A Meta-Model for the Isabelle API

Frédéric Tuong Burkhart Wolff

March 17, 2025

LRI, Univ. Paris-Sud, CNRS, CentraleSupélec, Université Paris-Saclay
bât. 650 Ada Lovelace, 91405 Orsay, France
frederic.tuong@lri.fr burkhart.wolff@lri.fr

IRT SystemX
8 av. de la Vauve, 91120 Palaiseau, France
frederic.tuong@irt-systemx.fr burkhart.wolff@irt-systemx.fr

Abstract

We represent a theory *of* (a fragment of) Isabelle/HOL *in* Isabelle/HOL. The purpose of this exercise is to write packages for domain-specific specifications such as class models, B-machines, ..., and generally speaking, any domain-specific languages whose abstract syntax can be defined by a HOL "datatype". On this basis, the Isabelle code-generator can then be used to generate code for global context transformations as well as tactic code.

Consequently the package is geared towards parsing, printing and code-generation to the Isabelle API. It is at the moment not sufficiently rich for doing meta theory on Isabelle itself. Extensions in this direction are possible though.

Moreover, the chosen fragment is fairly rudimentary. However it should be easily adapted to one's needs if a package is written on top of it. The supported API contains types, terms, transformation of global context like definitions and data-type declarations as well as infrastructure for Isar-setups.

This theory is drawn from the Featherweight OCL[1] project where it is used to construct a package for object-oriented data-type theories generated from UML class diagrams. The Featherweight OCL, for example, allows for both the direct execution of compiled tactic code by the Isabelle API as well as the generation of `.thy`-files for debugging purposes.

Gained experience from this project shows that the compiled code is sufficiently efficient for practical purposes while being based on a formal *model* on which properties of the package can be proven such as termination of certain transformations, correctness, etc.

Contents

Part I.

A Meta-Model for the Isabelle API

1. Initialization

theory *Init*
  imports *isabelle-home/src/HOL/Isabelle-Main0*
begin

1.1. Optimization on the String Datatype

The following types will allow to delay all concatenations on *integer list*, until we reach the end. As optimization, we also consider the use of *String.literal* besides *integer list*.

type-notation *natural* (‹*nat*›)
definition *Succ x* = *x* + *1*

datatype $string_{base}$ = *ST String.literal*
  | *ST'* *integer list*

datatype *abr-string* =
    SS-base $string_{base}$
  | *String-concatWith abr-string abr-string list*

syntax *-string1* :: - ⇒ *abr-string* (‹⟨(-)⟩›)
translations ⟨*x*⟩ ⇌ *CONST SS-base* (*CONST ST x*)

syntax *-string3* :: - ⇒ *abr-string* (‹≪(-)≫›)
translations ≪*x*≫ ⇌ *CONST SS-base* (*CONST ST' x*)

syntax *-integer1* :: - ⇒ *abr-string* (‹°(-)°›)
translations °*x*° ⇌ *CONST SS-base* (*CONST ST'* ((*CONST Cons*) *x* (*CONST Nil*)))

type-notation *abr-string* (‹*string*›)

1.2. Basic Extension of the Standard Library

1.2.1. Polymorphic Cartouches

We generalize the construction of cartouches for them to be used “polymorphically”, however the type inference is not automatic: types of all cartouche expressions will need to be specified earlier before their use (we will however provide a default type).

ML‹
```
structure Cartouche-Grammar = struct
  fun list-comb-mk cst n c = list-comb (Syntax.const cst, String-Syntax.mk-bits-syntax n c)
  val nil1 = Syntax.const @{const-syntax String.empty-literal}
  fun cons1 c l = list-comb-mk @{const-syntax String.Literal} 7 c $ l

  val default =
    [ ( char list
      , ( Const (@{const-syntax Nil}, @{typ char list})
        , fn c => fn l => Syntax.const @{const-syntax Cons} $ list-comb-mk @{const-syntax
Char} 8 c $ l
        , snd))
    , ( String.literal, (nil1 , cons1 , snd))
    , ( abr-string
      , ( nil1
        , cons1
        , fn (-, x) => Syntax.const @{const-syntax SS-base}
                       $ (Syntax.const @{const-syntax ST}
                          $ x)))]
end
```
›

ML‹
```
fun parse-translation-cartouche binding l f-integer accu =
  let val cartouche-type = Attrib.setup-config-string binding (K (fst (hd l)))
      (* if there is no type specified, by default we set the first element
         to be the default type of cartouches *) in
  fn ctxt =>
    let val cart-type = Config.get ctxt cartouche-type in
    case List.find (fn (s, -) => s = cart-type) l of
      NONE => error (Unregistered return type for the cartouche: \ ^ cart-type ^ \)
    | SOME (-, (nil0, cons, f)) =>
      string-tr f (f-integer, cons, nil0) accu (Symbol-Pos.cartouche-content o Symbol-Pos.explode)
    end
  end
```
›

parse-translation ‹
```
  [( @{syntax-const -cartouche-string}
   , parse-translation-cartouche @{binding cartouche-type} Cartouche-Grammar.default (K I)
())]
```
›

This is the special command which sets the type of subsequent cartouches. Note: here the given type is currently parsed as a string, one should extend it to be a truly "typed" type...

declare[[*cartouche-type* = *abr-string*]]

1.2.2. Operations on List

datatype ('a, 'b) *nsplit* = *Nsplit-text* 'a
| *Nsplit-sep* 'b
locale *L*
begin
definition *map* **where** *map f l* = *rev* (*foldl* (λ*l x*. *f x* # *l*) [] *l*)
definition *flatten l* = *foldl* (λ*acc l*. *foldl* (λ*acc x*. *x* # *acc*) *acc* (*rev l*)) [] (*rev l*)
definition *mapi f l* = *rev* (*fst* (*foldl* (λ(*l*,*cpt*) *x*. (*f cpt x* # *l*, *Succ cpt*)) ([], *0*::*nat*) *l*))
definition *iter f* = *foldl* (λ-. *f*) ()
definition *maps f x* = *L.flatten* (*L.map f x*)
definition *append* **where** *append a b* = *L.flatten* [*a*, *b*]
definition *filter* **where** *filter f l* = *rev* (*foldl* (λ*l x*. *if f x then x* # *l else l*) [] *l*)
definition *rev-map f* = *foldl* (λ*l x*. *f x* # *l*) []
definition *mapM f l accu* =
(*let* (*l*, *accu*) = *List.fold* (λ*x* (*l*, *accu*). *let* (*x*, *accu*) = *f x accu in* (*x* # *l*, *accu*)) *l* ([], *accu*) *in*
(*rev l*, *accu*))
definition *assoc x1 l* = *List.fold* (λ(*x2*, *v*). λ*None* ⇒ *if x1* = *x2 then Some v else None* | *x* ⇒ *x*) *l None*
definition *split* **where** *split l* = (*L.map fst l*, *L.map snd l*)
definition *upto* **where** *upto i j* =
(*let to-i* = λ*n*. *int-of-integer* (*integer-of-natural n*) *in*
L.map (*natural-of-integer o integer-of-int*) (*List.upto* (*to-i i*) (*to-i j*)))
definition *split-at f l* =
(*let f* = λ*x*. ¬ *f x in*
(*takeWhile f l*, *case dropWhile f l of* [] ⇒ (*None*, []) | *x* # *xs* ⇒ (*Some x*, *xs*)))
definition *take* **where** *take reverse lg l* = *reverse* (*snd* (*L.split* (*takeWhile* (λ(*n*, -). *n* < *lg*) (*enumerate 0* (*reverse l*)))))
definition *take-last* = *take rev*
definition *take-first* = *take id*
definition *replace-gen f-res l c0 lby* =
(*let Nsplit-text* = λ*l lgen*. *if l* = [] *then lgen else Nsplit-text l* # *lgen in*
case List.fold
(λ *c1* (*l*, *lgen*).
if c0 c1 then
(*lby*, *Nsplit-sep c1* # *Nsplit-text l lgen*)
else
(*c1* # *l*, *lgen*))
(*rev l*)
([], [])
of (*l*, *lgen*) ⇒ *f-res* (*Nsplit-text l lgen*))
definition *nsplit-f l c0* = *replace-gen id l c0* []
definition *replace* = *replace-gen* (*L.flatten o L.map* (λ *Nsplit-text l* ⇒ *l* | - ⇒ []))

fun *map-find-aux* **where**
map-find-aux accu f l = (λ [] ⇒ *List.rev accu*
| *x* # *xs* ⇒ (*case f x of Some x* ⇒ *List.fold Cons accu* (*x* # *xs*)
| *None* ⇒ *map-find-aux* (*x* # *accu*) *f xs*)) *l*
definition *map-find* = *map-find-aux* []

end
notation *L.append* (**infixr** ‹@@@@› *65*)

lemmas [*code*] =
— def
L.map-def
L.flatten-def
L.mapi-def
L.iter-def
L.maps-def
L.append-def
L.filter-def
L.rev-map-def
L.mapM-def
L.assoc-def
L.split-def
L.upto-def
L.split-at-def
L.take-def
L.take-last-def
L.take-first-def
L.replace-gen-def
L.nsplit-f-def
L.replace-def
L.map-find-def

— fun
L.map-find-aux.simps

1.2.3. Operations on Char

definition *ascii-of-literal* (‹*INT*›) **where**
ascii-of-literal = *hd o String.asciis-of-literal*

definition (*integer-escape* :: *integer*) = *0x09*
definition *ST0 c* = ≪[*c*]≫
definition *ST0-base c* = *ST′* [*c*]

1.2.4. Operations on String (I)

notation *String.asciis-of-literal* (‹*INTS*›)

locale *S*
locale *String*
locale $String_{base}$

definition (**in** *S*) *flatten* = *String-concatWith* ‹›
definition (**in** *String*) *flatten a b* = *S.flatten* [*a*, *b*]
notation *String.flatten* (**infixr** ‹@@› *65*)
definition (**in** *String*) *make n c* = ≪*L.map* (λ-. *c*) (*L.upto 1 n*)≫

definition (**in** $String_{base}$) *map-gen replace g* = (λ *ST s* ⇒ *replace* ‹›) (*Some s*) ‹›
| *ST′ s* ⇒ *S.flatten* (*L.map g s*))
fun (**in** *String*) *map-gen* **where**
map-gen replace g e =
(λ *SS-base s* ⇒ $String_{base}$*.map-gen replace g s*
| *String-concatWith abr l* ⇒ *String-concatWith* (*map-gen replace g abr*) (*List.map* (*map-gen replace g*) *l*)) *e*
definition (**in** *String*) *foldl-one f accu* = *foldl f accu o INTS*
definition (**in** $String_{base}$) *foldl* **where** *foldl f accu* = (λ *ST s* ⇒ *String.foldl-one f accu s*
| *ST′ s* ⇒ *List.foldl f accu s*)
fun (**in** *String*) *foldl* **where**
foldl f accu e =
(λ *SS-base s* ⇒ $String_{base}$*.foldl f accu s*
| *String-concatWith abr l* ⇒
(*case l of* [] ⇒ *accu*
| *x* # *xs* ⇒ *List.foldl* (λ*accu. foldl f* (*foldl f accu abr*)) (*foldl f accu x*) *xs*)) *e*
definition (**in** *S*) *replace-integers f s1 s s2* =
s1 @@ (*case s of None* ⇒ ‹› | *Some s* ⇒ *flatten* (*L.map f* (*INTS s*))) @@ *s2*
definition (**in** *String*) *map* **where** *map f* = *map-gen* (*S.replace-integers* (λ*c.* °*f c*°)) (λ*x.* °*f x*°)
definition (**in** *String*) *replace-integers f* = *map-gen* (*S.replace-integers* (λ*c. f c*)) *f*
definition (**in** *String*) *all f* = *foldl* (λ*b s. b* & *f s*) *True*
definition (**in** *String*) *length* **where** *length* = *foldl* (λ*n* -. *Suc n*) *0*
definition (**in** *String*) *to-list s* = *rev* (*foldl* (λ*l c. c* # *l*) [] *s*)
definition (**in** $String_{base}$) *to-list* = (λ *ST s* ⇒ *INTS s* | *ST′ l* ⇒ *l*)
definition (**in** *String*) *meta-of-logic* = *String.literal-of-asciis o to-list*
definition (**in** *String*) *to-*$String_{base}$ = (λ *SS-base s* ⇒ *s* | *s* ⇒ *ST′* (*to-list s*))
definition (**in** $String_{base}$) *to-String* = *SS-base*
definition (**in** $String_{base}$) *is-empty* = (λ *ST s* ⇒ *s* = *STR* ″″
| *ST′ s* ⇒ *s* = [])
fun (**in** *String*) *is-empty* **where**
is-empty e = (λ *SS-base s* ⇒ $String_{base}$*.is-empty s* | *String-concatWith* - *l* ⇒ *list-all is-empty l*) *e*
definition (**in** *String*) *equal s1 s2* = (*to-list s1* = *to-list s2*)
notation *String.equal* (**infixl** ‹≙› *50*)
definition (**in** *String*) *assoc x l* = *L.assoc* (*to-list x*) (*L.map* (*map-prod* $String_{base}$*.to-list id*) *l*)
definition (**in** *String*) *member l x* = *List.member* (*L.map* $String_{base}$*.to-list l*) (*to-list x*)
definition (**in** $String_{base}$) *flatten l* = $String_{base}$*.to-String* (*S.flatten* (*L.map to-String l*))

lemmas [*code*] =
— def
S.flatten-def
String.flatten-def
String.make-def
$String_{base}$*.map-gen-def*
String.foldl-one-def
$String_{base}$*.foldl-def*
S.replace-integers-def
String.map-def
String.replace-integers-def

String.all-def
String.length-def
String.to-list-def
$String_{base}$*.to-list-def*
String.meta-of-logic-def
String.to-$String_{base}$*-def*
$String_{base}$*.to-String-def*
$String_{base}$*.is-empty-def*
String.equal-def
String.assoc-def
String.member-def
$String_{base}$*.flatten-def*

— fun
String.map-gen.simps
String.foldl.simps
String.is-empty.simps

1.2.5. Operations on String (II)

definition *wildcard* = ‹-›

context *String*
begin
definition *lowercase* = *map* (λ*n*. *if n* < *97 then n* + *32 else n*)
definition *uppercase* = *map* (λ*n*. *if n* < *97 then n else n* − *32*)
definition *to-bold-number* = *replace-integers* (λ*n*. [‹**0**›, ‹**1**›, ‹**2**›, ‹**3**›, ‹**4**›, ‹**5**›, ‹**6**›, ‹**7**›, ‹**8**›, ‹**9**›] ! *nat-of-integer* (*n* − *48*))
fun *nat-to-digit10-aux* **where**
nat-to-digit10-aux l (*n* :: *Nat.nat*) = (*if n* < *10 then n* # *l else nat-to-digit10-aux* (*n mod 10* # *l*) (*n div 10*))
definition *nat-to-digit10 n* =
(*let nat-raw-to-str* = *L.map* (*integer-of-nat o* (+) *0x30*) *in*
≪*nat-raw-to-str* (*nat-to-digit10-aux* [] *n*)≫)
definition *natural-to-digit10* = *nat-to-digit10 o nat-of-natural*

declare[[*cartouche-type* = *String.literal*]]

definition *integer-to-digit16* =
(*let f* = *nth* (*INTS* ‹*0123456789ABCDEF*›) *o nat-of-integer in*
λ*n* ⇒ ≪[*f* (*n div 16*), *f* (*n mod 16*)]≫)
end
lemmas [*code*] =
— def
String.lowercase-def
String.uppercase-def
String.to-bold-number-def
String.nat-to-digit10-def
String.natural-to-digit10-def

String.integer-to-digit16-def

— fun
String.nat-to-digit10-aux.simps

definition *add-0 n* =
(*let n* = *nat-of-integer n in*
S.flatten (*L.map* (λ-. ‹*0*›) (*upt 0* (*if n* < *10 then 2 else if n* < *100 then 1 else 0*)))
@@ *String.nat-to-digit10 n*)

declare[[*cartouche-type* = *String.literal*]]

definition *is-letter* =
(*let int-A* = *INT* ‹*A*›; *int-Z* = *INT* ‹*Z*›; *int-a* = *INT* ‹*a*›; *int-z* = *INT* ‹*z*› *in*
(λ*n. n* ≥ *int-A* & *n* ≤ *int-Z* | *n* ≥ *int-a* & *n* ≤ *int-z*))
definition *is-digit* =
(*let int-0* = *INT* ‹*0*›; *int-9* = *INT* ‹*9*› *in*
(λ*n. n* ≥ *int-0* & *n* ≤ *int-9*))
definition *is-special* = *List.member* (*INTS* ‹ <>^-=−./(){}›)
context *String*
begin
definition *base255* = *replace-integers* (λ*c. if is-letter c then* °*c*° *else add-0 c*)
declare[[*cartouche-type* = *abr-string*]]
definition *isub* =
replace-integers (*let is-und* = *List.member* (*INTS* (*STR* ''-'')) *in*
(λ*c. if is-letter c* | *is-digit c* | *is-und c then* ‹› @@ °*c*° *else add-0 c*))
definition *isup s* = ‹--› @@ *s*
end
lemmas [*code*] =
— def
String.base255-def
String.isub-def
String.isup-def

declare[[*cartouche-type* = *abr-string*]]

definition *text-of-str str* =
(*let s* = ‹*c*›
; *ap* = ‹ # › *in*
S.flatten [‹(*let* ›, *s*, ‹ = *char-of* :: *nat* ⇒ *char in* ›
, *String.replace-integers* (λ*c.*
if is-letter c then
S.flatten [‹*CHR* ''›,°*c*°,‹''›,*ap*]
else
S.flatten [*s*, ‹ ›, *add-0 c*, *ap*])
str
, ‹[])›])
definition ‹*text2-of-str* = *String.replace-integers* (λ*c. S.flatten* [‹\›, ‹<›, °*c*°, ‹>›])›

```
definition textstr-of-str f-flatten f-integer f-str str =
 (let str0 = String.to-list str
    ; f-letter = λc. is-letter c | is-digit c | is-special c
    ; s = ‹c›
    ; f-text = λ Nsplit-text l ⇒ S.flatten [f-str (S.flatten [‹STR ''›,≪l≫,‹''›])]
              | Nsplit-sep c ⇒ S.flatten [f-integer c]
    ; str = case L.nsplit-f str0 (Not o f-letter) of
              [] ⇒ S.flatten [f-str ‹STR ''''›]
            | [x] ⇒ f-text x
            | l ⇒ S.flatten (L.map (λx. ‹(› @@ f-text x @@ ‹) # ›) l) @@ ‹[]› in
  if list-all f-letter str0 then
    str
  else
    f-flatten (S.flatten [ ‹(›, str, ‹)› ]))

definition ‹escape-sml = String.replace-integers (λn. if n = 0x22 then ‹\› else °n°)›
definition mk-constr-name name = (λ x. S.flatten [String.isub name, ‹-›, String.isub x])
definition mk-dot s1 s2 = S.flatten [‹.›, s1, s2]
definition mk-dot-par-gen dot l-s = S.flatten [dot, ‹(›, case l-s of [] ⇒ ‹› | x # xs ⇒ S.flatten
[x, S.flatten (L.map (λs. ‹, › @@ s) xs) ], ‹)›]
definition mk-dot-par dot s = mk-dot-par-gen dot [s]
definition mk-dot-comment s1 s2 s3 = mk-dot s1 (S.flatten [s2, ‹ /*›, s3, ‹*/›])

definition hol-definition s = S.flatten [s, ‹-def›]
definition hol-split s = S.flatten [s, ‹.split›]

end
```

2. Defining Meta-Models

2.1. (Pure) Term Meta-Model aka. AST definition of (Pure) Term

theory *Meta-Pure*
imports *../Init*
begin

2.1.1. Type Definition

type-synonym *indexname* = *string* × *nat*
type-synonym *class* = *string*
type-synonym *sort* = *class list*
datatype *typ* =
  Type string typ list |
  TFree string sort |
  TVar indexname sort
datatype *term* =
  Const string typ |
  Free string typ |
  Var indexname typ |
  Bound nat |
  Abs string typ term |
  App term term (**infixl** ‹$› *200*)

2.1.2. Operations of Fold, Map, ..., on the Meta-Model

fun *map-Const* **where**
  map-Const f expr = (λ *Const s ty* ⇒ *Const* (*f s ty*) *ty*
    | *Free s ty* ⇒ *Free s ty*
    | *Var i ty* ⇒ *Var i ty*
    | *Bound n* ⇒ *Bound n*
    | *Abs s ty term* ⇒ *Abs s ty* (*map-Const f term*)
    | *App term1 term2* ⇒ *App* (*map-Const f term1*)
        (*map-Const f term2*))
    expr

fun *fold-Const* **where**
  fold-Const f accu expr = (λ *Const s* - ⇒ *f accu s*
    | *Abs* - - *term* ⇒ *fold-Const f accu term*
    | *App term1 term2* ⇒ *fold-Const f* (*fold-Const f accu term1*) *term2*
    | - ⇒ *accu*)

expr

fun *fold-Free* **where**
fold-Free f accu expr = (λ *Free s - ⇒ f accu s*
| *Abs - - term ⇒ fold-Free f accu term*
| *App term1 term2 ⇒ fold-Free f* (*fold-Free f accu term1*) *term2*
| *- ⇒ accu*)
expr

end

2.2. SML Meta-Model aka. AST definition of SML

theory *Meta-SML*
imports *../Init*
begin

2.2.1. Type Definition

The following datatypes beginning with `semi__` represent semi-concrete syntax, deliberately not minimal abstract syntax like (Pure) Term, this is for example to facilitate the pretty-printing process, or for manipulating recursively data-structures through an abstract and typed API.

datatype *semi--val-fun* = *Sval*
| *Sfun*

datatype *semi--term′* = *SML-string string*
| *SML-rewrite semi--val-fun semi--term′* — left *string* — symb rewriting *semi--term′* — right
| *SML-basic string list*
| *SML-binop semi--term′ string semi--term′*
| *SML-annot semi--term′ string* — type
| *SML-function* (*semi--term′* — pattern × *semi--term′* — to return) *list*
| *SML-apply semi--term′ semi--term′ list*
| *SML-paren string* — left *string* — right *semi--term′*
| *SML-let-open string semi--term′*

2.2.2. Extending the Meta-Model

locale *SML*
begin
no-type-notation *abr-string* (‹*string*›) **definition** *string* = *SML-string*
definition *rewrite* = *SML-rewrite*
definition *basic* = *SML-basic*
definition *binop* = *SML-binop*
definition *annot* = *SML-annot*
definition *function* = *SML-function*
definition *apply* = *SML-apply*

definition *paren* = *SML-paren*
definition *let-open* = *SML-let-open*

definition *app s* = *apply* (*basic* [*s*])
definition *none* = *basic* [‹*NONE*›]
definition *some s* = *app* ‹*SOME*› [*s*]
definition *option′ f l* = (*case map-option f l of None* ⇒ *none* | *Some s* ⇒ *some s*)
definition *option* = *option′ id*
definition *parenthesis* — mandatory parenthesis = *paren* ‹(› ‹)›
definition *binop-l s l* = (*case rev l of x # xs* ⇒ *List.fold* (λ*x*. *binop x s*) *xs x*)
definition *list l* = (*case l of* [] ⇒ *basic* [‹[]›] | - ⇒ *paren* ‹[› ‹]› (*binop-l* ‹,› *l*))
definition *list′ f l* = *list* (*L.map f l*)
definition *pair e1 e2* = *parenthesis* (*binop e1* ‹,› *e2*)
definition *pair′ f1 f2* = (λ (*e1*, *e2*) ⇒ *parenthesis* (*binop* (*f1 e1*) ‹,› (*f2 e2*)))
definition *rewrite-val* = *rewrite Sval*
definition *rewrite-fun* = *rewrite Sfun*
end

lemmas [*code*] =
— def
SML.string-def
SML.rewrite-def
SML.basic-def
SML.binop-def
SML.annot-def
SML.function-def
SML.apply-def
SML.paren-def
SML.let-open-def
SML.app-def
SML.none-def
SML.some-def
SML.option′-def
SML.option-def
SML.parenthesis-def
SML.binop-l-def
SML.list-def
SML.list′-def
SML.pair-def
SML.pair′-def
SML.rewrite-val-def
SML.rewrite-fun-def

end

2.3. Isabelle Meta-Model aka. AST definition of Isabelle

theory *Meta-Isabelle*
imports *Meta-Pure*

Meta-SML
begin

2.3.1. Type Definition

The following datatypes beginning with `semi__` represent semi-concrete syntax, deliberately not minimal abstract syntax like (Pure) Term, this is for example to facilitate the pretty-printing process, or for manipulating recursively data-structures through an abstract and typed API.

datatype *semi--typ* = *Typ-apply semi--typ semi--typ list*
| *Typ-apply-bin string* — binop *semi--typ semi--typ*
| *Typ-apply-paren string* — left *string* — right *semi--typ*
| *Typ-base string*

datatype *datatype* = *Datatype string* — name
(*string* — name × *semi--typ list* — arguments) *list* — constructors

datatype *type-synonym* = *Type-synonym string* — name
string list — parametric variables
semi--typ — content

datatype *semi--term* = *Term-rewrite semi--term* — left *string* — symb rewriting *semi--term* — right
| *Term-basic string list*
| *Term-annot semi--term semi--typ*
| *Term-bind string* — symbol *semi--term* — arg *semi--term*
| *Term-fun-case semi--term* — value *option* — none: function (*semi--term* — pattern × *semi--term* — to return) *list*
| *Term-apply semi--term semi--term list*
| *Term-paren string* — left *string* — right *semi--term*
| *Term-if-then-else semi--term semi--term semi--term*
| *Term-term string list* — simulate a pre-initialized context (de bruijn variables under "lam")
term — usual continuation of inner syntax term

datatype *type-notation* = *Type-notation string* — name
string — content

datatype *instantiation* = *Instantiation string* — name
string — name in definition
semi--term

datatype *overloading* = *Overloading string* — name consts *semi--term*
string — name def *semi--term* — content

datatype *consts* = *Consts string* — name
semi--typ
string — expression in 'post' mixfix

datatype *definition* = *Definition semi--term*
| *Definition-where1 string* — name *semi--term* — syntax extension × *nat* — priority *semi--term*
| *Definition-where2 string* — name *semi--term* — syntax extension *semi--term*

datatype *semi--thm-attribute* = *Thm-thm string* — represents a single thm
| *Thm-thms string* — represents several thms
| *Thm-THEN semi--thm-attribute semi--thm-attribute*
| *Thm-simplified semi--thm-attribute semi--thm-attribute*
| *Thm-symmetric semi--thm-attribute*
| *Thm-where semi--thm-attribute* (*string* × *semi--term*) *list*
| *Thm-of semi--thm-attribute semi--term list*
| *Thm-OF semi--thm-attribute semi--thm-attribute*

datatype *semi--thm* = *Thms-single semi--thm-attribute*
| *Thms-mult semi--thm-attribute*

type-synonym *semi--thm-l* = *semi--thm list*

datatype *lemmas* = *Lemmas-simp-thm bool* — True : simp
string — name
semi--thm-attribute list
| *Lemmas-simp-thms string* — name
string — thms *list*

datatype *semi--method-simp* = *Method-simp-only semi--thm-l*
| *Method-simp-add-del-split semi--thm-l* — add *semi--thm-l* — del *semi--thm-l* — split

datatype *semi--method* = *Method-rule semi--thm-attribute option*
| *Method-drule semi--thm-attribute*
| *Method-erule semi--thm-attribute*
| *Method-intro semi--thm-attribute list*
| *Method-elim semi--thm-attribute*
| *Method-subst bool* — asm
string — nat *list* — pos
semi--thm-attribute
| *Method-insert semi--thm-l*
| *Method-plus semi--method list*
| *Method-option semi--method list*
| *Method-or semi--method list*
| *Method-one semi--method-simp*
| *Method-all semi--method-simp*
| *Method-auto-simp-add-split semi--thm-l string list*
| *Method-rename-tac string list*
| *Method-case-tac semi--term*
| *Method-blast nat option*
| *Method-clarify*

| *Method-metis string list* — e.g. *no-types* (*override-type-encs*)
semi--thm-attribute list

datatype *semi--command-final* = *Command-done*
| *Command-by semi--method list*
| *Command-sorry*

datatype *semi--command-state* = *Command-apply-end semi--method list* — **apply-end** (. . ., . . .)

datatype *semi--command-proof* = *Command-apply semi--method list* — **apply** (. . ., . . .)
| *Command-using semi--thm-l* — **using** . . .
| *Command-unfolding semi--thm-l* — **unfolding** . . .
| *Command-let semi--term* — name *semi--term*
| *Command-have string* — name
bool — true: add [*simp*]
semi--term
semi--command-final
| *Command-fix-let string list*
(*semi--term* — name × *semi--term*) *list* — let statements
(*semi--term list* — **show** . . . ⟹ . . .
× *semi--term list* — **when** . . . . . .) *option* — *None* ⇒ *?thesis*
semi--command-state list — **qed apply-end** . . .

datatype *lemma* = *Lemma string* — name *semi--term list* — specification to prove
semi--method list list — tactics: **apply** (. . ., . . .) **apply** . . .
semi--command-final
| *Lemma-assumes string* — name
(*string* — name × *bool* — true: add [*simp*] × *semi--term*) *list* — specification to prove (assms)
semi--term — specification to prove (conclusion)
semi--command-proof list
semi--command-final

datatype *axiomatization* = *Axiomatization string* — name
semi--term

datatype *section* = *Section nat* — nesting level
string — content

datatype *text* = *Text string*

datatype *ML* = *SML semi--term′*

datatype *setup* = *Setup semi--term′*

datatype *thm* = *Thm semi--thm-attribute list*

datatype *interpretation* = *Interpretation string* — name
string — locale name
semi--term list — locale param
semi--command-final

datatype *semi--theory* = *Theory-datatype datatype*
| *Theory-type-synonym type-synonym*
| *Theory-type-notation type-notation*
| *Theory-instantiation instantiation*
| *Theory-overloading overloading*
| *Theory-consts consts*
| *Theory-definition definition*
| *Theory-lemmas lemmas*
| *Theory-lemma lemma*
| *Theory-axiomatization axiomatization*
| *Theory-section section*
| *Theory-text text*
| *Theory-ML ML*
| *Theory-setup setup*
| *Theory-thm thm*
| *Theory-interpretation interpretation*

record *semi--locale* =
HolThyLocale-name :: *string*
HolThyLocale-header :: ((*semi--term* — name × *semi--typ* — **fix** statement) *list*
× (*string* — name × *semi--term* — **assumes** statement) *option* — None: no **assumes** to generate) *list*

datatype *semi--theories* = *Theories-one semi--theory*
| *Theories-locale semi--locale semi--theory list* — positioning comments can occur before and after this group of commands *list*

2.3.2. Extending the Meta-Model

locale *T*
begin
definition *thm* = *Thm-thm*
definition *thms* = *Thm-thms*
definition *THEN* = *Thm-THEN*
definition *simplified* = *Thm-simplified*
definition *symmetric* = *Thm-symmetric*
definition *where* = *Thm-where*
definition *of'* = *Thm-of*
definition *OF* = *Thm-OF*
definition *OF-l s l* = *List.fold* (λ*x acc. Thm-OF acc x*) *l s*
definition *simplified-l s l* = *List.fold* (λ*x acc. Thm-simplified acc x*) *l s*
end

lemmas [*code*] =

— def
T.thm-def
T.thms-def
T.THEN-def
T.simplified-def
T.symmetric-def
T.where-def
T.of'-def
T.OF-def
T.OF-l-def
T.simplified-l-def

definition *Opt s = Typ-apply (Typ-base ‹option›) [Typ-base s]*
definition *Raw = Typ-base*
definition *Type-synonym' n = Type-synonym n []*
definition *Type-synonym'' n l f = Type-synonym n l (f l)*
definition *Term-annot' e s = Term-annot e (Typ-base s)*
definition *Term-lambdas s = Term-bind ‹λ› (Term-basic s)*
definition *Term-lambda x = Term-lambdas [x]*
definition *Term-lambdas0 = Term-bind ‹λ›*
definition *Term-lam x f = Term-lambdas0 (Term-basic [x]) (f x)*
definition *Term-some = Term-paren ‹⌊› ‹⌋›*
definition *Term-parenthesis* — mandatory parenthesis *= Term-paren ‹(› ‹)›*
definition *Term-warning-parenthesis* — optional parenthesis that can be removed but a warning will be raised *= Term-parenthesis*
definition *Term-pat b = Term-basic [‹?› @@ b]*
definition *Term-And x f = Term-bind ‹⋀› (Term-basic [x]) (f x)*
definition *Term-exists x f = Term-bind ‹∃ › (Term-basic [x]) (f x)*
definition *Term-binop = Term-rewrite*
definition *term-binop s l = (case rev l of x # xs ⇒ List.fold (λx. Term-binop x s) xs x)*
definition *term-binop' s l = (case rev l of x # xs ⇒ List.fold (λx. Term-parenthesis o Term-binop x s) xs x)*
definition *Term-set l = (case l of [] ⇒ Term-basic [‹{}›] | - ⇒ Term-paren ‹{› ‹}› (term-binop ‹,› l))*
definition *Term-list l = (case l of [] ⇒ Term-basic [‹[]›] | - ⇒ Term-paren ‹[› ‹]› (term-binop ‹,› l))*
definition *Term-list' f l = Term-list (L.map f l)*
definition *Term-pair e1 e2 = Term-parenthesis (Term-binop e1 ‹,› e2)*
definition *Term-pair' l = (case l of [] ⇒ Term-basic [‹()›] | - ⇒ Term-paren ‹(› ‹)› (term-binop ‹,› l))*
definition *‹Term-string s = Term-basic [S.flatten [‹›, s, ‹›]]›*
definition *Term-applys0 e l = Term-parenthesis (Term-apply e (L.map Term-parenthesis l))*
definition *Term-applys e l = Term-applys0 (Term-parenthesis e) l*
definition *Term-app e = Term-applys0 (Term-basic [e])*
definition *Term-preunary e1 e2 = Term-apply e1 [e2]* — no parenthesis and separated with one space
definition *Term-postunary e1 e2 = Term-apply e1 [e2]* — no parenthesis and separated with one space
definition *Term-case = Term-fun-case o Some*

definition *Term-function* = *Term-fun-case None*
definition *Term-term'* = *Term-term* []
definition *Lemmas-simp* = *Lemmas-simp-thm True*
definition *Lemmas-nosimp* = *Lemmas-simp-thm False*
definition *Consts-value* = ‹(-)›
definition *Consts-raw0 s l e o-arg* =
Consts s l (*String.replace-integers* (λ*n*. *if n* = *0x5F then* ‹'-› *else* °*n*°) *e* @@ (*case o-arg of*
None ⇒ ‹›
| *Some arg* ⇒
let ap = λ*s*. ‹'(› @@ *s* @@ ‹')› *in*
ap (*if arg* = *0 then*
‹›
else
Consts-value @@ (*S.flatten* (*L.map* (λ-. ‹,› @@ *Consts-value*) (*L.upto 2 arg*))))))
definition *Ty-arrow* = *Typ-apply-bin* ‹⇒›
definition *Ty-times* = *Typ-apply-bin* ‹×›
definition *Ty-arrow' x* = *Ty-arrow x* (*Typ-base* ‹-›)
definition *Ty-paren* = *Typ-apply-paren* ‹(› ‹)›
definition *Consts' s l e* = *Consts-raw0 s* (*Ty-arrow* (*Typ-base* ‹'α›) *l*) *e None*
definition *Overloading' n ty* = *Overloading n* (*Term-annot* (*Term-basic* [*n*]) *ty*)

locale *M*
begin
definition *Method-simp-add-del l-a l-d* = *Method-simp-add-del-split l-a l-d* []
definition *Method-subst-l* = *Method-subst False*

definition *rule'* = *Method-rule None*
definition *rule* = *Method-rule o Some*
definition *drule* = *Method-drule*
definition *erule* = *Method-erule*
definition *intro* = *Method-intro*
definition *elim* = *Method-elim*
definition *subst-l0* = *Method-subst*
definition *subst-l* = *Method-subst-l*
definition *insert* **where** *insert* = *Method-insert o L.map Thms-single*
definition *plus* **where** *plus* = *Method-plus*
definition *option* = *Method-option*
definition *or* **where** *or* = *Method-or*
definition *meth-gen-simp* = *Method-simp-add-del* [] []
definition *meth-gen-simp-add2 l1 l2* = *Method-simp-add-del* (*L.flatten* [*L.map Thms-mult l1*
, *L.map* (*Thms-single o Thm-thm*) *l2*])
[]
definition *meth-gen-simp-add-del l1 l2* = *Method-simp-add-del* (*L.map* (*Thms-single o Thm-thm*)
l1)
(*L.map* (*Thms-single o Thm-thm*) *l2*)
definition *meth-gen-simp-add-del-split l1 l2 l3* = *Method-simp-add-del-split* (*L.map Thms-single*
l1)
(*L.map Thms-single l2*)
(*L.map Thms-single l3*)

definition *meth-gen-simp-add-split l1 l2 = Method-simp-add-del-split* (*L.map Thms-single l1*)
[]
(*L.map Thms-single l2*)
definition *meth-gen-simp-only l = Method-simp-only* (*L.map Thms-single l*)
definition *meth-gen-simp-only′ l = Method-simp-only* (*L.map Thms-mult l*)
definition *meth-gen-simp-add0 l = Method-simp-add-del* (*L.map Thms-single l*) []
definition *simp = Method-one meth-gen-simp*
definition *simp-add2 l1 l2 = Method-one* (*meth-gen-simp-add2 l1 l2*)
definition *simp-add-del l1 l2 = Method-one* (*meth-gen-simp-add-del l1 l2*)
definition *simp-add-del-split l1 l2 l3 = Method-one* (*meth-gen-simp-add-del-split l1 l2 l3*)
definition *simp-add-split l1 l2 = Method-one* (*meth-gen-simp-add-split l1 l2*)
definition *simp-only l = Method-one* (*meth-gen-simp-only l*)
definition *simp-only′ l = Method-one* (*meth-gen-simp-only′ l*)
definition *simp-add0 l = Method-one* (*meth-gen-simp-add0 l*)
definition *simp-add = simp-add2* []
definition *simp-all = Method-all meth-gen-simp*
definition *simp-all-add l = Method-all* (*meth-gen-simp-add2* [] *l*)
definition *simp-all-only l = Method-all* (*meth-gen-simp-only l*)
definition *simp-all-only′ l = Method-all* (*meth-gen-simp-only′ l*)
definition *auto-simp-add2 l1 l2 = Method-auto-simp-add-split* (*L.flatten* [*L.map Thms-mult l1*
, *L.map* (*Thms-single o Thm-thm*) *l2*]) []
definition *auto-simp-add-split l = Method-auto-simp-add-split* (*L.map Thms-single l*)
definition *rename-tac = Method-rename-tac*
definition *case-tac = Method-case-tac*
definition *blast = Method-blast*
definition *clarify = Method-clarify*
definition *metis = Method-metis* []
definition *metis0 = Method-metis*

definition *subst-asm b = subst-l0 b* [‹*0*›]
definition *subst = subst-l* [‹*0*›]
definition *auto-simp-add = auto-simp-add2* []
definition *auto = auto-simp-add* []
end

lemmas [*code*] =
— def
M.Method-simp-add-del-def
M.Method-subst-l-def
M.rule′-def
M.rule-def
M.drule-def
M.erule-def
M.intro-def
M.elim-def
M.subst-l0-def
M.subst-l-def
M.insert-def
M.plus-def

M.option-def
M.or-def
M.meth-gen-simp-def
M.meth-gen-simp-add2-def
M.meth-gen-simp-add-del-def
M.meth-gen-simp-add-del-split-def
M.meth-gen-simp-add-split-def
M.meth-gen-simp-only-def
M.meth-gen-simp-only′-def
M.meth-gen-simp-add0-def
M.simp-def
M.simp-add2-def
M.simp-add-del-def
M.simp-add-del-split-def
M.simp-add-split-def
M.simp-only-def
M.simp-only′-def
M.simp-add0-def
M.simp-add-def
M.simp-all-def
M.simp-all-add-def
M.simp-all-only-def
M.simp-all-only′-def
M.auto-simp-add2-def
M.auto-simp-add-split-def
M.rename-tac-def
M.case-tac-def
M.blast-def
M.clarify-def
M.metis-def
M.metis0-def
M.subst-asm-def
M.subst-def
M.auto-simp-add-def
M.auto-def

definition *ty-arrow l* = (*case rev l of x # xs* ⇒ *List.fold Ty-arrow xs x*)

locale *C*
begin
definition *done* = *Command-done*
definition *by* = *Command-by*
definition *sorry* = *Command-sorry*
definition *apply-end* = *Command-apply-end*
definition *apply* = *Command-apply*
definition *using* = *Command-using o L.map Thms-single*
definition *unfolding* = *Command-unfolding o L.map Thms-single*
definition *let′* = *Command-let*
definition *fix-let* = *Command-fix-let*

definition *fix l = Command-fix-let l [] None []*
definition *have n = Command-have n False*
definition *have0 = Command-have*
end

lemmas [*code*] =
— def
C.done-def
C.by-def
C.sorry-def
C.apply-end-def
C.apply-def
C.using-def
C.unfolding-def
C.let′-def
C.fix-let-def
C.fix-def
C.have-def
C.have0-def

fun *cross-abs-aux* **where**
cross-abs-aux f l x = (λ (Suc n, Abs s - t) ⇒ f s (cross-abs-aux f (s # l) (n, t))
| (-, e) ⇒ Term-term l e)
x

definition *cross-abs f n l = cross-abs-aux f [] (n, l)*

2.3.3. Operations of Fold, Map, ..., on the Meta-Model

definition *map-lemma f = (λ Theory-lemma x ⇒ Theory-lemma (f x)*
| x ⇒ x)

end

3. Parsing Meta-Models

3.1. Initializing the Parser

theory *Parser-init*
imports *../Init*
begin

3.1.1. Some Generic Combinators

definition *K x - = x*

definition *co1 = (o)*
definition *co2 f g x1 x2 = f (g x1 x2)*
definition *co3 f g x1 x2 x3 = f (g x1 x2 x3)*
definition *co4 f g x1 x2 x3 x4 = f (g x1 x2 x3 x4)*
definition *co5 f g x1 x2 x3 x4 x5 = f (g x1 x2 x3 x4 x5)*
definition *co6 f g x1 x2 x3 x4 x5 x6 = f (g x1 x2 x3 x4 x5 x6)*
definition *co7 f g x1 x2 x3 x4 x5 x6 x7 = f (g x1 x2 x3 x4 x5 x6 x7)*
definition *co8 f g x1 x2 x3 x4 x5 x6 x7 x8 = f (g x1 x2 x3 x4 x5 x6 x7 x8)*
definition *co9 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9)*
definition *co10 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10)*
definition *co11 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11)*
definition *co12 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12)*
definition *co13 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13)*
definition *co14 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14)*
definition *co15 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15)*

definition *ap1 a v0 f1 v1 = a v0 [f1 v1]*
definition *ap2 a v0 f1 f2 v1 v2 = a v0 [f1 v1, f2 v2]*
definition *ap3 a v0 f1 f2 f3 v1 v2 v3 = a v0 [f1 v1, f2 v2, f3 v3]*
definition *ap4 a v0 f1 f2 f3 f4 v1 v2 v3 v4 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4]*
definition *ap5 a v0 f1 f2 f3 f4 f5 v1 v2 v3 v4 v5 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5]*
definition *ap6 a v0 f1 f2 f3 f4 f5 f6 v1 v2 v3 v4 v5 v6 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6]*
definition *ap7 a v0 f1 f2 f3 f4 f5 f6 f7 v1 v2 v3 v4 v5 v6 v7 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7]*
definition *ap8 a v0 f1 f2 f3 f4 f5 f6 f7 f8 v1 v2 v3 v4 v5 v6 v7 v8 = a v0 [f1 v1, f2 v2, f3 v3,*

f4 v4, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*]
definition *ap9 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 v1 v2 v3 v4 v5 v6 v7 v8 v9 = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*]
definition *ap10 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*]
definition *ap11 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*]
definition *ap12 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *f12 v12*]
definition *ap13 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *f12 v12*, *f13 v13*]
definition *ap14 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *f12 v12*, *f13 v13*, *f14 v14*]
definition *ap15 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14 f15 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *f12 v12*, *f13 v13*, *f14 v14*, *f15 v15*]

definition *ar1 a v0 z = a v0* [*z*]
definition *ar2 a v0 f1 v1 z = a v0* [*f1 v1*, *z*]
definition *ar3 a v0 f1 f2 v1 v2 z = a v0* [*f1 v1*, *f2 v2*, *z*]
definition *ar4 a v0 f1 f2 f3 v1 v2 v3 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *z*]
definition *ar5 a v0 f1 f2 f3 f4 v1 v2 v3 v4 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *z*]
definition *ar6 a v0 f1 f2 f3 f4 f5 v1 v2 v3 v4 v5 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *z*]
definition *ar7 a v0 f1 f2 f3 f4 f5 f6 v1 v2 v3 v4 v5 v6 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *z*]
definition *ar8 a v0 f1 f2 f3 f4 f5 f6 f7 v1 v2 v3 v4 v5 v6 v7 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *z*]
definition *ar9 a v0 f1 f2 f3 f4 f5 f6 f7 f8 v1 v2 v3 v4 v5 v6 v7 v8 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *z*]
definition *ar10 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 v1 v2 v3 v4 v5 v6 v7 v8 v9 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *z*]
definition *ar11 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *z*]
definition *ar12 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *z*]
definition *ar13 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *f12 v12*, *z*]
definition *ar14 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *f12 v12*, *f13 v13*, *z*]
definition *ar15 a v0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 z = a v0* [*f1 v1*, *f2 v2*, *f3 v3*, *f4 v4*, *f5 v5*, *f6 v6*, *f7 v7*, *f8 v8*, *f9 v9*, *f10 v10*, *f11 v11*, *f12 v12*, *f13 v13*, *f14 v14*, *z*]

3.1.2. Generic Locale for Parsing

locale *Parse* =
  fixes *ext* :: *string* ⇒ *string*

  — (effective) first order
  fixes *of-string* :: (′*a* ⇒ ′*a list* ⇒ ′*a*) ⇒ (*string* ⇒ ′*a*) ⇒ *string* ⇒ ′*a*
  fixes *of-string*$_{base}$ :: (′*a* ⇒ ′*a list* ⇒ ′*a*) ⇒ (*string* ⇒ ′*a*) ⇒ *string*$_{base}$ ⇒ ′*a*
  fixes *of-nat* :: (′*a* ⇒ ′*a list* ⇒ ′*a*) ⇒ (*string* ⇒ ′*a*) ⇒ *natural* ⇒ ′*a*
  fixes *of-unit* :: (*string* ⇒ ′*a*) ⇒ *unit* ⇒ ′*a*
  fixes *of-bool* :: (*string* ⇒ ′*a*) ⇒ *bool* ⇒ ′*a*

  — (simulation) higher order
  fixes *Of-Pair Of-Nil Of-Cons Of-None Of-Some* :: *string*
begin

definition *of-pair a b f1 f2* = (λ*f*. λ(*c*, *d*) ⇒ *f c d*)
  (*ap2 a* (*b Of-Pair*) *f1 f2*)

definition *of-list a b f* = (λ*f0*. *rec-list f0 o co1 K*)
  (*b Of-Nil*)
  (*ar2 a* (*b Of-Cons*) *f*)

definition *of-option a b f* = *rec-option*
  (*b Of-None*)
  (*ap1 a* (*b Of-Some*) *f*)

end

lemmas [*code*] =
  Parse.of-pair-def
  Parse.of-list-def
  Parse.of-option-def

This theory and all the deriving one could also be prefixed by “print” instead of “parse”. In any case, we are converting (or printing) the above datatypes to another format, and finally this format will be “parsed” by Isabelle!

end

3.2. Instantiating the Parser of (Pure) Term

theory *Parser-Pure*
imports *Meta-Pure*
  Parser-init
begin

3.2.1. Main

context *Parse*

begin

definition *of-pure-indexname a b = of-pair a b (of-string a b) (of-nat a b)*

definition *of-pure-class = of-string*

definition *of-pure-sort a b = of-list a b (of-pure-class a b)*

definition *of-pure-typ a b = rec-typ*
(ap2 a (b ‹Type›) (of-string a b) (of-list a b snd))
(ap2 a (b ‹TFree›) (of-string a b) (of-pure-sort a b))
(ap2 a (b ‹TVar›) (of-pure-indexname a b) (of-pure-sort a b))

definition *of-pure-term a b = (λf0 f1 f2 f3 f4 f5. rec-term f0 f1 f2 f3 (co2 K f4) (λ- -. f5))*
(ap2 a (b ‹Const›) (of-string a b) (of-pure-typ a b))
(ap2 a (b ‹Free›) (of-string a b) (of-pure-typ a b))
(ap2 a (b ‹Var›) (of-pure-indexname a b) (of-pure-typ a b))
(ap1 a (b ‹Bound›) (of-nat a b))
(ar3 a (b ‹Abs›) (of-string a b) (of-pure-typ a b))
(ar2 a (b ‹App›) id)

end

lemmas [*code*] =
Parse.of-pure-indexname-def
Parse.of-pure-class-def
Parse.of-pure-sort-def
Parse.of-pure-typ-def
Parse.of-pure-term-def

end

4. Printing Meta-Models

4.1. Initializing the Printer

theory *Printer-init*
imports *../Init*
../isabelle-home/src/HOL/Isabelle-Main1
begin

At the time of writing, the following target languages supported by Isabelle are also supported by the meta-compiler: Haskell, OCaml, Scala, SML.

4.1.1. Kernel Code for Target Languages

lazy-code-printing code-module *CodeType* ⇀ (*Haskell*) ‹
module CodeType where {
type MlInt = Integer
; *type MlMonad a = IO a*
}› | **code-module** *CodeConst* ⇀ (*Haskell*) ‹
module CodeConst where {
import System.Directory
; *import System.IO*
; *import qualified CodeConst.Printf*

; *outFile1 f file = (do*
fileExists <− doesFileExist file
if fileExists then error (File exists ++ file ++ \n) else do
h <− openFile file WriteMode
f (\pat −> hPutStr h . CodeConst.Printf.sprintf1 pat)
hClose h)

; *outStand1 :: ((String −> String −> IO ()) −> IO ()) −> IO ()*
; *outStand1 f = f (\pat −> putStr . CodeConst.Printf.sprintf1 pat)*
}› | **code-module** *CodeConst.Monad* ⇀ (*Haskell*) ‹
module CodeConst.Monad where {
bind a = (>>=) a
; *return :: a −> IO a*
; *return = Prelude.return*
}› | **code-module** *CodeConst.Printf* ⇀ (*Haskell*) ‹
module CodeConst.Printf where {
import Text.Printf
; *sprintf0 = id*

```
; sprintf1 :: PrintfArg a => String -> a -> String
; sprintf1 = printf

; sprintf2 :: PrintfArg a => PrintfArg b => String -> a -> b -> String
; sprintf2 = printf

; sprintf3 :: PrintfArg a => PrintfArg b => PrintfArg c => String -> a -> b -> c ->
String
; sprintf3 = printf

; sprintf4 :: PrintfArg a => PrintfArg b => PrintfArg c => PrintfArg d => String -> a ->
b -> c -> d -> String
; sprintf4 = printf

; sprintf5 :: PrintfArg a => PrintfArg b => PrintfArg c => PrintfArg d => PrintfArg e =>
String -> a -> b -> c -> d -> e -> String
; sprintf5 = printf
}› | code-module CodeConst.String ⇀ (Haskell) ‹
module CodeConst.String where {
  concat s [] = []
; concat s (x : xs) = x ++ concatMap ((++) s) xs
}› | code-module CodeConst.Sys ⇀ (Haskell) ‹
module CodeConst.Sys where {
  import System.Directory
; isDirectory2 = doesDirectoryExist
}› | code-module CodeConst.To ⇀ (Haskell) ‹
module CodeConst.To where {
  nat = id
}› | code-module  ⇀ (OCaml) ‹
module CodeType = struct
  type mlInt = int

  type 'a mlMonad = 'a option
end

module CodeConst = struct
  let outFile1 f file =
    try
      let () = if Sys.file-exists file then Printf.eprintf File exists \%S\\n file else () in
      let oc = open-out file in
      let b = f (fun s a -> try Some (Printf.fprintf oc s a) with - -> None) in
      let () = close-out oc in
      b
    with - -> None

  let outStand1 f =
    f (fun s a -> try Some (Printf.fprintf stdout s a) with - -> None)

  module Monad = struct
```

```
  let bind = function
      None -> fun - -> None
    | Some a -> fun f -> f a
  let return a = Some a
end

module Printf = struct
  include Printf
  let sprintf0 = sprintf
  let sprintf1 = sprintf
  let sprintf2 = sprintf
  let sprintf3 = sprintf
  let sprintf4 = sprintf
  let sprintf5 = sprintf
end

module String = String

module Sys = struct open Sys
  let isDirectory2 s = try Some (is-directory s) with - -> None
end

module To = struct
  let nat big-int x = Big-int.int-of-big-int (big-int x)
end
end

› | code-module  ⇀ (Scala) ‹
object CodeType {
  type mlMonad [A] = Option [A]
  type mlInt = Int
}

object CodeConst {
  def outFile1 [A] (f : (String => A => Option [Unit]) => Option [Unit], file0 : String) :
Option [Unit] = {
    val file = new java.io.File (file0)
    file .isFile match {
      case true => None
      case false =>
        val writer = new java.io.PrintWriter (file)
        f ((fmt : String) => (s : A) => Some (writer .write (fmt .format (s))))
        Some (writer .close ())
    }
  }

  def outStand1 [A] (f : (String => A => Option [Unit]) => Option [Unit]) : Option[Unit] =
{
    f ((fmt : String) => (s : A) => Some (print (fmt .format (s))))
```

```
  }

  object Monad {
    def bind [A, B] (x : Option [A], f : A => Option [B]) : Option [B] = x match {
      case None => None
      case Some (a) => f (a)
    }
    def Return [A] (a : A) = Some (a)
  }
  object Printf {
    def sprintf0 (x0 : String) = x0
    def sprintf1 [A1] (fmt : String, x1 : A1) = fmt .format (x1)
    def sprintf2 [A1, A2] (fmt : String, x1 : A1, x2 : A2) = fmt .format (x1, x2)
    def sprintf3 [A1, A2, A3] (fmt : String, x1 : A1, x2 : A2, x3 : A3) = fmt .format (x1, x2,
x3)
     def sprintf4 [A1, A2, A3, A4] (fmt : String, x1 : A1, x2 : A2, x3 : A3, x4 : A4) = fmt
.format (x1, x2, x3, x4)
    def sprintf5 [A1, A2, A3, A4, A5] (fmt : String, x1 : A1, x2 : A2, x3 : A3, x4 : A4, x5 :
A5) = fmt .format (x1, x2, x3, x4, x5)
  }
  object String {
    def concat (s : String, l : List [String]) = l filter (- .nonEmpty) mkString s
  }
  object Sys {
    def isDirectory2 (s : String) = Some (new java.io.File (s) .isDirectory)
  }
  object To {
    def nat [A] (f : A => BigInt, x : A) = f (x) .intValue ()
  }
}

› | code-module ⇀ (SML) ‹
structure CodeType = struct
  type mlInt = string
  type 'a mlMonad = 'a option
end

structure CodeConst = struct
  structure Monad = struct
    val bind = fn
        NONE => (fn - => NONE)
      | SOME a => fn f => f a
    val return = SOME
  end

  structure Printf = struct
    local
      fun sprintf s l =
        case String.fields (fn #% => true | - => false) s of
```

```
        [] =>
      | [x] => x
      | x :: xs =>
          let fun aux acc l-pat l-s =
            case l-pat of
              [] => rev acc
            | x :: xs => aux (String.extract (x, 1, NONE) :: hd l-s :: acc) xs (tl l-s) in
          String.concat (x :: aux [] xs l)
    end
  in
    fun sprintf0 s-pat = s-pat
    fun sprintf1 s-pat s1 = sprintf s-pat [s1]
    fun sprintf2 s-pat s1 s2 = sprintf s-pat [s1, s2]
    fun sprintf3 s-pat s1 s2 s3 = sprintf s-pat [s1, s2, s3]
    fun sprintf4 s-pat s1 s2 s3 s4 = sprintf s-pat [s1, s2, s3, s4]
    fun sprintf5 s-pat s1 s2 s3 s4 s5 = sprintf s-pat [s1, s2, s3, s4, s5]
  end
end

structure String = struct
  val concat = String.concatWith
end

structure Sys = struct
  val isDirectory2 = SOME o File.is-dir o Path.explode handle ERROR - => K NONE
end

structure To = struct
  fun nat f = Int.toString o f
end

fun outFile1 f file =
  let
    val pfile = Path.explode file
    val () = if File.exists pfile then error (File exists \ ^ file ^ \\n) else ()
    val oc = Unsynchronized.ref []
    val - = f (fn a => fn b => SOME (oc := Printf.sprintf1 a b :: (Unsynchronized.! oc))) in
    try (fn () => File.write-list pfile (rev (Unsynchronized.! oc))) ()
  end

fun outStand1 f = outFile1 f (Unsynchronized.! stdout-file)
end

›
```

4.1.2. Interface with Types

datatype *ml-int* = *ML-int*
code-printing type-constructor *ml-int* ⇀ (*Haskell*) *CodeType.MlInt* — syntax!

| **type-constructor** *ml-int* ⇀ (*OCaml*) *CodeType.mlInt*
| **type-constructor** *ml-int* ⇀ (*Scala*) *CodeType.mlInt*
| **type-constructor** *ml-int* ⇀ (*SML*) *CodeType.mlInt*

datatype *'a ml-monad* = *ML-monad 'a*
code-printing type-constructor *ml-monad* ⇀ (*Haskell*) *CodeType.MlMonad* - — syntax!
| **type-constructor** *ml-monad* ⇀ (*OCaml*) - *CodeType.mlMonad*
| **type-constructor** *ml-monad* ⇀ (*Scala*) *CodeType.mlMonad* [-]
| **type-constructor** *ml-monad* ⇀ (*SML*) - *CodeType.mlMonad*

type-synonym *ml-string* = *String.literal*

4.1.3. Interface with Constants

module CodeConst

consts *out-file1* :: ((*ml-string* ⇒ *'α1* ⇒ *unit ml-monad*) — fprintf ⇒ *unit ml-monad*) ⇒ *ml-string* ⇒ *unit ml-monad*
code-printing constant *out-file1* ⇀ (*Haskell*) *CodeConst.outFile1*
| **constant** *out-file1* ⇀ (*OCaml*) *CodeConst.outFile1*
| **constant** *out-file1* ⇀ (*Scala*) *CodeConst.outFile1*
| **constant** *out-file1* ⇀ (*SML*) *CodeConst.outFile1*

consts *out-stand1* :: ((*ml-string* ⇒ *'α1* ⇒ *unit ml-monad*) — fprintf ⇒ *unit ml-monad*) ⇒ *unit ml-monad*
code-printing constant *out-stand1* ⇀ (*Haskell*) *CodeConst.outStand1*
| **constant** *out-stand1* ⇀ (*OCaml*) *CodeConst.outStand1*
| **constant** *out-stand1* ⇀ (*Scala*) *CodeConst.outStand1*
| **constant** *out-stand1* ⇀ (*SML*) *CodeConst.outStand1*

module Monad

consts *bind* :: *'a ml-monad* ⇒ (*'a* ⇒ *'b ml-monad*) ⇒ *'b ml-monad*
code-printing constant *bind* ⇀ (*Haskell*) *CodeConst.Monad.bind*
| **constant** *bind* ⇀ (*OCaml*) *CodeConst.Monad.bind*
| **constant** *bind* ⇀ (*Scala*) *CodeConst.Monad.bind*
| **constant** *bind* ⇀ (*SML*) *CodeConst.Monad.bind*

consts *return* :: *'a* ⇒ *'a ml-monad*
code-printing constant *return* ⇀ (*Haskell*) *CodeConst.Monad.return*
| **constant** *return* ⇀ (*OCaml*) *CodeConst.Monad.return*
| **constant** *return* ⇀ (*Scala*) *CodeConst.Monad.Return* — syntax!
| **constant** *return* ⇀ (*SML*) *CodeConst.Monad.return*

module Printf

consts *sprintf0* :: *ml-string* ⇒ *ml-string*
code-printing constant *sprintf0* ⇀ (*Haskell*) *CodeConst.Printf.sprintf0*
| **constant** *sprintf0* ⇀ (*OCaml*) *CodeConst.Printf.sprintf0*
| **constant** *sprintf0* ⇀ (*Scala*) *CodeConst.Printf.sprintf0*
| **constant** *sprintf0* ⇀ (*SML*) *CodeConst.Printf.sprintf0*

consts *sprintf1* :: *ml-string* ⇒ *'α1* ⇒ *ml-string*
code-printing constant *sprintf1* ⇀ (*Haskell*) *CodeConst.Printf.sprintf1*
| **constant** *sprintf1* ⇀ (*OCaml*) *CodeConst.Printf.sprintf1*
| **constant** *sprintf1* ⇀ (*Scala*) *CodeConst.Printf.sprintf1*
| **constant** *sprintf1* ⇀ (*SML*) *CodeConst.Printf.sprintf1*

consts *sprintf2* :: *ml-string* ⇒ *'α1* ⇒ *'α2* ⇒ *ml-string*
code-printing constant *sprintf2* ⇀ (*Haskell*) *CodeConst.Printf.sprintf2*
| **constant** *sprintf2* ⇀ (*OCaml*) *CodeConst.Printf.sprintf2*
| **constant** *sprintf2* ⇀ (*Scala*) *CodeConst.Printf.sprintf2*
| **constant** *sprintf2* ⇀ (*SML*) *CodeConst.Printf.sprintf2*

consts *sprintf3* :: *ml-string* ⇒ *'α1* ⇒ *'α2* ⇒ *'α3* ⇒ *ml-string*
code-printing constant *sprintf3* ⇀ (*Haskell*) *CodeConst.Printf.sprintf3*
| **constant** *sprintf3* ⇀ (*OCaml*) *CodeConst.Printf.sprintf3*
| **constant** *sprintf3* ⇀ (*Scala*) *CodeConst.Printf.sprintf3*
| **constant** *sprintf3* ⇀ (*SML*) *CodeConst.Printf.sprintf3*

consts *sprintf4* :: *ml-string* ⇒ *'α1* ⇒ *'α2* ⇒ *'α3* ⇒ *'α4* ⇒ *ml-string*
code-printing constant *sprintf4* ⇀ (*Haskell*) *CodeConst.Printf.sprintf4*
| **constant** *sprintf4* ⇀ (*OCaml*) *CodeConst.Printf.sprintf4*
| **constant** *sprintf4* ⇀ (*Scala*) *CodeConst.Printf.sprintf4*
| **constant** *sprintf4* ⇀ (*SML*) *CodeConst.Printf.sprintf4*

consts *sprintf5* :: *ml-string* ⇒ *'α1* ⇒ *'α2* ⇒ *'α3* ⇒ *'α4* ⇒ *'α5* ⇒ *ml-string*
code-printing constant *sprintf5* ⇀ (*Haskell*) *CodeConst.Printf.sprintf5*
| **constant** *sprintf5* ⇀ (*OCaml*) *CodeConst.Printf.sprintf5*
| **constant** *sprintf5* ⇀ (*Scala*) *CodeConst.Printf.sprintf5*
| **constant** *sprintf5* ⇀ (*SML*) *CodeConst.Printf.sprintf5*

module String

consts *String-concat* :: *ml-string* ⇒ *ml-string list* ⇒ *ml-string*
code-printing constant *String-concat* ⇀ (*Haskell*) *CodeConst.String.concat*
| **constant** *String-concat* ⇀ (*OCaml*) *CodeConst.String.concat*
| **constant** *String-concat* ⇀ (*Scala*) *CodeConst.String.concat*
| **constant** *String-concat* ⇀ (*SML*) *CodeConst.String.concat*

module Sys

consts *Sys-is-directory2* :: *ml-string* ⇒ *bool ml-monad*
code-printing constant *Sys-is-directory2* ⇀ (*Haskell*) *CodeConst.Sys.isDirectory2*
| **constant** *Sys-is-directory2* ⇀ (*OCaml*) *CodeConst.Sys.isDirectory2*
| **constant** *Sys-is-directory2* ⇀ (*Scala*) *CodeConst.Sys.isDirectory2*
| **constant** *Sys-is-directory2* ⇀ (*SML*) *CodeConst.Sys.isDirectory2*

module To

consts *ToNat* :: (*nat* ⇒ *integer*) ⇒ *nat* ⇒ *ml-int*
code-printing constant *ToNat* ⇀ (*Haskell*) *CodeConst.To.nat*
| **constant** *ToNat* ⇀ (*OCaml*) *CodeConst.To.nat*

```
| constant ToNat ⇀ (Scala) CodeConst.To.nat
| constant ToNat ⇀ (SML) CodeConst.To.nat
```

4.1.4. Some Notations (I): Raw Translations

```
syntax -sprint0 :: - ⇒ ml-string (‹sprint0 (-)´›)
translations sprint0 x´ ⇌ CONST sprintf0 x

syntax -sprint1 :: - ⇒ - ⇒ ml-string (‹sprint1 (-)´›)
translations sprint1 x´ ⇌ CONST sprintf1 x

syntax -sprint2 :: - ⇒ - ⇒ ml-string (‹sprint2 (-)´›)
translations sprint2 x´ ⇌ CONST sprintf2 x

syntax -sprint3 :: - ⇒ - ⇒ ml-string (‹sprint3 (-)´›)
translations sprint3 x´ ⇌ CONST sprintf3 x

syntax -sprint4 :: - ⇒ - ⇒ ml-string (‹sprint4 (-)´›)
translations sprint4 x´ ⇌ CONST sprintf4 x

syntax -sprint5 :: - ⇒ - ⇒ ml-string (‹sprint5 (-)´›)
translations sprint5 x´ ⇌ CONST sprintf5 x
```

4.1.5. Some Notations (II): Polymorphic Cartouches

```
syntax -cartouche-string′ :: String.literal
translations -cartouche-string ⇌ -cartouche-string′

parse-translation ‹
  [( @{syntax-const -cartouche-string′}
   , parse-translation-cartouche
       @{binding cartouche-type′}
       (( fun_printf
        , ( Cartouche-Grammar.nil1
          , Cartouche-Grammar.cons1
          , let fun f c x = Syntax.const c $ x in
            fn (0, x) => x
             | (1, x) => f @{const-syntax sprintf1} x
             | (2, x) => f @{const-syntax sprintf2} x
             | (3, x) => f @{const-syntax sprintf3} x
             | (4, x) => f @{const-syntax sprintf4} x
             | (5, x) => f @{const-syntax sprintf5} x
            end))
        :: Cartouche-Grammar.default)
       (fn 37 — #"%" => (fn x => x + 1)
         | - => I)
       0)]
›
```

4.1.6. Generic Locale for Printing

locale *Print* =
  fixes *To-string* :: *string* ⇒ *ml-string*
  fixes *To-nat* :: *nat* ⇒ *ml-int*
begin
  declare[[*cartouche-type′* = fun_{printf}]]
end

As remark, printing functions (like *sprintf5...*) are currently weakly typed in Isabelle, we will continue the typing using the type system of target languages.

end

4.2. Instantiating the Printer for (Pure) Term

theory *Printer-Pure*
imports *Meta-Pure*
  Printer-init
begin

context *Print*
begin

fun *of-pure-term* **where** *of-pure-term l e* = (λ
  Const s - ⇒ *To-string s*
 | *Free s -* ⇒ *To-string s*
 | *App t1 t2* ⇒ ‹(%*s*) (%*s*)› (*of-pure-term l t1*) (*of-pure-term l t2*)
 | *Abs s - t* ⇒
  let s = *To-string s in*
  ‹(λ %*s*. %*s*)› *s* (*of-pure-term* (*s* # *l*) *t*)
 | *Bound n* ⇒ ‹%*s*› (*l* ! *nat-of-natural n*)) *e*

end

lemmas [*code*] =
 — def
 — fun
 Print.of-pure-term.simps

end

4.3. Instantiating the Printer for SML

theory *Printer-SML*
imports *Meta-SML*
  Printer-init
begin

context *Print*
begin

definition *of-semi--val-fun* = (λ *Sval* ⇒ ‹*val*› | *Sfun* ⇒ ‹*fun*›)

fun *of-semi--term′* **where** ‹*of-semi--term′ e* = (λ
  SML-string s ⇒ ‹%*s*› (*To-string* (*escape-sml s*))
 | *SML-rewrite val-fun e1 symb e2* ⇒ ‹%*s* %*s* %*s* %*s*› (*of-semi--val-fun val-fun*) (*of-semi--term′ e1*) (*To-string symb*) (*of-semi--term′ e2*)
 | *SML-basic l* ⇒ ‹%*s*› (*String-concat* ‹ › (*L.map To-string l*))
 | *SML-binop e1 s e2* ⇒ ‹%*s* %*s* %*s*› (*of-semi--term′ e1*) (*of-semi--term′* (*SML-basic* [*s*])) (*of-semi--term′ e2*)
 | *SML-annot e s* ⇒ ‹(%*s*:%*s*)› (*of-semi--term′ e*) (*To-string s*)
 | *SML-function l* ⇒ ‹(*fn* %*s*)› (*String-concat* ‹
  | › (*List.map* (λ (*s1*, *s2*) ⇒ ‹%*s* => %*s*› (*of-semi--term′ s1*) (*of-semi--term′ s2*)) *l*))
 | *SML-apply e l* ⇒ ‹(%*s* %*s*)› (*of-semi--term′ e*) (*String-concat* ‹ › (*List.map* (λ *e* ⇒ ‹(%*s*)› (*of-semi--term′ e*)) *l*))
 | *SML-paren p-left p-right e* ⇒ ‹%*s*%*s*%*s*› (*To-string p-left*) (*of-semi--term′ e*) (*To-string p-right*)
 | *SML-let-open s e* ⇒ ‹*let open* %*s in* %*s end*› (*To-string s*) (*of-semi--term′ e*)) *e*›

end

lemmas [*code*] =
 — def
 Print.of-semi--val-fun-def
 — fun
 Print.of-semi--term′.simps

end

4.4. Instantiating the Printer for Isabelle

theory *Printer-Isabelle*
imports *Meta-Isabelle*
  Printer-Pure
  Printer-SML
begin

context *Print*
begin

fun *of-semi--typ* **where** *of-semi--typ e* = (λ
  Typ-base s ⇒ *To-string s*
 | *Typ-apply name l* ⇒ ‹%*s* %*s*› (*let s* = *String-concat* ‹, › (*List.map of-semi--typ l*) *in*
    case l of [-] ⇒ *s* | - ⇒ ‹(%*s*)› *s*)
   (*of-semi--typ name*)
 | *Typ-apply-bin s ty1 ty2* ⇒ ‹%*s* %*s* %*s*› (*of-semi--typ ty1*) (*To-string s*) (*of-semi--typ ty2*)

```
  | Typ-apply-paren s1 s2 ty ⇒ ‹%s%s%s› (To-string s1) (of-semi--typ ty) (To-string s2)) e

definition of-datatype - = (λ Datatype n l ⇒
  ‹datatype %s = %s›
    (To-string n)
    (String-concat ‹
                   | ›
      (L.map
        (λ(n,l).
         ‹%s %s›
          (To-string n)
          (String-concat ‹ › (L.map (λx. ‹%s› (of-semi--typ x)) l))) l) ))

definition of-type-synonym - = (λ Type-synonym n v l ⇒
  ‹type-synonym %s = %s› (if v = [] then
                            To-string n
                          else
                            of-semi--typ (Typ-apply (Typ-base n) (L.map Typ-base v)))
                         (of-semi--typ l))

fun of-semi--term where of-semi--term e = (λ
    Term-rewrite e1 symb e2 ⇒ ‹%s %s %s› (of-semi--term e1) (To-string symb) (of-semi--term
e2)
  | Term-basic l ⇒ ‹%s› (String-concat ‹ › (L.map To-string l))
  | Term-annot e s ⇒ ‹(%s::%s)› (of-semi--term e) (of-semi--typ s)
  | Term-bind symb e1 e2 ⇒ ‹(%s%s. %s)› (To-string symb) (of-semi--term e1) (of-semi--term
e2)
  | Term-fun-case e-case l ⇒ ‹(%s %s)›
      (case e-case of None ⇒ ‹λ›
                  | Some e ⇒ ‹case %s of› (of-semi--term e))
      (String-concat ‹
    | › (List.map (λ (s1, s2) ⇒ ‹%s ⇒ %s› (of-semi--term s1) (of-semi--term s2)) l))
   | Term-apply e l ⇒ ‹%s %s› (of-semi--term e) (String-concat ‹ › (List.map (λ e ⇒ ‹%s›
(of-semi--term e)) l))
    | Term-paren p-left p-right e ⇒ ‹%s%s%s› (To-string p-left) (of-semi--term e) (To-string
p-right)
  | Term-if-then-else e-if e-then e-else ⇒ ‹if %s then %s else %s› (of-semi--term e-if) (of-semi--term
e-then) (of-semi--term e-else)
  | Term-term l pure ⇒ of-pure-term (L.map To-string l) pure) e

definition of-type-notation - = (λ Type-notation n e ⇒
  ‹type-notation %s (%s)› (To-string n) (To-string e))

definition of-instantiation - = (λ Instantiation n n-def expr ⇒
  let name = To-string n in
  ‹instantiation %s :: object
begin
  definition %s-%s-def : %s
  instance ..
```

```
end›
    name
    (To-string n-def)
    name
    (of-semi--term expr))

definition of-overloading - = (λ Overloading n-c e-c n e ⇒
  ‹overloading %s ≡ %s
begin
  definition %s : %s
end› (To-string n-c) (of-semi--term e-c) (To-string n) (of-semi--term e))

definition of-consts - = (λ Consts n ty symb ⇒
  ‹consts %s :: %s (%s %s)› (To-string n) (of-semi--typ ty) (To-string Consts-value) (To-string
symb))

definition of-definition - = (λ
    Definition e ⇒ ‹definition %s› (of-semi--term e)
  | Definition-where1 name (abbrev, prio) e ⇒ ‹definition %s ((1%s) %d)
  where %s› (To-string name) (of-semi--term abbrev) (To-nat prio) (of-semi--term e)
  | Definition-where2 name abbrev e ⇒ ‹definition %s (%s)
  where %s› (To-string name) (of-semi--term abbrev) (of-semi--term e))

definition (of-semi--thm-attribute-aux-gen :: String.literal × String.literal ⇒ - ⇒ - ⇒ -) m lacc
s =
 (let s-base = (λs lacc. ‹%s[%s]› (To-string s) (String-concat ‹, › (L.map (λ(s, x). ‹%s %s› s x)
lacc))) in
  s-base s (m # lacc))

definition of-semi--thm-attribute-aux-gen-where l =
 (‹where›, String-concat ‹ and › (L.map (λ(var, expr). ‹%s = %s›
                                                (To-string var)
                                                (of-semi--term expr)) l))

definition of-semi--thm-attribute-aux-gen-of l =
 (‹of›, String-concat ‹ › (L.map (λexpr. ‹%s› (of-semi--term expr)) l))



fun of-semi--thm-attribute-aux where of-semi--thm-attribute-aux lacc e =
  (λ Thm-thm s ⇒ To-string s
   | Thm-thms s ⇒ To-string s

  | Thm-THEN (Thm-thm s) e2 ⇒ of-semi--thm-attribute-aux-gen (‹THEN›, of-semi--thm-attribute-aux
[] e2) lacc s
  | Thm-THEN (Thm-thms s) e2 ⇒ of-semi--thm-attribute-aux-gen (‹THEN›, of-semi--thm-attribute-aux
[] e2) lacc s
   | Thm-THEN e1 e2 ⇒ of-semi--thm-attribute-aux ((‹THEN›, of-semi--thm-attribute-aux []
e2) # lacc) e1
```

```
  | Thm-simplified (Thm-thm s) e2 ⇒ of-semi--thm-attribute-aux-gen (‹simplified›, of-semi--thm-attribute-aux
[] e2) lacc s
  | Thm-simplified (Thm-thms s) e2 ⇒ of-semi--thm-attribute-aux-gen (‹simplified›, of-semi--thm-attribute-aux
[] e2) lacc s
  | Thm-simplified e1 e2 ⇒ of-semi--thm-attribute-aux ((‹simplified›, of-semi--thm-attribute-aux
[] e2) # lacc) e1

  | Thm-symmetric (Thm-thm s) ⇒ of-semi--thm-attribute-aux-gen (‹symmetric›, ‹›) lacc s
  | Thm-symmetric (Thm-thms s) ⇒ of-semi--thm-attribute-aux-gen (‹symmetric›, ‹›) lacc s
  | Thm-symmetric e1 ⇒ of-semi--thm-attribute-aux ((‹symmetric›, ‹›) # lacc) e1

  | Thm-where (Thm-thm s) l ⇒ of-semi--thm-attribute-aux-gen (of-semi--thm-attribute-aux-gen-where
l) lacc s
  | Thm-where (Thm-thms s) l ⇒ of-semi--thm-attribute-aux-gen (of-semi--thm-attribute-aux-gen-where
l) lacc s
   | Thm-where e1 l ⇒ of-semi--thm-attribute-aux (of-semi--thm-attribute-aux-gen-where l #
lacc) e1

  | Thm-of (Thm-thm s) l ⇒ of-semi--thm-attribute-aux-gen (of-semi--thm-attribute-aux-gen-of
l) lacc s
  | Thm-of (Thm-thms s) l ⇒ of-semi--thm-attribute-aux-gen (of-semi--thm-attribute-aux-gen-of
l) lacc s
  | Thm-of e1 l ⇒ of-semi--thm-attribute-aux (of-semi--thm-attribute-aux-gen-of l # lacc) e1

  | Thm-OF (Thm-thm s) e2 ⇒ of-semi--thm-attribute-aux-gen (‹OF›, of-semi--thm-attribute-aux
[] e2) lacc s
  | Thm-OF (Thm-thms s) e2 ⇒ of-semi--thm-attribute-aux-gen (‹OF›, of-semi--thm-attribute-aux
[] e2) lacc s
   | Thm-OF e1 e2 ⇒ of-semi--thm-attribute-aux ((‹OF›, of-semi--thm-attribute-aux [] e2) #
lacc) e1) e

definition of-semi--thm-attribute = of-semi--thm-attribute-aux []

definition of-semi--thm = (λ Thms-single thy ⇒ of-semi--thm-attribute thy
                      | Thms-mult thy ⇒ of-semi--thm-attribute thy)

definition of-semi--thm-attribute-l l = String-concat ‹
                          › (L.map of-semi--thm-attribute l)
definition of-semi--thm-attribute-l1 l = String-concat ‹ › (L.map of-semi--thm-attribute l)

definition of-semi--thm-l l = String-concat ‹ › (L.map of-semi--thm l)

definition of-lemmas - = (λ Lemmas-simp-thm simp s l ⇒
  ‹lemmas%s%s = %s›
    (if String.is-empty s then ‹› else ‹ %s› (To-string s))
    (if simp then ‹[simp,code-unfold]› else ‹›)
    (of-semi--thm-attribute-l l)
                      | Lemmas-simp-thms s l ⇒
```

```
  ‹lemmas%s [simp,code-unfold] = %s›
    (if String.is-empty s then ‹› else ‹ %s› (To-string s))
    (String-concat ‹
                          › (L.map To-string l)))

definition (of-semi--attrib-genA :: (semi--thm list ⇒ String.literal)
   ⇒ String.literal ⇒ semi--thm list ⇒ String.literal) f attr l = — error reflection: to be merged
 (if l = [] then
    ‹›
  else
    ‹ %s: %s› attr (f l))

definition (of-semi--attrib-genB :: (string list ⇒ String.literal)
   ⇒ String.literal ⇒ string list ⇒ String.literal) f attr l = — error reflection: to be merged
 (if l = [] then
    ‹›
  else
    ‹ %s: %s› attr (f l))

definition of-semi--attrib = of-semi--attrib-genA of-semi--thm-l
definition of-semi--attrib1 = of-semi--attrib-genB (λl. String-concat ‹ › (L.map To-string l))

definition of-semi--method-simp (s :: — polymorphism weakening needed by code-reflect
                                    String.literal) =
 (λ Method-simp-only l ⇒ ‹%s only: %s› s (of-semi--thm-l l)
  | Method-simp-add-del-split l1 l2 [] ⇒ ‹%s%s%s›
      s
      (of-semi--attrib ‹add› l1)
      (of-semi--attrib ‹del› l2)
  | Method-simp-add-del-split l1 l2 l3 ⇒ ‹%s%s%s%s›
      s
      (of-semi--attrib ‹add› l1)
      (of-semi--attrib ‹del› l2)
      (of-semi--attrib ‹split› l3))

fun of-semi--method where of-semi--method expr = (λ
    Method-rule o-s ⇒ ‹rule%s› (case o-s of None ⇒ ‹›
                                         | Some s ⇒ ‹ %s› (of-semi--thm-attribute s))
  | Method-drule s ⇒ ‹drule %s› (of-semi--thm-attribute s)
  | Method-erule s ⇒ ‹erule %s› (of-semi--thm-attribute s)
  | Method-intro l ⇒ ‹intro %s› (of-semi--thm-attribute-l1 l)
  | Method-elim s ⇒ ‹elim %s› (of-semi--thm-attribute s)
  | Method-subst asm l s =>
      let s-asm = if asm then ‹(asm) › else ‹› in
      if L.map String.meta-of-logic l = [STR ''0''] then
        ‹subst %s%s› s-asm (of-semi--thm-attribute s)
      else
        ‹subst %s(%s) %s› s-asm (String-concat ‹ › (L.map To-string l)) (of-semi--thm-attribute
s)
```

```
  | Method-insert l => ‹insert %s› (of-semi--thm-l l)
  | Method-plus t ⇒ ‹(%s)+› (String-concat ‹, › (List.map of-semi--method t))
  | Method-option t ⇒ ‹(%s)?› (String-concat ‹, › (List.map of-semi--method t))
  | Method-or t ⇒ ‹(%s)› (String-concat ‹ | › (List.map of-semi--method t))
  | Method-one s ⇒ of-semi--method-simp ‹simp› s
  | Method-all s ⇒ of-semi--method-simp ‹simp-all› s
  | Method-auto-simp-add-split l-simp l-split ⇒ ‹auto%s%s›
     (of-semi--attrib ‹simp› l-simp)
     (of-semi--attrib1 ‹split› l-split)
  | Method-rename-tac l ⇒ ‹rename-tac %s› (String-concat ‹ › (L.map To-string l))
  | Method-case-tac e ⇒ ‹case-tac %s› (of-semi--term e)
  | Method-blast None ⇒ ‹blast›
  | Method-blast (Some n) ⇒ ‹blast %d› (To-nat n)
  | Method-clarify ⇒ ‹clarify›
  | Method-metis l-opt l ⇒ ‹metis %s%s› (if l-opt = [] then ‹›
                                    else
                                         ‹(%s) › (String-concat ‹, › (L.map To-string l-opt)))
(of-semi--thm-attribute-l1 l)) expr

definition of-semi--command-final = (λ Command-done ⇒ ‹done›
                                  | Command-by l-apply ⇒ ‹by(%s)› (String-concat ‹, › (L.map
of-semi--method l-apply))
                                | Command-sorry ⇒ ‹sorry›)

definition of-semi--command-state = (
  λ Command-apply-end [] ⇒ ‹›
  | Command-apply-end l-apply ⇒ ‹  apply-end(%s)
› (String-concat ‹, › (L.map of-semi--method l-apply)))

definition ‹of-semi--command-proof = (
  let thesis = ‹?thesis›
    ; scope-thesis-gen = λproof show when. ‹  proof − %s show %s%s
› proof
  show
  (if when = [] then
     ‹›
   else
     ‹ when %s› (String-concat ‹ › (L.map (λt. ‹%s› (of-semi--term t)) when)))
    ; scope-thesis = λs. scope-thesis-gen s thesis [] in
  λ Command-apply [] ⇒ ‹›
  | Command-apply l-apply ⇒ ‹  apply(%s)
› (String-concat ‹, › (L.map of-semi--method l-apply))
  | Command-using l ⇒ ‹  using %s
› (of-semi--thm-l l)
  | Command-unfolding l ⇒ ‹  unfolding %s
› (of-semi--thm-l l)
 | Command-let e-name e-body ⇒ scope-thesis (‹let %s = %s› (of-semi--term e-name) (of-semi--term
e-body))
  | Command-have n b e e-last ⇒ scope-thesis (‹have %s%s: %s %s› (To-string n) (if b then
```

```
‹[simp]› else ‹›) (of-semi--term e) (of-semi--command-final e-last))
  | Command-fix-let l l-let o-show - ⇒
      scope-thesis-gen
        (‹fix %s%s› (String-concat ‹ › (L.map To-string l))
                    (String-concat ‹
›                                  (L.map
                                    (λ(e-name, e-body).
                                       ‹          let %s = %s› (of-semi--term e-name) (of-semi--term
e-body))
                                    l-let)))
        (case o-show of None ⇒ thesis
                  | Some (l-show, -) ⇒ ‹%s› (String-concat ‹ ⟹ › (L.map of-semi--term l-show)))
        (case o-show of None ⇒ [] | Some (-, l-when) ⇒ l-when))›

definition of-lemma - =
 (λ Lemma n l-spec l-apply tactic-last ⇒
    ‹lemma %s : %s
%s%s›
      (To-string n)
      (String-concat ‹ ⟹ › (L.map of-semi--term l-spec))
      (String-concat ‹› (L.map (λ [] ⇒ ‹› | l-apply ⇒ ‹  apply(%s)
›                                                      (String-concat ‹, › (L.map of-semi--method l-apply)))
                               l-apply))
      (of-semi--command-final tactic-last)
  | Lemma-assumes n l-spec concl l-apply tactic-last ⇒
    ‹lemma %s : %s
%s%s %s›
      (To-string n)
      (String-concat ‹› (L.map (λ(n, b, e).
         ‹
assumes %s%s›
        (let (n, b) = if b then (‹%s[simp]› (To-string n), False) else (To-string n, String.is-empty
n) in
           if b then ‹› else ‹%s: › n)
          (of-semi--term e)) l-spec
      @@@@
      [‹
shows %s› (of-semi--term concl)]))
      (String-concat ‹› (L.map of-semi--command-proof l-apply))
      (of-semi--command-final tactic-last)
      (String-concat ‹ ›
        (L.map
          (λl-apply-e.
            ‹%sqed›
              (if l-apply-e = [] then
                 ‹›
               else
                 ‹
```

```
%s ›
                (String-concat ‹› (L.map of-semi--command-state l-apply-e))))
        (List.map-filter
          (λ Command-let - - ⇒ Some [] | Command-have - - - - ⇒ Some [] | Command-fix-let
- - - l ⇒ Some l | - ⇒ None)
          (rev l-apply)))))


definition of-axiomatization - = (λ Axiomatization n e ⇒ ‹axiomatization where %s:
%s› (To-string n) (of-semi--term e))

definition of-section - = (λ Section n section-title ⇒
   ‹%s ‹%s››
     (‹%ssection› (if n = 0 then ‹›
                   else if n = 1 then ‹sub›
                   else ‹subsub›))
     (To-string section-title))

definition of-text - = (λ Text s ⇒ ‹text ‹%s›› (To-string s))

definition of-ML - = (λ SML e ⇒ ‹ML ‹%s›› (of-semi--term′ e))

definition of-setup - = (λ Setup e ⇒ ‹setup ‹%s›› (of-semi--term′ e))

definition of-thm - = (λ Thm thm ⇒ ‹thm %s› (of-semi--thm-attribute-l1 thm))

definition ‹of-interpretation - = (λ Interpretation n loc-n loc-param tac ⇒
  ‹interpretation %s: %s%s
%s› (To-string n)
   (To-string loc-n)
   (String-concat ‹› (L.map (λs. ‹ %s› (of-semi--term s)) loc-param))
   (of-semi--command-final tac))›

definition of-semi--theory env =
 (λ Theory-datatype dataty ⇒ of-datatype env dataty
  | Theory-type-synonym ty-synonym ⇒ of-type-synonym env ty-synonym
  | Theory-type-notation ty-notation ⇒ of-type-notation env ty-notation
  | Theory-instantiation instantiation-class ⇒ of-instantiation env instantiation-class
  | Theory-overloading overloading ⇒ of-overloading env overloading
  | Theory-consts consts-class ⇒ of-consts env consts-class
  | Theory-definition definition-hol ⇒ of-definition env definition-hol
  | Theory-lemmas lemmas-simp ⇒ of-lemmas env lemmas-simp
  | Theory-lemma lemma-by ⇒ of-lemma env lemma-by
  | Theory-axiomatization axiom ⇒ of-axiomatization env axiom
  | Theory-section section-title ⇒ of-section env section-title
  | Theory-text text ⇒ of-text env text
  | Theory-ML ml ⇒ of-ML env ml
  | Theory-setup setup ⇒ of-setup env setup
  | Theory-thm thm ⇒ of-thm env thm
```

```
  | Theory-interpretation thm ⇒ of-interpretation env thm)

definition String-concat-map s f l = String-concat s (L.map f l)

definition ‹of-semi--theories env =
 (λ Theories-one t ⇒ of-semi--theory env t
  | Theories-locale data l ⇒
      ‹locale %s =
%s
begin
%s
end›   (To-string (HolThyLocale-name data))
        (String-concat-map
           ‹
›
           (λ (l-fix, o-assum).
               ‹%s%s› (String-concat-map ‹
› (λ(e, ty). ‹fixes %s :: %s› (of-semi--term e) (of-semi--typ ty)) l-fix)
                                 (case o-assum of None ⇒ ‹›
                                                | Some (name, e) ⇒ ‹
assumes %s: %s› (To-string name) (of-semi--term e)))
           (HolThyLocale-header data))
        (String-concat-map ‹

› (String-concat-map ‹

› (of-semi--theory env)) l))›

end

lemmas [code] =
  — def
  Print.of-datatype-def
  Print.of-type-synonym-def
  Print.of-type-notation-def
  Print.of-instantiation-def
  Print.of-overloading-def
  Print.of-consts-def
  Print.of-definition-def
  Print.of-semi--thm-attribute-aux-gen-def
  Print.of-semi--thm-attribute-aux-gen-where-def
  Print.of-semi--thm-attribute-aux-gen-of-def
  Print.of-semi--thm-attribute-def
  Print.of-semi--thm-def
  Print.of-semi--thm-attribute-l-def
  Print.of-semi--thm-attribute-l1-def
  Print.of-semi--thm-l-def
  Print.of-lemmas-def
  Print.of-semi--attrib-genA-def
```

Print.of-semi--attrib-genB-def
Print.of-semi--attrib-def
Print.of-semi--attrib1-def
Print.of-semi--method-simp-def
Print.of-semi--command-final-def
Print.of-semi--command-state-def
Print.of-semi--command-proof-def
Print.of-lemma-def
Print.of-axiomatization-def
Print.of-section-def
Print.of-text-def
Print.of-ML-def
Print.of-setup-def
Print.of-thm-def
Print.of-interpretation-def
Print.of-semi--theory-def
Print.String-concat-map-def
Print.of-semi--theories-def

— fun
Print.of-semi--typ.simps
Print.of-semi--term.simps
Print.of-semi--thm-attribute-aux.simps
Print.of-semi--method.simps

end

5. Main

We present two solutions for obtaining an Isabelle file.

5.1. Static Meta Embedding with Exportation

theory *Generator-static*
imports *Printer ../../Antiquote-Setup*
begin

In the "static" solution: the user manually generates the Isabelle file after writing by hand a Toy input to translate. The input is not written with the syntax of the Toy Language, but with raw Isabelle constructors.

5.1.1. Giving an Input to Translate

definition *Design* =
(*let n* = λ*n1 n2*. *ToyTyObj* (*ToyTyCore-pre n1*) (*case n2 of None* ⇒ [] | *Some n2* ⇒ [[*ToyTyCore-pre n2*]])
; *mk* = λ*n l*. *toy-class-raw.make n l* [] *False in*
[*mk* (*n* ‹*Galaxy*› *None*) [(‹*sound*›, *ToyTy-raw* ‹*unit*›), (‹*moving*›, *ToyTy-raw* ‹*bool*›)]
, *mk* (*n* ‹*Planet*› (*Some* ‹*Galaxy*›)) [(‹*weight*›, *ToyTy-raw* ‹*nat*›)]
, *mk* (*n* ‹*Person*› (*Some* ‹*Planet*›)) [(‹*salary*›, *ToyTy-raw* ‹*int*›)]])

Since we are in a Isabelle session, at this time, it becomes possible to inspect with the command **value** the result of the translations applied with *Design*. A suitable environment should nevertheless be provided, one can typically experiment this by copying-pasting the following environment initialized in the above *main*:

definition *main* =
(*let n* = λ*n1*. *ToyTyObj* (*ToyTyCore-pre n1*) []
; *ToyMult* = λ*m r*. *toy-multiplicity.make* [*m*] *r* [*Set*] *in*
write-file
(*compiler-env-config.extend*
(*compiler-env-config-empty True None* (*oidInit* (*Oid 0*)) *Gen-only-design* (*None*, *False*)
(‖ *D-output-disable-thy* := *False*
, *D-output-header-thy* := *Some* (‹*Design-generated*›
,[‹*../Toy-Library*›]
,‹*../embedding/Generator-dynamic-sequential*›) ‖))
(*L.map* (*META-class-raw Floor1*) *Design*
@@@@ [*META-association* (*toy-association.make*
ToyAssTy-association

(*ToyAssRel* [(*n* ‹*Person*›, *ToyMult* (*Mult-star*, *None*) *None*)
, (*n* ‹*Person*›, *ToyMult* (*Mult-nat 0*, *Some* (*Mult-nat 1*))
(*Some* ‹*boss*›))])
, *META-flush-all ToyFlushAll*]
, *None*)))

5.1.2. Statically Executing the Exportation

```
apply_code_printing ()
export_code main
(* in Haskell *)
(* in OCaml module_name M *)
(* in Scala module_name M *)
(* in SML   module_name M *)
```

After the exportation and executing the exported, we obtain an Isabelle `.thy` file containing the generated code associated to the above input.

end

5.2. Dynamic Meta Embedding with Reflection

theory *Generator-dynamic-sequential*
imports *Printer*
*../../isabelle-home/src/HOL/Isabelle-Main2***begin**

In the "dynamic" solution: the exportation is automatically handled inside Isabelle/jEdit. Inputs are provided using the syntax of the Toy Language, and in output we basically have two options:

- The first is to generate an Isabelle file for inspection or debugging. The generated file can interactively be loaded in Isabelle/jEdit, or saved to the hard disk. This mode is called the "deep exportation" mode or shortly the "deep" mode. The aim is to maximally automate the process one is manually performing in `Generator_static.thy`.
- On the other hand, it is also possible to directly execute in Isabelle/jEdit the generated file from the random access memory. This mode corresponds to the "shallow reflection" mode or shortly "shallow" mode.

In both modes, the reflection is necessary since the main part used by both was defined at Isabelle side. As a consequence, experimentations in "deep" and "shallow" are performed without leaving the editing session, in the same as the one the meta-compiler is actually running.

apply-code-printing-reflect ‹
val stdout-file = Unsynchronized.ref
›

This variable is not used in this theory (only in `Generator_static.thy`), but needed for well typechecking the reflected SML code.

code-reflect′ open *META*
functions
fold-thy-deep fold-thy-shallow

write-file

compiler-env-config-reset-all
compiler-env-config-update
oidInit
D-output-header-thy-update
map2-ctxt-term
check-export-code

isabelle-apply isabelle-of-compiler-env-config

5.2.1. Interface Between the Reflected and the Native

ML‹
val To-string0 = META.meta-of-logic
›

ML‹
structure From = struct
val string = META.SS-base o META.ST
val binding = string o Binding.name-of
(∗fun term ctxt s = string (Protocol-Message.clean-output (Syntax.string-of-term ctxt s))∗)
val internal-oid = META.Oid o Code-Numeral.natural-of-integer
val option = Option.map
val list = List.map
fun pair f1 f2 (x, y) = (f1 x, f2 y)
fun pair3 f1 f2 f3 (x, y, z) = (f1 x, f2 y, f3 z)

structure Pure = struct
val indexname = pair string Code-Numeral.natural-of-integer
val class = string
val sort = list class
fun typ e = (fn
Type (s, l) => (META.Type o pair string (list typ)) (s, l)
| TFree (s, s0) => (META.TFree o pair string sort) (s, s0)
| TVar (i, s0) => (META.TVar o pair indexname sort) (i, s0)
) e
fun term e = (fn
Const (s, t) => (META.Const o pair string typ) (s, t)
| Free (s, t) => (META.Free o pair string typ) (s, t)
| Var (i, t) => (META.Var o pair indexname typ) (i, t)
| Bound i => (META.Bound o Code-Numeral.natural-of-integer) i

```
  | Abs (s, ty, t) => (META.Abs o pair3 string typ term) (s, ty, t)
  | op $ (term1, term2) => (META.App o pair term term) (term1, term2)
 ) e
end

fun toy-ctxt-term thy expr =
  META.T-pure (Pure.term (Syntax.read-term (Proof-Context.init-global thy) expr))
end
›

ML‹fun List-mapi f = META.mapi (f o Code-Numeral.integer-of-natural)›

ML‹
structure Ty' = struct
fun check l-oid l =
 let val Mp = META.map-prod
     val Me = String.explode
     val Mi = String.implode
     val Ml = map in
 META.check-export-code
   (writeln o Mi)
   (warning o Mi)
   (fn s => writeln (Markup.markup (Markup.bad ()) (Mi s)))
   (error o To-string0)
   (Ml (Mp I Me) l-oid)
   ((META.SS-base o META.ST) l)
 end
end
›
```

5.2.2. Binding of the Reflected API to the Native API

```
ML‹
structure META-overload = struct
 val of-semi--typ = META.of-semi-typ To-string0
 val of-semi--term = META.of-semi-terma To-string0
 val of-semi--term' = META.of-semi-term To-string0
 val fold = fold
end
›

ML‹
structure Bind-Isabelle = struct
fun To-binding s = Binding.make (s, Position.none)
val To-sbinding = To-binding o To-string0

fun semi--method-simp g f = Method.Basic (fn ctxt => SIMPLE-METHOD (g (asm-full-simp-tac
(f ctxt))))
val semi--method-simp-one = semi--method-simp (fn f => f 1)
```

```
val semi--method-simp-all = semi--method-simp (CHANGED-PROP o PARALLEL-ALLGOALS)

datatype semi--thm' = Thms-single' of thm
                    | Thms-mult' of thm list

fun semi--thm-attribute ctxt = let open META open META-overload val S = fn Thms-single' t
=> t
                                                    val M = fn Thms-mult' t => t in
 fn Thm-thm s => Thms-single' (Proof-Context.get-thm ctxt (To-string0 s))
  | Thm-thms s => Thms-mult' (Proof-Context.get-thms ctxt (To-string0 s))
  | Thm-THEN (e1, e2) =>
      (case (semi--thm-attribute ctxt e1, semi--thm-attribute ctxt e2) of
         (Thms-single' e1, Thms-single' e2) => Thms-single' (e1 RSN (1, e2))
       | (Thms-mult' e1, Thms-mult' e2) => Thms-mult' (e1 RLN (1, e2)))
  | Thm-simplified (e1, e2) =>
       Thms-single' (asm-full-simplify (clear-simpset ctxt addsimps [S (semi--thm-attribute ctxt
e2)])
                                       (S (semi--thm-attribute ctxt e1)))
  | Thm-OF (e1, e2) =>
      Thms-single' ([S (semi--thm-attribute ctxt e2)] MRS (S (semi--thm-attribute ctxt e1)))
  | Thm-where (nth, l) =>
      Thms-single' (Rule-Insts.where-rule
                    ctxt
                    (List.map (fn (var, expr) =>
                                  (((To-string0 var, 0), Position.none), of-semi--term expr)) l)
                    []
                    (S (semi--thm-attribute ctxt nth)))
  | Thm-symmetric e1 =>
      let val e2 = S (semi--thm-attribute ctxt (Thm-thm (From.string sym))) in
        case semi--thm-attribute ctxt e1 of
          Thms-single' e1 => Thms-single' (e1 RSN (1, e2))
        | Thms-mult' e1 => Thms-mult' (e1 RLN (1, [e2]))
      end
  | Thm-of (nth, l) =>
      Thms-single' (Rule-Insts.of-rule
                    ctxt
                    (List.map (SOME o of-semi--term) l, [])
                    []
                    (S (semi--thm-attribute ctxt nth)))
end

fun semi--thm-attribute-single ctxt s = case (semi--thm-attribute ctxt s) of Thms-single' t => t

fun semi--thm-mult ctxt =
  let fun f thy = case (semi--thm-attribute ctxt thy) of Thms-mult' t => t
                                                       | Thms-single' t => [t] in
  fn META.Thms-single thy => f thy
   | META.Thms-mult thy => f thy
  end
```

```
fun semi--thm-mult-l ctxt l = List.concat (map (semi--thm-mult ctxt) l)

fun semi--method-simp-only l ctxt = clear-simpset ctxt addsimps (semi--thm-mult-l ctxt l)
fun semi--method-simp-add-del-split (l-add, l-del, l-split) ctxt =
  fold Splitter.add-split (semi--thm-mult-l ctxt l-split)
                          (ctxt addsimps (semi--thm-mult-l ctxt l-add)
                                delsimps (semi--thm-mult-l ctxt l-del))

fun semi--method expr = let open META open Method open META-overload in case expr of
    Method-rule o-s => Basic (fn ctxt =>
      METHOD (HEADGOAL o Classical.rule-tac
                          ctxt
                          (case o-s of NONE => []
                                     | SOME s => [semi--thm-attribute-single ctxt s])))
  | Method-drule s => Basic (fn ctxt => drule ctxt 0 [semi--thm-attribute-single ctxt s])
  | Method-erule s => Basic (fn ctxt => erule ctxt 0 [semi--thm-attribute-single ctxt s])
  | Method-elim s => Basic (fn ctxt => elim ctxt [semi--thm-attribute-single ctxt s])
  | Method-intro l => Basic (fn ctxt => intro ctxt (map (semi--thm-attribute-single ctxt) l))
  | Method-subst (asm, l, s) => Basic (fn ctxt =>
      SIMPLE-METHOD' ((if asm then EqSubst.eqsubst-asm-tac else EqSubst.eqsubst-tac)
                      ctxt
                      (map (fn s => case Int.fromString (To-string0 s) of
                                      SOME i => i) l)
                      [semi--thm-attribute-single ctxt s]))
  | Method-insert l => Basic (fn ctxt => insert (semi--thm-mult-l ctxt l))
  | Method-plus t => Combinator ( no-combinator-info
                                , Repeat1
                                , [Combinator (no-combinator-info, Then, List.map semi--method t)])
  | Method-option t => Combinator ( no-combinator-info
                                  , Try
                                  , [Combinator (no-combinator-info, Then, List.map semi--method t)])
  | Method-or t => Combinator (no-combinator-info, Orelse, List.map semi--method t)
  | Method-one (Method-simp-only l) => semi--method-simp-one (semi--method-simp-only l)
  | Method-one (Method-simp-add-del-split l) => semi--method-simp-one (semi--method-simp-add-del-split
l)
  | Method-all (Method-simp-only l) => semi--method-simp-all (semi--method-simp-only l)
  | Method-all (Method-simp-add-del-split l) => semi--method-simp-all (semi--method-simp-add-del-split
l)
  | Method-auto-simp-add-split (l-simp, l-split) =>
      Basic (fn ctxt => SIMPLE-METHOD (auto-tac (fold (fn (f, l) => fold f l)
            [(Simplifier.add-simp, semi--thm-mult-l ctxt l-simp)
            ,(Splitter.add-split, List.map (Proof-Context.get-thm ctxt o To-string0) l-split)]
            ctxt)))
  | Method-rename-tac l => Basic (K (SIMPLE-METHOD' (Tactic.rename-tac (List.map To-string0
l))))
  | Method-case-tac e =>
       Basic (fn ctxt => SIMPLE-METHOD' (Induct-Tacs.case-tac ctxt (of-semi--term e) []
NONE))
```

```
  | Method-blast n =>
      Basic (case n of NONE => SIMPLE-METHOD' o blast-tac
                     | SOME lim => fn ctxt => SIMPLE-METHOD' (depth-tac ctxt
(Code-Numeral.integer-of-natural lim)))
  | Method-clarify => Basic (fn ctxt => (SIMPLE-METHOD' (fn i => CHANGED-PROP
(clarify-tac ctxt i))))
  | Method-metis (l-opt, l) =>
      Basic (fn ctxt => (METHOD oo Metis-Tactic.metis-method)
                        ( (if l-opt = [] then NONE else SOME (map To-string0 l-opt), NONE)
                        , map (semi--thm-attribute-single ctxt) l)
                        ctxt)
end

fun then-tactic l = let open Method in
  (Combinator (no-combinator-info, Then, map semi--method l), (Position.none, Position.none))
end

fun local-terminal-proof o-by = let open META in case o-by of
    Command-done => Proof.local-done-proof
  | Command-sorry => Proof.local-skip-proof true
  | Command-by l-apply => Proof.local-terminal-proof (then-tactic l-apply, NONE)
end

fun global-terminal-proof o-by = let open META in case o-by of
    Command-done => Proof.global-done-proof
  | Command-sorry => Proof.global-skip-proof true
  | Command-by l-apply => Proof.global-terminal-proof (then-tactic l-apply, NONE)
end

fun proof-show-gen f (thes, thes-when) st = st
  |> Proof.proof
       (SOME ( Method.Source [Token.make-string (-, Position.none)]
             , (Position.none, Position.none)))
  |> Seq.the-result
  |> f
  |> Proof.show-cmd
       (thes-when = [])
       NONE
       (K I)
       []
       (if thes-when = [] then [] else [(Binding.empty-atts, map (fn t => (t, [])) thes-when)])
       [(Binding.empty-atts, [(thes, [])])]
       true
  |> snd

val semi--command-state = let open META-overload in
  fn META.Command-apply-end l => (fn st => st |> Proof.apply-end (then-tactic l)
                                              |> Seq.the-result )
end
```

```
val semi--command-proof = let open META-overload
                      val thesis = ?thesis
                      fun proof-show f = proof-show-gen f (thesis, []) in
  fn META.Command-apply l => (fn st => st |> Proof.apply (then-tactic l)
                                        |> Seq.the-result )
   | META.Command-using l => (fn st =>
       let val ctxt = Proof.context-of st in
       Proof.using [map (fn s => ([ s], [])) (semi--thm-mult-l ctxt l)] st
       end)
   | META.Command-unfolding l => (fn st =>
       let val ctxt = Proof.context-of st in
       Proof.unfolding [map (fn s => ([s], [])) (semi--thm-mult-l ctxt l)] st
       end)
   | META.Command-let (e1, e2) =>
       proof-show (Proof.let-bind-cmd [([of-semi--term e1], of-semi--term e2)])
   | META.Command-have (n, b, e, e-pr) => proof-show (fn st => st
       |> Proof.have-cmd true NONE (K I) [] []
                      [( (To-sbinding n, if b then [[Token.make-string (simp, Position.none)]] else
[])
                       , [(of-semi--term e, [])])]
                      true
       |> snd
       |> local-terminal-proof e-pr)
   | META.Command-fix-let (l, l-let, o-exp, -) =>
       proof-show-gen ( fold (fn (e1, e2) =>
                                Proof.let-bind-cmd [([of-semi--term e1], of-semi--term e2)])
                            l-let
                    o Proof.fix-cmd (List.map (fn i => (To-sbinding i, NONE, NoSyn)) l))
                    ( case o-exp of NONE => thesis | SOME (l-spec, -) =>
                      (String.concatWith ( ⟹ )
                                         (List.map of-semi--term l-spec))
                    , case o-exp of NONE => [] | SOME (-, l-when) => List.map of-semi--term
l-when)
end

fun semi--theory in-theory in-local = let open META open META-overload in (*let val f = *)fn
  Theory-datatype (Datatype (n, l)) => in-local
   (BNF-FP-Def-Sugar.co-datatype-cmd
      BNF-Util.Least-FP
      BNF-LFP.construct-lfp
      (Ctr-Sugar.default-ctr-options-cmd,
       [( ( ( (([], To-sbinding n), NoSyn)
            , List.map (fn (n, l) => ( ( (To-binding , To-sbinding n)
                                       , List.map (fn s => (To-binding , of-semi--typ s)) l)
                                     , NoSyn)) l)
          , (To-binding , To-binding , To-binding ))
        , [])]))
| Theory-type-synonym (Type-synonym (n, v, l)) => in-theory
```

```
  (fn thy =>
    let val s-bind = To-sbinding n in
    (snd o Typedecl.abbrev-global
           (s-bind, map To-string0 v, NoSyn)
           (Isabelle-Typedecl.abbrev-cmd0 (SOME s-bind) thy (of-semi--typ l))) thy
    end)
| Theory-type-notation (Type-notation (n, e)) => in-local
  (Local-Theory.type-notation-cmd true (, true) [(To-string0 n, Mixfix (Input.string (To-string0
e), [], 1000, Position.no-range))])
| Theory-instantiation (Instantiation (n, n-def, expr)) => in-theory
  (fn thy =>
    let val name = To-string0 n
        val tycos =
          [ let val Term.Type (s, -) = (Isabelle-Typedecl.abbrev-cmd0 NONE thy name) in s end
] in
    thy
    |> Class.instantiation (tycos, [], Syntax.read-sort (Proof-Context.init-global thy) object)
    |> fold-map (fn - => fn thy =>
       let val ((-, (-, ty)), thy) = Specification.definition-cmd
                                     NONE [] []
                                     ((To-binding (To-string0 n-def ^ - ^ name ^ -def), [])
                                       , of-semi--term expr) false thy in
        (ty, thy)
       end) tycos
    |-> Class.prove-instantiation-exit-result (map o Morphism.thm) (fn ctxt => fn thms =>
         Class.intro-classes-tac ctxt [] THEN ALLGOALS (Proof-Context.fact-tac ctxt thms))
    |-> K I
    end)
| Theory-overloading (Overloading (n-c, e-c, n, e)) => in-theory
  (fn thy => thy
    |> Overloading.overloading-cmd [(To-string0 n-c, of-semi--term e-c, true)]
    |> snd o Specification.definition-cmd NONE [] [] ((To-sbinding n, []), of-semi--term e) false
    |> Local-Theory.exit-global)
| Theory-consts (Consts (n, ty, symb)) => in-theory
  (Sign.add-consts-cmd [( To-sbinding n
                        , of-semi--typ ty
                        , Mixfix (Input.string ((-) ^ To-string0 symb), [], 1000, Position.no-range))])
| Theory-definition def => in-local
    let val (def, e) = case def of
        Definition e => (NONE, e)
      | Definition-where1 (name, (abbrev, prio), e) =>
          (SOME ( To-sbinding name
                , NONE
          , Mixfix (Input.string ((1 ^ of-semi--term abbrev ^)), [], Code-Numeral.integer-of-natural
prio, Position.no-range)), e)
      | Definition-where2 (name, abbrev, e) =>
          (SOME ( To-sbinding name
                , NONE
                , Mixfix (Input.string (( ^ of-semi--term abbrev ^)), [], 1000, Position.no-range)),
```

```
e) in
    (snd o Specification.definition-cmd def [] [] (Binding.empty-atts, of-semi--term e) false)
    end
| Theory-lemmas (Lemmas-simp-thm (simp, s, l)) => in-local
   (fn lthy => (snd o Specification.theorems Thm.theoremK
        [((To-sbinding s, List.map (fn s => Attrib.check-src lthy [Token.make-string (s, Posi-
tion.none)])
                        (if simp then [simp, code-unfold] else [])),
       List.map (fn x => ([semi--thm-attribute-single lthy x], [])) l)]
      []
      false) lthy)
| Theory-lemmas (Lemmas-simp-thms (s, l)) => in-local
   (fn lthy => (snd o Specification.theorems Thm.theoremK
        [((To-sbinding s, List.map (fn s => Attrib.check-src lthy [Token.make-string (s, Posi-
tion.none)])
                        [simp, code-unfold]),
       List.map (fn x => (Proof-Context.get-thms lthy (To-string0 x), [])) l)]
      []
      false) lthy)
| Theory-lemma (Lemma (n, l-spec, l-apply, o-by)) => in-local
   (fn lthy =>
          Specification.theorem-cmd true Thm.theoremK NONE (K I)
            Binding.empty-atts [] [] (Element.Shows [((To-sbinding n, [])
                                                    ,[((String.concatWith ( ⟹ )
                                                        (List.map of-semi--term l-spec)), [])])])
            false lthy
       |> fold (semi--command-proof o META.Command-apply) l-apply
       |> global-terminal-proof o-by)
| Theory-lemma (Lemma-assumes (n, l-spec, concl, l-apply, o-by)) => in-local
   (fn lthy => lthy
       |> Specification.theorem-cmd true Thm.theoremK NONE (K I)
           (To-sbinding n, [])
           []
           (List.map (fn (n, (b, e)) =>
                     Element.Assumes [( ( To-sbinding n
                                        , if b then [[Token.make-string (simp, Position.none)]] else [])
                                        , [(of-semi--term e, [])])])
                  l-spec)
           (Element.Shows [(Binding.empty-atts, [(of-semi--term concl, [])])])
           false
       |> fold semi--command-proof l-apply
       |> (case map-filter (fn META.Command-let - => SOME []
                             | META.Command-have - => SOME []
                             | META.Command-fix-let (-, -, -, l) => SOME l
                             | - => NONE)
                           (rev l-apply) of
            [] => global-terminal-proof o-by
          | - :: l => let val arg = (NONE, true) in fn st => st
            |> local-terminal-proof o-by
```

```
            |> fold (fn l => fold semi--command-state l o Proof.local-qed arg) l
            |> Proof.global-qed arg end))
| Theory-axiomatization (Axiomatization (n, e)) => in-theory
  (#2 o Specification.axiomatization-cmd [] [] [] [((To-sbinding n, []), of-semi--term e)])
| Theory-section - => in-theory I
| Theory-text - => in-theory I
| Theory-ML (SML ml) =>
    in-theory (Code-printing.reflect-ml (Input.source false (of-semi--term' ml)
                                                    (Position.none, Position.none)))
| Theory-setup (Setup ml) =>
    in-theory (Isar-Cmd.setup (Input.source false (of-semi--term' ml)
                                          (Position.none, Position.none)))
| Theory-thm (Thm thm) => in-local
   (fn lthy =>
    let val () =
      writeln
        (Pretty.string-of
          (Proof-Context.pretty-fact lthy (, List.map (semi--thm-attribute-single lthy) thm))) in
    lthy
    end)
| Theory-interpretation (Interpretation (n, loc-n, loc-param, o-by)) => in-local
   (fn lthy => lthy
    |> Interpretation.interpretation-cmd ( [ ( (To-string0 loc-n, Position.none)
                                     , ( (To-string0 n, true)
                                       , (if loc-param = [] then
                                            Expression.Named []
                                          else
                                            Expression.Positional (map (SOME o of-semi--term)
                                                                       loc-param), []))))]
                                  , [])
    |> global-terminal-proof o-by)
(*in fn t => fn thy => f t thy handle ERROR s => (warning s; thy)
 end*)
end

end

structure Bind-META = struct open Bind-Isabelle

fun all-meta aux ret = let open META open META-overload in fn
  META-semi-theories thy =>
    ret o (case thy of
       Theories-one thy => semi--theory I Named-Target.theory-map thy
     | Theories-locale (data, l) => fn thy => thy
       |> (   Expression.add-locale-cmd
              (To-sbinding (META.holThyLocale-name data))
              Binding.empty
              []
              ([], [])
```

```
            (List.concat
              (map
                (fn (fixes, assumes) => List.concat
                  [ map (fn (e,ty) => Element.Fixes [( To-binding (of-semi--term e)
                                                     , SOME (of-semi--typ ty)
                                                     , NoSyn)]) fixes
                  , case assumes of NONE => []
                                  | SOME (n, e) => [Element.Assumes [( (To-sbinding n, [])
                                                                     , [(of-semi--term e, [])])]]])
                (META.holThyLocale-header data)))
          #> snd)
      |> fold (fold (semi--theory Local-Theory.background-theory
                                  (fn f =>
                                    Local-Theory.new-group
                                    #> f
                                    #> Local-Theory.reset-group
                                    #> (fn lthy =>
                                       #1 (Target-Context.switch-named-cmd NONE (Context.Proof
lthy)) lthy
                                         |> Context.the-proof)))) l
      |> Local-Theory.exit-global)
| META-boot-generation-syntax - => ret o I
| META-boot-setup-env - => ret o I
| META-all-meta-embedding meta => fn thy =>
  aux
    (map2-ctxt-term
      (fn T-pure x => T-pure x
       | e =>
         let fun aux e = case e of
          T-to-be-parsed (s, -) => SOME let val t = Syntax.read-term (Proof-Context.init-global
thy)
                                                                     (To-string0 s) in
                                       (t, Term.add-frees t [])
                                       end
         | T-lambda (a, e) =>
           Option.map
             (fn (e, l-free) =>
              let val a = To-string0 a
                  val (t, l-free) = case List.partition (fn (x, -) => x = a) l-free of
                                      ([], l-free) => (Term.TFree ('a, [HOL.type]), l-free)
                                    | ([(-, t)], l-free) => (t, l-free) in
              (lambda (Term.Free (a, t)) e, l-free)
              end)
             (aux e)
         | - => NONE in
         case aux e of
           NONE => error nested pure expression not expected
         | SOME (e, -) => META.T-pure (From.Pure.term e)
         end) meta) thy
```

end

end
›**end**

Part II.

A Toy Example

5.3. A Toy Library for Objects in a State

theory *Toy-Library*
imports *Main*
begin

type-notation *option* (‹⟨-⟩$_\perp$›)
notation *Some* (‹⌊(-)⌋›)

fun *drop* :: 'α *option* ⇒ 'α (‹⌈(-)⌉›)
where *drop-lift*[*simp*]: ⌈⌊*v*⌋⌉ = *v*

type-synonym *oid* = *nat*

type-synonym 'α *val'* = *unit* ⇒ 'α
type-notation *val'* (‹·(-)›)

record ('$\mathfrak{A}$)*state* =
  heap :: *oid* ⇀ '$\mathfrak{A}$
  assocs :: *oid* ⇀ ((*oid list*) *list*) *list*

lemmas [*simp*,*code-unfold*] = *state.defs*

end

5.4. Example: A Class Model Converted into a Theory File

5.4.1. Introduction

theory
  Design-deep
imports
  ../embedding/Generator-dynamic-sequential
  ../../Antiquote-Setup
begin

In this example, we configure our package to generate a `.thy` file, without executing the associated generated code contained in this `.thy` file (c.f. `Design_shallow.thy` for a direct evaluation). This mode is particularly relevant for debugging purposes: while by default no evaluation occurs, the generated files (and their proofs!) can be executed on a step by step basis, depending on how we interact with the output window (by selectively clicking on what is generated).

After clicking on the generated content, the newly inserted content could depend on some theories which are not loaded by this current one. In this case, it is necessary to manually add all the needed dependencies above after the keyword **imports**. One should compare this current theory with `Design_shallow.thy` to see the differences of imported theories, and which ones to manually import (whenever an error happens).

generation-syntax [ ]

```
generation_syntax
  [ deep
      (generation_semantics [ design ])
      (THEORY Design_generated)
      (IMPORTS ["../Toy_Library", "../Toy_Library_Static"]
               "../embedding/Generator_dynamic_sequential")
      SECTION
      (*SORRY*) (*no_dirty*)
      [ (* in Haskell *)
        (* in OCaml module_name M *)
        (* in Scala module_name M *)
        in SML module_name M ]
      (output_directory "../document_generated")
  (*, syntax_print*) ]
```

While in theory it is possible to set the **deep** mode for generating in all target languages, i.e. by writing [*in Haskell*, *in OCaml module-name M*, *in Scala module-name M*, *in SML module-name M*], usually using only one target is enough, since the task of all target is to generate the same Isabelle content. However in case one language takes too much time to setup, we recommend to try the generation with another target language, because all optimizations are currently not (yet) seemingly implemented for all target languages, or differently activated.

5.4.2. Designing Class Models (I): Basics

The following example shows the definitions of a set of classes, called the "universe" of classes. Instead of providing a single command for building all the complete universe of classes directly in one block, we are constructing classes one by one. So globally the universe describing all classes is partial, it will only be fully constructed when all classes will be finished to be defined.

This allows to define classes without having to follow a particular order of definitions. Here *Atom* is defined before the one of *Molecule* (*Molecule* will come after):

Class *Atom* < *Molecule*
  Attributes *size* : *Integer*
End

The "blue" color of **End** indicates that **End** is not a "green" keyword. **End** and **Class** are in fact similar, they belong to the group of meta-commands (all meta-commands are defined in *Isabelle-Meta-Model.Generator-dynamic-sequential*). At run-time and in **deep** mode, the semantics of all meta-commands are approximately similar: all meta-commands displays some quantity of Isabelle code in the output window (as long as

meta-commands are syntactically correctly formed). However each meta-command is unique because what is displayed in the output window depends on the sequence of all meta-commands already encountered before (and also depends on arguments given to the meta-commands).

One particularity of **End** is to behave as the identity function when **End** is called without arguments. As example, here we are calling lots of **End** without arguments, and no Isabelle code is generated.

End End End

We remark that, like any meta-commands, **End** could have been written anywhere in this theory, for example before **Class** or even before **generation-syntax**... Something does not have to be specially opened before using an **End**.

Class *Molecule* < *Person*

As example, here no **End** is written.

The semantics of **End** is further precised here. We earlier mentioned that the universe of classes is partially constructed, but one can still examine what is partially constructed, and one possibility is to use **End** for doing so.

End can be seen as a lazy meta-command:

- without parameters, no code is generated,
- with some parameters (e.g., the symbol **!**), it forces the generation of the computation of the universe, by considering all already encountered classes. Then a partial representation of the universe can be interactively inspected.

Class *Galaxy*
  Attributes *wormhole* : *UnlimitedNatural*
  is-sound : *Void*
End!

At this position, in the output window, we can observe for the first time some generated Isabelle code, corresponding to the partial universe of classes being constructed.

Note: By default, *Atom* and *Molecule* are not (yet) present in the shown universe because *Person* has not been defined in a separate line (unlike *Galaxy* above).

Class *Person* < *Galaxy*
  Attributes *salary* : *Integer*
  boss : *Person*
  is-meta-thinking: *Boolean*

There is not only **End** which forces the computation of the universe, for example **Instance** declares a set of objects belonging to the classes earlier defined, but the entire universe is needed as knowledge, so there is no choice than forcing the generation of the universe.

Instance $X_{Person}1$ :: *Person* = [*salary* = *1300* , *boss* = $X_{Person}2$]
  and $X_{Person}2$ :: *Person* = [*salary* = *1800*]

Here we will call **Instance** again to show that the universe will not be computed again since it was already computed in the previous **Instance**.

Instance $X_{Person}3 :: Person = [\ salary = 1\ ]$

However at any time, the universe can (or will) automatically be recomputed, whenever we are adding meanwhile another class:

```
(* Class Big_Bang < Atom (* This will force the creation of a new universe.
*) *)
```

As remark, not only the universe is recomputed, but the recomputation takes also into account all meta-commands already encountered. So in the new setting, $X_{Person}1$, $X_{Person}2$ and $X_{Person}3$ will be resurrected... after the *Big-Bang*.

5.4.3. Designing Class Models (II): Jumping to Another Semantic Floor

Until now, meta-commands was used to generate lines of code, and these lines belong to the Isabelle language. One particularity of meta-commands is to generate pieces of code containing not only Isabelle code but also arbitrary meta-commands. In **deep** mode, this is particularly not a danger for meta-commands to generate themselves (whereas for **shallow** the recursion might not terminate).

In this case, such meta-commands must automatically generate the appropriate call to **generation-syntax** beforehand. However this is not enough, the compiling environment (comprising the history of meta-commands) are changing throughout the interactive evaluations, so the environment must also be taken into account and propagated when meta-commands are generating themselves. For example, the environment is needed for consultation whenever resurrecting objects, recomputing the universe or accessing the hierarchy of classes being defined.

As a consequence, in the next example a line **setup** is added after **generation-syntax** for bootstrapping the state of the compiling environment.

State $\sigma_1 =$
$[\ ([\ salary = 1000\ ,\ boss = \textbf{self}\ 1\ ] :: Person)$
$,\ ([\ salary = 1200\ ] :: Person)$

$,\ ([\ salary = 2600\ ,\ boss = \textbf{self}\ 3\ ] :: Person)$
$,\ X_{Person}1$
$,\ ([\ salary = 2300\ ,\ boss = \textbf{self}\ 2\ ] :: Person)$

$,\ X_{Person}2\ ]$

State $\sigma_1{}' =$
$[\ X_{Person}1$
$,\ X_{Person}2$
$,\ X_{Person}3\ ]$

In certain circumstances, the command **setup** must be added again between some par-

ticular interleaving of two meta-commands and this may not depend on the presence of **generation-syntax** (which is defined only once when generating the first meta-command). For more details, one can refer to the source code of *ignore-meta-header* and *bootstrap-floor*.

PrePost σ_1 $\sigma_1{}'$

The generation of meta-commands allows to perform various extensions on the Toy language being embedded, without altering the semantics of a particular command. **PrePost** usually only takes "bound variables" as parameters (not arbitrary λ-terms), however the semantics of **PrePost** was extended to mimic the support of some particular terms not restricted to variables. This extension was implemented by executing some steps of "ζ-reductions rewriting rules" operating on the meta-level of commands. First, it is at least needed to extend the syntax of expressions accepted by **PrePost**, we then modify the parsing so that a larger subset of λ-terms can be given as parameters. Starting from this expression:

```
(* PrePost \<sigma>\<^sub>1 [ ([ salary = 1000 , boss = self 1 ] :: Person)
] *)
```

the rewriting begins with a first call to the next semantic floor, we obtain the following meta-commands (where **PrePost** [*shallow*] is an expression in normal form):

```
(* State WFF_10_post = [ ([ "salary" = 1000, "boss" = self 1 ] :: Person)
]
PrePost[shallow] \<sigma>\<^sub>1 WFF_10_post *)
```

(*WFF-10-post* is an automatically generated name).

The rewriting of the above **State** is performed in its turn. Finally the overall ultimately terminates when reaching **Instance** being already in normal form:

```
(* Instance WFF_10_post_object0 :: Person = [ "salary" = 1000, "boss" = [
] ]
State[shallow] WFF_10_post = [ WFF_10_post_object0 ]
PrePost[shallow] \<sigma>\<^sub>1 WFF_10_post *)
```

5.4.4. Designing Class Models (III): Interaction with (Pure) Term

Meta-commands are obviously not restricted to manipulate expressions in the Outer Syntax level. It is possible to build meta-commands so that Inner Syntax expressions are directly parsed. However the dependencies of this theory have been minimized so that experimentations and debugging can easily occur in **deep** mode (this file only depends on *Isabelle-Meta-Model.Generator-dynamic-sequential*). Since the Inner Syntax expressions would perhaps manipulate expressions coming from other theories than *Isabelle-Meta-Model.Generator-dynamic-sequential*, it can be desirable to consider the Inner Syntax container as a string and leave the parsing for subsequent semantic floors. This is what is implemented here:

```
Context Person :: content ()
  Post "><"
```

Here the expression `><` is not well-typed in Isabelle, but an error is not raised because the above expression is not (yet) parsed as an Inner Syntax element[1].

However, this is not the same for the resulting generated meta-command:

```
Context [shallow] Person :: content ()
  Post : "(\<lambda> result self. (><))"
```

and an error is immediately raised because the parsing of Inner Syntax expressions is activated in this case.

For example, one can put the mouse, with the CTRL gesture, over the variable a, b or c to be convinced that they are free variables compared with above:

Context[**shallow**] *Person* :: *content* ()
  Post : $a + b = c$

5.4.5. Designing Class Models (IV): Saving the Generated to File

The experimentations usually finish by saving all the universe and generated Isabelle theory to the hard disk:

```
(* generation_syntax deep flush_all *)
```

5.4.6. Designing Class Models (V): Inspection of Generated Files

According to options given to the (first) command **generation-syntax** above, we retrieve the first generated file in the mentioned directory: `../document_generated/Design_generated.thy`.

Because this file still contains meta-commands, we are here executing again a new generating step inside this file, the new result becomes saved in `../document_generated/Design_generated_generated.thy`. As remark, in this last file, the dependency to *Isabelle-Meta-Model.Generator-dynamic-sequential* was automatically removed because the meta-compiler has detected the absence of meta-commands in the generated content.

Note: While the first generated file is intended to be always well-typed, it can happen that subsequent generations will lead to a not well-typed file. This is because the meta-compiler only saves the history of meta-commands. In case some "native" Isabelle declarations are generated among meta-commands, then these Isabelle declarations are not saved by the meta-compiler, so these declarations will not be again generated. Anyway, we see potential solutions for solving this and they would perhaps be implemented in a future version of the meta-compiler...

end

[1] In any case an error will not be raised, because the above code is written in verbatim in the real `.thy` file, however one can copy-paste this code out of the verbatim scope to see that no errors are really raised. For presentation purposes, it was embedded in verbatim because we will later discuss about meta-commands generating Isabelle code, and then what is generated by this meta-command is of course not well-typed!

5.5. Example: A Class Model Interactively Executed

5.5.1. Introduction

theory
Design-shallow
imports
../Toy-Library
../Toy-Library-Static
../embedding/Generator-dynamic-sequential
../../Antiquote-Setup
begin

In this example, we configure our package to execute tactic SML code (corresponding to some generated `.thy` file, `Design_deep.thy` details how to obtain such generated `.thy` file). Since SML code are already compiled (or reflected) and bound with the native Isabelle API in *Isabelle-Meta-Model.Generator-dynamic-sequential*, nothing is generated in this theory. The system only parses arguments given to meta-commands and immediately calls the corresponding compiled functions.

The execution time is comparatively similar as if tactics were written by hand, except that the generated SML code potentially inherits all optimizations performed by the raw code generation of Isabelle (if any).

generation-syntax [**shallow** (**generation-semantics** [**design**])

]

The configuration in **shallow** mode is straightforward: in this mode **generation-syntax** basically terminates in $O(1)$.

5.5.2. Designing Class Models (I): Basics

Class *Atom* < *Molecule*
Attributes *size* : *Integer*
End

End End End

Class *Molecule* < *Person*

Class *Galaxy*
Attributes *wormhole* : *UnlimitedNatural*
is-sound : *Void*
End!

Class *Person* < *Galaxy*
Attributes *salary* : *Integer*
boss : *Person*
is-meta-thinking: *Boolean*

Instance $X_{Person}1$ :: *Person* = [*salary* = *1300* , *boss* = $X_{Person}2$]
and $X_{Person}2$ :: *Person* = [*salary* = *1800*]

Instance $X_{Person}3$ :: *Person* = [*salary* = *1*]

5.5.3. Designing Class Models (II): Jumping to Another Semantic Floor

5.5.4. Designing Class Models (III): Interaction with (Pure) Term

Here in **shallow** mode, the following expression is directly rejected:

```
Context Person :: content ()
  Post "><"
```

Context[**shallow**] *Person* :: *content* ()
Post : $a + b = c$

end

Bibliography

[1] A. D. Brucker, F. Tuong, and B. Wolff. Featherweight ocl: A proposal for a machine-checked formal semantics for ocl 2.5. *Archive of Formal Proofs*, Jan. 2014. ISSN 2150-914x. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-featherweight-2014. http://isa-afp.org/entries/Featherweight_OCL.shtml, Formal proof development.

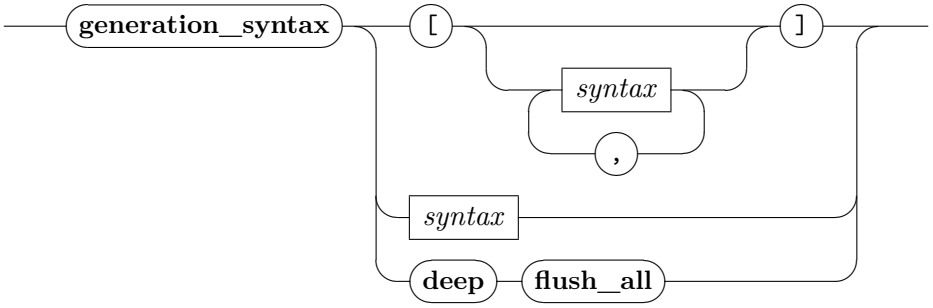
Part III.

Appendix

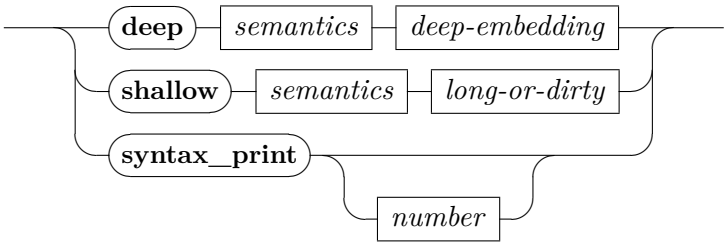
A. Grammars of Commands

A.1. Main Setup of Meta Commands

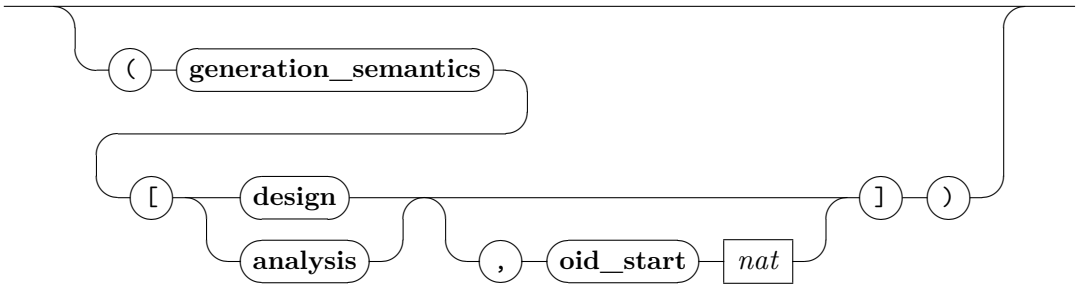
generation-syntax : *theory* → *theory*



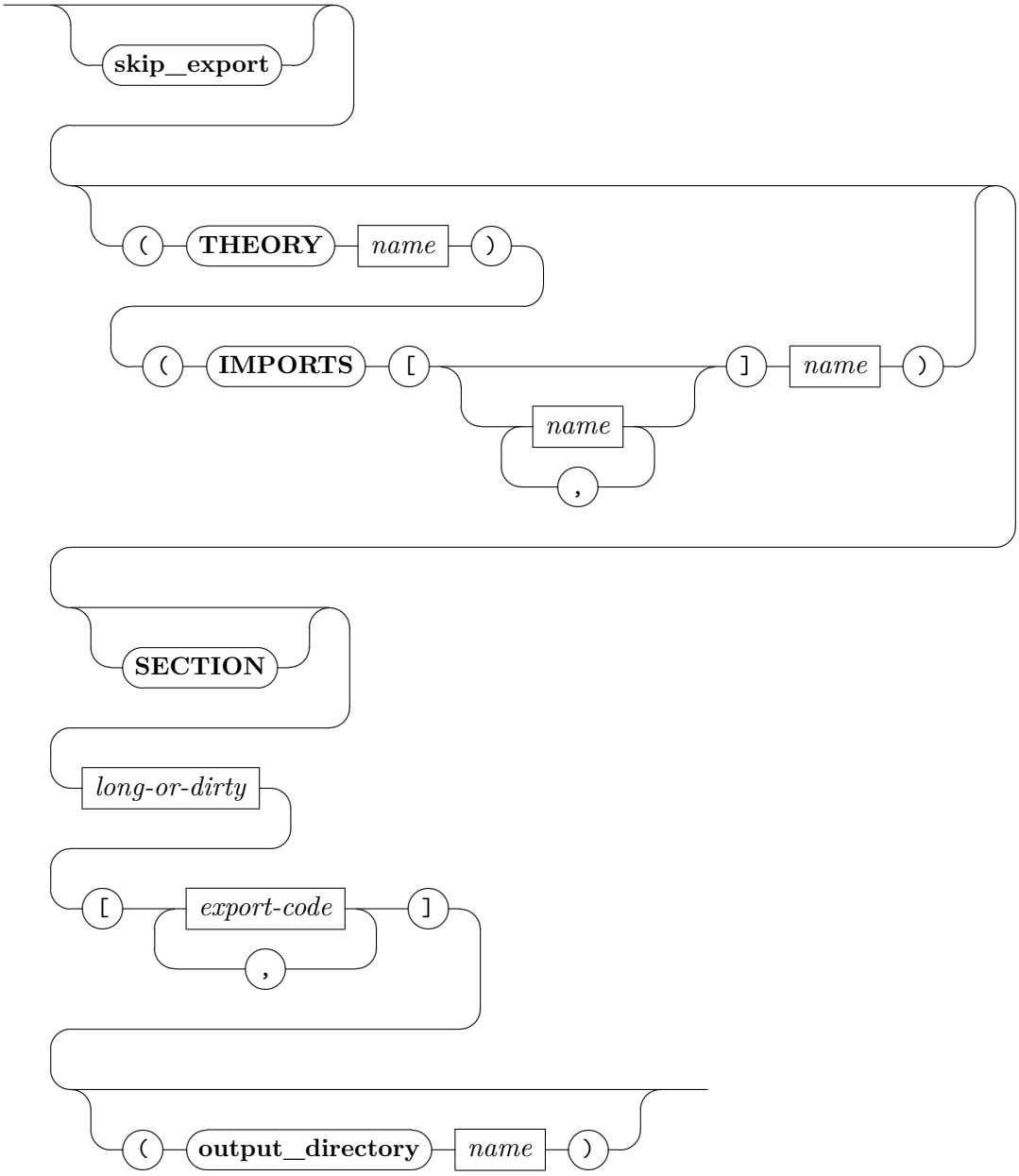
syntax



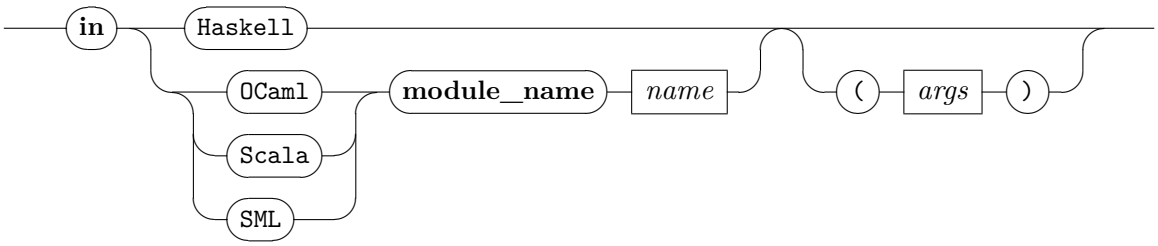
semantics



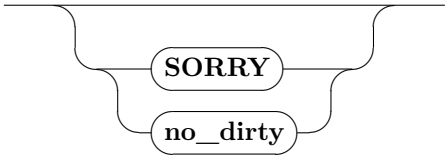
deep-embedding



export-code



long-or-dirty



generation-syntax sets the behavior of all incoming meta-commands. By default, without firstly writing **generation-syntax**, meta-commands will only print in output what they have parsed, this is similar as giving to **generation-syntax** a non-empty list having only **syntax-print** as elements (on the other hand, nothing is printed when an empty list is received). Additionally **syntax-print** can be followed by an integer indicating the printing depth in output, similar as declaring *ML-print-depth* with an integer, but the global option **syntax-print** is restricted to meta-commands. Besides the printing of syntaxes, several options are provided to further analyze the semantics of languages being embedded, and tell if their evaluation should occur immediately using the **shallow** mode, or to only display what would have been evaluated using the **deep** mode (i.e., to only show the generated Isabelle content in the output window).

Since several occurrences of **deep**, **shallow** or **syntax-print** can appear in the parameterizing list, for each meta-command the overall evaluation respects the order of events given in the list (from head to tail). At the time of writing, it is only possible to evaluate this list sequentially: the execution stops as soon as one first error is raised, thus ignoring remaining events.

generation-syntax deep flush-all performs as side effect the writing of all the generated Isabelle contents to the hard disk (all at the calling time), by iterating the saving for each **deep** mode in the list. In particular, this is only effective if there is at least one **deep** mode earlier declared.

As a side note, target languages for the **deep** mode currently supported are: Haskell, OCaml, Scala and SML. So in principle, all these targets generate the same Isabelle content and exit correctly. However, depending on the intended use, exporting with some targets may be more appropriate than other targets:

- For efficiency reasons, the meta-compiler has implemented a particular optimization for accelerating the process of evaluating incoming meta-commands. By default in Haskell and OCaml, the meta-compiler (at HOL side) is exported only

once, during the **generation-syntax** step. Then all incoming meta-commands are considered as arguments sent to the exported meta-compiler. As a compositionality aspect, these arguments are compiled then linked together with the (already compiled) meta-compiler, but this implies the use of one call of *unsafeCoerce* in Haskell and one *Obj.magic* statement in OCaml (otherwise another solution would be to extract the meta-compiler as a functor). Similar optimizations are not yet implemented for Scala and are only half-implemented for the SML target (which basically performs a step of marshalling to string in Isabelle/ML).

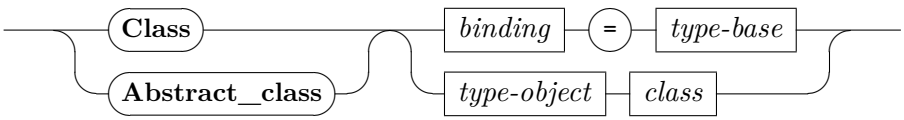
- For safety reasons, it simply suffices to extract all the meta-compiler together with the respective arguments in front of each incoming meta-commands everytime, then the overall needs to be newly compiled everytime. This is the current implemented behavior for Scala. For Haskell, OCaml and SML, it was also the default behavior in a prototyping version of the compiler, as a consequence one can restore that functionality for future versions.

Concerning the semantics of generated contents, if lemmas and proofs are generated, **SORRY** allows to explicitly skip the evaluation of all proofs, irrespective of the presence of **sorry** or not in generated proofs. In any cases, the semantics of **sorry** has not been overloaded, e.g., red background may appear as usual.

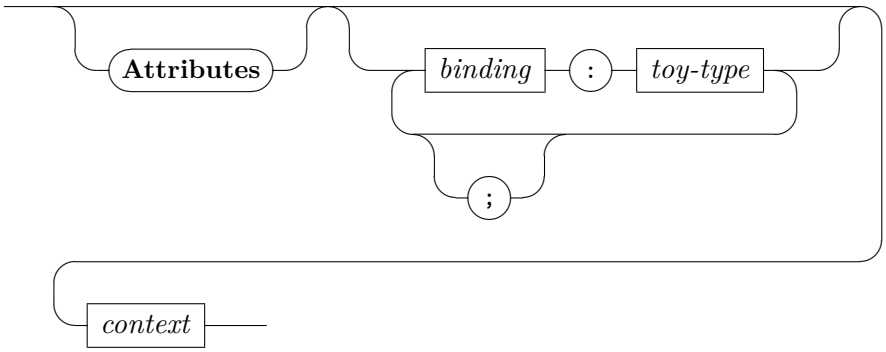
Finally **generation-semantics** is a container for specifying various options for varying the semantics of languages being embedded. For example, **design** and **analysis** are two options for specifying how the modelling of objects will be represented in the Toy Language. Similarly, this would be a typical place for options like *eager* or *lazy* for choosing how the evaluation should happen...

A.2. All Meta Commands of the Toy Language

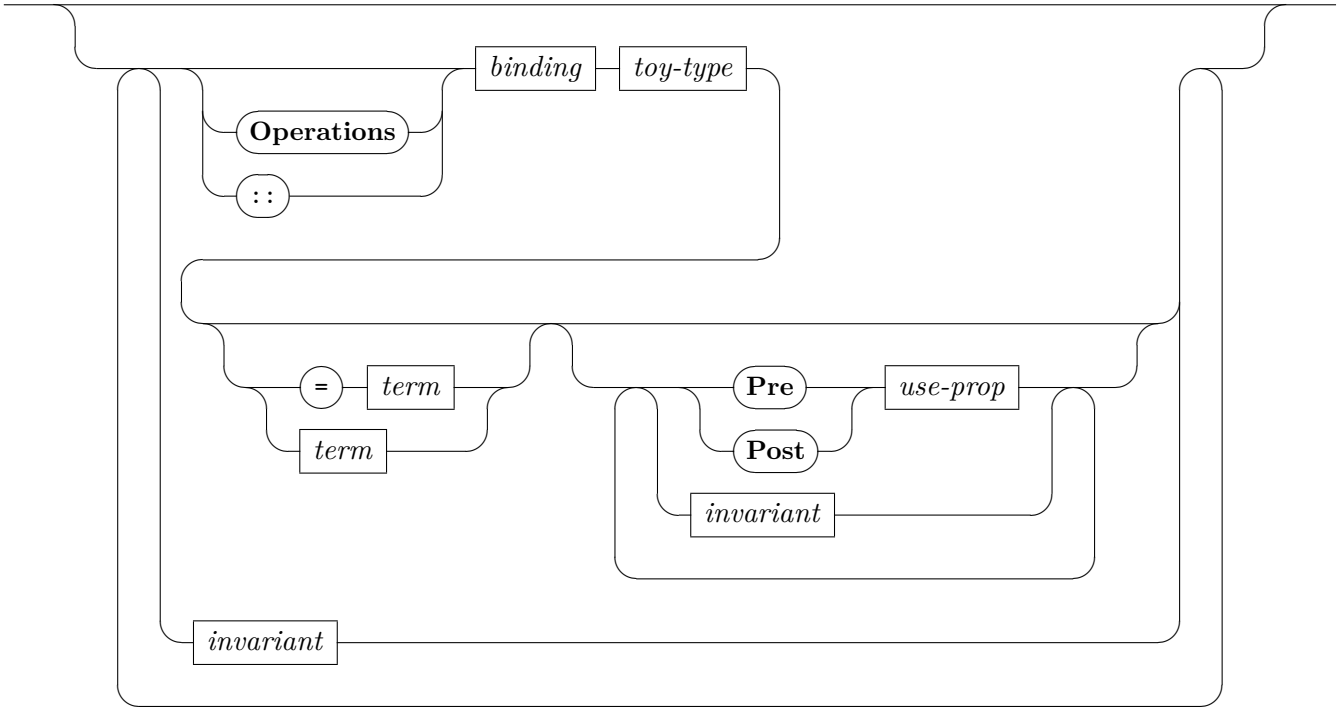
Class : *theory* → *theory*
Abstract-class : *theory* → *theory*



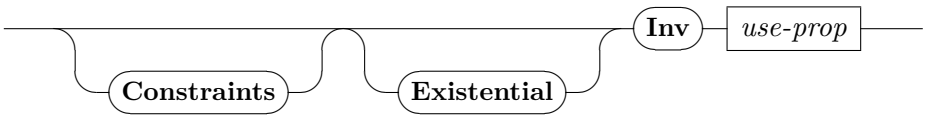
class



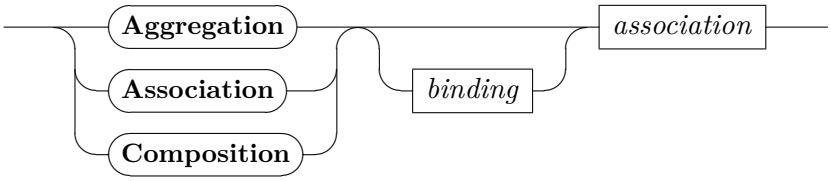
context



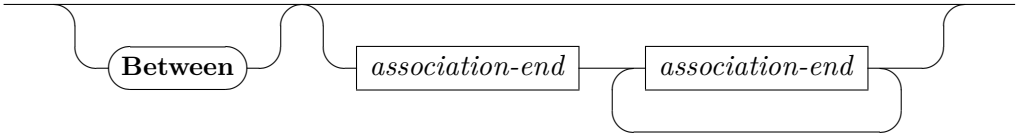
invariant



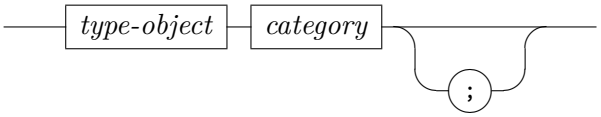
Aggregation : *theory* → *theory*
Association : *theory* → *theory*
Composition : *theory* → *theory*



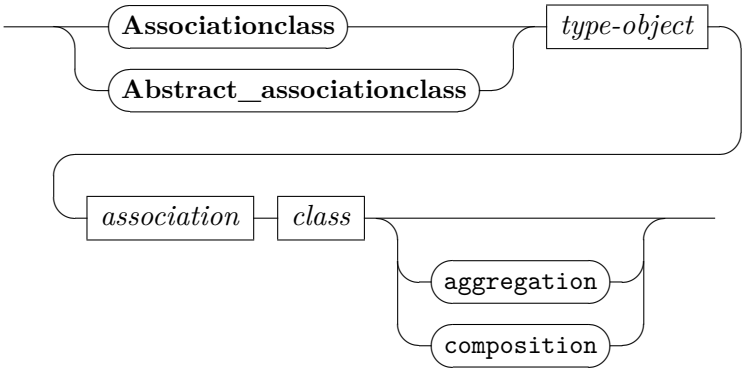
association



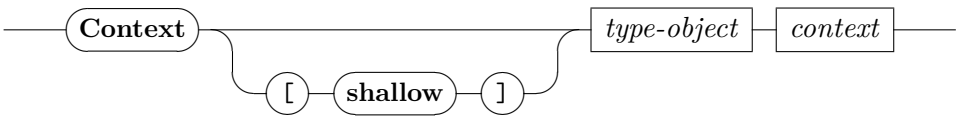
association-end



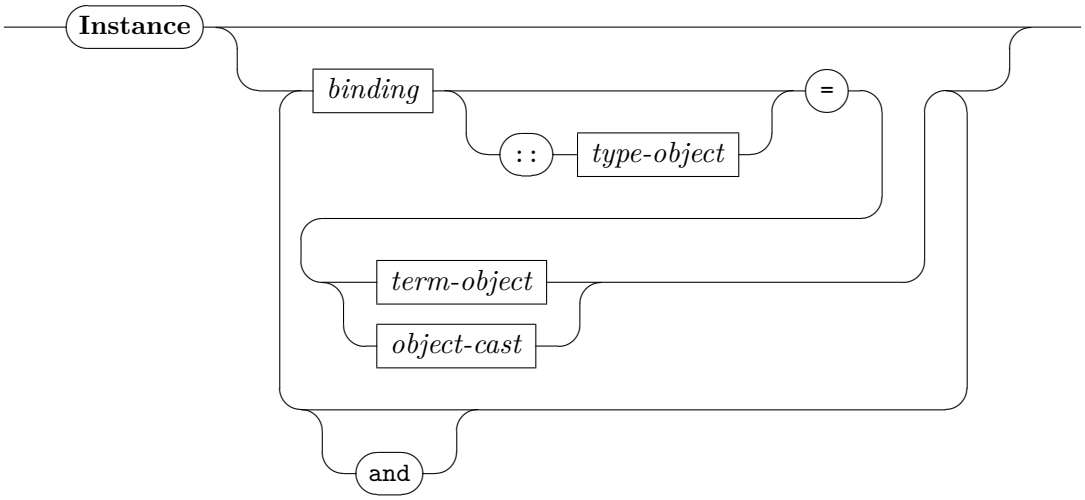
Associationclass : *theory* → *theory*
Abstract-associationclass : *theory* → *theory*



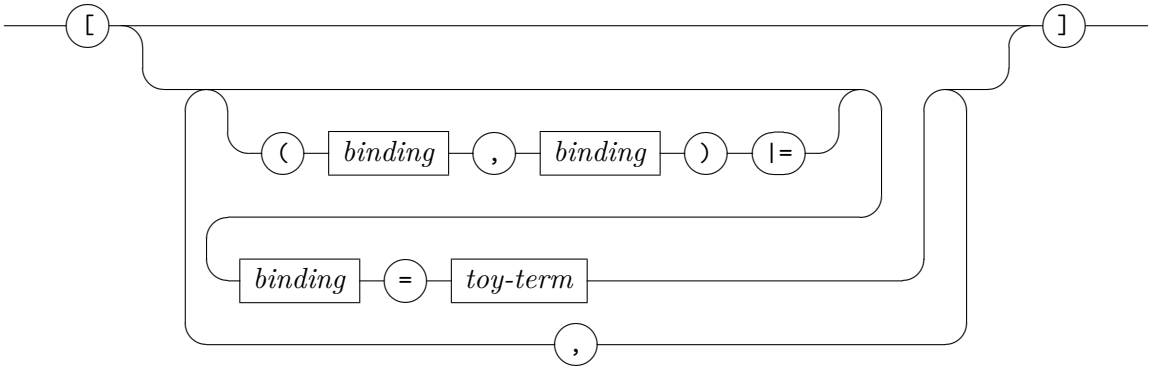
Context : *theory* → *theory*



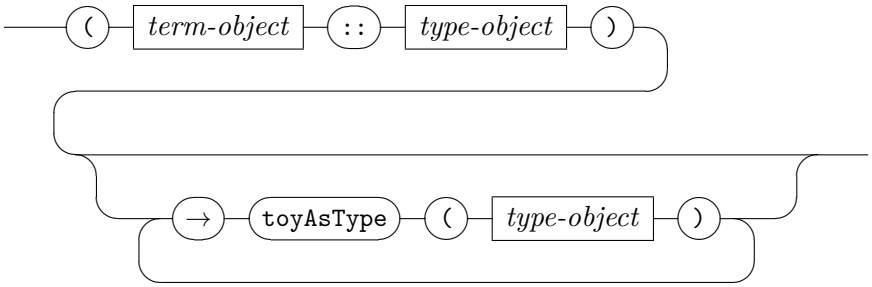
Instance : *theory* → *theory*



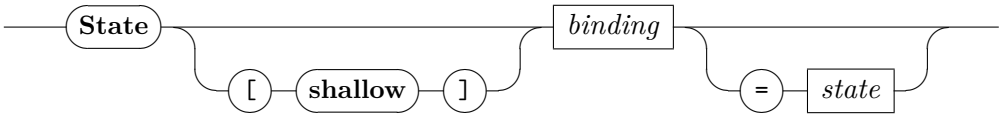
term-object



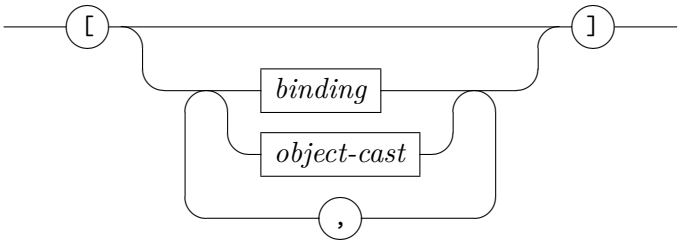
object-cast



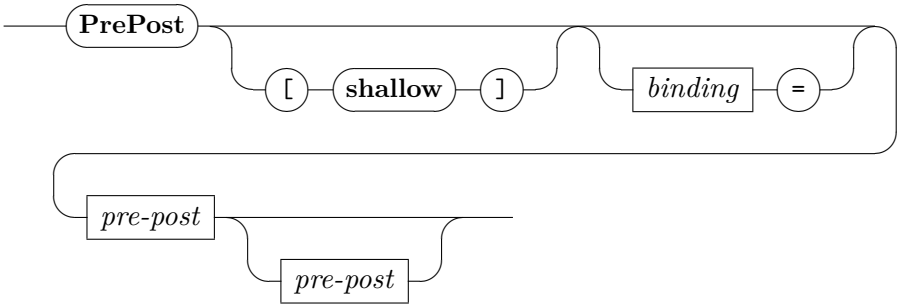
State : *theory* → *theory*



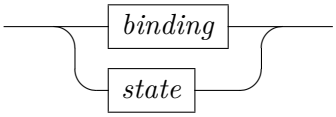
state



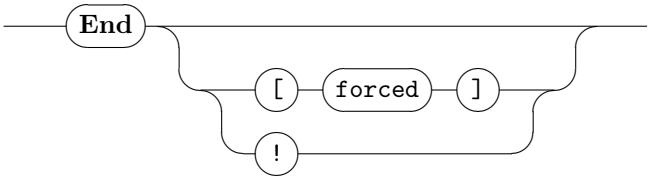
PrePost : *theory* → *theory*



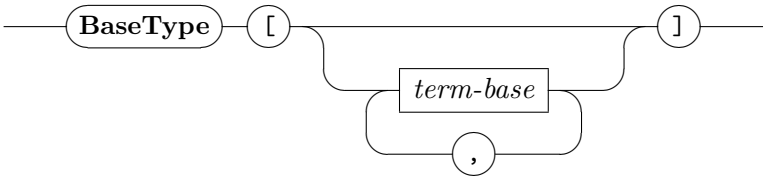
pre-post



End : *theory* → *theory*

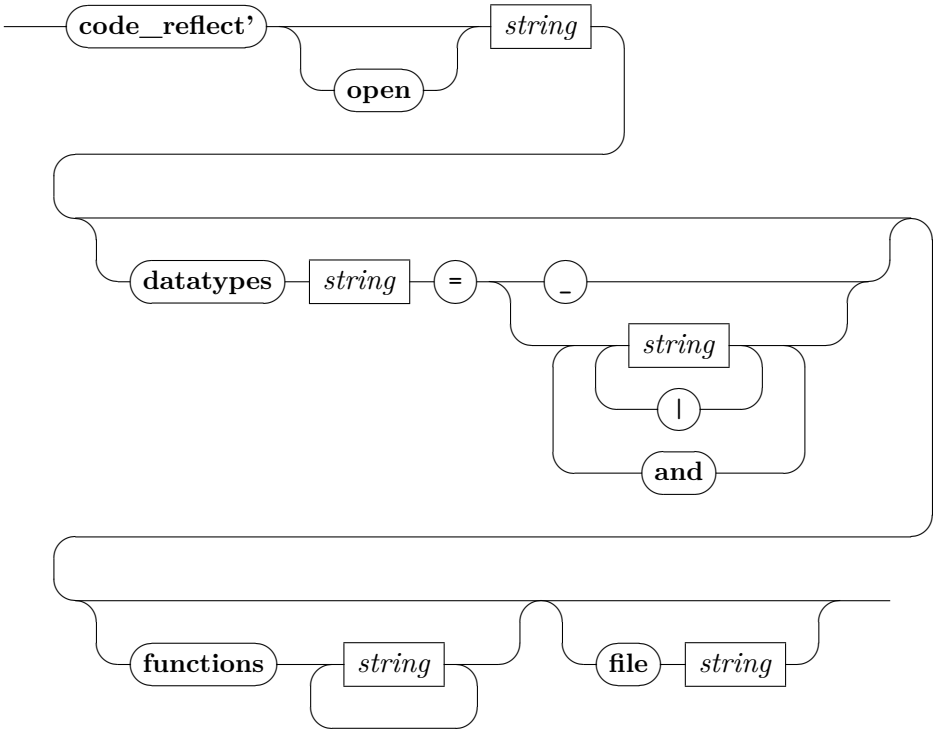


BaseType : *theory* → *theory*



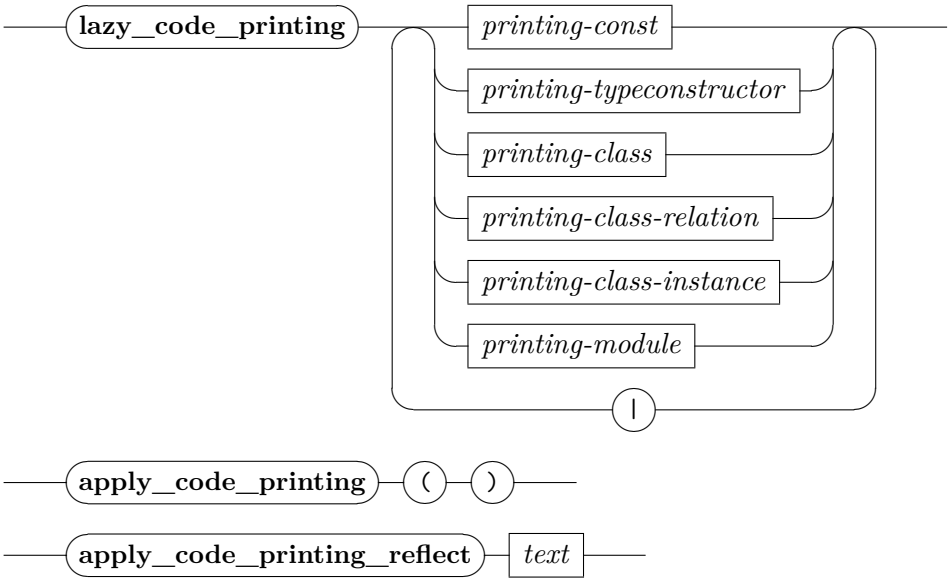
A.3. Extensions of Isabelle Commands

code-reflect′ : *theory* → *theory*



code-reflect′ has the same semantics as **code-reflect** except that it additionally contains the option **open** inspired from the command **export-code** (with the same semantics).

lazy-code-printing : *theory* → *theory*
apply-code-printing : *theory* → *theory*
apply-code-printing-reflect : *local-theory* → *local-theory*



lazy-code-printing has the same semantics as **code-printing** or **ML**, except that no side effects occur until we give more details about its intended future semantics: this will be precised by calling **apply-code-printing** or **apply-code-printing-reflect**.

apply-code-printing repeatedly calls **code-printing** to all previously registered elements with **lazy-code-printing** (the order is preserved).

apply-code-printing-reflect repeatedly calls **ML** to all previously registered elements with **lazy-code-printing** (the order is preserved). As a consequence, code for other targets (Haskell, OCaml, Scala) are ignored. Moreover before the execution of the overall, it is possible to give an additional piece of SML code as argument to priorly execute.

B. Content of the Directory isabelle_home

B.1. Extensions for Cartouches

- `./src/HOL/ex/Isabelle_Cartouche_Examples.thy` *Main0*:
  Some functions have been generalized for supporting cartouches.

B.2. Other Changes

- `./src/Tools/Code/Isabelle_code_runtime.thy` *Main1*:
  The option *open* was introduced in this file for the definition of $code\_reflect'$.
- `./src/Tools/Code/Isabelle_code_target.thy` *Main1*:
  Some signatures was removed for exposing the main structure, we have also defined at the end the implementation of $lazy\_code\_printing$, $apply\_code\_printing$ and $apply\_code\_printing\_reflect$.
- `./src/Pure/Isar/Isabelle_typedecl.thy` *Main2*:
  Short modification of the argument lifting a *binding* to a *binding option* with some signatures removed.

C. Content of One Generated File (as example)

theory *Design-generated-generated* **imports** *../Toy-Library ../Toy-Library-Static* **begin**

For certain concepts like classes and class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that "compiles" a concrete, closed-world class diagram into a "theory" of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Our data universe consists in the concrete class diagram just of node's, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

datatype $ty\mathcal{EXT}_{Atom} = mk\mathcal{EXT}_{Atom}$ *oid oid list option int option bool option nat option unit option*
datatype $ty_{Atom} = mk_{Atom}$ $ty\mathcal{EXT}_{Atom}$ *int option*
datatype $ty\mathcal{EXT}_{Molecule} = mk\mathcal{EXT}_{Molecule\text{-}Atom}$ ty_{Atom}
| $mk\mathcal{EXT}_{Molecule}$ *oid oid list option int option bool option nat option unit option*
datatype $ty_{Molecule} = mk_{Molecule}$ $ty\mathcal{EXT}_{Molecule}$
datatype $ty\mathcal{EXT}_{Person} = mk\mathcal{EXT}_{Person\text{-}Molecule}$ $ty_{Molecule}$
| $mk\mathcal{EXT}_{Person\text{-}Atom}$ ty_{Atom}
| $mk\mathcal{EXT}_{Person}$ *oid nat option unit option*
datatype $ty_{Person} = mk_{Person}$ $ty\mathcal{EXT}_{Person}$ *oid list option int option bool option*
datatype $ty\mathcal{EXT}_{Galaxy} = mk\mathcal{EXT}_{Galaxy\text{-}Person}$ ty_{Person}
| $mk\mathcal{EXT}_{Galaxy\text{-}Molecule}$ $ty_{Molecule}$
| $mk\mathcal{EXT}_{Galaxy\text{-}Atom}$ ty_{Atom}
| $mk\mathcal{EXT}_{Galaxy}$ *oid*
datatype $ty_{Galaxy} = mk_{Galaxy}$ $ty\mathcal{EXT}_{Galaxy}$ *nat option unit option*
datatype $ty\mathcal{EXT}_{ToyAny} = mk\mathcal{EXT}_{ToyAny\text{-}Galaxy}$ ty_{Galaxy}
| $mk\mathcal{EXT}_{ToyAny\text{-}Person}$ ty_{Person}
| $mk\mathcal{EXT}_{ToyAny\text{-}Molecule}$ $ty_{Molecule}$
| $mk\mathcal{EXT}_{ToyAny\text{-}Atom}$ ty_{Atom}
| $mk\mathcal{EXT}_{ToyAny}$ *oid*
datatype $ty_{ToyAny} = mk_{ToyAny}$ $ty\mathcal{EXT}_{ToyAny}$

Now, we construct a concrete "universe of ToyAny types" by injection into a sum type containing the class types. This type of ToyAny will be used as instance for all respective type-variables.

datatype $\mathfrak{A}$ = in_{Atom} ty_{Atom}
| $in_{Molecule}$ $ty_{Molecule}$
| in_{Person} ty_{Person}
| in_{Galaxy} ty_{Galaxy}
| in_{ToyAny} ty_{ToyAny}

Having fixed the object universe, we can introduce type synonyms that exactly correspond to Toy types. Again, we exploit that our representation of Toy is a "shallow embedding" with a one-to-one correspondance of Toy-types to types of the meta-language HOL.

type-synonym *Atom* = $\langle\langle ty_{Atom}\rangle_\perp\rangle_\perp$
type-synonym *Molecule* = $\langle\langle ty_{Molecule}\rangle_\perp\rangle_\perp$
type-synonym *Person* = $\langle\langle ty_{Person}\rangle_\perp\rangle_\perp$
type-synonym *Galaxy* = $\langle\langle ty_{Galaxy}\rangle_\perp\rangle_\perp$
type-synonym *ToyAny* = $\langle\langle ty_{ToyAny}\rangle_\perp\rangle_\perp$

definition oid_{Atom}*-0---boss = 0*
definition $oid_{Molecule}$*-0---boss = 0*
definition oid_{Person}*-0---boss = 0*

definition $switch_2$*-01* = (λ [*x0* , *x1*] ⇒ (*x0* , *x1*))
definition $switch_2$*-10* = (λ [*x0* , *x1*] ⇒ (*x1* , *x0*))

definition *oid1 = 1*
definition *oid2 = 2*
definition *inst-assoc1* = (λ*oid-class to-from oid*. ((*case* (*deref-assocs-list* ((*to-from*::*oid list list* ⇒ *oid list* × *oid list*)) ((*oid*::*oid*)) ((*drop* ((((*map-of-list* ([(*oid*$_{Person}$*-0---boss* , (*List.map* ((λ(*x* , *y*). [*x* , *y*]) *o* $switch_2$*-01*) ([[[*oid1*] , [*oid2*]]])))])))) ((*oid-class*::*oid*))))))) *of Nil* ⇒ *None*
| *l* ⇒ (*Some* (*l*)))::*oid list option*))

definition *oid3 = 3*
definition *inst-assoc3* = (λ*oid-class to-from oid*. ((*case* (*deref-assocs-list* ((*to-from*::*oid list list* ⇒ *oid list* × *oid list*)) ((*oid*::*oid*)) ((*drop* ((((*map-of-list* ([]))) ((*oid-class*::*oid*))))))) *of Nil* ⇒ *None*
| *l* ⇒ (*Some* (*l*)))::*oid list option*))

definition *oid4 = 4*
definition *oid5 = 5*
definition *oid6 = 6*
definition *oid7 = 7*

definition *inst-assoc4* = (λ*oid-class to-from oid*. ((*case* (*deref-assocs-list* ((*to-from*::*oid list list* ⇒ *oid list* × *oid list*)) ((*oid*::*oid*)) ((*drop* ((((*map-of-list* ([(*oid*$_{Person}$*-0---boss* , (*List.map* ((λ(*x* , *y*). [*x* , *y*]) *o switch*$_2$*-01*) ([[[*oid7*] , [*oid6*]] , [[*oid6*] , [*oid1*]] , [[*oid4*] , [*oid5*]]])))])))
((*oid-class*::*oid*))))))) *of Nil* ⇒ *None*
| *l* ⇒ (*Some* (*l*)))::*oid list option*))

locale *state-*σ_1 =
fixes *oid4* :: *nat*
fixes *oid5* :: *nat*
fixes *oid6* :: *nat*
fixes *oid1* :: *nat*
fixes *oid7* :: *nat*
fixes *oid2* :: *nat*
assumes *distinct-oid*: (*distinct* ([*oid4* , *oid5* , *oid6* , *oid1* , *oid7* , *oid2*]))
fixes σ_1*-object0*$_{Person}$ :: ty_{Person}
fixes σ_1*-object0* :: ·*Person*
assumes σ_1*-object0-def*: σ_1*-object0* = (λ-. ⌊⌊σ_1*-object0*$_{Person}$⌋⌋)
fixes σ_1*-object1*$_{Person}$ :: ty_{Person}
fixes σ_1*-object1* :: ·*Person*
assumes σ_1*-object1-def*: σ_1*-object1* = (λ-. ⌊⌊σ_1*-object1*$_{Person}$⌋⌋)
fixes σ_1*-object2*$_{Person}$ :: ty_{Person}
fixes σ_1*-object2* :: ·*Person*
assumes σ_1*-object2-def*: σ_1*-object2* = (λ-. ⌊⌊σ_1*-object2*$_{Person}$⌋⌋)
fixes X_{Person}*1*$_{Person}$ :: ty_{Person}
fixes X_{Person}*1* :: ·*Person*
assumes X_{Person}*1-def*: X_{Person}*1* = (λ-. ⌊⌊X_{Person}*1*$_{Person}$⌋⌋)
fixes σ_1*-object4*$_{Person}$ :: ty_{Person}
fixes σ_1*-object4* :: ·*Person*
assumes σ_1*-object4-def*: σ_1*-object4* = (λ-. ⌊⌊σ_1*-object4*$_{Person}$⌋⌋)
fixes X_{Person}*2*$_{Person}$ :: ty_{Person}
fixes X_{Person}*2* :: ·*Person*
assumes X_{Person}*2-def*: X_{Person}*2* = (λ-. ⌊⌊X_{Person}*2*$_{Person}$⌋⌋)
begin
definition σ_1 = (*state.make* ((*Map.empty* (*oid4* ↦ (in_{Person} (σ_1*-object0*$_{Person}$)), *oid5* ↦ (in_{Person} (σ_1*-object1*$_{Person}$)), *oid6* ↦ (in_{Person} (σ_1*-object2*$_{Person}$)), *oid1* ↦ (in_{Person} (X_{Person}*1*$_{Person}$)), *oid7* ↦ (in_{Person} (σ_1*-object4*$_{Person}$)), *oid2* ↦ (in_{Person} (X_{Person}*2*$_{Person}$))))) ((*map-of-list* ([(*oid*$_{Person}$*-0---boss* , (*List.map* ((λ(*x* , *y*). [*x* , *y*]) *o switch*$_2$*-01*) ([[[*oid4*] , [*oid2*]] , [[*oid6*] , [*oid4*]] , [[*oid1*] , [*oid6*]] , [[*oid7*] , [*oid3*]]])))])))))

lemma *perm-*σ_1 : σ_1 = (*state.make* ((*Map.empty* (*oid2* ↦ (in_{Person} (X_{Person}*2*$_{Person}$)), *oid7* ↦ (in_{Person} (σ_1*-object4*$_{Person}$)), *oid1* ↦ (in_{Person} (X_{Person}*1*$_{Person}$)), *oid6* ↦ (in_{Person} (σ_1*-object2*$_{Person}$)), *oid5* ↦ (in_{Person} (σ_1*-object1*$_{Person}$)), *oid4* ↦ (in_{Person} (σ_1*-object0*$_{Person}$))))) ((*assocs* (σ_1))))
apply(*simp add*: σ_1*-def*)
apply(*subst* (*1*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*2*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)

apply(*subst* (*1*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*3*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*2*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*1*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*4*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*3*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*2*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*1*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*5*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*4*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*3*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*2*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*1*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
by(*simp*)
end

locale *state-*$\sigma_1{}'$ =
fixes *oid1* :: *nat*
fixes *oid2* :: *nat*
fixes *oid3* :: *nat*
assumes *distinct-oid*: (*distinct* ([*oid1* , *oid2* , *oid3*]))
fixes $X_{Person}1_{Person}$ :: ty_{Person}
fixes $X_{Person}1$ :: ·*Person*
assumes X_{Person}*1-def*: $X_{Person}1 = (\lambda\text{-}.\ \lfloor\lfloor X_{Person}1_{Person}\rfloor\rfloor)$
fixes $X_{Person}2_{Person}$ :: ty_{Person}
fixes $X_{Person}2$ :: ·*Person*
assumes X_{Person}*2-def*: $X_{Person}2 = (\lambda\text{-}.\ \lfloor\lfloor X_{Person}2_{Person}\rfloor\rfloor)$
fixes $X_{Person}3_{Person}$ :: ty_{Person}
fixes $X_{Person}3$ :: ·*Person*
assumes X_{Person}*3-def*: $X_{Person}3 = (\lambda\text{-}.\ \lfloor\lfloor X_{Person}3_{Person}\rfloor\rfloor)$
begin
definition $\sigma_1{}' = (state.make\ ((Map.empty\ (oid1 \mapsto (in_{Person}\ (X_{Person}1_{Person})),\ oid2 \mapsto (in_{Person}\ (X_{Person}2_{Person})),\ oid3 \mapsto (in_{Person}\ (X_{Person}3_{Person})))))\ ((map\text{-}of\text{-}list\ ([(oid_{Person}\text{-}0\text{---}boss , (List.map\ ((\lambda(x\ ,\ y).\ [x\ ,\ y])\ o\ switch_2\text{-}01)\ ([[[oid1]\ ,\ [oid2]]]))))])))))$

lemma *perm-*$\sigma_1{}'$: $\sigma_1{}' = (state.make\ ((Map.empty\ (oid3 \mapsto (in_{Person}\ (X_{Person}3_{Person})),\ oid2 \mapsto (in_{Person}\ (X_{Person}2_{Person})),\ oid1 \mapsto (in_{Person}\ (X_{Person}1_{Person})))))\ ((assocs\ (\sigma_1{}'))))$
apply(*simp add*: $\sigma_1{}'$*-def*)
apply(*subst* (*1*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*2*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
apply(*subst* (*1*) *fun-upd-twist*, *metis distinct-oid distinct-length-2-or-more*)
by(*simp*)
end

locale *pre-post-*σ_1*-*$\sigma_1{}'$ =
fixes *oid1* :: *nat*
fixes *oid2* :: *nat*

fixes *oid3* :: *nat*
fixes *oid4* :: *nat*
fixes *oid5* :: *nat*
fixes *oid6* :: *nat*
fixes *oid7* :: *nat*
assumes *distinct-oid*: (*distinct* ([*oid1* , *oid2* , *oid3* , *oid4* , *oid5* , *oid6* , *oid7*]))
fixes $X_{Person}1_{Person}$ :: ty_{Person}
fixes $X_{Person}1$ :: ·*Person*
assumes $X_{Person}1$-*def*: $X_{Person}1 = (\lambda\text{-}.\ \lfloor\lfloor X_{Person}1_{Person}\rfloor\rfloor)$
fixes $X_{Person}2_{Person}$ :: ty_{Person}
fixes $X_{Person}2$ :: ·*Person*
assumes $X_{Person}2$-*def*: $X_{Person}2 = (\lambda\text{-}.\ \lfloor\lfloor X_{Person}2_{Person}\rfloor\rfloor)$
fixes $X_{Person}3_{Person}$ :: ty_{Person}
fixes $X_{Person}3$ :: ·*Person*
assumes $X_{Person}3$-*def*: $X_{Person}3 = (\lambda\text{-}.\ \lfloor\lfloor X_{Person}3_{Person}\rfloor\rfloor)$
fixes σ_1-$object0_{Person}$ :: ty_{Person}
fixes σ_1-*object0* :: ·*Person*
assumes σ_1-*object0-def*: σ_1-*object0* $= (\lambda\text{-}.\ \lfloor\lfloor \sigma_1\text{-}object0_{Person}\rfloor\rfloor)$
fixes σ_1-$object1_{Person}$ :: ty_{Person}
fixes σ_1-*object1* :: ·*Person*
assumes σ_1-*object1-def*: σ_1-*object1* $= (\lambda\text{-}.\ \lfloor\lfloor \sigma_1\text{-}object1_{Person}\rfloor\rfloor)$
fixes σ_1-$object2_{Person}$ :: ty_{Person}
fixes σ_1-*object2* :: ·*Person*
assumes σ_1-*object2-def*: σ_1-*object2* $= (\lambda\text{-}.\ \lfloor\lfloor \sigma_1\text{-}object2_{Person}\rfloor\rfloor)$
fixes σ_1-$object4_{Person}$ :: ty_{Person}
fixes σ_1-*object4* :: ·*Person*
assumes σ_1-*object4-def*: σ_1-*object4* $= (\lambda\text{-}.\ \lfloor\lfloor \sigma_1\text{-}object4_{Person}\rfloor\rfloor)$

assumes σ_1: (*state-*σ_1 (*oid4*) (*oid5*) (*oid6*) (*oid1*) (*oid7*) (*oid2*) (σ_1-$object0_{Person}$) (σ_1-*object0*) (σ_1-$object1_{Person}$) (σ_1-*object1*) (σ_1-$object2_{Person}$) (σ_1-*object2*) ($X_{Person}1_{Person}$) ($X_{Person}1$) (σ_1-$object4_{Person}$) (σ_1-*object4*) ($X_{Person}2_{Person}$) ($X_{Person}2$))

assumes $\sigma_1{}'$: (*state-*$\sigma_1{}'$ (*oid1*) (*oid2*) (*oid3*) ($X_{Person}1_{Person}$) ($X_{Person}1$) ($X_{Person}2_{Person}$) ($X_{Person}2$) ($X_{Person}3_{Person}$) ($X_{Person}3$))
begin
interpretation *state-*σ_1: *state-*σ_1 *oid4 oid5 oid6 oid1 oid7 oid2* σ_1-$object0_{Person}$ σ_1-*object0* σ_1-$object1_{Person}$ σ_1-*object1* σ_1-$object2_{Person}$ σ_1-*object2* $X_{Person}1_{Person}$ $X_{Person}1$ σ_1-$object4_{Person}$ σ_1-*object4* $X_{Person}2_{Person}$ $X_{Person}2$
by(*rule* σ_1)

interpretation *state-*$\sigma_1{}'$: *state-*$\sigma_1{}'$ *oid1 oid2 oid3* $X_{Person}1_{Person}$ $X_{Person}1$ $X_{Person}2_{Person}$ $X_{Person}2$ $X_{Person}3_{Person}$ $X_{Person}3$
by(*rule* $\sigma_1{}'$)

definition *heap-*σ_1 = (*heap* (*state-*σ_1.σ_1))

definition *heap-*$\sigma_1{}'$ = (*heap* (*state-*$\sigma_1{}'$.$\sigma_1{}'$))
end

end