A Meta-Model for the Isabelle API

Frédéric Tuong  Burkhart Wolff

September 28, 2015

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

IRT SystemX
8 av. de la Vauve, 91120 Palaiseau, France
frédéric.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 any languages that can be described with a sequence of “datatype” in HOL itself; the Isabelle code-generator can then be used to generate 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

## 1 A Meta-Model for the Isabelle API

1. Initialization  
   1.1 Optimization on the String Datatype  
   1.2 Basic Extension of the Standard Library  
      1.2.1 Polymorphic Cartouches  
      1.2.2 Operations on List  
      1.2.3 Operations on Char  
      1.2.4 Operations on String (I)  
      1.2.5 Operations on String (II)  
   1.3 Miscellaneous

2 Defining Meta-Models
   2.1 (Pure) Term Meta-Model aka. AST definition of (Pure) Term  
      2.1.1 Type Definition  
      2.1.2 Operations of Fold, Map, ..., on the Meta-Model  
   2.2 SML Meta-Model aka. AST definition of SML  
      2.2.1 Type Definition  
      2.2.2 Extending the Meta-Model  
   2.3 Isabelle Meta-Model aka. AST definition of Isabelle  
      2.3.1 Type Definition  
      2.3.2 Extending the Meta-Model  
      2.3.3 Operations of Fold, Map, ..., on the Meta-Model

3 Parsing Meta-Models
   3.1 Initializing the Parser  
      3.1.1 Some Generic Combinators  
      3.1.2 Generic Locale for Parsing  
   3.2 Instantiating the Parser of (Pure) Term  
      3.2.1 Main

4 Printing Meta-Models
   4.1 Initializing the Printer  
      4.1.1 Kernel Code for Target Languages  
      4.1.2 Interface with Types  
      4.1.3 Interface with Constants
Part I.

A Meta-Model for the Isabelle API
1. Initialization

theory Init
imports ~/src/HOL/Library/Code-Char
    isabelle-home/src/HOL/Isabelle-Main0
begin

1.1. Optimization on the String Datatype

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

type-notation natural (nat)
definition Succ x = x + 1
datatype stringbase = ST String.literal
    | ST' char list
datatype abr-string =
    SS-base stringbase
    | String-concatWith abr-string abr-string list

syntax -string1 :: - ⇒ abr-string (((-)))
translations ⟨x⟩ ⇒ CONST SS-base (CONST ST (CONST STR x))

syntax -string2 :: - ⇒ String.literal (()())
translations «x» ⇒ CONST STR x

syntax -string3 :: - ⇒ abr-string ((-))
translations «x» ⇒ CONST SS-base (CONST ST' x)

syntax -char1 :: - ⇒ 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

\[
\text{val cartouche-grammar =}
\begin{array}{l}
\text{[ (char list, snd)}
\text{, (String.literal, (fn (-, x) => Syntax.const @\{\text{const-syntax STR}\} x))}
\text{, (abr-string, (fn (-, x) => Syntax.const @\{\text{const-syntax SS-base}\})}
\text{\$ (Syntax.const @\{\text{const-syntax ST}\}}
\text{\$ x))]))
\end{array}
\]

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...

\[\text{declare[[cartouche-type = abr-string]]}\]

1.2.2. Operations on List

datatype (‘a, ‘b) nsplit = Nsplit-text ‘a
locale L
begin
definition map where map f l = rev (foldl (λl x. f x # l) [] l)
definition flatten l = foldl (λacc l. foldl (λacc 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 (λl x. if f x then x :: l else l) [] l)
definition rev-map f = foldl (λl x. f x # l) []
definition mapM f l accu =
  (let (l, accu) = List.fold (λx (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). λNone ⇒ 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 = λn. 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 = λx. ¬ f x in
    (takeWhile f l, case dropWhile f l of [] ⇒ (None, []) | x # xs ⇒ (Some x, xs)))
definition take where take reverse ly l = reverse (snd (L.split (takeWhile (λ(n, -). n < ly)
    (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 = λl 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-t 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] =

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

L.map-find-aux.simps

1.2.3. Operations on Char

definition char-escape = Char Nibble0 Nibble9
definition ST0 c = ≪[c]≫
definition ST0-base c = ST′ [c]

1.2.4. Operations on String (I)

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 s = foldl f accu (String.explode s)
definition (in Stringbase) foldl where foldl f accu = (\ ST s \Rightarrow String.foldl-one f accu s \\
| ST' s \Rightarrow List.foldl f accu s)

fun (in String) foldl where
foldl f accu e =
  (\ SS-base s \Rightarrow Stringbase.foldl f accu s \\
   | String-concatWith abr l \Rightarrow \\
     (case l of [] \Rightarrow accu \\
      | x \# xs \Rightarrow List.foldl (\accu. foldl f (foldl f accu abr)) (foldl f accu x) xs)) e
definition (in S) replace-chars f s1 s s2 =
  s1 @@ (case s of None \Rightarrow \nothing | Some s \Rightarrow flatten (L.map f (String.explode s))) @@ s2
definition (in String) map where map f = map-gen (S.replace-chars (\c. *f c*)) (\x. *f x*)
definition (in String) replace-chars f = map-gen (S.replace-chars (\c. f c)) f
definition (in String) all f = foldl (\b s. b \& f s) True
definition (in String) length where length = foldl (\n -. Suc n) 0
definition (in String) to-list s = rev (foldl (\l c. c \# l) [] s)
definition (in Stringbase) to-list = (\ ST s \Rightarrow String.explode s | ST' l \Rightarrow l)
definition (in String) to-Stringbase = (\ SS-base s \Rightarrow s | s \Rightarrow ST' (to-list s))
definition (in Stringbase) to-String = SS-base
definition (in Stringbase) is-empty = (\ ST s \Rightarrow s = STR """
  | ST' s \Rightarrow s = [])
fun (in String) is-empty where
  is-empty e = (\ SS-base s \Rightarrow Stringbase.is-empty s | String-concatWith - l \Rightarrow list-all is-empty l) e
definition (in String) equal s1 s2 = (to-list s1 = to-list s2)
notation String.equal (infixl \triangleq 50)
definition (in String) assoc x l = L.assoc (to-list x) (L.map (map-prod Stringbase.to-list id) l)
definition (in String) member l x = List.member (L.map Stringbase.to-list l) (to-list x)
definition (in Stringbase) flatten l = String.to-Stringbase (S.flatten (L.map to-String l))
1.2.5. Operations on String (II)

definition wildcard = (-)

definition nat-raw-to-str = L.map (λi. char-of-nat (nat-of-char (Char Nibble3 Nibble0) + i))

class String

begin

definition lowercase = map (λc. let n = nat-of-char c in if n < 97 then char-of-nat (n + 32) else c)

end

lemmas [code] =

String.lowercase-def
String.uppercase-def
String.to-bold-number-def
String.of-nat-def
String.of-natural-def

String.of-nat-aux.simps

definition add-0 n = (let n = nat-of-char n in
  S.flatten (L.map (λc. 0) (upt 0 (if n < 10 then 2 else if n < 100 then 1 else 0))))

context String

begin

definition is-letter n = (n ≥ CHR "A" & n ≤ CHR "Z" | n ≥ CHR "a" & n ≤ CHR "z")

definition is-digit n = (n ≥ CHR "0" & n ≤ CHR "9")

definition is-special = List.member "<>~-=./{ }

context String

begin

definition base255 = replace-chars (λc. if is-letter c then "c" else add-0 c)

16
definition isub = replace-chars (λc.
  if is-letter c | is-digit c | List.member "-" c then c else add-0 c)
definition isup s = (⋯) @ @ s
definition text-of-str str =
  (let s = (⋯)
  ; ap = (⋯)
  , String.replace-chars (λc.
    if is-letter c then
      S.flatten [CHR "", "", ap]
    else
      S.flatten [s, (⋯), add-0 c, ap])
  str
  , (⋯))]
definition' text2-of-str = String.replace-chars (λc. S.flatten [\", (⋯), "", (⋯)])
definition textstr-of-str f-flatten f-char f-str str =
  (let str0 = String.toList str
  ; f-letter = λc. is-letter c | is-digit c | is-special c
  ; s = (⋯)
  ; f-text = λ Nsplit-text l ⇒ S.flatten [f-str (S.flatten [STR "", "", (⋯)])]
  | Nsplit-sep c ⇒ S.flatten [f-char c]
  ; str = case L.nsplit-f str0 (Not o f-letter) of
    [] ⇒ S.flatten [f-str (STR ""]
  | l ⇒ S.flatten (L.map (λx. (⋯) @ @ f-text x @ @ (⋯) @ @ l) @ @ (⋯))
  in
if list-all f-letter str0 then
  str
  else
    f-flatten (S.flatten [⋯])
definition' escape-sml = String.replace-chars ((⋯) ERROR code-reflect *)
  λ Char Nibble2 Nibble2 ⇒ (\x ⇒ x ⇒ "x")
  λx. if x = Char Nibble2 Nibble2 then (\x ⇒ "x")
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 [⋯]
definition mk-dot-comment s1 s2 s3 = mk-dot s1 (S.flatten [s2, (⋯), s3, (⋯)])
definition hol-definition \( s = S.\text{flatten}[s, \cdot \text{-def}] \)
definition hol-split \( s = S.\text{flatten}[s, \cdot \text{-split}] \)

1.3. Miscellaneous

Syntactic errors in target languages can appear during extraction, so we explicitly output parenthesis around ambiguous expressions (by enclosing them in a `id` scope for instance).

**syntax** -Let\(OCaml:: [\text{letbinds}, 'a] \Rightarrow 'a ((\text{letOCaml} (-)/ \text{ in} (-)) [0, 10] 10)\)
**translations** -Let\(OCaml (-\text{binds} b bs) e \Rightarrow -\text{LetOCaml} b (-\text{Let} bs e)\)

\[
\text{letOCaml} x = a \text{ in } e = \text{CONST id (CONST Let } a (%x. e))
\]

**syntax** -\text{case-syntax}\(OCaml:: [‘a, cases-syn] \Rightarrow ‘b ((\text{caseOCaml} - \text{ of} / \cdot) 10)\)
**translations** case\(OCaml v \text{ of } w => x = \text{CONST id (-case-syntax } v (-\text{case1 } w \text{ x}))\)

\[
\text{caseOCaml} v \text{ of } w => x | y => z = \text{CONST id (-case-syntax } v (-\text{case2 } (-\text{case1 } w \text{ x}) (-\text{case1 } y \text{ z})))
\]

**syntax** -\text{case-syntax}\(Scala:: [‘a, cases-syn] \Rightarrow ‘b ((\text{caseScala} - \text{ of} / \cdot) 10)\)
**translations** case\(Scala v \text{ of } w => x = \text{CONST id (-case-syntax } v (-\text{case1 } w \text{ x}))\)

\[
\text{caseScala} v \text{ of } w => x | y => z = \text{CONST id (-case-syntax } v (-\text{case2 } (-\text{case1 } w \text{ x}) (-\text{case1 } y \text{ z})))
\]

end
2. Defining Meta-Models

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

type 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))

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)
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

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' string semi-term'
     | SML-basic string list
     | SML-binop semi-term' string semi-term'
     | SML-annot semi-term' string
     | SML-function (semi-term' (* pattern *) × semi-term' (* to return *)) list
     | SML-apply semi-term' semi-term' list
     | SML-paren string string 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 (λx. 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] =
  SML.string-def
  SML.rewrite-def
  SML.basic-def
  SML.binop-def
  SML.annotate-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
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.

```haskell
datatype semi--typ = Typ-apply semi--typ semi--typ list
    | Typ-apply-bin string semi--typ semi--typ
    | Typ-apply-paren string string semi--typ
    | Typ-base string

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

datatype type-synonym = Type-synonym string
    string list
    semi--typ

datatype semi--term = Term-rewrite semi--term string semi--term
    | Term-basic string list
    | Term-annot semi--term semi--typ
    | Term-bind string semi--term semi--term
    | Term-fun-case semi--term (* value *) option (semi--term (* pattern *) × semi--term (* to return *)) list
    | Term-apply semi--term semi--term list
    | Term-paren string string semi--term
    | Term-if-then-else semi--term semi--term semi--term semi--term
    | Term-term string list
    term

datatype type-notation = Type-notation string
    string

datatype instantiation = Instantiation string
    string
    semi--term

datatype defs =Defs-overloaded string semi--term

datatype consts = Consts string
    semi--typ
    string

datatype definition = Definition semi--term
    | Definition-where1 string semi--term (* syntax extension *) × nat (* priority *) semi--term
```

begin
Definition-where

**datatype** semi-thm-attribute = Thm-thm string
| Thm-thms string
| 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
| string
| semi-thm-attribute list
| Lemmas-simp-thms string
| string (∗ thms ∗) list

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

**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 list
| Method-subst bool
| string (∗ nat ∗) list
| 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
| 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

datatype semi--command-proof = Command-apply semi--method list
| Command-using semi--thm-l
| Command-unfolding semi--thm-l
| Command-let semi--term semi--term
| Command-have string
  bool
  semi--term
  semi--command-final
| Command-fix-let string list
  (semi--term (* name *) × semi--term) list
  semi--term list option
  semi--command-state list

datatype lemma = Lemma string semi--term list
  semi--method list list
  semi--command-final
| Lemma-assumes string
  (string (* name *) × bool (* true: add [simp] *) × semi--term) list
  semi--term
  semi--command-proof list
  semi--command-final

datatype axiomatization = Axiomatization string
  semi--term

datatype section = Section nat
  string

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
  string
  semi--term list
  semi--command-final

datatype semi--theory = Theory-datatype datatype
| Theory-type-synonym type-synonym
| Theory-type-notation type-notation
| Theory-instantiation instantiation
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 (λx acc. Thm-OF acc x) l s

definition simplified-l s l = List.fold (λx acc. Thm-simplified acc x) l s
end

lemmas [code] =

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-app (Typ-base (option)) [Typ-base s]
definition Raw = Typ-base
definition Type-synonym' n = Type-synonym n []
definition Type-synonym'' n f = Type-synonym n l (f l)
definition Term-annot' e s = Term-annot e (Typ-base s)
definition Term-lambdas s = Term-bind (\x) (Typ-base s)
definition Term-lambda x = Term-lambdas [x]
definition Term-lambdas0 = Term-bind (\)
definition Term-lam x f = Term-lambdas0 (Typ-base [x]) (f x)
definition Term-some = Term-paren (\t \n)
definition Term-parenthesis (\ mandatory parenthesis \) = Term-paren (\f \n)
definition Term-warn-parenthesis (\ optional parenthesis that can be removed but a warning will be raised \) = Term-parenthesis

definition Term-term = Term-preunary e1 e2
definition Term-app e = Term-apps0 (Typ-base [e])
definition Term-preunary e1 e2 = Term-apps0 [\] (e1 [\] e2)
definition Term-postunary e1 e2 = Term-apps0 [e1] [e2]
definition Term-case = Term-fun-case o Some

definition Term-case = Term-fun-case None

definition Term-term' = Term-term []
definition Lemmas-simp = Lemmas-simp-thm True

definition Lemmas-nosimp = Lemmas-simp-thm False

definition Cons value = (\)
definition Cons-rws0 s l o-arg = Cons s l (String.replace-chars (\c. if c = Char Nibble5 NibbleF then \n else "c") e \@\@ (case o-arg of
   None => \n | Some arg =>
    let ap = \ls. (\t \@\@ s \@\@ \n) \n in
    ap (if arg = 0 then

26
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 Consts' s l e = Consts-raw0 s (Ty-arrow (Typ-base ⟨α⟩ l) e None

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 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 = 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) 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)
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 metis = Method-metis
definition metis0 = Method-metis

definition subst-asn b = subst-l0 b [[]]
definition subst = subst-l [[]]
definition auto-simp-add = auto-simp-add2 []
definition auto = auto-simp-add []
end

lemmas [code] =
M.Method-simp-add-del-def
M.Method-subst-l-def
M.rule'-def
M.rule-def
M.drule-def
Merule-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
\begin{verbatim}
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] =
  C.done-def
  C.by-def
  C.sorry-def
  C.apply-end-def
  C.apply-def
  C.using-def
  C.unfolding-def
\end{verbatim}
fun cross-abs-aux where
    cross-abs-aux f l x = (\(Suc\ n\), Abs s - t) \Rightarrow f s (cross-abs-aux f (s ≠ l) (n, t))
    \| (-, e) \Rightarrow Term-term l e

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 = (\(\lambda\ Theory-lemma x \Rightarrow Theory-lemma (f x)\)
    \| x \Rightarrow 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 $\circ_1 = \circ$
definition $\circ_2 f g x_1 x_2 = f (g x_1 x_2)$
definition $\circ_3 f g x_1 x_2 x_3 = f (g x_1 x_2 x_3)$
definition $\circ_4 f g x_1 x_2 x_3 x_4 = f (g x_1 x_2 x_3 x_4)$
definition $\circ_5 f g x_1 x_2 x_3 x_4 x_5 = f (g x_1 x_2 x_3 x_4 x_5)$
definition $\circ_6 f g x_1 x_2 x_3 x_4 x_5 x_6 = f (g x_1 x_2 x_3 x_4 x_5 x_6)$
definition $\circ_7 f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7)$
definition $\circ_8 f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8)$
definition $\circ_9 f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9)$
definition $\circ_{10} f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10})$
definition $\circ_{11} f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11})$
definition $\circ_{12} f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12})$
definition $\circ_{13} f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13})$
definition $\circ_{14} f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14})$
definition $\circ_{15} f g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} = f (g x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15})$
definition $\circ_{16} a v_0 [f_1 v_1, f_2 v_2]$
definition $\circ_{17} a v_0 [f_1 f_2, f_3 v_3]$
definition $\circ_{18} a v_0 [f_1 f_2 f_3, f_4 v_4]$
definition $\circ_{19} a v_0 [f_1 f_2 f_3 f_4, f_5 v_5]$
definition $\circ_{20} a v_0 [f_1 f_2 f_3 f_4 f_5, f_6 v_6]$
definition $\circ_{21} a v_0 [f_1 f_2 f_3 f_4 f_5 f_6 f_7 v_7]$
\[ f_{4} v_{4}, f_{5} v_{5}, f_{6} v_{6}, f_{7} v_{7}, f_{8} v_{8} \]

**Definition** 
\[ a_{9} v_{0} f_{1} f_{2} f_{3} f_{4} f_{5} f_{6} f_{7} f_{8} f_{9} v_{1} v_{2} v_{3} v_{4} v_{5} v_{6} v_{7} v_{8} v_{9} = a_{0} v_{0} \]
\[ f_{1} v_{1}, f_{2} v_{2}, f_{3} v_{3}, f_{4} v_{4}, f_{5} v_{5}, f_{6} v_{6}, f_{7} v_{7}, f_{8} v_{8}, f_{9} v_{9}, f_{10} v_{10} \]

**Definition** 
\[ a_{10} v_{0} f_{1} f_{2} f_{3} f_{4} f_{5} f_{6} f_{7} f_{8} f_{9} f_{10} f_{11} f_{12} v_{1} v_{2} v_{3} v_{4} v_{5} v_{6} v_{7} v_{8} v_{9} v_{10} = a_{0} v_{0} \]
\[ f_{1} v_{1}, f_{2} v_{2}, f_{3} v_{3}, f_{4} v_{4}, f_{5} v_{5}, f_{6} v_{6}, f_{7} v_{7}, f_{8} v_{8}, f_{9} v_{9}, f_{10} v_{10}, f_{11} v_{11}, f_{12} v_{12} \]

**Definition** 
\[ a_{11} v_{0} f_{1} f_{2} f_{3} f_{4} f_{5} f_{6} f_{7} f_{8} f_{9} f_{10} f_{11} f_{12} f_{13} v_{1} v_{2} v_{3} v_{4} v_{5} v_{6} v_{7} v_{8} v_{9} v_{10} v_{11} v_{12} v_{13} v_{14} v_{15} = a_{0} v_{0} \]
\[ f_{1} v_{1}, f_{2} v_{2}, f_{3} v_{3}, f_{4} v_{4}, f_{5} v_{5}, f_{6} v_{6}, f_{7} v_{7}, f_{8} v_{8}, f_{9} v_{9}, f_{10} v_{10}, f_{11} v_{11}, f_{12} v_{12}, f_{13} v_{13}, f_{14} v_{14} \]

**Definition** 
\[ a_{12} v_{0} f_{1} f_{2} f_{3} f_{4} f_{5} f_{6} f_{7} f_{8} f_{9} f_{10} f_{11} f_{12} f_{13} f_{14} v_{1} v_{2} v_{3} v_{4} v_{5} v_{6} v_{7} v_{8} v_{9} v_{10} v_{11} v_{12} v_{13} v_{14} = a_{0} v_{0} \]
\[ f_{1} v_{1}, f_{2} v_{2}, f_{3} v_{3}, f_{4} v_{4}, f_{5} v_{5}, f_{6} v_{6}, f_{7} v_{7}, f_{8} v_{8}, f_{9} v_{9}, f_{10} v_{10}, f_{11} v_{11}, f_{12} v_{12}, f_{13} v_{13}, f_{14} v_{14} \]

**Definition** 
\[ a_{13} v_{0} f_{1} f_{2} f_{3} f_{4} f_{5} f_{6} f_{7} f_{8} f_{9} f_{10} f_{11} f_{12} f_{13} v_{1} v_{2} v_{3} v_{4} v_{5} v_{6} v_{7} v_{8} v_{9} v_{10} v_{11} v_{12} v_{13} v_{14} = a_{0} v_{0} \]
\[ f_{1} v_{1}, f_{2} v_{2}, f_{3} v_{3}, f_{4} v_{4}, f_{5} v_{5}, f_{6} v_{6}, f_{7} v_{7}, f_{8} v_{8}, f_{9} v_{9}, f_{10} v_{10}, f_{11} v_{11}, f_{12} v_{12} \]

**Definition** 
\[ a_{14} v_{0} f_{1} f_{2} f_{3} f_{4} f_{5} f_{6} f_{7} f_{8} f_{9} f_{10} f_{11} f_{12} f_{13} f_{14} v_{1} v_{2} v_{3} v_{4} v_{5} v_{6} v_{7} v_{8} v_{9} v_{10} v_{11} v_{12} v_{13} v_{14} = a_{0} v_{0} \]
\[ f_{1} v_{1}, f_{2} v_{2}, f_{3} v_{3}, f_{4} v_{4}, f_{5} v_{5}, f_{6} v_{6}, f_{7} v_{7}, f_{8} v_{8}, f_{9} v_{9}, f_{10} v_{10}, f_{11} v_{11}, f_{12} v_{12}, f_{13} v_{13}, f_{14} v_{14}, f_{15} v_{15} \]
3.1.2. Generic Locale for Parsing

locale Parse =
  fixes ext :: string ⇒ string

  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

  fixes Of-Pair Of-Nil Of-Cons Of-None Of-Some :: string

begin

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

definition of-list a b f = (λf0. 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 \( a \ b = \text{of-pair} \ a \ b \ \text{(of-string} \ a \ b) \ \text{(of-nat} \ a \ b) \)

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

**definition** of-pure-sort \( a \ b = \text{of-list} \ a \ b \ \text{(of-pure-class} \ a \ b) \)

**definition** of-pure-typ \( a \ b = \text{rec-typ} \)
\( \quad (\text{ap2} \ a \ (b \ : \text{:Type})) \ \text{(of-string} \ a \ b) \ \text{(of-list} \ a \ b \ \text{snd)}) \)
\( \quad (\text{ap2} \ a \ (b \ : \text{:TFree})) \ \text{(of-string} \ a \ b) \ \text{(of-pure-sort} \ a \ b) \)
\( \quad (\text{ap2} \ a \ (b \ : \text{:TVar})) \ \text{(of-pure-indexname} \ a \ b) \ \text{(of-pure-sort} \ a \ b) \)

**definition** of-pure-term \( a \ b = (\lambda f0 \ f1 \ f2 \ f3 \ f4 \ f5. \ \text{rec-term} \ f0 \ f1 \ f2 \ f3 \ (\text{co2} \ K \ f4) \ (\lambda - \ -. \ f5)) \)
\( \quad (\text{ap2} \ a \ (b \ : \text{:Const})) \ \text{(of-string} \ a \ b) \ \text{(of-pure-typ} \ a \ b) \)
\( \quad (\text{ap2} \ a \ (b \ : \text{:Free})) \ \text{(of-string} \ a \ b) \ \text{(of-pure-typ} \ a \ b) \)
\( \quad (\text{ap2} \ a \ (b \ : \text{:Var})) \ \text{(of-pure-indexname} \ a \ b) \ \text{(of-pure-typ} \ a \ b) \)
\( \quad (\text{ap1} \ a \ (b \ : \text{:Bound})) \ \text{(of-nat} \ a \ b) \)
\( \quad (\text{ar3} \ a \ (b \ : \text{:Abs})) \ \text{(of-string} \ a \ b) \ \text{(of-pure-typ} \ a \ b) \)
\( \quad (\text{ar2} \ a \ (b \ : \text{:App}) \ \text{id}) \)

end

**lemmas** [code] =
\( \text{Parse.of-pure-indexname-def} \)
\( \text{Parse.of-pure-class-def} \)
\( \text{Parse.of-pure-sort-def} \)
\( \text{Parse.of-pure-typ-def} \)
\( \text{Parse.of-pure-term-def} \)

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) :
  type MInt = Integer
  type MMonad a = IO a
  | code-module CodeConst → (Haskell) :
      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)

  | outFile1 :: ((String -> String -> IO ()) -> IO ()) -> IO ()
  | outFile1 f = f (\pat -> putStrLn . CodeConst_Printf.sprintf1 pat)
  | code-module CodeConst.Monad → (Haskell) :
      bind a = (>>=) a
  | return :: a -> IO a
  | return = Prelude.return
  | code-module CodeConst_Printf → (Haskell) :
      import Text.Printf
  | printf0 = id

  | printf1 :: PrintfArg a => String -> a -> String
  | printf1 = printf

  | printf2 :: 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) /
  | concat s [] = []
| code-module CodeConst.Sys -> (Haskell) /
  | isDirectory2 = doesDirectoryExist
| code-module CodeConst.To -> (Haskell) /
  | 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.cprintf File exists \"%S\n\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)
end

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

⟩

| 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) |
| if (file.isFile) { |
|   None |
| } else { |
|   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

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

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 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
SOME (File.write-list pfile (rev (Unsynchronized! oc))) handle - => NONE
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
  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
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
|
code-printing constant out-file1 → (OCaml) CodeConst.outFile1
|
code-printing constant out-file1 → (Scala) CodeConst.outFile1
|
code-printing 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
|
code-printing constant out-stand1 → (OCaml) CodeConst.outStand1
|
code-printing constant out-stand1 → (Scala) CodeConst.outStand1
|
code-printing 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
|
code-printing constant bind → (OCaml) CodeConst.Monad.bind
|
code-printing constant bind → (Scala) CodeConst.Monad.bind
|
code-printing constant bind → (SML) CodeConst.Monad.bind

consts return :: 'a ⇒ 'a ml-monad

code-printing constant return → (Haskell) CodeConst.Monad.return
|
code-printing constant return → (OCaml) CodeConst.Monad.return
|
code-printing constant return → (Scala) CodeConst.Monad.Return
|
code-printing constant return → (SML) CodeConst.Monad.return

module Printf

consts sprintf0 :: ml-string ⇒ ml-string

code-printing constant sprintf0 → (Haskell) CodeConstPrintf sprintf0
|
code-printing constant sprintf0 → (OCaml) CodeConstPrintf sprintf0
|
code-printing constant sprintf0 → (Scala) CodeConstPrintf sprintf0
|
code-printing constant sprintf0 → (SML) CodeConstPrintf sprintf0

consts sprintf1 :: ml-string ⇒ 'α1 ⇒ ml-string

code-printing constant sprintf1 → (Haskell) CodeConstPrintf sprintf1
|
code-printing constant sprintf1 → (OCaml) CodeConstPrintf sprintf1

40
| constant \( \text{sprintf1} \) \rightarrow (\text{Scala}) \text{CodeConst.Printf.sprintf1} |
| constant \( \text{sprintf1} \) \rightarrow (\text{SML}) \text{CodeConst.Printf.sprintf1} |

\text{consts} \( \text{sprintf2} :: \text{ml-string} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \text{ml-string} \)

\text{consts} \( \text{sprintf3} :: \text{ml-string} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \text{ml-string} \)

\text{consts} \( \text{sprintf4} :: \text{ml-string} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \text{ml-string} \)

\text{consts} \( \text{sprintf5} :: \text{ml-string} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \text{ml-string} \)

\text{module} \text{String} |
| \text{consts} \text{String-concat} :: \text{ml-string} \Rightarrow \text{ml-string list} \Rightarrow \text{ml-string} |
| \text{code-printing constant} \text{String-concat} :: \text{ml-string} \Rightarrow \text{ml-string list} \Rightarrow \text{ml-string} |

\text{module} \text{Sys} |
| \text{consts} \text{Sys-is-directory2} :: \text{ml-string} \Rightarrow \text{bool} \Rightarrow \text{ml-monad} |
| \text{code-printing constant} \text{Sys-is-directory2} :: \text{ml-string} \Rightarrow \text{bool} \Rightarrow \text{ml-monad} |

\text{module} \text{To} |
| \text{consts} \text{ToNat} :: (\text{nat} \Rightarrow \text{integer}) \Rightarrow \text{nat} \Rightarrow \text{ml-int} |
| \text{code-printing constant} \text{ToNat} :: (\text{nat} \Rightarrow \text{integer}) \Rightarrow \text{nat} \Rightarrow \text{ml-int} |
4.1.4. Some Notations (I): Raw Translations

\[
\text{syntax } \text{-sprint0 ::= } - \Rightarrow \text{ml-string } (\text{sprint0 } (-))
\]
\[
\text{translations } \text{sprint0 } x' \rightleftharpoons \text{CONST sprintf0 } x
\]

\[
\text{syntax } \text{-sprint1 ::= } - \Rightarrow - \Rightarrow \text{ml-string } (\text{sprint1 } (-))
\]
\[
\text{translations } \text{sprint1 } x' \rightleftharpoons \text{CONST sprintf1 } x
\]

\[
\text{syntax } \text{-sprint2 ::= } - \Rightarrow - \Rightarrow \text{ml-string } (\text{sprint2 } (-))
\]
\[
\text{translations } \text{sprint2 } x' \rightleftharpoons \text{CONST sprintf2 } x
\]

\[
\text{syntax } \text{-sprint3 ::= } - \Rightarrow - \Rightarrow \text{ml-string } (\text{sprint3 } (-))
\]
\[
\text{translations } \text{sprint3 } x' \rightleftharpoons \text{CONST sprintf3 } x
\]

\[
\text{syntax } \text{-sprint4 ::= } - \Rightarrow - \Rightarrow \text{ml-string } (\text{sprint4 } (-))
\]
\[
\text{translations } \text{sprint4 } x' \rightleftharpoons \text{CONST sprintf4 } x
\]

\[
\text{syntax } \text{-sprint5 ::= } - \Rightarrow - \Rightarrow \text{ml-string } (\text{sprint5 } (-))
\]
\[
\text{translations } \text{sprint5 } x' \rightleftharpoons \text{CONST sprintf5 } x
\]

4.1.5. Some Notations (II): Polymorphic Cartouches

\[
\text{syntax } \text{-cartouche-string' ::= String.literal}
\]
\[
\text{translations } \text{-cartouche-string} \rightleftharpoons \text{-cartouche-string}'
\]

parse-translation ( [
\begin{align*}
\text{[} &\text{@[\{syntax-const -cartouche-string\}]} \\
\text{, } &\text{parse-translation-cartouche} \\
\text{\@{binding cartouche-type'}} \\
\text{\{fun printf} \\
\text{\, let fun f x = Syntax.const \@{const-syntax STR} $ x} \\
\text{\quad fun f' c x = Syntax.const c $ f x in} \\
\text{\quad fn (0, x) \Rightarrow f x} \\
\text{\quad | (1, x) \Rightarrow f' \@{const-syntax sprintf1} x} \\
\text{\quad | (2, x) \Rightarrow f' \@{const-syntax sprintf2} x} \\
\text{\quad | (3, x) \Rightarrow f' \@{const-syntax sprintf3} x} \\
\text{\quad | (4, x) \Rightarrow f' \@{const-syntax sprintf4} x} \\
\text{\quad | (5, x) \Rightarrow f' \@{const-syntax sprintf5} x} \\
\text{\end{align*}
\end{verbatim}
\text{\} : cartouche-grammar)}
\text{(fn \text{37 (* #% *)} \Rightarrow (fn x => x + 1)}
\text{\quad | \text{-} \Rightarrow I)}
\text{\quad | \text{0}]}
\]

4.1.6. Generic Locale for Printing

locale Print =
\text{fixes } To-string :: string \Rightarrow ml-string
\text{fixes } To-nat :: nat \Rightarrow ml-int
As remark, printing functions (like `sprintf5...`) are currently weakly typed in Isabelle, we will continue the typing using the type system of target languages.

4.2. Instantiating the Printer for (Pure) Term

```isabelle
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] =

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 af-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 of-semi-term' l))
| SML-apply e l ⇒ (⟨s %s⟩) (of-semi-term' e) (String-concat ⟨⟩ (List.map (λ e ⇒ ⟨s %s⟩) (of-semi-term' e) l))
| SML-let-open s e ⇒ (⟨let open %s in %s end⟩) (To-string s) (of-semi-term' e) e)
end

lemmas [code] =

Print.of-semi-val-fun-def

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 c = (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) 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)
definition of-datatype _ = (λ Datatype n l ⇒
  (datatype %s = %s) (To-string n)

44
(String-concat (L.map (λ(n,l). (To-string n) (String-concat (L.map (λx. (To-string x) (af-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 ⇒ (if e1 then %s else of-semi-term e1) (of-semi-term e2)
| Term-basic l ⇒ %s (String-concat (L.map To-string l))
| Term-annot e s ⇒ (%s, %) (of-semi-term e) (of-semi-typ s)
| Term-bind symb e1 e2 ⇒ (%s, %) (of-semi-term e1) (of-semi-term e2)
| Term-fun-case e-case l ⇒ (af-semi-term (Typ-apply (Typ-base n) (L.map Typ-base v))) (of-semi-term l))
| Term-apply e l ⇒ %s (of-semi-term e) (af-semi-term l)
| Term-paren p-left p-right e ⇒ %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 object begin definition %s-%s-def : %s instance .. end name (To-string n-def) name (of-semi-term expr))
definition of-defs - = (λDefs-overloaded n e ⇒
(defs-overloaded) %s :: %s; (To-string n) (of-semi--term e))

definition of-consts - = (λConsts 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 - = (λDefinition 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)
where %s (To-string name) (of-semi--term abbrev) (of-semi--term e))

| Definition-where2 name abbrev e ⇒ ⟨definition %s %d
where %s (To-string name) (of-semi--term abbrev) (of-semi--term e))

fun of-semi--thm-attribute-aux where of-semi--thm-attribute-aux lacc e =
(λThm-thm s ⇒ To-string s
| Thm-thms s ⇒ To-string s
| Thm-THEN (Thm-thm s) e2 ⇒ of-semi--thm-attribute-aux %d e2 lacc s
| Thm-THEN (Thm-thms s) e2 ⇒ of-semi--thm-attribute-aux %d e2 lacc s
| Thm-THEN e1 e2 ⇒ of-semi--thm-attribute-aux (%d THEN, of-semi--thm-attribute-aux %d e2) # lacc e1

| Thm-simplified (Thm-thm s) e2 ⇒ of-semi--thm-attribute-aux %s simplified, of-semi--thm-attribute-aux %s e2 lacc s
| Thm-simplified (Thm-thms s) e2 ⇒ of-semi--thm-attribute-aux %s simplified, of-semi--thm-attribute-aux %s e2 lacc s
| Thm-simplified e1 e2 ⇒ of-semi--thm-attribute-aux (%s simplified, of-semi--thm-attribute-aux %s e2) # lacc e1
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 o else (%s) (To-string s))
  (if simp then ([simp,code-unfold] else o))
  (of-semi--thm-attribute-l l)
  ) Lemmas-simp-thms s l ⇒
  (lemmas\%s [simp,code-unfold] = %s)
  (if String.is-empty s then o 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 *)
\((\text{if } l = [] \text{ then} \\
\quad \text{)} \)

\(\text{else} \)

\(\langle \%s; %s \rangle \text{ attr (f l)}\)

definition (of-semi--attrib-genB :: (string list \Rightarrow \text{String.literal}) \Rightarrow \text{String.literal} \Rightarrow \text{string list} \Rightarrow \text{String.literal}) f attr l = (* error reflection: to be merged *)

\((\text{if } l = [] \text{ then} \\
\quad \text{)} \)

\(\text{else} \)

\(\langle \%s; %s \rangle \text{ attr (f l)}\)

definition of-semi--attrib = of-semi--attrib-genB of-semi--thm-l

definition of-semi--attribl = of-semi--attrib-genB (\lambda \text{. String-concat (\_)} (\text{L.map To-string l}))

fun of-semi--method where of-semi--method expr = (\lambda \\
\quad \text{Method-rule o-s \Rightarrow \langle rule\%s \rangle \text{ (case o-s of None \Rightarrow \text{)}}} \\
\quad \mid \text{Some s \Rightarrow \langle \%s \rangle \text{ (of-semi--thm-attribute s)}})

\mid \text{Method-drule s \Rightarrow \langle drule \%s \rangle \text{ (of-semi--thm-attribute s)}}

\mid \text{Method-erule s \Rightarrow \langle erule \%s \rangle \text{ (of-semi--thm-attribute s)}}

\mid \text{Method-intro l \Rightarrow \langle intro \%s \rangle \text{ (of-semi--thm-attribute-l1 l)}}

\mid \text{Method-elim s \Rightarrow \langle elim \%s \rangle \text{ (of-semi--thm-attribute s)}}

\mid \text{Method-subst asm l s =>}

\quad \text{let s-asn = if asm then \langle (asm) \rangle \text{ else} \text{ \_ in}}

\quad \text{if L.map String.to-list l = ["0"] then}

\quad \langle \text{\_} \rangle \text{ s-asn (of-semi--thm-attribute s)}

\quad \text{else}

\quad \langle \text{\_} \rangle \text{ s-asn (String-concat (\_)} (\text{L.map To-string l})) (of-semi--thm-attribute s)

\mid \text{Method-insert l => \langle insert \%s \rangle \text{ (of-semi--thm-l)}}

\mid \text{Method-plus t \Rightarrow \langle \%s\rangle \text{ (String-concat (\_)} (\text{List.map of-semi--method t}))}

\mid \text{Method-option t \Rightarrow \langle \%s\rangle \text{ (String-concat (\_)} (\text{List.map of-semi--method t}))}

\mid \text{Method-or t \Rightarrow \langle \%s\rangle \text{ (String-concat (\_)} (\text{List.map of-semi--method t}))}

\mid \text{Method-one (Method-simp-only l) \Rightarrow \langle simp only; \%s \rangle \text{ (of-semi--thm-l)}}

\mid \text{Method-one (Method-simp-add-del-split l2 l2 l2) \Rightarrow \langle simp\%s\%s \text{ (of-semi--attrib add l1) \text{ (of-semi--attrib del l2)}}}

\mid \text{Method-one (Method-simp-add-del-split l2 l2 l3) \Rightarrow \langle simp\%s\%s\%s \text{ (of-semi--attrib split l3)}}

\mid \text{Method-all (Method-simp-only l) \Rightarrow \langle simp-all only; \%s \rangle \text{ (of-semi--thm-l)}}

\mid \text{Method-all (Method-simp-add-del-split l2 l2 l2) \Rightarrow \langle simp-all\%s\%s \text{ (of-semi--attrib add l1) \text{ (of-semi--attrib del l2)}}}

\mid \text{Method-all (Method-simp-add-del-split l2 l2 l3) \Rightarrow \langle simp-all\%s\%s\%s \text{ (of-semi--attrib add l1) \text{ (of-semi--attrib del l2) \text{ (of-semi--attrib split l3)}}}}

\text{(48)
Method-auto-simp-add-split l-simp l-split ⇒ ⟨auto%ss%ss⟩ (of-semi-attrib simp l-simp) (of-semi-attrib split l-split)
Method-rename-tac l ⇒ ⟨rename-tac%ss⟩ (of-semi--attrib ⟨simp⟩ l-simp)
Method-case-tac e ⇒ ⟨case-tac%ss⟩ (of-semi--term e)
Method-blast None ⇒ ⟨blast⟩
Method-blast (Some n) ⇒ ⟨blast %db⟩ (To-nat n)
Method-clarify ⇒ ⟨clarify⟩
Method-metis l-opt l ⇒ ⟨metis%ss%ss⟩ (if l-opt = [] then ⟨⟩ else ⟨(%) ; (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%(ss)⟩ (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%(ss)⟩
| (String-concat ⟨⟩, ⟨⟩ (L.map of-semi--method l-apply)))

definition' (of-semi--command-proof = (let thesis = ⟨thesis⟩
| scope-thesis-gen = ⟨proof = %ss show %ss⟩
⟩;
| scope-thesis = λss. scope-thesis-gen s thesis in
| Command-command-end [] ⇒ ⟨⟩
| Command-command-end l-apply ⇒ ⟨apply%(ss)⟩
| (String-concat ⟨⟩, ⟨⟩ (L.map of-semi--method l-apply))
| Command-command-l ⇒ ⟨using %ss⟩
| (of-semi--thm-l l)
| Command-command-unfolding l ⇒ ⟨unfolding %ss⟩
| (of-semi--thm-l l)
| Command-command-let e-name e-body ⇒ scope-thesis (⟨let %ss = %ss⟩ (of-semi--term e-name) (of-semi--term e-body))
| Command-command-have n b c e-last ⇒ scope-thesis (⟨have %ssss %ss %ss⟩ (To-string n) (if b then ⟨simp⟩ else ⟨⟩) (of-semi--term e) (of-semi--command-final e-last))
| Command-command-fix-let l-lst l-lst o-show - ⇒
| scope-thesis-gen (⟨fix %ssss⟩ (String-concat ⟨⟩, ⟨⟩ (L.map To-string l-lst))
| (String-concat ⟨⟩
| (L.map
| (λ(e-name, e-body).
| : let %ss = %ss (of-semi--term e-name) (of-semi--term e-body))
| l-let)))
| (case o-show of None ⇒ thesis)
\textbf{definition} of-lemma - =
\begin{align*}
& (\lambda \text{Lemma n l-spec l-apply tactic-last } \Rightarrow \\
& \langle \text{lemma } \%s : \%s \\ \\
& (\text{To-string n}) \\
& (\text{String-concat} \ (\Rightarrow \ (L\ .\ map\ of-semi-term\ l-spec)) \\
& (\text{String-concat} \ (L\ .\ map\ (\lambda \ [[] \Rightarrow \ [] \ |\ \text{l-apply} \Rightarrow \ (\text{apply}(\%s) \\
& \))}) \rangle \\
& (\text{of-semi-command-final tactic-last}) \\
& | \text{Lemma-assumes n l-spec concl l-apply tactic-last } \Rightarrow \\
& \langle \text{lemma } \%s : \%s \\ \\
& (\text{To-string n}) \\
& (\text{String-concat} \ (\text{l-spec}) \\
& (\text{assumes } \%s \%s) \\
& \text{assumes } \%s \%s \%s \%s) \\
& (\text{let} \ (n, b) = \text{if b then } (\%s[\text{simp}] \ (\text{To-string n}), \text{False}) \text{ else } (\text{To-string n}, \text{String.is-empty} n) \text{ in} \\
& \text{if b then } \text{else } \%s \%s \text{n) } \\
& (\text{of-semi-term e}) \text{l-spec} \\
& \\
& (\text{shows } \%s \ (\text{of-semi-term concl}))) \\
& (\text{String-concat} \ (L\ .\ map\ of-semi-command-proof l-apply)) \\
& (\text{of-semi-command-final tactic-last}) \\
& (\text{String-concat } []) \\
& (L\ .\ map) \\
& (\lambda l\text{-apply-e.} \\
& \%\text{sqed}) \\
& (\text{if } l\text{-apply-e} = [] \text{ then} \\
& () \\
& \text{else} \\
& ( \\
& \%s ) \\
& (\text{String-concat} \ (L\ .\ map\ of-semi-command-state l-apply-e))) \\
& (\text{List.map-filter}) \\
& (\lambda \text{Command-let } - \Rightarrow \text{Some } [] \ | \ \text{Command-have } - - - \Rightarrow \text{Some } [] \ | \ \text{Command-fix-let} \\
& - - l \Rightarrow \text{Some } l \ | - \Rightarrow \text{None}) \\
& (\text{rev l-apply})))))
\end{align*}

\textbf{definition} of-axiomatization - = (\lambda \text{Axiomatization n e } \Rightarrow \langle \text{axiomatization where } \%s: \\
\%s \ (\text{To-string n}) \ (\text{of-semi-term e}) \rangle)

\textbf{definition} of-section - = (\lambda \text{Section n section-title } \Rightarrow

50
\%(s \%(s))
\%(section) (if n = 0 then \%(sub)
else if n = 1 then \%(subsub)
else \%(subsub))
(To-string section-title)

definition af-text - = (\lambda Text s \Rightarrow \%(text s) (To-string))
definition af-ML - = (\lambda SML e \Rightarrow \%(ML s) (of-semi-term' e))
definition af-setup - = (\lambda Setup e \Rightarrow \%(setup s) (of-semi-term' e))
definition af-thm - = (\lambda Thm thm \Rightarrow \%(thm s) (of-semi-thm-attribute-l1 thm))
definition' of-interpretation - = (\lambda Interpretation n loc-n loc-param tac \Rightarrow
interpretation s: s s
(To-string n)
(To-string loc-n)
(String-concat \%(L.map (\%(s) (of-semi-term s) loc-param))
(of-semi--command-final tac)));
definition af-semi-theory env =
(\lambda Theory-datatype dataty \Rightarrow of-datatype env dataty
\mid Theory-type-synonym ty-synonym \Rightarrow of-type-synonym env ty-synonym
\mid Theory-type-notation ty-notation \Rightarrow of-type-notation env ty-notation
\mid Theory-instantiation instantiation-class \Rightarrow of-instantiation env instantiation-class
\mid Theory-defs defs-overloaded \Rightarrow of-defs env defs-overloaded
\mid Theory-consts consts-class \Rightarrow of-consts env consts-class
\mid Theory-definition definition-hol \Rightarrow of-definition env definition-hol
\mid Theory-lemmas lemmas-simp \Rightarrow of-lemmas env lemmas-simp
\mid Theory-lemma lemma-by \Rightarrow of-lemma env lemma-by
\mid Theory-axiomatization axiom \Rightarrow of-axiomatization env axiom
\mid Theory-section section-title \Rightarrow of-section env section-title
\mid Theory-text text \Rightarrow of-text env text
\mid Theory-ML ml \Rightarrow of-ML env ml
\mid Theory-setup setup \Rightarrow of-setup env setup
\mid Theory-thm thm \Rightarrow of-thm env thm
\mid Theory-interpretation thm \Rightarrow of-interpretation env thm)
definition String-concat-map s f l = String-concat s (L.map f l)
definition' of-semi-theories env =
(\lambda Theories-one t \Rightarrow of-semi-theory env t
\mid Theories-locale data l \Rightarrow
locale s =
s begin
s end) (To-string (Hol Thy Locale-name data))
(String-concat-map
  ⟨
    (\(l-fix, o-assum\).
      (\%s\%s) (String-concat-map ≤
        (\(e, ty\). (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] =

  Print.of-datatype-def
  Print.of-type-synonym-def
  Print.of-type-notation-def
  Print.of-instantiation-def
  Print.of-defs-def
  Print.ofconsts-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--attribI-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

52
Print.of-thm-def
Print.of-interpretation-def
Print.of-semi-theory-def
Print.String-concat-map-def
Print.of-semi-theories-def

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
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 = λn1 n2. ToyTyObj (ToyTyCore-pre n1) (case n2 of None ⇒ [] | Some n2 ⇒ [[ToyTyCore-pre n2]])
    ; mk = λn t. toy-class-raw.make n t [] 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 envi-
ronment should nevertheless be provided, one can typically experiment this by copying-
pasting the following environment initialized in the above main:

definition main =
( let n = λn1. ToyTyObj (ToyTyCore-pre n1) []
    ; ToyMult = λm 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") ])
    ( L.map (META-class-raw Floor1) Design
        @ @ @ @ | META-association (toy-association.make
            ToyAssTy-association

55
5.1.2. Statically Executing the Exportation

```haskell
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

```isabelle
theory Generator-dynamic
imports Printer
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.

```isabelle
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.

56
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 = String.implode o META.to-list
fun To-nat (Code-Numeral.Nat i) = i

ML:
structure From = struct
val string = META.SS-base o META.ST
val binding = string o Binding.name-of
(*fun term ctxt s = string (XML.content-of (YXML.parse-body (Syntax.string-of-term ctxt s))))*)
val nat = Code-Numeral.Nat
val internal-oid = META.Oid o nat
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 nat
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 nat) 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 in-local decl thy =
  thy
|> Named-Target.theory-init
|> decl
|> Local-Theory.exit-global
)

ML: fun List-mapi f = META.mapi (f o To-nat)

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)
    (writeln o Markup.markup Markup.bad o Mi)
    (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-term To-string0
  val of-semi--term' = META.of-semi-term To-string0
  val fold = fold

ML:

structure Bind-Isabelle = struct

fun To-binding s = Binding.make (s, Position.none)
val To-binding = To-binding o To-string0

fun semi-method-simp g f = Method.Basic (fn ctxt => SIMPL-E-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-GOALS o 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)]))
  | 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, []))
  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 expr = let open META open Method open META-overload in case expr of Method-rule o-s => Basic (fn ctxt => METHOD (HEADGOAL o Isabelle-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' (blast-tac (depth-tac ctxt (To-nat lim)))
| SOME lim => fn ctxt => SIMPLE-METHOD' (depth-tac ctxt (To-nat 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 Isabelle-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 st = st
  |> Proof.enter-forward
  |> f
  |> Isar-Cmd.show ![(@{binding }, []), ![theses, []])] true

val semi-command-state = let open META-overload in
  fn META.Command-apply-end l => (fn st => st |> Proof.apply-end-results (then-tactic l)
    |> Seq.the-result )
end
val semi--command-proof = let open META-overload
                          in fn thesis = ?thesis
                          fun proof-show f = proof-show-gen f thesis in fn META.Command-apply l => (fn st => st |> Proof.apply-results (then-tactic l)
                          |> Seq.the-result )
                          | META.Command-using l => (fn st =>
                          let val ctxt = Proof.context-of st in
                          Proof.unfolding (map (fn s => (s, [])) (semi-thm-mult-l ctxt l))
                          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))
                          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
                          |> Isar.Cmd.have [( (To-binding n, if b then [Token.src (simp, Position.none) [] else []), [(of-semi-term e, [])])]
                          |> 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-binding i, NONE, NoSyn)) l))
                          (case o-exp of NONE => thesis | SOME l-spec =>
                          (String.concatWith (⇒ )
                          (List.map of-semi-term l-spec))
                          end

fun Theory-datatype in-theory in-local = let open META open META-overload in (*let val f = *)
                          fn (Datatype (n, l)) => in-local
                          (BNF-Util.Least-FP.
                          BNF-LFP.construct-lfp
                          (Ctr-Sugar.default-ctr-options-cmd,
                          [( ( ([]), To-binding n), NoSyn)
                          , List.map (fn (n, l) => ( (To-binding , To-binding n)
                          , List.map (fn s => (To-binding , of-semi-typ s)) l)
                          , NoSyn)) l)
                          , (To-binding , To-binding ))
                          , []))
                          | Theory-type-synonym (Type-synonym (n, v, l)) => in-theory (fn thy =>
                          let val s-bind = To-binding 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
(Specification.type-notation-cmd true, true) [(To-string0 n, Mixfix (To-string0 e, [], 1000))]
Theory-instantiation (Instantiation (n, n-def, expr)) => in-theory
(fn thy =>
  let val name = To-string0 n
  val tyco =
    [ let val Term.Type (s, -) = (Isabelle-TypeDecl.abbrev-cmd0 NONE thy name) in s end
  in
    thy
    |> Class.instantiation (tyco, [], 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" name -def), [])
            , of-semi-term expr)) false thy in
          (ty, thy)
        end) tyco
    |> 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-defs (Defs-overloaded (n, e)) => in-local
(Isar-Cmd.add-defs ((false, true), [(To-sharding n, of-semi-term e), []]))
Theory-consts (Consts (n, ty, symb)) => in-theory
(Sign.addconsts-cmd [(To-sharding n
                      , of-semi-typ ty
                      , Mixfix ("name" name symb, [], 1000))])
Theory-definition def => in-local
let val (def, e) = case def of
  Definition e => (NONE, e)
| Definition-where1 (name, (abbrev, prio), e) =>
    (SOME (To-sharding name
            , NONE
            , Mixfix ((1 ^ of-semi-term abbrev ^), [], To-nat prio)), e)
| Definition-where2 (name, abbrev, e) =>
    (SOME (To-sharding name
            , NONE
            , Mixfix (("name" name abbrev), [], 1000)), e) in
  (snd o Specification.definition-cmd (def, (\{binding\}, []), of-semi-term e)) false)
end
Theory-lemmas (Lemmas-simp-thm (simp, s, l)) => in-local
(fn lthy => (snd o Specification.thereoms Thm.lemmaK
  [((To-sharding s, List.map (fn s => Attrib.check-src lthy (Token.src (s, Position.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.thereoms Thm.lemmaK
  [((To-sharding s, List.map (fn s => Attrib.check-src lthy (Token.src (s, Position.none) [])))
    (if simp then [simp, code-unfold] else [])),
    List.map (fn x => ([semi-thm-attribute-single lthy x], [])) l]
  false) lthy)
[(To-binding s, List.map (fn s => Attrib.check-src lthy (Token.src (s, Position.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 Thm.lemmaK NONE (K I)

(@{binding }, [] [] (Element.Show ([(To-binding n, [])

, (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 Thm.lemmaK NONE (K I)

(List.map (fn (n, (b, e)) =>

Element.Assumes [( (To-binding n

, if b then [Token.src (simp, Position.none) [] else []

, (of-semi-term e, [])]))

l-spec)

(Element.Show ([(@{binding }, []), (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-binding 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)))

64
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
> Expression.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
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 l in-local thy
| Theories-locale (data, l) => fn thy => thy
> ( Expression.add-locale-cmd
      (To-binding (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-binding n, [])]
              , [(of-semi-term e, [])]]]]))
        (META.holThyLocale-header data))
      #> snd)
    |> fold (semi-theory Local-Theory.background-theory

65
(fn f => fn thy => thy
| Local-Theory.new-group
| f
| Local-Theory.reset-group
| Local-Theory.restore)) l

| Local-Theory.exit-global)
| META-boot-generation-syntax - => ret o l
| META-boot-setup-env - => ret o l
| 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
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)
    | ([(x, 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
end)

end
end
Part II.

A Toy Example
5.3. A Toy Library for Objects in a State

```plaintext
theory Toy-Library
imports Main
begin

  type-notation option (⟨-⟩⊥)
  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 (⟨[N]⟩)state =
    heap :: oid → 'α
    assocs :: oid → ([oid list] list)

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

end
```

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

5.4.1. Introduction

```plaintext
theory Design-deep
imports ../embedding/Generator-dynamic
begin

ML-file ~~/src/Doc/antiquote-setup.ML

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).

```
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 Generator-dynamic). 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 \texttt{End} is to behave as the identity function when \texttt{End} is called
without arguments. As example, here we are calling lots of \texttt{End} without arguments,
and no Isabelle code is generated.

\texttt{End End End}

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

\texttt{Class Molecule < Person}

As example, here no \texttt{End} is written.

The semantics of \texttt{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 \texttt{End} for doing so. \texttt{End} can be seen as a lazy meta-command:
\begin{itemize}
  \item without parameters, no code is generated,
  \item with some parameters (e.g., the symbol !), it forces the generation of the com-
    putation of the universe, by considering all already encountered classes. Then a
    partial representation of the universe can be interactively inspected.
\end{itemize}

\texttt{Class Galaxy}
\begin{itemize}
  \item \texttt{Attributes} \texttt{wormhole : UnlimitedNatural}
  \item \texttt{is-sound : Void}
\end{itemize}

\texttt{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.

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

\texttt{Class Person < Galaxy}
\begin{itemize}
  \item \texttt{Attributes} \texttt{salary : Integer}
  \item \texttt{boss : Person}
  \item \texttt{is-meta-thinking: Boolean}
\end{itemize}

There is not only \texttt{End} which forces the computation of the universe, for example
\texttt{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.

\texttt{Instance X_{Person1} :: Person = [ salary = 1300 , boss = X_{Person2} ]}
\texttt{and X_{Person2} :: Person = [ salary = 1800 ]}
Here we will call \texttt{Instance} again to show that the universe will not be computed again since it was already computed in the previous \texttt{Instance}.

\texttt{Instance} \( X_{\text{Person}^3} :: \text{Person} = [ \text{salary} = 1 ] \)

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

\texttt{(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, \( X_{\text{Person}^1} \), \( X_{\text{Person}^2} \) and \( X_{\text{Person}^3} \) will be resurrected... after the \texttt{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 \texttt{deep} mode, this is particularly not a danger for meta-commands to generate themselves (whereas for \texttt{shallow} the recursion might not terminate).

In this case, such meta-commands must automatically generate the appropriate call to \texttt{generation-synt}\n\texttt{ax} 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 \texttt{setup} is added after \texttt{generation-synt}\n\texttt{ax} for bootstrapping the state of the compiling environment.

\texttt{State} \( \sigma_1 = \)
\[ [ [ \text{salary} = 1000 \ , \ boss = \text{self}^1 ] :: \text{Person} ) \]
\[ , [ \text{salary} = 1200 ] :: \text{Person} ) \]
\[ , [ \text{salary} = 2600 \ , boss = \text{self}^3 ] :: \text{Person} ) \]
\[ , X_{\text{Person}^1} \]
\[ , [ \text{salary} = 2300 \ , boss = \text{self}^2 ] :: \text{Person} ) \]
\[ , X_{\text{Person}^2} ] \]

\texttt{State} \( \sigma_1' = \)
\[ [ X_{\text{Person}^1} \]
\[ , X_{\text{Person}^2} \]
\[ , X_{\text{Person}^3} ] \]

In certain circumstances, the command \texttt{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`.

\texttt{PrePost \(\sigma_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 \(\lambda\)-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 “\(\zeta\)-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 \(\lambda\)-terms can be given as parameters.

Starting from this expression:

\[
\text{(* PrePost } \langle\sigma_1\rangle \langle\text{sub}\rangle 1 [ (\langle\text{salary} = 1000, \text{boss} = \text{self 1} \rangle :: \text{Person}) ] *) \]

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):

\[
\text{(* State } \text{WFF-10_post} = [ (\langle\text{"salary"} = 1000, \text{"boss"} = \text{self 1} \rangle :: \text{Person}) ])
\]

\[
\text{PrePost[shallow] } \langle\sigma_1\rangle \langle\text{sub}\rangle 1 \text{ WFF-10_post }*) (\text{WFF-10-post is an automatically generated name}).
\]

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

\[
\text{(* Instance } \text{WFF-10_post_object0} :: \text{Person} = [ \langle\text{"salary"} = 1000, \text{"boss"} = [ \rangle ])
\]

\[
\text{State[shallow] } \text{WFF-10_post} = [ \text{WFF-10_post_object0} ]
\]

\[
\text{PrePost[shallow] } \langle\sigma_1\rangle \langle\text{sub}\rangle 1 \text{ WFF-10_post } *)
\]

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 `Generator-dynamic`). Since the Inner Syntax expressions would perhaps manipulate expressions coming from other theories than `Generator-dynamic`, 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:

\[
\text{Context Person :: content ()}
\]

\[
\text{Post } "\langle\text{close}\rangle\langle\text{open}\rangle"
\]
Here the expression `<close>`<open> 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. However, this is not the same for the resulting generated meta-command:

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

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 Generator-dynamic 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
begin
ML-file ~/src/Doc/antiquote-setup.ML

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 Generator-dynamic, 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 $X_{\text{Person}}^1 :: \text{Person} = [ \text{salary} = 1300 \ , \text{boss} = X_{\text{Person}}^2 ]$

and $X_{\text{Person}}^2 :: \text{Person} = [ \text{salary} = 1800 ]$

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

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

State $\sigma_1 =$

\[
[ ([ \text{salary} = 1000 \ , \text{boss} = \text{self } 1 ] :: \text{Person})
, ([ \text{salary} = 1200 ] :: \text{Person})
, ([ \text{salary} = 2600 \ , \text{boss} = \text{self } 3 ] :: \text{Person})
, X_{\text{Person}}^1
, ([ \text{salary} = 2300 \ , \text{boss} = \text{self } 2 ] :: \text{Person})
, X_{\text{Person}}^2 ]
\]

State $\sigma_1' =$

\[
[ X_{\text{Person}}^1
, X_{\text{Person}}^2
, X_{\text{Person}}^3 ]
\]

PrePost $\sigma_1 \sigma_1'$

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 "\<close\\open> *"

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

end
Bibliography

Part III.

Appendix
A. Grammars of Commands

A.1. Main Setup of Meta Commands

\[ \text{generation-syntax} : \text{theory} \rightarrow \text{theory} \]

\[ \text{syntax} \]

\[ \text{shallow semantics} \]

\[ \text{long-or-dirty} \]

\[ \text{syntex_print} \]

\[ \text{number} \]
\textit{export-code} sets the behavior of all incoming meta-commands. By default, without firstly writing \texttt{generation-syntax} meta-commands will only print in output what they have parsed, this is similar as giving to \texttt{generation-syntax} a non-empty list having only \texttt{syntax-print} as elements (on the other hand, nothing is printed when an empty list is received). Additionally \texttt{syntax-print} can be followed by an integer indicating the printing depth in output, similar as declaring \texttt{ML-print-depth} with an integer, but the global option \texttt{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 \texttt{shallow} mode, or to only display what would have been evaluated using the \texttt{deep} mode (i.e., to only show the generated Isabelle content in the output window).

Since several occurrences of \texttt{deep}, \texttt{shallow} or \texttt{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.

\texttt{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 \texttt{deep} mode in the list. In particular, this is only effective if there is at least one \texttt{deep} mode earlier declared.

As a side note, target languages for the \texttt{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-syntex 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-command 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**: \( \text{theory} \rightarrow \text{theory} \)
- **Abstract-class**: \( \text{theory} \rightarrow \text{theory} \)

```plaintext
Class
    Abstract_class
        binding = type-base
        type-object class
```
class

Attributes

binding : toy-type

context

context

Operations

::

binding : toy-type

= term

term

Pre

Post

use-prop

invariant

Invariant

Constraints

Existential

Aggregation : theory → theory
Association : theory → theory
Composition : theory → theory
Instance : theory \rightarrow theory

\begin{align*}
\text{Instance} & : \text{theory} \rightarrow \text{theory} \\
\text{binding} & \quad \vdash \quad \text{type-object} \\
\text{term-object} & \quad \text{object-cast} \\
\text{and} & \\
\text{term-object} & \\
\text{object-cast} & \\
\text{State} & : \text{theory} \rightarrow \text{theory}
\end{align*}
\[
\text{state} \quad \begin{cases} 
\text{shallow} \quad \text{binding} = \text{state} \\
\text{object-cast}
\end{cases}
\]

**PrePost**: \(\text{theory} \rightarrow \text{theory}\)

\[
\text{pre-post} \quad \begin{cases} 
\text{shallow} \quad \text{binding} = \text{pre-post} \\
\text{state}
\end{cases}
\]

**End**: \(\text{theory} \rightarrow \text{theory}\)

\[
\text{End} \quad \begin{cases} 
\text{forced} \quad \text{state}
\end{cases}
\]
A.3. Extensions of Isabelle Commands

BaseType : theory → theory

fun′ : local-theory → local-theory
definition′ : local-theory → local-theory
code-reflect′ : theory → theory

fun′
  target
  functionopts
  fixes
  where
  equations
definition′
  target
  thmdecl
  prop
  where
  decl
\texttt{fun} has the same semantics as \texttt{fun} except that it is possible to write the quote symbol (i.e., the symbol ") in all recursive enclosing cartouches.

\texttt{definition} has the same semantics as \texttt{definition} except that it is possible to write the quote symbol (i.e., the symbol ") in all recursive enclosing cartouches.

\texttt{code-reflect} has the same semantics as \texttt{code-reflect} except that it additionally contains the option \texttt{open} inspired from the command \texttt{export-code} (with the same semantics).

\texttt{laz}ey-code-printing : theory \rightarrow theory  
\texttt{apply-code-printing} : theory \rightarrow theory  
\texttt{apply-code-printing-reflect} : local-theory \rightarrow local-theory
lazy_code_printing

apply_code_printing

apply_code_printing_reflect

text

**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. Changes on Signatures

- ./src/HOL/Tools/Metis/Isabelle_metis_tactic.thy \(\text{Main2:}\)
- ./src/HOL/Tools/BNF/Isabelle_bnf_fp_def_sugar.thy \(\text{Main2:}\)
- ./src/Provers/Isabelle_classical.thy \(\text{Main2:}\)

Some signatures was removed for exposing the main structure.

B.2. Extensions for Cartouches

- ./src/HOL/ex/Isabelle_Cartouche_Examples.thy \(\text{Main0:}\)
  Some functions have been generalized for supporting cartouches.
- ./src/HOL/Tools/Function/Isabelle_fun.thy \(\text{Main0:}\)
  This file only contains the definition of \texttt{fun}.
- ./src/HOL/Tools/Function/Isabelle_function_common.thy \(\text{Main0:}\)
  Some functions have been generalized for supporting cartouches.
- ./src/Pure/Isar/Isabelle_parse_spec.thy \(\text{Main0:}\)
  Some functions have been generalized for supporting cartouches.
- ./src/Pure/Isar/Isabelle_isar_syn.thy \(\text{Main0:}\)
  This file only contains the definition of \texttt{definition}.

B.3. Other Changes

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

93
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 tyEXTAtom = mkEXTAtom oid oid list option int option bool option nat option unit option

datatype tyAtom = mkAtom tyEXTAtom int option

datatype tyEXTMolecule = mkEXTMolecule-Atom tyAtom
    | mkEXTMolecule oid oid list option int option bool option nat option unit option

datatype tyMolecule = mkMolecule tyEXTMolecule

datatype tyEXTPerson = mkEXTPerson-Atom tyAtom
    | mkEXTPerson tyMolecule oid oid list option int option bool option unit option

datatype tyPerson = mkPerson tyEXTPerson oid list option int option bool option

datatype tyEXTGalaxy = mkEXTGalaxy-Person tyPerson
    | mkEXTGalaxy-Molecule tyMolecule
    | mkEXTGalaxy-Atom tyAtom
    | mkEXTGalaxy oid

datatype tyGalaxy = mkGalaxy tyEXTGalaxy nat option unit option

datatype tyEXTToyAny = mkEXTToyAny-Galaxy tyGalaxy
    | mkEXTToyAny-Person tyPerson
    | mkEXTToyAny-Molecule tyMolecule
    | mkEXTToyAny-Atom tyAtom
    | mkEXTToyAny oid

datatype tyToyAny = mkToyAny tyEXTToyAny

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 \( \mathfrak{A} = \text{in}_{\text{Atom}} \; \text{ty}_{\text{Atom}} \)
| \text{in}_{\text{Molecule}} \; \text{ty}_{\text{Molecule}} \\
| \text{in}_{\text{Person}} \; \text{ty}_{\text{Person}} \\
| \text{in}_{\text{Galaxy}} \; \text{ty}_{\text{Galaxy}} \\
| \text{in}_{\text{ToyAny}} \; \text{ty}_{\text{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 \( \text{Atom} = \langle \langle \text{ty}_{\text{Atom}} \rangle \bot \rangle \bot \)

type-synonym \( \text{Molecule} = \langle \langle \text{ty}_{\text{Molecule}} \rangle \bot \rangle \bot \)

type-synonym \( \text{Person} = \langle \langle \text{ty}_{\text{Person}} \rangle \bot \rangle \bot \)

type-synonym \( \text{Galaxy} = \langle \langle \text{ty}_{\text{Galaxy}} \rangle \bot \rangle \bot \)

type-synonym \( \text{ToyAny} = \langle \langle \text{ty}_{\text{ToyAny}} \rangle \bot \rangle \bot \)

definition \( \text{oid}_{\text{Atom}} - 0 \; \text{boss} = 0 \)

definition \( \text{oid}_{\text{Molecule}} - 0 \; \text{boss} = 0 \)

definition \( \text{oid}_{\text{Person}} - 0 \; \text{boss} = 0 \)

definition \( \text{oid1} = 1 \)

definition \( \text{oid2} = 2 \)

definition \( \text{inst-assoc1} = (\lambda \text{oid-class} \; \text{to-from} \; \text{oid} \; . \; (\text{case} \; (\text{deref-assocs-list} \; (\langle \langle \text{to-from}::\text{oid} \rangle \text{list} \text{list} \Rightarrow \text{oid} \text{list} \times \text{oid} \text{list} \rangle) \; \langle \langle \text{oid}::\text{oid} \rangle \rangle) \; (\langle \langle \text{map-of-list} \; (\langle \langle \text{oid}::\text{oid} \rangle \rangle) \; (\langle \langle \text{drop} \; (\langle \langle \langle \text{map-of-list} \; (\langle \langle \text{oid::-boss} \rangle \text{list} \text{list} \rangle) \; (\text{List.map} \; (\langle \langle \lambda (x \; , \; y) \; . \; [x \; , \; y] \; \circ \; \text{switch2}::0I) \; (\langle \langle \langle \text{oid}1 \; , \; \text{oid2} \rangle \rangle) \rangle) \rangle) \rangle) \rangle) \rangle) \; \text{of Nil} \Rightarrow \text{None} \)
| \text{l} \Rightarrow (\text{Some} \; (\text{l}))::\text{oid} \text{list} \text{option}) \)

definition \( \text{oid3} = 3 \)

definition \( \text{inst-assoc3} = (\lambda \text{oid-class} \; \text{to-from} \; \text{oid} \; . \; (\text{case} \; (\text{deref-assocs-list} \; (\langle \langle \text{to-from}::\text{oid} \rangle \text{list} \text{list} \Rightarrow \text{oid} \text{list} \times \text{oid} \text{list} \rangle) \; \langle \langle \text{oid}::\text{oid} \rangle \rangle) \; (\langle \langle \text{drop} \; (\langle \langle \langle \text{map-of-list} \; (\langle \langle \text{oid}::\text{oid} \rangle \rangle) \rangle) \; (\langle \langle \text{oid::-boss} \rangle \text{list} \text{list} \rangle) \rangle) \rangle) \rangle) \; \text{of Nil} \Rightarrow \text{None} \)
| \text{l} \Rightarrow (\text{Some} \; (\text{l}))::\text{oid} \text{list} \text{option}) \)

definition \( \text{oid4} = 4 \)

definition \( \text{oid5} = 5 \)

definition \( \text{oid6} = 6 \)

definition \( \text{oid7} = 7 \)
definition inst-assoc4 = (λoid-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 switch2-01)) (((oid7) , [oid6]) , [[oid6] , [oid1]] , [[oid4] , [oid5]])))))))) (oid-class::oid))))))) of Nil ⇒ None

| l ⇒ (Some (l)))::oid list option)

locale state-σ1 =
fixes oid4 :: nat
fixes oid5 :: nat
fixes oid6 :: nat
fixes oid7 :: nat
fixes oid2 :: nat
assumes distinct-oid: (distinct ([oid4 , oid5 , oid6 , oid1 , oid7 , oid2]))
fixes σ1-object0 Person :: tyPerson
fixes σ1-object0 :: Person
assumes σ1-object0-def: σ1-object0 = (λ-. [σ1-object0 Person])
fixes σ1-object1 Person :: tyPerson
fixes σ1-object1 :: Person
assumes σ1-object1-def: σ1-object1 = (λ-. [σ1-object1 Person])
fixes σ1-object2 Person :: tyPerson
fixes σ1-object2 :: Person
assumes σ1-object2-def: σ1-object2 = (λ-. [σ1-object2 Person])
fixes XPerson1 Person :: tyPerson
fixes XPerson1 :: Person
assumes XPerson1-def: XPerson1 = (λ-. [XPerson1 Person])
fixes σ1-object4 Person :: tyPerson
fixes σ1-object4 :: Person
assumes σ1-object4-def: σ1-object4 = (λ-. [σ1-object4 Person])
fixes XPerson2 Person :: tyPerson
fixes XPerson2 :: Person
assumes XPerson2-def: XPerson2 = (λ-. [XPerson2 Person])
begin
definition σ1 = (state.make ((Map.empty (oid4 ⇒ (in_person (σ1-object0 Person)))) (oid5 ⇒ (in_person (σ1-object1 Person)))) (oid6 ⇒ (in_person (σ1-object2 Person)))) (oid1 ⇒ (in_person (XPerson1 Person)))) (oid7 ⇒ (in_person (σ1-object4 Person)))) (oid2 ⇒ (in_person (XPerson2 Person)))) (((map-of-list (((map-person 0--boss , (List.map ((λ(x , y). [x , y]) o switch2-01)) (((oid4) , [oid2]) , [[oid6] , [oid4]] , [[oid1] , [oid6]] , [[oid7] , [oid5]]))))))))

lemma perm-σ1 : σ1 = (state.make ((Map.empty (oid2 ⇒ (in_person (XPerson2 Person)))) (oid7 ⇒ (in_person (σ1-object4 Person)))) (oid1 ⇒ (in_person (XPerson1 Person)))) (oid6 ⇒ (in_person (σ1-object2 Person)))) (oid5 ⇒ (in_person (σ1-object1 Person)))) (oid4 ⇒ (in_person (σ1-object0 Person))))

apply (simp add: σ1-def)
apply (subt (1) fun-upd-twist, metis distinct-oid distinct-length-2-or-more)
apply (subt (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 (1) fun-upd-twist, metis distinct-oid distinct-length-2-or-more)
by(simp)
end

locale state-σ₁' =
fixes oid₁ :: nat
fixes oid₂ :: nat
fixes oid₃ :: nat
assumes distinct-oid: (distinct ([oid₁ , oid₂ , oid₃]))
fixes P_person P_person :: ty_person
fixes P_person ₁ :: :Person
assumes P_person P_person₁-def: P_person P_person₁ = (λ· [[P_person P_person₁]])
fixes P_person ₂ :: :Person
assumes P_person P_person₂-def: P_person P_person₂ = (λ· [[P_person P_person₂]])
fixes P_person ₃ :: :Person
assumes P_person P_person₃-def: P_person P_person₃ = (λ· [[P_person P_person₃]])
begin
definition σ₁' = (state.make ((Map.empty (oid₁ → (in_Person (P_person P_person₁)))) (oid₂ → (in_Person (P_person P_person₂)))) (oid₃ → (in_Person (P_person P_person₃)))) ((map-of-list ([[oid₁ , oid₂ , oid₃]])))

lemma perm-σ₁' : σ₁' = (state.make ((Map.empty (oid₁ → (in_Person (P_person P_person₁)))) (oid₂ → (in_Person (P_person P_person₂)))) (oid₃ → (in_Person (P_person P_person₃)))) ((map-of-list ([[oid₁ , oid₂ , oid₃]]))))
apply(simp add: σ₁'-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-σ₁-σ₁' =
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 Person :: tyPerson  
fixes Person1 :: Person  
assumes Person1-def: Person1 = (λ-. [[Person1 Person]])  
fixes Person2 :: Person  
assumes Person2-def: Person2 = (λ-. [[Person2 Person]])  
fixes Person3 :: Person  
assumes Person3-def: Person3 = (λ-. [[Person3 Person]])  
fixes σ1-object0 :: Person  
assumes σ1-object0-def: σ1-object0 = (λ-. [[σ1-object0 Person]])  
fixes σ1-object1 :: Person  
assumes σ1-object1-def: σ1-object1 = (λ-. [[σ1-object1 Person]])  
fixes σ1-object2 :: Person  
assumes σ1-object2-def: σ1-object2 = (λ-. [[σ1-object2 Person]])  
fixes σ1-object4 :: Person  
assumes σ1-object4-def: σ1-object4 = (λ-. [[σ1-object4 Person]])  
assumes σ1': (state-σ1 (oid4) (oid5) (oid6) (oid7) (oid2) (σ1-object0 Person) (σ1-object0) (σ1-object1 Person) (σ1-object1) (σ1-object2 Person) (σ1-object2) (XPerso

begin  
interpretation state-σ1: state-σ1 oid4 oid5 oid6 oid7 oid2 σ1-object0 Person σ1-object0 σ1-object1 Person σ1-object1 σ1-object2 Person σ1-object2 Person σ1-object4 Person σ1-object4 Person (XPerso

interpretation state-σ1': state-σ1' oid1 oid2 oid3 Person1 Person Person2 Person Person3 Person1 Person2 Person Person3 Person3 by (rule σ1')  
end  
end