Dictionary Construction

Lars Hupel

March 17, 2025

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

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* :: $'a \Rightarrow 'a \Rightarrow 'a$

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

datatype $'a$ *dict-plus* =
  mk-plus (*param-plus*: $'a \Rightarrow 'a \Rightarrow 'a$)

A function using the *Introduction.plus* constraint:

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

definition $double' :: {'a}\ dict\text{-}plus \Rightarrow {'a} \Rightarrow {'a}$ **where**
$double'\ dict\ x = param\text{-}plus\ dict\ x\ x$

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:

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

  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*
unfolding *cert-plus-def double'-def double-def*
by *auto*

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-datatype**s.
- 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* ⇒ *bool* **where**
of-prop-Trueprop [*simp*]: *of-prop* (*Trueprop P*) ⟷ *P* **and**
Trueprop-of-prop [*simp*]: *Trueprop* (*of-prop Q*) ≡ *PROP Q*

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

definition *is-sg* :: *'a itself* ⇒ *bool* **where**
is-sg TYPE('a) = *of-prop OFCLASS('a, semigroup-add-class)*

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

setup ‹*Sign.add-const-constraint* (@{*const-name plus*}, *SOME* @{*typ 'a*::{} => *'a* ⇒ *'a*}})›

definition *sg* :: (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *bool* **where**
sg plus ⟷ *plus* = *Groups.plus* ∧ *is-sg TYPE('a)* **for** *plus*

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

typedef (**overloaded**) *'a Sg* = *Collect sg* :: (*'a* ⇒ *'a* ⇒ *'a*) *set*
  morphisms *the-plus Sg*
  unfolding *sg-def*[*abs-def*]
  apply (*simp add*: *is-sg-def*)

We need to prove *OFCLASS('a, semigroup-add-class)* for arbitrary *'a*, which is impossible.

oops

end

2 Setup

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

definition *set-of* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* × *'b*) *set* **where**
set-of P = {(*x*, *y*). *P x y*}

lemma *wfP-implies-wf-set-of*: *wfP P* ⟹ *wf* (*set-of P*)
unfolding *wfp-def set-of-def* **.**

lemma *wf-set-of-implies-wfP*: $wf\ (set\text{-}of\ P) \Longrightarrow wfP\ P$
unfolding *wfp-def set-of-def* **.**

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'*
using *assms*
by (*metis in-inv-image wf-eq-minimal wf-inv-image*)

lemma *set-ofI*: $P\ x\ y \Longrightarrow (x,\ y) \in set\text{-}of\ P$
unfolding *set-of-def* **by** *simp*

lemma *set-ofD*: $(x,\ y) \in set\text{-}of\ P \Longrightarrow P\ x\ y$
unfolding *set-of-def* **by** *simp*

lemma *wfP-simulate-simple*:
  assumes *wfP r*
  assumes $\bigwedge x\ y.\ r'\ x\ y \Longrightarrow r\ (g\ x)\ (g\ y)$
  shows *wfP r'*
apply (*rule wf-set-of-implies-wfP*)
apply (*rule wf-simulate-simple*[**where** $g = g$])
apply (*rule wfP-implies-wf-set-of*)
apply (*fact assms*)
using *assms*(*2*) **by** (*auto intro*: *set-ofI dest*: *set-ofD*)

lemma *wf-implies-dom*: $wf\ (set\text{-}of\ R) \Longrightarrow All\ (Wellfounded.accp\ R)$
apply (*rule allI*)
apply (*rule accp-wfpD*)
apply (*rule wf-set-of-implies-wfP*) **.**

lemma *wfP-implies-dom*: $wfP\ R \Longrightarrow All\ (Wellfounded.accp\ R)$
by (*metis wfP-implies-wf-set-of wf-implies-dom*)

named-theorems *dict-construction-specs*

ML-file ‹*dict-construction-util.ML*›
ML-file ‹*transfer-termination.ML*›
ML-file ‹*congruences.ML*›
ML-file ‹*side-conditions.ML*›
ML-file ‹*class-graph.ML*›
ML-file ‹*dict-construction.ML*›

method-setup *fo-cong-rule* = ‹
  Attrib.thm >> (fn thm => fn ctxt => SIMPLE-METHOD′ (Dict-Construction-Util.fo-cong-tac ctxt thm))
› *resolve congruence rule using first−order matching*

declare [[*code drop*: (∧)]]

lemma [*code*]: *True* $\wedge$ *p* $\longleftrightarrow$ *p* *False* $\wedge$ *p* $\longleftrightarrow$ *False* **by** *auto*

declare [[*code drop*: ($\vee$)]]
lemma [*code*]: *True* $\vee$ *p* $\longleftrightarrow$ *True* *False* $\vee$ *p* $\longleftrightarrow$ *p* **by** *auto*

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* **..**

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.

local-setup ‹*Class-Graph.ensure-class* @{*class plus*} #> *snd*›
local-setup ‹*Class-Graph.ensure-class* @{*class zero*} #> *snd*›

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*)

by *pat-completeness auto*

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

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

The result of this analysis is captured in the inductive predicate *sum-list′-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-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-list-rel* has sort constraints, *sum-list′-rel* does not. (This will become important later on.)

context
notes [[*show-sorts*]]
begin

term *sum-list-rel*
— ′*a*::{*plus*,*zero*} *list* ⇒ ′*a*::{*plus*,*zero*} *list* ⇒ *bool*

term *sum-list′-rel*
— ′*a*::*type Groups-plus--dict* × ′*a*::*type Groups-zero--dict* × ′*a*::*type list* ⇒ ′*a*::*type Groups-plus--dict* × ′*a*::*type Groups-zero--dict* × ′*a*::*type list* ⇒ *bool*

end

Let us know 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:

$[\![\mathit{wfp}\ ?r;\ \bigwedge x\ y.\ ?r'\ x\ y \Longrightarrow ?r\ (?g\ x)\ (?g\ y)]\!] \Longrightarrow \mathit{wfp}\ ?r'$

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 Groups-plus--dict* × *'b Groups-zero--dict* × *'b list* ⇒ *'a 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 *snd* ∘ *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 *map* (λ-. *undefined*) ∘ *snd* ∘ *snd* as *g*:

thm *wfP-simulate-simple*[**where**
  r = *sum-list-rel* **and**
  r' = *sum-list'-rel* **and**
  g = *map* (λ-. *undefined*) ∘ *snd* ∘ *snd*]

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

termination *sum-list'*
proof −
  have *wfP sum-list'-rel*
  proof (*rule wfP-simulate-simple*)
    — We first need to obtain the well-foundedness theorem for *sum-list-rel* from the ML guts of the **function** package.
    show *wfP sum-list-rel*
      apply (*rule accp-wfpI*)
      apply (*tactic* ‹*resolve-tac* @{*context*} [*Function.get-info* @{*context*} @{*term sum-list*} |> #*totality* |> *the*] *1*›)
      done

   define *g* :: *'b Groups-plus--dict* × *'b Groups-zero--dict* × *'b list* ⇒ *'c*::{*plus*,*zero*} *list* **where**
      g = *map* (λ-. *undefined*) ∘ *snd* ∘ *snd*

    — Prove the simulation of *sum-list'-rel* by *sum-list-rel* by rule induction.
    show *sum-list-rel* (*g x*) (*g y*) **if** *sum-list'-rel x y* **for** *x y*
      using *that*
      proof (*induction x y rule*: *sum-list'-rel.induct*)
        case (*1 d-plus d-zero x xs*)
        show *?case*
          — Unfold the constituent parts of *g*:
          apply (*simp only*: *g-def comp-apply snd-conv list.map*)
          — Use the corresponding introduction rule of *sum-list-rel* and hope for the best:

```
      apply (rule sum-list-rel.intros(1))
      done
    qed
  qed

  — This is the goal that the function package expects.
  then show ∀ x. sum-list'-dom x
    by (rule wfP-implies-dom)
qed
```

This can be automated with a special tactic:

```
experiment
begin

termination sum-list'
  apply (tactic ‹
    Transfer-Termination.termination-tac
      (Function.get-info @{context} @{term sum-list'})
      (Function.get-info @{context} @{term sum-list})
      @{context}
      1›; fail)
  done

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
apply (rule accp-wfpI)
apply (tactic ‹resolve-tac @{context} [Function.get-info @{context} @{term f} |>
#totality |> the] 1›)
done

end
```

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

interpretation *dummy*: *foo 1* **by** *standard simp*

function f' **where**
f' *A 0* = *A* |
f' *A* (*Suc n*) = *Suc* (f' *A n*)
by *pat-completeness auto*

termination f'
  apply (*rule wfP-implies-dom*)
  apply (*rule wfP-simulate-simple*[**where** *g* = *snd*])
   apply (*rule dummy.f-total*)
  subgoal for *x y*
    apply (*induction x y rule*: f'*-rel.induct*)
    subgoal
     apply (*simp only*: *snd-conv*)
     apply (*rule dummy.f-rel.intros*)
     done
    done
  done

Automatic:

experiment
begin

termination f'
  apply (*tactic* ‹
    Transfer-Termination.termination-tac
      (*Function.get-info* @{*context*} @{*term* f'})
      (*Function.get-info* @{*context*} @{*term dummy.f*})
      @{*context*}
      1›; *fail*)
  done

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*)
by *auto*

experiment begin

declassify *valid*: *fold*
thm *valid*
lemma *List-fold* = *fold* **by** (*rule valid*)

end

4.2 Complex class hierarchies

local-setup ‹*Class-Graph.ensure-class* @{*class zero*} #> *snd*›
local-setup ‹*Class-Graph.ensure-class* @{*class plus*} #> *snd*›

experiment begin

local-setup ‹*Class-Graph.ensure-class* @{*class comm-monoid-add*} #> *snd*›
local-setup ‹*Class-Graph.ensure-class* @{*class ring*} #> *snd*›

typ *nat Rings-ring--dict*

end

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

ML‹@{*assert*} (*is-none* (*Class-Graph.node* @{*context*} @{*class ring*}))›

4.3 Instances with non-trivial arity

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

definition *g* :: '*a*::{*plus*,*zero*} *list* ⇒ '*a list* **where**
g x = *f x*

datatype *natt* = *Z* | *S natt*

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

fun *plus-natt* **where**
plus-natt Z x = *x* |
plus-natt (*S m*) *n* = *S* (*plus-natt m n*)

```
  instance ..
end

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

experiment begin


declassify valid: h
thm valid
lemma Test--Dict--Construction-h = h by (fact valid)

ML‹Dict-Construction.the-info @{context} @{const-name plus-natt-inst.plus-natt}›

end
```

Check that **declassify** does not leak out of locales

```
ML‹
  can (Dict-Construction.the-info @{context}) @{const-name plus-natt-inst.plus-natt}
  |> not |> @{assert}
›
```

4.4 [*fundef-cong*] rules

```
datatype 'a seq = Cons 'a 'a seq | Nil

experiment begin

declassify map-seq
```

Check presence of derived [*fundef-cong*] rule

```
ML‹
  Dict-Construction.the-info @{context} @{const-name map-seq}
  |> #fun-info
  |> the
  |> #fs
  |> the-single
  |> dest-Const
  |> fst
  |> Dict-Construction.cong-of-const @{context}
  |> the
›

end
```

4.5 Mutual recursion

```
fun odd :: nat ⇒ bool and even where
```

odd 0 ⟷ *False* |
even 0 ⟷ *True* |
odd (*Suc n*) ⟷ *even n* |
even (*Suc n*) ⟷ *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* ⇒ *bool* **and** *test-w* :: *v env* ⇒ *bool* **where**
test-v (*Closure env*) ⟷ *test-w env* |
test-w (*Env vs*) ⟷ *list-all test-v vs*

fun *test-v1* :: *v* ⇒ *'a*::{*one*,*monoid-add*} **and** *test-w1* :: *v env* ⇒ *'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 $c \equiv 0 :: nat$
definition $d\ x \equiv if\ x = 0\ then\ 0\ else\ x$

lemma *contrived*[*code*]: $c = d\ 0$ **unfolding** *c-def d-def* **by** *simp*

experiment begin

declassify *valid*: *c*
thm *valid*
lemma *Test--Dict--Construction-c* = *c* **by** (*fact valid*)

end

4.7 Pattern matching on *0*

definition *j* **where** $j\ (n::nat) = (0::nat)$

lemma [*code*]: $j\ 0 = 0\ j\ (Suc\ n) = j\ n$
unfolding *j-def* **by** *auto*

fun *k* **where**
$k\ 0 = (0::nat)$ |
$k\ (Suc\ n) = k\ n$

lemma *f-code*[*code*]: $k\ n = 0$
by (*induct n*) *simp*+

experiment begin

declassify *valid*: *j k*
thm *valid*
lemma
  Test--Dict--Construction-j = *j*
  Test--Dict--Construction-k = *k*
by (*fact valid*)+

end

4.8 Complex termination arguments

fun *fac* :: $nat \Rightarrow nat$ **where**
$fac\ n = (if\ n \leq 1\ then\ 1\ else\ n * fac\ (n - 1))$

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 assert-alt-total ctxt term = @{*assert*} (*Side-Conditions.is-total ctxt term*)
›

fun *head* **where**
head (*x* # -) = *x*

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms head.simps*} *NONE*›

lemma *head-side-eq*: *head-side xs* ⟷ *xs* ≠ []
by (*cases xs*) (*auto intro*: *head-side.intros elim*: *head-side.cases*)

declaration ‹*K* (*Side-Conditions.set-alt* @{*term head*} @{*thm head-side-eq*})›

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

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms map.simps*} (*SOME* @{*thms map.induct*})›
thm *map-side.intros*

ML ‹*assert-alt-total* @{*context*} @{*term map*}›

experiment begin

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

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

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms tail.simps*} *NONE*›

lemma *tail-side-eq*: *tail-side xs* $\longleftrightarrow$ *xs* $\neq$ []
by (*cases xs*) (*auto intro*: *tail-side.intros elim*: *tail-side.cases*)

declaration ‹*K* (*Side-Conditions.set-alt* @{*term tail*} @{*thm tail-side-eq*})›

function *map′* **where**
map′ f xs = (*if xs* = [] *then* [] *else f* (*head xs*) # *map′ f* (*tail xs*))
by *auto*

termination
apply (*relation measure* (*size* $\circ$ *snd*))
apply *rule*
subgoal for *f xs* **by** (*cases xs*) *auto*
done

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms map′.simps*} (*SOME* @{*thms map′.induct*})›
thm *map′-side.intros*

ML ‹*assert-alt-total* @{*context*} @{*term map′*}›

end

lemma *map-cong*:
assumes *xs* = *ys* $\bigwedge x.\ x \in$ *set ys* $\Longrightarrow$ *f x* = *g x*
shows *map f xs* = *map g ys*
unfolding *assms*(*1*)
using *assms*(*2*)
by (*induction ys*) *auto*

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

experiment begin

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

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms map-head-def*} *NONE*›
thm *map-head-side.intros*

lemma *map-head-side xs* $\longleftrightarrow$ ($\forall$ *x* $\in$ *set xs*. *x* $\neq$ [])
by (*auto intro*: *map-head-side.intros elim*: *map-head-side.cases*)

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

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms map-head'-def*} *NONE*›
thm *map-head'-side.intros*

lemma *map-head'-side xss* $\longleftrightarrow$ ($\forall$ *xs* $\in$ *set xss*. $\forall$ *x* $\in$ *set xs*. *x* $\neq$ [])
by (*auto intro*: *map-head'-side.intros elim*: *map-head'-side.cases*)

end

experiment begin

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms map-head-def*} *NONE*›
term *map-head-side*
thm *map-head-side.intros*

lemma $\neg$ *map-head-side xs*
by (*auto elim*: *map-head-side.cases*)

end

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

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms head-known-def*} *NONE*›
thm *head-known-side.intros*

ML‹*assert-alt-total* @{*context*} @{*term head-known*}›

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*

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms odd.simps even.simps*} (*SOME* @{*thms odd-even.induct*})›
thm *odd-side-even-side.intros*

ML‹*assert-alt-total* @{*context*} @{*term odd*}›
ML‹*assert-alt-total* @{*context*} @{*term even*}›

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

local-setup ‹*snd o Side-Conditions.mk-side* @{*thms odd-known-def*} *NONE*›
thm *odd-known-side.intros*

ML‹*assert-alt-total* @{*context*} @{*term odd-known*}›

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* ⇒ *Node* | *Fork x ts* ⇒ *Fork* (*f x*) (*map* (*map-tree f*) *ts*))
by (*induction t*) *auto*

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* **by** (*fact valid*)

end

definition *i* :: (*unit* × (*bool list* × *string* × *nat option*) *list*) *option* ⇒ *string*
where
i = *show*

experiment begin

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

declassify *valid*: *i*
thm *valid*

lemma *Test--Lazy--Case-i* = *i* **by** (*fact valid*)

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.