

# Chapter 1

## An Effective Proof Rule for General Type Classes

Ron van Kesteren<sup>1</sup>, Marko van Eekelen<sup>1</sup>, Maarten de Mol<sup>1</sup>

**Abstract:** Type classes are a widely adopted means of abstraction for overloading in functional programming languages. Although operating on different types, in general all instances of a class implement equivalent operations and hence have properties in common. Using formal reasoning, such a property can be proven by showing it holds for all instance definitions. This is not straightforward, however, because when instance definitions depend on each other, so will the proofs.

The proof assistant ISABELLE supports single parameter type classes and a proof rule for it based on structural induction on types. This method suffices for simple single parameter type classes, but does not lead to a user friendly tactic for more complex forms of overloading. In this paper, an effective proof rule is presented that works for all common extensions to type classes by using an induction scheme derived from the instance definitions. Moreover, it can be easily transformed into a user friendly tactic. This tactic will be implemented in the proof assistant SPARKLE.

### 1.1 INTRODUCTION

It is often stated that formulating properties about programs increases robustness and safety, especially when formal reasoning is used to prove these properties. Robustness and safety are becoming increasingly important considering the current dependence of society on technology. Research on formal reasoning has spawned many general purpose proof assistants, such as COQ [dt04], ISABELLE [NPW02], and PVS [OSRSC99]. Unfortunately, these general purpose tools are

---

<sup>1</sup>Nijmegen Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, Nijmegen, 6525 ED, The Netherlands; Phone: +031 (0)24-3653410; Email: rkestere@sci.ru.nl, M.vanEekelen@niii.ru.nl, M.deMol@niii.ru.nl

geared towards mathematicians and are hard to use when applied to more practical domains such as actual programming languages.

Because of this, proof assistants have been developed that are geared towards specific programming languages. This allows proofs to be conducted on the source program using specifically designed proof rules. Functional languages are especially suited for formal reasoning because they are referentially transparent. Examples of proof assistants for functional languages are EVT [NFD01] for ERLANG [AV91], SPARKLE [dMvEP01] for CLEAN [vEP01], and ERA [Win99] for HASKELL [Jon03].

### 1.1.1 Type classes

A feature that is commonly found in functional programming languages is overloading structured by *type classes* [WB89]. Type classes essentially are groups of types, the class *instances*, for which certain operations, the class members, are implemented. When such an operation is applied, an implementation is created from the available instance definitions based on the type at the application. Because these definitions may use class members as well, the created implementation may be different for each instance. Figure 1.1 shows an implementation of the equality operator using type classes and an overloaded function that uses it, which will be used as a running example throughout this paper.

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  x == y = predefinedeqint x y

instance (Eq a, Eq b) => Eq (a, b) where
  (x, y) == (u, v) = x == u && y == v

instance (Eq a) => Eq [a] where
  [] == [] = True
  (x:xs) == [] = False
  [] == (y:ys) = False
  (x:xs) == (y:ys) = x == y && xs == ys

isMember :: (Eq a) => a -> [a] -> Bool
isMember x [] = False
isMember x (y:ys) = x == y || isMember x ys
```

**FIGURE 1.1.** A type class for equality in HASKELL

In the most basic case, type classes have only one parameter that can be instantiated with a type as long as no two instances overlap. Furthermore, instances are

supposed to have a flat type, that is a type of the form  $X \alpha_1 \dots \alpha_n$  where  $X$  is a constructor and the  $\alpha$ 's are type variables. Several significant extensions have been proposed, most importantly allowing multiple parameters [JJM97], overlapping instances, instantiation with constructors [Jon93], and, more recently, functional dependencies [Jon00]. These extensions can be motivated by useful examples such as collections, coercion, isomorphisms and mapping. Figure 1.2 shows a multi parameter class for which transitivity holds. In this paper, the term *general type classes* is used for type classes with all mentioned extensions, except for functional dependencies, which are not relevant for this work.

```

class Inclusion a b where
  in :: a b -> Bool

instance Inclusion Int Int
  x in y = x == y

instance (Inclusion a b) => Inclusion a [b]
  x in []      = False
  x in (y:ys) = x in y || x in ys

instance (Inclusion a b) => Inclusion [a] b
  []      in y = True
  (x:xs) in y = x in y && xs in y

```

**FIGURE 1.2.** A multi parameter class in HASKELL

An important observation regarding type classes, is that in general the defined instances will be semantically related. For example, all instances of the equality operator usually implement an equivalence relation. It can be very useful to prove these properties for all available instance definitions, of which there are only a few. Unfortunately, this is not straightforward because the instance definitions may depend on each other and hence so will the proofs. For example, equality on lists is only an equivalence relation if equality on the list members is so as well.

### 1.1.2 Contributions

The only proof assistant with special support for overloading and type classes that we know of is ISABELLE [Nip93, Wen97]. Unfortunately, it only supports single parameter type classes, which is a severe limitation. Theoretically, structural induction on types, the approach taken by ISABELLE, can be used for all of these extensions. However, transforming this in a prover friendly tactic is not straightforward. We show that an induction scheme on types based on the instance definitions solves this problem and argue that it is more likely to be adaptable to (future) extensions of type classes. Using this induction scheme, a proof rule and tactic are defined that are both strong enough and easy to use.

As a proof of concept the tactic will be implemented in the proof assistant SPARKLE. The results however are generally applicable and can, for example, also be used for ISABELLE, if it would support the specification of general type classes, or HASKELL. In fact, the examples here use HASKELL syntax because most readers will be more familiar with HASKELL than with CLEAN. Furthermore, when the proof of concept implementation is finished, it can be used to prove properties about HASKELL programs by translating them to CLEAN using the HACLE translator [Nay04].

### 1.1.3 Outline

The rest of this paper is structured as follows. First, section 1.2 presents the proof assistant SPARKLE. Then, section 1.3 introduces basic definitions for types, expressions and type classes. Class constrained properties are defined in section 1.4. Then, in section 1.5, it is shown that although structural induction on types theoretically suffices, it is not immediately useful in practice. After that, section 1.6 presents the proof rule and that use induction based on the instance definitions which is extended in section 1.7 to multiple class constraints. Section 1.8 discusses the plans for implementation. Related work is discussed in section 1.9. In conclusion, section 1.10 presents a summary of the results.

## 1.2 SPARKLE

The need for this work arose when trying to improve the support for type classes in SPARKLE. SPARKLE is a proof assistant specifically geared towards CLEAN, which means it can reason about CLEAN concepts using rules based on CLEAN's semantics. Properties are specified in a first order predicate logic extended with equality on expressions, for example:

$$\forall n:\text{Int}[n \neq 1] \forall a \forall xs:[a] [\text{take } n \text{ } xs \text{ ++ drop } n \text{ } xs = xs]$$

These properties can be proven by using *tactics*, which are user friendly operations that transform a property into a number of logically stronger properties, the proof obligations or goals, that are easier to prove. A tactic is the implementation of (a combination of) theoretically sound *proof rules*. Whereas a proof rule is theoretically simple but not very prover friendly, a tactic is prover friendly but often theoretically more complex. The proof is complete when all remaining proof obligations are trivial. Some useful tactics are, for example, reduction of expressions, induction on type variables, and rewriting using hypotheses.

In SPARKLE, properties that contain member functions can only be proven for specific instances of that function. For example

$$\forall x:[\text{Int}] [x == x]$$

can be easily proven by induction on lists using reflexivity of equality on integers. Note that SPARKLE actually uses a translation of type classes, as will be explained

in section 1.3. Proving that something holds for *all* instances, however, is not possible in general. Consider for example reflexivity of equality:

$$\forall_a[\text{Eq} :: a \Rightarrow \forall_{x:a}[x == x]]$$

where  $\text{Eq} :: a$  denotes the constraint that equality must be defined for type  $a$ , that currently is not available in SPARKLE. This property can be split into a property for every instance definition, which gives among others the property for the instance for tuples:

$$\forall_{a,b}[\text{Eq} :: a \Rightarrow \text{Eq} :: b \Rightarrow \forall_{(x,y):(a,b)}[x == x \ \&\& \ y == y]]$$

It is clear that this property is true as long as it is true for instances  $a$  and  $b$ . Unfortunately, this hypothesis is not available. By using an approach based on induction, however, we may assume this hypothesis and are able to prove the property.

### 1.3 PRELIMINARIES

In this section, basic definitions for types, expressions and type classes are given. Instead of defining a proof rule that operates on the properties from section 1.2, it is assumed that the use of overloading is made explicit, as described elsewhere [WB89]. Here we will only briefly explain the result and present an example.

Key in the translation is the introduction of *evidence values* for predicates. The evidence value for predicate  $c :: a$  is the evidence that there is an instance of class  $c$  for type  $a$  (the implementation of the member). Hence, an evidence value exists if and only if the predicate is true. A program is translated by converting all instance definitions to functions (distinct names are created by suffixes). In expressions, the evidence value is substituted for member applications. When functions require certain classes to be defined, the evidence values for these constraints are passed as a parameter. Figure 1.3 shows an example of the result of the translation of the equality class from figure 1.1.

Because we intend to support constructor classes, types are formalized by a language of constructors [Jon93].

$$\tau ::= \alpha \mid \mathcal{X} \mid \tau \tau'$$

where  $\alpha$  and  $\mathcal{X}$  range over a given set of type variables and type constructors respectively.  $TV(\tau)$  denotes the set of type variables occurring in  $\tau$ . To ensure well-typedness, every type has a corresponding kind.

$$\kappa ::= * \mid \kappa \rightarrow \kappa'$$

The function *Kind* returns the kind of a type. The kinds of variables  $\alpha$  and constructors  $\mathcal{X}$  are fixed, for example `Int` and `Char` are of kind  $*$ , the function constructor `->` is of kind  $* \rightarrow * \rightarrow *$ , and the list constructor `[]` is of kind  $* \rightarrow *$ .

```

eqint :: Int -> Int -> Bool
eqint = predefinedeqint

eqtuple :: (a -> a -> Bool) -> (b -> b -> Bool)
         -> ((a, b) -> (a, b) -> Bool)
eqtuple ev1 ev2 (x, y) (u, v) = ev1 x u && ev2 y v

eqlist :: (a -> a -> Bool) -> ([a] -> [a] -> Bool)
eqlist ev [] [] = True
eqlist ev (x:xs) [] = False
eqlist ev [] (y:ys) = False
eqlist ev (x:xs) (y:ys) = ev x y && eqlist ev xs ys

isMember :: (a -> a -> Bool) -> a -> [a]
isMember ev x [] = False
isMember ev x (y:ys) = ev x y || isMember ev x ys

```

**FIGURE 1.3. Translation of figure 1.1**

The kind of application types is defined by the rule:

$$\frac{Kind(\tau) = \kappa \rightarrow \kappa' \quad Kind(\tau') = \kappa}{Kind(\tau \tau') = \kappa'}$$

For example,  $\text{Int} \rightarrow \text{Char}$  and  $[\text{Int}]$  are of kind  $*$ . Some might call only terms of kind  $*$  types and terms of kind  $\kappa \rightarrow \kappa'$  constructors. Here we will not make this distinction and call every term  $\tau$  a type.

Predicates are used to indicate that an instance of a certain class exists. An instance can be identified by an instantiation of the class parameters. The predicate  $c :: \bar{\tau}$  denotes that there is an instance of the class  $c$  for instantiation  $\bar{\tau}$  of the class parameters.

$$\pi ::= c :: \bar{\tau}$$

where  $c$  ranges over a given set of class names. For example  $\text{Eq} :: [\text{Int}]$  and  $\text{Eq} :: (\text{Int}, \text{Int})$  denote that there is an instance of the  $\text{Eq}$  class for types  $[\text{Int}]$  and  $(\text{Int}, \text{Int})$  respectively. Because these predicates will be used to constrain types to a certain class, they will be called *class constraints*. Class constraints in which only type variables occur in the type, for example  $\text{Eq} :: a$ , are called *simple*. Adding class constraints to types yields *class constrained types*, which are a special case of qualified types [Jon92] and are used to type class members and instances.

$$\sigma ::= \bar{\pi} \Rightarrow \tau$$

For example, the instance of  $\text{Eq}$  for tuples has type  $(\text{Eq} :: a, \text{Eq} :: b) \Rightarrow (a, b)$ .  $TV(\pi)$  and  $TV(\sigma)$  are straightforwardly defined.

Without loss of generality, for reasons of readability we restrict ourselves to type classes that have only one member and no subclasses throughout this paper. Member and subclasses can be supported by using dictionaries as evidence values.

An instance definition

$$\text{inst } \bar{\pi} \Rightarrow c :: \bar{\tau} = e$$

where  $e$  is a translated expressions, defines an instance  $\bar{\pi} \Rightarrow \bar{\tau}$  of class  $c$  that provides an instance of member  $c$ . The functions  $Type(\text{inst } c :: \bar{\pi} \Rightarrow \tau = e) = \tau$  and  $Context(\text{inst } c :: \bar{\pi} \Rightarrow \tau = e) = \bar{\pi}$  will be used to retrieve the type and context from an instance respectively.

The program context  $\psi$  is a set of function, class and instance definitions. The function  $Idefs_{\psi}(c)$  returns the set of instance definitions of class  $c$  defined in program  $\psi$ .

#### 1.4 CLASS CONSTRAINED PROPERTIES

Properties are formalized by a first order predicate logic extended with equality on expressions. This is a limited version of properties allowed in SPARKLE extended with class constraints.

$$\begin{aligned} p_c &::= \text{true} \mid \text{false} \\ p_u &::= \neg \\ p_b &::= \wedge \mid \vee \mid \rightarrow \mid \leftrightarrow \\ p_q &::= \forall \alpha \mid \forall x:\tau \\ p &= p_c \mid p_u p \mid p p_b p' \mid p_q p \mid e = e' \mid \pi \Rightarrow p \end{aligned}$$

These properties will be referred to as *class constrained properties*. Figure 1.4 shows some examples.

$$\begin{aligned} &\forall a[\mathbb{E}q :: a \Rightarrow \forall x:a[\text{ev}_{\mathbb{E}q::a} \ x \ x]] \\ &\forall a[\mathbb{E}q :: a \Rightarrow \forall x,y:a[\text{ev}_{\mathbb{E}q::a} \ x \ y \rightarrow \text{ev}_{\mathbb{E}q::a} \ y \ x]] \\ &\forall a[\mathbb{E}q :: a \Rightarrow \forall x,y,z:a[\text{ev}_{\mathbb{E}q::a} \ x \ y \rightarrow \text{ev}_{\mathbb{E}q::a} \ y \ z \rightarrow \text{ev}_{\mathbb{E}q::a} \ x \ z]] \end{aligned}$$

**FIGURE 1.4. Properties of the  $\mathbb{E}q$  class**

The property  $c :: \bar{\tau} \Rightarrow p$  means that in property  $p$  it is assumed that  $\bar{\tau}$  is an instance of class  $c$ . The evidence value for this class constraint is assigned to  $ev_{c::\bar{\tau}}$ . Thus, the semantics of the property  $\pi \Rightarrow p$  is defined as  $p_{[ev_{\pi} \mapsto Ev_{\psi}(\pi)]}$ , where  $Ev_{\psi}(\pi)$  is the evidence value for class constraint  $\pi$ . The rule for evidence creation is important for our purpose. Two definitions are required before it can be defined.

Firstly,  $Ai_{\psi}(\pi)$  determines the most specific instance definition applicable to constraint  $\pi$ . This is required because instances may overlap and hence more than one instance may be applicable.  $Ai_{\psi}$  is also defined for types that contain variables as long as it can be determined which instance definition should be applied.

Secondly, the *dependencies* of an instance are the instances it depends on:

$$Deps(c :: \bar{\tau}, i) = *_{Type(i) \rightarrow \bar{\tau}}(Context(i))$$

where  $*_{\bar{\tau} \rightarrow \bar{\tau}'}$  denotes the substitutor that maps type variables in  $\bar{\tau}$  so that  $*(\bar{\tau}) = \bar{\tau}'$ . When  $i$  is not provided,  $Ai_{\Psi}(c :: \bar{\tau})$  is assumed.

Evidence values are created by applying to expression of the most specific instance definition to the evidence values of its dependencies.

$$\frac{Deps(\pi) = \pi_1, \dots, \pi_n \quad Ai_{\Psi}(\pi) = \text{inst } c :: \bar{\pi}' \Rightarrow \bar{\tau}' = e}{Ev_{\Psi}(\pi) = e \quad Ev_{\Psi}(\pi_1) \quad \dots \quad Ev_{\Psi}(\pi_n)}$$

In the specification of proof rules *partial evidence creation* is used. Given an instance definition, an evidence value is created assuming the evidence values for the dependencies are already assigned to expression variables.

$$\frac{Deps(\pi, i) = \pi_1, \dots, \pi_n \quad i = \text{inst } c :: \bar{\pi}' \Rightarrow \bar{\tau}' = e}{Ev^p_{\Psi}(\pi, i) = e \quad x_{\pi_1} \quad \dots \quad x_{\pi_n}}$$

A specific instance definition  $i$  can be provided, because  $Ai_{\Psi}(\pi)$  might now be known in proofs. However, when  $i$  is not provided,  $Ai_{\Psi}(\pi)$  is assumed.

## 1.5 STRUCTURAL INDUCTION

The approach used for proving properties that contain overloaded identifiers taken in ISABELLE essentially is structural induction on types. In this section, proof rules and a tactic for structural induction are defined and it is argued that the proof rule for general type classes should be based on another induction scheme.

Reduction of expressions will often require the evidence values to be expanded. For example,  $ev_{Eq::[a]}$  can be expanded to  $eq\text{list } ev_{Eq::a}$  (see figure 1.3). Evidence expansion requires an instance definition to be selected, which requires information about the type at the application. Structural induction on types can provide this information and moreover allows the property to be assumed for structurally smaller types.

This leads to three proof rules. The first rule is the structural induction on types. The second rule is used to expand evidence values when it is clear which instance definition to use. A third rule is required to prove the cases where the

assumed type is no instance of the class.

$$\frac{\forall_{\mathcal{X} \bar{\alpha}'} \mid \text{Kind}(\mathcal{X} \bar{\alpha}') = \text{kind}(\alpha) [\forall_{\alpha'' \in \bar{\alpha}'} [p(\alpha'')] \rightarrow p(\mathcal{X} \bar{\alpha}')] }{\forall_{\alpha} [p(\alpha)]} \quad \text{(struct-ind)}$$

$$\frac{\forall_{TV(\pi)} [\pi \Rightarrow p(x_{\pi}, TV(\pi))]}{\forall_{TV(Deps(\pi))} [Deps(\pi) \Rightarrow p(Ev^p_{\Psi}(\pi), TV(\pi))]} \quad \text{(expand)}$$

$$\frac{\exists_{TV(\bar{\tau})} [ev = Ev_{\Psi}(c :: \bar{\tau})] \quad c :: \bar{\tau} \Rightarrow p}{\text{true}} \quad \text{(clean-up)}$$

When only single parameter, flat, and non-overlapping instances are supported, as in ISABELLE, these three rules are straightforwardly combined in a prover friendly tactic. The outermost constructor uniquely defines which instance to use, hence after applying **(struct-ind)** either **(expand)** or **(clean-up)** can be applied. Furthermore, the property should only be assumed for structurally smaller types that are in the context of the predicate. This combination can be defined as:

$$\frac{\forall_{i \in \text{Idefs}_{\Psi}(c) \forall_{TV(\text{Type}(i))}} [ \text{Deps}(c :: \text{Type}(i), i) \Rightarrow \forall_{c :: \tau \in \text{Deps}(c :: \text{Type}(i), i)} [p(d_{c :: \tau}, \tau)] \rightarrow p(Ev^p_{\Psi}(c :: \tau), i) ]}{\forall_{\alpha} [c :: \alpha \Rightarrow p(d, \alpha)]} \quad \text{(struct-tactic)}$$

assuming that  $\text{Type}(i)$  only introduces fresh variables. Using this tactic, the property

$$\forall_{\mathbf{a}} [\text{Eq} :: \mathbf{a} \Rightarrow \forall_{x:\mathbf{a}} [\text{evEq}::\mathbf{a} \ x \ x]]$$

can be proven by proving the following three properties (one for each instance definition):

$$\forall_{x:\text{Int}} [\text{eqInt} \ x \ x]$$

$$\forall_{\mathbf{a}, \mathbf{b}} [\text{Eq} :: \mathbf{a} \Rightarrow \text{Eq} :: \mathbf{b} \Rightarrow \forall_{x:\mathbf{a}} [\text{evEq}::\mathbf{a} \ x \ x] \rightarrow \forall_{x:\mathbf{b}} [\text{evEq}::\mathbf{b} \ x \ x] \rightarrow \forall_{(x,y):(a,b)} [\text{eqtuple} \ \text{evEq}::\mathbf{a} \ \text{evEq}::\mathbf{b} \ (x, y) \ (x, y)]]$$

$$\forall_{\mathbf{a}} [\text{Eq} :: \mathbf{a} \Rightarrow \forall_{x:\mathbf{a}} [\text{evEq}::\mathbf{a} \ x \ x] \rightarrow \forall_{x:[\mathbf{a}]} [\text{eqList} \ \text{evEq}::\mathbf{a} \ x \ x]]$$

which are easily proven using the available tactics.

For simple single parameter type classes, structural induction naturally leads to a user friendly tactic. Unfortunately, for general type classes the structural order does not really suffice. A user friendly tactic should generate a goal for every available instance definition and generate a hypothesis for all dependencies

on the same class. For general type classes, these dependencies are not always structurally smaller.

Consider for example the definitions in figure 1.5 where the member definitions have been left out. The instance of  $c$  for  $[a] \text{ (Tree } b)$  depends on  $\text{(Tree } b) \text{ (Tree } b)$  which is not structurally smaller. Hence, this hypothesis cannot be assumed when structural order is used, but when an order based on the instance definitions is used it can.

```

class c a b where c :: a b -> Bool
instance c Int Int where ...
instance (c b b) => [a] b where ...
instance (c a b) => (Tree a) (Tree b) where ...

```

FIGURE 1.5. Instances not necessarily depend on structural smaller types

## 1.6 INDUCTION ON INSTANCES

The induction scheme proposed in the previous section is induction on the set of defined instances of a class. Remember that the instances of a class are identified by sequences of closed types.  $\bar{\tau}$  is an instance of class  $c$  if an evidence value can be generated for the class constraint  $c :: \bar{\tau}$ .

$$Inst_{\Psi}(c) = \{\bar{\tau} \mid \forall c'::\bar{\tau} \in Deps(c::\bar{\tau}) [\bar{\tau}' \in Inst_{\Psi}(c')]\}$$

where it is assumed that no cyclic dependencies can occur.

An order on these sets is straightforwardly defined. Because the idea is to follow the instance definitions, an instance  $\tau'$  is one step smaller than  $\bar{\tau}$  if the evidence for  $\bar{\tau}$  depends on the evidence for  $\tau'$ , that is, if  $c :: \tau'$  is in the context of the most specific instance definition of  $c :: \bar{\tau}$ .

$$\bar{\tau} <_{(\Psi,c)}^1 \bar{\tau}' \Leftrightarrow c :: \bar{\tau}' \in Deps(c :: \bar{\tau})$$

Well-founded induction requires a well-founded partial order which is the reflexive transitive closure of  $<_{(\Psi,c)}^1$ . Using this order and the well-founded induction theorem the following proof rule can be straightforwardly derived.

$$\frac{\begin{array}{l} \forall i \in Idets_{\Psi}(c) \forall TV(c::Type(i)) \\ [ i = Ai_{\Psi}(c :: Type(i)) \\ \rightarrow \forall \bar{\tau} <_{(\Psi,c)}^1 Type(i) [p(Ev_{\Psi}(c :: \tau), \bar{\tau})] \\ \rightarrow p(x_{c::Type(i)}, Type(i)) \\ ] \end{array}}{\forall \bar{\alpha} \in Inst_{\Psi}(c) [c :: \bar{\alpha} \Rightarrow p(x_{c::\bar{\alpha}}, \bar{\alpha})]} \quad \text{(inst-ind)}$$

Especially important here is the assumption that  $i$  is the most specific instance definition. This allows this rule to be combined with **(expand)** and **(clean-up)** yielding the tactic:

$$\frac{\begin{array}{l} \forall_{i \in \text{Idefs}_\Psi(c)} \forall_{TV(c::\text{Type}(i))} \\ [ \text{Deps}(c :: \text{Type}(i), i) \\ \Rightarrow \forall_{c::\bar{c} \in \text{Deps}(c::\text{Type}(i), i)} [p(d_{c::\bar{c}}, \bar{c})] \\ \rightarrow p(\text{Ev}^p_\Psi(c :: \text{Type}(i), i), \bar{c}) \\ ] \end{array}}{\forall_{\bar{\alpha}} [c :: \bar{\alpha} \Rightarrow p(x_{c::\bar{\alpha}}, \bar{\alpha})]} \quad \text{(inst-tactic)}$$

which is only applicable when all variables in  $\bar{\alpha}$  are distinct.

This result is a generalization of **(struct-tactic)** to general type classes. For simple single parameter type classes it works the same as **(struct-tactic)**. Hence the proof of reflexivity using **(inst-tactic)** is equivalent to the proof in section 1.5. However, also hypotheses about instances that are possibly not structurally smaller may be assumed. Consider for example a property about the definitions in figure 1.5:

$$\forall_{a,b} [c :: a \ b \Rightarrow \forall_{x:a} \forall_{y:b} [\text{ev}_{c::a \ b \ x \ y} = \text{True}]]$$

This generates a a proof goal for every instance definition:

$$\begin{array}{l} \forall_{x:\text{Int}} \forall_{y:\text{Int}} [\text{ev}_{\text{cint}::a \ b \ x \ y} = \text{True}] \\ \forall_{a,b} [c :: b \ b \Rightarrow \forall_{x:b} \forall_{y:b} [\text{ev}_{c::b \ b \ x \ y} = \text{True}] \\ \rightarrow \forall_{x:[a]} \forall_{y:b} [\text{clist } \text{ev}_{c::b \ b \ x \ y} = \text{True}]] \\ \forall_{a,b} [c :: a \ b \Rightarrow \forall_{x:a} \forall_{y:b} [\text{ev}_{c::a \ b \ x \ y} = \text{True}] \\ \rightarrow \forall_{x:a} \forall_{y:\text{Tree } b} [\text{ctree } \text{ev}_{c::a \ b \ x \ y} = \text{True}]] \end{array}$$

where in the second proof obligation, the hypothesis could not have been assumed when using a structural order.

The tactic is easily derived from the well-founded induction theorem using the order based on the structure in instance definitions. Note that this is also the actual structure the prover uses, which makes this approach more likely to be adaptable to future extensions of type classes.

## 1.7 MULTIPLE CLASS CONSTRAINTS

The proof rule presented in the previous section works well when the property has only one class constraint. In the case of multiple class constraints, however, the rule might not be powerful enough. Consider the two class definitions in figure 1.6.

Given the property

$$\forall_a [f :: a \Rightarrow g :: a \Rightarrow [\text{ev}_{f::a \ x} = \text{ev}_{g::a \ x}]]$$

```

class f a where f :: a -> Bool

instance f [a] where
  f x = True

instance (f a, g a) => f (Tree a) where
  f Leaf = True
  f (Node x l r) = f x == g x

class g a where g :: a -> Bool

instance g (Tree a) where
  g x = True

instance (g a, f a) => g [a] where
  g [] = True
  g (x:xs) = g x == f x

```

FIGURE 1.6. Problematic class definitions

we can apply (**inst-tactic**), which yields among others the goal

$$\forall_a [g :: [a] \Rightarrow \forall_{x:[a]} [f \text{ list } x = \text{ev}_{g::a} x]]$$

This goal has a non-simple class constraint, which can only be removed by evidence expansion (**expand**), resulting in:

$$\forall_a [f :: a \Rightarrow g :: a \Rightarrow \forall_{x:[a]} [f \text{ list } x = g \text{ list } \text{ev}_{f::a} \text{ev}_{g::a} x]]$$

The expression `f list x` is of course `True`, but `(g list evf::a evg::a x)` is only `True` if `(evf::a x = evg::a x)` is. Unfortunately, the induction scheme did not allow us to assume this hypothesis. This problem is caused by the fact that type variables may occur in more than one class constraint. The solution is to take multiple class constraints into account in the induction scheme.

First, the set of type sequences that are instances of all classes that occur in a list of class constraints is defined. If  $\bar{\alpha}$  is a vector of all free variables in the simple class constraints  $\pi$ ,  $\bar{\tau}$  is in the set if all constraints are satisfied when all variables  $\bar{\alpha}$  are replaced by the corresponding type in  $\bar{\tau}$ .

$$\text{SetInst}_\Psi(\bar{\pi}, \bar{\alpha}) = \{\bar{\tau} \mid \forall_{x:\bar{\alpha}' \in \bar{\pi}} [*_{\bar{\alpha} \rightarrow \bar{\tau}}(\bar{\alpha}') \in \text{Inst}_\Psi(x)]\}$$

The order used here is a straightforward extension of the order on single class constraints to sets.

$$\bar{\tau} \leq_{(\Psi, \bar{\pi}, \bar{\alpha})} \bar{\tau}' \Leftrightarrow *_{\bar{\alpha} \rightarrow \bar{\tau}}(\bar{\pi}) \subseteq \bigcup_{\pi \in \bar{\pi}} [\text{Deps}(*_{\bar{\alpha} \rightarrow \bar{\tau}'}(\pi))]$$

The final tactic can be created using the same process as for single class constraints. Essentially, every possible combination of instance definitions is tried.

Every instance definition assumes types for some type variables. A combination is only possible if the types assigned to the type variables are unifiable.

$$\begin{aligned} & \text{SetMgu}((c_1 :: \bar{\alpha}_1, \dots, c_n :: \bar{\alpha}_n), (\tau_1, \dots, \tau_n)) = * \\ & \Leftrightarrow \\ & \forall_{1 \leq i \leq n} [* (\bar{\alpha}_i) = \tau_i] \wedge \forall_{*'} \forall_{1 \leq i \leq n} [*' (\bar{\alpha}_i) = \tau_i] \Rightarrow \exists *'' [*' = *'' \circ *] \end{aligned}$$

By the well-founded induction theorem, the property may be assumed for all smaller type sequences. This corresponds to all subsets of dependencies that are not less restrictive than the set of class constraints.

$$\frac{\begin{array}{l} \forall_{\bar{i} \in \text{Idefs}_\psi(\bar{\pi})} \\ [ * = \text{SetMgu}(\bar{\pi}, \text{Type}(\bar{i})) \\ \rightarrow \forall_{TV(*(\bar{\pi}))} \\ [ \text{Deps}(\bar{\pi}, \bar{i}) \\ \Rightarrow \forall_{\bar{\pi}' \subseteq \text{Deps}(\bar{\pi}, \bar{i})} [\exists *' [*'(\bar{\pi}) = \bar{\pi}' \Rightarrow p(x_{\bar{\pi}}, *'(TV(\bar{\pi})))] \\ \rightarrow p(E\nu^\psi(\bar{\pi}, \bar{i})) \\ ] \\ ] \end{array}}{\forall_{TV(\bar{\pi})} [\bar{\pi} \Rightarrow p(\bar{\pi}, TV(\bar{\pi}))]} \quad \text{(multi-tactic)}$$

where straightforward extensions of definitions to vectors have been used.

Note that applying this rule may result in non-simple class constraints when non-flat instance types are used. For non-simple class constraints, the induction tactics can not be applied, but the **(expand)** and **(clean-up)** rules might be used. However, in practice most instance definitions will have flat types.

Applying this tactic to the previously problematic property

$$\forall_a [f :: a \Rightarrow g :: a \Rightarrow \forall_{x:a} [e\nu_{f::a} x = e\nu_{g::a} x]]$$

results in two proof obligations:

$$\begin{aligned} & \forall_a [f :: a \Rightarrow g :: a \Rightarrow \forall_{x:a} [e\nu_{f::a} x = e\nu_{g::a} x]] \\ & \rightarrow \forall_{x:[a]} [f\text{list } x = g\text{list } e\nu_{f::a} e\nu_{g::a} x] \end{aligned}$$

$$\begin{aligned} & \forall_a [f :: a \Rightarrow g :: a \Rightarrow \forall_{x:a} [e\nu_{f::a} x = e\nu_{g::a} x]] \\ & \rightarrow \forall_{x:\text{Tree } a} [f\text{tree } e\nu_{f::a} e\nu_{g::a} x = g\text{tree } x] \end{aligned}$$

which can be now proven because of the available hypotheses.

The solution for multiple class constraints has some parallels to the constraint set satisfiability problem (CS-SAT), the problem of determining if there are types that satisfy a set of class constraints. The general CS-SAT problem is undecidable. However, recently, an algorithm was proposed that essentially tries to create a type that satisfies all constraints by trying all combinations of instance definitions, as we have been doing in our proof rule [CFV04].

## 1.8 IMPLEMENTATION

As a proof of concept, the **(multi-tactic)** tactic extended for multiple members and subclasses will be implemented in SPARKLE. Because SPARKLE's philosophy is to stay close to CLEAN concepts and syntax, the implementation will use translations to hide the explicit evidence values from the user. Furthermore, only the final proof rule will be implemented with the default behavior of using every class constraint. The implementation will be finished in december 2004.

## 1.9 RELATED WORK

As mentioned in section 1.1, the general proof assistant ISABELLE supports overloading and single parameter type classes. ISABELLE's notion of type classes is different from HASKELL's in that it represents types that satisfy certain properties instead of types for which certain values are defined. Nevertheless, the problems to be solved are equivalent. The rule used in ISABELLE is theoretically powerful enough for all significant extensions of type classes, hence if ISABELLE would support these extensions, most importantly multi parameter classes, our tactic could be defined in ISABELLE.

Essentially, the implementation of the tactic we proposed extends the induction techniques available in SPARKLE. Leonard Lensink proposed and implemented extensions of SPARKLE for induction and co-induction for mutually recursive functions and datatypes [Len04]. The main goal was to make the induction scheme match the structure of the program to ease the proofs. Together with this work this significantly increases the applicability of SPARKLE.

## 1.10 CONCLUSION

In this paper, we have presented an effective proof rule for class constrained properties that was straightforwardly transformed into a user friendly tactic. Structural induction on types is theoretically powerful enough but we showed that a user friendly tactic is more easily arrived at when an induction scheme is used that is based on the instance definitions. This rule was generalized to properties with multiple class constraints. The resulting rule is directly useable as a tactic and more user friendly than a tactic that could have been derived using the structural induction scheme. This shows that this approach is more flexible and adaptable to possible extensions of type classes. As a proof of concept, the resulting rule will be implemented in SPARKLE. Furthermore, the tactic can also be used for proving properties about HASKELL programs. If ISABELLE would support extensions for type classes, the tactic could be implemented in ISABELLE as well.

## ACKNOWLEDGEMENTS

We would like to thank Sjaak Smetsers for his suggestions and advice concerning this work, especially on the semantics of type classes in CLEAN.

## REFERENCES

- [AV91] J. L. Armstrong and R. Viriding. Erlang – An Experimental Telephony Switching Language. In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.
- [CFV04] C. Camarao, L. Figueiredo, and C. Vasconcellos. Constraint-set satisfiability for overloading. In *International Conference on Principles and Practice of Declarative Programming*, Verona, Italy, August 2004.
- [dMvEP01] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - SPARKLE: A functional theorem prover. In Th. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, LNCS 2312, pages 55–71, Älvsjö, Sweden, September 2001.
- [dt04] The Coq development team. *The Coq proof assistant reference manual (version 8.0)*. LogiCal Project, 2004.
- [JJM97] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.
- [Jon92] M. P. Jones. A theory of qualified types. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., 1992.
- [Jon93] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., 1993. ACM Press.
- [Jon00] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, LNCS 1782, pages 230–244. Springer-Verlag, 2000.
- [Jon03] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
- [Len04] L. Lensink. Induction and co-induction in Sparkle. In *Workshop Proceedings of the Fifth Symposium on Trends in Functional Programming (TFP 2004)*, 2004.
- [Nay04] M. Naylor. Haskell to Clean translation. University of York, 2004.
- [NFD01] T. Noll, L. Fredlund, and D. Gurov. The EVT Erlang verification tool. In *proceedings of the 7th international Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pages 582–585, Stockholm, 2001. Springer.
- [Nip93] T. Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188, 1993.
- [NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *A Proof Assistant for Higher-Order Logic*. Number XIII in LNCS 2283. Springer-Verlag, 2002.

- [OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
- [vEP01] M. van Eekelen and R. Plasmeijer. *Concurrent Clean Language Report (version 2.0)*. University of Nijmegen, December 2001.
- [WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.
- [Wen97] M. Wenzel. Type classes and overloading in higher-order logic. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 307–322, Murray Hill, New Jersey, 1997.
- [Win99] N. Winstanley. *Era User Manual (version 2.0)*. University of Glasgow, 1999.