Chapter 1

Proof Support for General Type Classes

Ron van Kesteren[1], Marko van Eekelen[1], Maarten de Mol[1]

Abstract: We present a proof rule and an effective tactic for proving properties about HASKELL type classes by proving them for the available instance definitions. This is not straightforward, because instance definitions may depend on each other. The proof assistant ISABELLE handles this problem for single parameter type classes by structural induction on types. However, this does not suffice for an effective tactic for more complex forms of overloading. We solve this using an induction scheme derived from the instance definitions. The tactic based on this rule is 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 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 transpar-

[1] 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

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

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

instance Eq Char where
  x == y = predefinedeqchar x y

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

FIGURE 1.1. A type class for equality in Haskell

ent. 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. These implementations are created from the available instance definitions and may be different for each instance. The type of an instance definition is called the *instance head*. The equality operator will be used as a running example throughout this paper (figure 1.1).

In the most basic case, type classes have only one parameter and instance heads are flat, that is, a single constructor applied to a list of type variables. Furthermore, no two instance definitions may overlap.

Several significant extensions have been proposed, such as multiple parameters [JJM97], overlapping instances, and instantiation with constructors [Jon93], that have useful applications such as collections, coercion, isomorphisms and mapping. In this paper, the term *general type classes* is used for systems of type classes that support these extensions and non-flat instance heads. Figure 1.2 shows a multi parameter class for the symmetric operation `eq2`.

An important observation regarding type classes is that, in general, the defined instances should be semantically related. For example, all instances of the equality operator usually implement an equivalence relation. These properties can be proven for all instances at once by proving them for the available instance definitions. Unfortunately, this is not straightforward because the instance definitions may depend on each other and hence so will the proofs. For example, equality on

```
class Eq2 a b where
  eq2 ::  a -> b -> Bool where

instance Eq2 Int Int where
  eq2 x y = x == y

instance Eq2 Char Char where
  eq2 x y = x == y

instance (Eq2 a b, Eq2 b c) => Eq2 (a, b) [c] where
  eq2 (x, y) [u, v] = eq2 x u && eq2 y v
  eq2 x y           = False

instance (Eq2 a b, Eq2 b c) => Eq2 [c] (a, b) where
  eq2 x y = eq2 y x
```

FIGURE 1.2. A multi parameter class in HASKELL

lists is only symmetric if equality on the list members is so as well.

1.1.2 Contributions

The only proof assistant with special support for overloading that we know of is ISABELLE [Nip93, Wen97], which essentially supports single parameter type classes and a proof rule for it based on structural induction on types. However, we show that for general type classes, an effective tactic is not easily derived when structural induction is used. We use an induction scheme on types based on the instance definitions to solve this problem. Using this induction scheme, a proof rule and tactic are defined that are both strong enough and effective.

As a proof of concept, we have implemented the tactic in the proof assistant SPARKLE for the programming language CLEAN. The results, however, are generally applicable and can, for example, also be used for HASKELL and ISABELLE, if ISABELLE would support the specification of general type classes. In fact, the examples here are presented using HASKELL syntax. SPARKLE is dedicated to CLEAN, but can also 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, the proof assistant SPARKLE is presented (section 1.2). Then, basic definitions for instance definitions, evidence values, and class constrained properties are introduced (section 1.3). After showing why structural induction does not suffice (section 1.4), the proof rule and tactic based on the instance definitions are defined (section 1.5) and extended to multiple class constraints (section 1.6). We end with a discussion of the implementation (section 1.7), related and future work (section 1.8), and a summary of the results (section 1.9).

1.2 SPARKLE

The need for this work arose whilst improving the proof support for type classes in SPARKLE. SPARKLE is a proof assistant specifically geared towards CLEAN, which means that 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. An example of this, using a slightly simplified syntax, is:

example: $\forall_{\mathtt{n:Int}|\mathtt{n}\neq\bot}\forall_{\mathsf{a}}\forall_{\mathtt{xs}:[\mathsf{a}]}[\mathtt{take\ n\ xs\ {+}{+}\ drop\ n\ xs\ =\ xs}]$

These properties can be proven 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 in general 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 expression variables, and rewriting using hypotheses.

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

sym$_{[\mathtt{Int}]}$: $\forall_{\mathtt{x}:[\mathtt{Int}]}\forall_{\mathtt{y}:[\mathtt{Int}]}[\mathtt{x\ ==\ y} \rightarrow \mathtt{y\ ==\ x}]$

can be easily proven by induction on lists using symmetry of equality on integers. Proving that something holds for *all* instances, however, is not possible in general. Consider for example symmetry of equality:

sym: $\forall_{\mathsf{a}}[\mathsf{Eq} :: \mathsf{a} \Rightarrow \forall_{\mathtt{x:a}}\forall_{\mathtt{y:x}}[\mathtt{x\ ==\ y} \rightarrow \mathtt{y\ ==\ x}]]$

where $\mathsf{Eq} :: \mathsf{a}$ denotes the, previously not available, constraint that equality must be defined for type a. This property can be split into a property for every instance definition, which gives among others the property for the instance for lists:

sym$_{[\mathsf{a}]}$: $\forall_{\mathsf{a}}[\mathsf{Eq} :: \mathsf{a} \Rightarrow \forall_{\mathtt{x}:[\mathsf{a}]}\forall_{\mathtt{y}:[\mathsf{a}]}[\mathtt{x\ ==\ y} \rightarrow \mathtt{y\ ==\ x}]]$

It is clear that this property is true as long as it is true for instance a. Unfortunately, this hypothesis is not available. Using an approach based on induction, however, we may be able to assume the hypotheses for all instances the instance definition depends on, and hence will be able to prove the property.

Internally, SPARKLE translates type classes to *evidence values* or *dictionaries* [WB89], that make the use of overloading explicit. The evidence value for a class

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

eqchar ::  Char -> Char -> Bool
eqchar = predefinedeqchar

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
```

FIGURE 1.3. Translation of figure 1.1

constraint `c` :: `a` is the evidence that there is an (implementation of the) instance of class `c` for type `a`. Hence, an evidence value exists if and only if the class constraint is satisfied. As usual, we will use the implementation itself as the evidence value. 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.

1.3 PRELIMINARIES

Instead of defining a proof rule that operates on the example properties from section 1.2, we define both instances and properties at the level that explicitly uses evidence values. In this section, basic definitions for instance definitions, evidence values, and class constrained properties are given.

1.3.1 Instance definitions

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 α and $\mathcal{X}$ range over a given set of type variables and type constructors respectively. For example, τ can be `Int`, `[Int]`, and `Tree Char`, but also the `[]`, `Tree`, and `->` constructors that take types as an argument and yield a list, tree, or function type respectively. $TV(\tau)$ denotes the set of type variables occurring in τ. The set of closed types $\mathcal{T}^c$ is the set of types for which $TV(\tau)$ is empty.

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. For example, `Eq` :: `[Int]` and `Eq` :: `(Int, Int)` denote that there is an instance of the `Eq` class for types `[Int]` and `(Int, Int)` respectively:

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

Because these predicates are used to constrain types to a certain class, they are called *class constraints*. Class constraints in which only type variables occur in the type, for example `Eq` :: `a`, are called *simple*. For reasons of simplicity, it is assumed that all type variables that occur in a class constraint are distinct.

Without loss of generality, throughout this paper we restrict ourselves to type classes that have only one member and no subclasses. Multiple members and subclasses can be supported using records of expressions for the evidence values. An instance definition:

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

defines an instance $\bar{\tau}$ of class c for types that satisfy class constraints $\bar{\pi}$. The instance definition provides the translated expression e for the class member c. The functions $Head(\mathsf{inst}\ c :: \bar{\pi} \Rightarrow \tau = e) = \tau$ and $Context(\mathsf{inst}\ c :: \bar{\pi} \Rightarrow \tau = e) = \bar{\pi}$ will be used to retrieve the instance head and context respectively.

The program context ψ, that contains the function and class definitions, also includes the available instance definitions. The function $Idefs_{\psi}(c)$ returns the set of instance definitions of class c defined in program ψ.

1.3.2 Evidence values

From the translation from type classes to evidence values, as briefly summarized in section 1.2, the rule for evidence creation is important for our purpose. Two definitions are required before it can be defined.

Firstly, because instance definitions are allowed to overlap, a mechanism is needed that chooses between them. Since the exact definition is not important for our purpose, we assume that the function $Ai_{\psi}(c :: \bar{\tau})$ determines the most specific instance definition applicable for instance $\bar{\tau}$ of class c. Ai_{ψ} 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:

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

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

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

$$\frac{\begin{array}{c} Deps(\pi) = \langle \pi_1, \ldots, \pi_n \rangle \\ Ai_{\psi}(\pi) = \mathsf{inst}\ c :: \bar{\pi}' \Rightarrow \bar{\tau}' = e \end{array}}{Ev_{\psi}(\pi) = e\ Ev_{\psi}(\pi_1)\ \ldots\ Ev_{\psi}(\pi_n)}$$

In proofs, evidence values will be created assuming the evidence values for the dependencies are already assigned to expression variables:

$$\frac{Deps(\pi, i) = \langle \pi_1, \ldots, \pi_n \rangle \qquad i = \mathsf{inst}\ c :: \bar{\pi}' \Rightarrow \bar{\tau}' = e}{Ev^p{}_{\psi}(\pi, i) = e\ \texttt{ev}_{\pi_1}\ \ldots\ \texttt{ev}_{\pi_n}}$$

assuming that the evidence for π is assigned to the variable $\texttt{ev}_{\pi}$. A specific instance definition i can be provided, because $Ai_{\psi}(\pi)$ might not be known in proofs.

1.3.3 Class constrained properties

In SPARKLE, properties are formalized by a first order predicate logic extended with equality on expressions. The equality on expressions is designed to handle infinite and undefined expressions well.

We extend these properties with class constraints, that can be used to constrain types to a certain class. These properties will be referred to as *class constrained properties*. For example, consider symmetry and transitivity of equality:

sym: $\forall_{\texttt{a}}[\texttt{Eq} :: \texttt{a} \Rightarrow \forall_{\texttt{x,y:a}}[\texttt{ev}_{\textbf{Eq::a}}\ \texttt{x}\ \texttt{y} \rightarrow \texttt{ev}_{\textbf{Eq::a}}\ \texttt{y}\ \texttt{x}]]$

trans: $\forall_{\texttt{a}}[\texttt{Eq} :: \texttt{a} \Rightarrow \forall_{\texttt{x,y,z:a}}[\texttt{ev}_{\textbf{Eq::a}}\ \texttt{x}\ \texttt{y} \rightarrow \texttt{ev}_{\textbf{Eq::a}}\ \texttt{y}\ \texttt{z}$
$\rightarrow \texttt{ev}_{\textbf{Eq::a}}\ \texttt{x}\ \texttt{z}]]$

The property $c :: \bar{\tau} \Rightarrow p$ means that in property p it is assumed that $\bar{\tau}$ is an instance of class c and the evidence value for this class constraint is assigned to $\texttt{ev}_{c::\bar{\tau}}$. Thus, the semantics of the property $\pi \Rightarrow p$ is defined as $p_{[\texttt{ev}_{\pi} \mapsto Ev_{\psi}(\pi)]}$.

1.4 STRUCTURAL INDUCTION

The approach for proving properties that contain overloaded identifiers taken in ISABELLE essentially is structural induction on types. In this section it is argued that the proof rule for general type classes should use another induction scheme.

Structural induction on types seems an effective approach because it gives more information about the type of an evidence value. This information can be used to expand evidence values. For example, $\texttt{ev}_{\textbf{Eq::[a]}}$ can be expanded to `eqlist` $\texttt{ev}_{\textbf{Eq::a}}$ (see figure 1.3).

$$\frac{Ai_{\psi}(\pi) = i \qquad \forall_{TV(\pi)}[Deps(\pi) \Rightarrow p(Ev^p{}_{\psi}(\pi))]}{\forall_{TV(\pi)}[\pi \Rightarrow p(\texttt{ev}_{\pi})]} \quad \textbf{(expand)}$$

More importantly, structural induction allows the property to be assumed for structurally smaller types. Ideally the hypothesis should be assumed for all dependencies on the same class. Unfortunately, structural induction does not always allow this for multi parameter classes.

Consider for example the multi parameter class in figure 1.2. The instance of `Eq2` for `[Int] (Char, Char)` depends on the instance for `Char Int`, which is not structurally smaller because `Char` is not structurally smaller than `[Int]`, and `Int` is not structurally smaller than `(Char, Char)`. Hence, the hypothesis cannot be assumed for this dependency. This problem can be solved by basing the induction scheme on the instance definitions.

1.5 INDUCTION ON INSTANCES

The induction scheme proposed in the previous section can be used on the set of defined instances of a class. In this section, a proof rule and tactic that use this scheme are defined and applied to some examples.

1.5.1 Proof rule and tactic

We first define the set of instances of a class and an order based on the instance definitions on it. The well-founded induction theorem applied to the defined set and order yields the proof rule. Then, the tactic is presented that can be derived from this rule.

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 $c :: \bar{\tau}$. Hence, the set of instances of class c can be defined as:

$$\mathit{Inst}_{\psi}(c) = \{\bar{\tau} \mid \forall_{c' :: \bar{\tau}' \in \mathit{Deps}(c :: \bar{\tau})}[\bar{\tau}' \in \mathit{Inst}_{\psi}(c')]\}$$

For example, $\mathit{Inst}_{\psi}(\texttt{Eq}) = \{\texttt{Int}, \texttt{Char}, \texttt{[Int]}, \texttt{[Char]}, \texttt{[[Int]]}, \ldots\}$.

An order on this set is straightforwardly defined. Because the idea is to base the order on the instance definitions, an instance $\bar{\tau}'$ is considered one step smaller than $\bar{\tau}$ if the evidence for $\bar{\tau}$ depends on the evidence for $\bar{\tau}'$, that is, if $c :: \bar{\tau}'$ is a dependency of the most specific instance definition for $c :: \bar{\tau}$. For example, `Int` $<^1_{(\psi,\texttt{Eq})}$ `[Int]` and `[Char]` $<^1_{(\psi,\texttt{Eq})}$ `[[Char]]`.

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

Note that there is a specific set of instances for each class and therefore also a specific order for each class.

Well-founded induction requires a well-founded partial order, for which we use the reflexive transitive closure of $<^1_{(\psi,c)}$. It can be easily derived from the way evidence values are generated that this is indeed a well-founded partial order. Applying this order, $\leq_{(\psi,c)}$, to the well-founded induction theorem yields the following proof rule:

$$\frac{\forall_{\bar{\tau} \in \mathit{Inst}_{\psi}(c)}[\forall_{\bar{\tau}' \leq_{(\psi,\mathrm{c})} \bar{\tau}}[p(\bar{\tau}')] \rightarrow p(\bar{\tau})]}{\forall_{\bar{\alpha} \in \mathit{Inst}_{\psi}(c)}[p(\bar{\alpha})]} \quad \textbf{(inst-rule)}$$

Rewriting the proof rule using the definitions of $\mathit{Inst}_{\psi}(c)$, $\leq_{(\psi,c)}$, evidence creation, and class constrained properties results in a tactic that can be directly applied to class constrained properties. For all class constraints $c :: \bar{\alpha}$:

$$\frac{\begin{array}{l}\forall_{i \in \mathit{Idefs}_{\psi}(c)} \forall_{\mathit{Head}(i) \in \langle \mathcal{T}^c \rangle} \\ \quad [\ \mathit{Deps}(c :: \mathit{Head}(i), i) \\ \quad\quad \Rightarrow \forall_{c' :: \bar{\tau}' \in \mathit{Deps}(c :: \mathit{Head}(i), i)} [c = c' \Rightarrow p(\mathtt{ev}_{c :: \bar{\tau}'}, \bar{\tau}')] \\ \quad\quad \rightarrow p(\mathit{Ev}^p{}_{\psi}(c :: \mathit{Head}(i), i), \mathit{Head}(i)) \\ \quad] \end{array}}{\forall_{\bar{\alpha} \in \langle \mathcal{T}^c \rangle} [c :: \bar{\alpha} \Rightarrow p(\mathtt{ev}_{c :: \bar{\alpha}}, \bar{\alpha})]} \quad \textbf{(inst-tactic)}$$

where it is assumed that all variables in $\mathit{Head}(i)$ are fresh. When the tactic is applied to a class constrained property, it generates a proof obligation for every available instance definition with hypotheses for all dependencies on the same class.

1.5.2 Results

The result is both a proof rule and a user friendly tactic. The tactic is nicely illustrated by symmetry of equality (figure 1.1 and 1.3). When **(inst-tactic)** is applied to:

sym: $\forall_{\mathsf{a}} [\mathsf{Eq} :: \mathsf{a} \Rightarrow \forall_{\mathsf{x:a}} \forall_{\mathsf{y:a}} [\mathtt{ev}_{\mathsf{Eq::a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathsf{Eq::a}}\ \mathtt{y}\ \mathtt{x}]]$

it generates the following three proof obligations (one for each instance definition):

sym$_{\mathtt{Int}}$**:** $\forall_{\mathtt{x:Int}} \forall_{\mathtt{y:Int}} [\mathtt{eqint}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{eqint}\ \mathtt{y}\ \mathtt{x}]$

sym$_{\mathtt{Char}}$**:** $\forall_{\mathtt{x:Char}} \forall_{\mathtt{y:Char}} [\mathtt{eqchar}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{eqchar}\ \mathtt{y}\ \mathtt{x}]$

sym$_{[\mathsf{a}]}$**:**

$$\begin{array}{l} \forall_{\mathsf{a}}\ [\ \mathsf{Eq} :: \mathsf{a} \\ \quad \Rightarrow \forall_{\mathsf{x:a}} \forall_{\mathsf{y:a}} [\mathtt{ev}_{\mathsf{Eq::a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathsf{Eq::a}}\ \mathtt{y}\ \mathtt{x}] \\ \quad \rightarrow \forall_{\mathsf{x:[a]}} \forall_{\mathsf{y:[a]}} [\mathtt{eqlist}\ \mathtt{ev}_{\mathsf{Eq::a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{eqlist}\ \mathtt{ev}_{\mathsf{Eq::a}}\ \mathtt{y}\ \mathtt{x}] \\ \quad] \end{array}$$

which are easily proven using the already available tactics.

The previous step could also have been taken using a tactic based on structural induction on types. However, **(inst-tactic)** can also assume hypotheses for dependencies that are possibly not structurally smaller. Consider for example the symmetry of `eq2` in figure 1.2:

sym2:

$$\begin{array}{l} \forall_{\mathsf{a,b}}\ [\ \mathsf{Eq2} :: \mathsf{a}\ \mathsf{b} \Rightarrow \mathsf{Eq2} :: \mathsf{b}\ \mathsf{a} \\ \quad \Rightarrow \forall_{\mathsf{x:a}} \forall_{\mathsf{y:b}} [\mathtt{ev}_{\mathsf{Eq2::a}\ \mathtt{b}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathsf{Eq2::b}\ \mathtt{a}}\ \mathtt{y}\ \mathtt{x}] \\ \quad] \end{array}$$

Applying **(inst-tactic)** to this property generates a proof obligation for every instance definition, including one for the fourth instance of `Eq2` in figure 1.2, where `eq2list` is the translation of that instance definition:

$$
\begin{array}{ll}
\textbf{sym2}_{[\mathsf{a}]}\textbf{:} & \forall_{\mathsf{a,b,c}} \\
& \quad [\,\mathsf{Eq2} :: \mathsf{b}\ \mathsf{a} \Rightarrow \mathsf{Eq2} :: \mathsf{c}\ \mathsf{a} \\
& \qquad \Rightarrow [\mathsf{Eq2} :: \mathsf{a}\ \mathsf{b} \Rightarrow \forall_{\mathsf{x:b}}\forall_{\mathsf{y:a}}[\mathtt{ev}_{\mathsf{Eq2::b}\ \mathtt{a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathsf{Eq2::a}\ \mathtt{b}}\ \mathtt{y}\ \mathtt{x}]] \\
& \qquad \rightarrow [\mathsf{Eq2} :: \mathsf{a}\ \mathsf{c} \Rightarrow \forall_{\mathsf{x:c}}\forall_{\mathsf{y:a}}[\mathtt{ev}_{\mathsf{Eq2::c}\ \mathtt{a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathsf{Eq2::a}\ \mathtt{c}}\ \mathtt{y}\ \mathtt{x}]] \\
& \qquad \rightarrow \mathsf{Eq2} :: (\mathsf{b},\mathsf{c})\ [\mathsf{a}] \Rightarrow \forall_{\mathsf{x:[a]}}\forall_{\mathsf{y:(b,c)}}[\\
& \qquad\qquad\qquad\qquad \mathtt{eq2list}\ \mathtt{ev}_{\mathsf{Eq2::b}\ \mathtt{a}}\ \mathtt{ev}_{\mathsf{Eq2::c}\ \mathtt{a}}\ \mathtt{x}\ \mathtt{y} \\
& \qquad\qquad\qquad\qquad \rightarrow \mathtt{ev}_{\mathsf{Eq2::(b,c)}\ [\mathsf{a}]}\ \mathtt{y}\ \mathtt{x}] \\
& \quad]
\end{array}
$$

In this proof obligation, the hypotheses could not have been assumed when using structural induction on types (see section 1.4), hence our tactic is useful in more cases.

1.6 MULTIPLE CLASS CONSTRAINTS

The proof rule and tactic presented in the previous section work well when the property has only one class constraint. In case of multiple class constraints, however, the rules might not be powerful enough. In this section it is shown that this problem does indeed occur. Therefore, a more general proof rule and tactic are defined and applied to some examples.

The problem

Consider the two class definitions in figure 1.4. The translated instance definitions are respectively called `fint`, `flist`, `ftree`, `gint`, `gtree`, and `glist` at the level of dictionaries. Given the property:

$$\textbf{same:}\qquad \forall_{\mathsf{a}}[\mathtt{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow [\mathtt{ev}_{\mathsf{f::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]$$

Applying **(inst-tactic)** yields among others the goal:

$$\textbf{same}_{[\mathsf{a}]}\textbf{f:}\ \ \forall_{\mathsf{a}}[\mathsf{g} :: [\mathsf{a}] \Rightarrow \forall_{\mathsf{x:[a]}}[\mathtt{flist}\ \ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]$$

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

$$
\begin{array}{l}
\textbf{same}_{[\mathsf{a}]}\textbf{f':}\ \forall_{\mathsf{a}}[\mathtt{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow \forall_{\mathsf{x:[a]}}[\mathtt{flist}\ \ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad = \mathtt{glist}\ \ \mathtt{ev}_{\mathsf{f::a}}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]
\end{array}
$$

After some reduction steps, this can be transformed into:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)

class f a where f ::  a -> Bool

instance f Int where
  f x = x == x

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

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 Int where
  g x = x == x

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

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

FIGURE 1.4. Problematic class definitions

$$\mathbf{same}_{[a]}\mathbf{f''}\text{:}\, \forall_{a}[f :: a \Rightarrow g :: a \Rightarrow \forall_{x:[a]}[\mathtt{ev}_{g::a}\ \mathtt{x}\ \mathtt{==}\ \mathtt{ev}_{g::a}\ \mathtt{x} \\ = \mathtt{ev}_{f::a}\ \mathtt{x}\ \mathtt{==}\ \mathtt{ev}_{g::a}\ \mathtt{x}]]$$

This proof obligation is true when $\mathtt{ev}_{f::a}\ \mathtt{x} = \mathtt{ev}_{g::a}\ \mathtt{x}$. Unfortunately, the induction scheme did not allow us to assume this hypothesis. Since this problem is caused by the fact that the type variables occur in more than one class constraint, the natural solution is to take multiple class constraints into account in the induction scheme.

1.6.1 Proof rule and tactic

We take the same approach as in the previous section. We first define the set of instances, the order, the proof rule and the tactic. Then, in section 1.6.2, it is shown that the new tactic solves the problem.

First, the set of type sequences that are instances of all classes that occur in a list of class constraints is defined. $\bar{\tau}$ is a member of the set if all class constraints $\bar{\pi}$ are satisfied when all variables $TV(\bar{\pi})$ are replaced by the corresponding type from $\bar{\tau}$. We assume here that $TV(\bar{\pi})$ is a linearly ordered, for example lexicographically, sequence and that the elements of $\bar{\tau}$ are in the corresponding order. For example, $\mathit{SetInst}_{\psi}(\texttt{f} :: \texttt{a}, \texttt{g} :: \texttt{a}) = \{\texttt{Int}, \texttt{[Int]}, \texttt{Tree Int}, \texttt{[[Int]]}, \ldots\}$.

$$\mathit{SetInst}_{\psi}(\bar{\pi}) = \{\bar{\tau} \mid \forall_{c::\bar{\alpha'} \in \bar{\pi}}[*_{TV(\bar{\pi}) \to \bar{\tau}}(\bar{\alpha'}) \in \mathit{Inst}_{\psi}(c)]\}$$

The order on this set is an extension of the order for single class constraints to sets. A sequence of types τ is considered one step smaller than τ' if $*_{TV(\pi) \to \tau}(\pi)$ is a subset of the dependencies of $*_{TV(\pi) \to \tau}(\pi)$. For example, $\texttt{[Int]} <^{1}_{(\psi, \langle \texttt{f::a}, \texttt{g::a} \rangle)} \texttt{([[Int]])}$ because $\{\texttt{f} :: \texttt{[Int]}, \texttt{g} :: \texttt{[Int]}\}$ is a subset of $\mathit{Deps}(\texttt{g} :: \texttt{[[Int]]}) \cup \mathit{Deps}(\texttt{f} :: \texttt{[[Int]]})$. Here, sequences of class constraints are lifted to sets when required:

$$\bar{\tau} <^{1}_{\psi, \bar{\pi}} \tau' \Leftrightarrow *_{TV(\bar{\pi}) \to \bar{\tau}}(\bar{\pi}) \subseteq \bigcup_{\pi \in \bar{\pi}} [\mathit{Deps}(*_{TV(\bar{\pi}) \to \bar{\tau}'}(\pi))])$$

Again, it can be derived from the evidence creation that the reflexive transitive closure of this order, $\leq_{(\psi, \bar{\pi})}$, is a well-founded partial order.

Applying the well-founded induction theorem to this set and order yields the proof rule for multiple class constraints. For every sequence of simple class constraints $\bar{\pi}$:

$$\frac{\forall_{\bar{\tau} \in \mathit{SetInst}_{\psi}(\bar{\pi})}[\forall_{\bar{\tau}' \leq_{(\psi, \bar{\pi})} \bar{\tau}}[p(\bar{\tau}')] \to p(\bar{\tau})]}{\forall_{\bar{\tau} \in \mathit{SetInst}_{\psi}(\bar{\pi})}[p(\bar{\tau})]} \quad \textbf{(multi-rule)}$$

Because multiple class constraints are involved, defining the final tactic is a bit more complicated. Instead of all instance definitions, every combination of instance definitions, one for each class constraint, has to be tried. All of these instance definitions make assumptions about the types of the type variables, and these assumptions should be unifiable. Therefore, we define the most general unifier that takes the sharing of type variables across class constraints into account:

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

Furthermore, for readability of the final tactic, some straightforward extensions of existing definitions to vectors are used:

$$\begin{array}{lcl} \mathit{Idefs}_{\psi}(\langle \pi_1, \ldots, \pi_n \rangle) & = & \{i_1, \ldots, i_n \mid i_j \in \mathit{Idefs}_{\psi}(\pi_j)\} \\ \mathit{Head}(\langle i_1, \ldots, i_n \rangle) & = & \langle \mathit{Head}(i_1), \ldots, \mathit{Head}(i_n) \rangle \\ \mathit{Ev^p}_{\psi}(\langle \pi_1, \ldots, \pi_n \rangle, \langle i_1, \ldots, i_n \rangle) & = & \langle \mathit{Ev^p}_{\psi}(\pi_1, i_1), \ldots, \mathit{Ev^p}_{\psi}(\pi_n, i_n) \rangle \\ \texttt{ev}_{\langle \pi_1, \ldots, \pi_n \rangle} & = & \langle \texttt{ev}_{\pi_1}, \ldots, \texttt{ev}_{\pi_n} \rangle \\ \mathit{Deps}(\langle \pi_1, \ldots, \pi_n \rangle, \langle i_1, \ldots, i_n \rangle) & = & \langle \mathit{Deps}(\pi_1, i_1), \ldots, \mathit{Deps}(\pi_n, i_n) \rangle \end{array}$$

Finally, using the presented definitions, evidence creation, class constrained properties, and the proof rule, the tactic can be defined. For every sequence of simple class constraints $\bar{\pi}$:

$$\frac{\begin{array}{l} \forall_{\bar{\imath} \in \mathit{Idefs}_{\psi}(\bar{\pi})} \exists_{* \mid * = \mathit{SetMgu}(\bar{\pi}, \mathit{Head}(\bar{\imath}))} \forall_{*(\mathit{Head}(\bar{\imath})) \in \langle \mathcal{T}^{c} \rangle} \\ \quad [\ \mathit{Deps}(*(\bar{\pi}), \bar{\imath}) \\ \qquad \Rightarrow \forall_{*' \mid *'(\bar{\pi}) \subseteq \mathit{Deps}(*(\bar{\pi}), \bar{\imath})} [p(\mathtt{ev}_{*'(\bar{\pi})}, *'(\mathit{TV}(\bar{\pi})))] \\ \qquad \rightarrow p(\mathit{Ev^{p}}_{\psi}(*(\bar{\pi}), \bar{\imath}), *(\mathit{Head}(\bar{\imath}))) \\ \quad] \end{array}}{\forall_{\mathit{TV}(\bar{\pi})} [\bar{\pi} \Rightarrow p(\mathtt{ev}_{\bar{\pi}}, \mathit{TV}(\bar{\pi}))]} \quad \textbf{(multi-tactic)}$$

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

This 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 [CFV04] 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 tactic.

1.6.2 Results

In this section, we have generalized the proof rule and tactic from section 1.5 to multiple class constraints. In case of a single class constraint, the new rules behave exactly the same as **(inst-rule)** and **(inst-tactic)**. However, now we can apply **(multi-tactic)** to multiple class constraints at once. Given the previously problematic property:

same: $\quad \forall_{\mathsf{a}}[\mathsf{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow [\mathtt{ev}_{\mathsf{f::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]$

this yields three proof obligations, one for every unifiable combination of instance definitions:

same$_{\mathtt{Int}}$: $\quad \forall_{\mathsf{a}}[\mathtt{fint\ x\ =\ gint\ x}]$

same$_{[\mathsf{a}]}$: $\quad \forall_{\mathsf{a}}[\mathsf{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow \forall_{\mathsf{x:a}}[\mathtt{ev}_{\mathsf{f::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]$
$\qquad\qquad \rightarrow \forall_{\mathsf{x:[a]}}[\mathtt{flist}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}\ =\ \mathtt{glist}\ \mathtt{ev}_{\mathsf{f::a}}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]$

same$_{\mathtt{Tree\ a}}$: $\forall_{\mathsf{a}}[\mathsf{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow \forall_{\mathsf{x:a}}[\mathtt{ev}_{\mathsf{f::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]$
$\qquad\qquad \rightarrow \forall_{\mathsf{x:\mathtt{Tree}\ a}}[\mathtt{ftree}\ \mathtt{ev}_{\mathsf{f::a}}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}\ =\ \mathtt{gtree}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]$

The goal **same$_{[\mathsf{a}]}$** (and **same$_{\mathtt{Tree\ a}}$**) now has a hypothesis that can be used to prove the goal using the already available tactics. Hence, by taking multiple class constraints into account the problem is solved.

1.7 IMPLEMENTATION

As a proof of concept, we have implemented the **(multi-tactic)** tactic extended for multiple members and subclasses in SPARKLE. Because of the similarity to the already available induction tactic and the clearness of the code, the implementation of the tactic took very little time. However, to implement the tactic, the typing rules had to be extended. The translation of type classes to dictionaries is only typeable in general using rank-2 polymorphism, which is currently not supported by SPARKLE. This was worked around by handling the dictionary creation and selection in a way that hides the rank-2 polymorphism. Ideally, the use of dictionaries should be completely hidden from the user as well.

The tactic has been used to prove, amongst others, the examples in this paper. The implementation is available at:
`http://www.student.kun.nl/ronvankesteren/SparkleGTC.zip`

1.8 RELATED AND FUTURE WORK

As mentioned in section 1.1, the general proof assistant ISABELLE [NPW02] supports overloading and single parameter type classes. ISABELLE's notion of type classes is somewhat 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. ISABELLE [Nip93, Wen97] uses a proof rule based on structural induction on types, which suffices for the supported type classes. However, if ISABELLE would support more extensions, most importantly multi parameter classes, it would be useful to define our proof rule and a corresponding tactic 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 data types [LvE04]. The main goal was to ease proofs by making the induction scheme match the structure of the program. Together with this work this significantly increases the applicability of SPARKLE.

Because generics is often presented as an extension of type classes [HJ00], it would be nice to extend this work to generics as well. Currently, in CLEAN generics are translated to normal type classes where classes are created for every available data type [AP01]. There is a library for HASKELL that generates classes with boilerplate code for every available data type [LJ03]. The tactic presented here can already be used to prove properties about generic functions by working on these generated type classes. However, the property is only proven for the data types that are used in the program and a separate proof is required for each data type. That is, after all, the main difference between normal type classes and generics. Hence, it remains useful to define a proof rule specifically for generics.

1.9 CONCLUSION

In this paper, we have presented a proof rule for class constrained properties and an effective tactic based on it. Although structural induction on types is theoretically powerful enough, we showed that for an effective tactic an induction scheme should be used that is based on the instance definitions. The tactic is effective, because, using the defined proof rule, it allows all sensible hypotheses to be assumed. The rule and tactic were first defined for single class constraints and then generalized to properties with multiple class constraints.

As a proof of concept, the resulting tactic is implemented in SPARKLE for the programming language CLEAN, but it can also be used for proving properties about HASKELL programs. This is, to our knowledge, the first implementation of an effective tactic for general type classes. 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, and Fermín Reig for a valuable discussion on generic proofs at the TFP2004 workshop.

REFERENCES

[AP01] A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. 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 168–185, Älvsjö, Sweden, September 24-26 2001. Springer.

[AV91] J. L. Armstrong and R. Virding. Erlang – An Experimental Telephony Switching Language. In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.

[CFV04] C. Camarão, 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.

[HJ00] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2000.

[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.

[Jon93] M. 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.

[Jon03] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[LJ03] R. Lämmel and S. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, Januari 2003. ACM.

[LvE04] L. Lensink and M. van Eekelen. Induction and Co-induction in Sparkle. In Hans-Wolfgang Loidl, editor, *Fifth Symposium on Trends in Functional Programming (TFP 2004)*, pages 273–293. Ludwig-Maximilians Universität, München, November 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. CUP, 1993.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 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.