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
frederic.tuong@lri.fr burkhart.wolff@lri.fr

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

We represent a theory of (a fragment of) Isabelle/HOL in Isabelle/HOL. The purpose of this exercise is to write packages for domain-specific specifications such as class models, B-machines, . . ., and generally 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 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 9

1 Initialization 11
1.1 Optimization on the String Datatype 11
1.2 Basic Extension of the Standard Library 12
1.2.1 Polymorphic Cartouches 12
1.2.2 Operations on List 12
1.2.3 Operations on Char 13
1.2.4 Operations on String (I) 14
1.2.5 Operations on String (II) 15
1.3 Miscellaneous 17

2 Defining Meta-Models 19
2.1 (Pure) Term Meta-Model aka. AST definition of (Pure) Term 19
2.1.1 Type Definition 19
2.1.2 Operations of Fold, Map, ..., on the Meta-Model 19
2.2 SML Meta-Model aka. AST definition of SML 20
2.2.1 Type Definition 20
2.2.2 Extending the Meta-Model 20
2.3 Isabelle Meta-Model aka. AST definition of Isabelle 21
2.3.1 Type Definition 22
2.3.2 Extending the Meta-Model 25
2.3.3 Operations of Fold, Map, ..., on the Meta-Model 30

3 Parsing Meta-Models 31
3.1 Initializing the Parser 31
3.1.1 Some Generic Combinators 31
3.1.2 Generic Locale for Parsing 33
3.2 Instantiating the Parser of (Pure) Term 33
3.2.1 Main 33

4 Printing Meta-Models 35
4.1 Initializing the Printer 35
4.1.1 Kernel Code for Target Languages 35
4.1.2 Interface with Types 39
4.1.3 Interface with Constants 40
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 \texttt{char list}, until we reach the end. As optimization, we also consider the use of \texttt{String.literal} besides \texttt{char list}.

type-notation natural (nat)
definition $\text{Succ} \ x = x + 1$
datatype \texttt{string\_base} = $\text{ST} \ \text{String.literal}$
| \text{ST'} \ \text{char list}$

datatype \texttt{abr-string} =
\text{SS-base} \ \text{string\_base}$
| \text{String-concatWith} \ \text{abr-string} \ \text{abr-string list}$
syntax \texttt{-string1} :: - $\Rightarrow$ \texttt{abr-string} \ (\langle\rangle)
translations $\langle x \rangle$ $\Leftarrow$ \texttt{CONST SS-base} \ (CONST ST \ (CONST STR \ x))
syntax \texttt{-string2} :: - $\Rightarrow$ \texttt{String.literal} \ (\langle\rangle)$
translations $\langle x \rangle$ $\Leftarrow$ \texttt{CONST STR \ x}$
syntax \texttt{-string3} :: - $\Rightarrow$ \texttt{abr-string} \ (\langle\rangle)$
translations $\langle x \rangle$ $\Leftarrow$ \texttt{CONST SS-base} \ (CONST ST' \ x)$
syntax \texttt{-char1} :: - $\Rightarrow$ \texttt{abr-string} \ (\langle\rangle)$
translations \texttt{"x"} $\Leftarrow$ \texttt{CONST SS-base} \ (CONST ST' \ ((CONST Cons) \ x \ (CONST Nil)))
type-notation \texttt{abr-string} (\texttt{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).

\[ \text{ML} \]

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...

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

1.2.2. Operations on List

datatype (‘a, ‘b) nsplit = Nsplit-text ‘a
                  | Nsplit-sep ‘b
locale L
begin
definition map where map f l = rev (foldl (λl x. f x # l) [] l)
definition flatten l = foldl (λacc x. x # acc) acc (rev l) [] (rev l)
definition mapi f l = rev (fst (foldl (λl x. f x # l, Succ cpt) ([], θ::nat) l))
definition append a b = L.map flatten [a, b]
definition filter f l = rev (foldl (λl x. if f x then x # l else l) [] l)
definition mapM f l accu =
  (let (l, accu) = List.fold (λx (l, accu). λNone ⇒ if x1 = x2 then Some v else None | x ⇒ x) l None
    in
  (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)))
    in
  (let f = λx. ¬ f x in
    (takeWhile f l, case dropWhile f l of [] ⇒ (None, []) | x # xs ⇒ (Some x, xs)))
  in
  (enumerate 0 (reverse l))))

definition take where take reverse lg l = reverse (snd (L.split (takeWhile (λ(n, -). n < lg) (enumerate 0 (reverse l)))))
definition take-last = take rev
definition take-first = take id
definition replace-gen f-res l c0 lby =
  (let Nsplit-text = λl lgen. if l = [] then lgen else Nsplit-text l # lgen in
case List.fold
  (λ c1 (l, lgen).
    if c0 c1 then
      lby
    else
      Nsplit-text lby c1 lgen
  )
)
(lby, Nsplit-sep c1 ≠ Nsplit-text l lgen)
else
  (c1 ≠ l, lgen))
(rev l)
([], [])
of (l, lgen) ⇒ f-res (Nsplit-text l lgen))
definition nsplit-f l c0 = replace-gen id l c0 []
definition replace = replace-gen (L.flatten o L.map (λ Nsplit-text l ⇒ l | - ⇒ []))

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

definition map-find = map-find-aux []
end
notation L.append (infixr @@@@ 65)

lemmas [code] =
  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 Stringbase

definition (in S) flatten = String-concatWith ⊥
definition (in String) flatten a b = S.flatten [a, b]
notation String.flatten (infixx @@[65])
definition (in String) make n c = ❱L.map (λ-. c) (L.upto 1 n)❱
definition (in Stringbase) 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 =
| 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 (Stringexplode s)
definition (in Stringbase) foldl where foldl f accu = (λ ST s ⇒ String.foldl-one f accu s
| ST' s ⇒ List.foldl f accu s)

fun (in String) foldl where
foldl f accu e =
| String-concatWith abr l ⇒
| case l of [] ⇒ accu
| x # xs ⇒ List.foldl (λaccu. foldl f (foldl f accu abr)) (foldl f accu x) xs)) e

definition (in S) replace-chars f s1 s2 =
s1 @@ (case s of None ⇒ ⊥ | Some s ⇒ flatten (L.map f (Stringexplode s)))) @@ s2

definition (in String) map where map f = map-gen (S.replace-chars (λc. "f c") (λx. "f x"))
definition (in String) replace-chars f = map-gen (S.replace-chars (λc. f c)) f

definition (in String) replace-chars f = foldl (λb s. b & f s) True

definition (in String) length where length = foldl (λn. - Suc n) 0

definition (in String) to-list s = rev (foldl (λl c. c # l) [] s)
definition (in Stringbase) to-list = (λ ST s ⇒ String.explode s | ST' l ⇒ l)
definition (in String) to-Stringbase = (λ SS-base s ⇒ s | s ⇒ ST' (to-list s))
definition (in Stringbase) to-String = Stringbase

definition (in Stringbase) is-empty = (λ ST s ⇒ s = STR "''")
| ST' s ⇒ s = []

fun (in String) is-empty where
is-empty e = (λ SS-base s ⇒ Stringbase.is-empty s | String-concatWith - l ⇒ list-all is-empty l) e

definition (in String) equal s1 s2 = (to-list s1 = to-list s2)
notation String.equal (infixl ≡ 50)
definition (in String) assoc x l = L.assoc (to-list x) (L.map (map Prod Stringbase.to-list id) l)
definition (in String) member l x = L.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))

lemmas [code] =
S.flatten-def
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))

**context** String

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

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

**definition** to-bold-number = replace-chars (λc. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] ! (nat-of-char c - 48))

**fun** of-nat-aux l (n :: Nat.nat) = (if n < 10 then n # l else of-nat-aux (n mod 10 # l) (n div 10))

**definition** of-nat where of-nat n = ≡nat-raw-to-str (of-nat-aux [] n)

**definition** of-natural = of-nat o nat-of-natural

**end**

**lemmas** [code] =

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

definition add-0 n =
(let n = nat-of-char n in
 S.flatten (L.map (λ_. (0)) (upt 0 (if n < 10 then 2 else if n < 100 then 1 else 0))))
@@@ String.of-nat n
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)
definition isub = replace-chars (λc.
  if is-letter c | is-digit c | List.member "<","" c then (λ. "c" else add-0 c)
definition insup s = (λ. s)
end
lemmas [code] =
String.base255-def
String.isub-def
String.insup-def

definition text-of-str str =
(let s = (λ)
  ; ap = (λ #) in
 S.flatten [ (let ), s, ( = char-of-nat in )
  , String.replace-chars (λc.
    if is-letter c then
      S.flatten [CHR "", "c", "",ap]
  else
      S.flatten [s, ( , add-0 c, ap)]
  str
 , ()]))
definition text2-of-str = String.replace-chars (λc. S.flatten [(\), ("), "c", ()])

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

definition' escape-sml = String.replace-chars ((\) (* ERROR code-reflect *))
\lambda Char Nibble2 Nibble2 \Rightarrow (\) | x \Rightarrow "x"*
\lambda x. if x = Char Nibble2 Nibble2 then (\) else "x"

definition mk-constr-name = (\ 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 [] \Rightarrow \nothing | x \# xs \Rightarrow S.flatten [x, S.flatten (L.map (\s. (\s). @@ s) xs)], (\):]
definition mk-dot-par-dot s = mk-dot-par-gen dot [s]
definition mk-dot-comment s1 s2 s3 = mk-dot s1 (S.flatten [s2, (\), s3, (\)])
definition hol-definition s = S.flatten [s, (\)-def]
definition hol-split s = S.flatten [s, (\)-split]

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 -LetOCaml :: [letbinds, 'a] \Rightarrow 'a ((letOCaml (-) in (-)) [0, 10] 10)
translations -LetOCaml (-binds b bs) e \Rightarrow -LetOCaml b (-Let bs e)
  letOCaml x = a in e \Rightarrow CONST id (CONST Let a (\%) e)

syntax -case-syntaxOCaml :: ['a, cases-syn] => 'b ((caseOCaml - of/ -) 10)
translations caseOCaml v of w => x \Rightarrow CONST id (case-syntax v (-case1 w x)
  (-case1 y z))

syntax -LambdaScala :: [pttrn, bool] \Rightarrow 'a ((\lambdaScala -/ -) [0, 10] 10)
  -LambdaScala :: [pttrn, pttrn, bool] \Rightarrow 'a ((\lambdaScala -/ -) [0, 0, 10] 10)
translations \lambdaScala x y. P \Rightarrow CONST id (\%) x y. P
  \lambdaScala x. P \Rightarrow CONST id (\%) x. P

syntax -case-syntaxScala :: ['a, cases-syn] => 'b ((caseScala - of/ -) 10)
translations caseScala v of w => x \Rightarrow CONST id (case-syntax v (-case1 w x)
  (-case1 y z)))

end
2. Defining Meta-Models

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

theory Meta-Pure
imports ../Init
begin

2.1.1. Type Definition

type-synonym indexname = string × nat
type-synonym class = string
type-synonym sort = class list

datatype typ =
  Type string typ list |
  TFree string sort |
  TVar indexname sort

datatype term =
  Const string typ |
  Free string typ |
  Var indexname typ |
  Bound nat |
  Abs string typ term |
  App term term (infixl $200$

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

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

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 [\texttt{NONE}]
definition some s = app (\texttt{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 (List.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

definition rewrite-val-def = rewrite Sval
end

lemmas [code] =

SML.string-def
SML.rewrite-def
SML.basic-def
SML.binop-def
SML.annot-def
SML.function-def
SML.apply-def
SML.paren-def
SML.let-open-def
SML.app-def
SML.none-def
SML.some-def
SML.option'-def
SML.option-def
SML.parenthesis-def
SML.binop-l-def
SML.list-def
SML.list'-def
SML.pair-def
SML.pair'-def
SML.rewrite-val-def
SML.rewrite-fun-def

end

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

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

2.3.1. Type Definition

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

datatype `semi--typ` = Typ-apply `semi--typ` `semi--typ` list
  | Typ-apply-bin string `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`
Definition-where2 string semi-term semi-term

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
| 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

24
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
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 l f = Type-synonym n l (f l)
definition Term-annot' e s = Term-annot e (Typ-base s)
definition Term-lambda s = Term-bind (\x) (Term-basic s)
definition Term-lambda x = Term-lambdas [x]
definition Term-lambdas0 = Term-bind (\)
definition Term-lam x f = Term-lambdas0 (Term-basic [x]) (f x)
definition Term-some = Term-paren (\( )
definition Term-parenthesis (* mandatory parenthesis *) = Term-paren (\() ()
definition Term-warn-parenthesis (* optional parenthesis that can be removed but a warning will be raised *) = Term-parenthesis

definition Term-pat b = Term-basic [? b @ b]
definition Term-And x f = Term-bind (\x) (Term-basic [x]) (f x)
definition Term-exists x f = Term-bind (\x) (Term-basic [x]) (f x)
definition Term-binop = Term-rewrite
definition term-binop s l = (case rev l of x # xs ⇒ List.fold (λx. Term-binop x s) xs x)
definition term-binop's l = (case rev l of x # xs ⇒ List.fold (λx. Term-parenthesis o Term-binop x s) xs x)
definition Term-set l = (case l of [] ⇒ Term-basic [{}]) | - ⇒ Term-paren (\( (term-binop (\) l))
definition Term-list l = (case l of [] ⇒ Term-basic [{}]) | - ⇒ Term-paren (\( (term-binop (\) l))
definition Term-list' f l = Term-list (L.map f l)
definition Term-pair e1 e2 = Term-parenthesis (Term-binop e1 (\) e2)
definition Term-pair's l = (case l of [] ⇒ Term-basic [()]) | - ⇒ Term-paren (\( (term-binop (\) l))
definition Term-string s = Term-basic [S.flatten [0, s, 0]]
definition Term-appsys0 e l = Term-parenthesis (Term-app e (L.map Term-parenthesis l))
definition Term-appsys e l = Term-appsys0 (Term-parenthesis e) l
definition Term-app e = Term-appsys0 (Term-basic [e])
definition Term-preunary e1 e2 = Term-preunary e1 [e2]
definition Term-postunary e1 e2 = Term-app e1 [e2]
definition Term-case = Term-fun-case o Some
definition Lemmas-simp = Lemmas-simp-thm True
definition Lemmas-nosimp = Lemmas-simp-thm False

definition Consts-value = (\()

definition Consts-raw0 s l e o-arg = Consts s l (String.replace-chars (λc. if c = Char Nibble5 NibbleF then (\) else "c") e @ @ (case o-arg of
  None ⇒ (λ)
  | Some arg ⇒
    let ap = λs. (\( (λ) s (λ) (λ)) 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 ⟨×⟩
declaration 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) l1)
definition meth-gen-simp-add-del-split l1 l2 l3 = Method-simp-add-del-split (L.map Thms-single l1) (L.map Thms-single l2) (L.map Thms-single l3)
definition meth-gen-simp-add-split l1 l2 = Method-simp-add-del-split (L.map Thms-single l1) (L.map Thms-single l2)
definition meth-gen-simp-only l = Method-simp-only (L.map Thms-single l)
definition meth-gen-simp-only′ l = Method-simp-only (L.map Thms-mult l)
definition meth-gen-simp-add0 l = Method-simp-add-del (L.map Thms-single l) []
definition simp = Method-one meth-gen-simp
definition simp-add2 l1 l2 = Method-one (meth-gen-simp-add2 l1 l2)
definition simp-add-del l1 l2 = Method-one (meth-gen-simp-add-del l1 l2)
definition simp-add-del-split l1 l2 l3 = Method-one (meth-gen-simp-add-del-split l1 l2 l3)
definition simp-add-split l1 l2 = Method-one (meth-gen-simp-add-split l1 l2)
definition simp-only l = Method-one (meth-gen-simp-only l)
definition simp-only′ l = Method-one (meth-gen-simp-only′ l)
definition simp-add0 l = Method-one (meth-gen-simp-add0 l)
definition simp-add = simp-add2 []
definition simp-all = Method-all meth-gen-simp
definition simp-all-add l = Method-all (meth-gen-simp-add2 [] l)
definition simp-all-only l = Method-all (meth-gen-simp-only l)
definition simp-all-only' l = Method-all (meth-gen-simp-only' l)
definition auto-simp-add2 l1 l2 = Method-auto-simp-add-split (L.flatten (L.map Thms-mult l1), L.map (Thms-single o Thms-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 [0]
definition subst = subst-l [0]
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
definition \( ty\text{-}arrow \ l = (\text{case } \text{rev } l \ \text{of } x \# \ xs \Rightarrow \text{List} \cdot \text{fold} \ Ty\text{-}arrow \ xs \ x) \) 

locale \( C \)
begin
  definition \( \text{done} = \text{Command}\text{-}done \)
  definition \( \text{by} = \text{Command}\text{-}by \)
  definition \( \text{sorry} = \text{Command}\text{-}sorry \)
  definition \( \text{apply-end} = \text{Command}\text{-}apply\text{-}end \)
  definition \( \text{apply} = \text{Command}\text{-}apply \)
  definition \( \text{using} = \text{Command}\text{-}using \ o \ \text{L} \cdot \text{map} \ \text{Thms}\text{-}single \)
  definition \( \text{unfolding} = \text{Command}\text{-}unfolding \ o \ \text{L} \cdot \text{map} \ \text{Thms}\text{-}single \)
  definition \( \text{let'} = \text{Command}\text{-}let \)
  definition \( \text{fix}\text{-}let = \text{Command}\text{-}fix\text{-}let \)
  definition \( \text{fix } l = \text{Command}\text{-}fix\text{-}let \ l \ [\ None \ ] \)
  definition \( \text{have } n = \text{Command}\text{-}have \ n \ False \)
  definition \( \text{have0} = \text{Command}\text{-}have \)
end

lemmas [code] =

  \( C\text{-}done\text{-}def \)
  \( C\text{-}by\text{-}def \)
  \( C\text{-}sorry\text{-}def \)
  \( C\text{-}apply\text{-}end\text{-}def \)
  \( C\text{-}apply\text{-}def \)
  \( C\text{-}using\text{-}def \)
  \( C\text{-}unfolding\text{-}def \)
fun cross-abs-aux where

\[
\text{cross-abs-aux } f \, l \, x = (\lambda \ (\text{Suc } n, \text{Abs } s - t) \Rightarrow f \ s \ (\text{cross-abs-aux } f \ (s \# \ l) \ (n, \ t))
\]

| (-, e) ⇒ Term-term l e |

\]

\[
\text{definition cross-abs } f \ n \ l = \text{cross-abs-aux } f \ [] \ (n, \ l)
\]

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

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

| x ⇒ x)
3. Parsing Meta-Models

3.1. Initializing the Parser

theory Parser-init
imports ../Init
begin

3.1.1. Some Generic Combinators

definition K x = x

definition co1 o = o

definition co2 f (g x1 x2) = f (g x1 x2)

definition co3 f (g x1 x2 x3) = f (g x1 x2 x3)

definition co4 f (g x1 x2 x3 x4) = f (g x1 x2 x3 x4)

definition co5 f (g x1 x2 x3 x4 x5) = f (g x1 x2 x3 x4 x5)

definition co6 f (g x1 x2 x3 x4 x5 x6) = f (g x1 x2 x3 x4 x5 x6)

definition co7 f (g x1 x2 x3 x4 x5 x6 x7) = f (g x1 x2 x3 x4 x5 x6 x7)

definition co8 f (g x1 x2 x3 x4 x5 x6 x7 x8) = f (g x1 x2 x3 x4 x5 x6 x7 x8)

definition co9 f (g x1 x2 x3 x4 x5 x6 x7 x8 x9) = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9)

definition co10 f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10) = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10)

definition co11 f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11) = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11)

definition co12 f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12) = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12)

definition co13 f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13) = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13)

definition co14 f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14) = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14)

definition co15 f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15) = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15)

definition ap1 a v0 (f1 v1) = a v0 [f1 v1]

definition ap2 a v0 (f1 f2 v1 v2) = a v0 [f1 v1, f2 v2]

definition ap3 a v0 f1 f2 f3 v1 v2 v3 = a v0 [f1 v1, f2 v2, f3 v3]

definition ap4 a v0 f1 f2 f3 f4 f5 v1 v2 v3 v4 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4]

definition ap5 a v0 f1 f2 f3 f4 f5 f6 v1 v2 v3 v4 v5 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5]

definition ap6 a v0 f1 f2 f3 f4 f5 f6 f7 v1 v2 v3 v4 v5 v6 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6]

definition ap7 a v0 f1 f2 f3 f4 f5 f6 f7 v1 v2 v3 v4 v5 v6 v7 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7]

definition ap8 a v0 f1 f2 f3 f4 f5 f6 f7 f8 v1 v2 v3 v4 v5 v6 v7 v8 = a v0 [f1 v1, f2 v2, f3 v3,
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 = af-pair a b (of-string a b) (of-nat a b)
definition of-pure-class = of-string
definition of-pure-sort a b = of-list a b (of-pure-class a b)
definition of-pure-typ a b = rec-typ
  (ap2 a (b :: Type)) (of-string a b) (of-list a b snd))
  (ap2 a (b :: TFree) (of-string a b) (of-pure-sort a b))
  (ap2 a (b :: TVar) (of-pure-indexname a b) (of-pure-sort a b))
definition of-pure-term a b = (λf0 f1 f2 f3 f4 f5. rec-term f0 f1 f2 f3 (co2 K f4) (λ- -. f5))
  (ap2 a (b :: Const) (of-string a b) (of-pure-typ a b))
  (ap2 a (b :: Free) (of-string a b) (of-pure-typ a b))
  (ap2 a (b :: Var) (of-pure-indexname a b) (of-pure-typ a b))
  (ap1 a (b :: Bound) (of-nat a b))
  (ar3 a (b :: Abs) (of-string a b) (of-pure-typ a b))
  (ar2 a (b :: App) id)

end

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

end
4. Printing Meta-Models

4.1. Initializing the Printer

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

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

4.1.1. Kernel Code for Target Languages

lazy-code-printing code-module CodeType ⇒ (Haskell):
  type MlInt = Integer;
  type MlMonad 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_Printfsprintf1 pat)
      hClose h)

| outStand1 :: ((String → String → IO ()) → IO ()) → IO ()
| outStand1 f = f (\pat → putStrLn . CodeConst_Printfsprintf1 pat)
| code-module CodeConst.Monad ⇒ (Haskell):
  bind a = (>>=) a
| return :: a → IO a
| return = Prelude.return
| code-module CodeConst_Printf ⇒ (Haskell):
  import Text.Printf
  sprintf0 = id

| sprintf1 :: PrintfArg a => String → a → String
| sprintf1 = printf

| sprintf2 :: PrintfArg a => PrintfArg b => String → a → b → String
; `sprintf2 = printf`

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

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

; `sprintf5 :: PrintfArg a => PrintfArg b => PrintfArg c => PrintfArg d => PrintfArg e => String -> a -> b -> c -> d -> e -> String`
; `sprintf5 = printf`

| code-module CodeConst.String → (Haskell) |
| __concat s [] = [] |
| __concat s (x : xs) = x ++ __concatMap ((++) s) xs |

| code-module CodeConst.Sys → (Haskell) |
| import System.Directory |
; `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.eprintf File exists \%S\n file else () in
      let oc = open-out file in
      let b = f (fun s a -> try Some (Printf.fprintf oc s a) with - -> None) in
      let () = close-out oc in
      b
    with - -> None
  let outStand1 f =
    f (fun s a -> try Some (Printf.fprintf stdout s a) with - -> None)
end

module Monad = struct
  let bind = function
    None -> fun - -> None
  | Some a -> fun f - -> f a
  let return a = Some a
end

36
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.of-int-of-big-int (big-int x)
end

⟩

```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 |
\[ x :: xs => aux (String.extract (x, 1, NONE) :: hd l-s :: acc) xs (tl l-s) \text{ in} String.concat (x :: aux [] xs l) \]

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]

structure String = struct
val concat = String.concatWith
end

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

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

fun outFile1 f file =
let
val pfile = Path.explode file
val () = if File.exists pfile then error (File exists \ ~ file \n) else ()
val oc = Unsynchronized.ref []
val - = f (fn a => fn b => SOME (oc := Printf.printf1 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

\[ \ 4.1.2. \ Interface \ with \ Types \  \]

\textbf{datatype} \ ml-int = ML-int
\textbf{code-printing type-constructor} \ ml-int \to (Haskell) CodeType.MlInt
    | type-constructor ml-int \to (OCaml) CodeType.mlInt
    | type-constructor ml-int \to (Scala) CodeType.mlInt
    | type-constructor ml-int \to (SML) CodeType.mlInt

\textbf{datatype} \ 'a ml-monad = ML-monad 'a
\textbf{code-printing type-constructor} \ ml-monad \to (Haskell) CodeType.MlMonad -
type-constructor ml-monad $\rightarrow$ (OCaml) - CodeType.mlMonad

| type-constructor ml-monad $\rightarrow$ (Scala) CodeType.mlMonad |
| type-constructor ml-monad $\rightarrow$ (SML) - CodeType.mlMonad |

type-synonym ml-string = String.literal

4.1.3. Interface with Constants

module CodeConst

consts out-file1 :: (ml-string $\Rightarrow$ 'a1 $\Rightarrow$ unit ml-monad) (* fprintf *) $\Rightarrow$ unit ml-monad $\Rightarrow$ ml-string $\Rightarrow$ unit ml-monad

code-printing constant out-file1 $\rightarrow$ (Haskell) CodeConst.outFile1
| constant out-file1 $\rightarrow$ (OCaml) CodeConst.outFile1 |
| constant out-file1 $\rightarrow$ (Scala) CodeConst.outFile1 |
| constant out-file1 $\rightarrow$ (SML) CodeConst.outFile1 |

consts out-stand1 :: (ml-string $\Rightarrow$ 'a1 $\Rightarrow$ unit ml-monad) (* fprintf *) $\Rightarrow$ unit ml-monad $\Rightarrow$ unit ml-monad

code-printing constant out-stand1 $\rightarrow$ (Haskell) CodeConst.outStand1
| constant out-stand1 $\rightarrow$ (OCaml) CodeConst.outStand1 |
| constant out-stand1 $\rightarrow$ (Scala) CodeConst.outStand1 |
| constant out-stand1 $\rightarrow$ (SML) CodeConst.outStand1 |

module Monad

consts bind :: 'a ml-monad $\Rightarrow$ ('a $\Rightarrow$ 'b ml-monad) $\Rightarrow$ 'b ml-monad

code-printing constant bind $\rightarrow$ (Haskell) CodeConst.Monad.bind
| constant bind $\rightarrow$ (OCaml) CodeConst.Monad.bind |
| constant bind $\rightarrow$ (Scala) CodeConst.Monad.bind |
| constant bind $\rightarrow$ (SML) CodeConst.Monad.bind |

consts return :: 'a $\Rightarrow$ 'a ml-monad

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

module Printf

consts sprintf0 :: ml-string $\Rightarrow$ ml-string

code-printing constant sprintf0 $\rightarrow$ (Haskell) CodeConst_Printfsprintf0
| constant sprintf0 $\rightarrow$ (OCaml) CodeConst_Printfsprintf0 |
| constant sprintf0 $\rightarrow$ (Scala) CodeConst_Printfsprintf0 |
| constant sprintf0 $\rightarrow$ (SML) CodeConst_Printfsprintf0 |

consts sprintf1 :: ml-string $\Rightarrow$ 'a1 $\Rightarrow$ ml-string

code-printing constant sprintf1 $\rightarrow$ (Haskell) CodeConst_Printfsprintf1
| constant sprintf1 $\rightarrow$ (OCaml) CodeConst_Printfsprintf1 |

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

\textbf{consts}  \text{sprintf2}  \colon  \text{ml-string}  \Rightarrow  \text{ml-string}  \\
\textbf{code-printing constant}  \text{sprintf2}  \mapsto  \text{(Haskell)} \; \text{CodeConst.Printf}.sprintf2  \\
| constant  sprintf2  \mapsto  \text{(OCaml)} \; \text{CodeConst.Printf}.sprintf2  \\
| constant  sprintf2  \mapsto  \text{(Scala)} \; \text{CodeConst.Printf}.sprintf2  \\
| constant  sprintf2  \mapsto  \text{(SML)} \; \text{CodeConst.Printf}.sprintf2  \\

\textbf{consts}  \text{sprintf3}  \colon  \text{ml-string}  \Rightarrow  \text{ml-string}  \\
\textbf{code-printing constant}  \text{sprintf3}  \mapsto  \text{(Haskell)} \; \text{CodeConst.Printf}.sprintf3  \\
| constant  sprintf3  \mapsto  \text{(OCaml)} \; \text{CodeConst.Printf}.sprintf3  \\
| constant  sprintf3  \mapsto  \text{(Scala)} \; \text{CodeConst.Printf}.sprintf3  \\
| constant  sprintf3  \mapsto  \text{(SML)} \; \text{CodeConst.Printf}.sprintf3  \\

\textbf{consts}  \text{sprintf4}  \colon  \text{ml-string}  \Rightarrow  \text{ml-string}  \\
\textbf{code-printing constant}  \text{sprintf4}  \mapsto  \text{(Haskell)} \; \text{CodeConst.Printf}.sprintf4  \\
| constant  sprintf4  \mapsto  \text{(OCaml)} \; \text{CodeConst.Printf}.sprintf4  \\
| constant  sprintf4  \mapsto  \text{(Scala)} \; \text{CodeConst.Printf}.sprintf4  \\
| constant  sprintf4  \mapsto  \text{(SML)} \; \text{CodeConst.Printf}.sprintf4  \\

\textbf{consts}  \text{sprintf5}  \colon  \text{ml-string}  \Rightarrow  \text{ml-string}  \\
\textbf{code-printing constant}  \text{sprintf5}  \mapsto  \text{(Haskell)} \; \text{CodeConst.Printf}.sprintf5  \\
| constant  sprintf5  \mapsto  \text{(OCaml)} \; \text{CodeConst.Printf}.sprintf5  \\
| constant  sprintf5  \mapsto  \text{(Scala)} \; \text{CodeConst.Printf}.sprintf5  \\
| constant  sprintf5  \mapsto  \text{(SML)} \; \text{CodeConst.Printf}.sprintf5  \\

\text{module String}  \\
\textbf{consts}  \text{String-concat}  \colon  \text{ml-string}  \Rightarrow  \text{ml-string list}  \Rightarrow  \text{ml-string}  \\
\textbf{code-printing constant}  \text{String-concat}  \mapsto  \text{(Haskell)} \; \text{CodeConst.String}.concat  \\
| constant  \text{String-concat}  \mapsto  \text{(OCaml)} \; \text{CodeConst.String}.concat  \\
| constant  \text{String-concat}  \mapsto  \text{(Scala)} \; \text{CodeConst.String}.concat  \\
| constant  \text{String-concat}  \mapsto  \text{(SML)} \; \text{CodeConst.String}.concat  \\

\text{module Sys}  \\
\textbf{consts}  \text{Sys-is-directory2}  \colon  \text{ml-string}  \Rightarrow  \text{bool ml-monad}  \\
\textbf{code-printing constant}  \text{Sys-is-directory2}  \mapsto  \text{(Haskell)} \; \text{CodeConst.Sys}.isDirectory2  \\
| constant  \text{Sys-is-directory2}  \mapsto  \text{(OCaml)} \; \text{CodeConst.Sys}.isDirectory2  \\
| constant  \text{Sys-is-directory2}  \mapsto  \text{(Scala)} \; \text{CodeConst.Sys}.isDirectory2  \\
| constant  \text{Sys-is-directory2}  \mapsto  \text{(SML)} \; \text{CodeConst.Sys}.isDirectory2  \\

\text{module To}  \\
\textbf{consts}  \text{ToNat}  \colon  \text{nat}  \Rightarrow  \text{integer}  \Rightarrow  \text{nat}  \Rightarrow  \text{ml-int}  \\
\textbf{code-printing constant}  \text{ToNat}  \mapsto  \text{(Haskell)} \; \text{CodeConst.To}.nat  \\
| constant  \text{ToNat}  \mapsto  \text{(OCaml)} \; \text{CodeConst.To}.nat  \\
| constant  \text{ToNat}  \mapsto  \text{(Scala)} \; \text{CodeConst.To}.nat  \\
| constant  \text{ToNat}  \mapsto  \text{(SML)} \; \text{CodeConst.To}.nat
4.1.4. Some Notations (I): Raw Translations

```plaintext
syntax -sprint0 :: - ⇒ ml-string (sprint0 (-)´)
translations sprint0 x´ ≝ CONST sprintf0 x

syntax -sprint1 :: - ⇒ - ⇒ ml-string (sprint1 (-)´)
translations sprint1 x´ ≝ CONST sprintf1 x

syntax -sprint2 :: - ⇒ - ⇒ ml-string (sprint2 (-)´)
translations sprint2 x´ ≝ CONST sprintf2 x

syntax -sprint3 :: - ⇒ - ⇒ ml-string (sprint3 (-)´)
translations sprint3 x´ ≝ CONST sprintf3 x

syntax -sprint4 :: - ⇒ - ⇒ ml-string (sprint4 (-)´)
translations sprint4 x´ ≝ CONST sprintf4 x

syntax -sprint5 :: - ⇒ - ⇒ ml-string (sprint5 (-)´)
translations sprint5 x´ ≝ CONST sprintf5 x
```

4.1.5. Some Notations (II): Polymorphic Cartouches

```plaintext
syntax -cartouche-string´ :: String.literal
translations -cartouche-string = -cartouche-string´

⟨ML⟩
```

4.1.6. Generic Locale for Printing

```plaintext
locale Print =
  fixes To-string :: string ⇒ ml-string
  fixes To-nat :: nat ⇒ ml-int
begin
  declare[cartouche-type´ = fun printf]
end
```

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

end

4.2. Instantiating the Printer for (Pure) Term

```plaintext
theory Printer-Pure
imports Meta-Pure
  Printer-init
begin

context Print
begin

```

42
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 of-semi-val-fun = (λ Sval ⇒ ⟨val⟩ |
      Sfun ⇒ ⟨fun⟩)

  fun′ of-semi-term′ where of-semi-term′ e = (λ
      SML-string s ⇒ ⟨%s⟩ (To-string (escape-sml s)) |
      SML-rewrite val-fun e1 symb e2 ⇒ ⟨%s%ss%%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 ⇒ ⟨%ss%ss⟩ (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)%s⟩ (String-concat ⟨⟩ (List.map (λ e ⇒ ⟨%(s)%s⟩ (of-semi-term′ e)) l)) |
      SML-apply e l ⇒ ⟨%(s%ss)%s⟩ (of-semi-term′ e) (String-concat ⟨⟩ (List.map (λ e ⇒ ⟨%(s)%s⟩ (of-semi-term′ e)) l)) |
      SML-paren p-left p-right e ⇒ ⟨%ss%ss⟩ (To-string p-left) (of-semi-term′ e) (To-string p-right) |
      SML-let-open s e ⇒ ⟨let open %s in %s end⟩ (To-string s) (of-semi-term′ e)⟩ e
end
lemmas [code] =
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 e = (λ
  Typ-base s ⇒ To-string s
  | Typ-apply name l ⇒ ⟨%
  s%
  ⟩ (let s = String-concat ⟨,⟩ (List.map of-semi--typ l) in
    case l of [] ⇒ s | _ ⇒ (⟨%s⟩ s)
  )
  (of-semi--typ name)
  | Typ-apply-bin s ty1 ty2 ⇒ ⟨%
  s%
  %
  s%
  ⟩ (of-semi--typ ty1) (To-string s) (of-semi--typ ty2)
  | Typ-apply-paren s1 s2 ty ⇒ ⟨%
  s%
  %
  s%
  ⟩ (To-string s1) (of-semi--typ ty) (To-string s2)
  ) e

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

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

fun of-semi--term where of-semi--term e = (λ
  Term-rewrite e1 symb e2 ⇒ ⟨%s
  %s
  ⟩ (of-semi--term e1) (To-string symb) (of-semi--term e2)
definition of-type-notation - = (λ Type-notation n e ⇒
    (type-notation %s (%s)) (To-string n) (To-string e))

definition of-instantiation - = (λ Instantiation n n-def expr ⇒
    let name = To-string n in
    instantiation %s :: object
    begin
      definition %s-%s-def : %s
      instance ..
    end;
    name
    (To-string n-def)
    name
    (of-semi-term expr))

definition of-defs - = (λ Defs-overloaded n e ⇒
    (defs(overloaded)) %s :: %s (To-string n) (of-semi-term e))

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

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

definition (of-semi-thm-attribute-aux-gen :: String.literal × String.literal ⇒ → → ⇒) m lacc
s =
    (let s-base = (λs lacc. %s%(%s)) (To-string s) (String-concat _ _) (L.map (λ(s, x). %s %s) s x)
definition of-semi-thm-attribute-aux-gen-where \( l = \langle \text{where}, \text{String-concat} \circ \text{and} ; (L.\text{map} (\lambda (\text{var}, \text{expr}). \% s = \% s) (\text{To-string} \ \text{var}) (\text{of-semi-term} \ \text{expr})) \rangle \)

definition of-semi-thm-attribute-aux-gen-of \( l = \langle \text{of}, \text{String-concat} \circ ; (L.\text{map} (\lambda \text{expr}. \% s \ (\text{of-semi-term} \ \text{expr})) \rangle \)

fun of-semi-thm-attribute-aux where of-semi-thm-attribute-aux lacc e =
(\lambda \text{Thm-thm} s \Rightarrow \text{To-string} s |
\text{Thm-thms} s \Rightarrow \text{To-string} s
|
\text{Thm-THEN} (\text{Thm-thm} s) e2 \Rightarrow of-semi-thm-attribute-aux-gen ((\text{THEN}), of-semi-thm-attribute-aux [\[] e2) \ lacc \ s |
\text{Thm-THEN} (\text{Thm-thms} s) e2 \Rightarrow of-semi-thm-attribute-aux-gen ((\text{THEN}), of-semi-thm-attribute-aux [\[] e2) \ lacc \ s |
\text{Thm-THEN} e1 e2 \Rightarrow of-semi-thm-attribute-aux ((\langle \text{THEN} \rangle, of-semi-thm-attribute-aux [\[] e2) \ # \ (\text{lacc}) \ e1
|
\text{Thm-simplified} (\text{Thm-thm} s) e2 \Rightarrow of-semi-thm-attribute-aux-gen ((\text{simplified}), of-semi-thm-attribute-aux [\[] e2) \ lacc \ s |
\text{Thm-simplified} (\text{Thm-thms} s) e2 \Rightarrow of-semi-thm-attribute-aux-gen ((\text{simplified}), of-semi-thm-attribute-aux [\[] e2) \ lacc \ s |
\text{Thm-simplified} e1 e2 \Rightarrow of-semi-thm-attribute-aux ((\langle \text{simplified} \rangle, of-semi-thm-attribute-aux [\[] e2) \ # \ (\text{lacc}) \ e1
|
\text{Thm-symmetric} (\text{Thm-thm} s) \Rightarrow of-semi-thm-attribute-aux-gen ((\text{symmetric}), (\text{of-lacc}) \ s |
\text{Thm-symmetric} (\text{Thm-thms} s) \Rightarrow of-semi-thm-attribute-aux-gen ((\text{symmetric}), (\text{of-lacc}) \ s |
\text{Thm-symmetric} e1 \Rightarrow of-semi-thm-attribute-aux ((\langle \text{symmetric} \rangle, (\text{of-lacc}) \ # \ (\text{lacc}) \ e1
|
\text{Thm-where} (\text{Thm-thm} s) l \Rightarrow of-semi-thm-attribute-aux-gen ((\text{of-semi-thm-attribute-aux-gen-where l}) \ lacc \ s |
\text{Thm-where} (\text{Thm-thms} s) l \Rightarrow of-semi-thm-attribute-aux-gen ((\text{of-semi-thm-attribute-aux-gen-where l}) \ lacc \ s |
\text{Thm-where} e1 l \Rightarrow of-semi-thm-attribute-aux ((\text{of-semi-thm-attribute-aux-gen-where l}) \ # \ (\text{lacc}) \ e1
|
\text{Thm-of} (\text{Thm-thm} s) l \Rightarrow of-semi-thm-attribute-aux-gen ((\text{of-semi-thm-attribute-aux-gen-of l}) \ lacc \ s |
\text{Thm-of} (\text{Thm-thms} s) l \Rightarrow of-semi-thm-attribute-aux-gen ((\text{of-semi-thm-attribute-aux-gen-of l}) \ lacc \ s |
\text{Thm-of} e1 l \Rightarrow of-semi-thm-attribute-aux ((\text{of-semi-thm-attribute-aux-gen-of l}) \ # \ (\text{lacc}) \ e1
|
\text{Thm-OF} (\text{Thm-thm} s) e2 \Rightarrow of-semi-thm-attribute-aux-gen ((\text{OF}), of-semi-thm-attribute-aux [\[ e2) \ lacc \ s |
\text{Thm-OF} (\text{Thm-thms} s) e2 \Rightarrow of-semi-thm-attribute-aux-gen ((\text{OF}), of-semi-thm-attribute-aux [\[ e2) \ lacc \ s

46
Method-drule \( s \Rightarrow \text{drule} \% s \) (of-semi-thm-attribute \( s \))
Method-erule \( s \Rightarrow \text{erule} \% s \) (of-semi-thm-attribute \( s \))
Method-intro \( l \Rightarrow \text{intro} \% s \) (of-semi-thm-attribute-l \( l \))
Method-elim \( s \Rightarrow \text{elim} \% s \) (of-semi-thm-attribute \( s \))
Method-subst asm \( l \Rightarrow s \)
  let \( s-asm \) = if asm then \( \text{(asm)} \) \else \( \text{in} \)
  if \( \text{L.map String.to-list} \ l = \text{["0"]} \) then
    \( \text{⟨ subst \% \% s \% s \% s - asm (of-semi-thm-attribute \% s) } \text{ else} \)
    \( \text{⟨ subst \% \% s \% s \% s - asm (String-concat \% s \% (L.map To-string \ l)) (of-semi-thm-attribute \% s) } \)
Method-insert \( l \Rightarrow \text{insert} \% s \) (of-semi-thm-l \( l \))
Method-plus \( t \Rightarrow \text{⟨ \% s \% s \% String-concat \% s \% (List.map of-semi-method \% t) } \)
Method-option \( t \Rightarrow \text{⟨ \% s \% s \% String-concat \% s \% (List.map of-semi-method \% t) } \)
Method-or \( t \Rightarrow \text{⟨ \% s \% s \% String-concat \% s \% (List.map of-semi-method \% t) } \)
Method-one \( \text{⟨ Method-simp-only \% l } \Rightarrow \text{⟨ simp only \% s \% s - (of-semi-thm-l \% l) } \)
Method-one \( \text{⟨ Method-simp-add-del-split \% l2 \% l3 } \Rightarrow \text{⟨ simp \% s \% s \% s - (of-semi-thm-l \% l2 \% l3) } \)
Method-all \( \text{⟨ Method-simp-only \% l } \Rightarrow \text{⟨ simp all only \% s \% s - (of-semi-thm-l \% l) } \)
Method-all \( \text{⟨ Method-simp-add-del-split \% l2 \% l3 } \Rightarrow \text{⟨ simp all \% s \% s - (of-semi-thm-l \% l2 \% l3) } \)
Method-auto-simp-add-del-split \( \text{l-simp l-split } \Rightarrow \text{⟨ auto \% s \% s - (of-semi-thm-l \% l-simp \% l-split) } \)
Method-rename-tac \( l \Rightarrow \text{⟨ rename-tac \% s \% (String-concat \% s \% (L.map To-string \ l)) } \)
Method-case-tac \( e \Rightarrow \text{⟨ case-tac \% s \% (of-semi-term \% e) } \)
Method-blast \( \text{None } \Rightarrow \text{⟨ blast \% s - (To-nat \% n) } \)
Method-clarify \( \Rightarrow \text{⟨ clarify \% s - (String-concat \% s \% (L.map To-string \ l-opt) } \)
\text{⟨ if l-opt = \% s - (String-concat \% s \% (L.map To-string \ l-opt) } \)
\text{⟨ else} \)
\( \text{⟨ (of-semi-thm-attribute-l \% l) } \text{ expr} \)

**definition** of-semi-command-final = \( \lambda \text{Command-done } \Rightarrow \text{⟨ done \% s - (String-concat \% s \% (L.map To-string \ l-apply) } \)
\text{⟨ Command-by l-apply } \Rightarrow \text{⟨ by \% s \% (String-concat \% s \% (L.map To-string \ l-apply) } \)
\text{⟨ Command-sorry } \Rightarrow \text{⟨ sorry \% s - (String-concat \% s \% (L.map To-string \ l-simp) } \)
\text{⟨ else} \)
\( \text{⟨ (of-semi-thm-attribute-l \% l) } \text{ expr} \)

48
definition of-semi--command-state = ( λ Command-apply-end [] ⇒ o 
  Command-apply-end l-apply ⇒ λ apply-end(%s) 
  (String-concat ⟨ , ⟩ (L.map of-semi--method l-apply)))

definition' (of-semi--command-proof = ( 
  let thesis = ⟨ thesis ⟩ ; scope-thesis-gen = ⟨ proof = %s show %s in ⟩ ; scope-thesis = λ s. scope-thesis-gen s thesis in λ Command-apply [] ⇒ o 
  Command-apply l-apply ⇒ λ apply(%s) 
  (String-concat ⟨ , ⟩ (L.map of-semi--method l-apply)) 
  Command-using l ⇒ λ using %s 
  (of-semi--thm-l l) 
  Command-unfolding l ⇒ λ unfolding %s 
  (of-semi--thm-l l) 
  Command-let e-name e-body ⇒ scope-thesis (λ %s = %s; (of-semi--term e-name) (of-semi--term e-body)) 
  Command-have n b e e-last ⇒ scope-thesis (λ have %s %s; %s %s; (To-string n) (if b then ![simp] else o) (of-semi--term e) (of-semi--command-final e-last)) 
  Command-fix-let l l-let o-show - ⇒ scope-thesis-gen 
  (String-concat ⟨ ⟩ (L.map To-string l)) 
  (String-concat (L.map (λ (e-name, e-body). 
      let %s = %s; (of-semi--term e-name) (of-semi--term e-body)) 
      l-let)) 
  (case o-show of None ⇒ thesis 
   Some l-show ⇒ ⟨ %s; (String-concat ⟨ ⟩ (L.map of-semi--term l-show))⟩)

definition of-lemma - = (λ Lemma n l-spec l-apply tactic-last ⇒ 
  ⟨ lemma %s : %s 
   %s %s; (To-string n) 
   (String-concat ⟨ ⟩ (L.map of-semi--term l-spec)) 
   (String-concat (λ (l [] ⇒ o | l-apply ⇒ λ apply(%s) 
       (String-concat (λ (L.map of-semi--method l-apply)) 
       l-apply)))) 
   (of-semi--command-final tactic-last) 
   Lemma-assumes n l-spec concl l-apply tactic-last ⇒ 
   ⟨ lemma %s : %s 
    %s %s %s; (To-string n) 
    (String-concat (λ (L.map (λ (n, b, e). 

49
assumes \(\%s\%s\)

\[
\begin{aligned}
\text{let } (n, b) &= \text{if } b \text{ then } (\%s[\text{simp}]; \text{To-string } n, \text{False}) \text{ else } (\text{To-string } n, \text{String.is-empty } n) \text{ in } \\
\text{if } b \text{ then } \text{\%s} \text{ else } \text{\%s} \text{ n}
\end{aligned}
\]

\[\text{l-spec}\]

shows \(\%s\) \(\text{(of-semi-term concl)\})\)

\[
\begin{aligned}
\text{String-concat } \text{\%s} \text{ (L.map of-semi-command-proof l-apply))} \\
\text{(of-semi--command-final tactic-last)} \\
\text{(String-concat \{\})} \\
\text{(L.map (\lambda \text{l-apply-e.}} \\
\text{\%sqed) \text{ if l-apply-e = [] then}} \\
\text{\} else}} \\
\text{\%s) \\
\text{(String-concat } \text{\%s} \text{ \text{(of-semi--command-state l-apply-e))))) \\
\text{(List.map-filter}} \\
\text{\text{(\lambda \text{Command-let - - \Rightarrow Some [] | Command-have - - - - \Rightarrow Some [] | Command-fix-let - - l \Rightarrow Some l | - \Rightarrow None)}} \\
\text{(rev l-apply))\))\}
\end{aligned}
\]

definition of-axiomatization - = (\lambda \text{Axiomatization n e \Rightarrow \{axiomatization where \%s:
\%s \text{ (To-string n) \text{ (of-semi--term e)\}}
\}
\]
definition of-section - = (\lambda \text{Section n section-title \Rightarrow
\%s \text{ \%section (if n = 0 then \%s else if n = 1 then \{sub\} else \{subsub\} \})
\text{ (To-string section-title)\})
\]
definition of-text - = (\lambda \text{Text s \Rightarrow \{text \%s\} (To-string s)\})
\]
definition of-ML - = (\lambda \text{SML e \Rightarrow \{ML \%s\} \text{ (of-semi--term' e)\})
\]
definition of-setup - = (\lambda \text{Setup e \Rightarrow \{setup \%s\} \text{ (of-semi--term' e)\})
\]
definition of-thm - = (\lambda \text{Thm thm \Rightarrow \{thm \%s\} \text{ (of-semi--thm-attribute-II thm)\})
\]
definition \text{of-interpretation - = (\lambda \text{Interpretation n loc-n loc-param tac \Rightarrow
\{interpretation \%s; \%s\%s\%s \text{ (To-string n) \text{ (To-string loc-n)\}}
\} \text{ (rev apply))\}}\)
\]
definition of-semi-theory env =
(λ Theory-datatype dataty ⇒ of-datatype env dataty
 | Theory-type-synonym ty-synonym ⇒ of-type-synonym env ty-synonym
 | Theory-type-notation ty-notation ⇒ of-type-notation env ty-notation
 | Theory-instantiation instantiation-class ⇒ of-instantiation env instantiation-class
 | Theory-defs defs-overloaded ⇒ of-defs env defs-overloaded
 | Theory-consts consts-class ⇒ of-consts env consts-class
 | Theory-definition definition-hol ⇒ of-definition env definition-hol
 | Theory-lemmas lemmas-simp ⇒ of-lemmas env lemmas-simp
 | Theory-lemma lemma-by ⇒ of-lemma env lemma-by
 | Theory-axiomatization axiom ⇒ of-axiomatization env axiom
 | Theory-section section-title ⇒ of-section env section-title
 | Theory-text text ⇒ of-text env text
 | Theory-ML ml ⇒ of-ML env ml
 | Theory-setup setup ⇒ of-setup env setup
 | Theory-thm thm ⇒ of-thm env thm
 | Theory-interpretation thm ⇒ of-interpretation env thm)

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

definition′ (of-semi-theories env =
(λ Theories-one t ⇒ of-semi-theory env t
 | Theories-locale data l ⇒
   locale %s =
   %s
   begin
   %s
   end) (To-string (HolThyLocale-name data))
   (String-concat-map
   ⟨
   ⟩
   (λ (l-fix, o-assum).
     (%s,%s) (String-concat-map :
     ⟩
   (λ(e, ty). (fixes %s :: %s) (of-semi-term e) (of-semi-typ ty)) l-fix)
   (case o-assum of None ⇒ ⟨⟩
   assumes %s: %s (To-string name) (of-semi-term e)))
   (HolThyLocale-header data))
   (String-concat-map :
   ⟩
   (of-semi-theory env)) l))
end
lemmas \[\text{code} \] =

Print.of-datatype-def
Print.of-type-synonym-def
Print.of-type-notation-def
Print.of-instantiation-def
Print.of-defs-def
Print.of-consts-def
Print.of-definition-def
Print.of-semi--thm-attribute-aux-gen-def
Print.of-semi--thm-attribute-aux-gen-where-def
Print.of-semi--thm-attribute-aux-gen-of-def
Print.of-semi--thm-attribute-def
Print.of-semi--thm-def
Print.of-semi--thm-attribute-l-def
Print.of-semi--thm-attribute-l1-def
Print.of-semi--thm-l-def
Print.of-lemmas-def
Print.of-semi--attrib-genA-def
Print.of-semi--attrib-genB-def
Print.of-semi--attrib-def
Print.of-semi--attrib1-def
Print.of-semi--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-ML-def
Print.of-setup-def
Print.of-thm-def
Print.of-interpretation-def
Print.of-semi--theory-def
Print.String-concat-map-def
Print.of-semi--theories-def

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 l. toy-class-raw.make n l [] False in
[mk (n ("Galaxy") None) ([("sound"), ToyTy-raw ("unit")), ("moving"), ToyTy-raw ("bool")])
, mk (n ("Planet") (Some ("Galaxy"))) [("weight"), ToyTy-raw ("nat")])
, mk (n ("Person") (Some ("Planet"))) [("salary"), ToyTy-raw ("int")]) ])

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

53
5.1.2. Statically Executing the Exportation

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

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

end

5.2. Dynamic Meta Embedding with Reflection

theory Generator-dynamic
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.

apply-code-printing-reflect

val stdout-file = Unsynchronized.ref

This variable is not used in this theory (only in Generator_static.thy), but needed for well typechecking the reflected SML code.
code-reflect open META
functions
fold-thy-deep fold-thy-shallow

write-file

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

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

5.2.1. Interface Between the Reflected and the Native
⟨ML⟩

5.2.2. Binding of the Reflected API to the Native API
⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩langle 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 ⟨∀⟩state =
  heap :: oid ⇒ ∀
  assocs :: oid ⇒ ((oid list) list) list

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

end
```

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

5.4.1. Introduction

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

⟨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).

59
```
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):

```plaintext
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.

\begin{verbatim}
End End End
\end{verbatim}

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}.

\textbf{Class} \textit{Molecule} < \textit{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}

\textbf{Class} \textit{Galaxy}
\begin{verbatim}
Attributes wormhole : UnlimitedNatural
  is-sound : Void
End!
\end{verbatim}

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.

\textbf{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).

\textbf{Class} \textit{Person} < \textit{Galaxy}
\begin{verbatim}
Attributes salary : Integer
  boss : Person
  is-meta-thinking: Boolean
\end{verbatim}

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.

\textbf{Instance} \texttt{X_{Person}1 :: Person = [ salary = 1300 , boss = \texttt{X_{Person}2} ]}
\texttt{and X_{Person}2 :: Person = [ salary = 1800 ]}
Here we will call \textbf{Instance} again to show that the universe will not be computed again since it was already computed in the previous \textbf{Instance}.

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

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

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

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

\subsection*{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 \textbf{deep} mode, this is particularly not a danger for meta-commands to generate themselves (whereas for \textbf{shallow} the recursion might not terminate).

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

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

\texttt{State} $\sigma_1 = [ [ (\text{salary} = 1000, \text{boss} = \text{self}^1) :: Person ]$

\texttt{, [ (\text{salary} = 1200 ) :: Person ]$

\texttt{, [ (\text{salary} = 2600, \text{boss} = \text{self}^3) :: Person ]$

\texttt{, [ $X_{Person}^1$ ]$

\texttt{, [ [ (\text{salary} = 2300, \text{boss} = \text{self}^2) :: Person ]$

\texttt{, [ $X_{Person}^2$ ]}$

\texttt{State} $\sigma_1' = [ [ X_{Person}^1 ]$

\texttt{, [ X_{Person}^2 ]$

\texttt{, [ X_{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`.

**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:

(* **PrePost** \(\sigma_1\) \(\sigma_1'\) [ ([ salary = 1000 , boss = self 1 ] :: Person) ] *)

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

(* State WFF_10_post = [ ([ "salary" = 1000 , "boss" = self 1 ] :: Person) ]

**PrePost**[shallow] \(\sigma_1\) WFF_10_post *) (WFF-10-post is an automatically generated name).

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

(* **Instance** WFF_10_post_object0 :: Person = [ "salary" = 1000 , "boss" = [ ] ]

**State**[shallow] WFF_10_post = [ WFF_10_post_object0 ]

**PrePost**[shallow] \(\sigma_1\) WFF_10_post *)

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

Meta-commands are obviously not restricted to manipulate expressions in the Outer Syntax level. It is possible to build meta-commands so that Inner Syntax expressions are directly parsed. However the dependencies of this theory have been minimized so that experimentations and debugging can easily occur in **deep** mode (this file only depends on **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:

Context Person :: content ()

Post "</close>"
Here the expression `<close>` 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:

```plaintext
(* 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:

```plaintext
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:

```plaintext
(* 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⟩
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 $\sigma_1 =$

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

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

(ML)

A.1. Main Setup of Meta Commands

\[
\text{generation-syntax} : \text{theory} \rightarrow \text{theory}
\]
generation-syntax sets the behavior of all incoming meta-commands. By default, without firstly writing `generation-syntax` meta-commands will only print in output what they have parsed, this is similar as giving to `generation-syntax` a non-empty list having only `syntax-print` as elements (on the other hand, nothing is printed when an empty list is received). Additionally `syntax-print` can be followed by an integer indicating the printing depth in output, similar as declaring `ML-print-depth` with an integer, but the global option `syntax-print` is restricted to meta-commands. Besides the printing of syntaxes, several options are provided to further analyze the semantics of languages being embedded, and tell if their evaluation should occur immediately using the `shallow` mode, or to only display what would have been evaluated using the `deep` mode (i.e., to only show the generated Isabelle content in the output window). Since several occurrences of `deep`, `shallow`, or `syntax-print` can appear in the parameterizing list, for each meta-command the overall evaluation respects the order of events given in the list (from head to tail). At the time of writing, it is only possible to evaluate this list sequentially: the execution stops as soon as one first error is raised, thus ignoring remaining events.

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

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

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

- For safety reasons, it simply suffices to extract all the meta-compiler together with the respective arguments in front of each incoming meta-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, \texttt{SORRY} allows to explicitly skip the evaluation of all proofs, irrespective of the presence of \texttt{sorry} or not in generated proofs. In any cases, the semantics of \texttt{sorry} has not been overloaded, e.g., red background may appear as usual.

Finally, \texttt{generation-semantics} is a container for specifying various options for varying the semantics of languages being embedded. For example, \texttt{design} and \texttt{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 \texttt{eager} or \texttt{lazy} for choosing how the evaluation should happen...

### A.2. All Meta Commands of the Toy Language

\[
\begin{align*}
\text{Class} & : \text{theory } \rightarrow \text{theory} \\
\text{Abstract-class} & : \text{theory } \rightarrow \text{theory}
\end{align*}
\]

\[
\begin{tikzpicture}
\node (class) {Class};
\node (abstract_class) [below of=class] {Abstract_class};
\node (binding) [right of=class] {binding} edge [->] (class);
\node (type_base) [right of=binding] {type-base};
\node (type_object) [below of=type_base] {type-object};
\node (class) [below of=type_object] {class};
\end{tikzpicture}
\]
class

Attributes

- binding : toy-type

context

context

Operations

:.

binding : toy-type

binding : toy-type

term = term

Pre

Post

use-prop

inv

inv

invariant

invariant

Constraints

Existential

Aggregation

Association

Composition

: theory → theory

: theory → theory

: theory → theory
**Aggregation**

**Association**

**Composition**

**binding**

**association**

**Between**

**association-end**

**association-end**

**type-object**

**category**

**;**

**Context**

: theory → theory

**Abstract-associationclass**

: theory → theory

**Abstract_associationclass**

**type-object**

**association**

**class**

**aggregation**

**composition**

**Context**

: theory → theory

**type-object**

**context**

1

**shallow**

1

76
Instance  :  theory → theory

State       :  theory → theory
\[
\text{state} = \text{state} \left[ \text{shallow} \right] \text{binding} = \text{state} \\
\text{PrePost} : \text{theory} \rightarrow \text{theory} \\
\text{pre-post} \\
\text{End} : \text{theory} \rightarrow \text{theory} \\
\text{End} = \text{forced} \\
\]
A.3. Extensions of Isabelle Commands

\textbf{fun'} : local-theory → local-theory
\textbf{definition'} : local-theory → local-theory
\textbf{code-reflect'} : theory → theory
fun has the same semantics as fun except that it is possible to write the quote symbol (i.e., the symbol ") in all recursive enclosing cartouches.

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

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

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

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

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

B.1. Changes on Signatures

- ./src/HOL/Tools/Metis/Isabelle_metis_tactic.thy  
  Some signatures was removed for exposing the main structure.
- ./src/HOL/Tools/BNF/Isabelle_bnf_fp_def_sugar.thy  
  Some signatures was removed for exposing the main structure.
- ./src/Provers/Isabelle_classical.thy  
  Some signatures was removed for exposing the main structure.

B.2. Extensions for Cartouches

- ./src/HOL/ex/Isabelle_Cartouche_Examples.thy  
  Some functions have been generalized for supporting cartouches.
- ./src/HOL/Tools/Function/Isabelle_fun.thy  
  This file only contains the definition of fun'.
- ./src/HOL/Tools/Function/Isabelle_function_common.thy  
  Some functions have been generalized for supporting cartouches.
- ./src/Pure/Isar/Isabelle_parse_spec.thy  
  Some functions have been generalized for supporting cartouches.
- ./src/Pure/Isar/Isabelle_isar_syn.thy  
  This file only contains the definition of definition'.

B.3. Other Changes

- ./src/Tools/Code/Isabelle_code_runtime.thy  
  The option open was introduced in this file for the definition of code_reflect'.
- ./src/Tools/Code/Isabelle_code_target.thy  
  Some signatures was removed for exposing the main structure, we have also defined 
  at the end the implementation of lazy_code_printing, apply_code_printing and 
  apply_code_printing_reflect.
- ./src/Pure/Isar/Isabelle_typedecl.thy  
  Short modification of the argument lifting a binding to a binding option with some 
  signatures removed.
C. Content of One Generated File (as example)

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

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

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

```
datatype extAtom = mkExtAtom oid oid list option int option bool option nat option unit option

datatype extAtom = mkAtom extAtom int option

datatype extMolecule = mkExtMolecule extAtom extAtom

| mkExtMolecule oid oid list option int option bool option nat option unit option

datatype extMolecule = mkMolecule extAtom

datatype extPerson = mkExtPerson extPerson extMolecule extMolecule

| mkExtPerson oid nat option unit option

datatype extPerson = mkPerson extPerson extAtom extPerson

| mkExtPerson oid list option int option bool option

datatype extGalaxy = mkExtGalaxy extGalaxy extPerson extPerson

| mkExtGalaxy oid

datatype extGalaxy = mkGalaxy extGalaxy extGalaxy

| mkExtGalaxy nat option unit option

datatype extToyAny = mkExtToyAny extToyAny extGalaxy extGalaxy

| mkExtToyAny pid extToyAny

datatype extToyAny = mkToyAny extToyAny extToyAny extToyAny

| mkExtToyAny oid
```

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 \( \mathcal{A} = in_{\text{Atom}} ty_{\text{Atom}} \mid in_{\text{Molecule}} ty_{\text{Molecule}} \mid in_{\text{Person}} ty_{\text{Person}} \mid in_{\text{Galaxy}} ty_{\text{Galaxy}} \mid in_{\text{ToyAny}} ty_{\text{ToyAny}} \)

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 correspondence of Toy-types to types of the meta-language HOL.

\[
\begin{align*}
type-synonym \text{ Atom} & = \langle \langle ty_{\text{Atom}} \rangle \bot \rangle \bot \\
type-synonym \text{ Molecule} & = \langle \langle ty_{\text{Molecule}} \rangle \bot \rangle \bot \\
type-synonym \text{ Person} & = \langle \langle ty_{\text{Person}} \rangle \bot \rangle \bot \\
type-synonym \text{ Galaxy} & = \langle \langle ty_{\text{Galaxy}} \rangle \bot \rangle \bot \\
type-synonym \text{ ToyAny} & = \langle \langle ty_{\text{ToyAny}} \rangle \bot \rangle \bot
\end{align*}
\]

definition oid_{\text{Atom}}-0---boss = 0
definition oid_{\text{Molecule}}-0---boss = 0
definition oid_{\text{Person}}-0---boss = 0

definition switch_{2\cdot01} = (\lambda [x0, x1] \Rightarrow (x0, x1))
definition switch_{2\cdot10} = (\lambda [x0, x1] \Rightarrow (x1, x0))

definition oid_{1} = 1
definition oid_{2} = 2
definition inst-assoc_{1} = (\lambda oid-class to-from oid. ((case (deref-assocs-list ((to-from::oid list list \Rightarrow oid list \times oid list)) ((oid::oid)) ((drop (((map-of-list (((oid_{\text{Person}}-0---boss , (List.map ((\lambda(x , y) . [x , y]) o switch_{2\cdot01})) (((oid::oid)))))))) of Nil \Rightarrow None \\
| l \Rightarrow (Some (l))))::oid list option))

definition oid_{3} = 3
definition inst-assoc_{3} = (\lambda oid-class to-from oid. ((case (deref-assocs-list ((to-from::oid list list \Rightarrow oid list \times oid list)) ((oid::oid)) ((drop (((((map-of-list ([[]])) (((oid-class::oid)))))))) of Nil \Rightarrow None \\
| l \Rightarrow (Some (l))))::oid list option))

definition oid_{4} = 4
definition oid_{5} = 5
definition oid_{6} = 6
definition oid_{7} = 7
definition inst-assoc4 = (\lambda oid-class to-from oid. ((case (deref-assocs-list ((to-from::oid list list ⇒ oid list × oid list)) (oid::oid)) ((drop (((map-of-list (\((\oid\_person\_0--boss\ , (List.map (\((\lambda(x , y) . [x , y]) \circ switch2-01) (\(((\oid7\ , [\oid6]\) , [\oid6]\) , [\oid1]\) , [\oid4]\) , [\oid5]\)))))))) ((oid-class::oid)))))))) of Nil ⇒ None
  | l ⇒ (Some (l))::oid list option)

locale state-\sigma_1 =
  fixes oid4 :: nat
  fixes oid5 :: nat
  fixes oid6 :: nat
  fixes oid7 :: nat
  fixes oid1 :: nat
  fixes oid2 :: nat
  assumes distinct-oid: (distinct ([oid4 , oid5 , oid6 , oid1 , oid7 , oid2]))
  fixes \sigma_1-object0 Person :: ty Person
  fixes \sigma_1-object0 :: Person
  assumes \sigma_1-object0-def: \sigma_1-object0 = (\lambda- [[\sigma_1-object0 Person]])
  fixes \sigma_1-object1 Person :: ty Person
  fixes \sigma_1-object1 :: Person
  assumes \sigma_1-object1-def: \sigma_1-object1 = (\lambda- [[\sigma_1-object1 Person]])
  fixes \sigma_1-object2 Person :: ty Person
  fixes \sigma_1-object2 :: Person
  assumes \sigma_1-object2-def: \sigma_1-object2 = (\lambda- [[\sigma_1-object2 Person]])
  fixes X Person1 Person :: ty Person
  fixes X Person1 :: Person
  assumes X Person1-def: X Person1 = (\lambda- [[X Person1 Person]])
  fixes \sigma_1-object4 Person :: ty Person
  fixes \sigma_1-object4 :: Person
  assumes \sigma_1-object4-def: \sigma_1-object4 = (\lambda- [[\sigma_1-object4 Person]])
  fixes X Person2 Person :: ty Person
  fixes X Person2 :: Person
  assumes X Person2-def: X Person2 = (\lambda- [[X Person2 Person]])
begin
definition \sigma_1 = (state.make ((Map.empty (oid4 ⇒ (in Person (\sigma_1-object0 Person)))) (oid5 ⇒ (in Person (\sigma_1-object1 Person)))) (oid6 ⇒ (in Person (\sigma_1-object2 Person)))) (oid7 ⇒ (in Person (\sigma_1-object4 Person))))
lemma perm-\sigma_1: \sigma_1 = (state.make ((Map.empty (oid2 ⇒ (in Person (X Person2 Person)))) (oid7 ⇒ (in Person (\sigma_1-object4 Person)))) (oid1 ⇒ (in Person (X Person1 Person)))) (oid6 ⇒ (in Person (\sigma_1-object2 Person)))) (oid5 ⇒ (in Person (\sigma_1-object1 Person)))) (oid4 ⇒ (in Person (\sigma_1-object0 Person))))
  ⟨proof⟩
  end
locale state-σ1' =
fixes oid1 :: nat
fixes oid2 :: nat
fixes oid3 :: nat
assumes distinct-oid: (distinct ([oid1 , oid2 , oid3]))
fixes X_Person1_Person :: ty_Person
fixes X_Person1 :: :: Person
assumes X_Person1-def: X_Person1 = (λ-. [[X_Person1_Person]])
fixes X_Person2_Person :: ty_Person
fixes X_Person2 :: :: Person
assumes X_Person2-def: X_Person2 = (λ-. [[X_Person2_Person]])
fixes X_Person3_Person :: ty_Person
fixes X_Person3 :: :: Person
assumes X_Person3-def: X_Person3 = (λ-. [[X_Person3_Person]])
begin
definition σ1' = (state.make ((Map.empty (oid1 → (in_Person (X_Person1_Person)))) (oid2 → (in_Person (X_Person2_Person)))) (oid3 → (in_Person (X_Person3_Person)))) ((map-of-list ([(oid_Person-0---boss , (List.map ((x , y) [x , y] o switch2-0I) ([[oid1] , [oid2]])])))])
lemma perm-σ1' : σ1' = (state.make ((Map.empty (oid3 → (in_Person (X_Person3_Person)))) (oid2 → (in_Person (X_Person2_Person)))) (oid1 → (in_Person (X_Person1_Person)))) ((assocs (σ1'))))
(proof)
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 X_Person1_Person :: ty_Person
fixes X_Person1 :: :: Person
assumes X_Person1-def: X_Person1 = (λ-. [[X_Person1_Person]])
fixes X_Person2_Person :: ty_Person
fixes X_Person2 :: :: Person
assumes X_Person2-def: X_Person2 = (λ-. [[X_Person2_Person]])
fixes X_Person3_Person :: ty_Person
fixes X_Person3 :: :: Person
assumes X_Person3-def: X_Person3 = (λ-. [[X_Person3_Person]])
fixes σ1-object0_Person :: ty_Person
fixes σ1-object0 :: :: Person
assumes σ1-object0-def: σ1-object0 = (λ-. [[σ1-object0_Person]])
fixes σ1-object1_Person :: ty_Person
fixes σ1-object1 :: :: Person
assumes σ1-object1-def: σ1-object1 = (λ-. [[σ1-object1_Person]])
assumes $\sigma_1$-object1-def: $\sigma_1$-object1 = ($\lambda$. $[[\sigma_1$-object1 Person]])

fixes $\sigma_1$-object2 Person :: ty Person

fixes $\sigma_1$-object2 :: · Person

assumes $\sigma_1$-object2-def: $\sigma_1$-object2 = ($\lambda$. $[[\sigma_1$-object2 Person]])

assumes $\sigma_1$: (state-$\sigma_1$ (oid4) (oid5) (oid6) (oid1) (oid7) (oid2) (\sigma_1-object0 Person) (\sigma_1-object0 Person) (\sigma_1-object1 Person) (\sigma_1-object1 Person) (\sigma_1-object2 Person) (\sigma_1-object2 Person) (\sigma_1-object2 Person) (\sigma_1-object2 Person) (X Person) (\sigma_1-object4 Person) (\sigma_1-object4 Person) (X Person) (X Person))

begin

interpretation state-$\sigma_1$: state-$\sigma_1$ oid4 oid5 oid6 oid1 oid7 oid2 $\sigma_1$-object0 Person $\sigma_1$-object0 Person $\sigma_1$-object1 Person $\sigma_1$-object2 Person $\sigma_1$-object2 Person $\sigma_1$-object2 Person $\sigma_1$-object4 Person $\sigma_1$-object4 Person $\sigma_1$-object4 Person $\sigma_1$-object4 Person (\sigma_1-object4 Person) (X Person) (X Person)

⟨proof⟩

interpretation state-$\sigma_1$: state-$\sigma_1$ oid1 oid2 oid3 Person1 Person Person2 Person Person3 Person (\sigma_1-object4 Person) (\sigma_1-object4 Person) (\sigma_1-object4 Person) (\sigma_1-object4 Person) (\sigma_1-object4 Person) (\sigma_1-object4 Person)

⟨proof⟩

end

end