A Meta-Model for the Isabelle API

Frédéric Tuong  Burkhart Wolff

August 16, 2018

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

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

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

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

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

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

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

1. **A Meta-Model for the Isabelle API** 7

1. **Initialization** 9
   1.1. Optimization on the String Datatype 9
   1.2. Basic Extension of the Standard Library 9
      1.2.1. Polymorphic Cartouches 9
      1.2.2. Operations on List 10
      1.2.3. Operations on Char 11
      1.2.4. Operations on String (I) 11
      1.2.5. Operations on String (II) 13

2. **Defining Meta-Models** 17
   2.1. (Pure) Term Meta-Model aka. AST definition of (Pure) Term 17
      2.1.1. Type Definition 17
      2.1.2. Operations of Fold, Map, ..., on the Meta-Model 17
   2.2. SML Meta-Model aka. AST definition of SML 18
      2.2.1. Type Definition 18
      2.2.2. Extending the Meta-Model 18
   2.3. Isabelle Meta-Model aka. AST definition of Isabelle 19
      2.3.1. Type Definition 20
      2.3.2. Extending the Meta-Model 23
      2.3.3. Operations of Fold, Map, ..., on the Meta-Model 28

3. **Parsing Meta-Models** 29
   3.1. Initializing the Parser 29
      3.1.1. Some Generic Combinators 29
      3.1.2. Generic Locale for Parsing 31
   3.2. Instantiating the Parser of (Pure) Term 31
      3.2.1. Main 31

4. **Printing Meta-Models** 33
   4.1. Initializing the Printer 33
      4.1.1. Kernel Code for Target Languages 33
      4.1.2. Interlace with Types 37
      4.1.3. Interlace with Constants 38
      4.1.4. Some Notations (I): Raw Translations 39
      4.1.5. Some Notations (II): Polymorphic Cartouches 40
Part I.

A Meta-Model for the Isabelle API
1. Initialization

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

1.1. Optimization on the String Datatype

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

type-notation natural (nat)
definition Succ x = x + 1
datatype stringbase = ST String.literal
  | ST’ integer list

datatype abr-string =
  SS-base stringbase
  | String-concatWith abr-string abr-string list

syntax -string1 :: - ⇒ abr-string (((-)))
translations (x) = CONST SS-base (CONST ST x)
syntax -string3 :: - ⇒ abr-string (≪(-)≫)
translations ≪x≫ = CONST SS-base (CONST ST’ x)
syntax -integer1 :: - ⇒ abr-string ("(-)"
translations "x" = CONST SS-base (CONST ST’ ((CONST Cons) x (CONST Nil)))

type-notation abr-string (string)

1.2. Basic Extension of the Standard Library

1.2.1. Polymorphic Cartouches

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

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

### 1.2.2. Operations on List

**datatype** ('a, 'b) \texttt{nsplit} = \texttt{Nsplit-text 'a} \\
| \texttt{Nsplit-sep 'b}

**locale** \texttt{L} begin

\texttt{definition map \ where \ map \ f \ l = \ rev \ (foldd \ (\lambda x. f x \# l) \ [] \ l)}

\texttt{definition flatten \ l = foldl \ (\lambda acc. l. foldl \ (\lambda acc. x. x \# acc) \ acc \ (rev \ l)) \ [] \ (rev \ l)}

\texttt{definition mapi \ f \ l = \ rev \ (fst \ (foldl \ (\lambda l. cpt) \ x. \ (f \ x \# l, Succ \ cpt)) \ ([], 0::nat) \ l))}

\texttt{definition iter \ f = \ foldl \ (\lambda -. f) \ ()}

\texttt{definition maps \ f \ x = L.flatten \ (L.map \ f \ x)}

\texttt{definition append \ where \ append \ a \ b = L.flatten \ [a, b]}

\texttt{definition filter \ where \ filter \ f \ l = \ rev \ (foldl \ (\lambda x. if \ f \ x \ then \ x \# l \ else \ l) \ [] \ l)}

\texttt{definition rev-map \ f = \ foldl \ (\lambda l. f x \# l) \ []}

\texttt{definition mapM \ f \ l \ accu =}

\hspace{1cm} (let \ (l, accu) = \texttt{List.fold \ (\lambda x. (l, accu). let \ (x, accu) = \texttt{f x \ accu \ in \ (x \# l, accu)) \ l \ [] \ (accu) \ in \ (rev \ l, accu)})

\texttt{definition assoc \ x1 \ l = \texttt{List.fold} \ (\lambda (x2, v). \texttt{\lambda None} \Rightarrow \texttt{if} \ x1 = x2 \texttt{then} \texttt{Some v} \texttt{else} \texttt{None} \ | \ x \Rightarrow x) \ l \ \texttt{None}

\texttt{definition split \ where \ split \ l = (L.map \ fst \ l, L.map \ snd \ l)}

\texttt{definition upto \ where \ upto \ i \ j =}

\hspace{1cm} (let \ to-i = \texttt{\lambda n. int-of-integer \ (integer-of-natural \ n) \ in}

\hspace{2cm} L.map \ (\texttt{natural-of-integer \ o \ integer-of-int}) \ (\texttt{List.upto} \ (\texttt{to-i}) \ (\texttt{to-i} \ j)))

\texttt{definition split-at \ f \ l =}

\hspace{1cm} (let \ f = \texttt{\lambda x. \neg \ f \ x \ in}

\hspace{2cm} (\texttt{takeWhile} \ f \ l, \texttt{case dropWhile} \ f \ l \ of \ [] \Rightarrow \texttt{(None, [] | x \# xs} \Rightarrow \texttt{Some x, xs}))

\texttt{definition take \ where \ take \ reverse \ lg \ l = \texttt{reverse} \ (snd \ (L.split \ \texttt{takeWhile} \ (\lambda (n, -). \texttt{n < lg}) \ (\texttt{enumerate} \ 0 \ (\texttt{reverse} \ l))))}

\texttt{definition take-last = \texttt{take rev}}

\texttt{definition take-first = \texttt{take id}}

\texttt{definition replace-gen \ f-res \ l \ c0 \ bby =}

\hspace{1cm} (let \ Nsplit-text = \texttt{\lambda l \ igen. \ if} \ l = [] \texttt{then igen \ else \ Nsplit-text} \ l \ # \ igen \ \texttt{in case List.fold}

\hspace{2cm} (\texttt{\lambda c1 \ (l, igen),}

\hspace{3cm} \texttt{if} \ c0 \ c1 \texttt{then}

\hspace{4cm} (bby, \texttt{Nsplit-sep \ c1 \# Nsplit-text} \ l \ igen)

\hspace{3cm} \texttt{else}

\hspace{4cm} (c1 \# l, igen))

\hspace{2cm} (\texttt{rev} \ l)

\hspace{1.5cm} ([], []))

\texttt{of \ (l, igen) \Rightarrow \texttt{f-res} \ (Nsplit-text \ l \ igen))}

\texttt{definition nsplit-f \ l \ c0 = \texttt{replace-gen \ id} \ l \ c0 \ []}

\texttt{definition replace = \texttt{replace-gen} \ (L.flatten \ o \ L.map \ (\lambda Nsplit-text \ l \Rightarrow \texttt{l | - \Rightarrow []})}
fun map-find-aux where
    map-find-aux accu f l = (λ [] ⇒ List.rev accu
                           | x # xs ⇒ (case f x of Some x ⇒ List.fold Cons accu (x # xs)
                           | None ⇒ map-find-aux (x # accu) f xs)) l

definition map-find = map-find-aux []

end
notation L.append (infixr @@@@ 65)

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

    — fun
    L.map-find-aux.simps

1.2.3. Operations on Char

definition ascii-of-literal (INT) where
    ascii-of-literal = hd o String.ascii-of-literal

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

1.2.4. Operations on String (I)

notation String.ascii-of-literal (INTS)

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 (infixr ≫≫ 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 (λ s) ⊥ (Some s) ⊥)
  | ST’ s ⇒ S.flatten (L.map g s))

fun (in String) map-gen where
  map-gen replace g e =
  (λ SS-base s ⇒ Stringbase.map-gen replace g s
   | String-concatWith abr l ⇒ String-concatWith (map-gen replace g abr) (List.map (map-gen replace g) l)) e

definition (in String) foldl-one f accu = foldl f accu o INTS

definition (in 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 =
  (λ SS-base s ⇒ Stringbase.foldl f accu s
   | 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-integers f s1 s s2 =
  s1 @@@ (case s of None ⇒ ⊥ | Some s ⇒ flatten (L.map f (INTS s))) @@@ s2

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

definition (in String) replace-integers f = map-gen (S.replace-integers (λc. f c)) f

definition (in String) all 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 ⇒ INTS s | ST’ l ⇒ l)

definition (in String) meta-of-logic = String.literal-of-asciis o to-list

definition (in String) to-Stringbase = (λ SS-base s ⇒ s | s ⇒ ST’ (to-list s))

definition (in Stringbase) to-String = SS-base

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 = List.member (L.map Stringbase.to-list l) (to-list x)
definition (in Stringbase) flatten l = String.to-Stringbase (S.flatten (L.map to-String l))

lemmas [code] =
  — def
  S.flatten-def
  String.flatten-def

12
1.2.5. Operations on String (II)

definition wildcard = ⟨⟩

context String

begin

definition lowercase = \(\lambda n. \text{if } n < 97 \text{ then } n + 32 \text{ else } n\)
definition uppercase = \(\lambda n. \text{if } n < 97 \text{ then } n \text{ else } n - 32\)
definition to-bold-number = replace-integers (\(\lambda n. \langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle!\))

nat-of-integer (\(n - 48\))
definition nat-to-digit10 aux where
  nat-to-digit10 aux \(l (n :: \text{Nat.nat}) = (\text{if } n < 10 \text{ then } n \# l \text{ else } \text{nat-to-digit10 aux} (n \text{ mod 10} \# l) (n \text{ div 10})\))
definition nat-to-digit10 = (let nat-raw-to-str = L.map (integer-of-nat o (+) 0x30) in
  \(\langle\text{nat-raw-to-str} (\text{nat-to-digit10 aux} [] n)\rangle\))
definition natural-to-digit10 = nat-to-digit10 o nat-of-natural

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

definition integer-to-digit16 =
  (let \(f = \text{nth} (\text{INTS} \langle 0123456789ABCDEF \rangle) \text{ o } \text{nat-of-integer}\) \(\text{in}\\n\  \lambda n \Rightarrow (\langle f (n \text{ div 16}), f (n \text{ mod 16}) \rangle)\))
end
lemmas [code] =

— def
String.lowercase-def
String.uppercase-def
String.to-bold-number-def
String.nat-to-digit10-def
String.natural-to-digit10-def
String.integer-to-digit16-def

— fun
String.nat-to-digit10-aux.simps

definition add-0 n =
(let n = nat-of-integer n in
S.flatten \(L.map (\lambda \cdot \langle 0 \rangle) \text{ (upt } 0 \text{ if } n < 10 \text{ then } 2 \text{ else if } n < 100 \text{ then } 1 \text{ else } 0))\)@@ String.nat-to-digit10 n)

declare[\{\text{cartouche-type }= \text{String.literal}\}]

definition is-letter =
(let int-A = INT \langle A \rangle; int-Z = INT \langle Z \rangle; int-a = INT \langle a \rangle; int-z = INT \langle z \rangle \text{ in}
(\lambda n. n \geq \text{int-A} \& n \leq \text{int-Z} \mid n \geq \text{int-a} \& n \leq \text{int-z}))
definition is-digit =
(let int-0 = INT \langle 0 \rangle; int-9 = INT \langle 9 \rangle \text{ in}
(\lambda n. n \geq \text{int-0} \& n \leq \text{int-9}))
definition is-special = List.member (INTS \langle < \rangle \sim - = . / \{ \})

begin

definition base255 = replace-integers (\lambda c. \text{if is-letter c then } c \text{ else add-0 c})
declare[\{\text{cartouche-type }= \text{abr-string}\}]aison 3]
definition isub =
replace-integers (let is-und = List.member (INTS (STR \"\-")') \text{ in}
(\lambda c. \text{if is-letter c } \mid \text{ is-digit c } \mid \text{ is-und c then } \langle \rangle \text{ else add-0 c}))
definition isup s = (\sim) @@ s
end

lemmas [code] =

— def
String.base255-def
String.isub-def
String.isup-def

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

definition text-of-str str =
(let s = \langle c \rangle
; ap = \langle \# \rangle \text{ in}
S.flatten [ \langle (let \rangle, s, \langle = \text{char-of :: nat } \Rightarrow \text{char in } \rangle
, String.replace-integers (\lambda c.
if is-letter c then

14
\begin{align*}
S.\text{flatten} & \left[ \langle \text{CHR} \rangle', \langle \text{CHR} \rangle', \langle \text{CHR} \rangle', \text{ap} \right] \\
\text{else} & \\
S.\text{flatten} & \left[ s, (\cdot), \text{add-0} \ c, \text{ap} \right]
\end{align*}

\text{str}

, \left[ [] \right]

\text{definition} \ \text{text2-of-str} = \text{String.replace-integers} \left( \lambda c. \ S.\text{flatten} \left[ \langle \cdot \rangle', \langle \cdot \rangle', \langle \cdot \rangle', \langle \cdot \rangle \right] \right)

\text{definition} \ \text{strstr-of-str} \ f.\text{flatten} \ f.\text{integer} \ f.\text{str} \ str = 

\left( \text{let str0} = \text{String.to-list} \ str \right)

; f.\text{letter} = \lambda. \ \text{is-letter} \ c \ | \ \text{is-digit} \ c \ | \ \text{is-special} \ c

; s = \langle \cdot \rangle

; f.\text{text} = \lambda \ N\text{split-text} \ l \Rightarrow S.\text{flatten} \left[ f.\text{str} \left( S.\text{flatten} \left[ \langle \text{STR} \rangle', \langle \cdot \rangle', \langle \cdot \rangle' \right] \right) \right]

; str = \text{case} \ L.\text{nsplit-f} \ str0 \ \text{(Not o f.letter)} \ of

[] \Rightarrow S.\text{flatten} \left[ f.\text{str} \langle \text{STR} \rangle' \right]

| \left[ x \right] \Rightarrow f.\text{text} \ x

| l \Rightarrow S.\text{flatten} \left( L.\text{map} \ \left( \lambda x. \ (\cdot) @ @ f.\text{text} \ x @ @ \langle \cdot \rangle \right) \ l \right) @ @ [\cdot] \ \text{in}

\text{if list-all f.letter} \ str0 \ \text{then}

str

\text{else}

f.\text{flatten} \left( S.\text{flatten} \ [ \langle \cdot, \ str, \ \cdot \rangle, [] \right] \right)

\text{definition} \ \langle \text{escape-sm} \rangle = \text{String.replace-integers} \left( \lambda n. \ \text{if} \ n = 0x22 \ \text{then} \ \langle \cdot \rangle \ \text{else} \ \langle \cdot \rangle \right)

\text{definition} \ \text{mk-constr-name} \ name = \left( \lambda x. \ S.\text{flatten} \ [ \text{String.isub} \ name, \langle \cdot \rangle, \text{String.isub} \ x \right)

\text{definition} \ \text{mk-dot} \ s1 \ s2 = S.\text{flatten} \ [\cdot, s1, s2]

\text{definition} \ \text{mk-dot-par-gen} \ dot \ l-s = S.\text{flatten} \ [dot, (\cdot), \text{case} \ l-s \ \text{of} \ [] \Rightarrow (\cdot) | x # xs \Rightarrow S.\text{flatten} \ [x, S.\text{flatten} \ [L.\text{map} \ \left( \lambda s. \ (\cdot) @ @ s \right) \ xs], \langle \cdot \rangle] \right)

\text{definition} \ \text{mk-dot-comment} \ s1 \ s2 \ s3 = \text{mk-dot-par-gen} \ dot \ [s]

\text{definition} \ \text{mk-dot-case} \ s1 \ s2 = \text{mk-dot-par-gen} \ \left( s1, (\cdot), s2, \langle \cdot \rangle \right)

\text{definition} \ \text{hol-definition} \ s = S.\text{flatten} \ [s, \langle \cdot, \text{def} \rangle]

\text{definition} \ \text{hol-split} \ s = S.\text{flatten} \ [s, \langle \cdot, \text{split} \rangle]

\text{end}
2. Defining Meta-Models

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

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

2.1.1. Type Definition

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

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

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

fun fold-Const where
  fold-Const f accu expr = (λ Const s - ⇒ f accu s |
    Abs - - term ⇒ fold-Const f accu term |
    App term1 term2 ⇒ fold-Const f (fold-Const f accu term1) term2 |
    - ⇒ accu)
\begin{align*}
\text{fun } \textit{fold-Free} \text{ where} \\
\textit{fold-Free } f \text{ accu } \textit{expr} = \lambda \text{ Free } s \rightarrow f \text{ accu } s \\
\mid \text{ Abs } - \text{ term } \Rightarrow \textit{fold-Free } f \text{ accu } \textit{term} \\
\mid \text{ App } \textit{term1 } \textit{term2} \Rightarrow \textit{fold-Free } f \ (\textit{fold-Free } f \text{ accu } \textit{term1}) \textit{term2} \\
\mid - \Rightarrow \text{ accu} \\
\textit{expr}
\end{align*}

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

theory \textit{Meta-SML} imports ../\textit{Init} begin
\textbf{2.2.1. Type Definition}

The following datatypes beginning with \texttt{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.

\textbf{datatype \texttt{semi--val-fun} = \texttt{Sval} | \texttt{Sfun}}

\textbf{datatype \texttt{semi--term}' = \texttt{SML-string string} | \texttt{SML-rewrite semi--val-fun semi--term}' — left string — symb rewriting} \\
\texttt{semi--term}' — right \\
\mid \texttt{SML-basic string list} \\
\mid \texttt{SML-binop semi--term' string semi--term'} \\
\mid \texttt{SML-annot semi--term' string — type} \\
\mid \texttt{SML-function (semi--term' — pattern \times semi--term' — to return) list} \\
\mid \texttt{SML-apply semi--term' semi--term' list} \\
\mid \texttt{SML-paren string — left string — right semi--term'} \\
\mid \texttt{SML-let-open string semi--term'}

\textbf{2.2.2. Extending the Meta-Model}

locale \texttt{SML} begin
\textbf{no-type-notation \texttt{abr-string (string) definition string = SML-string}}
\textbf{definition rewrite = SML-rewrite} \\
\textbf{definition basic = SML-basic} \\
\textbf{definition binop = SML-binop} \\
\textbf{definition annot = SML-annot} \\
\textbf{definition function = SML-function} \\
\textbf{definition apply = SML-apply}
definition paren = SML-paren
definition let-open = SML-let-open

definition app s = apply (basic [s])
definition none = basic [NONE]
definition some s = app ⟨SOME⟩ [s]
definition option' f l = (case map-option f l of None ⇒ none | Some s ⇒ some s)
definition option = option' id

definition parenthesis — mandatory parenthesis = paren ⟨(⟨⟩)⟩
definition binop-l s l = (case rev l of x ≠ xs ⇒ List.fold (λx. binop x s) xs x)
definition list l = (case l of [] ⇒ basic [⟨] ] | - ⇒ paren ⟨[ ]⟩ (binop-l ⟨⟩ l))
definition list' f l = list (L.map f l)
definition pair e1 e2 = parenthesis (binop e1 ⟨⟩ e2)
definition pair' f1 f2 = (λ(e1, e2) ⇒ parenthesis (binop (f1 e1) ⟨⟩ (f2 e2)))
definition rewrite-val = rewrite Sval
definition rewrite-fun = rewrite Sfun
end

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

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

theory Meta-Isabelle
imports Meta-Pure
2.3.1. Type Definition

The following datatypes beginning with \texttt{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.

\begin{verbatim}
datatype semi--typ = Typ-apply semi--typ semi--typ list
                   | Typ-apply-bin string — binop semi--typ semi--typ
                   | Typ-apply-paren string — left string — right semi--typ
                   | Typ-base string

datatype datatype = Datatype string — name
                   (string — name \times semi--typ list — arguments) list — constructors

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

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

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

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

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

datatype consts = Consts string — name
                 semi--typ
                 string — expression in 'post' mixfix
\end{verbatim}

\textit{Meta-SML}

\begin{verbatim}
begin
end
\end{verbatim}
\textbf{datatype} \textit{definition} = \textit{Definition} \textit{semi-term} \\
\hspace{1em} | \textit{Definition}-where1 \textit{string} — \textit{name} \textit{semi-term} — syntax extension \times \textit{nat} — priority \textit{semi-term} \\
\hspace{1em} | \textit{Definition}-where2 \textit{string} — \textit{name} \textit{semi-term} — syntax extension \textit{semi-term}

\textbf{datatype} \textit{semi-thm-attribute} = \textit{Thm-thm} \textit{string} — represents a single \text{thm} \\
\hspace{1em} | \textit{Thm-thms} \textit{string} — represents several \text{thms} \\
\hspace{1em} | \textit{Thm-THEN} \textit{semi-thm-attribute} \textit{semi-thm-attribute} \\
\hspace{1em} | \textit{Thm-simplified} \textit{semi-thm-attribute} \textit{semi-thm-attribute} \\
\hspace{1em} | \textit{Thm-symmetric} \textit{semi-thm-attribute} \\
\hspace{1em} | \textit{Thm-where} \textit{semi-thm-attribute} \textit{(string} \times \textit{semi-term}) \text{list} \\
\hspace{1em} | \textit{Thm-of} \textit{semi-thm-attribute} \textit{semi-term} \text{list} \\
\hspace{1em} | \textit{Thm-OF} \textit{semi-thm-attribute} \textit{semi-thm-attribute}

\textbf{datatype} \textit{semi-thm} = \textit{Thms-single} \textit{semi-thm-attribute} \\
\hspace{1em} | \textit{Thms-mult} \textit{semi-thm-attribute}

\textbf{type-synonym} \textit{semi-thm-l} = \textit{semi-thm} \text{list}

\textbf{datatype} \textit{lemmas} = \textit{Lemmas-simp-thm} \textit{bool} — \textit{True} : \textit{smp} \\
\hspace{1em} | \textit{Lemmas-simp-thms} \textit{string} — \textit{name} \textit{semi-thm-attribute} \text{list} \\
\hspace{1em} | \textit{Lemmas-simp-thms} \textit{string} — \textit{name} \textit{semi-thm-attribute} \text{list}

\textbf{datatype} \textit{semi-method-simp} = \textit{Method-simp-only} \textit{semi-thm-l} \\
\hspace{1em} | \textit{Method-simp-add-del-split} \textit{semi-thm-l} — add \textit{semi-thm-l} — del \textit{semi-thm-l} — split

\textbf{datatype} \textit{semi-method} = \textit{Method-rule} \textit{semi-thm-attribute} \textit{option} \\
\hspace{1em} | \textit{Method-drule} \textit{semi-thm-attribute} \\
\hspace{1em} | \textit{Method-erule} \textit{semi-thm-attribute} \\
\hspace{1em} | \textit{Method-intro} \textit{semi-thm-attribute} \text{list} \\
\hspace{1em} | \textit{Method-elim} \textit{semi-thm-attribute} \\
\hspace{1em} | \textit{Method-subst} \textit{bool} — \textit{asm} \text{list} \\
\hspace{1em} | \textit{Method-insert} \textit{semi-thm-l} \\
\hspace{1em} | \textit{Method-plus} \textit{semi-method} \text{list} \\
\hspace{1em} | \textit{Method-option} \textit{semi-method} \text{list} \\
\hspace{1em} | \textit{Method-or} \textit{semi-method} \text{list} \\
\hspace{1em} | \textit{Method-one} \textit{semi-method-simp} \\
\hspace{1em} | \textit{Method-all} \textit{semi-method-simp} \\
\hspace{1em} | \textit{Method-auto-simp-add-split} \textit{semi-thm-l} \textit{string} \text{list} \\
\hspace{1em} | \textit{Method-rename-tac} \textit{string} \text{list} \\
\hspace{1em} | \textit{Method-case-tac} \textit{semi-term} \\
\hspace{1em} | \textit{Method-blast} \textit{nat} \text{option} \\
\hspace{1em} | \textit{Method-clarify}
| Method-metis string list — e.g. no-types (override-type-encs)  
  | semi-thm-attribute list

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

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

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

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

datatype axiomatization = Axiomatization string — name  
  | semi-term

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

datatype text = Text string

datatype ML = SML semi-term'

datatype setup = Setup semi-term'

datatype thm = Thm semi-thm-attribute list
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-\ell s l = List.fold (\x acc. Thm-OF acc x) l s
definition simplified-\ell s l = List.fold (\x acc. Thm-simplified acc x) l s
end

lemmas [code] =
— def
T.thm-def
T.thms-def
T.THEN-def
T.simplified-def
T.symmetric-def
T.where-def
T.of′-def
T.OF-def
T.OF-l-def
T.simplified-l-def

definition Opt s = Typ-apply (Typ-base ⟨option⟩) [Typ-base s]
definition Raw = Typ-base
definition Type-synonym′ n = Type-synonym n []
definition Type-synonym′′ n f l = Type-synonym n l (f l)
definition Term-annot′ e s = Term-annot e (Typ-base s)
definition Term-lambdas x = Term-lambdas [x]
definition Term-lambdas0 = Term-bind ⟨λ⟩ (Term-basic s)
definition Term-some = Term-paren ⟨⌊⟩ ⟨⌋⟩
definition Term-parenthesis — mandatory parenthesis = Term-paren ⟨(⟩ ⟨)⟩
definition Term-warning-parenthesis — optional parenthesis that can be removed but a warning will be raised = Term-parenthesis
definition Term-pat b = Term-basic [⟨?⟩ b]
definition Term-And x f = Term-bind ⟨⋀⟩ (Term-basic [x]) (f x)
definition Term-exists x f = Term-bind ⟨∃⟩ (Term-basic [x]) (f x)
definition Term-binop = Term-rewrite
definition term-binop s l = (case rev l of x # xs ⇒ List.fold (λ x. Term-binop x s) xs x)
definition term-binop′ s l = (case rev l of x # xs ⇒ List.fold (Term-parenthesis o Term-binop ⟨(⟩ ⟨)⟩) xs x)
definition Term-set l = (case l of [] ⇒ Term-basic [⟨{}⟩] | - ⇒ Term-paren ⟨{⟩ ⟨}⟩ (term-binop ⟨(⟩ ⟨)⟩ l))
definition Term-list l = (case l of [] ⇒ Term-basic [⟨[]⟩] | - ⇒ Term-paren ⟨{⟩ ⟨}⟩ (term-binop ⟨(⟩ ⟨)⟩ l))
definition Term-list′ f l = Term-list (L.map f l)
definition Term-pair e1 e2 = Term-parenthesis (Term-binop e1 ⟨(⟩ ⟨)⟩ e2)
definition Term-pair′ l = (case l of [] ⇒ Term-basic [⟨[]⟩] | - ⇒ Term-paren ⟨{⟩ ⟨}⟩ (term-binop ⟨(⟩ ⟨)⟩ l))
definition Term-string s = Term-basic [S.flatten ⟨v, s, ⟨⟩⟩]
definition Term-apps0 e l = Term-parenthesis (Term-app e (L.map Term-parenthesis l))
definition Term-apps e l = Term-apps0 (Term-parenthesis e) l
definition Term-app e = Term-apps0 (Term-basic [e])
definition Term-preunary e1 e2 = Term-app e1 [e2] — no parenthesis and separated with one space
definition Term-postunary e1 e2 = Term-app e1 [e2] — no parenthesis and separated with one space
definition Term-case = Term-fun-case o Some
definition Term-function = Term-fun-case None
definition Term-term’ = Term-term []
definition Lemmas-simp = Lemmas-simp-thm True
definition Lemmas-nosimp = Lemmas-simp-thm False
definition Consts-value = ⟨-⟩
definition Consts-raw0 s l e o-arg = Consts s l (String.replace-integers (λn. if n = 0x5F then ‘-’ else “n”) e @@ (case o-arg of None ⇒ () | Some arg ⇒ let ap = λs. ‘(’ @@ s @@ ‘)’ in ap (if arg = 0 then () else Consts-value @@ (S.flatten (L.map (λ-. ⟨-⟩ @@ Consts-value) (L.upto 2 arg))))))
definition Ty-arrow = Typ-apply-bin (⇒)
definition Ty-times = Typ-apply-bin (×)
definition Ty-arrow’ x = Ty-arrow x (Typ-base (⟨⟩))
definition Ty-paren = Typ-apply-paren (⟨⟩ ⟨⟩)
definition Consts’ s l e = Consts-raw0 s (Ty-arrow (Typ-base (⟨⟩)) l) e None
definition Overloading’ n ty = Overloading n (Term-annot (Term-basic [n]) ty)
locale M
begin
definition Method-simp-add-del l-a l-d = Method-simp-add-del-split l-a l-d []
definition Method-subst-l = Method-subst False
definition rule’ = Method-rule None
definition rule = Method-rule o Some
definition drule = Method-drule
definition erule = Method-erule
definition intro = Method-intro
definition elim = Method-elim
definition subst-l0 = Method-subst
definition subst-l = Method-subst-l
definition insert where insert = Method-insert o L.map Thms-single
definition plus where plus = Method-plus
definition option = Method-option
definition or = Method-or
definition meth-gen-simp = Method-simp-add-del [] []
definition meth-gen-simp-add2 ll l2 = Method-simp-add-del (L.flatten [ L.map Thms-mult ll , L.map (Thms-single o Thm-thm) l2])

[]
definition meth-gen-simp-add-del ll l2 = Method-simp-add-del (L.map (Thms-single o Thm-thm) ll) (L.map (Thms-single o Thm-thm) l2)
definition meth-gen-simp-add-del-split ll l2 l3 = Method-simp-add-del-split (L.map Thms-single ll) (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-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-split l1 l2 l3 = Method-one (meth-gen-simp-add-del-split l1 l2 l3) 
definition simp-add-split l1 l2 = Method-one (meth-gen-simp-add-split l1 l2) 
definition simp-only l = Method-one (meth-gen-simp-only l) 
definition simp-only' l = Method-one (meth-gen-simp-only' l) 
definition simp-add0 l = Method-one (meth-gen-simp-add0 l) 
definition simp-add = simp-add2 [] 
definition simp-all = Method-all meth-gen-simp 
definition simp-all-add l = Method-all (meth-gen-simp-add2 [] l) 
definition simp-all-only l = Method-all (meth-gen-simp-only l) 
definition simp-all-only' l = Method-all (meth-gen-simp-only' l) 
definition auto-simp-add2 l1 l2 = Method-auto-simp-add-split (L.flatten [L.map Thms-mult l1], L.map (Thms-single o Thm-thm) l2) [] 
definition auto-simp-add-split l = Method-auto-simp-add-split (L.map Thms-single l) 
definition rename-tac = Method-rename-tac 
definition case-tac = Method-case-tac 
definition blast = Method-blast 
definition clarify = Method-clarify 
definition metis = Method-metis [] 
definition metis0 = Method-metis 
definition subst-asn b = subst-l0 b [[]] 
definition subst = subst-l [[]] 
definition auto-simp-add = auto-simp-add2 [] 
definition auto = auto-simp-add [] 
end

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

26
definition ty-arrow l = (case rev l of x # xs ⇒ List.fold Ty-arrow xs x)

locale C
begin
definition done = Command-done
definition by = Command-by
definition sorry = Command-sorry
definition apply-end = Command-apply-end
definition apply = Command-apply
definition using = Command-using o L.map Thms-single
definition unfolding = Command-unfolding o L.map Thms-single
definition let' = Command-let
definition fix-let = Command-fix-let
definition fix l = Command-fix-let l [] None []
definition have n = Command-have n False
definition have0 = Command-have
end

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

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

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

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

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

end
3. Parsing Meta-Models

3.1. Initializing the Parser

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

3.1.1. Some Generic Combinators

definition K x = x

definition co1 = (λ)
definition co2 f g x1 x2 = f (g x1 x2)
definition co3 f g x1 x2 x3 = f (g x1 x2 x3)
definition co4 f g x1 x2 x3 x4 = f (g x1 x2 x3 x4)
definition co5 f g x1 x2 x3 x4 x5 = f (g x1 x2 x3 x4 x5)
definition co6 f g x1 x2 x3 x4 x5 x6 = f (g x1 x2 x3 x4 x5 x6)
definition co7 f g x1 x2 x3 x4 x5 x6 x7 = f (g x1 x2 x3 x4 x5 x6 x7)
definition co8 f g x1 x2 x3 x4 x5 x6 x7 x8 = f (g x1 x2 x3 x4 x5 x6 x7 x8)
definition co9 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9)
definition co10 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10)
definition co11 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11)
definition co12 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12)
definition co13 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13)
definition co14 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14)
definition co15 f g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 = f (g x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15)
definition ap1 a v0 f1 v1 = a v0 [f1 v1]
definition ap2 a v0 f1 f2 v1 v2 = a v0 [f1 v1, f2 v2]
definition ap3 a v0 f1 f2 f3 v1 v2 v3 = a v0 [f1 v1, f2 v2, f3 v3]
definition ap4 a v0 f1 f2 f3 f4 v1 v2 v3 v4 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4]
definition ap5 a v0 f1 f2 f3 f4 f5 v1 v2 v3 v4 v5 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5]
definition ap6 a v0 f1 f2 f3 f4 f5 f6 v1 v2 v3 v4 v5 v6 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6]
definition ap7 a v0 f1 f2 f3 f4 f5 f6 f7 v1 v2 v3 v4 v5 v6 v7 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7]
definition ap8 a v0 f1 f2 f3 f4 f5 f6 f7 f8 v1 v2 v3 v4 v5 v6 v7 v8 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8]
\[ f_4 v_4, f_5 v_5, f_6 v_6, f_7 v_7, f_8 v_8 \]

definition \( ap9 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 v1 v2 v3 v4 v5 v6 v7 v8 v9 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9]

definition \( ap10 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10]

definition \( ap11 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11]

definition \( ap12 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, f12 v12]

definition \( ap13 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, f12 v12, f13 v13]

definition \( ap14 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 = a v0 [f0 v0, f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, f12 v12, f13 v13, f14 v14]

definition \( ap15 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14 f15 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 = a v0 [f0 v0, f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, f12 v12, f13 v13, f14 v14, f15 v15]

definition \( ar1 \) a v0 z = a v0 [z]

definition \( ar2 \) a v0 f0 v0 v1 z = a v0 [f0 v0, z]

definition \( ar3 \) a v0 f0 f1 f2 v1 v2 z = a v0 [f1 v1, f2 v2, z]

definition \( ar4 \) a v0 f0 f1 f2 f3 v1 v2 v3 z = a v0 [f1 v1, f2 v2, f3 v3, z]

definition \( ar5 \) a v0 f0 f1 f2 f3 f4 v1 v2 v3 v4 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, z]

definition \( ar6 \) a v0 f0 f1 f2 f3 f4 f5 v1 v2 v3 v4 v5 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, z]

definition \( ar7 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 v1 v2 v3 v4 v5 v6 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, z]

definition \( ar8 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 v1 v2 v3 v4 v5 v6 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, z]

definition \( ar9 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 v1 v2 v3 v4 v5 v6 v7 v8 v9 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, z]

definition \( ar10 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, z]

definition \( ar11 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, z]

definition \( ar12 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, f12 v12, z]

definition \( ar13 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, f12 v12, f13 v13, z]

definition \( ar14 \) a v0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 z = a v0 [f1 v1, f2 v2, f3 v3, f4 v4, f5 v5, f6 v6, f7 v7, f8 v8, f9 v9, f10 v10, f11 v11, f12 v12, f13 v13, f14 v14, z]
3.1.2. Generic Locale for Parsing

locale Parse =
  fixes ext :: string ⇒ string

— (effective) first order
  fixes of-string :: ('a ⇒ 'a list ⇒ 'a) ⇒ (string ⇒ 'a) ⇒ string ⇒ 'a
  fixes of-string_base :: ('a ⇒ 'a list ⇒ 'a) ⇒ (string ⇒ 'a) ⇒ string_base ⇒ 'a
  fixes of-nat :: ('a ⇒ 'a list ⇒ 'a) ⇒ (string ⇒ 'a) ⇒ natural ⇒ 'a
  fixes of-unit :: (string ⇒ 'a) ⇒ unit ⇒ 'a
  fixes of-bool :: (string ⇒ 'a) ⇒ bool ⇒ 'a

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

begin

definition of-pair a b f1 f2 = (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
definition of-pure-indexname a b = of-pair a b (of-string a b) (of-nat a b)
definition of-pure-class = of-string
definition of-pure-sort a b = of-list a b (of-pure-class a b)

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

definition of-pure-term a b = (λ f0 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)

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
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 CodeConstPrintf
    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 . CodeConstPrintf.sprintf1 pat)
        hClose h)

  |
  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 . CodeConstPrintf.sprintf1 pat)
      hClose h)

  |
  outStand1 :: (((String → String → IO ()) → IO ()) → IO ()
  |
  outStand1 f = f (\pat → putStrLn . CodeConstPrintf.sprintf1 pat)
  |
  code-module CodeConst.Monad ↔ (Haskell) :
    bind a = (>>=) a
    return :: a → IO a
  |
  return = Prelude.return
  |
  code-module CodeConst.Printf ↔ (Haskell) :
    import Text.Printf
    printf0 = id

  |
  printf1 :: PrintfArg a ⇒ String → a → String
  |
  printf1 = printf

  |
  printf2 :: PrintfArg a ⇒ PrintfArg b ⇒ String → a → b → String
```
; sprintf2 = printf

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

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

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

\[ \text{concat } s \]\[
\begin{array}{l}
\text{concat } s \cdot [x : xs] = x ++ \text{concatMap } ((++) s) \cdot xs
\end{array}\]

\[ \text{concat s (x : xs) = x ++ concatMap ((++) s) \: xs} \]

\[ \text{concat s x} = x ++ \text{concatMap } ((++) s) \\cdot xs \]

; | code-module CodeConst.String \rightarrow\text{(Haskell)} \rangle
; \text{concat s [] = []} \newline
; \text{concat s (x : xs)} = x ++ \text{concatMap } ((++) s) \cdot xs \newline

; | code-module CodeConst.Sys \rightarrow\text{(Haskell)} \rangle
; \text{import System.Directory} \newline
; \text{isDirectory2 = doesDirectoryExist} \newline

; | code-module CodeConst.To \rightarrow\text{(Haskell)} \rangle
; \text{nat = id} \newline

; | code-module \rightarrow\text{(OCaml)} \rangle

\begin{align*}
\text{module CodeType = struct} \\
\text{type mlInt = int} \\
\text{end} \\
\text{module CodeConst = struct} \\
\text{let outFile1 f file =} \\
\text{try} \\
\text{let () = if Sys.file-exists file then Printf.cprintf File exists \%S\\n file else () in} \newline
\text{let oc = open-out file in} \\
\text{let b = f (fun s a -> try Some (Printf.fprintf oc s a) with - \rightarrow None) in} \newline
\text{let () = close-out oc in} \\
\text{b} \newline
\text{with - \rightarrow None} \\
\text{let outStand1 f =} \\
\text{f (fun s a -> try Some (Printf.fprintf stdout s a) with - \rightarrow None)} \\
\text{module Monad = struct} \\
\text{let bind = function} \\
\text{None \rightarrow \text{fun} \rightarrow None} \\
\text{| Some a \rightarrow \text{fun f \rightarrow f a}} \newline
\text{let return a = Some a} \newline
\text{end} \\
\end{align*}
module Printf = struct
  include Printf
  let sprintf0 = sprintf
  let sprintf1 = sprintf
  let sprintf2 = sprintf
  let sprintf3 = sprintf
  let sprintf4 = sprintf
  let sprintf5 = sprintf
end

module String = String

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

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

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)
    }
  }
}
```scheme
(\[ A \] (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 (l . 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) |
| code-block |

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

end
in
fun sprintf0 s-pat = s-pat
fun sprintf1 s-pat s1 = sprintf s-pat [s1]
fun sprintf2 s-pat s1 s2 = sprintf s-pat [s1, s2]
fun sprintf3 s-pat s1 s2 s3 = sprintf s-pat [s1, s2, s3]
fun sprintf4 s-pat s1 s2 s3 s4 = sprintf s-pat [s1, s2, s3, s4]
fun sprintf5 s-pat s1 s2 s3 s4 s5 = sprintf s-pat [s1, s2, s3, s4, s5]
end

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 := Printfsprintf1 a b :: (Unsynchronized! oc)) in
      SOME (File.write-list pfile (rev (Unsynchronized! oc))) handle - => NONE
    )
  in
    SOME (File.writeFile pfile (unsynchronized! stdout-file))
  end

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

4.1.2. Interface with Types

datatype ml-int = ML-int
code-printing type-constructor ml-int -> (Haskell) CodeType.MlInt - syntax!
  | type-constructor ml-int -> (OCaml) CodeType.mlInt
  | type-constructor ml-int -> (Scala) CodeType.mlInt
  | type-constructor ml-int -> (SML) CodeType.mlInt

datatype 'a ml-monad = ML-monad 'a
code-printing type-constructor ml-monad -> (Haskell) CodeType.MlMonad - syntax!
4.1.3. Interface with Constants

module CodeConst

consts out-file1 :: ((ml-string ⇒ 'α1 ⇒ unit ml-monad) — fprintf ⇒ unit ml-monad) ⇒ ml-string ⇒ unit ml-monad

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

consts out-stand1 :: ((ml-string ⇒ 'α1 ⇒ unit ml-monad) — fprintf ⇒ unit ml-monad) ⇒ unit ml-monad

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

module Monad

consts bind :: 'a ml-monad ⇒ ('a ⇒ 'b ml-monad) ⇒ 'b ml-monad

code-printing constant bind ⇀ (Haskell) CodeConst.Monad.bind
| constant bind ⇀ (OCaml) CodeConst.Monad.bind
| constant bind ⇀ (Scala) CodeConst.Monad.bind
| constant bind ⇀ (SML) CodeConst.Monad.bind

consts return :: 'a ⇒ 'a ml-monad

code-printing constant return ⇀ (Haskell) CodeConst.Monad.return
| constant return ⇀ (OCaml) CodeConst.Monad.return
| constant return ⇀ (Scala) CodeConst.Monad.Return — syntax!
| constant return ⇀ (SML) CodeConst.Monad.return

module Printf

consts sprintf0 :: ml-string ⇒ ml-string

code-printing constant sprintf0 ⇀ (Haskell) CodeConst.Printf sprintf0
| constant sprintf0 ⇀ (OCaml) CodeConst.Printf sprintf0
| constant sprintf0 ⇀ (Scala) CodeConst.Printf sprintf0
| constant sprintf0 ⇀ (SML) CodeConst.Printf sprintf0

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

code-printing constant sprintf1 ⇀ (Haskell) CodeConst.Printf sprintf1
| constant sprintf1 ⇀ (OCaml) CodeConst.Printf sprintf1
| constant sprintf1 ⇀ (Scala) CodeConst.Printf sprintf1
| constant sprintf1 ⇀ (SML) CodeConst.Printf sprintf1
consts sprint2 :: ml-string ⇒ 'α1 ⇒ 'α2 ⇒ ml-string
code-printing constant sprint2 → (Haskell) CodeConst.Printf.sprintf2
  | constant sprint2 → (OCaml) CodeConst.Printf.sprintf2
  | constant sprint2 → (Scala) CodeConst.Printf.sprintf2
  | constant sprint2 → (SML) CodeConst.Printf.sprintf2

consts sprint3 :: ml-string ⇒ 'α1 ⇒ 'α2 ⇒ 'α3 ⇒ ml-string
code-printing constant sprint3 → (Haskell) CodeConst Printf.sprintf3
  | constant sprint3 → (OCaml) CodeConst Printf.sprintf3
  | constant sprint3 → (Scala) CodeConst Printf.sprintf3
  | constant sprint3 → (SML) CodeConst Printf.sprintf3

consts sprint4 :: ml-string ⇒ 'α1 ⇒ 'α2 ⇒ 'α3 ⇒ 'α4 ⇒ ml-string
code-printing constant sprint4 → (Haskell) CodeConst Printf.sprintf4
  | constant sprint4 → (OCaml) CodeConst Printf.sprintf4
  | constant sprint4 → (Scala) CodeConst Printf.sprintf4
  | constant sprint4 → (SML) CodeConst Printf.sprintf4

consts sprint5 :: ml-string ⇒ 'α1 ⇒ 'α2 ⇒ 'α3 ⇒ 'α4 ⇒ 'α5 ⇒ ml-string
code-printing constant sprint5 → (Haskell) CodeConst Printf.sprintf5
  | constant sprint5 → (OCaml) CodeConst Printf.sprintf5
  | constant sprint5 → (Scala) CodeConst Printf.sprintf5
  | constant sprint5 → (SML) CodeConst Printf.sprintf5

module String

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

module Sys

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

module To

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

4.1.4. Some Notations (1): Raw Translations

syntax -sprint0 :: - ⇒ ml-string (sprint0 (-)' )
translations sprint0 x' := CONST sprintf0 x

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

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

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

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

syntax -sprint5 :: - ⇒ - ⇒ ml-string (sprint5 (-)´)
translations sprint5 x' := CONST sprintf5 x

4.1.5. Some Notations (II): Polymorphic Cartouches

syntax -cartouche-string' :: String.literal
translations -cartouche-string := -cartouche-string'

⟨ML⟩

4.1.6. Generic Locale for Printing

locale Print =
  fixes To-string :: string ⇒ ml-string
  fixes To-nat :: nat ⇒ ml-int
begin
  declare[cartouche-type' = fun_printf]
end

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

end

4.2. Instantiating the Printer for (Pure) Term

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

context Print
begin

fun of-pure-term where of-pure-term l e = (λ
  Const s - ⇒ To-string s
4.3. Instantiating the Printer for SML

theory Printer-SML
imports Meta-SML
  Printer-init
begin
  context Print
  begin
    definition of-semi--val-fun = (λ Sval ⇒ ⟨val⟩ | Sfun ⇒ ⟨fun⟩)
    fun of-semi--term' where ⟨of-semi--term' e⟩ = (λ
      SML-string s ⇒ ⟨s⟩ (To-string (escape-sml s))
    | SML-rewrite val-fun e1 symb e2 ⇒ ⟨s s s s⟩ (of-semi--val-fun val-fun) (of-semi--term' e1) (To-string symb) (of-semi--term' e2)
    | SML-basic l ⇒ ⟨s⟩ (String-concat ⟨⟩ (List.map To-string l))
    | SML-binop e1 s e2 ⇒ ⟨s s s⟩ (of-semi--term' e1) (of-semi--term' (SML-basic [s])) (of-semi--term' e2)
    | SML-annot e s ⇒ ⟨s s⟩ (of-semi--term' e) (To-string s)
    | SML-function l ⇒ ⟨fn⟩ (String-concat ⟨⟩ (List.map (λ e ⇒ ⟨%s⟩) (of-semi--term' e) l))
    | SML-apply e l ⇒ ⟨s s⟩ (of-semi--term' e) (String-concat ⟨⟩ (List.map (λ e ⇒ ⟨%s⟩) (of-semi--term' e) l))
    | SML-paren p-left p-right e ⇒ ⟨s s s⟩ (To-string p-left) (of-semi--term' e) (To-string p-right)
    | SML-let-open s e ⇒ ⟨let open %s in %s end⟩ (To-string s) (of-semi--term' e) e
    end
  end

lemmas [code] =
  — def
4.4. Instantiating the Printer for Isabelle

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

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

fun of-semi--term :: (λ
term-rewrite e1 symb e2 ⇒ ⟨%s %s %s⟩ (of-semi--term e1) (To-string symb) (of-semi--term e2)
| term-basic l ⇒ ⟨%s⟩ (String-concat ⟨ ⟩ L.map To-string l)
| term-annot e s ⇒ ⟨%s %s⟩ (of-semi--term e) (of-semi--typ s)
| term-bind symb e1 e2 ⇒ ⟨%s %s %s⟩ (To-string symb) (of-semi--term e1) (of-semi--term e2)

end

Print.of-semi--val-fun-def
— fun
Print.of-semi--term'.simps

context Print
begin

fun of-semi--typ where of-semi--typ e = (λ
Typ-base s ⇒ To-string s
| Typ-apply name l ⇒ ⟨%s⟩ (List.map of-semi--typ l) in
case l of [ ] ⇒ s | - ⇒ (⟨%s⟩ s)
(of-semi--typ name)
| Typ-apply-bin s ty1 ty2 ⇒ ⟨%s⟩ (of-semi--typ ty1) (To-string s) (of-semi--typ ty2)
| Typ-apply-paren s1 s2 ty ⇒ ⟨%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 ⟨ ⟩
(λ(n,l).
% s % s)
(To-string n)
(String-concat ⟨ ⟩ L.map (λx. %s (of-semi--typ x)) l))) l ))

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

definition of-instantiation = \( \lambda \) Instantiation n n-def expr ⇒
let name = To-string n in
\( \alpha \) Instantiation (%) : object
begin
\( \alpha \) Instantiation (%) (%) (To-string n-def)
instance ..
end:
name
(To-string n-def)
name
(To-string expr))

definition of-overloading = \( \lambda \) Overloading n-c e-c n e ⇒
\( \alpha \) Overloading (%) (%) (To-string n) (To-string e-c) (To-string n) (To-string e))

definition of-consts = \( \lambda \) Constrs n ty symb ⇒
\( \alpha \) Constrs (%) (%) (%) (%) (To-string n) (To-string ty) (To-string Constrs-value) (To-string symb))

definition of-definition = \( \lambda \)
\( \alpha \) Definition e ⇒ \( \alpha \) Definition (%) (To-string e)
\( \alpha \) Definition-where1 name (abbrev, prio) e ⇒ \( \alpha \) Definition (%) %e %d
\( \alpha \) Definition-where2 name abbrev e ⇒ \( \alpha \) Definition (%) %e
\( \alpha \) Definition-where3 name abbrev (To-string name) (To-string abbrev) (To-string prio) (To-string e)
\( \alpha \) Definition-where4 name abbrev e ⇒ \( \alpha \) Definition (%) %e
\( \alpha \) Definition-where5 name abbrev (To-string name) (To-string abbrev) (To-string e)

definition (of-semi--thm-attribute-aux-gen :: String.literal × String.literal ⇒ - ⇒ - ⇒ -) m lacc s =
\( \alpha \) Definition-where6 name abbrev e ⇒ \( \alpha \) Definition (%) %e
\( \alpha \) Definition-where7 name abbrev (To-string name) (To-string abbrev) (To-string e)
\( \alpha \) Definition-where8 name abbrev e ⇒ \( \alpha \) Definition (%) %e
\( \alpha \) Definition-where9 name abbrev (To-string name) (To-string abbrev) (To-string e)
\( \text{in} \)
\( s\text{-base } s \ (m \neq \text{lacc}) \)

**Definition** \( \text{of-semi-thm-attribute-aux-gen-where } l = (\langle \text{where} \rangle, \text{String-concat \ and } ; (L\text{-map } (\lambda \text{var, expr}. \%s = \%s) \ (\text{To-string } \text{var}) \ (\text{of-semi-term expr})) \ l)) \)

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

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

\[
\text{Thm-OF (Thm-thms } s \text{) } e2 \Rightarrow \text{of-semi-thm-attribute-aux } \langle \text{OF} \rangle, \text{of-semi-thm-attribute-aux} \]

\[ e2 \rightarrow \text{lacc } s \]

\[
\text{Thm-OF } e1 \ e2 \Rightarrow \text{of-semi-thm-attribute } \langle \langle \text{OF} \rangle, \text{of-semi-thm-attribute-aux} \rangle \ e2 \ # \ (\text{lacc } e1) \ e
\]

**Definition**

\[
\text{of-semi-thm-attribute } = \text{of-semi-thm-attribute-aux } \langle \text{⟩}
\]

**Definition**

\[
\text{of-semi-thm } = (\lambda \text{Thms-single thy }\Rightarrow \text{of-semi-thm-attribute thy} \text{)}
\]

| Thms-mult thy ⇒ of-semi-thm-attribute thy |

**Definition**

\[
\text{of-semi-thm-attribute-l } l = \text{String-concat } (L.\text{map of-semi-thm-attribute } l)
\]

**Definition**

\[
\text{of-semi-thm-attribute-l1 } l = \text{String-concat } (\langle \rangle (L.\text{map of-semi-thm-attribute } l))
\]

**Definition**

\[
\text{of-semi-thm-l } l = \text{String-concat } (\langle \rangle (L.\text{map of-semi-thm } l))
\]

**Definition**

\[
\text{of-semi-attrib-genA } :: (\text{semi-thm list } \Rightarrow \text{String.literal}) \Rightarrow \text{String.literal } \Rightarrow \text{semi-thm list } \Rightarrow \text{String.literal}) \ f \ \text{attr } l = — \text{error reflection: to be merged}
\]

| (if \ l = [] then \\
| () \\
| else \\
| \langle %s: %s: attr (f l) \rangle |

**Definition**

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

| (if \ l = [] then \\
| () \\
| else \\
| \langle %s: %s: attr (f l) \rangle |

**Definition**

\[
\text{of-semi-attrib } = \text{of-semi-attrib-genA of-semi-thm-l}
\]

**Definition**

\[
\text{of-semi-attrib1 } = \text{of-semi-attrib-genB } (\lambda l. \text{String-concat } (\langle \rangle (L.\text{map To-string } l)))
\]

**Definition**

\[
\text{of-semi-method-simp } (s :: — \text{polymorphism weakening needed by code-reflect String.literal}) =
\]

| (\lambda \text{Method-simp-only } l ⇒ \langle %s only: %s s (of-semi-thm-l l) |
| \text{Method-simp-add-del-split } l1 \ l2 \ [] ⇒ \langle %s s s s s \rangle |

45
fun of-semi-method where of-semi-method expr = (λ Method-rule o-s ⇒ ⟨rule%s⟩ (of-semi-thm-attribute s))
| Method-drule s ⇒ ⟨drule%s⟩ (of-semi-thm-attribute s)
| Method-intro l ⇒ ⟨intro%s⟩ (of-semi-thm-attribute-l1 l)
| Method-elim s ⇒ ⟨elim%s⟩ (of-semi-thm-attribute s)
| Method-subst asm l s = let s-asm = if asm then ⟨asm⟩ else ⟨⟩ in
| Method-insert l s => ⟨insert%s⟩ (of-semi-thm-l l)
| Method-plus t ⇒ ⟨(s)+⟩ (List.map of-semi-method t)
| Method-option t ⇒ ⟨(s)?⟩ (List.map of-semi-method t)
| Method-one s ⇒ of-semi-simp-add-split simp l-simp
| Method-rename-tac l ⇒ ⟨rename-tac%s⟩ (List.map To-string l)
| Method-case-tac e ⇒ ⟨case-tac%s⟩ (of-semi-term e)
| Method-blast None ⇒ ⟨blast⟩
| Method-blast (Some n) ⇒ ⟨blast%d⟩ (To-nat n)
| Method-clarify ⇒ ⟨clarify⟩
| Method-metis l-opt l ⇒ ⟨metis%s⟩ (if l-opt = [] then ⟨⟩ else ⟨(s)%⟩ (List.map To-string l-opt))
\[
\text{definition (of-semi--command-proof) = (}
\]
\[
\begin{align*}
\text{let thesis} &= \langle \text{thesis} \rangle \\
\text{scope-thesis-gen} &= \lambda \text{proof show when. \langle proof \rangle show \%s \rangle \text{thesis} [] in}
\end{align*}
\]
\[
\begin{align*}
\text{Command-apply} [] \Rightarrow \langle \rangle & \\
\text{Command-apply} l-apply \Rightarrow \langle \text{apply}(\%s) \rangle \\
\text{Command-using} l \Rightarrow \langle \text{using \%s} \rangle \\
\text{(of-semi--thm-l l)} & \\
\text{Command-unfolding} l \Rightarrow \langle \text{unfolding \%s} \rangle \\
\text{(of-semi--thm-l l)} & \\
\text{Command-let e-name e-body} \Rightarrow \text{scope-thesis} (\langle \text{let \%s = \%s} \rangle \text{of-semi--term e-name} \text{of-semi--term e-body}) \\
\text{Command-have n b e e-last} \Rightarrow \text{scope-thesis} (\langle \text{have \%s \%s: \%s \%s} \rangle \text{To-string n} \text{if b then (simp) else 0} \rangle \text{of-semi--command-final e-last}) \\
\text{Command-fix-let l l-let o-show -} \Rightarrow \\
\text{scope-thesis-gen} (\langle \text{fix \%s \%s} \rangle \text{String-concat} (\langle \text{map To-string l} \rangle \text{String-concat} \langle \text{map (\%s = \%s)} \rangle \text{of-semi--term e-name} \text{of-semi--term e-body} \rangle) \\
\text{l-let}) \\
\text{case o-show of None \Rightarrow thesis} \\
\text{case o-show of Some \langle l-show, \_ \rangle \Rightarrow \%s \rangle \text{String-concat} \langle \_ \rangle \text{of-semi--term l-show}) \\
\text{case o-show of None \Rightarrow [] | Some \langle \_ \rangle \Rightarrow l-when}) \\
\end{align*}
\]
\[
\text{definition of-lemma - =} \\
(\lambda \text{Lemma n l-spec l-apply tactic-last} \Rightarrow \\
\text{lemma \%s : \%s} \\
\%s \%s) \\
\text{(To-string n)} \\
\text{String-concat} \langle \_ \rangle \text{of-semi--term l-spec}) \\
\text{of-semi--command-final tactic-last}) \\
\text{Lemma-assumes n l-spec concl l-apply tactic-last} \Rightarrow
\]\
(lemma %s : %s
%ss%ss %s
  (To-string n)
  (String-concat o (L.map (λ (n, b, e).
    (assumes %s
      (let (n, b) = if b then (%s[simp]) (To-string n), False) else (To-string n, String.is-empty
        n) in
        if b then o else (%s : n)
      (of-semi-term e)) l-spec
  @@ @
    [])
  shows %s
    (of-semi-term concl))))
  (String-concat o (L.map of-semi-command-proof l-apply))
  (of-semi-command-final tactic-last)
  (String-concat )
  (L.map
    (λ l-apply-e.
      %sqed:
      (if l-apply-e = [] then
        ()
      else
        ( %s
          (String-concat o (L.map of-semi-command-state l-apply-e))))
      (List.map-filter
        (λ Command-let - - ⇒ Some [] | Command-have - - - ⇒ Some [] | Command-fix-let
          - - - l ⇒ Some l | - ⇒ None)
        (rev l-apply)))))))

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

definition of-section - = (λ Section n section-title ⇒
%ss%sector
  (%sssection
    (if n = 0 then ()
    else if n = 1 then (sub)
    else (subsub))
  (To-string section-title))

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

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

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

definition of-thm - = (λ Thm thm ⇒ (thm %s) (of-semi-thm-attribute-I thm))
definition \texttt{of-interpretation} - = (\lambda \text{Interpretation } n \text{ loc-n loc-param tac} \Rightarrow \\
\\text{interpretation } \%s: \%s/\%s \\
\\text{(To-string } n) \\
\\text{(To-string loc-n) } \\
\text{(String-concat } \circ (\text{L.map } (\lambda s. (\%s) (\text{of-semi--term } s)) \text{ loc-param})) \\
\text{(of-semi--command-final } tac)) \\

definition \texttt{of-semi--theory } env = \\
(\lambda \text{Theory-datatype } dataty \Rightarrow \text{of-datatype } env \text{ dataty} \\
| \text{Theory-type-synonym } ty-synonym \Rightarrow \text{of-type-synonym } env \text{ ty-synonym} \\
| \text{Theory-type-notation } ty-notation \Rightarrow \text{of-type-notation } env \text{ ty-notation} \\
| \text{Theory-instantiation } instantiation-class \Rightarrow \text{of-instantiation } env \text{ instantiation-class} \\
| \text{Theory-overloading } overloading \Rightarrow \text{of-overloading } env \text{ overloading} \\
| \text{Theory-consts } consts-class \Rightarrow \text{of-consts } env \text{ consts-class} \\
| \text{Theory-definition } definition-hol \Rightarrow \text{of-definition } env \text{ definition-hol} \\
| \text{Theory-lemmas } lemmas-simp \Rightarrow \text{of-lemmas } env \text{ lemmas-simp} \\
| \text{Theory-lemma } lemma-by \Rightarrow \text{of-lemma } env \text{ lemma-by} \\
| \text{Theory-axiomatization } axiom \Rightarrow \text{of-axiomatization } env \text{ axiom} \\
| \text{Theory-section } section-title \Rightarrow \text{of-section } env \text{ section-title} \\
| \text{Theory-text } text \Rightarrow \text{of-text } env \text{ text} \\
| \text{Theory-ML } ml \Rightarrow \text{of-ML } env \text{ ml} \\
| \text{Theory-setup } setup \Rightarrow \text{of-setup } env \text{ setup} \\
| \text{Theory-thm } thm \Rightarrow \text{of-thm } env \text{ thm} \\
| \text{Theory-interpretation } thm \Rightarrow \text{of-interpretation } env \text{ thm}) \\

definition \texttt{String-concat-map } s f l = \text{String-concat } s \text{ (L.map } f \text{ l)
lemmas [code] =
  — def
  Print.of-datatype-def
  Print.of-type-synonym-def
  Print.of-type-notation-def
  Print.of-instantiation-def
  Print.of-overloading-def
  Print.of-consts-def
  Print.of-definition-def
  Print.of-semi-thm-attribute-aux-gen-def
  Print.of-semi-thm-attribute-aux-gen-where-def
  Print.of-semi-thm-attribute-aux-gen-of-def
  Print.of-semi-thm-attribute-def
  Print.of-semi-thm-def
  Print.of-semi-thm-attribute-l-def
  Print.of-semi-thm-attribute-l1-def
  Print.of-semi-thm-l-def
  Print.of-lemmas-def
  Print.of-semi-attrib-genA-def
  Print.of-semi-attrib-genB-def
  Print.of-semi-attrib-def
  Print.of-semi-attrib1-def
  Print.of-semi-method-simp-def
  Print.of-semi-command-final-def
  Print.of-semi-command-state-def
  Print.of-semi-command-proof-def
  Print.of-lemma-def
  Print.of-axiomatization-def
  Print.of-section-def
  Print.of-text-def
  Print.of-ML-def
  Print.of-setup-def
  Print.of-thm-def
  Print.of-interpretation-def
  Print.of-semi-theory-def
  Print.String-concat-map-def
  Print.of-semi-theories-def

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

end
5. Main

We present two solutions for obtaining an Isabelle file.

5.1. Static Meta Embedding with Exportation

theory Generator-static
imports Printer
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 None (oidInit (Oid 0)) Gen-only-design (None, False)
 | D-output-disable-thy := False
 , D-output-header-thy := Some (⟨Design-generated:
 ;[⟨../Toy-Library] ;⟨../embedding/Generator-dynamic-sequential⟩⟩)
 ( L.map (META-class-raw Floor1) Design
 @@ @@ | META-association (toy-association.make
 ToyAssTy-association
(ToyAssRel [ (n : Person, ToyMult (Mult-star, None) None) None
 , (n : Person, ToyMult (Mult-null 0, Some (Mult-nat 1))
  , (Some (boss)))])
 , META-flush-all ToyFlushAll]
 , None)])

5.1.2. Statically Executing the Exportation

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

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

end

5.2. Dynamic Meta Embedding with Reflection

definition Generator-dynamic-sequential

theory Generator-dynamic-sequential

import Printer

../../isabelle-home/src/HOL/Isabelle-Main2

begin

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

- The first is to generate an Isabelle file for inspection or debugging. The generated file can interactively be loaded in Isabelle/jEdit, or saved to the hard disk. This mode is called the “deep exportation” mode or shortly the “deep” mode. The aim is to maximally automate the process one is manually performing in Generator_static.thy.

- On the other hand, it is also possible to directly execute in Isabelle/jEdit the generated file from the random access memory. This mode corresponds to the “shallow reflection” mode or shortly “shallow” mode.

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

apply-code-printing-reflect

val stdout-file = Unsynchronized.ref

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

write-file

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

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

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

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⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩⟨ML⟩end
Part II.

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

theory Toy-Library
imports Main
begin

type-notation option (⟨⋅⟩⊥)
notation Some (⌊⋅⌋)

fun drop :: ‘α option ⇒ ‘α (⌈⟨⋅⟩⌉)
where drop-lift[simp]: ⌈⌊v⌋⌉ = v

type-synonym oid = nat

record (‘α)state =
  heap :: oid ⇒ ‘α
  assocs :: oid ⇒ ((oid list) list) list

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

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

5.4.1. Introduction

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

5.4.2. Designing Class Models (I): Basics

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

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

Class Atom < Molecule
    Attributes size : Integer
End

The “blue” color of End indicates that End is not a “green” keyword. End and Class are in fact similar, they belong to the group of meta-commands (all meta-commands are defined in Isabelle-Meta-Model Generator-dynamic-sequential). At run-time and in deep mode, the semantics of all meta-commands are approximately similar: all meta-commands displays some quantity of Isabelle code in the output window (as long as
meta-commands are syntactically correctly formed). However each meta-command is unique because what is displayed in the output window depends on the sequence of all meta-commands already encountered before (and also depends on arguments given to the meta-commands).

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

\texttt{End End End}

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

\texttt{Class Molecule < Person}

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

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

\texttt{End} can be seen as a lazy meta-command:

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

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

\texttt{End!}

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

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

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

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

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

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

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

\((* \text{ Class Big}_\text{Bang} < \text{Atom} (* \text{This will force the creation of a new universe.} \,*\*) *)\)

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

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

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

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

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

**State** \( \sigma_1 = \)

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

**State** \( \sigma_1' = \)

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

In certain circumstances, the command **setup** must be added again between some par-
ticular interleaving of two meta-commands and this may not depend on the presence of the `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 \}[ \{ [ salary = 1000 , boss = self 1 ] : Person] *)
```

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

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

PrePost[shallow] \{sigma\}<sub>1</sub> WFF_10_post *)
```

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

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

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

State[shallow] WFF_10_post = \[ WFF_10_post_object0 \]

PrePost[shallow] \{sigma\}<sub>1</sub> WFF_10_post *)
```

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

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

Context Person :: content ()

Post "\<close>\<open>"
Here the expression \(<\text{close}>\<\text{open}>\) is not well-typed in Isabelle, but an error is not raised because the above expression is not (yet) parsed as an Inner Syntax element\(^1\). However, this is not the same for the resulting generated meta-command:

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

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:

\[
\text{Context[shallow]} \text{Person :: content ()}
\text{Post : } a + b = c
\]

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:

\[
(* \text{generation_syntax deep flush_all } *)
\]

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

According to options given to the (first) command \texttt{generation-syntax} above, we retrieve the first generated file in the mentioned directory: 

\[
../\text{document\_generated/}
\text{Design\_generated.thy}
\]

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 

\[
../\text{document\_generated/}
\text{Design\_generated\_generated.thy}
\]

As remark, in this last file, the dependency to \texttt{Isabelle-Meta-Model.Generator-dynamic-sequential} was automatically removed because the meta-compiler has detected the absence of meta-commands in the generated content.

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

\[
\text{end}
\]

\(^1\) 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!

62
5.5. Example: A Class Model Interactively Executed

5.5.1. Introduction

theory
  Design-shallow
imports
  ../Toy-Library
  ../Toy-Library-Static
  ../embedding/Generator-dynamic-sequential
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 Isabelle-Meta-Model.Generator-dynamic-sequential, nothing is generated in this theory. The system only parses arguments given to meta-commands and immediately calls the corresponding compiled functions.

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

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

]  

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

5.5.2. Designing Class Models (I): Basics

Class Atom < Molecule
  Attributes size : Integer
End

End End End

Class Molecule < Person

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

Class Person < Galaxy
  Attributes salary : Integer
  boss : Person
  is-meta-thinking : Boolean
Instance \( \mathbf{X}\text{Person}^1 :: \text{Person} = [ \text{salary} = 1300 , \text{boss} = \mathbf{X}\text{Person}^2 ] \)
and \( \mathbf{X}\text{Person}^2 :: \text{Person} = [ \text{salary} = 1800 ] \)

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

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

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

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

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:

\[
(* \text{Context } \text{Person} :: \text{content} ()
\text{Post } "\langle\text{close}\rangle\langle\text{open}\rangle" \*)
\]

Context [shallow] \( \text{Person} :: \text{content} () \)
\text{Post} : a + b = c

end
Bibliography

Part III.

Appendix
A. Grammars of Commands

(ML)

A.1. Main Setup of Meta Commands

\[
\text{generation-syntex} : \text{theory} \rightarrow \text{theory}
\]
deep-embedding
export-code

\begin{align*}
\text{module\_name} & \quad \text{name} \\
\text{args} & \quad )
\end{align*}

\text{long-or-dirty}

\begin{align*}
\text{SORRY} \\
\text{no\_dirty}
\end{align*}

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

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

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

As a side note, target languages for the \text{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, `SORRY` allows to explicitly skip the evaluation of all proofs, irrespective of the presence of `sorry` or not in generated proofs. In any cases, the semantics of `sorry` has not been overloaded, e.g., red background may appear as usual.

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

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><code>theory → theory</code></td>
</tr>
<tr>
<td><strong>Abstract-class</strong></td>
<td><code>theory → theory</code></td>
</tr>
</tbody>
</table>

- `Class`: The `Class` command takes a `theory` as input and outputs another `theory`.
- `Abstract-class`: The `Abstract-class` command also takes a `theory` as input and outputs another `theory`.

```
Class: theory → theory
Abstract-class: theory → theory
```

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

Attributes

binding : toy-type

context

context

Operations

binding : toy-type

= term

Pre

Post

term

use-prop

invariant

invariant

inv

Inv

use-prop

Constraints

Existential

Aggregation : theory → theory

Association : theory → theory

Composition : theory → theory
Aggregation
Association
Composition

association

Between
association-end
association-end

association-end

[type-object] [category]

;  

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

Associationclass
Abstract_associationclass

[association] [class]

aggregation
composition

Context : theory → theory

Context

1 shallow 1
Instance : theory \rightarrow theory

State : theory \rightarrow theory
\begin{align*}
\text{State} & \quad \text{binding} \\
\text{state} & \quad \text{binding} \\
\text{PrePost} : \ & \text{theory} \to \text{theory} \\
\text{End} : \ & \text{theory} \to \text{theory} \\
\end{align*}
A.3. Extensions of Isabelle Commands

`code-reflect` : \textit{theory} $\rightarrow$ \textit{theory}

`code-reflect` has the same semantics as `code-reflect` except that it additionally contains the option `open` inspired from the command `export-code` (with the same semantics).
**lazy-code-printing** : theory → theory

**apply-code-printing** : theory → theory

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

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

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

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

B.1. Extensions for Cartouches

- `./src/HOL/ex/Isabelle_Cartouche_Examples.thy`  \textit{Main0}:
  Some functions have been generalized for supporting cartouches.

B.2. Other Changes

- `./src/Tools/Code/Isabelle_code_runtime.thy`  \textit{Main1}:
  The option \textit{open} was introduced in this file for the definition of \textit{code\_reflect}'.

- `./src/Tools/Code/Isabelle_code_target.thy`  \textit{Main1}:
  Some signatures was removed for exposing the main structure, we have also defined at the end the implementation of \textit{lazy\_code\_printing}, \textit{apply\_code\_printing} and \textit{apply\_code\_printing\_reflect}.

- `./src/Pure/Isar/Isabelle_typedecl.thy`  \textit{Main2}:
  Short modification of the argument lifting a \textit{binding} to a \textit{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:

```plaintext
datatype tyEXTAtom = mkEXTAtom oid oid list option int option bool option nat option unit option

datatype tyAtom = mkAtom tyEXTAtom int option

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

datatype tyMolecule = mkMolecule tyEXTMolecule

datatype tyEXTPerson = mkEXTPerson-Molecule tyMolecule
| mkEXTPerson-Molecule-Atom tyAtom
| mkEXTPerson oid nat option unit option

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

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

datatype tyGalaxy = mkGalaxy tyEXTGalaxy nat option unit option

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

datatype tyToyAny = mkToyAny tyEXTToyAny
```

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

datatype $A = \text{in}_{\text{Atom}} \text{ty}_{\text{Atom}}$
$\mid \text{in}_{\text{Molecule}} \text{ty}_{\text{Molecule}}$
$\mid \text{in}_{\text{Person}} \text{ty}_{\text{Person}}$
$\mid \text{in}_{\text{Galaxy}} \text{ty}_{\text{Galaxy}}$
$\mid \text{in}_{\text{ToyAny}} \text{ty}_{\text{ToyAny}}$

definition $\text{oid}_{\text{Atom}}::\text{boss} = 0$
definition $\text{oid}_{\text{Molecule}}::\text{boss} = 0$
definition $\text{oid}_{\text{Person}}::\text{boss} = 0$

definition $\text{oid}_1 = 1$
definition $\text{oid}_2 = 2$
definition $\text{inst-assoc}_1 = (\lambda \text{oid-class to-from oid}. ((\text{case (deref-assocs-list (to-from::oid list list} 
\Rightarrow oid list \times oid list)) ((oid::oid)) ((\text{drop ((map-of-list ((oid::oid))::oid list option)) of Nil} \Rightarrow \text{None]))) 
\mid l \Rightarrow \text{(Some (l)))::oid list option}))$

definition $\text{oid}_3 = 3$
definition $\text{inst-assoc}_3 = (\lambda \text{oid-class to-from oid}. ((\text{case (deref-assocs-list (to-from::oid list list} 
\Rightarrow oid list \times oid list)) ((oid::oid)) ((\text{drop (map-of-list (\text{[]}))) of Nil} \Rightarrow \text{None}) 
\mid l \Rightarrow \text{(Some (l)))::oid list option}))$

definition $\text{oid}_4 = 4$
definition $\text{oid}_5 = 5$
definition $\text{oid}_6 = 6$
definition $\text{oid}_7 = 7$
definition inst-assoc4 = (\oid-class to-from oid. ((case (deref-assocs-list ((to-from::oid list list ⇒ oid list × oid list)) (oid::oid)) ((drop (((map-of-list (((\oid1 Person.0--boss , (List.map ((\x y. [x 1 y]) o switch2-01) [[[\oid7 , [\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_{\mathbf{Person}} :: \mathbf{ty}\mathbf{Person}
fixes \sigma_1-object0 :: \mathbf{Person}
assumes \sigma_1-object0-def: \sigma_1-object0 = (\lam\ oid0_{\mathbf{Person}} :: \mathbf{ty}\mathbf{Person}
fixes \sigma_1-object1 :: \mathbf{Person}
assumes \sigma_1-object1-def: \sigma_1-object1 = (\lam\ oid1_{\mathbf{Person}} :: \mathbf{ty}\mathbf{Person}
fixes \sigma_1-object2 :: \mathbf{Person}
assumes \sigma_1-object2-def: \sigma_1-object2 = (\lam\ oid2_{\mathbf{Person}} :: \mathbf{ty}\mathbf{Person}
fixes X\mathbf{Person}1 :: \mathbf{Person}
assumes X\mathbf{Person}1-def: X\mathbf{Person}1 = (\lam\ X\mathbf{Person}1_{\mathbf{Person}})
fixes \sigma_1-object4 :: \mathbf{Person}
assumes \sigma_1-object4-def: \sigma_1-object4 = (\lam\ oid4_{\mathbf{Person}} :: \mathbf{ty}\mathbf{Person}
fixes X\mathbf{Person}2 :: \mathbf{Person}
assumes X\mathbf{Person}2-def: X\mathbf{Person}2 = (\lam\ X\mathbf{Person}2_{\mathbf{Person}})
begin
definition \sigma_1 = (state.make ((\Map.empty (oid4 ⇒ (in\mathbf{Person} (\sigma1-object0_{\mathbf{Person}}))) (oid5 ⇒ (in\mathbf{Person} (\sigma1-object1_{\mathbf{Person}}))) (oid6 ⇒ (in\mathbf{Person} (\sigma1-object2_{\mathbf{Person}}))) (oid1 ⇒ (in\mathbf{Person} (\sigma1-object1_{\mathbf{Person}}))))\mathbf{X}\mathbf{Person}1_{\mathbf{Person}}))\mathbf{X}\mathbf{Person}2_{\mathbf{Person}})))
((map-of-list (((\oid_{\mathbf{Person}}.0--boss , (List.map ((\lam\ x y. [x y]) o switch2-01) [[[\oid4] , [\oid2] , [\oid1] , [\oid6] , [\oid7] , [\oid5]]]))))))
lemma perm-\sigma_1: \sigma_1 = (state.make ((\Map.empty (oid2 ⇒ (in\mathbf{Person} (\sigma1-object4_{\mathbf{Person}}))) (oid7 ⇒ (in\mathbf{Person} (\sigma1-object4_{\mathbf{Person}}))) (oid1 ⇒ (in\mathbf{Person} (\mathbf{X}\mathbf{Person}1_{\mathbf{Person}})))))\mathbf{X}\mathbf{Person}2_{\mathbf{Person}})))
(\proof
end

83
locale state-$\sigma_1'$ =
fixes oid1 :: nat
fixes oid2 :: nat
fixes oid3 :: nat
assumes distinct-oid: (distinct ([oid1, oid2, oid3]))
fixes $\text{Person}$ :: $\text{ty}$
fixes $\text{Person}$1 :: $\text{Person}$
assumes $\text{Person}$1-def: $\text{Person}$1 = (\lambda $\text{Person}$-oid1 :: $\text{ty}$)
fixes $\text{Person}$2 :: $\text{Person}$
fixes $\text{Person}$-oid2 :: $\text{ty}$
assumes $\text{Person}$2-def: $\text{Person}$2 = (\lambda $\text{Person}$-oid2 :: $\text{ty}$)
fixes $\text{Person}$3 :: $\text{Person}$
fixes $\text{Person}$-oid3 :: $\text{ty}$
assumes $\text{Person}$3-def: $\text{Person}$3 = (\lambda $\text{Person}$-oid3 :: $\text{ty}$)
begin
definition $\sigma_1'$ = (state.make ((\text{Map}.empty (oid1 \mapsto (\text{in} \text{Person} (\text{Person}1)))) (oid2 \mapsto (\text{in} \text{Person} (\text{Person}2)))) (oid3 \mapsto (\text{in} \text{Person} (\text{Person}3)))) ((\text{map-of-list} (\text{oid} \text{Person}1 \mapsto \text{boss}, (\text{List}.\text{map} ((\text{x}, \text{y}) \mapsto (\text{switch} \text{2-01} ((\text{oid}1, \text{oid}2)))))())))
lemma perm-$\sigma_1'$ : $\sigma_1'$ = (state.make ((\text{Map}.empty (oid3 \mapsto (\text{in} \text{Person} (\text{Person}3)))) (oid2 \mapsto (\text{in} \text{Person} (\text{Person}2)))) (oid1 \mapsto (\text{in} \text{Person} (\text{Person}1)))) ((\text{assocs} ($\sigma_1'$)))
(proof)
end

locale pre-post-$\sigma_1$-$\sigma_1'$ =
fixes oid1 :: nat
fixes oid2 :: nat
fixes oid3 :: nat
fixes oid4 :: nat
fixes oid5 :: nat
fixes oid6 :: nat
fixes oid7 :: nat
assumes distinct-oid: (distinct ([oid1, oid2, oid3, oid4, oid5, oid6, oid7]))
fixes $\text{Person}$ :: $\text{ty}$
fixes $\text{Person}$1 :: $\text{Person}$
assumes $\text{Person}$1-def: $\text{Person}$1 = (\lambda $\text{Person}$-oid1 :: $\text{ty}$)
fixes $\text{Person}$2 :: $\text{Person}$
fixes $\text{Person}$-oid2 :: $\text{ty}$
assumes $\text{Person}$2-def: $\text{Person}$2 = (\lambda $\text{Person}$-oid2 :: $\text{ty}$)
fixes $\text{Person}$3 :: $\text{Person}$
fixes $\text{Person}$-oid3 :: $\text{ty}$
assumes $\text{Person}$3-def: $\text{Person}$3 = (\lambda $\text{Person}$-oid3 :: $\text{ty}$)
fixes $\sigma$1-object0 :: $\text{ty}$
fixes $\sigma$1-object0 :: $\text{Person}$
assumes $\sigma$1-object0-def: $\sigma$1-object0 = (\lambda $\sigma$1-object0 :: $\text{ty}$)
fixes $\sigma$1-object1 :: $\text{ty}$
fixes $\sigma$1-object1 :: $\text{Person}$
assumes \(\sigma_1\)-object1-def: \(\sigma_1\)-object1 = (\(\lambda\). [\(\sigma_1\)-object1 Person])
fixes \(\sigma_1\)-object2 Person :: tyPerson
fixes \(\sigma_1\)-object2 :: \(\cdot\) Person
assumes \(\sigma_1\)-object2-def: \(\sigma_1\)-object2 = (\(\lambda\). [\(\sigma_1\)-object2 Person])
fixes \(\sigma_1\)-object4 Person :: tyPerson
fixes \(\sigma_1\)-object4 :: \(\cdot\) Person
assumes \(\sigma_1\)-object4-def: \(\sigma_1\)-object4 = (\(\lambda\). [\(\sigma_1\)-object4 Person])

assumes \(\sigma_1\): (state-\(\sigma_1\) (oid4) (oid5) (oid6) (oid7) (oid2) (\(\sigma_1\)-object0 Person) (\(\sigma_1\)-object0) (\(\sigma_1\)-object1 Person) (\(\sigma_1\)-object1) (\(\sigma_1\)-object2 Person) (\(\sigma_1\)-object2) (\(\cdot\) Person1) (\(\cdot\) Person1) (\(\sigma_1\)-object4 Person) (\(\sigma_1\)-object4) (\(\cdot\) Person2) (\(\cdot\) Person2))

assumes \(\sigma_1\)’: (state-\(\sigma_1\)’ (oid1) (oid2) (oid3) (\(\cdot\) Person1) (\(\cdot\) Person1) (\(\cdot\) Person2) (\(\cdot\) Person2) (\(\cdot\) Person3) (\(\cdot\) Person3))

begin

interpretation state-\(\sigma_1\): state-\(\sigma_1\) oid4 oid5 oid6 oid1 oid7 oid2 \(\sigma_1\)-object0 Person \(\sigma_1\)-object0
\(\sigma_1\)-object1 Person \(\sigma_1\)-object1 \(\sigma_1\)-object2 Person \(\sigma_1\)-object2 \(\cdot\) Person1 \(\cdot\) Person1 \(\sigma_1\)-object4 Person \(\sigma_1\)-object4 \(\cdot\) Person2 \(\cdot\) Person2
(proof)

interpretation state-\(\sigma_1\)’: state-\(\sigma_1\)’ oid1 oid2 oid3 \(\cdot\) Person1 \(\cdot\) Person1 \(\cdot\) Person2 \(\cdot\) Person2 \(\cdot\) Person3 \(\cdot\) Person3
(proof)
end