

Dictionary Construction

Lars Hupel

February 23, 2021

Abstract

Isabelle’s code generator natively supports type classes. For targets that do not have language support for classes and instances, it performs the well-known *dictionary translation*, as described by Haftmann and Nipkow [1]. This translation happens outside the logic, i.e., there is no guarantee that it is correct, besides the pen-and-paper proof. This work implements a certified dictionary translation that produces new class-free constants and derives equality theorems.

Contents

1	Dictionary Construction	2
1.1	Introduction	2
1.2	Encoding classes	2
1.3	Encoding instances	3
1.4	Implementation	4
1.5	Impossibility of hiding sort constraints	5
2	Setup	5
3	Termination heuristics	6
4	Test cases for dictionary construction	10
4.1	Code equations with different number of explicit arguments	10
4.2	Complex class hierarchies	10
4.3	Instances with non-trivial arity	10
4.4	[<i>fundef-cong</i>] rules	11
4.5	Mutual recursion	11
4.6	Non-trivial code dependencies; code equations where the head is not fully general	13
4.7	Pattern matching on θ	13

4.8	Complex termination arguments	13
4.9	Combination of various things	14
4.10	Interaction with the code generator	14
4.11	Contrived side conditions	14
4.12	Interaction with <i>Lazy-Case</i>	17

1 Dictionary Construction

```

theory Introduction
imports Main
begin

```

1.1 Introduction

Isabelle’s logic features *type classes* [2, 3]. These are built into the kernel and are used extensively in theory developments. The existing *code generator*, when targeting Standard ML, performs the well-known dictionary construction or *dictionary translation* [1]. 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 [1, §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 certified fashion, that is, the equivalence is a theorem inside the logic.

1.2 Encoding classes

The choice of representation of a dictionary itself is straightforward: We model it as a **datatype**, along with functions returning values of that type. The alternative here would have been to use the **record** package. The obvious advantage is that we could easily model subclass relationships through record inheritance. However, records do not support multiple inheritance. Since records offer no advantage over datatypes in that regard, we opted for the more modern **datatype** package.

Consider the following example:

```

class plus =
  fixes plus :: 'a ⇒ 'a ⇒ 'a

```

This will get translated to a **datatype** with a single constructor taking a single argument:

```

datatype 'a dict-plus =
  mk-plus (param-plus: 'a ⇒ 'a ⇒ 'a)

```

A function using the *Introduction.plus* constraint:

definition *double* :: 'a::plus \Rightarrow 'a **where**
double x = plus x x

definition *double'* :: 'a dict-plus \Rightarrow 'a \Rightarrow 'a **where**
double' dict x = param-plus dict x x

1.3 Encoding instances

A more controversial design decision is how to represent dictionary certificates. For example, given a value of type *nat dict-plus*, how do we know that this is a faithful representation of the *Introduction.plus* instance for *nat*?

- Florian Haftmann 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 their 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. However, any attempt at “hiding” the constraints behind a type definition ultimately does not work: The nonemptiness proof requires a witness of a valid dictionary for an arbitrary, but fixed type 'a, which is of course not possible (see §1.5 for details).

- 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:

$$\text{cert-plus dict} = (\forall a b c. \text{param-plus dict} (\text{param-plus dict } a b) c = \text{param-plus dict } a (\text{param-plus dict } b c))$$

Proving that instances satisfy this certificate is trivial.

However, the equality proof of f' and f is impossible: they are 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 *Introduction.plus-class.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.

- 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 below. 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.

definition $cert-plus :: 'a::plus dict-plus \Rightarrow bool$ **where**
 $cert-plus\ dict \longleftrightarrow (param-plus\ dict = plus)$

Based on that definition, we can prove that $double$ and $double'$ are equivalent:

lemma $cert-plus\ dict \Longrightarrow double'\ dict = double$
<proof>

An unconditional equation can be obtained by specializing the theorem to a ground type and supplying a valid dictionary.

1.4 Implementation

When translating a constant f , we use existing mechanisms in Isabelle to obtain its *code graph*. The graph contains the code equations of all transitive dependencies (i.e., other constants) of f . In general, we have to re-define each of these dependencies. For that, we use the internal interface of the **function** package and feed it the code equations after performing the dictionary construction. 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 imposes some restrictions:

- The new constant needs to be proven terminating. We apply some heuristics to transfer the original termination proof to the new definition. This only works when the termination condition does not rely on class axioms. (See §3 for details.)
- Pattern matching must be performed on datatypes, instead of the more general **code-datatypes**.
- The set of code equations must be exhaustive and non-overlapping.

end

1.5 Impossibility of hiding sort constraints

Coauthor of this section: Florian Haftmann

```
theory Impossibility
imports Main
begin
```

```
axiomatization of-prop :: prop  $\Rightarrow$  bool where
of-prop-Trueprop [simp]: of-prop (Trueprop P)  $\longleftrightarrow$  P and
Trueprop-of-prop [simp]: Trueprop (of-prop Q)  $\equiv$  PROP Q
```

A type satisfies the certificate if there is an instance of the class.

```
definition is-sg :: 'a itself  $\Rightarrow$  bool where
is-sg TYPE('a) = of-prop OFCLASS('a, semigroup-add-class)
```

We trick the parser into ignoring the sort constraint of (+).

<ML>

```
definition sg :: ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  bool where
sg plus  $\longleftrightarrow$  plus = Groups.plus  $\wedge$  is-sg TYPE('a) for plus
```

Attempt: Define a type that contains all legal (+) functions.

```
typedef (overloaded) 'a Sg = Collect sg :: ('a  $\Rightarrow$  'a  $\Rightarrow$  'a) set
morphisms the-plus Sg
<proof>
```

end

2 Setup

```
theory Dict-Construction
imports Automatic-Refinement.Refine-Util
keywords declassify :: thy-decl
begin
```

```
definition set-of :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\times$  'b) set where
set-of P = {(x, y). P x y}
```

```
lemma wfP-implies-wf-set-of: wfP P  $\Longrightarrow$  wf (set-of P)
<proof>
```

```
lemma wf-set-of-implies-wfP: wf (set-of P)  $\Longrightarrow$  wfP P
<proof>
```

```
lemma wf-simulate-simple:
```

```
assumes wf r
assumes  $\bigwedge x y. (x, y) \in r' \Longrightarrow (g\ x, g\ y) \in r$ 
shows wf r'
```

<proof>

lemma *set-ofI*: $P\ x\ y \implies (x, y) \in \text{set-of } P$
<proof>

lemma *set-ofD*: $(x, y) \in \text{set-of } P \implies P\ x\ y$
<proof>

lemma *wfP-simulate-simple*:
 assumes *wfP* *r*
 assumes $\bigwedge x\ y. r'\ x\ y \implies r\ (g\ x)\ (g\ y)$
 shows *wfP* *r'*
<proof>

lemma *wf-implies-dom*: $wf\ (\text{set-of } R) \implies All\ (Wellfounded.\text{accp } R)$
<proof>

lemma *wfP-implies-dom*: $wfP\ R \implies All\ (Wellfounded.\text{accp } R)$
<proof>

named-theorems *dict-construction-specs*

<ML>

declare *[[code drop: (\wedge)]]*
lemma *[code]*: $True \wedge p \longleftrightarrow p\ False \wedge p \longleftrightarrow False$ *<proof>*

declare *[[code drop: (\vee)]]*
lemma *[code]*: $True \vee p \longleftrightarrow True\ False \vee p \longleftrightarrow p$ *<proof>*

declare *comp-cong**[fundef-cong del]*
declare *fun.map-cong**[fundef-cong]*

end

3 Termination heuristics

theory *Termination*
 imports *../Dict-Construction*
begin

As indicated in the introduction, the newly-defined functions must be proven terminating. In general, we cannot reuse the original termination proof, as the following example illustrates:

```
fun f :: nat  $\Rightarrow$  nat where  
f 0 = 0 |  
f (Suc n) = f n
```

lemma [code]: $f\ x = f\ x$ *<proof>*

The invocation of **declassify** f would fail, because f 's code equations are not terminating.

Hence, in the general case where users have modified the code equations, we need to fall back to an (automated) attempt to prove termination.

In the remainder of this section, we will illustrate the special case where the user has not modified the code equations, i.e., the original termination proof should “morally” be still applicable. For this, we will perform the dictionary construction manually.

— Some ML incantations to ensure that the dictionary types are present
<ML>

```
fun sum-list :: 'a::{plus,zero} list  $\Rightarrow$  'a where  
sum-list [] = 0 |  
sum-list (x # xs) = x + sum-list xs
```

The above function carries two distinct class constraints, which are translated into two dictionary parameters:

```
function sum-list' where  
sum-list' d-plus d-zero [] = Groups-zero--class-zero--field d-zero |  
sum-list' d-plus d-zero (x # xs) = Groups-plus--class-plus--field d-plus x (sum-list'  
d-plus d-zero xs)  
<proof>
```

Now, we need to carry out the termination proof of $sum\text{-list}'$. The **function** package analyzes the function definition and discovers one recursive call. In pseudo-notation:

$$(d\text{-plus}, d\text{-zero}, x \# xs) \rightsquigarrow (d\text{-plus}, d\text{-zero}, xs)$$

The result of this analysis is captured in the inductive predicate $sum\text{-list}'\text{-rel}$. Its introduction rules look as follows:

```
thm sum-list'-rel.intros  
— sum-list'-rel (?d-plus, ?d-zero, ?xs) (?d-plus, ?d-zero, ?x # ?xs)
```

Compare this to the relation for $Termination.sum\text{-list}$:

```
thm sum-list-rel.intros  
— sum-list-rel ?xs (?x # ?xs)
```

Except for the additional (unchanging) dictionary arguments, these relations are more or less equivalent to each other. There is an important difference, though: $sum\text{-list}\text{-rel}$ has sort constraints, $sum\text{-list}'\text{-rel}$ does not. (This will become important later on.)

context

notes $[[\textit{show-sorts}]]$
begin

term *sum-list-rel*
 $\text{--- } 'a::\{\textit{plus,zero}\} \textit{list} \Rightarrow 'a::\{\textit{plus,zero}\} \textit{list} \Rightarrow \textit{bool}$

term *sum-list'-rel*
 $\text{--- } 'a::\textit{type Groups-plus--dict} \times 'a::\textit{type Groups-zero--dict} \times 'a::\textit{type list} \Rightarrow 'a::\textit{type Groups-plus--dict} \times 'a::\textit{type Groups-zero--dict} \times 'a::\textit{type list} \Rightarrow \textit{bool}$

end

Let us now discuss the rough concept of the termination proof for *sum-list'*. The goal is to show that *sum-list'-rel* is well-founded. Usually, this is proved by specifying a *measure function* that

1. maps the arguments to natural numbers
2. decreases for each recursive call.

Here, however, we want to instead show that each recursive call in *sum-list'* has a corresponding recursive call in *Termination.sum-list*. In other words, we want to show that the existing proof of well-foundedness of *sum-list-rel* can be lifted to a proof of well-foundedness of *sum-list'-rel*. This is what the theorem *wfP-simulate-simple* states:

$$[[\textit{wfP } ?r; \bigwedge x y. ?r' x y \implies ?r (?g x) (?g y)]] \implies \textit{wfP } ?r'$$

Given any well-founded relation r and a function g that maps function arguments from r' to r , we can deduce that r' is also well-founded.

For our example, we need to provide a function g of type $'b \textit{Groups-plus--dict} \times 'b \textit{Groups-zero--dict} \times 'b \textit{list} \Rightarrow 'a \textit{list}$. Because the dictionary parameters are not changing, they can safely be dropped by g . However, because of the sort constraint in *sum-list-rel*, the term $\textit{snd} \circ \textit{snd}$ is not a well-typed instantiation for g .

Instead (this is where the heuristic comes in), we assume that the original function *Termination.sum-list* is parametric, i.e., termination does not depend on the elements of the list passed to it, but only on the structure of the list. Additionally, we assume that all involved type classes have at least one instantiation.

With this in mind, we can use $\textit{map} (\lambda-. \textit{undefined}) \circ \textit{snd} \circ \textit{snd}$ as g :

thm *wfP-simulate-simple***[where**
 $r = \textit{sum-list-rel}$ **and**
 $r' = \textit{sum-list'-rel}$ **and**
 $g = \textit{map} (\lambda-. \textit{undefined}) \circ \textit{snd} \circ \textit{snd}$]

Finally, we can prove the termination of *sum-list'*.

```
termination sum-list'  
⟨proof⟩
```

This can be automated with a special tactic:

```
experiment  
begin  
  
termination sum-list'  
  ⟨proof⟩  
  
end
```

A similar technique can be used for making functions defined in locales executable when, for some reason, the definition of a “defs” locale is not feasible.

```
locale foo =  
  fixes A :: nat  
  assumes  $A > 0$   
begin
```

```
fun f where  
f 0 = A |  
f (Suc n) = Suc (f n)
```

— We carry out this proof in the locale for simplicity; a real implementation would probably have to set up a local theory properly.

```
lemma f-total: wfP f-rel  
⟨proof⟩
```

```
end
```

— The dummy interpretation serves the same purpose as the assumption that class constraints have at least one instantiation.

```
interpretation dummy: foo 1 ⟨proof⟩
```

```
function f' where  
f' A 0 = A |  
f' A (Suc n) = Suc (f' A n)  
⟨proof⟩
```

```
termination f'  
  ⟨proof⟩
```

Automatic:

```
experiment  
begin
```

```
termination f'
  ⟨proof⟩
```

```
end
```

```
end
```

4 Test cases for dictionary construction

```
theory Test-Dict-Construction
```

```
imports
```

```
  Dict-Construction
```

```
  HOL-Library.ListVector
```

```
begin
```

4.1 Code equations with different number of explicit arguments

```
lemma [code]: fold f [] = id fold f (x # xs) s = fold f xs (f x s) fold f [x, y] u ≡ f
y (f x u)
⟨proof⟩
```

```
experiment begin
```

```
  declassify valid: fold
```

```
  thm valid
```

```
  lemma List-fold = fold ⟨proof⟩
```

```
end
```

4.2 Complex class hierarchies

```
⟨ML⟩
```

```
experiment begin
```

```
  ⟨ML⟩
```

```
  typ nat Rings-ring--dict
```

```
end
```

Check that *Class-Graph* does not leak out of locales

```
⟨ML⟩
```

4.3 Instances with non-trivial arity

```
fun f :: 'a::plus ⇒ 'a where
f x = x + x
```

```

definition  $g :: 'a::\{plus,zero\} list \Rightarrow 'a list$  where
 $g\ x = f\ x$ 

datatype  $natt = Z \mid S\ natt$ 

instantiation  $natt :: \{zero,plus\}$  begin
  definition  $zero-natt$  where
     $zero-natt = Z$ 

  fun  $plus-natt$  where
     $plus-natt\ Z\ x = x \mid$ 
     $plus-natt\ (S\ m)\ n = S\ (plus-natt\ m\ n)$ 

  instance  $\langle proof \rangle$ 
end

definition  $h :: natt\ list$  where
 $h = g\ [Z,S\ Z]$ 

experiment begin

declassify  $valid: h$ 
thm  $valid$ 
lemma  $Test--Dict--Construction-h = h$   $\langle proof \rangle$ 

 $\langle ML \rangle$ 

end

Check that declassify does not leak out of locales
 $\langle ML \rangle$ 

4.4 [fundef-cong] rules
datatype  $'a\ seq = Cons\ 'a\ 'a\ seq \mid Nil$ 

experiment begin

declassify  $map-seq$ 

Check presence of derived [fundef-cong] rule
 $\langle ML \rangle$ 

end

4.5 Mutual recursion
fun  $odd :: nat \Rightarrow bool$  and  $even$  where

```

```

odd 0  $\longleftrightarrow$  False |
even 0  $\longleftrightarrow$  True |
odd (Suc n)  $\longleftrightarrow$  even n |
even (Suc n)  $\longleftrightarrow$  odd n

```

experiment begin

```

declassify valid: odd even
thm valid

```

end

```

datatype 'a bin-tree = Leaf | Node 'a 'a bin-tree 'a bin-tree

```

experiment begin

```

declassify valid: map-bin-tree rel-bin-tree
thm valid

```

end

```

datatype 'v env = Env 'v list
datatype v = Closure v env

```

context

```

notes is-measure-trivial[where f = size-env size, measure-function]
begin

```

```

fun test-v :: v  $\Rightarrow$  bool and test-w :: v env  $\Rightarrow$  bool where
test-v (Closure env)  $\longleftrightarrow$  test-w env |
test-w (Env vs)  $\longleftrightarrow$  list-all test-v vs

```

```

fun test-v1 :: v  $\Rightarrow$  'a::{one,monoid-add} and test-w1 :: v env  $\Rightarrow$  'a where
test-v1 (Closure env) = 1 + test-w1 env |
test-w1 (Env vs) = sum-list (map test-v1 vs)

```

end

experiment begin

```

declassify valid: test-w test-v
thm valid

```

end

experiment begin

```
declassify valid: test-w1 test-v1
thm valid
```

end

4.6 Non-trivial code dependencies; code equations where the head is not fully general

```
definition c ≡ 0 :: nat
```

```
definition d x ≡ if x = 0 then 0 else x
```

```
lemma contrived[code]: c = d 0 ⟨proof⟩
```

experiment begin

```
declassify valid: c
```

```
thm valid
```

```
lemma Test--Dict--Construction-c = c ⟨proof⟩
```

end

4.7 Pattern matching on 0

```
definition j where j (n::nat) = (0::nat)
```

```
lemma [code]: j 0 = 0 j (Suc n) = j n
⟨proof⟩
```

```
fun k where
```

```
k 0 = (0::nat) |
```

```
k (Suc n) = k n
```

```
lemma f-code[code]: k n = 0
⟨proof⟩
```

experiment begin

```
declassify valid: j k
```

```
thm valid
```

```
lemma
```

```
Test--Dict--Construction-j = j
```

```
Test--Dict--Construction-k = k
```

```
⟨proof⟩
```

end

4.8 Complex termination arguments

```
fun fac :: nat ⇒ nat where
```

```
fac n = (if n ≤ 1 then 1 else n * fac (n - 1))
```

```

experiment begin

declassify valid: fac

end

```

4.9 Combination of various things

```

experiment begin

declassify valid: sum-list

end

```

4.10 Interaction with the code generator

```

declassify h
export-code Test--Dict--Construction-h in SML

end

```

4.11 Contrived side conditions

```

theory Test-Side-Conditions
imports Dict-Construction
begin

  <ML>

  fun head where
    head (x # -) = x

  <ML>

  lemma head-side-eq: head-side xs  $\longleftrightarrow$  xs  $\neq$  []
  <proof>

  <ML>

  fun map where
    map f [] = [] |
    map f (x # xs) = f x # map f xs

  <ML>
  thm map-side.intros

  <ML>

```

experiment begin

Functions that use partial functions always in their domain are processed correctly.

fun *tail* **where**
tail (- # *xs*) = *xs*

<ML>

lemma *tail-side-eq*: *tail-side xs* \longleftrightarrow *xs* \neq []
<proof>

<ML>

function *map'* **where**
map' f xs = (if *xs* = [] then [] else *f* (head *xs*) # *map' f* (tail *xs*))
<proof>

termination
<proof>

<ML>

thm *map'-side.intros*

<ML>

end

lemma *map-cong*:
assumes *xs* = *ys* \wedge $\bigwedge x. x \in \text{set } ys \implies f x = g x$
shows *map f xs* = *map g ys*
<proof>

definition *map-head* **where**
map-head xs = *map head xs*

experiment begin

declare *map-cong*[*fundef-cong*]

<ML>

thm *map-head-side.intros*

lemma *map-head-side* *xs* \longleftrightarrow ($\forall x \in \text{set } xs. x \neq []$)
<proof>

definition *map-head'* **where**
map-head' xss = *map (map head) xss*

```

  <ML>
  thm map-head'-side.intros

  lemma map-head'-side xss  $\longleftrightarrow$  ( $\forall xs \in set\ xss. \forall x \in set\ xs. x \neq []$ )
  <proof>

end

experiment begin

  <ML>
  term map-head-side
  thm map-head-side.intros

  lemma  $\neg$  map-head-side xs
  <proof>

end

definition head-known where
  head-known xs = head (3 # xs)

  <ML>
  thm head-known-side.intros

  <ML>

  fun odd :: nat  $\Rightarrow$  bool and even where
    odd 0  $\longleftrightarrow$  False |
    even 0  $\longleftrightarrow$  True |
    odd (Suc n)  $\longleftrightarrow$  even n |
    even (Suc n)  $\longleftrightarrow$  odd n

  <ML>
  thm odd-side-even-side.intros

  <ML>

  definition odd-known where
    odd-known = odd (Suc 0)

  <ML>
  thm odd-known-side.intros

  <ML>

end

```


4.12 Interaction with *Lazy-Case*

theory *Test-Lazy-Case*

imports

Dict-Construction

Lazy-Case.Lazy-Case

Show.Show-Instances

begin

datatype 'a tree = Node | Fork 'a 'a tree list

lemma *map-tree*[code]:

map-tree f t = (case t of Node \Rightarrow Node | Fork x ts \Rightarrow Fork (f x) (map (map-tree

f) ts))

<proof>

experiment begin

Dictionary construction of *map-tree* requires the [fundef-cong] rule of *Test-Lazy-Case.tree.case-lazy*.

declassify *valid: map-tree*

thm *valid*

lemma *Test--Lazy--Case-tree-map--tree = map-tree* *<proof>*

end

definition *i* :: (unit \times (bool list \times string \times nat option) list) option \Rightarrow string

where

i = show

experiment begin

This currently requires *Lazy-Case.Lazy-Case* because of *divmod-nat*.

declassify *valid: i*

thm *valid*

lemma *Test--Lazy--Case-i = i* *<proof>*

end

end

References

- [1] 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.

- [2] 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.
- [3] 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.