Karlsruher Institut für Technologie

Open Inductive Predicates
and an Implementation in Isabelle

Masterarbeit
von

Richard Molitor

An der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Programmierparadigmen

Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter:	Dipl.-Math. Dipl.-Inform. Joachim Breitner

Bearbeitungszeit:
September 2014 – Februar 2014

Contents

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den

.....

Richard Molitor

Abstract

Theorem provers allow mechanic verification of formal proofs of mathematical theorems. An important feature for formalizing recursive structures such as programming language semantics are inductively defined predicates. The currently existing provers allow inductive specification of predicates with a fixed set of introduction rules. This thesis explores the idea of open inductive predicates which allow addition of introduction rules after definition and proofs for theorems on a per-introduction-rule basis. It also presents an implementation of the concept for the Isabelle theorem prover.

1 Introduction

Computer Languages are becoming increasingly complex. It thus becomes more and more important to verify that their definitions have the desired properties. Given their complexity this is impossible to do convincingly by reading their definition and thinking hard. The alternative is to model their semantics formally and prove mathematically that they have the claimed properties. Since these proofs include many tedious steps, instead of doing them manually, proof assistants like Isabelle are used to partially automate this task.

As an example consider the very simple language of integer arithmetic expressions consisting only of constants and additions. It can be modeled in the proof assistant Isabelle as an inductively defined datatype and an inductively defined predicate on this type that captures the evaluation semantics:

```
datatype expr =
  Const int
| Add expr expr

inductive eval::"expr ⇒ int ⇒ bool" where
  const: "eval  (Const n) n"
| add: "eval  a ra  ⟹  eval b rb  ⟹  ra + rb = n  ⟹  eval (Add a b) n"
```

Using this predicate, facts about the evaluation of these expressions can be shown. For example, the value of an expression can be doubled by doubling every constant that occurs within. To double constants, we first define an auxiliary function that operates on expressions:

```
fun cdouble where
  "cdouble (Const n) = Const (2 ∗ n)"
| "cdouble (Add a b) = Add (cdouble a) (cdouble b)"
```

Then the doubling theorem can be formulated. The proof is by induction, using the induction theorem that is defined for the eval predicate. This yields one case for constants and one for additions, both of which are easy to show[1]:

```
theorem double: "eval e n  ⟹  eval (cdouble e) (2 ∗ n)"
proof (induction rule: eval.induct)
  fix  n
  have "cdouble (Const n) = Const (2 ∗ n)" by (rule cdouble.simps(1))
  with eval.const show "eval (cdouble (Const n)) (2 ∗ n)" by simp
```

[1] In fact, the whole proof can also be found by *auto*, but the Isar version has the advantage of showing the structure of the proof. And of course, more complex proofs aren't always found by *auto* in general.

```
next
  fix a ra b rb n
  assume a: "eval (cdouble a) (2 * ra)"
    and b: "eval (cdouble b) (2 * rb)"
    and ab: "ra + rb = n"
  have "cdouble (Add a b) = Add (cdouble a) (cdouble b)" by (rule cdouble.simps(2))
  with eval.add a b ab show "eval (cdouble (Add a b)) (2 * n)" by simp
qed
```

Another theorem one might want to show is the fact that such expressions are commutative, i.e. invariant under swapping of arguments. To show this, one again needs an auxiliary function that performs the argument swapping on expressions:

```
fun swap where
  "swap (Const n) = Const n"
| "swap (Add a b) = Add (swap b) (swap a)"
```

Then again the theorem can be formulated and shown by induction using the induction theorem of the eval predicate:

```
theorem commutes: "eval e n ⟹ eval (swap e) n"
proof (induction rule: eval.induct)
  fix n
  have "swap (Const n) = Const n" by (rule swap.simps(1))
  with eval.const show "eval (swap (Const n)) n" by simp
next
  fix a ra b rb n
  assume a: "eval (swap a) ra"
    and b: "eval (swap b) rb"
    and ab: "ra + rb = n"
  have "swap (Add a b) = Add (swap b) (swap a)" by (rule swap.simps(2))
  with eval.add a b ab show "eval (swap (Add a b)) n" by simp
qed
```

1.1 Language Extension

If the language is to be extended to also allow subtractions, a new datatype and predicate can be defined:

```
datatype expr' =
  Const int
| Add expr' expr'
| Sub expr' expr'

inductive eval'::"expr' ⇒ int ⇒ bool" where
  const: "eval' (Const n) n"
```

```
| add: "eval' a ra ⟹ eval' b rb ⟹ ra + rb = n ⟹ eval' (Add a b) n"
| sub: "eval' a ra ⟹ eval' b rb ⟹ ra − rb = n ⟹ eval' (Sub a b) n"
```

The constant and addition parts are identical to the previous definition, the subtraction rule is of course new.

Like for the previous definition of arithmetic expressions, the doubling theorem is also true for evaluation of this kind of expressions. It can be formulated and proved as follows:

```
fun cdouble' where
  "cdouble' (Const n) = Const (2 ∗ n)"
| "cdouble' (Add a b) = Add (cdouble' a) (cdouble' b)"
| "cdouble' (Sub a b) = Sub (cdouble' a) (cdouble' b)"

theorem double': "eval' e n ⟹ eval' (cdouble' e) (2 ∗ n)"
proof (induction rule: eval'.induct)
  fix n
  have "cdouble' (Const n) = Const (2 ∗ n)" by (rule cdouble'.simps(1))
  with eval'.const show "eval' (cdouble' (Const n)) (2 ∗ n)" by simp
next
  fix a ra b rb n
  assume a: "eval' (cdouble' a) (2 ∗ ra)"
    and b: "eval' (cdouble' b) (2 ∗ rb)"
    and ab: "ra + rb = n"
  have "cdouble' (Add a b) = Add (cdouble' a) (cdouble' b)" by (rule cdouble'.simps(2))
  with eval'.add a b ab show "eval' (cdouble' (Add a b)) (2 ∗ n)" by simp
next
  fix a ra b rb n
  assume a: "eval' (cdouble' a) (2 ∗ ra)"
    and b: "eval' (cdouble' b) (2 ∗ rb)"
    and ab: "ra − rb = n"
  have "cdouble' (Sub a b) = Sub (cdouble' a) (cdouble' b)" by (rule cdouble'.simps(3))
  with eval'.sub a b ab show "eval' (cdouble' (Sub a b)) (2 ∗ n)" by simp
qed
```

Note that the proofs for the constant and addition cases remain unchanged. Formulating these proofs is thus very repetitive, and Copy-and-Pasting the previous proof is very tempting.

Surely there must be a better solution? Of course, one might simply extend the original eval predicate. This saves the copying and repetition, but comes with a different downside: The commutes theorem can no longer be shown! It is not true for the new language with subtractions, and there is no simple way to show it just for constants and additions, since the predicate with just these rules is no longer present.

1.2 Using Open Inductive

To alleviate this problem, a new package for Isabelle called Open Inductive was developed. It allows defining sets of introduction rules for inductive predicates in an independent fashion, using the commands **open_inductive** and **add_intro**:

open_inductive eval::"expr ⇒ int ⇒ bool"
add_intro eval const: "eval (Const n) n"
add_intro eval add: "eval a ra ⟹ eval b rb ⟹ ra + rb = n ⟹ eval (Add a b) n"
add_intro eval sub: "eval a ra ⟹ eval b rb ⟹ ra − rb = n ⟹ eval (Sub a b) n"

Inductive proofs can then be carried out separately per introduction rule. This is the doubling theorem with inductive proof for constants, additions and subtractions using the commands **open_theorem** and **show_open**:

open_theorem double **shows** "eval e n ⟹ eval (cdouble e) (2 ∗ n)"

show_open double **for** const
proof−
  fix n
  have "cdouble (Const n) = Const (2 ∗ n)" **by** (rule cdouble.simps(1))
  with eval.const **show** "eval (cdouble (Const n)) (2 ∗ n)" **by** simp
qed

show_open double **for** add
proof−
  fix a ra b rb n
  assume a: "eval (cdouble a) (2 ∗ ra)"
    and b: "eval (cdouble b) (2 ∗ rb)"
    and ab: "ra + rb = n"
  have "cdouble (Add a b) = Add (cdouble a) (cdouble b)" **by** (rule cdouble.simps(2))
  with eval.add a b ab **show** "eval (cdouble (Add a b)) (2 ∗ n)" **by** simp
qed

show_open double **for** sub
proof−
  fix a ra b rb n
  assume a: "eval (cdouble a) (2 ∗ ra)"
    and b: "eval (cdouble b) (2 ∗ rb)"
    and ab: "ra − rb = n"
  have "cdouble (Sub a b) = Sub (cdouble a) (cdouble b)" **by** (rule cdouble.simps(3))
  with eval.sub a b ab **show** "eval (cdouble (Sub a b)) (2 ∗ n)" **by** simp
qed

In Chapter 3 we explain under what circumstances such sharing of sub-proofs is possible in general.

The commutation theorem can be written analogously, but obviously proofs can only be given for constants and additions:

```
open_theorem commutes shows "eval e n ⟹ eval (swap e) n"

show_open commutes for const
proof–
  fix n
  have "swap (Const n) = Const n" by (rule swap.simps(1))
  with eval.const show "eval (swap (Const n)) n" by simp
qed

show_open commutes for add
proof–
  fix a ra b rb n
  assume a: "eval (swap a) ra"
    and b: "eval (swap b) rb"
    and ab: "ra + rb = n"
  have "swap (Add a b) = Add (swap b) (swap a)" by (rule swap.simps(2))
  with eval.add a b ab show "eval (swap (Add a b)) n" by simp
qed
```

To create the two versions of the predicate named eval and eval', once without and once with the rule for subtraction, it suffices to use the new command **close_inductive**:

```
close_inductive eval assumes const and add for eval
close_inductive eval assumes const add and sub for eval'
```

These calls automatically yield a version of the doubling theorem for each definition of eval as well as the commutation theorem for eval without subtraction. In Chapter 4 the full effects of this (and the other commands mentioned in this section) are explained in detail.

Open Inductive thus allows to keep theorems that are only valid for restricted versions of the language. At the same time it eliminates the need to copy proofs for extended variants since identical sub-proofs are shared automatically.

2 Background

This chapter provides the reader with background knowledge that helps understand where the motivation for open inductive predicates lies.

The first section explains what theorem provers are and lists some well-known implementations to show their different points of focus. Both traditional theorem provers such as Coq and Isabelle as well as provers integrated into programming languages are discussed.

Section 2.2 reviews what is understood as the expression problem in programming language design. The typical drawbacks of object-oriented as well as functional programming languages are highlighted.

The next section explains what inductively defined predicates are in a logical mathematical sense. It also shows how they can be formulated in the theorem prover Isabelle and what some common usage patterns are.

Section 2.4 discusses some work that is related to open inductive predicates and their implementation in theorem provers.

2.1 Theorem Provers

Interactive theorem provers allow the user to formulate definitions, propositions and proofs. A theorem prover implements a logic calculus and uses it to determine whether a given proof is valid. If this is the case, the prover yields a theorem, which can then be used in proofs of subsequent propositions.

Most theorem provers also have features like (semi-)automatic proof search, support for multiple logics, code generation and others.

2.1.1 Coq

Developed in France mainly by INRIA since 1984, the interactive theorem prover Coq is the most established software in the field. It has been used in the proof of the four color theorem [9] and to construct an optimizing compiler for (a subset of) the C programming language [1].

Coq is based on the higher-order type theory called "Calculus of Constructions" (CoC) [3], a higher-order typed lambda calculus. This has later been extended to the "Calculus of Inductive Constructions" [8] and "Calculus of Coinductive Constructions" [7] adding inductive types and coinduction, respectively.

The user of Coq is presented with a dependently typed functional programming language that uses this calculus as its type system. Proofs in Coq are written as series of tactics that transform the proposition into a true statement step-by-step.

Coq is implemented in OCaml (also a development of INRIA) [23], a language in the ML family.

2.1.2 Agda, Idris and ATS

Agda, Idris and ATS are experimental, dependently typed programming languages. In such languages, certain aspects of program behavior (such as lengths of arrays) can be captured statically in the type system without run-time-checks (and without the information being available at run-time). Since it cannot be in general inferred automatically at compile time if such a property holds, these languages include a proof-sublanguage used to write out proofs of program properties to aid the type checker in verifying program correctness.

Agda is being developed since 1999, first at Chalmers University of Technology. It was later rewritten, the first release of the new version was in 2007 [18]. It is a dependently typed functional programming language with a Haskell-like syntax. Agda does allow dependently typed pattern matching to make writing recursive functions (and alongside them, inductive proofs) more natural. It also features metavariables as a means of incremental program refinement. On the other hand, it does not allow writing proof scripts using tactics like Coq. Agda is implemented in Haskell.

Idris is an even younger dependently typed programming language developed at the University of St Andrews, first released in 2008 [2]. It allows both sophisticated incremental program/proof construction like Agda and traditional proofs through application of tactics. The syntax of Idris is also heavily influenced by Haskell. Like Agda, Idris is implemented in Haskell.

ATS – short for "Applied Type System" – is a descendant of the Dependent ML programming language, developed at Boston University [4]. It is a lower-level language than Agda and Idris, allowing tight integration with C, manual memory management and control over memory layout. ATS allows both functional and imperative programming, its syntax is a hybrid of C and ML. The ATS compiler is itself implemented in ATS but can be bootstrapped from C.

In all these languages, proof integrity is captured in their type system, which naturally is a part of them. The correctness of proofs thus relies on a correct implementation of the type system in the language compiler or interpreter.

2.1.3 Isabelle

The theorem prover Isabelle is a development of the University of Cambridge and the Technical University of Munich [20]. Unlike Coq it only implements a small metalogic and allows multiple client logics. The most commonly used client logic is the "Higher Order Logic" (HOL) [10], a descendant of the "Logic for Computable Functions"

(LCF) [16]. HOL allows programming in a typed lambda calculus with user-defined data constructors (datatypes), but (unlike CoC/Coq) does not have dependent types.

In addition to the classical proof scripts that consist of a list of tactics to be applied sequentially, Isabelle features a more high-level proof language called Isar, which is meant to make proofs look more natural, as they would when written on paper [30]. The goal is to show to the human reader why the proof is correct, instead of just showing (to the logic kernel) how the theorem can be proved.

Isabelle is implemented in Standard ML. The implementation uses different types for terms (the components used to formulate propositions) and theorems. Since the type system of ML has been shown to be sound [15], instances of the theorem type can only be obtained through valid proofs.

2.2 The Expression Problem

The expression problem is essentially answering the question "How to extend a datatype and its associated operations in a type-safe manner?":

> The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).
>
> Philip Wadler, 1998 [26]

Philip Wadler observed that in 1998, it was still not possible to extend both a datatype and its associated operations in a clean way neither in main-stream object-oriented nor in functional programming languages.

As an example, he considers a simple language of arithmetic expressions consisting of integer constants "Num" and addition "Add". The only supported operation is to evaluate an expression using "eval".

This language is then to be extended with a new kind of data, multiplication "Mul" and a new operation "show" which prints an expression in human-readable form. It turns out that for both functional and object-oriented languages at least one of those extensions is problematic.

2.2.1 Object-oriented Languages

The initial language can be implemented in Java as an interface "lang" with two classes "Num" and "Add" both implementing the interface.

```
package lang;

public interface T { public int eval(); };
```

```
public class Num implements T {
    protected static int val;

    public  Num(int val) { this.val = val; }

    public  int eval()  { return this.val; }
}

public class Add implements T {
    protected static T lhs;
    protected static T rhs;

    public  Add(T lhs, T rhs) {
        this.lhs    = lhs;
        this.rhs    = rhs;
    }

    public  int eval()  { return lhs.eval()  + rhs.eval();  }
}
```

Extending the language with a multiplication operation is possible without touching the existing code: One simply writes a new class "Mul" which implements the same interface.

Adding pretty-printing however is impossible without changing the existing interface and all existing classes to add the "show" method. If the existing code is to remain untouched, adding "show" entails writing a new interface "lang2" (which can conveniently inherit "eval" from "lang") and new implementations of "Num" and "Add".

```
package lang2;

public interface T extends lang.T { public String show(); }

public class Num extends lang.Num implements T {

    public  Num(int val) { super(val); }

    public  String show() { return Integer.toString(val);  }
}

public class Add extends lang.Add implements T {
    protected static T lhs;
    protected static T rhs;

    public  Add(T lhs, T rhs) {
        super(lhs,   rhs);
```

```
            this.lhs    = lhs;
            this.rhs    = rhs;
        }

        public  String show() { return lhs.show() + "␣+␣" + rhs.show(); }
}

public class Mul implements T {
        protected static T lhs;
        protected static T rhs;

        public  Mul(T lhs, T rhs) {
            this.lhs    = lhs;
            this.rhs    = rhs;
        }

        public  int eval()  { return lhs.eval()  * rhs.eval();   }

        public  String show() { return lhs.show() + "␣*␣" + rhs.show(); }
}
```

Thanks to inheritance, re-implementation of "eval" is not necessary for the existing classes. The constructors need to be adapted to the new types however, as can be seen in the "Add" class.

Note carefully that both the call to "super()" as well as the manual initialization of "lhs" and "rhs" are necessary in the constructor of "Add". Leaving out the call to the parent constructor results in a compile-time error, while not re-initializing the variables lead to the beloved "NullPointerException" when calling "show()" on the lang.T-initialized variables.

2.2.2 Functional Languages

In Standard ML, the initial language can be implemented using an inductive datatype with a "Num" and an "Add" constructor and a free function "eval":

```
signature LANG =
sig
  type t

  val Num: int → t
  val Add: {rhs: t,  lhs:  t}  → t

  val eval:  t  → int
end
```

```
structure Lang :> LANG =
struct
  datatype t =
    Num of int
  | Add of {rhs: t,  lhs:  t}

  fun eval (Num i) = i
    | eval (Add {lhs, rhs}) = (eval lhs)  + (eval rhs)
end
```

When extending this language, it is possible to add the pretty-printing by simply adding a "show" function without changing any existing code[1]:

Adding multiplication however would entail changing the "lang" datatype and the existing "eval" function, so to cleanly add it, a new structure becomes necessary:

```
signature LANG2 =
sig
  include LANG

  val Mul: {rhs: t,  lhs:  t} → t

  val show: t → string
end

structure Lang2 :> LANG2 =
struct
  datatype t =
    Num of int
  | Add of {rhs: t,  lhs:  t}
  | Mul of {rhs: t,  lhs:  t}

  fun eval (Num v) = Lang.eval (Lang.Num v)
    | eval (Add {lhs, rhs}) = Lang.eval (Lang.Add
        {lhs = Lang.Num (eval lhs),
         rhs = Lang.Num (eval rhs)})
    | eval (Mul {lhs, rhs}) = (eval lhs) * (eval  rhs)

  fun show (Num i) = Int.toString i
    | show (Add {lhs, rhs}) = (show lhs) ^ " + " ^ (show rhs)
    | show (Mul {lhs, rhs}) = (show lhs) ^ " * " ^ (show rhs)
end
```

[1] It is necessary to add "show" to the signature and the structure, but in the Java case, the package signature would also change. The point is that the datatype definition of "t" and the "eval" function need not be touched.

The signature can again be extended from the first "LANG". The datatype and its constructor need to be implemented again, though. To reuse the function "Lang.eval" in the implementation of "Lang2.eval" some ugly wrapping is necessary. The inheritance in Java makes this wrapping look a lot cleaner—on the other hand there is no way to accidentally produce the "NullPointerException" in Standard ML.

2.2.3 Summary

As demonstrated, object-oriented and functional programming languages traditionally solve different aspects of the expression problem. They do a good job solving one part of the problem and make the other awkward to implement.

In an object-oriented language like Java, it is easy to add new types (just add new classes), but hard to add operations (break an existing interface?). In a functional language like Standard ML, it is easy to add new operations (just add new functions), but hard to add types (break an existing datatype?).

Neither discipline solves both aspects in a clean way. It needs to be noted however, that both Java and Standard ML are rather old languages (by computer science standards anyway) and modern "hybrid" languages like Scala [25] or Rust [24] have better answers to the expression problem.

2.3 Inductively Defined Predicates

Predicates are functions with a boolean result, so a predicate evaluates its arguments and returns true or false [22]. For some predicates it is convenient to define them inductively, i.e. so that they refer to themselves in their definition.

2.3.1 In Higher-Order Logic

An example of an inductively defined predicate is the "eval" function from the previous section written as a predicate.

We first define the datatype of an arithmetic expression that consists of a constant (Const) or an addition (Add) inductively:

expr = Const int $\vee$ Add expr expr

Here "int" refers to an integer. Assuming we have also defined + as an integer operation, the eval function can then be defined as an inductive predicate that satisfies the following two "introduction rules":

- $\forall n.\ \text{eval}\ (\text{Const}\ n)\ n$
- $\forall a\ r_a\ b\ r_b\ r.\ \text{eval}\ a\ r_a\ \wedge\ \mathit{eval}\ b\ r_b\ \wedge\ r_a + r_b = r \rightarrow\ \text{eval}\ (\text{Add}\ a\ b)\ r$

That is, any constant evaluates to itself, and an addition evaluates to the result of evaluating its left and right argument and adding them. The induction theorem for this predicate is:

$$\forall P.\ \frac{\forall n.\ P\ (\text{Const}\ n)\ n \qquad \forall a\ r_a\ b\ r_b\ n.\ P\ a\ r_a\ \wedge\ P\ b\ r_b\ \wedge\ r_a + r_b = n \rightarrow P\ (\text{Add}\ a\ b)\ n}{\forall e\ n.\ \text{eval}\ e\ n \rightarrow P\ e\ n}$$

To put it differently, we want eval to be the smallest predicate that fulfills its introduction rules, so they are the only facts that we can assume for the induction theorem. This rule is also called the axiom of induction. The reason for calling it an axiom is that it cannot be proved in some logics, notably first-order-logic where it is illegal to quantify over P. When induction is necessary in such logics, the induction rules need to be defined as axioms.

Of course, eval could just as well be written as a function, using lambda terms. It is used here as an example since it is easy to understand and to highlight the parallels with programming languages reviewed in the previous section. In general not all predicates can be written as functions. A more complex example would distract more attention from the question how predicates can be defined inductively.

2.3.2 In Isabelle/HOL

In the Isabelle theorem prover, inductive predicates can be defined on any built-in or user-defined datatype as both predicates or sets. The syntax is very close to the logic notation shown above.

To define the inductive datatype of arithmetic expressions in Isabelle/HOL we write:

```
datatype expr =
  Const int
| Add expr expr
```

The predicate eval is then defined inductively by specifying its introduction rules:

```
inductive eval::"expr ⇒ int ⇒ bool" where
  const: "eval  (Const n) n"
| add: "eval  a ra  ⟹  eval b rb  ⟹  ra + rb = n  ⟹  eval (Add a b) n"
```

Compared to our logical definition earlier, all the universal quantifications have disappeared. This is because in Isabelle, every free variable is implicitly universally quantified. Using this input, the Inductive package produces a definition that guarantees the properties we expect:

$$\begin{aligned}\text{eval} \equiv \text{lfp}\ (\lambda P\ x_1\ x_2.(\exists n.\ x_1 = \text{Const}\ n \wedge x_2 = n) \vee \\ (\exists a\ r_a\ b\ r_b\ n.\ x_1 = \text{Add}\ a\ b \wedge x_2 = n \wedge P\ a\ r_a \wedge P\ b\ r_b \wedge r_a + r_b = n))\end{aligned}$$

While this definition is a bit complex, it is possible to see where the introduction rules are integrated. All of this is a λ-function wrapped in "lfp". lfp stands for least fixed point, which ensures that we really get the *smallest* predicate that fulfills the introduction rules. The definition of lfp in Isabelle simply uses the set-theoretic infimum:

$$\text{lfp}\ f = \text{Inf}\{u.\ f\ u \leq u\}$$

Isabelle automatically proves rules about this new predicate. The most obvious one is the induction theorem, which, with HOL being a higher order logic, can actually be proved from the definition above:

$$\begin{array}{l}\text{eval}\, x_1\, x_2 \implies \\ \quad (\bigwedge n.\ P\ (\text{Const}\, n)\ n) \implies \\ \quad (\bigwedge a\ r_a\ b\ r_b\ n. \\ \qquad \text{eval}\, a\ r_a \implies P\ a\ r_a \implies \\ \qquad \text{eval}\, b\ r_b \implies P\ b\ r_b \implies \\ \qquad r_a + r_b = n \implies P\ (\text{Add}\, a\ b)\ n) \implies \\ \quad P\ x_1\ x_2\end{array}$$

Here, the logic operators like "$\wedge$" have been replaced by meta-logic connectors that make the rule more convenient to use in proofs. In Isabelle, the "$\bigwedge$"-symbol is the meta-logic universal quantification ($\forall$), and the "$\implies$"-symbol is the meta-logic implication ($\rightarrow$).

The call to inductive also shows the cases rule:

$$\begin{array}{l}\text{eval}\, x_1\, x_2 \implies \\ \quad (\bigwedge n.\ x_1 = \text{Const}\, n \implies x_2 = n \implies P) \implies \\ \quad (\bigwedge a\ r_a\ b\ r_b\ n. \\ \qquad x_1 = \text{Add}\, a\ n \implies x_2 = n \implies \\ \qquad \text{eval}\, a\ r_a \implies \text{eval}\, b\ r_b \implies r_a + r_b = n \implies P) \implies \\ \quad P\end{array}$$

This is used for case analysis where no induction is necessary, therefore P is not recursive in this rule. Since this is the opposite of using the introduction rules, this strategy is also called rule inversion [17, p. 130].

Finally, it proves the introduction rules, which are used to show which terms fulfill the predicate:

- $\text{eval}\ (\text{Const}\, n)\ n$
- $\text{eval}\, a\ r_a \implies \text{eval}\, b\ r_b \implies r_a + r_b = n \implies \text{eval}\ (\text{Add}\, a\ b)\ n$

These are identical to the definition, which is no surprise since the predicate is defined in terms of its introduction rules.

What Isabelle/HOL provides is thus a convenient way to formulate inductive predicates by giving their introduction rules, and a set of automatic transformations to prove a number of useful and often-needed rules to work with them.

2.4 Related Work

This section provides an overview of how inductive predicates and datatypes can be implemented in a formal manner, to be used in a theorem prover.

Just like in the case of object-oriented and functional programming, one can ask the question, how such an inductive predicate and its datatype can be extended. The second part of this section addresses this question.

2.4.1 Implementations of Inductive Definitions

There are several publications on how to implement inductive predicates in theorem provers.

Isabelle. The approach used in Isabelle is described in "A fixedpoint approach to implementing (co)inductive definitions" by Paulson [19]. This describes the process of modeling the inductive predicate from its introduction rules, which was mentioned in the previous section. It also explains the origin of the least-fixed-point operator "lfp" mentioned earlier, and the type of functions on which it can be used.

Coq. Another mechanism for implementing inductive predicates is the Calculus of Co-Inductive Constructions, an extension of the Calculus of Constructions by Giménez [8]. This is, as the name implies, the variant implemented in Coq.

HOL. A third mechanism by Melham is based on the natural numbers [14]. In this approach, lists are defined as functions over the natural numbers and trees are defined using Gödel numbering. In essence, the idea is to have one inductive data structure, the natural numbers, and use it to construct others as necessary. This strategy is used in the HOL theorem prover.

Others. Other, earlier theorem provers have no mechanism to handle inductive definitions without introducing new axioms. This approach is of course fragile, since it does bear the risk of introducing unsoundness.

Paulson writes in [19] that to introduce inductive definitions in a definitional fashion, either higher-order logic or Zermelo-Fraenkel set theory is necessary. For this, early theorem provers like LCF, which is a first-order logic, were simply not powerful enough, so other ways had to be found.

2.4.2 Extensions of Inductive Definitions

Inductive predicates are often used in the context of modeling operational semantics of programming languages. In this context, it is often helpful to first define a small version of a language and then extend it later. Thus, inductive predicates are informally extended in many papers on programming language semantics. For example, in his paper "Natural Semantics for Lazy Evaluation" Launchbury first defines a minimal functional language and then extends it with data constructors [13]. He does not re-prove the theorems about the minimal language again for the extended version—and in the context of the paper such a proof would certainly be out of place.

It does however beg the question, under which circumstances proofs remain correct for such an extension.

In "Production Lines of Theorems" Delaware et al. [5] model the operational semantics of Featherweight Java in Coq and show how to extend it with casts and generics. Their approach requires a fair amount of manual wrapping of predicates and proofs,

similar to the wrapping of the eval function in Section 2.2.2. Compared to the Open Inductive approach presented here, this requires more manual work. Conversely, by relying on wrapping by the user it is of course more flexible than the automated solution implemented in Open Inductive.

A different approach by Jaskelioff et al. is to replace inductive definitions by free monads [11]. Several such definitions can then be combined using monadic coproducts. The authors use Haskell to implement their semantics framework. Being implemented in Haskell the focus of the authors lies on implementing interpreters that adhere to these semantics. There is no way to formally prove properties about the defined languages in this approach.

3 Open Inductive Predicates

Inductively defined predicates with a fixed set of introduction rules are useful and sufficient for well-understood concepts that are unlikely to change. They are also useful in formalizing the evaluation semantics of formal languages. In this case, it is often useful to have several closely related variants of a language. This yields multiple inductive definitions with overlapping sets of introduction rules and multiple induction theorems.

Maintaining proofs for each variant separately is cumbersome and error-prone. This chapter explores the idea of modularizing such definitions and proofs in a way that allows their reuse.

3.1 Motivation

As a motivating example consider the evaluation of arithmetic expressions. With a datatype for expressions and the eval predicate from Section 2.3.1 and its induction theorem

Definition 1. *Induction theorem for eval:*

$$\forall P.\ \frac{\forall n.\ P\ (\text{Const}\ n)\ n \qquad \forall a\ r_a\ b\ r_b\ n.\ P\ a\ r_a\ \wedge\ P\ b\ r_b\ \wedge\ r_a + r_b = n \rightarrow P\ (\text{Add}\ a\ b)\ n}{\forall e\ n.\ \text{eval}\ e\ n \rightarrow P\ e\ n}$$

it is possible to prove theorems about eval by induction. To deconstruct the inductive cases, we need the introduction rules for constants and additions:

Definition 2. *Constant introduction for eval:*

$$\forall n.\ \frac{}{\text{eval}\ (\text{Const}\ n)\ n}$$

Definition 3. *Addition introduction for eval:*

$$\forall a\ r_a\ b\ r_b\ n.\ \frac{\text{eval}\ a\ r_a \qquad \text{eval}\ b\ r_b \qquad r_a + r_b = n}{\text{eval}\ (\text{Add}\ a\ b)\ n}$$

For example, the value of an expression can be doubled by doubling all constants. To show this, we first define an auxiliary function that performs the doubling of constants:

Definition 4. *Constant doubling:*

$$\text{cdbl}\ e \equiv \begin{cases} \text{Const}\ (2 * n) & \text{if } e = \text{Const}\ n \\ \text{Add}\ (\text{cdbl}\ a)\ (\text{cdbl}\ b) & \text{if } e = \text{Add}\ a\ b \end{cases}$$

Theorem 1. *The value of an expression can be doubled by doubling every constant:*

$$\text{eval}\, e\, n \to \text{eval}\, (\text{cdbl}\, e)\, (2 * n)$$

Proof. The proof is again by induction on e. Applying the induction theorem from Definition 1 yields two cases that need to be proved:

Constant case:

$$\forall n.\, \text{eval}\, (\text{cdbl}\, (\text{Const}\, n))\, (2 * n)$$

Addition case:

$$\forall a\ r_a\, b\, r_b\, n.\, \text{eval}\, (\text{cdbl}\, a)\, (2 * r_a) \wedge \text{eval}\, (\text{cdbl}\, b)\, (2 * r_b) \wedge r_a + r_b = n \to \text{eval}\, (\text{cdbl}\, (\text{Add}\, a\, b))\, (2 * n)$$

Constant case. We first show the case for constants. The application of cdbl can be simplified using Definition 4, yielding:

$$\forall n.\, \text{eval}\, (\text{Const}\, (2 * n))\, (2 * n)$$

This matches Definition 2, the introduction rule for constants, completing the proof for the constant case.

Addition case. To show the case for addition we simplify again with the definition of cdbl. This changes the right-hand-side of the implication:

$$\forall a\ r_a\, b\, r_b\, n.\, \text{eval}\, (\text{cdbl}\, a)\, (2 * r_a) \wedge \text{eval}\, (\text{cdbl}\, b)\, (2 * r_b) \wedge r_a + r_b = n \to \text{eval}\, (\text{Add}\, (\text{cdbl}\, a)\, (\text{cdbl}\, b))\, (2 * n)$$

This form now matches Definition 3, the introduction rule for addition. □

To extend the eval predicate with a new introduction rule for subtraction, the definitions need to be adapted. A new introduction rule is necessary for subtractions:

Definition 5. *Subtraction introduction for eval':*

$$\forall a\, b\, n\, r_a\, r_b.\ \frac{\text{eval}'\, a\, r_a \qquad \text{eval}'\, b\, r_b \qquad r_a - r_b = n}{\text{eval}'\, (\text{Sub}\, a\, b)\, n}$$

The introduction rules for constants and additions can remain as before, only eval needs to be replaced with eval'. The induction theorem now has three premises instead of two:

Definition 6. *Induction theorem for eval':*

$$\forall P.\ \frac{\begin{array}{cc}\forall n.\, P\, (\text{Const}\, n)\, n & \forall a\, r_a\, b\, r_b\, n.\, P\, a\, r_a \wedge P\, b\, r_b \wedge r_a + r_b = n \to P\, (\text{Add}\, a\, b)\, n \\ \multicolumn{2}{c}{\forall a\, r_a\, b\, r_b\, n.\, P\, a\, r_a \wedge P\, b\, r_b \wedge r_a - r_b = n \to P\, (\text{Sub}\, a\, b)\, n}\end{array}}{\forall e\, n.\, \text{eval}'\, e\, n \to P\, e\, n}$$

To show the doubling theorem for eval' a new proof must be given. To formulate the proposition, the doubling function needs to be adapted first:

Definition 7. *Constant doubling for eval':*

$$\text{cdbl}'\ e \equiv \begin{cases} \text{Const}\ (2 * n) & \text{if } e = \text{Const}\ n \\ \text{Add}\ (\text{cdbl}'\ a)\ (\text{cdbl}'\ b) & \text{if } e = \text{Add}\ a\ b \\ \text{Sub}\ (\text{cdbl}'\ a)\ (\text{cdbl}'\ b) & \text{if } e = \text{Sub}\ a\ b \end{cases}$$

Then the equivalent of Theorem 1 can be shown for eval':

Theorem 2. *The value of an expression can be doubled by doubling every constant:*

$$\text{eval}'\ e\ n \rightarrow \text{eval}'\ (\text{cdbl}'\ e)\ (2 * n)$$

Proof. The proof is, as before, by induction on e. Applying the induction theorem now yields three cases:

Constant case:

$$\forall n.\ \text{eval}'\ (\text{cdbl}'\ (\text{Const}\ n))\ (2 * n)$$

Addition case:

$$\forall a\ r_a\ b\ r_b\ n.\ \text{eval}'\ (\text{cdbl}'\ a)\ (2 * r_a) \wedge \text{eval}'\ (\text{cdbl}'\ b)\ (2 * r_b) \wedge r_a + r_b = n \rightarrow \text{eval}'\ (\text{cdbl}'\ (\text{Add}\ a\ b))\ (2 * n)$$

Subtraction case:

$$\forall a\ r_a\ b\ r_b\ n.\ \text{eval}'\ (\text{cdbl}'\ a)\ (2 * r_a) \wedge \text{eval}'\ (\text{cdbl}'\ b)\ (2 * r_b) \wedge r_a - r_b = n \rightarrow \text{eval}'\ (\text{cdbl}'\ (\text{Sub}\ a\ b))\ (2 * n)$$

Constant and Addition cases. The proofs for constants and additions are analogous to the proof of Theorem 1.

Subtraction case. The proof for subtraction first requires simplifying the occurrence of the constant doubling function:

$$\forall a\ r_a\ b\ r_b\ n.\ \text{eval}'\ (\text{cdbl}'\ a)\ (2 * r_a) \wedge \text{eval}'\ (\text{cdbl}'\ b)\ (2 * r_b) \wedge r_a - r_b = n \rightarrow \text{eval}'\ (\text{Sub}\ (\text{cdbl}'\ a)\ (\text{cdbl}'\ b))\ (2 * n)$$

The introduction rule for subtraction given in Definition 7 matches this form, concluding the proof. □

Since the premises for constants and addition in the induction theorem of eval are unchanged in the induction theorem of eval' with subtraction, the same sequence of rule applications can be used to prove these cases. The result is two very similar proofs – after the application of the induction theorem the two cases for constants and additions can be shown analogously.

The remainder of this chapter discusses the question when such sharing of subproofs is correct.

3.2 Inductive Predicate Definitions

Let I be an index set and R a set of formulas of the form

$$R_i \equiv \forall x.\, F_i[P, \vec{g_i}, x] \to P\,(f_i\,x) \quad \text{for } i \in I$$

where P is a predicate symbol of one variable, each F_i is a formula monotonous in P and a variable x. The f_i and $\vec{g_i}$ are functions used for transforming the variable before passing it to P.

This set R then describes the introduction rules for a predicate P with the induction theorem:

$$\forall Q.\, \frac{\forall x.\, F_i[P, Q \cdot \vec{g_i}, x] \to Q\,(f_i\,x) \quad \text{for } i \in I}{P\,x \to Q\,x}$$

here the f_i and $\vec{g_i}$ are again functions that are applied to the argument before passing it to the predicate P. Q is any formula, it stands for the proposition that is to be shown with the induction theorem. The notation $Q \cdot \vec{g_i}$ means that both the transformations of Q as well as $\vec{g_i}$ need to be applied to the arguments of P in the inductive case.

How to generate the actual definition that can be used to prove these rules is described by Paulson in [19].

In the example above, P is eval, the f and $\vec{g}$ are used to make the constructor calls to Const and Add. For the constant introduction rule, F is empty, for the others it contains the inductive calls to eval and the side-conditions like $r_a + r_b = n$. In Theorem 1 the Q makes the call do cdbl and produces the $2 * n$ from n.

3.3 Modularizing Proofs

A proof that uses the induction theorem of P as first rule then has one case for each of the introduction rules in R:

case for R_i for $i \in I$

A subset $J \subseteq I$ can now be used to select a subset of the introduction rules R, using the substitution $R[P/P']$ for a predicate P'. The induction theorem of this predicate P' is then of the form:

$$\forall Q.\, \frac{\forall x.\, F_j[P', Q \cdot \vec{g_j}, x] \to Q\,(f_j\,x) \text{ for } j \in J \subseteq I}{P'\,x \to Q\,x} \tag{3.1}$$

A proof for a theorem on P' that uses induction as its primary proof method then has cases of the form:

case for R_j for $j \in J \subseteq I$

Provided the proof for each introduction rule is independent of the others, the proof for the theorem can reuse the exact same proofs as with the introduction theorem of the larger predicate.

In terms of the example, if you have a proof for Theorem 1 for a predicate that includes evaluation of constants, addition and subtraction, you can use the subproofs for the constant and addition cases to show the same theorem for a predicate that just allows evaluation of these two constructs, Theorem 2 above, after substituting eval with eval'.

3.4 Incremental Development

This only explains how the proof of a theorem for the largest desired predicate can be split apart to find proofs for smaller ones with subsets of the induction rules. In development, the largest predicate is usually an extension that is developed after proofs for simpler versions have already been found.

Instead of assuming a large induction theorem with cases for many different introduction rules, it is simpler to actually assume the smallest possible induction theorem for each introduction rule.

Often, each subproof can be written using a simpler induction theorem of the form:

$$\forall P\ Q.\ \frac{\forall x.\ F_i[P, Q \cdot \vec{g_i}, x] \to Q\ (f_i\ x)}{P\ x \to Q\ x} \qquad \text{with } i \in I \tag{3.2}$$

Note that here P is universally quantified. That means the definition of no specific P can be used in the proof. A proof found after application of Equation 3.2 instead of Equation 3.1 is thus general for all P that have an induction theorem that produces this specific case, i.e. predicates with the introduction rule R_i.

This approach works for the proof of doubling by constant doubling (Theorems 1 and 2 above) since the proofs for the cases are self-sufficient and can be instantiated for eval and eval'.

3.5 Additional Dependencies

Since the definition of no specific predicate P is usable in the proof, this method does not allow the proof of cases that rely on using other facts about the predicate in question. Simply making the definition available is not a solution, since the proof is then not generally instantiatable for different P.

Instead the idea is to make specific aspects of P available during the proof, namely introduction rules. If all subproofs in I need access to a selection of introduction rules R_k with $k \in K \subseteq I$, proof of each rule can be given by:

R_k **for** $k \in K; R_i \vdash$ **case for** R_i for $i \in I$

Here, the rules to the left of the turnstile ($\vdash$) symbol refer to rules present in the context of the proof to the right. In every context where these rules hold, the proof to the right holds true as well.

The context needed here includes the introduction rules selected by K. Additionally, the introduction rule R_i for the case at hand needs to be made available, too. This latter addition is not a real restriction though, because it will always be present in the context where the proof for R_i is needed. The case for R_i on the right-hand-side is again meant to be generated by the generalized induction theorem (Equation 3.2).

Proofs of this form can be then used to assemble valid proofs for any predicate P with induction theorems of the form:

$$\forall Q.\ \frac{\forall x.\ F_j[P_0, Q \cdot \vec{g_j}, x] \to Q\ (f_j\ x) \text{ for } j \in K \subseteq J \subseteq I}{P_0\ x \to Q\ x} \tag{3.3}$$

This is again a regular induction theorem for a defined predicate P_0, not the version universally quantified for P. The predicate P_0 here needs to include at least the first introduction rules in K, since they are dependencies in proofs of the others. To use the proofs for the cases that were produced using Equation 3.1, the general P now needs to be instantiated for the special P_0 that is now defined. Then the proof for each case can be applied. The context for these is as required, since the definition of P_0 allows proof of the introduction rules for K.

3.6 Non-inductive Proofs

Until this point, we have discussed proofs where the primary proof method is induction, i.e. the first rule application to the proposition is the induction theorem.

This is not a necessary requirement: The principle of generalizing for all P can be applied for non-inductive proofs as well. To show a proposition Q for a generalized predicate, it suffices to find a set of necessary introduction rules (again indexed by K), make them available in the context and generalize all occurrences for P:

R_k **for** $k \in K \subseteq I \vdash$ **proof for** $\forall P.\ Q$

This time the proof is direct and not per-introduction rule, so using it as proof for a defined predicate P_0 with at least the introduction rules specified by K is even simpler: Just instantiate P with P_0.

3.7 Limitations

The method described in this chapter has some limitations that make it applicable only to certain kinds of proofs. Since the inductive predicate is not defined in the proof context, only facts about it that are made available explicitly can be used. In this chapter we have shown how it is possible to make selected introduction rules available in the proof context.

Another often-needed rule in proofs is the rule for case analysis[1]. While in principle this rule could also be made available in the context, this would severely restrict the generality of the proof: The proposition of the cases rule is that only certain cases of syntactic forms are part of the predicate. By making a cases rule with a set R of cases available in a per-introduction-rule proof, this precludes closing the predicate later for any real superset $R' \supset R$ since the new cases are not accounted for in the cases theorem. When closing the predicate later, it must only contain a subset of the rules specified by the case analysis, i.e. $\tilde{R} \subseteq R$.

This example shows that care needs to be taken when choosing which rules to make available during the general proof, since it is easy to accidentally over-specify the predicate and lose genericity.

[1] For example in proofs for determinism

4 User Interface

This chapter describes the user interface of the Open Inductive package that was implemented for Isabelle. It is compatible with the Isabelle 2014 stable release as well as the repository versions up to and including changeset `c85e018be3a3`. The package depends on theory Main of Isabelle/HOL. To use it, it needs to be imported at the top of the theory.

The first part of this chapter describes the commands that the package defines. The second part shows them in action: They are used to define and prove the evaluation of arithmetic expressions and some theorems about them that were discussed previously.

4.1 Registering Predicates

Defining an inductive predicate is a two-step process. First, the name of the predicate needs to be registered. Then one or more defining introduction rules can be associated with it.

Predicate Registration. The command to register a new inductive predicate in the current theory is **open_inductive**. Its syntax is:

Command 1. ***open_inductive*** *pred-name[::type]*

Here, *pred-name* can be any valid identifier. The type annotation is optional. Successfully registering a predicate will result in a diagnostic message:

Message 1. Two variants of this message exist, depending on whether a type annotation was given:

- *Registered open inductive predicate* name *without type.*
- *Registered open inductive predicate* name *with type* type.

It is legal to register the same predicate multiple times. This can be used when a predicate has some introduction rules defined in one file and is to be extended in another to remind the reader of its presence. If a type was specified, it cannot be changed later however. This leads to an error with the message:

Error 1. *Can't re-type predicate, was* old-type *would become* new-type.

Also, if a type was specified the predicate cannot be registered later without a type. This leads to an error with the message:

Error 2. *Can't delete type from predicate, was* type.

Introduction Rules. The command to associate an introduction rule with an open predicate is **add_intro**, its syntax is:

Command 2. ***add_intro*** *pred-name intro-name****:*** *term*

Here, *pred-name* refers to a previously registered predicate, *intro-name* is a new name for the introduction rule and *term* is an inner syntax expression containing the introduction rule.

Successful registration of an introduction rules results in a diagnostic message:

Message 2. *Registered introduction rule* rule-name*:* term *for* pred-name

The predicate that **add_intro** refers to as *pred-name* needs to be registered beforehand using Command 1. Using **add_intro** with an unregistered predicate results in an error with the message:

Error 3. *No such open inductive predicate:* pred-name

The defining term is checked for syntax and internal consistency by the parser. The occurrences of the predicate in the rule are also checked to match the type of the predicate at the point of registration (if specified). Additionally they are checked structurally by **add_intro**, so it is impossible to add a rule that cannot be used as introduction rule (i.e. when the conclusion does not contain the predicate).

4.2 Registering Theorems

Using an open inductive predicate, theorems can be proved by induction with a separate proof for each introduction rule. Like predicates, theorems need to be registered as open theorems first. This is accomplished by the command **open_theorem**, its syntax is:

Command 3. ***open_theorem*** *theorem-name* ***shows*** *term*

Here, *theorem-name* is the name with which the theorem is later installed in the theory, *term* is the proposition written in as an inner syntax term.

There is no need to explicitly specify which open inductive predicates occur in the theorem. The package parses the term, finds them and reports them back to the user. The message after successful registration is:

Message 3. *Declared open theorem* theorem-name *as* term *on* predicate-name(s).

Here, *predicate-name(s)* is the list of open inductive predicates that are found in the theorem. Again, all occurring predicates are checked to be compatible with their annotated type (if specified).

4.3 Proving Theorems

A registered open theorem can now be proved by induction. The case for each introduction rule is a separate proof. The command to begin a proof is **show_open**, its syntax is:

Command 4. ***show_open*** *theorem-name* ***for*** *intro-name [****assumes*** *intro-name(s) . . .]*

Here, *theorem-name* refers to the theorem that is to be proved and *intro-name* refers to the name of the introduction rule for which the inductive case is to be instantiated. The optional list of *intro-name(s)* are names of additional introduction rules that shall be added to the proof context as assumptions. At this point, the type of the predicate is matched with its inferred type in the introduction rule. If this fails a type unification error is printed.

The theorem needs to be registered first with Command 3. Referring to an unregistered theorem leads to an error with the message:

Error 4. *No such theorem:* theorem-name

Likewise, the introduction rule needs to be added to the predicate occurring in the proposition using Command 2. Referring to an unregistered theorem leads to an error with the message:

Error 5. *No introduction rule named* intro-name *in pred-name(s)*

If no error occurs, Isabelle switches to proof mode and the theorem can now be proved for the selected introduction rule. The proof context is enriched with the selected introduction rule(s), they are named *pred-name.intro-name*, just as they would be when produced by the Inductive package.

Following a successful proof no theorem is installed in the current theory. The package saves the proof internally and only exports the complete theorems, not the case for each introduction rule.

Non-inductive proofs. It also possible to prove theorems that do not require induction. In this case the invocation is:

Command 5. ***show_open*** *theorem-name [****assumes*** *intro-name(s) . . .]*

No induction theorem is applied to the proposition. As with inductive proofs, the proof context is enriched with the selected introduction rules, registered with their usual names.

4.4 Closing Predicates

To get theory-level definitions of the predicates and theorems with proofs assembled from the cases, the **close_inductive** command is used. Its syntax is:

Command 6. ***close_inductive*** *pred-name* ***assumes*** *intro-name [****and*** *intro-name . . .]* ***for*** *close-name*

The predicate named *pred-name* will be created with the listed introduction rules with the theory-level name *close-name*. Here, *close-name* can be identical to *pred-name*.

The predicate referred to by *pred-name* needs to be registered using Command 1. Referring to an unregistered predicate leads to an error with the message:

Error 6. *Undefined open predicate:* pred-name

Likewise, each of the rules listed as *intro-name* needs to be registered in the correct predicate using Command 2. Referring to an introduction rule not added to the predicate leads to an error with the message:

Error 7. *No introduction rule name* intro-name *defined in open predicate* pred-name

If no error occurs, a message is printed:

Message 4. *Closing inductive predicate* pred-name *with* intro-name(s) *as* close-name. *Candidates for closing:* theorem-name(s)

The candidates for closing are open theorems (registered with Command 3) that contain the specified predicate. If a direct proof given using Command 5 exists for a theorem, this proof is preferred. If such a proof cannot be used because of introduction rules that are assumed but not present in the close predicate, a warning is printed:

Message 5. *Cannot close open theorem* theorem-name*, missing introduction rules:* intro-name(s)

If no direct proof exists, the inductive proofs are checked for completeness. If any of these theorems cannot be closed because of missing proofs (given using Command 4), a warning is printed:

Message 6. *Cannot close open theorem* theorem-name*, missing proofs for* intro-name(s)

If all inductive proofs exist, but contain assumptions that cannot be satisfied, Message 5 is printed here, too. Open theorems for which a complete proof can be assembled are registered in the theory by their names. For successful registration a message is printed:

Message 7. *Installing* name*:* term

Here, *term* is the (now proved) proposition. The name is of the form *theorem-name_pred-name* with *theorem-name* being the name of the registered open theorem and *close-name* the name of the predicate chosen with **close_inductive**. This is to avoid name clashes when closing multiple variants of the same predicate that have proofs for the same theorems.

4.5 Example: Evaluation of Arithmetic Expressions

This section shows the interaction of these commands and how they can be used to modularize inductive proofs on our earlier example, the evaluation of arithmetic expressions.

4.5.1 Definitions

The definition of the datatype for an expression is unchanged, using the **datatype** command. Here, an expression is defined with all the constructors that will be used later, even though not all of them will occur in every predicate.

```
datatype expr =
  Const int
| Add expr expr
| Sub expr expr
```

The definition of the *eval* predicate is now done with **open_inductive**, using the **add_intro** command to specify three introduction rules corresponding to the datatype constructors.

```
open_inductive eval::"expr ⇒ int ⇒ bool"
add_intro eval const: "eval (Const n) n"
add_intro eval add: "eval a ra ⟹ eval b rb ⟹ ra + rb = n ⟹ eval (Add a b) n"
add_intro eval sub: "eval a ra ⟹ eval b rb ⟹ ra − rb = n ⟹ eval (Sub a b) n"
```

4.5.2 Theorem: Double

One way to double the value of an expression is to add that same expression to itself.

```
open_theorem double_add shows "eval e n ⟹ eval (Add e e) (2 ∗ n)"

show_open double_add assumes add
proof (rule eval.add, simp_all)
  show "n + n = 2 ∗ n" by presburger
qed
```

Note that this proof is not by induction, so it uses the bare version of **show_open**. The proof needs access to the introduction rule for addition.

Another way is to double every constant. For this we define a function *cdouble* that accepts an expression and recursively replaces all constants by their doubled equivalents. This theorem can be shown inductively for all three introduction rules.

```
fun cdouble where
  "cdouble (Const n) = Const (2 ∗ n)"
| "cdouble (Add a b) = Add (cdouble a) (cdouble b)"
| "cdouble (Sub a b) = Sub (cdouble a) (cdouble b)"

open_theorem double shows "eval e n ⟹ eval (cdouble e) (2 ∗ n)"
```

```
show_open double for const
proof–
  fix n
  have "cdouble (Const n) = Const (2 ∗ n)" by (rule cdouble.simps(1))
  with eval.const show "eval (cdouble (Const n)) (2 ∗ n)"  by simp
qed

show_open double for add
proof–
  fix a ra b rb n
  assume a: "eval (cdouble a) (2 ∗ ra)"
    and b: "eval (cdouble b) (2 ∗ rb)"
    and ab: "ra + rb = n"
  have "cdouble (Add a b) = Add (cdouble a) (cdouble b)" by (rule cdouble.simps(2))
  with eval.add a b ab show "eval (cdouble (Add a b)) (2 ∗ n)" by simp
qed

show_open double for sub
proof–
  fix a ra b rb n
  assume a: "eval (cdouble a) (2 ∗ ra)"
    and b: "eval (cdouble b) (2 ∗ rb)"
    and ab: "ra − rb = n"
  have "cdouble (Sub a b) = Sub (cdouble a) (cdouble b)" by (rule cdouble.simps(3))
  with eval.sub a b ab show "eval (cdouble (Sub a b)) (2 ∗ n)" by simp
qed
```

The proofs themselves are simple and analog to the ones given for the equivalent theorem in the previous chapter. They do not need access to other introduction rules.

4.5.3 Theorem: Commutes

An expression only consisting of constants and additions is commutative, i.e. yields the same value when swapping all arguments:

```
fun swap where
  "swap (Const n) = Const n"
| "swap (Add a b) = Add (swap b) (swap a)"
| "swap (Sub a b) = Sub (swap b) (swap a)"

open_theorem commutes shows "eval e n ⟹ eval (swap e) n"

show_open commutes for const
proof–
```

```
  fix n
  have "swap (Const n) = Const n" by (rule swap.simps(1))
  with eval.const show "eval (swap (Const n)) n" by simp
qed

show_open commutes for add
proof–
  fix a ra b rb n
  assume a: "eval (swap a) ra"
    and b: "eval (swap b) rb"
    and ab: "ra + rb = n"
  have "swap (Add a b) = Add (swap b) (swap a)" by (rule swap.simps(2))
  with eval.add a b ab show "eval (swap (Add a b)) n" by simp
qed
```

Naturally, this cannot be shown for the subtraction introduction rule. For constants and additions the proofs are again very straightforward.

4.5.4 Theorem: No-Add

An example for a theorem where an inductive proof for one introduction rule needs access to other introduction rules is the following: Every expression that evaluates can be replaced by an expression containing no additions that evaluates to the same value.

First we implement a function to replace all additions by subtractions and constants as well as a function to verify that this actually eliminates all additions and show that it works as intended:

```
fun andf where
  "andf True True = True"
| "andf _ _ = False"

fun no_add where
  "no_add (Const _) = True"
| "no_add (Add _ _) = False"
| "no_add (Sub a b) = andf (no_add a) (no_add b)"

fun rem_add where
  "rem_add (Const n) = Const n"
| "rem_add (Add a b) = Sub (rem_add a) (Sub (Const 0) (rem_add b))"
| "rem_add (Sub a b) = Sub (rem_add a) (rem_add b)"

lemma rem_add_removes: "no_add (rem_add e)"
  by (induction e) auto
```

Then we show that after application of *rem_add* the resulting value stays the same:

open_theorem rem_add_correct **shows** "eval e n ⟹ eval (rem_add e) n"

show_open rem_add_correct **for** const
  using eval.const **by** simp

show_open rem_add_correct **for** add
  assumes sub const
  apply simp
  apply (erule eval.sub)
  apply (rule eval.sub)
  apply (rule eval.const)
  apply simp_all
  apply presburger
  done

show_open rem_add_correct **for** sub
  using eval.sub **by** simp

Here the proof for addition needs access to the introduction rules for constants and subtractions since they are used to show the correctness of the replacement expression.

4.5.5 Closure

Now that we have defined the predicate and used it to show the theorems we're interested in, we can close variants of the predicate for different sets of introduction rules. After specifying the intro rules, the package does the work of assembling the final proofs and installing the theorems for us:

close_inductive eval **assumes** const **and** add **for** eval
close_inductive eval **assumes** const add **and** sub **for** eval'

The resulting messages show what happens. The message for closing *eval* is:

```
Closing inductive predicate eval with const, add as eval.
  Candidates for closing: rem_add_correct, commutes, double, double_add
Cannot close open theorem "rem_add_correct", missing introduction rules: sub
Installing double_add_eval: eval ?e ?n ⟹ eval (Add ?e ?e) (2 * ?n)
Installing double_eval: eval ?e ?n ⟹ eval (cdouble ?e) (2 * ?n)
Installing commutes_eval: eval ?e ?n ⟹ eval (swap ?e) ?n
```

So all open theorems except for *rem_add_correct* could be closed. This is as expected, since the proof for addition for *rem_add_correct* also needs the subtraction introduction rule which is not included.

The message for closing *eval'* is:

```
Closing inductive predicate eval with const, add, sub as eval'.
  Candidates for closing: rem_add_correct, commutes, double, double_add
Cannot close open theorem "commutes", missing proofs for sub
```

```
Installing double_add_eval': eval' ?e ?n ⟹ eval' (Add ?e ?e) (2 * ?n)
Installing double_eval': eval' ?e ?n ⟹ eval' (cdouble ?e) (2 * ?n)
Installing rem_add_correct_eval': eval' ?e ?n ⟹ eval' (rem_add ?e) ?n
```

Here, *commutes* could not be closed because of the missing proof for sub. This is precisely what we want, since there is no proof for the commutativity of subtraction (which is why we could not give one).

5 Implementation

This chapter describes how the Open Inductive package is implemented in Isabelle. The implementation is done in Isabelle/ML, a dialect of ML that is developed alongside with Isabelle. The most significant difference from Standard ML is the presence of so-called Antiquotations that enable Isabelle syntax to create values within ML [29]. Otherwise, knowledge of Standard ML [21, 6] is sufficient to understand this implementation.

The first part of this chapter is concerned with the data that the package operates on. The hope is that this makes the actual operations easier to understand, or as Fred Brooks put it:

> Show me your tables, and I won't usually need your flowcharts; they'll be obvious.
>
> Fred Brooks in [12, p. 102]

Thus, the first section describes the data that is saved internally to model open inductive predicates. Section 5.2 explains what data is saved to model open theorems, that is their proposition and proofs. The next section gives an overview of the implementation strategy. It is not meant as a complete walk-through.

Section 5.4 describes the ML signature of the package to show how it can be used from other ML packages.

Finally, in Section 5.5 some peculiarities of the implementation are noted, that are not obvious from the overview.

5.1 Predicate Data

To store the data that represents an open inductive predicate, a record type named *open_predicate* is used. It's definition is:

```
type thm_handle = string
type intro = string * string

type open_predicate =
  {typ: string  option,
   thms: thm_handle list,
   intros:  intro  list}

val empty_open_pred =
  {typ = NONE,
```

```
  thms = [],
  intros  = []}
```

The *typ* field is used to save the optional type when declaring a predicate using *open_inductive*. If *open_inductive* is used without type annotation, the *empty_open_pred* record is used for initialization. The *thms* list holds a list of handles to theorems that use this predicate, it is updated when the predicate is found in a definition given by *open_theorem*. The *intros* list associates an introduction rule name to the actual rule in string form. This list is extended each time a new introduction rule is defined using *add_intro*.

5.2 Theorem Data

To store the data that represents an open theorem, a record type named *open_thm* is used. It's definition is:

```
type open_proof = string list * Proof.context * thm
type intro_proof = string * (string * open_proof) list

type open_thm =
  {prop: string,
   preds: string  list,
   proofs:  intro_proof list,
   direct_proof:  open_proof option}

val empty_thm =
  {prop = "",
   preds = [],
   proofs  = [],
   direct_proof = NONE}
```

The *prop* field stores the actual proposition in string form. The *preds* field stores handles to all open inductive predicates that occur in the proposition. These two fields are initialized when an open theorem is defined using *open_theorem*.

The other two fields are used to store proofs. For inductive proofs, *proofs* stores a two-level association list of predicates to introduction rule names to proofs. This is extended when giving per-introduction rule proofs using *show_open . . . for intro-name*.

The *direct_proof* field stores a direct proof for the whole proposition given using *show_open*. If a direct proof is present, the per-introduction-rule proofs can still be given but will be ignored by *close_inductive*.

5.3 Overview

The commands *open_inductive*, *add_intro* and *open_theorem* simply store their arguments in the records described above. Isabelle allows storing generic data of any type in

theory contexts. This package stores two symbol tables, an *open_predicate Symtab.table* of open predicates and an *open_thm Symtab.table* of open theorems, both indexed by their respective names.

The per-introduction-rule proofs have a theorem and an introduction rule argument. They present the user with a version of the theorem rewritten for the case of the selected introduction rule. This is achieved by generating a temporary predicate with just that one introduction rule using the Inductive package included in the Isabelle distribution. The induction theorem of this predicate is then applied to the theorem and the resulting proposition is given to the user to prove in a context where the temporary predicate definition does not exist. The context with the temporary definition is discarded.

For direct proofs, the predicate is simply fixed as a constant with the correct arity and type but no further information is exported in the proof context.

For closing an inductive predicate with a set of introduction rules using *close_inductive*, the predicate with these introduction rules is generated using the Inductive package. Then, all open theorems containing the predicate are looked up and considered for export.

If a direct proof is present, it can simply be exported, after the predicate constant is generalized and instantiated with the definition of the predicate that is now defined.

For inductive proofs, if proofs for all introduction rules are present, the induction theorem of the full predicate is applied to the proposition. This yields a number of cases, each of which can be closed using the previously stored general theorem for the respective introduction rule after instantiating it with the definition of the full predicate.

5.4 Isabelle/ML Interface

The ML interface of the Open Inductive package consists of two parts. The high-level interface just provides ML access to the functions implementing the Isar commands described in the previous chapter. It is not discussed here.

The low-level interface accepts symbol tables for storing predicates and theorems as arguments and returns updated versions as results.

open inductive. This function is used for registering new open predicates.

```
val open_inductive: Proof.context → string → string option →
  open_predicate Symtab.table → open_predicate Symtab.table
```

The string argument is the name of the predicate to be registered. The string option can be used to pass a type annotation. The Proof.context is used to parse the type annotation. The new *open_predicate* is then added to the symbol table and the updated symbol table is returned.

add intro. This function is used to add an introduction rule to a previously registered open predicate.

```
val add_intro: Proof.context → string → string → string →
  open_predicate Symtab.table → open_predicate Symtab.table
```

The three string arguments are, in this order, the name of the predicate where the rule is to be added, the name of the rule and the term defining the rule. The Proof.context is used to parse the term. The new introduction rule is added to the respective predicate in the symbol table and the updated table is returned as result.

open theorem. This function is used to register a new open theorem.

```
val open_theorem: Proof.context → string → string →
  open_predicate Symtab.table → open_thm Symtab.table →
  open_predicate Symtab.table * open_thm Symtab.table
```

The first string argument is the name for the theorem, the second its proposition. The Proof.context is used to parse the proposition, looking for open predicates used. The open theorem is registered in the *open_thm* symbol table and marked as used by the occurring predicates in the *open_predicate* symbol table. The pair of updated symbol tables is returned as result.

show intro. This function is used to prove an open theorem for a specific introduction rule.

```
val show_intro: Proof.context →
  (open_thm Symtab.table → Proof.context → Proof.context) →
  string → string → string list → open_predicate Symtab.table →
  open_thm Symtab.table → Proof.state
```

The string arguments are the name of the open theorem and the introduction rule. The string list may contain the name of additional introduction rules that shall be made available in the proof state as named assumptions. After the proof is finished in the returned proof state, the updated symbol table is passed as argument to the continuation "after_qed" function (the second argument).

show open. This function is used to prove an open theorem non-inductively.

```
val show_open: Proof.context →
  (open_thm Symtab.table → Proof.context → Proof.context) →
  string → string list → open_predicate Symtab.table →
  open_thm Symtab.table → Proof.state
```

The parameters are the same as in the previous function, except the name of the introduction rule which is not needed.

close inductive. This function is used to close an inductive predicate with a set of introduction rules and prove the final theorems.

val close_inductive: Proof.context → string → string list → string →
open_predicate Symtab.table → open_thm Symtab.table →
Proof.context ∗ (string ∗ thm) list

The first string argument is the name of the open predicate that is to be closed, the string list contains the names of the introduction rules that shall be used. The next string parameter is the name that shall be used for the definition of the final predicate, it can be the same as before. The function returns a Proof.context where the final predicate is defined, as well as an association list of theorem-name to theorem.

5.4.1 Helper functions

The types *open_predicate* and *open_thm* are opaque, since their values should only be modified using the public interface. When combining open predicates or theorems from different sources it is sometimes necessary to merge introduction rules or proofs. For this reason the interface exports two helper functions:

val pred_merge: open_predicate ∗ open_predicate → open_predicate
val thm_merge: open_thm ∗ open_thm → open_thm

These take a pair of predicate/theorem data as arguments and perform a merge. They can be used as arguments to *Symtab.join* to merge symbol tables returned by the other commands. The package uses them internally when merging data from different theory files.

5.5 Additional Notes

Following are some further notes regarding the implementation that do not fit in with the high-level overview in Section 5.3. They are meant to be helpful for somebody who attempts to read (and possibly improve or extend) the source code. There is some emphasis on what can be improved in the current implementation or what might be done differently in a rewrite.

The Good. Since the author has never written an ML package for Isabelle before, it probably does not contain much technical sophistication. Care was taken to stick to the general naming conventions, coding style and format of Isabelle source code [29].

Since the importance of this was mentioned on the Isabelle mailing list not to long ago [27], care was taken to have both an Isabelle/Isar as well as an ML interface to the package.

To avoid user confusion and namespace clutter, this package introduces no custom keywords at the Isar level.

The Bad. It is quite probable that the handling of contexts is rather unclean and sub-par in this package. This should be attributed to the author's rather shallow understanding of what they are and are not meant to be used for.

When assembling inductive proofs, the induction rule is just used by rule combination. Isabelle also contains a proper induction method [28], but the author could not make it work from ML.

The package contains some undocumented error messages of things that really should be impossible. In these cases it was unfortunately not possible to statically convince the compiler that this is the case.

The Ugly. The ML interface accepts strings in many places where at first glance terms would be more appropriate. The problem is that terms need to be parsed in a context where the right constants, e.g. for open predicates are present. Instead of forcing the user to set up this context himself, the functions accept a context and a string as parameters. The context is first adapted with the appropriate constant fixes and then used to parse the string to yield the right form of terms. This solution does seem ugly, but was deemed the most user-friendly option. It is also the way other packages, for example Inductive, handle this situation.

The *show_intro* function is very close to the Isar-level command and its implementing *show_intro_cmd* function. It does not accept a tactic to finish the proof but instead returns a *Proof.state* which must then be used to finish the proof. The main reason for having it this way is that it allowed more code reuse, probably at a convenience cost for the user of the ML interface.

All commands defined in Open Inductive print colorful output in Isabelle/Jedit (the "PIDE"). Since there is no general syntax coloring mechanism in the editor interface, existing syntax classes are reused to produce these colors. When a more general interface is implemented, this can be done in a cleaner way.

6 Evaluation

This chapter evaluates the usefulness of the Open Inductive package that was implemented for Isabelle. It discusses advantages and shortcomings of both the general concept as well as the current implementation.

In Section 6.4 some areas that allow for beneficial future work are highlighted.

Finally, Section 6.5 revisits the expression problem in the context of theorem proving. We discuss what role open inductive predicates play and how they change the situation.

6.1 Comparison to Direct Use of the Inductive Package

The overall goal of the Open Inductive package is to allow modular development of inductively defined predicates and theorems about them. When using the Inductive package in Isabelle, to expand an inductive predicate that is defined in an existing theory along with some theorems, the user has two options:

- Extend the predicate in that theory and adapt all proofs for the extended version. The obvious drawback is that all previous proofs are lost, and theorems that cannot be proved for the extended version no longer have a place.
- Copy-and-Paste the old definitions and theorems to a new theory, extend the predicate there. Then, adapt all proofs that can be adapted and discard the rest.

The key benefit of using Open Inductive is to avoid Copy-and-Paste without losing any of the old definitions or theorems.

For such a feature, it is of course hard to quantify the convenience gain for the user in the general case. Instead we discuss the savings in effort at our running example, the evaluation of arithmetic expressions.

The first version of the predicate, eval, consists of the two introduction rules for constants and addition. The theory also contains proofs for the two doubling theorems (double_add and double in Section 4.5.2) and the commutativity theorem (commutes in Section 4.5.3).

To extend this predicate with the introduction rule for subtraction and prove all theorems that are still valid for this version, the user needs to

- Copy-and-Paste the old definition of eval
- Add the introduction rule for subtraction
- Copy-and-Paste the theorems double_add and double and their proofs

- Extend the proof of double with the case for subtraction[1]

When using the Open Inductive package, the workflow is as follows:

- Add the introduction rule for subtraction to the open predicate eval
- Add the proof case for subtraction to the theorem double
- Issue one **close_inductive** command to produce eval'

Obviously this process is less tedious and error-prone than the manual one.

Maintenance benefits. The manual process has its weaknesses in maintenance as well. An error found in the definition of an introduction rule makes a fix necessary in two places – where it was found, and wherever the erroneous definition was copied.

Then all proofs that rely on this definition or one of its copies need to be fixed. In essence, the horror that is to be expected of Copy-and-Paste ensues: Maintenance of multiple nearly identical proof scripts. Identical or modified after copying – this is a troubling question when maintaing Copy-and-Pasted code. In this example, the proof for double_add is identical, the proof for double is not.

The benefits will be larger in a real-world setting, when more theorems exist that use the predicate and the definitions need to be maintained. When an error is found in a definition that has since been copied to several places, the hunt for all the places that need the fix begins. If Open Inductive is used, there only exists one erroneous definition so there is only one place to fix it. Likewise, for each theorem the case for this introduction rule is only proved once, so again only one fix is needed.

6.2 Benefits of Open Inductive

This section lists the benefits the user can expect from switching to Open Inductive for the definition of inductive predicates.

Flexible introduction rules. Unlike with all-in-one Inductive definitions, new introduction rules can be added in any place.

Flexible proofs. Like predicates, inductive proofs are not all-in-one but instead per introduction rule. Subproofs for new introduction rules can be added in any place.

Checked Closure. When closing a predicate, the package can detect for which open theorems all necessary subproofs exist and is able to warn about missing subproofs for specific introduction rules per theorem.

[1] The proof for double_add is not inductive and only relies on the addition introduction rule, so it works unchanged.

Proof assembly. In addition to detection of theorems that can be theoretically proved by assembling subproofs, the package will actually automatically assemble these and produce the finished theorems for the user.

Inter-dependent proofs. Using the definitions other introduction rules in the inductive proof for the case of a specific introduction rule is supported. The names of these additionally needed rules can be passed as parameters to *assumes*.

Inter-theorem merging. It is possible to define an open predicate and open theorem in one theory, then extend it with two different introduction rules and proofs for the respective rule in independent child theories. In a grandchild-theory that imports both of these children, the stored data is merged and both introduction rules and proofs are present.

6.3 Missing Features

This section lists some features that a user of the Inductive package might expect but that are not provided by Open Inductive at this point.

Inter-dependent theorems. It is not possible to use a previously proved open theorem in the proof of a later one. Like dependencies among introduction rules, this is mostly a question of handling assumptions correctly between differing proof contexts.

Automatic extension. When new open theorems and/or proofs are added after a call to **close_inductive**, the theorems are not closed for the existing predicate even if they could be.

This can be implemented as an automatic check after each call to **show_open** at the cost of a small amount of additional bookkeeping.

Arbitrary variables. The induction proof method in Isar accepts several arguments. An often indispensable one is *arbitrary*: The ability to declare which variables may vary in the inductive cases. Open Inductive as of yet does not support any arguments to induction, it simply applies the induction theorem as is.

To implement this, knowledge of proof methods and specifically the induction method is necessary.

Performance. At the moment, the storage of introduction rules in open predicates as well as the storage of sub-proofs for theorems is in association lists. For large inputs (e.g. machine-generated inductive predicates with thousands of introduction rules) this will be a performance bottleneck.

These performance issues are easily fixed by using more efficient data structures when the need arises.

6.4 Future Work

This section discusses some features that need more work, both conceptually and implementation-wise, before they can be added to Open Inductive.

Multiple Open Predicates. While the package allows the declaration and use of multiple open predicates, they can only be used independent of each other. Proofs that depend on more than one open predicate cannot be closed using the current *close_inductive* command.

More thought on the possible interactions between predicates and their introduction rules is necessary before such a feature can be attempted.

Mutually Recursive Predicates. The current implementation does not allow the definition of mutually recursive predicates, like even and odd defined in terms of each other.

Open Datatypes. A bigger project for the future are open datatypes: much like inductive predicates are defined by introduction rules, datatypes are defined by constructors. It is possible to imagine variants of the same datatype with different sets of constructors to be defined in a fashion similar to open predicates. This could work with commands like **open_datatype**, **add_constructor** and **close_datatype**. Clearly, open datatypes would synergize with open predicates extremely well, since they would remove the need for unused kinds of data in the smaller predicates.

To stick with our example: when closing the eval predicate for constants and addition there is really no reason to even have the "Sub" constructor be part of the expr type.

6.5 The Expression Problem Revisited

In the first chapter, the expression problem in the context of programming languages was discussed. A very similar expression problem exists for theorem proving: How to extend predicates and datatypes in with new constructors and introduction rules without breaking previous proofs?

The implementation of open inductive predicates solves one part of the question by allowing predicates to be extended with new introduction rules while preserving inductive proofs. It also makes reuse of existing proofs possible for the extended predicates.

To tackle the other half of the problem, an implementation of open datatypes as discussed in the previous section is necessary. This would then allow to add constructors in a modular way alongside with introduction rules. It is also necessary to implement modular proofs where the induction theorem of the datatype, not the predicate is necessary. An example is totality, e.g. for the running example:

$$\forall e.\ \exists n.\ \text{eval}\ e\ n$$

To prove this, all possible cases of e need to be considered, so it needs induction on the expression datatype. To show that the respective results exists, the introduction

rules of the eval predicate are needed, too, but the point is that the induction theorem of eval does not suffice here.

With having open datatypes a new way to define the helper functions that operate on these types, like swap and cdouble becomes necessary. This brings the expression problem back to the world of programming, which means solutions that have been found there can possibly be adapted for use in theorem proving as well.

Bibliography

[1] Sylvie Boldo et al. “A Formally-Verified C Compiler Supporting Floating-Point Arithmetic”. In: *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*. 2013, pp. 107–115. DOI: 10.1109/ARITH.2013.30. URL: http://doi.ieeecomputersociety.org/10.1109/ARITH.2013.30.

[2] Edwin Brady. “Idris, a general-purpose dependently typed programming language: Design and implementation”. In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X. URL: http://dx.doi.org/10.1017/S095679681300018X.

[3] Thierry Coquand and Gérard P. Huet. “The Calculus of Constructions”. In: *Inf. Comput.* 76.2/3 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3. URL: http://dx.doi.org/10.1016/0890-5401(88)90005-3.

[4] Sa Cui, Kevin Donnelly, and Hongwei Xi. “ATS: A Language That Combines Programming with Theorem Proving”. In: *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*. 2005, pp. 310–320. DOI: 10.1007/11559306_19. URL: http://dx.doi.org/10.1007/11559306_19.

[5] Benjamin Delaware, William R. Cook, and Don S. Batory. “Product lines of theorems”. In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 2011, pp. 595–608. DOI: 10.1145/2048066.2048113. URL: http://doi.acm.org/10.1145/2048066.2048113.

[6] Matthias Felleisen and Daniel P. Friedman. *The little MLer*. MIT Press, 1996. ISBN: 978-0-262-56114-3.

[7] Eduardo Giménez. “An Application of Co-inductive Types in Coq: Verification of the Alternating Bit Protocol”. In: *Types for Proofs and Programs, International Workshop TYPES’95, Torino, Italy, June 5-8, 1995, Selected Papers*. 1995, pp. 135–152. DOI: 10.1007/3-540-61780-9_67. URL: http://dx.doi.org/10.1007/3-540-61780-9_67.

[8] Eduardo Giménez. “Codifying Guarded Definitions with Recursive Schemes”. In: *Types for Proofs and Programs, International Workshop TYPES’94, Båstad, Sweden, June 6-10, 1994, Selected Papers*. 1994, pp. 39–59. DOI: 10.1007/3-540-60579-7_3. URL: http://dx.doi.org/10.1007/3-540-60579-7_3.

[9] Georges Gonthier. "The Four Colour Theorem: Engineering of a Formal Proof". In: *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*. 2007, p. 333. DOI: 10.1007/978-3-540-87827-8_28. URL: http://dx.doi.org/10.1007/978-3-540-87827-8_28.

[10] Michael J. C. Gordon. "Introduction to the HOL System". In: *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, August 1991, Davis, California, USA*. 1991, pp. 2–3.

[11] Mauro Jaskelioff, Neil Ghani, and Graham Hutton. "Modularity and Implementation of Mathematical Operational Semantics". In: *Electr. Notes Theor. Comput. Sci.* 229.5 (2011), pp. 75–95. DOI: 10.1016/j.entcs.2011.02.017. URL: http://dx.doi.org/10.1016/j.entcs.2011.02.017.

[12] Frederick P. Brooks Jr. *The mythical man-month - Essays on software engineering*. Addison-Wesley, 1975. ISBN: 978-0-201-00650-2.

[13] John Launchbury. "A Natural Semantics for Lazy Evaluation". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. 1993, pp. 144–154. DOI: 10.1145/158511.158618. URL: http://doi.acm.org/10.1145/158511.158618.

[14] Thomas F. Melham. "Automating Recursive Type Definitions in Higher Order Logic". In: *Current Trends in Hardware Verification and Automated Theorem Proving*. Ed. by G. Birtwistle and P. A. Subrahmanyam. Springer-Verlag, 1989, pp. 341–386. ISBN: 3-540-96988-8. URL: http://www.cs.ox.ac.uk/tom.melham/pub/Melham-1989-ART.pdf.

[15] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: http://dx.doi.org/10.1016/0022-0000(78)90014-4.

[16] Robin Milner. "LCF: A Way of Doing Proofs with a Machine". In: *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*. 1979, pp. 146–159. DOI: 10.1007/3-540-09526-8_11. URL: http://dx.doi.org/10.1007/3-540-09526-8_11.

[17] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9. URL: http://dx.doi.org/10.1007/3-540-45949-9.

[18] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[19] Lawrence C. Paulson. *A Fixedpoint Approach to Implementing (Co)Inductive Definitions*. Tech. rep. 320. Computer Laboratory, University of Cambridge, Dec. 1993.

[20] Lawrence C. Paulson. "Isabelle: The Next 700 Theorem Provers". In: *Logic and Computer Science*. Ed. by P. Odifreddi. Academic Press, 1990, pp. 361–386.

[21] Lawrence C. Paulson. *ML for the working programmer (2. ed.)* Cambridge University Press, 1996. ISBN: 978-0-521-57050-3.

[22] Wolfgang Rautenberg. *Einführung in die mathematische Logik - ein Lehrbuch mit Berücksichtigung der Logikprogrammierung*. Vieweg, 1996. ISBN: 978-3-528-06754-0.

[23] *The OCaml Programming Language*. URL: http://ocaml.org/.

[24] *The Rust Programming Language*. URL: http://www.rust-lang.org/.

[25] *The Scala Programming Language*. URL: http://www.scala-lang.org/.

[26] Philip Wadler. *The expression problem*. 1998. URL: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[27] Makarius Wenzel. *Accessing lift_definition from the ML-level*. 2014. URL: https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2014-October/msg00031.html.

[28] Makarius Wenzel. "Structured Induction Proofs in Isabelle/Isar". In: *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK, August 11-12, 2006, Proceedings*. 2006, pp. 17–30. DOI: 10.1007/11812289_3. URL: http://dx.doi.org/10.1007/11812289_3.

[29] Makarius Wenzel et al. *The Isabelle/Isar Implementation Manual*. 2014. URL: http://isabelle.in.tum.de/dist/Isabelle2014/doc/implementation.pdf.

[30] Markus Wenzel. "Isar - A Generic Interpretative Approach to Readable Formal Proof Documents". In: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*. 1999, pp. 167–184. DOI: 10.1007/3-540-48256-3_12. URL: http://dx.doi.org/10.1007/3-540-48256-3_12.