Certifying Dictionary Construction in Isabelle/HOL

Lars Hupel

September 13, 2018

Type classes are a well-known extensions to various type systems. Classes usually participate in type inference; that is, the type checker will automatically deduce class constraints and select appropriate instances. Compilers for such languages face the challenge that concrete instances are generally not directly mentioned in the source text. In the runtime, type class operations need to be packaged into dictionaries that are passed around as pointers. This article presents the most common approach for compilation of type classes – the dictionary construction – carried out in a trustworthy fashion in Isabelle/HOL, a proof assistant.

1 Introduction

Isabelle is an interactive theorem prover in the *LCF* tradition [30]: The system is based on a small and well-established kernel implemented in Standard ML. All higher-level specification and proof tools, e.g. for inductive predicates, functional programs, or proof search, have to go through this kernel, by calling the appropriate methods to compose theorems. Roughly speaking, in the context of Isabelle's most commonly-used logic *HOL*, functional programs consist of a series of recursive datatypes and functions, akin to a programming language like Haskell or Standard ML. Additionally, a *code generator* allows users to extract source code in Haskell, OCaml, Scala or ML which can subsequently be compiled and executed [11].

On top of these features, Isabelle's *Pure* logic supports *type classes* [12, 38]. These are built into the kernel and are used extensively in theory developments. The existing generator, when targeting Standard ML, performs the well-known dictionary construction or *dictionary translation* [11]. This works by replacing type classes with records, instances with values, and occurrences with explicit parameters.

Haftmann and Nipkow give a pen-and-paper correctness proof of this construction [11, §4.1], based on a notion of *higher-order rewrite systems.* The resulting theorem then states that any well-typed term is reduction-equivalent before and after class elimination. In this work, the dictionary construction is performed in a certifying fashion, that is, the equivalence is a theorem inside the logic. The key idea is to transform existing definitions using classes into new definitions with additional dictionary parameters.

The construction has been implemented and made available in the *Archive of Formal Proofs* [16], a library of formal developments and tools for Isabelle. It may be used as a preprocessor for Isabelle's code generator. More importantly, it is required for previous work by Hupel and Nipkow, who have described a compilation toolchain from Isabelle/HOL to CakeML [17]. Similarly to Standard ML, CakeML does not support type classes. Because the compilation from Isabelle to CakeML is fully verified, a verified dictionary construction is also required. This paper expands on the final section of the paper by Hupel and Nipkow, by fully describing the underlying construction.

The process is fully automatic: users that want to derive new definitions without sort constraints have to write only a single command.

The paper is structured as follows. I first give a primer on Isabelle notation and terminology. Then, I discuss the key ideas how to represent a type class inside the logic and contrast it with the current treatement of the code generator (§2). This is followed by a detail description of the implementation and Isabelle-specific issues (§3). I conclude with a discussion of remaining problems of the construction (§4).

Notation Type variables are greek letters: α. Constants are typeset in `typewriter` font, variables in *italics*. The function arrow is $\Rightarrow$. Applying a function to arguments is written in standard functional notation; i.e., with spaces instead of parentheses: $f\ x\ y\ z$.

Terminology In Isabelle parlance, the term *class* refers to a type class. Its defining constants are officially referred to as *class parameters*. Classes and parameters live in different name spaces. In the example in Listing 2, `plus` is a parameter of the `plus` class. In this paper, I will use the term *class constant* instead of class parameter, to avoid the ambiguity with function parameters.

A set of classes is called a *sort*.[1] Type variables may carry *sort constraints*, which are preceded by double colons: $\alpha :: \{\texttt{plus}, \texttt{times}\}$. Constants are said to have sort constrains if their types contain type variables with sort constraints.

As an example, consider the function $\texttt{f}\ x = x + x$. In HOL, the $+$ operator is defined in the type class `plus`. This means that the type of `f` is $\alpha :: \texttt{plus} \Rightarrow \alpha$.

Classes can extend other classes, with the inheritance relationship forming a directed acyclic graph (and with it, a partial order). Sorts can be normalized according to this partial order. Assuming that the class `group` extends both `zero` and `plus`, the sort $\{\texttt{group}, \texttt{zero}\}$ can equivalently be written as $\{\texttt{group}\}$. In this paper, I always assume that sorts are *normal*.

Higher-level tools in Isabelle are usually refered to as *packages*. For example, the two facilities that enable functional programming are the **function** and the **datatype** packages [2, 21, 22]. All of these packages have two modes of operation: Isabelle users can use them as regular constructs in an Isabelle theory (see for example Listing 2).

[1]Formally, a sort is an intersection of classes.

```
inductive even :: nat ⇒ nat where
even 0 |
even n  ⟹  even (n + 2)
```

$$\frac{\texttt{even}\ x \qquad P\ 0 \qquad (\forall n.\ \texttt{even}\ n \implies P\ n \implies P\ (\texttt{Suc}\ n))}{P\ x}$$

Listing 1: An example inductive predicate with its induction principle

Additionally, developers of other Isabelle tools can write code that calls the ML interface of those packages for internal constructions. The latter is the primary way these packages are used in this work.

Apart from **function**, which enables definition of recursive functions, there is also the **definition** command. It only supports non-recursive, non-pattern-matching definitions. For the purposes of this paper, their internal implementation differences are not relevant: both allow introducing constants into a theory based on *defining equations.*

The **inductive** package can be used to introduce inductive predicates. An example predicate, classifying even numbers, is given in Listing 1. The two rules declare that 0 is even, and if n is even, so is $n + 2$. The **inductive** command derives a least fixed-point based on these two rules, which are referred to as *introduction rules.* It also generates an induction schema `even.induct` that can be used to prove properties of the form $\texttt{even}\ n \implies Q\ n\ \ldots \implies P\ n$.

Related work Strategies to compile type classes in programming languages have been studied in literature. Type classes have been pioneered by the Haskell programming language [27]. There, a very similar construction is used, replacing classes by records and instances by functions [1, 5, 34]. Additional complications arise because Haskell admits cyclic dependencies between class instances [23], which are prohibited in Isabelle and hence pose no problem for this work. A further simplification compared to Haskell is that Isabelle only features an ML-style simply-typed polymorphic lambda calculus without constructor or multi-parameter classes.

Idris, a dependently-typed functional language, generalizes the concept of type classes to classes that can be parametrized by any value.[2] Still, the elaboration of expressions with class constraints and class and instance declarations is surprisingly similar [4].

In Scala, type classes are represented in an object-oriented style with objects and implicits [33]. Type classes are just regular classes that are subject to the same compilation process to the Java Virtual Machine and other backends. Instances are also regular functions and constraints regular function arguments; however, those arguments are determined automatically by the compiler and do not have to be specified by the programmer. The end result again is similar to that of Haskell and Idris, albeit it is mostly visible in the syntax instead of in some intermediate compiler representation.

[2]In more recent versions of Idris, they are called *interfaces.*

```
class plus =
  fixes plus :: α ⇒ α ⇒ α (infixl + 65)

definition f :: α::plus ⇒ α where
f x = x + x
```

(2.a) Source program (formalized in Isabelle)

```
type 'a plus = {plus : 'a -> 'a -> 'a};
val plus = #plus : 'a plus -> 'a -> 'a -> 'a;

fun f dict x = plus dict x x;
```

(2.b) Target program (Standard ML)

Listing 2: Dictionary construction in Isabelle (current state)

2 Elimination of classes

The basic idea is to replace classes by *dictionaries* containing all class constants and to replace instances by values. Constants with sort constraints are rewritten in a way that they require additional dictionary parameters.

This transformation is an integral part of Isabelle's code generator. It is described in detail by Haftmann and Nipkow [11, §4], together with an informal correctness proof.

A full example for `f` $x = x+x$ is reproduced in Listing 2. Note that this translation is only required for target languages that do not support type classes (OCaml, Standard ML). For other languages (Haskell, Scala), type classes are "passed through" with only minor syntactic changes.

Current state In the code generator, the dictionary construction happens outside the logic. It starts with a set of *code equations* that represent the program to be exported. These code equations are proper theorems and are generated automatically by various commands for datatype and function definitions. To improve efficiency, the user may provide alternative (verified) code equations, for example, to replace a naive recursive implementation of a function by a more stack-efficient tail-recursive definition.

Then, these equations are internalized into an intermediate language nicknamed *Mini-Haskell*. The dictionary construction then proceeds in this internal language, following the approach outlined by Hall et.al. [13].

2.1 Certifying translation

In this work, dictionary translation is performed *before* internalizing the code equations into Mini-Haskell. It is a procedure implemented in ML which takes existing HOL

```
datatype α dict_plus = mk_plus (param_plus: α ⇒ α ⇒ α)

definition cert_plus :: α::plus dict_plus ⇒ bool where
cert_plus dict = (param_plus dict = plus)

fun f' :: α dict_plus ⇒ α ⇒ α where
f' dict x = param_plus dict x x

lemma f'_eq: cert_plus dict ⟹ f' dict = f
(* proof omitted *)
```

Listing 3: Source program after dictionary construction in HOL (certifying translation)

definitions and derives new HOL definitions, coupled with theorems certifying their equivalence.

To continue with the above example: My mechanism introduces a derived constant $\texttt{f}'$ with an additional dictionary parameter $\mathit{dict} :: \alpha\ \texttt{dict\_plus}$. Then, it proves a theorem stating that for any *valid* dictionary dict, $\texttt{f}'$ is equivalent to $\texttt{f}$:

$$\texttt{cert\_plus}\ \mathit{dict} \implies \texttt{f}'\ \mathit{dict} = \texttt{f}$$

Validity of a dictionary is captured by the `cert_plus` predicate. Intuitively, $\texttt{cert\_}c\ \mathit{dict}$ means that *dict* represents a known and lawful instance of class c. The precise notion of "validity" is mainly dictated by technical considerations and discussed in the following section.

Additionally, for each type class instance $\kappa :: (s_1, \ldots, s_k)\ c$, where κ is an k-ary type constructor and $s_i = \{c_{i,1}, \ldots, c_{i,n_i}\}$ are sorts, a new constant $\texttt{inst\_}c\texttt{\_}\kappa$. Given the dictionaries for the s_i, it computes the dictionary for $\kappa :: c$. Its correctness theorem is of the form

$$\begin{aligned}
&\texttt{cert\_}c_{1,1}\ \mathit{dict}_{1,1} \implies \cdots \implies \texttt{cert\_}c_{1,n_1}\ \mathit{dict}_{1,n_1} \implies \\
&\qquad\qquad \cdots \implies \cdots \\
&\texttt{cert\_}c_{k,1}\ \mathit{dict}_{k,1} \implies \cdots \implies \texttt{cert\_}c_{k,n_k}\ \mathit{dict}_{k,n_k} \implies \\
&\qquad\qquad\qquad \texttt{cert\_}c\ (\texttt{inst\_}c\texttt{\_}\kappa\ \mathit{dict}_{1,1}\ \ldots\ \mathit{dict}_{1,n_1}\ \ldots\ \mathit{dict}_{k,1}\ \ldots\ \mathit{dict}_{k,n_k})
\end{aligned}$$

For both instances and constants, each constituent class of each type variable's sort constraints gets assigned a dictionary argument and a premise certifying its validity.

The resulting program (as it would have been written by a user) is reproduced in Listing 3. My procedure defines the types and constants through the ML interfaces of various Isabelle packages, that is, users never see its results directly. Instead, users would write **declassify** `f`, which is a command that has the same effect as the hand-written definitions in Listing 3.

2.2 Possible encodings

The choice of the representation of dictionaries is straightforward: We can model it as a datatype, along with functions returning values of that type. The alternative here would have been to use Isabelle's extensible records [29]. The obvious advantage of records is that we could easily model subclass relationships through record inheritance. However, records do not support multiple inheritance. Consequently, records offer no advantage over datatypes. Instead, I opted for the more modern **datatype** command [2]. As of February 2018,[3] I have also introduced a **datatype_record** command that provides a subset of the syntax of records, but internally constructs a BNF-based datatype.

A more controversial design question is how to represent dictionary certificates. For example, given a value of type `nat dict_plus`, how can one know that this is a faithful representation of the `plus` instance for `nat`?

1. Florian Haftmann, in private communication, proposed a "shallow encoding." It works by exploiting the internal treatment of constants with sort constraints in the Isabelle kernel. Constants themselves do not carry sort constraints, only its definitional equations. The fact that a constant only appears with these constraints on the surface of the system is a feature of type inference.

   Instead, we can instruct the system to ignore these constraints. Isabelle's logic supports definitions of *subtypes*: a type copy of an existing type that imposes additional constraints on values. For example, non-empty lists can be defined as a copy of lists with the constraint $xs \neq []$. Because HOL is a total logic, i.e., all types are non-empty, the system demands a witness satisfying the constraint.

   Applied to this situation, the key idea is to introduce a new type with a parameter α and the constraint that α implements a type class. However, this is ultimately futile: The nonemptiness proof requires a witness of a valid dictionary for an arbitrary, but fixed type α, which is of course not possible, because type classes in general cannot be instantiated for all types.[4]

2. The certificates contain the class axioms directly. For example, the `semigroup_add` class requires $(a + b) + c = a + (b + c)$. Translated into a definition, this would look as follows:[5]

   ```
   definition cert_plus :: α dict_plus ⇒ bool where
   cert_plus dict = (∀x y z. param_plus dict (param_plus dict x y) z =
                                      param_plus dict x (param_plus dict y z))
   ```

   Proving that instances satisfy this certificate is trivial. However, the equality proof of a constant before and after the construction is impossible: they are

[3] http://isabelle.in.tum.de/repos/isabelle/rev/7929240e44d4

[4] The technical details of the construction can be found in the Archive of Formal Proofs: https://www.isa-afp.org/browser_info/current/AFP/Dict_Construction/Impossibility.html

[5] In fact, a similar definition is automatically generated as `class.c` when defining a type class `c` with axioms.

simply not equal in general. Nothing would prevent someone from defining an alternative dictionary using multiplication instead of addition and the certificate would still hold; but obviously functions using `plus` on numbers would expect addition. Intuitively, this makes sense: the above notion of "certificate" establishes no connection between original instantiation and newly-generated dictionaries.

   Instead of proving equality, one would have to "lift" all existing theorems over the old constants to the new constants. This requires proof terms and replaying all proofs accordingly, which would be prohibitively expensive.

3. In order for equality between new and old constants to hold, the certificate needs to capture that the dictionary corresponds exactly to the class constants. This is achieved by the representation in Listing 3. It literally states that the fields of the dictionary are equal to the class constants. The condition of the resulting equation can only be instantiated with dictionaries corresponding to existing class instances. This constitutes a *closed world* assumption, i.e., callers of generated code may not invent own instantiations.

Our choice of representation is the third of these possibilities: We expect dictionaries to be identical to the class constants. For the user, that means that the conditions of the equivalence theorems ($\mathtt{f}'\ \mathit{dict} = \mathtt{f}$) can only be instantiated with existing class instantiations. Unconditional equivalences can be achieved by monomorphizing constants. Applied to the example in Listing 2, that would mean defining a constant $\mathtt{f}_{\mathtt{nat}} :: \mathtt{nat} \Rightarrow \mathtt{nat}$. Its correctness theorem is unconditional, because there are no more sort constraints "on the surface."

3 Implementation

In this section, I will describe the mechanism that transforms definitional equations, followed by highlighting technical challenges. The transformation is similar to the one described by Haftmann and Nipkow [11, §4], which is presently used by the code generator to target OCaml and Standard ML.

Translating constants is the top-level operation in the dictionary construction. The user invokes it with a set of constants. Internally, the procedure uses existing mechanisms in Isabelle to obtain the *code graph* of that set. That graph contains all (definitional) code equations of the set and of all its transitive dependencies (i.e., other constants). Each of these dependencies has to be re-defined as a new constant in some way, depending on whether or not it is a class constant.

Strictly speaking, data constructors are also constants that may have class constraints. The dictionary construction does not support those in general. Additionally, the underlying BNF-based type definition largely ignores sort constraints.[7] Conse-

[6]Readers familiar with Isabelle's internals will notice that the code graph has been slightly redacted: The zero constructor for `nat` is actually the overloaded constant `zero` from the type class `zero`. This introduces technical complications, but does not in principle affect the dictionary construction.

[7]https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2018-May/msg00065.html

```
class plus =
  fixes plus :: α ⇒ α ⇒ α

instantiation nat :: plus
begin

fun plus_nat where
0 + n = (n::nat)
Suc m + n = Suc (m + n)

instance ..

end

definition f :: α::plus ⇒ α where
f x = x + x

(* f specialized to nat *)
definition g :: nat ⇒ nat where
g x = f x
```

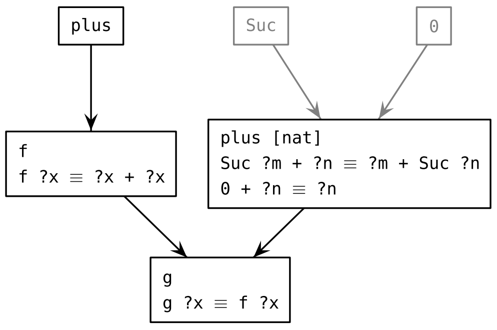


Figure 1: A slightly extended (from Listing 2) source program and its code graph[6]

quently, they do not participate in the dictionary construction at all and are not relevant for this section.

Along the way, auxiliary objects have to be defined, for example the dictionary types for classes. As opposed to the existing code generator, all of these steps have to be carried out inside the logic and are hence bound by its constraints. Most notably, all definitions have to be sequentialized to avoid forward references. This means the implementation comprises mutually recursive, state-updating functions.

The code graph of a small program is given in Figure 1. As can be seen, the constant `g` depends on the constant `f` and the instance `nat` :: `plus`. The data constructors `Suc` and `0` are greyed out. The graph has to be traversed in topological order.

Throughout this section, the overloaded notations $[\![\cdot]\!]$ and $(\!|\cdot|\!)$ are used to describe the translation of various kinds of objects. I will first explain how types and classes themselves are processed. Then, assuming a translation for terms exists, I will give a translation for type schemes and constants. Lastly, the "knot is tied" by explaining how terms are processed. In the actual implementation, all of these steps are intertwined.

Types Isabelle distinguishes between *type variables* and *schematic type variables.* The former are fixed and cannot be instantiated, whereas the latter can. In the theory of simple types, schematic type variables are those type variables that are quantified

```
datatype α dict_c = mk_c
  (super_c₁: α dict_c₁) (super_c₂: α dict_c₂) ... (super_cₙ: α dict_cₙ)
  (param_f₁: ⟦τ₁⟧) (param_f₂: ⟦τ₂⟧) ... (param_fₘ: ⟦τₘ⟧)

definition cert_c :: α::c dict_c ⇒ bool where
cert_c dict =
  (cert_c₁ (super_c₁ dict) ∧ cert_c₂ (super_c₂ dict) ∧ ... cert_cₙ (super_cₙ dict) ∧
   param_f₁ dict = f₁ ∧ param_f₂ dict = f₂ ∧ ... ∧ param_fₘ dict = fₘ)
```

Listing 4: Dictionary datatype and certificate predicate

in a *type scheme* (or *polytype* in more modern literature) [14, 26].

Simple types, i.e., types that contain no schematic type variables, can be translated very easily: $[\![\tau]\!]$ forgets all sort constraints. This is possible because those cannot have intrinsic sort constraints; those are imposed from the context and will be introduced accordingly when dealing with type schemes, which will be explained later.

Classes A class c over a type variable α may have superclasses $c_1, c_2, \ldots, c_n$ and constants $f_1{::}\tau_1, \ldots, f_m{::}\tau_m$. Assuming the set $\{c_1, c_2, \ldots, c_n\}$ is normal, this generates the definitions in Listing 4. Note that the only type variable that may occur in the τ_i is α itself, which is an Isabelle restriction. Consequently, it is not necessary to perform a recursive dictionary translation on the class constants, and we can get away with using the translation for simple types.

This newly-introduced constructor and its fields have the following types:

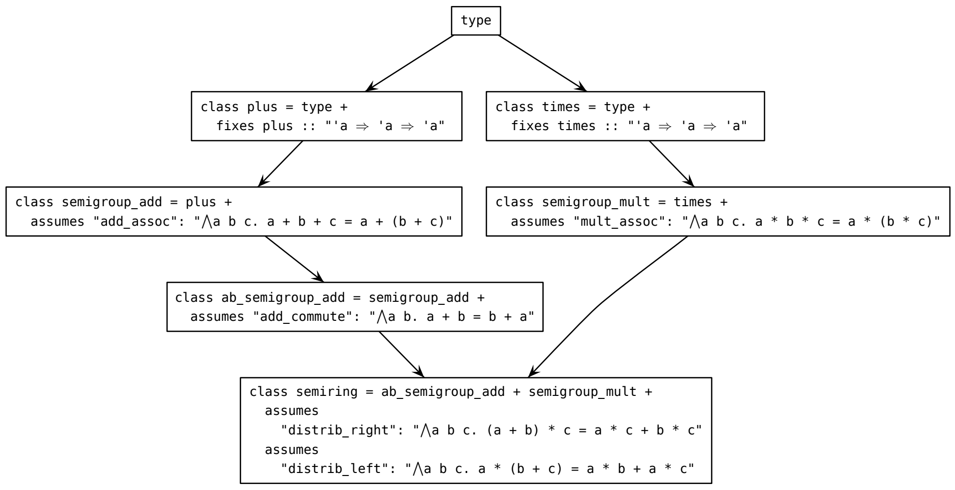
$$\begin{aligned}
\mathtt{mk\_}c &:: \alpha\ \mathtt{dict\_}c_1 \Rightarrow \ldots \Rightarrow \alpha\ \mathtt{dict\_}c_n \Rightarrow \alpha\ \mathtt{dict\_}c \\
\mathtt{const\_}f_i &:: \alpha\ \mathtt{dict\_}c \Rightarrow [\![\tau_i]\!] \\
\mathtt{super\_}c_i &:: \alpha\ \mathtt{dict\_}c \Rightarrow \alpha\ \mathtt{dict\_}c_i \\
\mathtt{cert\_}c &:: \alpha :: c \Rightarrow \mathtt{bool}
\end{aligned}$$

Apart from the certificate definition (which is only required for the correctness proofs), no sort constraints are left.

For any class constant f of a class c, let $(\!|f|\!)$ denote the corresponding constant field. If d is a direct superclass of c, we use $(\!|c \rightsquigarrow d|\!)$ to denote the corresponding superclass field. In other words, $(\!|f|\!) = \mathtt{c.const\_}f$ and $(\!|c \rightsquigarrow d|\!) = \mathtt{c.super\_}d$.[8]

Superclass paths The class hierarchy in HOL is rather intricate. An excerpt, relating to the running example, is reproduced in Figure 2. For example, to obtain the `plus` operation from a `semiring` constraint, one has to follow three subclass–superclasses edges.

[8]In Isabelle, fields of a datatype can be qualified by prefixing the name of a field with the name of the datatype.



```
type

class plus = type +
  fixes plus :: "'a ⇒ 'a ⇒ 'a"

class times = type +
  fixes times :: "'a ⇒ 'a ⇒ 'a"

class semigroup_add = plus +
  assumes "add_assoc": "⋀a b c. a + b + c = a + (b + c)"

class semigroup_mult = times +
  assumes "mult_assoc": "⋀a b c. a * b * c = a * (b * c)"

class ab_semigroup_add = semigroup_add +
  assumes "add_commute": "⋀a b. a + b = b + a"

class semiring = ab_semigroup_add + semigroup_mult +
  assumes
    "distrib_right": "⋀a b c. (a + b) * c = a * c + b * c"
  assumes
    "distrib_left": "⋀a b c. a * (b + c) = a * b + a * c"
```

Figure 2: Class hierarchy for the `comm_semigroup_add` class

In general, for any two classes c and d, there may be multiple different paths from the subclass c and the (possibly indirect) superclass d. It is not obvious that the choice of path is irrelevant for the semantics of the generated program, i.e., that the system is *coherent* according to Jones [20]. Isabelle's type system guarantees coherence [31,32], which the dictionary construction assumes. If that assumption were violated, the equivalence proof (§3.3) would fail. Coherence is consequently a meta-theorem and not internalised in the logic.

For the purpose of this presentation, it is sufficient to assume that the implementation uses the "first" path according to the kernel-defined order of superclasses. It is straightforward to extend the notation $(\!|c \rightsquigarrow d|\!) :: \alpha\ \texttt{dict\_}c \Rightarrow \alpha\ \texttt{dict\_}d$ for an indirect superclass d of c, where the edges are conjoined using the function composition operator $\circ$.

Non-class constants Class constants can be easily distinguished from non-class ones: The former have no defining equations. They are only given meaning by an instance of a class.

In the example in Figure 1, the constants `f`, `g` and `plus_nat` are non-class constants, whereas `plus` is a class constant. This is reflected in the graph: the box for `plus` has no defining equations. Note that while `plus_nat` participates in the instantiation of a type class, it is itself not considered to be a class constant.

Assume we want to transform a non-class constant f with a set of defining equations eq_i. Each of the eq_i is of the form $f\ p_{i,1}\ p_{i,2}\ \ldots\ p_{i,n_i} \equiv rhs_i$, with the $p_{i,j}$ being constructor patterns. Furthermore, the type of f is a type scheme, i.e., it is of the form $\forall \alpha_1 :: s_1 \ldots \forall \alpha_k :: s_k.\ \tau$. Each of the schematic type variables α_i may carry a sort constraint $s_i = \{c_{i,1}, \ldots, c_{i,m_i}\}$ that is assumed to be normal.

Let $[\![t]\!]_\Gamma$ denote the translation of terms in a context Γ (to be defined later). Also, let $(\!|f|\!)$ mean a fresh name, e.g. f' to refer to the newly-defined constant.

I can now explain the translation of the defining equations. Each equation eq_i gives rise to a new equation $[\![eq_i]\!]$ as follows:

- For every class constraint of every type variable, a new parameter is introduced.
- The existing parameters stay unchanged, because data constructors do not participate in the dictionary construction.
- The right-hand side is translated with all new parameters as context.

Formally:

$$\begin{aligned}
[\![eq_i]\!] &= (lhs'_i \equiv [\![rhs_i]\!]_\Gamma) \\
\Gamma &= [dict\_c_{1,1}, \ldots, dict\_c_{k,m_k}] \\
lhs'_i &= (\!|f|\!)\ (dict\_c_{1,1} :: \alpha_1\ \texttt{dict\_}c_{1,1})\ \ldots\ (dict\_c_{k,m_k} :: \alpha_k\ \texttt{dict\_}c_{k,m_k})\ p_{i,1}\ p_{i,2}\ \ldots\ p_{i,n_i}
\end{aligned}$$

Note that the translation for left-hand and right-hand sides differs: left-hand sides, consisting only of patterns, need no context.

All resulting equations are considered as defining equations for $(\!|f|\!)$. Subsequently, they are fed into the internal interface of the **function** command to produce a new logical constant. The additional technical challenges of this are documented in the following sections.

Instance definitions and composition An instance $\kappa :: (s_1, \ldots, s_k)\ c$ is treated as if it is a (non-class) constant with no arguments, returning a dictionary containing instantiations of all class constants. Consequently, each instance gives rise to a new definition that we refer to as $(\!|\kappa :: c|\!)$.

However, it is also necessary to compose instances from contexts. This might mean a combination of following along superclass paths and applying instance definitions to arguments. We use $[\![\tau :: c]\!]_\Gamma$ as notation for this, where τ is a simple type and Γ a context.

I will first describe the (deterministic) algorithm to obtain $[\![\tau :: c]\!]_\Gamma$.

1. If τ is a type variable, find an instance *dict* for $\tau :: c'$ in Γ where c' is a subclass of c. Then, $[\![\tau :: c]\!]_\Gamma = (\!|c' \rightsquigarrow c|\!)\ dict$.
2. Otherwise, τ is of the form $(\tau_1, \ldots, \tau_k)\ \kappa$, i.e., a k-ary type constructor κ applied to k types. Find an instance definition $\kappa :: (s_1, \ldots, s_k)\ c'$ where:
   - c' is a subclass of c and
   - for each constraint $\tau_i :: c_{i,j}$ stemming from the s_i, $r_{i,j} = [\![\tau_i :: c_{i,j}]\!]_\Gamma$ is defined

   Then, $[\![\tau :: c]\!]_\Gamma = (\!|c' \rightsquigarrow c|\!)\ ((\!|\kappa :: c'|\!)\ r_{1,1}\ \ldots\ r_{k,m_k})$.
3. If no suitable instance exists, fail.

For any well-sorted judgement $\tau :: c$, this algorithm is guaranteed to find at least one composed instance. Similar to finding superclass paths, the choice of instance is irrelevant. This is a meta-theorem based on the *coregularity* property that is guaranteed by Isabelle's type system [31,32].

It remains to treat instance definitions $(\!|\kappa :: c|\!)$. Assuming the same naming conventions as above, the generated definition is of the following form:

$$(\!|\kappa{::}c|\!)\ dict_{1,1}\ \ldots = \texttt{mk\_}c\ [\![(\alpha_1,\ldots,\alpha_k)\ \kappa{::}c_1]\!]_\Gamma\ \ldots\ [\![(\alpha_1,\ldots,\alpha_k)\ \kappa{::}c_n]\!]_\Gamma\ [\![f_1]\!]_\Gamma\ \ldots\ [\![f_m]\!]_\Gamma$$

Terms We define the translation of terms $[\![t]\!]$ that are not constants recursively as follows:

$$\begin{aligned} [\![x]\!]_\Gamma &= x && \text{(where } x \text{ is a variable)} \\ [\![t\ u]\!]_\Gamma &= [\![t]\!]_\Gamma\ [\![u]\!]_\Gamma \\ [\![\lambda x.\, t]\!]_\Gamma &= \lambda x.\, [\![t]\!]_\Gamma \end{aligned}$$

The rule for constants is a bit more involved. Let f be a constant with k type parameters, i.e., of type scheme $\forall\alpha_1 :: s_1 \ldots \forall\alpha_k :: s_k$. In any occurrence of f in a term, these type parameters are instantiated with simple types $\tau_1, \ldots, \tau_k$.

$$[\![f]\!]_\Gamma = (\!|f|\!)\ [\![\tau_1 :: c_{1,1}]\!]_\Gamma\ \ldots\ [\![\tau_k :: c_{k,m_k}]\!]_\Gamma$$

Challenges In the standard case, where the user has not performed a custom code setup, the resulting function looks similar to its original definition. But the user may have also changed the implementation of a function significantly afterwards. This poses some challenges:

- The new constants need to be proven terminating. The routine applies some heuristics to transfer the original termination proof to the new definitions (§3.1). This only works when the termination condition does not rely on class axioms.
- The domain of functions must be tracked, because even though HOL is a total logic, functions may be under-specified. Congruence rules are used to construct an inductive predicate representing the *side condition* of a function (§3.2).
- In order to fine-tune executable code, the code generator allows users to specify different constructors of a datatype than those it has been defined with, or even to introduce constructors for non-datatypes. However, the **function** command does not support that in general (§4).

3.1 Preservation of termination

As indicated above, the newly-defined functions must be proven terminating. In general, we cannot reuse the original termination proof, as the example in Listing 5 illustrates. While the original function is primitively-recursive, and hence trivially proved to be terminating, the user has added a code equation that characterizes a

```
fun f :: nat ⇒ nat where
f 0 = 0
f (Suc n) = f n

lemma [code]: f x = f x by simp
```

Listing 5: Pathological example of a non-terminating code equation

non-terminating implementation. My construction cannot deal with such pathological cases, but fortunately they are rare in practice. The invocation of the dictionary construction would just fail for this example.

Instead, based on my experience, the most common cases are that users either

- do not adapt the code equations at all,
- adapt them without changing the termination scheme, or
- adapt them to use different recursive calls, while still being terminating.

For the last case, it is impossible to port the existing termination proof, because it is not applicable any more. Hence, the construction falls back to use the same automated proof method as the **function** package.

However, the other cases are more interesting. In the remainder of this section, I will illustrate the first case, which is a specialization of the second one. The original termination proof should "morally" be still applicable.

The running example will be a function that sums up values in a list. The empty list is denoted by [] and a cons cell – a pair of head and tail – by the operator #.

```
fun sum_list :: α::{plus,zero} list ⇒ α where
sum_list [] = 0
sum_list (x # xs) = x + sum_list xs
```

This function carries two distinct class constraints – arising from the use of addition and zero, both of which are provided by a class in Isabelle – which are translated into two dictionary parameters:

```
sum_list' dict_plus dict_zero [] =
    param_zero dict_zero
sum_list' dict_plus dict_zero (x # xs) =
    param_plus dict_plus x (sum_list' dict_plus dict_zero xs)
```

Here, the termination argument has not changed: While two additional parameters have been introduced, they remain unchanged in between recursive calls. Observe that – whenever sort constraints are present – the dictionary construction always introduces new arguments, but keeps the termination scheme.

We now have to carry out the termination proof of `sum_list′`. The **function** package analyses the structure of recursive calls and collects them into a set of constraints.

As a notation for constraints, I will use $\bar{p} \rightsquigarrow \bar{x}$. $\bar{p}$ stands for the (tupled) patterns on the left-hand side of an equation and $\bar{x}$ for the (also tupled) actual parameters passed to a recursive invocation.

For the above example, this looks as follows:

$$\{(x \mathbin{\#} xs) \rightsquigarrow xs\} \qquad (\texttt{sum\_list})$$
$$\{(\mathit{dict\_plus}, \mathit{dict\_zero}, x \mathbin{\#} xs) \rightsquigarrow (\mathit{dict\_plus}, \mathit{dict\_zero}, xs)\} \qquad (\texttt{sum\_list}')$$

Internally, for every function $f :: \sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau$, the package defines an inductive relation f`_rel` $:: (\sigma_1, \sigma_2, \ldots, \sigma_n) \Rightarrow (\sigma_1, \sigma_2, \ldots, \sigma_n) \Rightarrow$ `bool` with one introduction rule per constraint. Note that the arguments are tupled, i.e. all function arguments participate in the definition of this *termination relation.*

In our example, the predicate `sum_list_rel` is defined by the following introduction rule:

$$\frac{}{\texttt{sum\_list\_rel}\ xs\ (x \mathbin{\#} xs)}$$

For details on how the **function** package assembles the termination relation based on the constraints, in particular for more complicated recursion schemes, refer to Krauss' thesis [21].

To prove that a function terminates, it is sufficient to show that its termination relation is *well-founded.* In the majority of cases, this happens by supplying a suitable *measure function* that maps the arguments to natural numbers and decreases for each recursive call. The **function** package is able to try out various measure functions automatically.

In this setting however, the termination of f has already been proved, either automatically or by the user. The construction tries to re-use that proof, i.e., the well-foundedness theorem of f`_rel`, for the proof of well-foundedness of f'`_rel`, where f' is the result of applying the dictionary construction to f. Except for the additional (unchanging) dictionary arguments, these relations are more or less equivalent to each other.

Theorem 1 (Well-founded simulation)**.** *Let* $P :: \tau \Rightarrow \tau \Rightarrow$ `bool` *be a well-founded relation and* $g :: \sigma \Rightarrow \tau$ *a function such that*

$$\forall x\ y.\ P'\ x\ y \Longrightarrow P\ (g\ x)\ (g\ y)$$

Then, $P' :: \sigma \Rightarrow \sigma \Rightarrow$ `bool` *is also a well-founded relation.*

This theorem allows us to *simulate* the structure of the recursive calls of f' with those of f (depicted in Figure 3). There is an important difference, though: f`_rel` may have sort constraints, f'`_rel` does not.

Instantiating the above lemma with the two termination relations entails choosing a suitable function g that maps arguments of `sum_list′` to arguments of `sum_list`, i.e., a function of type

$$(\alpha\ \texttt{dict\_plus} \times \alpha\ \texttt{dict\_zero} \times \alpha\ \texttt{list}) \Rightarrow \beta :: \{\texttt{plus}, \texttt{zero}\}\ \texttt{list}$$

$$\begin{tikzcd}
P & x_1 \arrow[r] & x_2 \arrow[r] & x_3 \arrow[r] & \cdots \\
P' & x'_1 \arrow[r] \arrow[u, dashed, "g"'] & x'_2 \arrow[r] \arrow[u, dashed, "g"'] & x'_3 \arrow[r] \arrow[u, dashed, "g"'] & \cdots
\end{tikzcd}$$

Figure 3: Well-founded simulation

for an arbitrary type β. Obviously, g can drop the first two elements of the tuple. The challenge arises when we try to map a list with element type α to one with element type β. We cannot instantiate $\beta = \alpha$, because β carries a sort constraint.

In a parametric setting, this would be the end of it, because it is impossible to write such a function [19, 24, 35]. Isabelle however offers us an escape hatch: recall that all types are non-empty. The polymorphic constant $\texttt{undefined} :: \alpha$ can serve as a witness for an arbitrary type. Assuming that there is at least one concrete type τ that satisfies the sort constraints of β, we can instantiate $\beta = \tau$. The desired mapping function can now be specified as follows:

$$g\ (\_, \_, \mathit{xs}) = \texttt{map}\ (\lambda\_.\ \texttt{undefined} :: \tau)\ \mathit{xs}$$

In case there is no such concrete τ, the above expressions fails to type check, causing the heuristics to fail.

It remains to show how the premise of the well-founded simulation theorem is proved in this particular case:

$$\forall x\ y.\ \texttt{sum\_list'\_rel}\ x\ y \Longrightarrow \texttt{sum\_list\_rel}\ (g\ x)\ (g\ y)$$

The proof proceeds by induction using the induction principle of the $\texttt{sum\_list'\_rel}$ inductive predicate, which gives one case per introduction rule, that is:

$$\forall \mathit{d\_plus}\ \mathit{d\_zero}\ x\ \mathit{xs}.\ \texttt{sum\_list\_rel}\ (g\ (\mathit{d\_plus}, \mathit{d\_zero}, \mathit{xs}))\ (g\ (\mathit{d\_plus}, \mathit{d\_zero}, x \mathbin{\#} \mathit{xs}))$$

After unfolding the definition of g, this turns into:

$$\forall \mathit{xs}.\ \texttt{sum\_list\_rel}\ (\texttt{map}\ (\lambda\_.\ \texttt{undefined})\ \mathit{xs})\ (\texttt{undefined} \mathbin{\#} \texttt{map}\ (\lambda\_.\ \texttt{undefined})\ \mathit{xs})$$

This can now trivially be proved by using the introduction rule of $\texttt{sum\_list\_rel}$.

More generally, this construction allows the proof of well-foundedness of any relation

$$R' :: (\beta_1 \times \ldots \times \beta_n \times (\alpha_1, \ldots, \alpha_k)\ \tau) \Rightarrow (\beta_1 \times \ldots \times \beta_n \times (\alpha_1, \ldots, \alpha_k)\ \tau) \Rightarrow \texttt{bool}$$

given a well-founded relation

$$R :: (\alpha_1, \ldots, \alpha_k)\ \tau' \Rightarrow (\alpha_1, \ldots, \alpha_k)\ \tau' \Rightarrow \texttt{bool}$$

where τ is a suitable type constructor equipped with a functorial $\mathtt{map}_\tau$,[9] τ and τ' differ only in sort constraints, R and R' are structurally equivalent and parametric in all α_i. The mapping function g is defined as follows:

$$g :: (\beta_1 \times \ldots \times \beta_n \times (\alpha_1, \ldots, \alpha_k)\ \tau) \Rightarrow (\alpha_1, \ldots, \alpha_k)\ \tau'$$
$$g\ (\_, \ldots, \_, t) = \mathtt{map}_\tau\ \underbrace{(\lambda\_.\ \mathtt{undefined})\ \ldots\ (\lambda\_.\ \mathtt{undefined})}_{\text{one for each } \alpha_i}\ t$$

3.2 Partially-specified functions

HOL is a total logic, that is, we can always assign a value to a function $f :: \alpha \Rightarrow \beta$ applied to any argument $x :: \alpha$. This immediately raises the question how to represent *partially-specified* (or *under-specified*) functions, e.g. to obtain the head of a list:

```
fun hd :: α list ⇒ α where
hd (x # xs) = x
```

Obviously, in this function definition, the case for the empty list is omitted. Note that under-specification and non-termination are different kinds of partiality; the latter of which is not supported by this work. There are various ways to deal with under-specification:

1. Lift the result type into `option`, i.e. $\alpha \Rightarrow \beta$ turns into $\alpha \Rightarrow \beta$ `option`. Arguments for which the function is not specified get assigned a `None` value. The domain of the function $\mathtt{dom}_f$ can conveniently be expressed as a set $\{x \mid f\ x \neq \mathtt{None}\}$.

2. Function definitions are "artificially" completed to be always specified. In HOL, `undefined` is an unspecified constant of arbitrary type. The meta-theorem about this unspecified constant is that for all predicates P, P `undefined` is provable if and only if $P\ x$ is provable for all x.

3. Based on the function equations, derive a set carrying the *side condition* of the function. We allow reasoning over a function application $f\ x$ only if $x \in \mathtt{side}_f$ holds. This is comparable to refinement types [6], but where the constraints are external and not part of the type.

A more detailed survey can be found in the introduction of the paper by Finn *et al.* [9].

Let us examine the specification of the `hd` function for each of the different approaches.

Lifting This approach is preferred in many functional programming languages, like Haskell, where types may be non-empty (ignoring exceptions). The major problem is that it may lead to complicated proof statements when reasoning about such functions.

[9] The precise nature of "suitability" is not relevant for the discussion. In the implementation, bounded natural functors as introduced by Blanchette *et al.* [2, 3] without dead variables are considered suitable.

```
fun hd :: α list ⇒ α option where
hd (x # xs) = Some x
hd [] = None
```

In our setting, this would require non-trivial transformation of existing code equations. Wimmer *et al.* [39] solve a similar problem in the context of memoization: they lift functions into the state monad. The main weakness is that higher-order functions need to be lifted manually. Because many existing Isabelle formalizations make use of custom combinators, their approach is not feasible here.

Completion Isabelle's **function** package uses this approach by default. For any given function definition, a catch-all clause is added (a process called *completion*):

```
fun hd :: α list ⇒ α where
hd (x # xs) = x
hd _ = undefined
```

As far as the **function** package is concerned, this function is now specified for all input values.

To avoid leaking this implementation detail to users, Isabelle's simplifier will not rewrite the term `hd` [] to `undefined`. But the completion is visible in the generated induction principle `hd.induct`:

$$\frac{\forall x\ xs.\ P\ (x \mathbin{\#} xs) \qquad P\ []}{P\ a}$$

The premise $P\ []$ is necessary for this theorem to hold. Otherwise, $P\ xs = (xs \neq [])$ would be a counterexample.

While this approach is conceptually simple, it poses a significant challenge for the dictionary construction and associated proof tactics. The reason is as profound as it is technical: Applications of functions to values on which they are not specified are practically "opaque"; in the sense that it is difficult to rewrite or prove anything about them. To make matters worse, identities like $\texttt{undefined}\ x = \texttt{undefined}$ are unprovable, meaning `undefined` behaves differently than e.g. $\perp$ in Haskell (where $\perp\ x = \perp$ holds). It is hence an insufficient approximation of under-specification for the purposes of code generation.

Side conditions Short of modifying the internal mechanics of the **function** package, and consistent with Myreen and Owens' approach [28], I have chosen to track side conditions of functions. They are represented as **inductive** predicates. In the case of the head function, the predicate is specified as follows:

$$\frac{}{\texttt{hd\_side}\ (x \mathbin{\#} xs)}$$

When producing a certificate for the dictionary translation (§2) for an under-specified function `f`, the routine introduces a new premise:

$$\texttt{cert\_plus}\ \mathit{dict} \implies \texttt{f\_side}\ x \implies \texttt{f}'\ \mathit{dict}\ x = \texttt{f}\ x$$

A more subtle change is that the theorem now has to be stated in η-expanded form. This may limit its applicability in higher-order position, e.g. `map f`.

Before describing the construction of the inductive predicates for side conditions, I will rehash the concept of *congruence rules*.

3.2.1 Congruence rules

The notion of *congruence rules* goes back to the literature on term rewriting [7, 8]. Later, they have become instrumental in the context of admitting recursive definitions in higher-order logics [21, 22, 36].

Definition 2 (Congruence rule)**.** *A* congruence rule *for the function* `c` *is a theorem of the form*

$$\frac{P_1 \qquad \cdots \qquad P_n}{\mathtt{c}\; x_1 \;\ldots\; x_n = \mathtt{c}\; y_1 \;\ldots\; y_n}$$

where the P_i *may refer to arbitrary* x_i *and* y_i*.*

Usually, the P_i takes either of these two forms:

- $Q_i \Longrightarrow x_i = y_i$, when $x_i :: \tau$ and τ is not a function type
- $\forall \bar{z}.\ Q_i\ \bar{z} \Longrightarrow x_i\ \bar{z} = y_i\ \bar{z}$, otherwise

Slind [36, §2.7.1] calls a congruence rule where no functions are passed as arguments *simple*. Two examples of those are given below:

$$\mathtt{If}\ \frac{C_1 = C_2 \qquad C_1 \Longrightarrow x_1 = x_2 \qquad \neg C_1 \Longrightarrow y_1 = y_2}{\textbf{if } C_1 \textbf{ then } x_1 \textbf{ then } y_1 = \textbf{if } C_2 \textbf{ then } x_2 \textbf{ then } y_2}$$

$$\mathtt{conj}\ \frac{A_1 = A_2 \qquad A_1 \Longrightarrow B_1 = B_2}{A_1 \wedge B_1 = A_2 \wedge B_2}$$

The purpose of these rules is to track *evaluation context.* This is important because HOL itself has no notion of evaluation order, but the target language[10] does. In particular, both in Slind's **recdef** and Krauss' **function** package, congruence rules are used to determine the termination relation of a function. Both take the Q_i of the rules into account to guard recursive invocations, like in the following example:

```
fun fac :: nat ⇒ nat where
fac n = (if n = 0 then 1 else n * fac (n - 1))
```

A naive termination analysis would complain that `fac` never terminates, because there is always a recursive call. The **function** package however derives the following termination relation:

$$\frac{n \neq 0}{\mathtt{fac\_rel}\ (n-1)\ n}$$

[10]In this case CakeML, but it is also relevant for all other targets from the code generator.

This is clearly well-founded, i.e. `fac` terminates on all inputs, because there is no n' such that `fac_rel` n' 0.

More complex cases arise when higher-order recursion is present. Consider this datatype and function:

```
datatype α tree = Fork (α tree list) | Leaf α

fun map_tree where
map_tree f (Fork ts) = Fork (map (map_tree f) ts)
map_tree f (Leaf x) = Leaf (f x)
```

It takes more work to understand this *nested* recursion principle. It is not directly obvious on which values `map_tree` is called recursively, because it only appears in partially-applied form in the function body. The **function** package uses the higher-order congruence rule for `map` to deduce the termination relation:

$$\frac{\forall x.\ x \in \texttt{set}\ xs \Longrightarrow f\ x = g\ x \qquad xs = ys}{\texttt{map}\ f\ xs = \texttt{map}\ g\ ys}$$

Intuitively speaking, this tells us that the function passed to `map` is applied to each element in the set of xs, where `set` is a function that turns a list into a set. Consequently, the termination relation states exactly that:

$$\frac{t \in \texttt{set}\ ts}{\texttt{map\_tree\_rel}\ (f, t)\ (f, \texttt{Fork}\ ts)}$$

Well-foundedness can be proved by appealing to the size of the arguments. The **datatype** package provides a $\texttt{size}_\tau$ function for each type constructor τ that counts the number of data constructors in the value. Consequently, if t is an element of ts, then the size of t is smaller than the size of `Fork` ts.

All of the necessary infrastructure for this is fully automated in Isabelle:

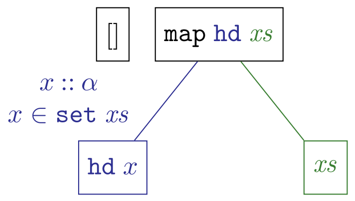
- generation of $\texttt{map}_\tau$, $\texttt{set}_\tau$, and $\texttt{size}_\tau$ functions,
- proof of a suitable higher-order congruence rule,
- setup of the **function** package.

The only occasion when a user has to adjust the setup is when they introduce a custom higher-order recursion combinator, or when a function definition uses a more complicated termination measure than the size of the inputs.

3.2.2 Specifiedness

Congruence rules can also be used to determine on which inputs functions are specified. A similar routine as in the **function** package can be employed to analyse function definitions. Here, the goal is to construct an inductive predicate capturing the set of arguments for which a function is specified.

For example, consider the following (contrived) function definition:

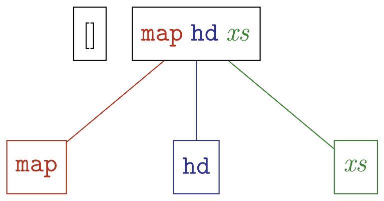


(a) Forest with congruence rule for `map`

$$\frac{}{\texttt{hd\_tl\_side}\ []}$$

$$\frac{\begin{array}{c}\texttt{map\_side hd}\ xs\\ \forall x.\ x \in \texttt{set}\ xs \Longrightarrow \texttt{hd\_side}\ x\end{array}}{\texttt{hd\_tl\_side}\ (x \mathbin{\#} xs)}$$

(b) Side condition with congruence rule for `map`



(c) Forest without congruence rule for `map`

$$\frac{}{\texttt{hd\_tl\_side}\ []}$$

$$\frac{\texttt{map\_side hd}\ xs \qquad \forall x.\ \texttt{hd\_side}\ x}{\texttt{hd\_tl\_side}\ (x \mathbin{\#} xs)}$$

(d) Side condition without congruence rule for `map`

Figure 4: Context forests and resulting side conditions of `hd_tl`

```
fun hd_tl :: α list list ⇒ α list
hd_tl [] = []
hd_tl (x # xs) = map hd xs
```

The function itself has no obvious unspecified behaviour, because all possible inputs are covered by pattern matching. However, the function `hd` is unspecified for empty lists. The desired side condition is:

$$\frac{}{\texttt{hd\_tl\_side}\ []} \qquad \frac{\forall x.\ x \in \texttt{set}\ xs \Longrightarrow \texttt{hd\_side}\ x}{\texttt{hd\_tl\_side}\ (x \mathbin{\#} xs)}$$

This can further be simplified by noting that $\texttt{hd\_side}\ x \Longleftrightarrow x \neq []$. This inductive definition can be obtained by performing a recursive analysis on the defining equations of a constant. Each equation of `f` gives rise to a rule in `f_side`.

Note that as far as Isabelle's total logic is concerned, this function is total: The `hd` function just returns `undefined` on empty lists. Because of this, any notion of *specifiedness* cannot be fully formalized and has to be – to some extent – a heuristic. A good intuition that is that we want to characterize all inputs x to a function `f` such that evaluation after code generation to a target language does not yield a runtime exception.

Transformation to forest The routine starts by converting each right-hand side of all defining equations into a *context tree.* Each node of the tree is labelled with a term and optionally, a congruence rule, and may have arbitrarily many children. An edge from a node to a child is labelled with a *context:* a list of variables and of assumptions.

An example based on the `hd_tl` function is given in 4. It illustrates how the congruence rule of `map` participates in the transformation.

The transformation algorithm itself can be summarized as follows. For a term t, a node is generated. Then, it adds children to the node by case distinction on the shape of t:

- If t is atomic, it becomes a leaf node.
- If t is a function application `f` $x_1 \ \dots \ x_n$, it tries to find a congruence rule that matches the term. k children are added to the node according to the k premises of the rule, tracking their variables and assumptions as context of the respective child. If there is no matching rule, the function and its arguments are considered separately, hence creating $n + 1$ children.
- Otherwise, t is an abstraction $\lambda x.\ u$. One child for u is added with x as context.

In Figure 4a, there are two trees.

- `hd_tl` $[] = []$ gives rise to the tree with just a leaf node $[]$, because $[]$ is an atomic constant.
- `hd_tl` $(x \# xs)$ = `map hd` xs produces a node with two children, after applying the congruence rule for `map`. The left child has a context enriched with a variable and assumption, whereas the right child is the atomic xs.

Ignoring the congruence rule results in the forest in Figure 4c, where three children are generated for the binary call to `map`.

Transformation to predicate Finally, the tree is transformed into a set of introduction rules for the inductive predicate representing the specifiedness. Figure 4b illustrates the result of the transformation. In a tree, each path from root to the side condition generates one assumption in the side condition based on the pre-existing side conditions, unless:

- it is the first child of a function application with no matching congruence rule (nothing is known about that function call), or
- it is a free variable (always specified), or
- no side condition is known about the term (assumed total).

Crucially, each layer of the tree still contributes to the side condition. In the running example, this means that the root node contributes the assumption `map_side hd` xs.

A special case arises in Figure 4d, where an assumption is generated for the node `hd`. Because the forest has been created without a suitable congruence rule, there is no variable in the context of the node. The transformation hence introduces a synthetic variable and universally quantifies over it (marked brown in the figure). It can be proved that $\forall x.$ `hd_side` x is false, because there is a list that violates `hd_side`: the empty list. In general, absence of congruence rules or congruence rules that are too weak may lead to vacuous side conditions.

The side condition for `undefined` is a prime example for being vacuous by definition: there are no defining equations, hence no context trees, hence the inductive predicate is the empty least-fixed point. The **inductive** package admits such empty predicates. They are definitionally equivalent to false, i.e., `undefined_side` $\Longleftrightarrow$ `False`.

Simplification To avoid overly complicated side conditions, there are two strategies to simplify them. The routine tries to:

1. prove totality, i.e. $\forall x_1 \ldots x_n.\ \texttt{f\_side}\ x_1\ \ldots\ x_n$, and
2. discharge auxiliary side conditions, e.g. $\texttt{hd\_side}\ (x \mathbin{\#} xs) = \texttt{True}$.

Both work by suitable preprocessing of the goal, then running Isabelle's full simplifier. While this can make the result unpredictable, I have found that this prevents many redundant assumptions.

For the example in Figure 4, this removes the assumption $\texttt{map\_side}\ \texttt{hd}\ xs$, because `map_side` can be proved to be total.

It is important to note that the generated side conditions are *shallow*, that is, they only characterize the specifiedness of one function, but not any other non-constant functions, i.e. functions that are passed in as parameters, that are called along the way. This is nicely illustrated by this example: While the `map` function is obviously fully specified, it can be used in a partially-specified way; namely, when the mapping function is only partially specified. The challenges to fully capture specifiedness are described in §4.

3.2.3 Differences to Krauss' routine

As indicated above, the **function** package employs a similar routine. The differences are mainly technical in nature, but are significant enough to prevent code reuse.

- The internally produced congruence tree is not exported as a data structure.
- Traversal happens on an intermediate constant that represents all functions in a mutually-recursive bundle with tupled arguments. For example, simultaneous recursive definitions of `odd` and `even` functions would internally be presented as a single function of type $(\texttt{nat} + \texttt{nat}) \Rightarrow (\texttt{bool} + \texttt{bool})$.[11]
- Side conditions of auxiliary constants (in the running example: `map` and `hd`) are not considered: after defining and proving a function to be terminating, it is "total" by virtue of completion.

Notably, the extraction of a termination relation – just like specifiedness – critically depends on the presence of appropriate congruence rules. Similarly to the special case described above, absence of congruence rules may lead to unprovable termination relations. Consequently, it is reasonable to assume that a user of the **function** package is aware of this required setup; hence, it is not an extra burden to require the same setup for the dictionary construction.

[11] $\alpha + \beta$ is the sum type of α and β.

3.3 Correctness proofs

There are two kinds of propositions that need to be proved in the routine: dictionary certificates and equivalence theorems (§2.1). In general, they are of the form:

$$\ldots \Longrightarrow \texttt{cert\_}c\,(\texttt{inst\_}c\texttt{\_}\kappa\ \mathit{dict}_1\ \ldots)$$
$$\ldots \Longrightarrow \texttt{f}'\ \mathit{dict}_1\ \ldots\ x_1\ \ldots = \texttt{f}\ x_1\ \ldots$$

Both can carry preconditions for auxiliary dictionary certificates, and in the case of the equivalence theorems, also side conditions of the arguments x_i (§3.2). The proof strategies for both kinds of theorems differ, so I will discuss them separately. Both strategies have in common that they require the proofs to happen in exactly the same (topological) order as the dictionary construction itself (Figure 1). At any point during a sequence of proofs, the previous correctness theorems are referred to as *base theorems*.

Dictionary certificates Recall the definition of cert_c and $(\!|\kappa :: c|\!)$. The latter is a plain constructor application (mk_c); the former inspects each field. In other words, $(\!|\kappa::c|\!)$ "bundles" existing constants into a dictionary. Consequently, the proof proceeds by simple application of the base theorems.

Equivalence theorems In general, these theorems have to be proved by induction using the induction scheme generated by the side condition, or if the side condition is trivial, the termination relation of the function. Both are similar, so I focus on the latter.

Applying the induction principle creates one proof obligation per defining equation.[12] Recall the sum_list function from §3.1. The proof obligations after induction are (dictionary certificates omitted):

$$\texttt{sum\_list}'\ d_1\ d_2\ [] = \texttt{sum\_list}\ []$$
$$\texttt{sum\_list}'\ d_1\ d_2\ xs = \texttt{sum\_list}\ xs \Longrightarrow \texttt{sum\_list}'\ d_1\ d_2\ (x \mathbin{\#} xs) = \texttt{sum\_list}\ (x \mathbin{\#} xs)$$

These can be discharged by first unfolding the defining equations of sum_list′ and sum_list. Then, base theorems and induction hypotheses are applied by walking the congruence tree. Note that the base theorems include equivalences for class constants and the corresponding dictionary fields.

4 Limitations

Specifiedness A particularly thorny issue is presented by functions that return other functions. While *currying* itself is a common idiom in functional programming, manipulation of partially-applied functions would require a non-trivial data flow analysis.

[12] The reality is a bit more complicated: one code equation may create multiple defining equations, because the **function** command disambiguates equations. For example, consider the definition single $[x]$ = True and single xs = False. The package instantiates the second equation ($xs = y \mathbin{\#} ys$ and $xs = []$) to avoid ambiguities.

```
datatype α list = Cons α (α list) | Nil

fun append :: α list ⇒ α list ⇒ α list where
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

fun Snoc :: α list ⇒ α ⇒ α list where
Snoc xs x = append xs [x]

code_datatype Nil Snoc
```

(6.a) Full definition of a "cons list"

```
datatype 'a list = Snoc of 'a list * 'a | Nil;
```

(6.b) Generated datatype definition in Standard ML

```
lemma [code]:
  append xs Nil = xs
  append xs (Snoc ys y) = Snoc (append xs ys) y
(* proof omitted *)
```

(6.c) Code equations for `append`

Listing 6: Adapting a datatype to a different representation

As a synthetic example, consider the expression `map div` *xs* :: (`nat` ⇒ `nat`) `list`, with `div` being the division operator on natural numbers. Clearly, the expression is fully-specified for all *xs*, but its resulting list of functions is not: Passing in 0 to any of the resulting functions yields unspecified behaviour, or at runtime in a target language, throw an exception.

The underlying problem is that the congruence rule for `map` can only be used to extract side conditions when the return type of the function that is passed to it is not itself a function type. More formally, the heuristics requires that no type variables are instantiated with a function type in order to work correctly.

A similar situation arises in practice in the commonly-used `show` derivation framework by Sternagel and Thiemann [37]. They employ Hughes' difference list representation of strings [15]. Luckily, all these functions are fully-specified, i.e. the side condition is always true.

Patterns in function definitions The **function** package, by default, allows only function definitions where the left-hand side matches on constructors. Consider as an example that we want to treat list as "snoc lists" instead of "cons lists," i.e., a pair of *init* and *last* instead of *head* and *tail*. A full example is given in Listing 6. It first introduces the datatype for "cons lists," and defines the `append` and `Snoc` functions. Then, it instructs the code generator to use `Nil` and `Snoc` in the target language representation.

However, the code generator cannot export code for the `append` function, because it is defined in terms of `Cons`, and aborts with an error message that `Cons` is not a constructor on the left-hand side of the code equation.

To fix this, Listing 6 demonstrates how to add code equations for `append` that are defined in terms of `Snoc`. But now the **function** package would not accept such a definition of `append`, because while `Snoc` is a constructor as far as the code generator is concerned, it is still not a constructor as far as the **datatype** package is concerned. The key problem is that both subsystems may have diverging notions of exactly which constructors a datatype is comprised of.

The workaround for this problem is that it is possible to use the **function** package in a mode which allows to use arbitrary patterns on the left-hand side of a defining equation. It is only a workaround because the package demands some additional proofs (exhaustiveness, well-definedness) that are tedious to do by hand and impossible to automate in general.

For that reason, any type adaptations, including data refinement [10], are not supported by this work.

Preservation of termination The following pathological example exhibits the problem that some functions cannot be proved to terminate after elimination of sort constraints:

```
function sum_set :: α::{finite,comm_monoid_add,linorder} set ⇒ α where
sum_set S = (if S = {} then 0 else Min S + sum_set (S - {Min S}))
```

This function, analogously to `sum_list`, should compute the sum of a set. It can only be proved terminating because of the sort constraints: otherwise, there may be infinitely many elements in S, no well-defined minimum element, or the result may be depending on the order. With the constraint, the termination relation is the cardinality of the set which decreases with each recursive call.

However, the termination heuristics cannot cope with this example. The dictionary construction removes all sort constraints from the type variables; instead introducing value parameters. The dictionary type for `finite` would be isomorphic to `unit`, because the class does not have any parameters. But now, the new termination relation cannot be simulated by the old one: it has to deal with arbitrary, possibly infinite sets.

Fortunately, function definitions like this are rare. If necessary, they can be replaced by a recursion on lists as follows:

```
lemma sum_set_remdups_list_eq[code_unfold]:
  sum_set (set xs) = sum_list (remdups xs)
```

This replaces all occurrences of `sum_set` applied to a finite set of elements xs by an application of `sum_list` (after removing duplicate elements). After (automatic) preprocessing by the code generator, no traces of recursion through sets will be left.

This pattern of replacing logical recursion on sets by executable recursion on lists is common in Isabelle's standard libraries.

Furthermore, the heuristic cannot prove all function definitions that are still terminating after class elimination to be terminating. The following circumstances prevent a termination proof to be ported:

- Recursive calls that receive the result of another recursive call as an argument. A classic example is *McCarthy's 91 function* [25], which is defined as follows:

  $$\texttt{f91}\ n = (\textbf{if}\ 100 < n\ \textbf{then}\ n - 10\ \textbf{else}\ \texttt{f91}\ (\texttt{f91}\ (n + 11)))$$

  The termination proof requires an additional lemma that gives an estimate on the return value. Krauss' **function** package can deal with this specification [21, §2.7.2]. My heuristic cannot, because the generated termination condition mentions the function itself.

- Functions with at least one polymorphic parameter that is not a suitable type constructor. The termination heuristic will ask the **datatype** package to deliver a map function from the new relation to the old relation, which is impossible in some cases.

Should the heuristic fail, the system will fall back to an automatic termination proof using the lexicographic order method. Presently, for technical reasons, it is impossible to give a manual proof should the automatic proof also fail.

5 Conclusion & future work

I have presented an automatic routine to transform definitions using type classes and instances into equivalent definitions using explicit dictionary-passing. Every time the routine is invoked, it uses existing Isabelle facilities to introduce new definitions and prove theorems certifying their equivalence. A possible future extension is a "semi-automatic mode," where users are given the opportunity to perform manual termination and well-definedness proofs.

This work has been carried out to facilitate the compilation toolchain from Isabelle to CakeML [17], but it can also be used stand-alone with the existing code generator. In the future, it could also enable implementing an *OpenTheory* export tool for Isabelle [18]. OpenTheory is an exchange format between different provers in the HOL family. Because Isabelle supports type classes – other HOL provers and OpenTheory do not – only an import tool,[13] but no export tool is available today.

References

[1] Lennart Augustsson. Implementing haskell overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 65–73, New York, NY, USA, 1993. ACM.

[13] https://github.com/xrchz/isabelle-opentheory

[2] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In *Interactive Theorem Proving*, pages 93–110. Springer International Publishing, 2014.

[3] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Operations on bounded natural functors. *Archive of Formal Proofs*, December 2017. http://isa-afp.org/entries/BNF_Operations.html, Formal proof development.

[4] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, sep 2013.

[5] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 170–181, New York, NY, USA, 1992. ACM.

[6] Gordon Andrew D. and Fournet Cédric. Principles and applications of refinement types. *NATO Science for Peace and Security Series - D: Information and Communication Security*, 25(Logics and Languages for Reliability and Security):73–104, 2010.

[7] Amy Felty. Higher-order conditional rewriting in the l-lambda logic programming language. 1992.

[8] Amy Felty. A logic programming approach to implementing higher-order term rewriting. In L. H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming*, pages 135–161. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[9] Simon Finn, Michael P. Fourman, and John Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, Feb 1997.

[10] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data refinement in isabelle/hol. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 100–115, 2013.

[11] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, pages 103–117, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] Florian Haftmann and Makarius Wenzel. Constructive type classes in Isabelle. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs: International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, pages 160–174, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[13] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, mar 1996.

[14] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29, dec 1969.

[15] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, mar 1986.

[16] Lars Hupel. Dictionary construction. *Archive of Formal Proofs*, May 2017. `http://isa-afp.org/entries/Dict_Construction.html`, Formal proof development.

[17] Lars Hupel and Tobias Nipkow. A verified compiler from isabelle/hol to cakeml. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 999–1026, Cham, 2018. Springer International Publishing.

[18] Joe Hurd. The opentheory standard theory library. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 177–191, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[19] Dmitriy Traytel Jan Gilcher, Andreas Lochbihler. Conditional parametricity in Isabelle/HOL. Extended abstract, 2017.

[20] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[21] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. Dissertation, Technische Universität München, München, 2009.

[22] Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *Journal of Automated Reasoning*, 44(4):303–336, 2010.

[23] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 204–215, New York, NY, USA, 2005. ACM.

[24] Andreas Lochbihler. Probabilistic functions and cryptographic oracles in higher order logic. In Peter Thiemann, editor, *Programming Languages and Systems (ESOP 2016)*, volume 9632 of *LNCS*, pages 503–531. Springer, 2016.

[25] Zohar Manna and Amir Pnueli. Formalization of properties of functional programs. *Journal of the ACM*, 17(3):555–569, jul 1970.

[26] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, dec 1978.

[27] John Garrett Morris. *Type Classes and Instance Chains: A Relational Approach.* PhD thesis, Portland State University, 2013.

[28] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2-3):284–315, 005 2014.

[29] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Lecture Notes in Computer Science*, pages 349–366. Springer Berlin Heidelberg, 1998.

[30] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. Springer, 2014.

[31] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[32] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In *Functional Programming Languages and Computer Architecture*, pages 1–14. Springer Berlin Heidelberg, 1991.

[33] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

[34] John Peterson and Mark Jones. Implementing type classes. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 227–236, New York, NY, USA, 1993. ACM.

[35] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[36] Konrad Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, Technische Universität München, 1999.

[37] Christian Sternagel and René Thiemann. Haskell's show class in isabelle/hol. *Archive of Formal Proofs*, July 2014. `http://isa-afp.org/entries/Show.html`, Formal proof development.

[38] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97 Murray Hill, NJ, USA, August 19–22, 1997 Proceedings*, pages 307–322, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[39] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. Verified memoization and dynamic programming. In *Interactive Theorem Proving*, pages 579–596. Springer International Publishing, 2018.