forall a b. (Eq a) => a -> b -> [(a, b)] -> [(a, b)];
updatea k v [] = [(k, v)];
updatea k v (x : xs) =
  (if k == fst x then (k, v) : xs else x : updatea k v xs);

member :: forall a. (Eq a) => [a] -> a -> Bool;
member [] y = False;
member (x : xs) y = x == y || member xs y;

remdups :: forall a. (Eq a) => [a] -> [a];
remdups [] = [];
remdups (x : xs) = (if member xs x then remdups xs else x : remdups xs);

lengtha :: forall a. [a] -> Nat;
lengtha [] = Zero_nat;
lengtha (x : xs) = Suc (lengtha xs);

size :: forall a b. (Eq a) => Map a b -> Nat;
size (AList xs) = lengtha (remdups (mapa fst xs));
```

```
empty :: forall a b. Map a b;
empty = AList [];

delete :: forall a b. (Eq a) => a -> Map a b -> Map a b;
delete k (AList xs) = AList (deletea k xs);

lookupa :: forall a b. (Eq a) => Map a b -> a -> Maybe b;
lookupa (AList xs) = lookupb xs;

update :: forall a b. (Eq a) => a -> b -> Map a b -> Map a b;
update k v (AList xs) = AList (updatea k v xs);

}
```

C.4 Mappings — implementation by binary trees

(see §4.1.3)

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module Mapping_Tree where {


data Nat = Zero_nat | Suc Nat;

class Orda a where {
  less_eq :: a -> a -> Bool;
  less :: a -> a -> Bool;
};

mapa :: forall b a. (b -> a) -> [b] -> [a];
mapa f [] = [];
mapa f (x : xs) = f x : mapa f xs;

data Tree a b = Empty | Branch b a (Tree a b) (Tree a b);

append :: forall a. [a] -> [a] -> [a];
append [] ys = ys;
append (x : xs) ys = x : append xs ys;

class (Orda a) => Preorder a where {
};

class (Preorder a) => Order a where {
};

class (Order a) => Linorder a where {
};

keys :: forall a b. (Linorder a) => Tree a b -> [a];
keys Empty = [];
keys (Branch uu k l r) = k : append (keys l) (keys r);

member :: forall a. (Eq a) => [a] -> a -> Bool;
member [] y = False;
member (x : xs) y = x == y || member xs y;

remdups :: forall a. (Eq a) => [a] -> [a];
remdups [] = [];
remdups (x : xs) = (if member xs x then remdups xs else x : remdups xs);
```

```
lookupb :: forall a b. (Eq a, Linorder a) => Tree a b -> a -> Maybe b;
lookupb Empty = (\ uu -> Nothing);
lookupb (Branch v k l r) =
  (\ k' ->
    (if k' == k then Just v
      else (if less_eq k' k then lookupb l k' else lookupb r k')));

is_none :: forall b. Maybe b -> Bool;
is_none (Just x) = False;
is_none Nothing = True;

filtera :: forall a. (a -> Bool) -> [a] -> [a];
filtera p [] = [];
filtera p (x : xs) = (if p x then x : filtera p xs else filtera p xs);

lengtha :: forall a. [a] -> Nat;
lengtha [] = Zero_nat;
lengtha (x : xs) = Suc (lengtha xs);

sizea :: forall a b. (Eq a, Linorder a) => Tree a b -> Nat;
sizea t =
  lengtha
    (filtera (\ x -> not (is_none x))
      (mapa (lookupb t) (remdups (keys t))));

newtype (Linorder a) => Map a b = Tree (Tree a b);

updatea ::
  forall a b. (Eq a, Linorder a) => a -> b -> Tree a b -> Tree a b;
updatea k v Empty = Branch v k Empty Empty;
updatea k' v' (Branch v k l r) =
  (if k' == k then Branch v' k l r
    else (if less_eq k' k then Branch v k (updatea k' v' l) r
           else Branch v k l (updatea k' v' r)));

size :: forall a b. (Eq a, Linorder a) => Map a b -> Nat;
size (Tree t) = sizea t;

empty :: forall a b. (Linorder a) => Map a b;
empty = Tree Empty;

lookupa :: forall a b. (Eq a, Linorder a) => Map a b -> a -> Maybe b;
lookupa (Tree t) = lookupb t;

update ::
  forall a b. (Eq a, Linorder a) => a -> b -> Map a b -> Map a b;
update k v (Tree t) = Tree (updatea k v t);

}
```

C.5 Beta-normalisation of λ-terms

(see §4.2.3)

```
structure Lambda =
struct

type 'a eq = {eq : 'a -> 'a -> bool};
fun eq (A_:'a eq) = #eq A_;

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

```
fun eqop A_ a b = eq A_ a b;

fun eq_nat (Suc a) Zero_nat = false
  | eq_nat Zero_nat (Suc a) = false
  | eq_nat (Suc nat) (Suc nat') = eq_nat nat nat'
  | eq_nat Zero_nat Zero_nat = true;

val eq_nata = {eq = eq_nat} : nat eq;

val one_nat : nat = Suc Zero_nat

fun less_nat m (Suc n) = less_eq_nat m n
  | less_nat n Zero_nat = false
and less_eq_nat (Suc m) n = less_nat m n
  | less_eq_nat Zero_nat n = true;

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

datatype 'a seq = Empty | Insert of 'a * 'a pred |
  Join of 'a pred * 'a seq
and 'a pred = Seq of (unit -> 'a seq);

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

fun bind (Seq g) f = Seq (fn u => apply f (g ()))
and apply f Empty = Empty
  | apply f (Insert (x, p)) = Join (f x, Join (bind p f, Empty))
  | apply f (Join (p, xq)) = Join (bind p f, apply f xq);

val bot_pred : 'a pred = Seq (fn u => Empty)

fun single x = Seq (fn u => Insert (x, bot_pred));

fun seq_case f1 f2 f3 (Join (pred, seq)) = f3 pred seq
  | seq_case f1 f2 f3 (Insert (a, pred)) = f2 a pred
  | seq_case f1 f2 f3 Empty = f1;

fun adjunct p Empty = Join (p, Empty)
  | adjunct p (Insert (x, q)) = Insert (x, sup_pred q p)
  | adjunct p (Join (q, xq)) = Join (q, adjunct p xq)
and sup_pred (Seq f) (Seq g) =
  Seq (fn u =>
        (case f () of Empty => g ()
           | Insert (x, p) => Insert (x, sup_pred p (Seq g))
           | Join (p, xq) => adjunct (Seq g) (Join (p, xq))));

datatype lambda = Var of nat | App of lambda * lambda | Abs of lambda;

fun lift k (Var i) =
  (if less_nat i k then Var i else Var (plus_nat i one_nat))
  | lift k (App (s, t)) = App (lift k s, lift k t)
  | lift k (Abs s) = Abs (lift (plus_nat k one_nat) s);

fun subst k s (Var i) =
  (if less_nat k i then Var (minus_nat i one_nat)
    else (if eqop eq_nata i k then s else Var i))
  | subst k s (App (t, u)) = App (subst k s t, subst k s u)
  | subst k s (Abs t) =
    Abs (subst (plus_nat k one_nat) (lift Zero_nat s) t);

fun lambda_case f1 f2 f3 (Abs lambda) = f3 lambda
  | lambda_case f1 f2 f3 (App (lambda1, lambda2)) = f2 lambda1 lambda2
  | lambda_case f1 f2 f3 (Var nat) = f1 nat;
```

```
fun beta_1 x1 =
  sup_pred
    (bind (single x1)
      (fn a =>
        (case a of Var nat => bot_pred
           | App (lambda1, t) =>
             (case lambda1 of Var nat => bot_pred
                | App (lambda1a, lambda2b) => bot_pred
                | Abs s => single (subst Zero_nat t s))
           | Abs lambda => bot_pred)))
    (sup_pred
      (bind (single x1)
        (fn a =>
          (case a of Var nat => bot_pred
             | App (s, u) =>
               bind (beta_1 s) (fn x => single (App (x, u)))
             | Abs lambda => bot_pred)))
      (sup_pred
        (bind (single x1)
          (fn a =>
            (case a of Var nat => bot_pred
               | App (u, s) =>
                 bind (beta_1 s) (fn x => single (App (u, x)))
               | Abs lambda => bot_pred)))
        (bind (single x1)
          (fn a =>
            (case a of Var nat => bot_pred
               | App (lambda1, lambda2) => bot_pred
               | Abs s => bind (beta_1 s) (fn x => single (Abs x)))))));

end; (*struct Lambda*)
```

APPENDIX **D**

Cantor's first diagonalisation argument

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last.
John McCarthy, computer science pioneer, from:
A Basis for a Mathematical Theory of Computation,
1963

We give explicit proofs for the encoding of countable types from and to natural numbers using Cantor's first diagonalisation argument (e.g. [51]), as required for the heap construction in §4.3.2. The diagonalisation is implemented by two complementary operations:

$$\begin{aligned}&\mathit{diagonalize} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\\&\mathit{tupelize} :: \mathit{nat} \Rightarrow \mathit{nat} \times \mathit{nat}\end{aligned}$$

with the characteristic properties:

$$\begin{aligned}&\mathit{tupelize\_diagonalize}\colon \bigwedge m\ n.\ \mathit{tupelize}\ (\mathit{diagonalize}\ m\ n) = (m,\ n)\\&\mathit{diagonalize\_inject}\colon \bigwedge a\ b\ c\ d.\ \mathit{diagonalize}\ a\ b = \mathit{diagonalize}\ c\ d \Longrightarrow a = c\\&\bigwedge a\ b\ c\ d.\ \mathit{diagonalize}\ a\ b = \mathit{diagonalize}\ c\ d \Longrightarrow b = d\end{aligned}$$

One suitable choice for *diagonalize* is

$$\mathit{diagonalize}\ m\ n = (\mathit{let}\ q = m + n\ \mathit{in}\ q * \mathit{Suc}\ q\ \mathit{div}\ 2 + m)$$

This enables us to provide suitable instances for class *countable*:

class *countable* =
  fixes *nat_of* :: $\alpha \Rightarrow \mathit{nat}$
  fixes *of_nat* :: $\mathit{nat} \Rightarrow \alpha$
  assumes *nat_of_inject*: $\mathit{nat\_of}\ x = \mathit{nat\_of}\ y \Longrightarrow x = y$
  assumes *of_nat_of*: $\mathit{of\_nat}\ (\mathit{nat\_of}\ x) = x$

For the natural numbers the encoding is just identity:

instantiation *nat* :: *countable*
begin

definition
[*simp*]: $nat\_of = (id :: nat \Rightarrow nat)$

definition
[*simp*]: $of\_nat = (id :: nat \Rightarrow nat)$

instance proof
qed *simp_all*

end

Finite types like *bool* can be encoded directly:

instantiation *bool* :: *countable*
begin

definition
$nat\_of\ b = (if\ b\ then\ (1{::}nat)\ else\ 0)$

definition
$of\_nat\ (n{::}nat) \longleftrightarrow (n \neq 0)$

instance proof
qed (*simp_all add*: *nat_of_bool_def of_nat_bool_def split*: *if_splits*)

end

The interesting case are products, where we use the diagonalisation argument:

instantiation $*$ :: (*countable*, *countable*) *countable*
begin

definition
$nat\_of\ p = diagonalize\ (nat\_of\ (fst\ p))\ (nat\_of\ (snd\ p))$

definition
$of\_nat\ n = (let\ (m,\ q) = tupelize\ n\ in\ (of\_nat\ m,\ of\_nat\ q))$

instance proof
qed (*auto simp add*: *split_paired_all nat_of_prod_def of_nat_prod_def*
tupelize_diagonalize of_nat_of dest: *diagonalize_inject nat_of_inject*)

end

Sums can already use the existing encoding of products:

```
instantiation + :: (countable, countable) countable
begin

definition
  nat_of z = (case z of Inl x ⇒ nat_of (False, nat_of x)
    | Inr y ⇒ nat_of (True, nat_of y))

definition
  of_nat n = (let (b, m) = of_nat n in if b
    then Inr (of_nat m)
    else Inl (of_nat m))

instance proof
qed (auto simp add: nat_of_sum_def of_nat_sum_def of_nat_of
  dest: nat_of_inject split: sum.splits)

end
```

The remaining concept for datatypes is recursion, for which lists are the canonical example:

```
instantiation list :: (countable) countable
begin

primrec nat_of_list where
    nat_of [] = 0
  | nat_of (x : xs) = Suc (nat_of (x, nat_of xs))

fun of_nat_list where
   of_nat 0 = []
 | of_nat (Suc n) = (let (x, m) = of_nat n in x : of_nat m)

instance proof
  fix xs ys :: α list
  assume eq: nat_of xs = nat_of ys
  then have length xs = length ys
  proof (induct xs arbitrary: ys)
    case Nil then show ?case by (cases ys) simp_all
  next
    case (Cons x xs)
    from Cons.hyps
    have nat_of xs = nat_of (tail ys) ⟹ length xs = length (tail ys) .
    with Cons.prems show ?case
      by (cases ys) (auto dest: nat_of_inject)
  qed then show xs = ys using eq by (induct rule: list_induct2)
    (auto dest: nat_of_inject)
next
  fix xs :: α list
  show of_nat (nat_of xs) = xs
    by (induct xs) (simp_all add: of_nat_of)
```

qed

end

All these example instances may serve as patterns how arbitrary first-order datatypes can be equipped with *countable* instances.

Bibliography

[1] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs '08: Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 2008.

[2] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In *Types for Proofs and Programs (TYPES 2003)*, volume 2277 of *Lecture Notes in Computer Science*, pages 34–50. Springer-Verlag, 2004.

[3] Friedrich Ludwig Bauer. Was ist Programmtransformation? *Elektronische Rechenanlagen*, 18:229–233, 1976.

[4] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[5] Stefan Berghofer. Program extraction in simply-typed higher order logic. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2277 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 2002.

[6] Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs '09: Proceedings of the 22th International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 131–146. Springer-Verlag, 2009.

[7] Stefan Berghofer and Tobias 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 *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[8] Stefan Berghofer and Makarius Wenzel. Logic-free reasoning in Isabelle/Isar. In *Mathematical Knowledge Management (MKM 2008)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2008.

[9] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[10] R. S. Boyer and J Strother 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.

[11] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java, Second Edition.* Prentice Hall International, September 2003.

[12] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs '08: Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 2008.

[13] Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *TPHOLs '07: Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, 2007.

[14] F. Warren Burton. An efficient functional implementation of FIFO queues. In *Information processing letters*, volume 14, pages 205–206, 1982.

[15] Amine Chaieb and Makarius Wenzel. SML with antiquotations embedded into Isabelle/Isar. In J. Carette and F. Wiedijk, editors, *Programming Languages for Mechanized Mathematics Workshop (CALCULEMUS 2007)*, Hagenberg, Austria, June 2007.

[16] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.

[17] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[18] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '06*, pages 206–217, New York, NY, USA, 2006. ACM Press.

[19] Jean H. Gallier. On the correspondence between proofs and lambda-terms. In Philippe de Groote, editor, *The Curry-Howard isomorphism*, volume 8, pages 5–138. 2003.

[20] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informatica*, 66(4):353–366, April/May 2005.

[21] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[22] Mike Gordon. From LCF to HOL: a short history. pages 169–185, Cambridge, MA, USA, 2000. MIT Press.

[23] Florian Haftmann. *Code generation from Isabelle theories.* http://isabelle.in.tum.de/doc/codegen.pdf.

[24] Florian Haftmann and Makarius Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[25] Florian Haftmann and Makarius Wenzel. Local theory specifications in isabelle/isar. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 of *Lecture Notes in Computer Science*, pages 153–168. Springer-Verlag, 2009.

[26] Cordelia Hall, Kevin Hammond, S. Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2), 1996.

[27] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[28] John Harrison. HOL Light: a tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269, 1996.

[29] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.

[30] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[31] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.

[32] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[33] Alexander Krauss. Partial recursive functions in higher-order logic. In *International Joint Conference on Automated Reasoning*, pages 589–603, 2006.

[34] P. L'Ecuyer. Efficient and portable combined random number generators. *Commun. ACM*, 31(6):742–751, 1988.

[35] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 2–12, Washington, DC, USA, 2005. IEEE Computer Society.

[36] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[37] John Matthews. Recursive function definition over coinductive types. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 73–90. Springer-Verlag, 1999.

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

[39] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[40] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[41] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*. Cambridge University Press, 1993.

[42] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[43] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 2006.

[44] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, EPFL Lausanne, Switzerland, 2004.

[45] Chris Okasaki. Catenable double-ended queues. In *In Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 66–74. ACM Press, 1997.

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

[47] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 148–161, London, UK, 1994. Springer-Verlag.

[48] Lawrence C. Paulson. Defining functions on equivalence classes. *ACM Trans. Comput. Logic*, 7(4):658–675, 2006.

[49] Microsoft Research. *The F# Language*. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.

[50] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[51] Uwe Schöning. *Theoretische Informatik – kurzgefasst.* Spektrum, March 2001.

[52] Tom Shippey. *J. R. R. T. Tolkien: author of the century.* HarperCollins, 2001.

[53] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.

[55] Tobias Nipkow Stefan Berghofer. Random testing in Isabelle/HOL. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 230–239, Washington, DC, USA, 2004. IEEE Computer Society.

[56] Gerald J. Sussman and Jr. Guy L. Steele. An interpreter for extended lambda calculus. Technical report, Cambridge, MA, USA, 1975.

[57] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.1*, July 2006. http://coq.inria.fr.

[58] G. Van Rossum. *The Python Language Reference Manual.* Network Theory Ltd., 2003.

[59] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, April 1990.

[60] 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.

[61] Tjark Weber. *SAT-based Finite Model Generation for Higher-Order Logic.* PhD thesis, April 2008.

[62] Makarius Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 307–322, London, UK, 1997. Springer-Verlag.

[63] Makarius Wenzel and Burkhart Wolff. Building formal method tools in the Isabelle/Isar framework. In Klaus Schneider and Jens Brandt, editors, *TPHOLs '07: Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, pages 352–367. Springer-Verlag, 2007.

[64] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.