

Shivers' Control Flow Analysis

Joachim Breitner

September 23, 2021

Abstract

In his dissertation [3], Olin Shivers introduces a concept of control flow graphs for functional languages, provides an algorithm to statically derive a safe approximation of the control flow graph and proves this algorithm correct. In this research project [1], Shivers' algorithms and proofs are formalized using the HOLCF extension of the logic HOL in the theorem prover Isabelle.

Contents

I. The definitions	2
1. Syntax	2
2. Standard semantics	4
3. Exact nonstandard semantics	7
4. Abstract nonstandard semantics	13
II. The main results	17
5. The exact call cache is a map	17
5.1. Preparations	17
5.2. The proof	20
6. The abstract semantics is correct	25
6.1. Abstraction functions	25
6.2. Lemmas about abstraction functions	27
6.3. Approximation relation	27
6.4. Lemmas about the approximation relation	28

6.5. Lemma 7	30
6.6. Lemmas 8 and 9	30
7. Generic Computability	38
7.1. Non-branching case	39
7.2. Branching case	40
8. The abstract semantics is computable	42
8.1. Towards finiteness	44
8.2. A decomposition	49
8.3. The iterative equation	50
III. The auxiliary theories	50
9. Syntax tree helpers	50
10. General utility lemmas	57
11. Set-valued maps	58
12. Sets of maps	60
13. HOLCF Utility lemmas	62
14. Fixed point transformations	67

Part I.

The definitions

1. Syntax

```
theory CPSScheme
  imports Main
begin
```

First, we define the syntax tree of a program in our toy functional language, using continuation passing style, corresponding to section 3.2 in Shivers' dissertation.

We assume that the program to be investigated is already parsed into a syntax tree. Furthermore, we assume that distinct labels were added to distinguish different code positions and that the program has been alphanised, i.e. that each variable name is only

bound once. This binding position is, as a convenience, considered part of the variable name.

type-synonym *label* = *nat*
type-synonym *var* = *label* × *string*

definition *binder* :: *var* ⇒ *label* **where** [*simp*]: *binder* *v* = *fst* *v*

The syntax consists now of lambda abstractions, call expressions and values, which can either be lambdas, variable references, constants or primitive operations. A program is a lambda expression.

Shivers' language has as the set of basic values integers plus a special value for *false*. We simplified this to just the set of integers. The conditional *If* considers zero as false and any other number as true.

Shivers also restricts the values in a call expression: No constant maybe be used as the called value, and no primitive operation may occur as an argument. This restriction is dropped here and just leads to runtime errors when evaluating the program.

datatype *prim* = *Plus* *label* | *If* *label* *label*
datatype *lambda* = *Lambda* *label* *var* *list* *call*
and *call* = *App* *label* *val* *val* *list*
| *Let* *label* (*var* × *lambda*) *list* *call*
and *val* = *L* *lambda* | *R* *label* *var* | *C* *label* *int* | *P* *prim*

datatype-compat *lambda* *call* *val*

type-synonym *prog* = *lambda*

lemmas *mutual-lambda-call-var-inducts* =
compat-lambda.induct
compat-call.induct
compat-val.induct
compat-val-list.induct
compat-nat-char-list-prod-lambda-prod-list.induct
compat-nat-char-list-prod-lambda-prod.induct

Three example programs. These were generated using the Haskell implementation of Shivers' algorithm that we wrote as a prototype[2].

abbreviation *ex1* == (*Lambda* 1 [(1,"cont'')] (*App* 2 (*R* 3 (1,"cont'')) [(*C* 4 0)]))
abbreviation *ex2* == (*Lambda* 1 [(1,"cont'')] (*App* 2 (*P* (*Plus* 3)) [(*C* 4 1), (*C* 5 1), (*R* 6 (1,"cont''))]))
abbreviation *ex3* == (*Lambda* 1 [(1,"cont'')] (*Let* 2 [((2,"rec''),(*Lambda* 3 [(3,"p''), (3,"i''), (3,"c'')] (*App* 4 (*P* (*If* 5 6)) [(*R* 7 (3,"i''), (*L* (*Lambda* 8 [] (*App* 9 (*P* (*Plus* 10)) [(*R* 11 (3,"p''), (*R* 12 (3,"i''), (*L* (*Lambda* 13 [(13,"p-')] (*App* 14 (*P* (*Plus* 15)) [(*R* 16 (3,"i''), (*C* 17 (- 1)), (*L* (*Lambda* 18 [(18,"i-')] (*App* 19 (*R* 20 (2,"rec'')) [(*R* 21 (13,"p-')), (*R*

```

22 (18, "i-"), (R 23 (3, "c-"))]]]]]]]])), (L (Lambda 24 [] (App 25 (R 26 (3, "c-")) [(R 27
(3, "p'"))]]]]]))] (App 28 (R 29 (2, "rec'")) [(C 30 0), (C 31 10), (R 32 (1, "cont'"))]))

```

end

2. Standard semantics

```

theory Eval
  imports HOLCF HOLCFUtils CPSScheme
begin

```

We begin by giving the standard semantics for our language. Although this is not actually used to show any results, it is helpful to see that the later algorithms “look similar” to the evaluation code and the relation between calls done during evaluation and calls recorded by the control flow graph.

We follow the definition in Figure 3.1 and 3.2 of Shivers’ dissertation, with the clarifications from Section 4.1. As explained previously, our set of values encompasses just the integers, there is no separate value for *false*. Also, values and procedures are not distinguished by the type system.

Due to recursion, one variable can have more than one currently valid binding, and due to closures all bindings can possibly be accessed. A simple call stack is therefore not sufficient. Instead we have a *contour counter*, which is increased in each evaluation step. It can also be thought of as a time counter. The variable environment maps tuples of variables and contour counter to values, thus allowing a variable to have more than one active binding. A contour environment lists the currently visible binding for each binding position and is preserved when a lambda expression is turned into a closure.

```

type-synonym contour = nat
type-synonym benv = label  $\rightarrow$  contour
type-synonym closure = lambda  $\times$  benv

```

The set of semantic values consist of the integers, closures, primitive operations and a special value *Stop*. This is passed as an argument to the program and represents the terminal continuation. When this value occurs in the first position of a call, the program terminates.

```

datatype d = DI int
           | DC closure
           | DP prim
           | Stop

type-synonym venv = var  $\times$  contour  $\rightarrow$  d

```

The function \mathcal{A} evaluates a syntactic value into a semantic datum. Constants and primitive operations are left untouched. Variable references are resolved in two stages: First the current binding contour is fetched from the binding environment β , then the stored value is fetched from the variable environment ve . A lambda expression is bundled with the current contour environment to form a closure.

```

fun evalV :: val  $\Rightarrow$  benv  $\Rightarrow$  venv  $\Rightarrow$  d (A)
  where A (C - i)  $\beta$  ve = DI i
    | A (P prim)  $\beta$  ve = DP prim
    | A (R - var)  $\beta$  ve =
      (case  $\beta$  (binder var) of
        Some l  $\Rightarrow$  (case ve (var,l) of Some d  $\Rightarrow$  d))
    | A (L lam)  $\beta$  ve = DC (lam,  $\beta$ )

```

The answer domain of our semantics is the set of integers, lifted to obtain an additional element denoting bottom. Shivers distinguishes runtime errors from non-termination. Here, both are represented by \perp .

type-synonym ans = int lift

To be able to do case analysis on the custom datatypes *lambda*, *d*, *call* and *prim* inside a function defined with *fixrec*, we need continuity results for them. These are all of the same shape and proven by case analysis on the discriminator.

lemma cont2cont-case-lambda [simp, cont2cont]:

assumes $\bigwedge a b c. cont (\lambda x. f x a b c)$

shows cont $(\lambda x. case\text{-}lambda (f x) l)$

using *assms*

by (cases l) *auto*

lemma cont2cont-case-d [simp, cont2cont]:

assumes $\bigwedge y. cont (\lambda x. f1 x y)$

and $\bigwedge y. cont (\lambda x. f2 x y)$

and $\bigwedge y. cont (\lambda x. f3 x y)$

and cont $(\lambda x. f4 x)$

shows cont $(\lambda x. case\text{-}d (f1 x) (f2 x) (f3 x) (f4 x) d)$

using *assms*

by (cases d) *auto*

lemma cont2cont-case-call [simp, cont2cont]:

assumes $\bigwedge a b c. cont (\lambda x. f1 x a b c)$

and $\bigwedge a b c. cont (\lambda x. f2 x a b c)$

shows cont $(\lambda x. case\text{-}call (f1 x) (f2 x) c)$

using *assms*

by (cases c) *auto*

lemma cont2cont-case-prim [simp, cont2cont]:

assumes $\bigwedge y. cont (\lambda x. f1 x y)$

and $\bigwedge y z. cont (\lambda x. f2 x y z)$

```

shows cont ( $\lambda x. \text{case-prim } (f1\ x) (f2\ x) p$ )
using assms
by (cases p) auto

```

As usual, the semantics of a functional language is given as a denotational semantics. To that end, two functions are defined here: \mathcal{F} applies a procedure to a list of arguments. Here closures are unwrapped, the primitive operations are implemented and the terminal continuation *Stop* is handled. \mathcal{C} evaluates a call expression, either by evaluating procedure and arguments and passing them to \mathcal{F} , or by adding the bindings of a *Let* expression to the environment.

Note how the contour counter is incremented before each call to \mathcal{F} or when a *Let* expression is evaluated.

With mutually recursive equations, such as those given here, the existence of a function satisfying these is not obvious. Therefore, the *fixrec* command from the *HOLCF* package is used. This takes a set of equations and builds a functional from that. It mechanically proves that this functional is continuous and thus a least fixed point exists. This is then used to define \mathcal{F} and \mathcal{C} and proof the equations given here. To use the *HOLCF* setup, the continuous function arrow \rightarrow with application operator \cdot is used and our types are wrapped in *discr* and *lift* to indicate which partial order is to be used.

```

type-synonym fstate = (d  $\times$  d list  $\times$  venv  $\times$  contour)
type-synonym cstate = (call  $\times$  benv  $\times$  venv  $\times$  contour)

```

```

fixrec evalF :: fstate discr  $\rightarrow$  ans ( $\mathcal{F}$ )
and evalC :: cstate discr  $\rightarrow$  ans ( $\mathcal{C}$ )
where evalF.fstate = (case undiscr fstate of
  (DC (Lambda lab vs c,  $\beta$ ), as, ve, b)  $\Rightarrow$ 
    (if length vs = length as
      then let  $\beta' = \beta$  (lab  $\mapsto$  b);
            ve' = map-upds ve (map ( $\lambda v.(v,b)$ ) vs) as
            in  $\mathcal{C} \cdot$  (Discr (c, $\beta'$ ,ve',b))
      else  $\perp$ )
  | (DP (Plus c),[DI a1, DI a2, cnt],ve,b)  $\Rightarrow$ 
    let b' = Suc b;
         $\beta = [c \mapsto b]$ 
    in  $\mathcal{F} \cdot$  (Discr (cnt,[DI (a1 + a2)],ve,b'))
  | (DP (prim.If ct cf),[DI v, contt, contf],ve,b)  $\Rightarrow$ 
    (if v  $\neq$  0
      then let b' = Suc b;
             $\beta = [ct \mapsto b]$ 
        in  $\mathcal{F} \cdot$  (Discr (contt,[],ve,b'))
      else let b' = Suc b;
             $\beta = [cf \mapsto b]$ 
        in  $\mathcal{F} \cdot$  (Discr (contf,[],ve,b')))

```

```

    | (Stop,[DI i],-,) ⇒ Def i
    | - ⇒ ⊥
  )
| C.cstate = (case undiscr cstate of
  (App lab f vs,β,ve,b) ⇒
    let f' = A f β ve;
        as = map (λv. A v β ve) vs;
        b' = Suc b
    in F.(Discr (f',as,ve,b'))
| (Let lab ls c',β,ve,b) ⇒
  let b' = Suc b;
      β' = β (lab ↦ b');
      ve' = ve ++ map-of (map (λ(v,l). ((v,b'), A (L l) β' ve)) ls)
  in C.(Discr (c',β',ve',b'))
)

```

To evaluate a full program, it is passed to \mathcal{F} with proper initializations of the other arguments. We test our semantics function against two example programs and observe that the expected value is returned.

definition *evalCPS* :: prog ⇒ ans (PR)
where PR l = (let ve = Map.empty;
 β = Map.empty;
 f = A (L l) β ve
 in F.(Discr (f,[Stop],ve,0)))

lemma *correct-ex1*: PR ex1 = Def 0
unfolding *evalCPS-def*
by *simp*

lemma *correct-ex2*: PR ex2 = Def 2
unfolding *evalCPS-def*
by *simp*

end

3. Exact nonstandard semantics

theory *ExCF*
imports *HOLCF HOLCFUtils CPSScheme Utils*
begin

We now alter the standard semantics given in the previous section to calculate a control flow graph instead of the return value. At this point, we still “run” the program in full, so this is not yet the static analysis that we aim for. Instead, this is the reference for

the correctness proof of the static analysis: If an edge is recorded here, we expect it to be found by the static analysis as well.

In preparation of the correctness proof we change the type of the contour counters. Instead of plain natural numbers as in the previous sections we use lists of labels, remembering at each step which part of the program was just evaluated.

Note that for the exact semantics, this information is not used in any way and it would have been possible to just use natural numbers again. This is reflected by the preorder instance for the contours which only look at the length of the list, but not the entries.

definition *contour* = (*UNIV::label list set*)

typedef *contour* = *contour*
unfolding *contour-def* **by** *auto*

definition *initial-contour* (*b₀*)
where *b₀* = *Abs-contour* []

definition *nb*
where *nb b c* = *Abs-contour* (*c # Rep-contour b*)

instantiation *contour* :: *preorder*

begin

definition *le-contour-def*: $b \leq b' \longleftrightarrow \text{length } (\text{Rep-contour } b) \leq \text{length } (\text{Rep-contour } b')$

definition *less-contour-def*: $b < b' \longleftrightarrow \text{length } (\text{Rep-contour } b) < \text{length } (\text{Rep-contour } b')$

instance proof

qed(*auto simp add:le-contour-def less-contour-def Rep-contour-inverse Abs-contour-inverse contour-def*)

end

Three simple lemmas helping Isabelle to automatically prove statements about contour numbers.

lemma *nb-le-less[iff]*: $nb\ b\ c \leq b' \longleftrightarrow b < b'$

unfolding *nb-def*

by (*auto simp add:le-contour-def less-contour-def Rep-contour-inverse Abs-contour-inverse contour-def*)

lemma *nb-less[iff]*: $b' < nb\ b\ c \longleftrightarrow b' \leq b$

unfolding *nb-def*

by (*auto simp add:le-contour-def less-contour-def Rep-contour-inverse Abs-contour-inverse contour-def*)

declare *less-imp-le*[**where** '*a* = *contour*, *intro*]

The other types used in our semantics functions have not changed.

type-synonym *benv* = *label* \rightarrow *contour*
type-synonym *closure* = *lambda* \times *benv*

datatype *d* = *DI int*
| *DC closure*
| *DP prim*
| *Stop*

type-synonym *venv* = *var* \times *contour* \rightarrow *d*

As we do not use the type system to distinguish procedural from non-procedural values, we define a predicate for that.

primrec *isProc*
where *isProc* (*DI* -) = *False*
| *isProc* (*DC* -) = *True*
| *isProc* (*DP* -) = *True*
| *isProc* *Stop* = *True*

To please *HOLCF*, we declare the discrete partial order for our types:

instantiation *contour* :: *discrete-cpo*

begin

definition [*simp*]: (*x::contour*) \sqsubseteq *y* \longleftrightarrow *x* = *y*

instance by *standard simp*

end

instantiation *d* :: *discrete-cpo* **begin**

definition [*simp*]: (*x::d*) \sqsubseteq *y* \longleftrightarrow *x* = *y*

instance by *standard simp*

end

instantiation *call* :: *discrete-cpo* **begin**

definition [*simp*]: (*x::call*) \sqsubseteq *y* \longleftrightarrow *x* = *y*

instance by *standard simp*

end

The evaluation function for values has only changed slightly: To avoid worrying about incorrect programs, we return zero when a variable lookup fails. If the labels in the program given are correct, this will not happen. Shivers makes this explicit in Section 4.1.3 by restricting the function domains to the valid programs. This is omitted here.

fun *evalV* :: *val* \Rightarrow *benv* \Rightarrow *venv* \Rightarrow *d* (*A*)

where *A* (*C* - *i*) β *ve* = *DI i*

| *A* (*P* *prim*) β *ve* = *DP prim*

| *A* (*R* - *var*) β *ve* =

(*case* β (*binder* *var*) *of*

Some l \Rightarrow (*case* *ve* (*var*,*l*) *of* *Some d* \Rightarrow *d* | *None* \Rightarrow *DI 0*)

 | *None* \Rightarrow *DI 0*)

| $\mathcal{A} (L \text{ lam}) \beta \text{ ve} = DC (\text{lam}, \beta)$

To be able to do case analysis on the custom datatypes *lambda*, *d*, *call* and *prim* inside a function defined with *fixrec*, we need continuity results for them. These are all of the same shape and proven by case analysis on the discriminator.

lemma *cont2cont-case-lambda* [*simp*, *cont2cont*]:

assumes $\bigwedge a \ b \ c. \text{cont} (\lambda x. f \ x \ a \ b \ c)$

shows $\text{cont} (\lambda x. \text{case-lambda} (f \ x) \ l)$

using *assms*

by (*cases l*) *auto*

lemma *cont2cont-case-d* [*simp*, *cont2cont*]:

assumes $\bigwedge y. \text{cont} (\lambda x. f1 \ x \ y)$

and $\bigwedge y. \text{cont} (\lambda x. f2 \ x \ y)$

and $\bigwedge y. \text{cont} (\lambda x. f3 \ x \ y)$

and $\text{cont} (\lambda x. f4 \ x)$

shows $\text{cont} (\lambda x. \text{case-d} (f1 \ x) (f2 \ x) (f3 \ x) (f4 \ x) \ d)$

using *assms*

by (*cases d*) *auto*

lemma *cont2cont-case-call* [*simp*, *cont2cont*]:

assumes $\bigwedge a \ b \ c. \text{cont} (\lambda x. f1 \ x \ a \ b \ c)$

and $\bigwedge a \ b \ c. \text{cont} (\lambda x. f2 \ x \ a \ b \ c)$

shows $\text{cont} (\lambda x. \text{case-call} (f1 \ x) (f2 \ x) \ c)$

using *assms*

by (*cases c*) *auto*

lemma *cont2cont-case-prim* [*simp*, *cont2cont*]:

assumes $\bigwedge y. \text{cont} (\lambda x. f1 \ x \ y)$

and $\bigwedge y \ z. \text{cont} (\lambda x. f2 \ x \ y \ z)$

shows $\text{cont} (\lambda x. \text{case-prim} (f1 \ x) (f2 \ x) \ p)$

using *assms*

by (*cases p*) *auto*

Now, our answer domain is not any more the integers, but rather call caches. These are represented as sets containing tuples of call sites (given by their label) and binding environments to the called value. The argument types are unaltered.

In the functions \mathcal{F} and \mathcal{C} , upon every call, a new element is added to the resulting set. The *STOP* continuation now ignores its argument and returns the empty set instead. This corresponds to Figure 4.2 and 4.3 in Shivers' dissertation.

type-synonym *ccache* = $((\text{label} \times \text{benv}) \times d) \text{ set}$

type-synonym *ans* = *ccache*

type-synonym *fstate* = $(d \times d \text{ list} \times \text{venv} \times \text{contour})$

type-synonym *cstate* = $(\text{call} \times \text{benv} \times \text{venv} \times \text{contour})$

```

fixrec evalF :: fstate discr → ans (F)
and evalC :: cstate discr → ans (C)
where F.fstate = (case undiscr fstate of
  (DC (Lambda lab vs c, β), as, ve, b) ⇒
    (if length vs = length as
      then let β' = β (lab ↦ b);
            ve' = map-upds ve (map (λv.(v,b)) vs) as
            in C.(Discr (c,β',ve',b))
      else ⊥)
  | (DP (Plus c),[DI a1, DI a2, cnt],ve,b) ⇒
    (if isProc cnt
      then let b' = nb b c;
            β = [c ↦ b]
            in F.(Discr (cnt,[DI (a1 + a2)],ve,b'))
            ∪ {((c, β),cnt)}
      else ⊥)
  | (DP (prim.If ct cf),[DI v, contt, contf],ve,b) ⇒
    (if isProc contt ∧ isProc contf
      then
        (if v ≠ 0
          then let b' = nb b ct;
                β = [ct ↦ b]
                in (F.(Discr (contt,[],ve,b'))
                  ∪ {((ct, β),contt)})
          else let b' = nb b cf;
                β = [cf ↦ b]
                in (F.(Discr (contf,[],ve,b'))
                  ∪ {((cf, β),contf)})
        else ⊥)
    | (Stop,[DI i],-,-) ⇒ {}
    | - ⇒ ⊥
  )
  | C.cstate = (case undiscr cstate of
    (App lab f vs,β,ve,b) ⇒
      let f' = A f β ve;
          as = map (λv. A v β ve) vs;
          b' = nb b lab
          in if isProc f'
            then F.(Discr (f',as,ve,b')) ∪ {((lab, β),f')}
            else ⊥
    | (Let lab ls c',β,ve,b) ⇒
      let b' = nb b lab;
          β' = β (lab ↦ b');
          ve' = ve ++ map-of (map (λ(v,l). ((v,b'), A (L l) β' ve)) ls)
          in C.(Discr (c',β',ve',b'))
    )

```

In preparation of later proofs, we give the cases of the generated induction rule names and also create a large rule to deconstruct the an value of type *fstate* into the various cases that were used in the definition of \mathcal{F} .

lemmas *evalF-evalC-induct* = *evalF-evalC.induct*[*case-names Admissibility Bottom Next*]

lemmas *cl-cases* = *prod.exhaust*[*OF lambda.exhaust, of - λ a - . a*]

lemmas *ds-cases-plus* = *list.exhaust*[

OF - d.exhaust, of - - λ a - . a,
OF - list.exhaust, of - - λ- x - . x,
OF - - d.exhaust, of - - λ- - - a - . a,
OF - - list.exhaust, of - - λ- - - - x - . x,
OF - - - list.exhaust, of - - λ- - - - - x . x

]

lemmas *ds-cases-if* = *list.exhaust*[*OF - d.exhaust, of - - λ a - . a,*

OF - list.exhaust[*OF - list.exhaust*[*OF - list.exhaust, of - - λ- x . x*], *of - - λ- x . x*], *of - - λ- x - . x*]

lemmas *ds-cases-stop* = *list.exhaust*[*OF - d.exhaust, of - - λ a - . a,*

OF - list.exhaust, of - - λ- x - . x]

lemmas *fstate-case* = *prod-cases4*[*OF d.exhaust, of - λ x - - - . x,*

OF - cl-cases prim.exhaust, of - - λ - - - - a . a λ - - - - a . a,

OF - case-split ds-cases-plus ds-cases-if ds-cases-stop,

of - - λ- as - - - - - vs - . length vs = length as λ - ds - - - - . ds λ - ds - - - - . ds λ - ds - - . ds,

case-names x Closure x x x Plus x x x x x x x x x If-True If-False x x x x x Stop x x x x x]

The exact semantics of a program again uses \mathcal{F} with properly initialized arguments. For the first two examples, we see that the function works as expected.

definition *evalCPS* :: *prog* \Rightarrow *ans* (\mathcal{PR})

where \mathcal{PR} *l* = (*let* *ve* = *Map.empty*;
 β = *Map.empty*;
f = \mathcal{A} (*L l*) β *ve*
in $\mathcal{F}.$ (*Discr* (*f*, [*Stop*], *ve*, *b₀*)))

lemma *correct-ex1*: \mathcal{PR} *ex1* = {((2, [1 \mapsto *b₀*]), *Stop*)}

unfolding *evalCPS-def*

by *simp*

lemma *correct-ex2*: \mathcal{PR} *ex2* = {((2, [1 \mapsto *b₀*]), *DP* (*Plus* 3)),

((3, [3 \mapsto *nb b₀* 2]), *Stop*)}

unfolding *evalCPS-def*

by *simp*

end

4. Abstract nonstandard semantics

```
theory AbsCF
  imports HOLCF HOLCFUtils CPSScheme Utils SetMap
begin
```

```
default-sort type
```

After having defined the exact meaning of a control graph, we now alter the algorithm into a statically computable. We note that the contour pointer in the exact semantics is taken from an infinite set. This is unavoidable, as recursion depth is unbounded. But if this were not the case and the set were finite, the function would be calculable, having finite range and domain.

Therefore, we make the set of contour counter values finite and accept that this makes our result less exact, but calculable. We also do not work with values any more but only remember, for each variable, what possible lambdas can occur there. Because we do not have exact values any more, in a conditional expression, both branches are taken.

We want to leave the exact choice of the finite contour set open for now. Therefore, we define a type class capturing the relevant definitions and the fact that the set is finite. Isabelle expects type classes to be non-empty, so we show that the *unit* type is in this type class.

```
class contour = finite +
  fixes nb-a :: 'a ⇒ label ⇒ 'a (nb̂)
  and a-initial-contour :: 'a (b̂₀)
```

```
instantiation unit :: contour
```

```
begin
```

```
definition nb̂ - - = ()
```

```
definition b̂₀ = ()
```

```
instance by standard auto
```

```
end
```

Analogous to the previous section, we define types for binding environments, closures, procedures, semantic values (which are now sets of possible procedures) and variable environment. Their types are parametrized by the chosen set of abstract contours.

The abstract variable environment is a partial map to sets in Shivers' dissertation. As he does not need to distinguish between a key not in the map and a key mapped to the empty set, this presentation is redundant. Therefore, I encoded this as a function from keys to sets of values. The theory *Shivers-CFA.SetMap* contains functions and lemmas to work with such maps, symbolized by an appended dot (e.g. $\{\}., \cup.$).

```
type-synonym 'c a-benv = label → 'c (- b̂env [1000])
```

type-synonym 'c a-closure = lambda × 'c \widehat{benv} (- $\widehat{closure}$ [1000])

datatype 'c \widehat{proc} (- \widehat{proc} [1000])
 = PC 'c $\widehat{closure}$
 | PP prim
 | AStop

type-synonym 'c a-d = 'c \widehat{proc} set (- \widehat{d} [1000])

type-synonym 'c a-venv = var × 'c ⇒ 'c \widehat{d} (- \widehat{venv} [1000])

The evaluation function now ignores constants and returns singletons for primitive operations and lambda expressions.

fun evalV-a :: val ⇒ 'c \widehat{benv} ⇒ 'c \widehat{venv} ⇒ 'c \widehat{d} (\widehat{A})
where \widehat{A} (C - i) β ve = {}
 | \widehat{A} (P prim) β ve = {PP prim}
 | \widehat{A} (R - var) β ve =
 (case β (binder var) of
 Some l ⇒ ve (var, l)
 None ⇒ {})
 | \widehat{A} (L lam) β ve = {PC (lam, β)}

The types of the calculated graph, the arguments to \widehat{F} and \widehat{C} resemble closely the types in the exact case, with each type replaced by its abstract counterpart.

type-synonym 'c a-ccache = ((label × 'c \widehat{benv}) × 'c \widehat{proc}) set (- \widehat{ccache} [1000])

type-synonym 'c a-ans = 'c \widehat{ccache} (- \widehat{ans} [1000])

type-synonym 'c a-fstate = ('c \widehat{proc} × 'c \widehat{d} list × 'c \widehat{venv} × 'c) (- \widehat{fstate} [1000])

type-synonym 'c a-cstate = (call × 'c \widehat{benv} × 'c \widehat{venv} × 'c) (- \widehat{cstate} [1000])

And yet again, cont2cont results need to be shown for our custom data types.

lemma cont2cont-case-lambda [simp, cont2cont]:

assumes $\bigwedge a b c. cont (\lambda x. f x a b c)$

shows cont ($\lambda x. case-lambda (f x) l$)

using assms

by (cases l) auto

lemma cont2cont-case-proc [simp, cont2cont]:

assumes $\bigwedge y. cont (\lambda x. f1 x y)$

and $\bigwedge y. cont (\lambda x. f2 x y)$

and cont ($\lambda x. f3 x$)

shows cont ($\lambda x. case-proc (f1 x) (f2 x) (f3 x) d$)

using assms

by (cases d) auto

lemma *cont2cont-case-call* [*simp*, *cont2cont*]:

assumes $\bigwedge a b c. \text{cont } (\lambda x. f1 x a b c)$
and $\bigwedge a b c. \text{cont } (\lambda x. f2 x a b c)$
shows $\text{cont } (\lambda x. \text{case-call } (f1 x) (f2 x) c)$

using *assms*

by (*cases c*) *auto*

lemma *cont2cont-case-prim* [*simp*, *cont2cont*]:

assumes $\bigwedge y. \text{cont } (\lambda x. f1 x y)$
and $\bigwedge y z. \text{cont } (\lambda x. f2 x y z)$
shows $\text{cont } (\lambda x. \text{case-prim } (f1 x) (f2 x) p)$

using *assms*

by (*cases p*) *auto*

We can now define the abstract nonstandard semantics, based on the equations in Figure 4.5 and 4.6 of Shivers' dissertation. In the *AStop* case, $\{\}$ is returned, while for wrong arguments, \perp is returned. Both actually represent the same value, the empty set, so this is just a aesthetic difference.

fixrec *a-evalF* :: 'c::contour \widehat{fstate} *discr* \rightarrow 'c \widehat{ans} ($\widehat{\mathcal{F}}$)
and *a-evalC* :: 'c::contour \widehat{cstate} *discr* \rightarrow 'c \widehat{ans} ($\widehat{\mathcal{C}}$)
where $\widehat{\mathcal{F}}.fstate = (\text{case } \widehat{undiscr} \text{ } fstate \text{ of}$
 $(PC \ (Lambda \ lab \ vs \ c, \ \beta), \ as, \ ve, \ b) \Rightarrow$
 $(\text{if } length \ vs = length \ as$
 $\text{then let } \beta' = \beta \ (lab \mapsto \ b);$
 $\quad ve' = ve \cup. (\bigcup. (\text{map } (\lambda(v,a). \{(v,b) := a\}.) \ (zip \ vs \ as)))$
 $\quad \text{in } \widehat{\mathcal{C}}.(Discr \ (c, \beta', ve', b))$
 $\text{else } \perp)$
 $| (PP \ (Plus \ c), [-, -, cnts], ve, b) \Rightarrow$
 $\text{let } b' = \widehat{nb} \ b \ c;$
 $\quad \beta = [c \mapsto b]$
 $\text{in } (\bigcup_{cnt \in cnts}. \widehat{\mathcal{F}}.(Discr \ (cnt, [\{\}], ve, b'))$
 $\quad \cup$
 $\quad \{((c, \beta), cont) \mid cont . cont \in cnts\}$
 $| (PP \ (prim.If \ ct \ cf), [-, cntts, cntfs], ve, b) \Rightarrow$
 $((\text{let } b' = \widehat{nb} \ b \ ct;$
 $\quad \beta = [ct \mapsto b]$
 $\text{in } (\bigcup_{cnt \in cntts}. \widehat{\mathcal{F}}.(Discr \ (cnt, [], ve, b'))$
 $\quad \cup \{((ct, \beta), cnt) \mid cnt . cnt \in cntts\}$
 $) \cup$
 $(\text{let } b' = \widehat{nb} \ b \ cf;$
 $\quad \beta = [cf \mapsto b]$
 $\text{in } (\bigcup_{cnt \in cntfs}. \widehat{\mathcal{F}}.(Discr \ (cnt, [], ve, b'))$
 $\quad \cup \{((cf, \beta), cnt) \mid cnt . cnt \in cntfs\}$
 $)$
 $| (AStop, [-, -, -]) \Rightarrow \{\}$
 $| - \Rightarrow \perp$

```

)
|  $\widehat{C}$ .cstate = (case undiscr cstate of
  (App lab f vs, $\beta$ ,ve,b)  $\Rightarrow$ 
    let fs =  $\widehat{A}$  f  $\beta$  ve;
        as = map ( $\lambda v$ .  $\widehat{A}$  v  $\beta$  ve) vs;
        b' =  $\widehat{nb}$  b lab
    in ( $\bigcup f' \in fs$ .  $\widehat{F}$ .(Discr (f',as,ve,b')))
         $\cup \{((lab, \beta), f') \mid f' \in fs\}$ )
| (Let lab ls c', $\beta$ ,ve,b)  $\Rightarrow$ 
  let b' =  $\widehat{nb}$  b lab;
       $\beta' = \beta$  (lab  $\mapsto$  b');
      ve' = ve  $\cup$ . ( $\bigcup$ . (map ( $\lambda(v,l)$ .  $\{(v,b') := (\widehat{A} (L l) \beta' ve)\}$ .) ls))
  in  $\widehat{C}$ .(Discr (c', $\beta'$ ,ve',b'))
)

```

Again, we name the cases of the induction rule and build a nicer case analysis rule for arguments of type \widehat{fstate} .

lemmas *a-evalF-evalC-induct* = *a-evalF-a-evalC.induct*[case-names *Admissibility Bottom Next*]

fun *a-evalF-cases*

```

where a-evalF-cases (PC (Lambda lab vs c,  $\beta$ )) as ve b = undefined
| a-evalF-cases (PP (Plus cp)) [a1, a2, cnt] ve b = undefined
| a-evalF-cases (PP (prim.If cp1 cp2)) [v,cntt,cntf] ve b = undefined
| a-evalF-cases AStop [v] ve b = undefined

```

lemmas *a-fstate-case-x* = *a-evalF-cases.cases*[
OF case-split, of - λ - vs - - as - - . length vs = length as,
case-names *Closure Closure-inv Plus If Stop*]

lemmas *a-cl-cases* = *prod.exhaust*[*OF lambda.exhaust*, of - λ a - . a]

lemmas *a-ds-cases* = *list.exhaust*[
OF list.exhaust, of - - λ - x. x,
OF - - list.exhaust ,of - - λ - - - x. x ,
OF - - - list.exhaust,of - - λ - - - - x. x
]

lemmas *a-ds-cases-stop* = *list.exhaust*[*OF list.exhaust*, of - - λ - x. x]

lemmas *a-fstate-case* = *prod-cases4*[*OF proc.exhaust*, of - λ x - - - . x,
OF a-cl-cases prim.exhaust, of - λ - - - - a . a - λ - - - - a. a,
OF case-split a-ds-cases a-ds-cases a-ds-cases-stop,
of - λ - as - - - - - vs - . length vs = length as - λ - ds - - - - . ds λ - ds - - - - . ds λ - ds - - .
ds]

Not surprisingly, the abstract semantics of a whole program is defined using \widehat{F} with suitably initialized arguments. The function *the-elm* extracts a value from a singleton set. This works because we know that \widehat{A} returns such a set when given a lambda expression.


```

definition evalCPS-a :: prog ⇒ ('c::contour)  $\widehat{ans}$  ( $\widehat{\mathcal{PR}}$ )
  where  $\widehat{\mathcal{PR}} l = (let\ ve = \{\};$ 
            $\beta = Map.empty;$ 
            $f = \widehat{A} (L l) \beta ve$ 
           in  $\widehat{\mathcal{F}}.(Discr (the\ elem\ f, [\{AStop\}], ve, \widehat{b}_0)))$ 
end

```

Part II.

The main results

5. The exact call cache is a map

```

theory ExCFSV
imports ExCF
begin

```

5.1. Preparations

Before we state the main result of this section, we need to define

- the set of binding environments occurring in a semantic value (which exists only if it is a closure),
- the set of binding environments in a variable environment, using the previous definition,
- the set of contour counters occurring in a semantic value and
- the set of contour counters occurring in a variable environment.

```

fun benv-in-d :: d ⇒ benv set
  where benv-in-d (DC (l,β)) = {β}
         | benv-in-d - = {}

```

```

definition benv-in-ve :: venv ⇒ benv set
  where benv-in-ve ve =  $\bigcup \{benv-in-d\ d \mid d . d \in ran\ ve\}$ 

```

```

fun contours-in-d :: d ⇒ contour set
  where contours-in-d (DC (l,β)) = ran β
         | contours-in-d - = {}

```

```

definition contours-in-ve :: venv ⇒ contour set

```

where *contours-in-ve* $ve = \bigcup \{ \text{contours-in-d } d \mid d . d \in \text{ran } ve \}$

The following 6 lemmas allow us to calculate the above definition, when applied to constructs used in our semantics function, e.g. map updates, empty maps etc.

lemma *benv-in-ve-upds*:

assumes *eq-length*: $\text{length } vs = \text{length } ds$

and $\forall \beta \in \text{benv-in-ve } ve. Q \beta$

and $\forall d' \in \text{set } ds. \forall \beta \in \text{benv-in-d } d'. Q \beta$

shows $\forall \beta \in \text{benv-in-ve } (ve(\text{map } (\lambda v. (v, b'')) vs [\mapsto] ds)). Q \beta$

proof

fix β

assume *ass*: $\beta \in \text{benv-in-ve } (ve(\text{map } (\lambda v. (v, b'')) vs [\mapsto] ds))$

then obtain d **where** $\beta \in \text{benv-in-d } d$ **and** $d \in \text{ran } (ve(\text{map } (\lambda v. (v, b'')) vs [\mapsto] ds))$ **unfolding** *benv-in-ve-def* **by** *auto*

moreover have $\text{ran } (ve(\text{map } (\lambda v. (v, b'')) vs [\mapsto] ds)) \subseteq \text{ran } ve \cup \text{set } ds$ **using** *eq-length* **by** (*auto intro! :ran-upds*)

ultimately

have $d \in \text{ran } ve \vee d \in \text{set } ds$ **by** *auto*

thus $Q \beta$ **using** *assms(2,3)* $\langle \beta \in \text{benv-in-d } d \rangle$ **unfolding** *benv-in-ve-def* **by** *auto*

qed

lemma *benv-in-eval*:

assumes $\forall \beta' \in \text{benv-in-ve } ve. Q \beta'$

and $Q \beta$

shows $\forall \beta \in \text{benv-in-d } (A v \beta ve). Q \beta$

proof (*cases v*)

case (*R - var*)

thus *?thesis*

proof (*cases* β (*fst var*))

case *None* **with** *R* **show** *?thesis* **by** *simp next*

case (*Some cnt*) **show** *?thesis*

proof (*cases ve* (*var,cnt*))

case *None* **with** *Some R* **show** *?thesis* **by** *simp next*

case (*Some d*)

hence $d \in \text{ran } ve$ **unfolding** *ran-def* **by** *blast*

thus *?thesis* **using** *Some* $\langle \beta$ (*fst var*) = *Some cnt* \rangle *R* *assms(1)*

unfolding *benv-in-ve-def* **by** *auto*

qed

qed next

case (*L l*) **thus** *?thesis* **using** *assms(2)* **by** *simp next*

case *C* **thus** *?thesis* **by** *simp next*

case *P* **thus** *?thesis* **by** *simp*

qed

lemma *contours-in-ve-empty[simp]*: $\text{contours-in-ve } \text{Map.empty} = \{\}$

unfolding *contours-in-ve-def* **by** *auto*

lemma *contours-in-ve-upds*:

assumes *eq-length*: $\text{length } vs = \text{length } ds$
and $\forall b' \in \text{contours-in-ve } ve. Q b'$
and $\forall d' \in \text{set } ds. \forall b' \in \text{contours-in-d } d'. Q b'$
shows $\forall b' \in \text{contours-in-ve } (ve(\text{map } (\lambda v. (v, b'')) vs [\mapsto] ds)). Q b'$
proof–
have $\text{ran } (ve(\text{map } (\lambda v. (v, b'')) vs [\mapsto] ds)) \subseteq \text{ran } ve \cup \text{set } ds$ **using** *eq-length* **by**(*auto intro!:ran-upds*)
thus *?thesis* **using** *assms(2,3)* **unfolding** *contours-in-ve-def* **by** *blast*
qed

lemma *contours-in-ve-upds-binds*:
assumes $\forall b' \in \text{contours-in-ve } ve. Q b'$
and $\forall b' \in \text{ran } \beta'. Q b'$
shows $\forall b' \in \text{contours-in-ve } (ve ++ \text{map-of } (\text{map } (\lambda(v,l). ((v,b''), \mathcal{A} (L l) \beta' ve)) ls)). Q b'$
proof
fix *b'* **assume** $b' \in \text{contours-in-ve } (ve ++ \text{map-of } (\text{map } (\lambda(v,l). ((v,b''), \mathcal{A} (L l) \beta' ve)) ls))$
then obtain *d* **where** $d: d \in \text{ran } (ve ++ \text{map-of } (\text{map } (\lambda(v,l). ((v,b''), \mathcal{A} (L l) \beta' ve)) ls))$
and $b': b' \in \text{contours-in-d } d$ **unfolding** *contours-in-ve-def* **by** *auto*

have $\text{ran } (ve ++ \text{map-of } (\text{map } (\lambda(v,l). ((v,b''), \mathcal{A} (L l) \beta' ve)) ls)) \subseteq \text{ran } ve \cup \text{ran } (\text{map-of } (\text{map } (\lambda(v,l). ((v,b''), \mathcal{A} (L l) \beta' ve)) ls))$
by(*auto intro!:ran-concat*)
also
have $\dots \subseteq \text{ran } ve \cup \text{snd } \langle \text{set } (\text{map } (\lambda(v,l). ((v,b''), \mathcal{A} (L l) \beta' ve)) ls) \rangle$
by (*rule Un-mono[of ran ve ran ve, OF subset-refl ran-map-of]*)
also
have $\dots \subseteq \text{ran } ve \cup \text{set } (\text{map } (\lambda(v,l). (\mathcal{A} (L l) \beta' ve)) ls)$
by (*rule Un-mono[of ran ve ran ve, OF subset-refl]*)*auto*
finally
have $d \in \text{ran } ve \cup \text{set } (\text{map } (\lambda(v,l). (\mathcal{A} (L l) \beta' ve)) ls)$ **using** *d* **by** *auto*
thus $Q b'$ **using** *assms b* **unfolding** *contours-in-ve-def* **by** *auto*
qed

lemma *contours-in-eval*:
assumes $\forall b' \in \text{contours-in-ve } ve. Q b'$
and $\forall b' \in \text{ran } \beta. Q b'$
shows $\forall b' \in \text{contours-in-d } (\mathcal{A} f \beta ve). Q b'$
unfolding *contours-in-ve-def*
proof(*cases f*)
case (*R - var*)
thus *?thesis*
proof (*cases* β (*fst var*))
case *None* **with** *R* **show** *?thesis* **by** *simp next*
case (*Some cnt*) **show** *?thesis*
proof (*cases ve* (*var,cnt*))
case *None* **with** *Some R* **show** *?thesis* **by** *simp next*
case (*Some d*)
hence $d \in \text{ran } ve$ **unfolding** *ran-def* **by** *blast*
thus *?thesis* **using** *Some* $\langle \beta$ (*fst var*) = *Some cnt* $\rangle R \langle \forall b' \in \text{contours-in-ve } ve. Q b' \rangle$

```

    unfolding contours-in-ve-def
  by auto
qed
qed next
case (L l) thus ?thesis using ⟨∀ b' ∈ ran β. Q b'⟩ by simp next
case C thus ?thesis by simp next
case P thus ?thesis by simp
qed

```

5.2. The proof

The set returned by \mathcal{F} and \mathcal{C} is actually a partial map from callsite/binding environment pairs to called values. The corresponding predicate in Isabelle is *single-valued*.

We would like to show an auxiliary result about the contour counter passed to \mathcal{F} and \mathcal{C} (such that it is an unused counter when passed to \mathcal{F} and others) first. Unfortunately, this is not possible with induction proofs over fixed points: While proving the inductive case, one does not show results for the function in question, but for an information-theoretical approximation. Thus, any previously shown results are not available. We therefore intertwine the two inductions in one large proof.

This is a proof by fixpoint induction, so we have are obliged to show that the predicate is admissible and that it holds for the base case, i.e. the empty set. For the proof of admissibility, *HOLCF* provides a number of introduction lemmas that, together with some additions in *Shivers-CFA.HOLCFUtils* and the continuity lemmas, mechanically prove admissibility. The base case is trivial.

The remaining case is the preservation of the properties when applying the recursive equations to a function known to have have the desired property. Here, we break the proof into the various cases that occur in the definitions of \mathcal{F} and \mathcal{C} and use the induction hypotheses.

lemma *cc-single-valued'*:

```

  [[ ∀ b' ∈ contours-in-ve ve. b' < b
    ; ∀ b' ∈ contours-in-d d. b' < b
    ; ∀ d' ∈ set ds. ∀ b' ∈ contours-in-d d'. b' < b
  ]]
  ⇒
  ( single-valued (F·(Discr (d,ds,ve,b)))
    ∧ (∀ ((lab,β),t) ∈ F·(Discr (d,ds,ve, b)). ∃ b'. b' ∈ ran β ∧ b ≤ b')
  )
and [[ b ∈ ran β'
    ; ∀ b' ∈ ran β'. b' ≤ b
    ; ∀ b' ∈ contours-in-ve ve. b' ≤ b
  ]]
  ⇒

```

```

      ( single-valued (C.(Discr (c,β',ve,b)))
        ∧ (∀ ((lab,β),t) ∈ C.(Discr (c,β',ve,b)). ∃ b'. b' ∈ ran β ∧ b ≤ b')
      )
proof(induct arbitrary:d ds ve b c β' rule:evalF-evalC-induct)
case Admissibility show ?case
  by (intro adm-lemmas adm-ball' adm-prod-split adm-not-conj adm-not-mem adm-single-valued
cont2cont)
next
  case Bottom {
    case 1 thus ?case by auto next
    case 2 thus ?case by auto
  }
next
  case (Next evalF evalC)

```

Nicer names for the hypotheses:

```

note hyps-F-sv = Next.hyps(1)[THEN conjunct1]
note hyps-F-b = Next.hyps(1)[THEN conjunct2, THEN bspec]
note hyps-C-sv = Next.hyps(2)[THEN conjunct1]
note hyps-C-b = Next.hyps(2)[THEN conjunct2, THEN bspec]
{
case (1 d ds ve b)
thus ?case
proof (cases (d,ds,ve,b) rule:fstate-case, auto simp del:Un-insert-left Un-insert-right)

```

Case Closure

```

fix lab' and vs :: var list and c and β' :: benv
assume prem-d: ∀ b' ∈ ran β'. b' < b
assume eq-length: length vs = length ds
have new: b ∈ ran (β'(lab' ↦ b)) by simp

have b-dom-beta: ∀ b' ∈ ran (β'(lab' ↦ b)). b' ≤ b
proof fix b' assume b' ∈ ran (β'(lab' ↦ b))
  hence b' ∈ ran β' ∨ b' ≤ b by (auto dest:ran-upd[THEN subsetD])
  thus b' ≤ b using prem-d by auto
qed
from contours-in-ve-upds[OF eq-length 1.prem(1) 1.prem(3)]
have b-dom-ve: ∀ b' ∈ contours-in-ve (ve(map (λv. (v, b)) vs [↦] ds)). b' ≤ b
  by auto

show single-valued (evalC.(Discr (c, β'(lab' ↦ b), ve(map (λv. (v, b)) vs [↦] ds), b))
  by (rule hyps-C-sv[OF new b-dom-beta b-dom-ve, of c])

fix lab and β and t
assume ((lab, β), t) ∈ evalC.(Discr(c, β'(lab' ↦ b), ve(map (λv. (v, b)) vs [↦] ds), b)
thus ∃ b'. b' ∈ ran β ∧ b ≤ b'
  by (auto dest: hyps-C-b[OF new b-dom-beta b-dom-ve])

```

next

Case Plus

```

fix cp and i1 and i2 and cnt
assume  $\forall b' \in \text{contours-in-d cnt}. b' < b$ 
hence b-dom-d:  $\forall b' \in \text{contours-in-d cnt}. b' < \text{nb } b \text{ cp}$  by auto
have b-dom-ds:  $\forall d' \in \text{set } [DI (i1+i2)]. \forall b' \in \text{contours-in-d } d'. b' < \text{nb } b \text{ cp}$  by auto
have b-dom-ve:  $\forall b' \in \text{contours-in-ve ve}. b' < \text{nb } b \text{ cp}$  using 1.prem(1) by auto
{
  fix t
  assume  $((cp, [cp \mapsto b]), t) \in \text{evalF} \cdot (\text{Discr } (cnt, [DI (i1 + i2)], ve, \text{nb } b \text{ cp}))$ 
  hence False by  $(\text{auto dest:hypos-F-b}[OF \text{ b-dom-ve b-dom-d b-dom-ds}])$ 
}
with hypos-F-sv[OF b-dom-ve b-dom-d b-dom-ds]
show single-valued  $((\text{evalF} \cdot (\text{Discr } (cnt, [DI (i1 + i2)], ve, \text{nb } b \text{ cp})))$ 
   $\cup \{((cp, [cp \mapsto b]), cnt)\})$ 
by  $(\text{auto intro:single-valued-insert})$ 

fix lab β t
assume  $((lab, \beta), t) \in \text{evalF} \cdot (\text{Discr } (cnt, [DI (i1 + i2)], ve, \text{nb } b \text{ cp}))$ 
thus  $\exists b'. b' \in \text{ran } \beta \wedge b \leq b'$ 
by  $(\text{auto dest:hypos-F-b}[OF \text{ b-dom-ve b-dom-d b-dom-ds}])$ 
next

```

Case If (true branch)

```

fix cp1 cp2 i cntt cntf
assume  $\forall b' \in \text{contours-in-d cntt}. b' < b$ 
hence b-dom-d:  $\forall b' \in \text{contours-in-d cntt}. b' < \text{nb } b \text{ cp1}$  by auto
have b-dom-ds:  $\forall d' \in \text{set } []. \forall b' \in \text{contours-in-d } d'. b' < \text{nb } b \text{ cp1}$  by auto
have b-dom-ve:  $\forall b' \in \text{contours-in-ve ve}. b' < \text{nb } b \text{ cp1}$  using 1.prem(1) by auto
{
  fix t
  assume  $((cp1, [cp1 \mapsto b]), t) \in \text{evalF} \cdot (\text{Discr } (cntt, [], ve, \text{nb } b \text{ cp1}))$ 
  hence False by  $(\text{auto dest:hypos-F-b}[OF \text{ b-dom-ve b-dom-d b-dom-ds}])$ 
}
with Next.hypos(1)[OF b-dom-ve b-dom-d b-dom-ds, THEN conjunct1]
show single-valued  $((\text{evalF} \cdot (\text{Discr } (cntt, [], ve, \text{nb } b \text{ cp1})))$ 
   $\cup \{((cp1, [cp1 \mapsto b]), cntt)\})$ 
by  $(\text{auto intro:single-valued-insert})$ 

fix lab β t
assume  $((lab, \beta), t) \in \text{evalF} \cdot (\text{Discr } (cntt, [], ve, \text{nb } b \text{ cp1}))$ 
thus  $\exists b'. b' \in \text{ran } \beta \wedge b \leq b'$ 
by  $(\text{auto dest:hypos-F-b}[OF \text{ b-dom-ve b-dom-d b-dom-ds}])$ 
next

```

Case If (false branch). Variable names swapped for easier code reuse.

```

fix cp2 cp1 i cntf cntt
assume  $\forall b' \in \text{contours-in-d cntt}. b' < b$ 
hence b-dom-d:  $\forall b' \in \text{contours-in-d cntt}. b' < \text{nb } b \text{ cp1}$  by auto
have b-dom-ds:  $\forall d' \in \text{set } []. \forall b' \in \text{contours-in-d } d'. b' < \text{nb } b \text{ cp1}$  by auto
have b-dom-ve:  $\forall b' \in \text{contours-in-ve ve}. b' < \text{nb } b \text{ cp1}$  using 1.prem1 by auto
{
  fix t
  assume  $((cp1, [cp1 \mapsto b]), t) \in \text{evalF} \cdot (\text{Discr } (cntt, [], ve, \text{nb } b \text{ cp1}))$ 
  hence False by (auto dest:hyps-F-b[OF b-dom-ve b-dom-d b-dom-ds])
}
with Next.hyps(1)[OF b-dom-ve b-dom-d b-dom-ds, THEN conjunct1]
show single-valued  $((\text{evalF} \cdot (\text{Discr } (cntt, [], ve, \text{nb } b \text{ cp1})))$ 
   $\cup \{((cp1, [cp1 \mapsto b]), cntt)\})$ 
by (auto intro:single-valued-insert)

fix lab  $\beta$  t
assume  $((lab, \beta), t) \in \text{evalF} \cdot (\text{Discr } (cntt, [], ve, \text{nb } b \text{ cp1}))$ 
thus  $\exists b'. b' \in \text{ran } \beta \wedge b \leq b'$ 
by (auto dest:hyps-F-b[OF b-dom-ve b-dom-d b-dom-ds])
qed
next
case (2 ve b c  $\beta'$ )
thus ?case
proof (cases c, auto simp add:HOL.Let-def simp del:Un-insert-left Un-insert-right evalV.simps)

```

Case App

```

fix lab' f vs

have prem2':  $\forall b' \in \text{ran } \beta'. b' < \text{nb } b \text{ lab'}$  using 2.prem1 by auto
have prem3':  $\forall b' \in \text{contours-in-ve ve}. b' < \text{nb } b \text{ lab'}$  using 2.prem2 by auto
note c-in-e = contours-in-eval[OF prem3' prem2']

have b-dom-d:  $\forall b' \in \text{contours-in-d } (evalV f \beta' ve). b' < \text{nb } b \text{ lab'}$  by (rule c-in-e)
have b-dom-ds:  $\forall d' \in \text{set } (map (\lambda v. evalV v \beta' ve) vs). \forall b' \in \text{contours-in-d } d'. b' < \text{nb } b$ 
lab'
using c-in-e by auto
have b-dom-ve:  $\forall b' \in \text{contours-in-ve ve}. b' < \text{nb } b \text{ lab'}$  by (rule prem3')

have  $\forall y. ((lab', \beta'), y) \notin \text{evalF} \cdot (\text{Discr } (evalV f \beta' ve, map (\lambda v. evalV v \beta' ve) vs, ve, \text{nb } b$ 
lab'))
proof(rule allI, rule notI)
fix y assume  $((lab', \beta'), y) \in \text{evalF} \cdot (\text{Discr } (evalV f \beta' ve, map (\lambda v. evalV v \beta' ve) vs, ve,$ 
nb b lab'))
hence  $\exists b'. b' \in \text{ran } \beta' \wedge \text{nb } b \text{ lab'} \leq b'$ 
by (auto dest:hyps-F-b[OF b-dom-ve b-dom-d b-dom-ds])
thus False using prem2' by (auto iff:less-le-not-le)
qed

```

```

with hyps-F-sv[OF b-dom-ve b-dom-d b-dom-ds]
show single-valued (evalF·(Discr (evalV f  $\beta'$  ve, map ( $\lambda v.$  evalV v  $\beta'$  ve) vs, ve, nb b lab'))
     $\cup \{((lab', \beta'), evalV f \beta' ve)\}$ 
by (auto intro:single-valued-insert)

fix lab  $\beta$  t
assume  $((lab, \beta), t) \in (evalF \cdot (Discr (evalV f \beta' ve, map (\lambda v. evalV v \beta' ve) vs, ve, nb b lab'))) lab')$ 
thus  $\exists b'. b' \in ran \beta \wedge b \leq b'$ 
by (auto dest:hyps-F-b[OF b-dom-ve b-dom-d b-dom-ds])
next

```

Case Let

```

fix lab' ls c'
have prem2':  $\forall b' \in ran (\beta'(lab' \mapsto nb b lab')). b' \leq nb b lab'$ 
proof
  fix b' assume  $b' \in ran (\beta'(lab' \mapsto nb b lab'))$ 
  hence  $b' \in ran \beta' \vee b' = nb b lab'$  by (auto dest:ran-upd[THEN subsetD])
  thus  $b' \leq nb b lab'$  using 2.prem2(2) by auto
qed
have prem3':  $\forall b' \in contours-in-ve ve. b' \leq nb b lab'$  using 2.prem2(3)
by auto

note c-in-e = contours-in-eval[OF prem3' prem2']
note c-in-ve' = contours-in-ve-upds-binds[OF prem3' prem2']

have b-dom-ve:  $\forall b' \in contours-in-ve (ve ++ map-of (map (\lambda(v,l). ((v, nb b lab'), evalV (L l) ((\beta'(lab' \mapsto nb b lab')) ve)) ls)). b' \leq nb b lab'$ 
by (rule c-in-ve')
have b-dom-beta:  $\forall b' \in ran (\beta'(lab' \mapsto nb b lab')). b' \leq nb b lab'$  by (rule prem2')
have new:  $nb b lab' \in ran (\beta'(lab' \mapsto nb b lab'))$  by simp

from hyps-C-sv[OF new b-dom-beta b-dom-ve, of c']
show single-valued (evalC·(Discr (c',  $\beta'(lab' \mapsto nb b lab')$ ,
  ve ++ map-of (map ( $\lambda(v, l).$   $((v, nb b lab'), evalV (L l) (\beta'(lab' \mapsto nb b lab')) ve))$  ls),
  nb b lab')).

fix lab  $\beta$  t
assume  $((lab, \beta), t) \in evalC \cdot (Discr (c', \beta'(lab' \mapsto nb b lab'),$ 
  ve ++ map-of (map ( $\lambda(v, l).$   $((v, nb b lab'), \mathcal{A} (L l) (\beta'(lab' \mapsto nb b lab')) ve))$  ls),
  nb b lab')
thus  $\exists b'. b' \in ran \beta \wedge b \leq b'$ 
by  $-(drule hyps-C-b$ [OF new b-dom-beta b-dom-ve], auto)
qed
}
qed

```

lemma *single-valued* (*PR prog*)


```

unfolding evalCPS-def
by ((subst HOL.Let-def)+, rule cc-single-valued'[THEN conjunct1], auto)
end

```

6. The abstract semantics is correct

```

theory AbsCFCorrect
  imports AbsCF ExCF HOL-Library.Adhoc-Overloading
begin

default-sort type

```

The intention of the abstract semantics is to safely approximate the real control flow. This means that every call recorded by the exact semantics must occur in the result provided by the abstract semantics, which in turn is allowed to predict more calls than actually done.

6.1. Abstraction functions

This relation is expressed by abstraction functions and approximation relations. For each of our data types, there is an abstraction function $abs-<type>$, mapping the a value from the exact setup to the corresponding value in the abstract view. The approximation relation then expresses the fact that one abstract value of such a type is safely approximated by another.

Because we need an abstraction function for contours, we extend the *contour* type class by the abstraction functions and two equations involving the nb and b_0 symbols.

```

class contour-a = contour +
  fixes abs-cnt :: contour  $\Rightarrow$  'a
  assumes abs-cnt-nb[simp]: abs-cnt (nb b lab) =  $\widehat{nb}$  (abs-cnt b) lab
  and abs-cnt-initial[simp]: abs-cnt(b0) =  $\widehat{b}_0$ 

instantiation unit :: contour-a
begin
definition abs-cnt - = ()
instance by standard auto
end

```

It would be unwieldly to always write out $abs-<type> x$. We would rather like to write $|x|$ if the type of x is known, as Shivers does it as well. Isabelle allows one to use the same syntax for different symbols. In that case, it generates more than one parse tree and picks the (hopefully unique) tree that typechecks.

Unfortunately, this does not work well in our case: There are eight *abs-<type>* functions and some expressions later have multiple occurrences of these, causing an exponential blow-up of combinations.

Therefore, we use a module by Christian Sternagel and Alexander Krauss for ad-hoc overloading, where the choice of the concrete function is done at parse time and immediately. This is used in the following to set up the the symbol $|-$ for the family of abstraction functions.

consts *abs* :: 'a \Rightarrow 'b ($|-$)

ad hoc-overloading

abs abs-cnt

definition *abs-benv* :: *benv* \Rightarrow 'c::contour-a $\widehat{\text{benv}}$
where *abs-benv* $\beta = \text{map-option } \text{abs-cnt} \circ \beta$

ad hoc-overloading

abs abs-benv

primrec *abs-closure* :: *closure* \Rightarrow 'c::contour-a $\widehat{\text{closure}}$
where *abs-closure* $(l, \beta) = (l, |\beta|)$

ad hoc-overloading

abs abs-closure

primrec *abs-d* :: *d* \Rightarrow 'c::contour-a \widehat{d}
where *abs-d* $(DI\ i) = \{\}$
 $| \text{abs-d } (DP\ p) = \{PP\ p\}$
 $| \text{abs-d } (DC\ cl) = \{PC\ |cl|\}$
 $| \text{abs-d } (Stop) = \{AStop\}$

ad hoc-overloading

abs abs-d

definition *abs-venv* :: *venv* \Rightarrow 'c::contour-a $\widehat{\text{venv}}$
where *abs-venv* $ve = (\lambda(v, b-a). \bigcup \{(case\ ve\ (v, b)\ of\ Some\ d \Rightarrow |d| \mid None \Rightarrow \{\}) \mid b. |b| = b-a\})$

ad hoc-overloading

abs abs-venv

definition *abs-ccache* :: *ccache* \Rightarrow 'c::contour-a $\widehat{\text{ccache}}$
where *abs-ccache* $cc = (\bigcup ((c, \beta), d) \in cc . \{(c, \text{abs-benv } \beta), p \mid p . p \in \text{abs-d } d\})$

ad hoc-overloading

abs abs-ccache

```

fun abs-fstate :: fstate  $\Rightarrow$  'c::contour-a  $\widehat{fstate}$ 
  where abs-fstate (d,ds,ve,b) = (the-elem |d|, map abs-d ds, |ve|, |b| )

```

adhoc-overloading

abs abs-fstate

```

fun abs-cstate :: cstate  $\Rightarrow$  'c::contour-a  $\widehat{cstate}$ 
  where abs-cstate (c, $\beta$ ,ve,b) = (c, | $\beta$ |, |ve|, |b| )

```

adhoc-overloading

abs abs-cstate

6.2. Lemmas about abstraction functions

Some results about the abstractions functions.

```

lemma abs-benv-empty[simp]: |Map.empty| = Map.empty
unfolding abs-benv-def by simp

```

```

lemma abs-benv-upd[simp]: | $\beta(c \mapsto b)$ | = | $\beta$ | (c  $\mapsto$  |b| )
unfolding abs-benv-def by simp

```

lemma the-elem-is-Proc:

assumes isProc cnt

shows the-elem |cnt| \in |cnt|

using assms **by** (cases cnt)auto

```

lemma [simp]: |{|}| = {|}| unfolding abs-ccache-def by auto

```

```

lemma abs-cache-singleton [simp]: |{|((c, $\beta$ ),d)}| = {|((c, | $\beta$ | ), p) |p. p  $\in$  |d|}|
unfolding abs-ccache-def by simp

```

```

lemma abs-venv-empty[simp]: |Map.empty| = {|}|.
apply (rule ext) by (auto simp add: abs-venv-def smap-empty-def)

```

6.3. Approximation relation

The family of relations defined here capture the notion of safe approximation.

```

consts approx :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (-  $\lesssim$  -)

```

```

definition venv-approx :: 'c  $\widehat{venv} \Rightarrow$  'c  $\widehat{venv} \Rightarrow$  bool
  where venv-approx = smap-less

```

adhoc-overloading

approx venv-approx

```

definition ccache-approx :: 'c  $\widehat{ccache} \Rightarrow$  'c  $\widehat{ccache} \Rightarrow$  bool

```

where $ccache\text{-}approx = less\text{-}eq$

ad hoc overloading

$approx\ ccache\text{-}approx$

definition $d\text{-}approx :: 'c\ \widehat{d} \Rightarrow 'c\ \widehat{d} \Rightarrow bool$

where $d\text{-}approx = less\text{-}eq$

ad hoc overloading

$approx\ d\text{-}approx$

definition $ds\text{-}approx :: 'c\ \widehat{d}\ list \Rightarrow 'c\ \widehat{d}\ list \Rightarrow bool$

where $ds\text{-}approx = list\text{-}all2\ d\text{-}approx$

ad hoc overloading

$approx\ ds\text{-}approx$

inductive $fstate\text{-}approx :: 'c\ \widehat{fstate} \Rightarrow 'c\ \widehat{fstate} \Rightarrow bool$

where $\llbracket ve \lesssim ve' ; ds \lesssim ds' \rrbracket$

$\implies fstate\text{-}approx\ (proc, ds, ve, b)\ (proc, ds', ve', b)$

ad hoc overloading

$approx\ fstate\text{-}approx$

inductive $cstate\text{-}approx :: 'c\ \widehat{cstate} \Rightarrow 'c\ \widehat{cstate} \Rightarrow bool$

where $\llbracket ve \lesssim ve' \rrbracket \implies cstate\text{-}approx\ (c, \beta, ve, b)\ (c, \beta, ve', b)$

ad hoc overloading

$approx\ cstate\text{-}approx$

6.4. Lemmas about the approximation relation

Most of the following lemmas reduce an approximation statement about larger structures, as they are occurring the semantics functions, to statements about the components.

lemma $venv\text{-}approx\text{-}trans[trans]$:

fixes $ve1\ ve2\ ve3 :: 'c\ \widehat{venv}$

shows $\llbracket ve1 \lesssim ve2 ; ve2 \lesssim ve3 \rrbracket \implies (ve1 \lesssim ve3)$

unfolding $venv\text{-}approx\text{-}def$ **by** $(rule\ smap\text{-}less\text{-}trans)$

lemma $abs\text{-}venv\text{-}union$: $|ve1 ++ ve2| \lesssim |ve1| \cup |ve2|$

by $(auto\ simp\ add: venv\text{-}approx\text{-}def\ smap\text{-}less\text{-}def\ abs\text{-}venv\text{-}def\ smap\text{-}union\text{-}def, split\ option.\ split\text{-}asm, auto)$

lemma $abs\text{-}venv\text{-}map\text{-}of\text{-}rev$: $|map\text{-}of\ (rev\ l)| \lesssim \bigcup. (map\ (\lambda(v,k). \llbracket v \mapsto k \rrbracket))\ l$

proof $(induct\ l)$

case Nil **show** $?case$ **unfolding** $abs\text{-}venv\text{-}def$ **by** $(auto\ simp: venv\text{-}approx\text{-}def\ smap\text{-}less\text{-}def)$

next

case (*Cons a l*)
obtain $v\ k$ **where** $a=(v,k)$ **by** (*rule prod.exhaust*)
hence $|map-of\ (rev\ (a\#l))| \lesssim (|[v \mapsto k]| \cup |map-of\ (rev\ l)|) :: 'a\ \widehat{venv}$
by (*auto intro: abs-venv-union*)
also
have $\dots \lesssim |[v \mapsto k]| \cup (\bigcup. (map\ (\lambda(v,k). |[v \mapsto k]|)\ l))$
by (*auto intro!: smap-union-mono[OF smap-less-refl Cons[unfolded venv-approx-def]] simp: venv-approx-def*)
also
have $\dots = \bigcup. (|[v \mapsto k]| \# map\ (\lambda(v,k). |[v \mapsto k]|)\ l)$
by (*rule smap-Union-union*)
also
have $\dots = \bigcup. (map\ (\lambda(v,k). |[v \mapsto k]|)\ (a\#l))$
using $\langle a = (v,k) \rangle$
by *auto*
finally
show $?case$.
qed

lemma *abs-venv-map-of*: $|map-of\ l| \lesssim \bigcup. (map\ (\lambda(v,k). |[v \mapsto k]|)\ l)$
using *abs-venv-map-of-rev[of rev l]* **by** *simp*

lemma *abs-venv-singleton*: $|[(v,b) \mapsto d]| = \{(v,|b|) := |d|\}$.
by (*rule ext, auto simp add: abs-venv-def smap-singleton-def smap-empty-def*)

lemma *ccache-approx-empty[simp]*:
fixes $x :: 'c\ \widehat{ccache}$
shows $\{\} \lesssim x$
unfolding *ccache-approx-def* **by** *simp*

lemmas *ccache-approx-trans[trans]* = *subset-trans[where 'a = ((label × 'c \widehat{benv}) × 'c \widehat{proc}), folded ccache-approx-def]*

lemmas *Un-mono-approx* = *Un-mono[where 'a = ((label × 'c \widehat{benv}) × 'c \widehat{proc}), folded ccache-approx-def]*

lemmas *Un-upper1-approx* = *Un-upper1[where 'a = ((label × 'c \widehat{benv}) × 'c \widehat{proc}), folded ccache-approx-def]*

lemmas *Un-upper2-approx* = *Un-upper2[where 'a = ((label × 'c \widehat{benv}) × 'c \widehat{proc}), folded ccache-approx-def]*

lemma *abs-ccache-union*: $|c1 \cup c2| \lesssim |c1| \cup |c2|$
unfolding *ccache-approx-def abs-ccache-def* **by** *auto*

lemma *d-approx-empty[simp]*: $\{\} \lesssim (d :: 'c\ \widehat{d})$
unfolding *d-approx-def* **by** *simp*

lemma *ds-approx-empty[simp]*: $\square \lesssim \square$
unfolding *ds-approx-def* **by** *simp*

6.5. Lemma 7

Shivers' lemma 7 says that $\widehat{\mathcal{A}}$ safely approximates \mathcal{A} .

```

lemma lemma7:
  assumes |ve::venv|  $\lesssim$  ve-a
  shows | $\mathcal{A}$  f  $\beta$  ve|  $\lesssim$   $\widehat{\mathcal{A}}$  f | $\beta$ | ve-a
proof(cases f)
case (R - v)
  from assms have assm':  $\bigwedge v b.$  case-option {} abs-d (ve (v,b))  $\lesssim$  ve-a (v,|b| )
  by (auto simp add:d-approx-def abs-venv-def venv-approx-def smap-less-def elim!:allE)
  show ?thesis
  proof(cases  $\beta$  (binder v))
  case None thus ?thesis using R by auto next
  case (Some b)
    thus ?thesis using R assm'[of v b]
    by (auto simp add:abs-benv-def split:option.split)
  qed
qed (auto simp add:d-approx-def)

```

6.6. Lemmas 8 and 9

The main goal of this section is to show that $\widehat{\mathcal{F}}$ safely approximates \mathcal{F} and that $\widehat{\mathcal{C}}$ safely approximates \mathcal{C} . This has to be shown at once, as the functions are mutually recursive and requires a fixed point induction. To that end, we have to augment the set of continuity lemmas.

```

lemma cont2cont-abs-ccache[cont2cont,simp]:
  assumes cont f
  shows cont ( $\lambda x.$  abs-ccache(f x))
unfolding abs-ccache-def
using assms
by (rule cont2cont)(rule cont-const)

```

Shivers proves these lemmas using parallel fixed point induction over the two fixed points (the one from the exact semantics and the one from the abstract semantics). But it is simpler and equivalent to just do induction over the exact semantics and keep the abstract semantics functions fixed, so this is what I am doing.

```

lemma lemma89:
  fixes fstate-a :: 'c::contour-a  $\widehat{fstate}$  and cstate-a :: 'c::contour-a  $\widehat{cstate}$ 
  shows |fstate|  $\lesssim$  fstate-a  $\implies$  | $\mathcal{F}$ .(Discr fstate)|  $\lesssim$   $\widehat{\mathcal{F}}$ .(Discr fstate-a)
  and |cstate|  $\lesssim$  cstate-a  $\implies$  | $\mathcal{C}$ .(Discr cstate)|  $\lesssim$   $\widehat{\mathcal{C}}$ .(Discr cstate-a)
proof(induct arbitrary: fstate fstate-a cstate cstate-a rule: evalF-evalC-induct)
case Admissibility show ?case
  unfolding ccache-approx-def
  by (intro adm-lemmas adm-subset adm-prod-split adm-not-conj adm-not-mem adm-single-valued
cont2cont)

```

```

next
case Bottom {
  case 1 show ?case by simp next
  case 2 show ?case by simp next
}
next
case (Next evalF evalC) {
case 1
  obtain d ds ve b where fstate: fstate = (d,ds,ve,b)
  by (cases fstate, auto)
  moreover
  obtain proc ds-a ve-a b-a where fstate-a: fstate-a = (proc,ds-a,ve-a,b-a)
  by (cases fstate-a, auto)
  ultimately
  have abs-d: the-elem |d| = proc
  and abs-ds: map abs-d ds  $\lesssim$  ds-a
  and abs-ve: |ve|  $\lesssim$  ve-a
  and abs-b: |b| = b-a
  using 1 by (auto elim:fstate-approx.cases)

  from abs-ds have dslength: length ds = length ds-a
  by (auto simp add:ds-approx-def dest!:list-all2-lengthD)

  from fstate fstate-a abs-d abs-ds abs-ve abs-ds dslength
  show ?case
  proof(cases fstate rule:fstate-case, auto simp del:a-evalF.simps a-evalC.simps set-map)

```

Case Lambda

```

fix  $\beta$  and lab and vs:: var list and c
assume ds-a-length: length vs = length ds-a

have  $|\beta(\text{lab} \mapsto b)| = |\beta| (\text{lab} \mapsto b-a)$ 
  unfolding below-fun-def using abs-b by simp
moreover

{ have  $|ve(\text{map} (\lambda v. (v, b)) vs [\mapsto] ds)|$ 
   $\lesssim |ve| \cup. |\text{map-of} (\text{rev} (\text{zip} (\text{map} (\lambda v. (v, b)) vs) ds))|$ 
  unfolding map-upds-def by (intro abs-venv-union)
  also
  have  $\dots \lesssim ve-a \cup. (\bigcup. (\text{map} (\lambda(v,k). |[v \mapsto k]|) (\text{zip} (\text{map} (\lambda v. (v, b)) vs) ds)))$ 
  using abs-ve abs-venv-map-of-rev
  by (auto intro:smap-union-mono simp add:venv-approx-def)
  also
  have  $\dots = ve-a \cup. (\bigcup. (\text{map} (\lambda(v,y). |[v,b \mapsto y]|) (\text{zip} vs ds)))$ 
  by (auto simp add: zip-map1 o-def split-def)
  also
  have  $\dots \lesssim ve-a \cup. (\bigcup. (\text{map} (\lambda(v,y). \{(v,b-a) := y\}.) (\text{zip} vs ds-a)))$ 
  proof-

```

from *abs-b abs-ds*
have *list-all2 venv-approx* (*map* ($\lambda(v, y). \llbracket (v, b) \mapsto y \rrbracket$) (*zip vs ds*))
 (*map* ($\lambda(v, y). \{(v, b-a) := y\}$) (*zip vs ds-a*))
by (*auto simp add: ds-approx-def d-approx-def venv-approx-def abs-venv-singleton list-all2-conv-all-nth*
intro:smap-singleton-mono list-all2I)
thus *?thesis*
by (*auto simp add:venv-approx-def intro: smap-union-mono[OF smap-less-refl smap-Union-mono]*)
qed
finally
have $|ve(\text{map } (\lambda v. (v, b)) \text{ vs } [\mapsto] \text{ ds})|$
 $\lesssim ve-a \cup. (\bigcup. (\text{map } (\lambda(v, y). \{(v, b-a) := y\}) (\text{zip vs ds-a})))$.
}
ultimately
have *prem*: $|(\text{c}, \beta(\text{lab} \mapsto \text{b}), ve(\text{map } (\lambda v. (v, b)) \text{ vs } [\mapsto] \text{ ds}), \text{b})|$
 $\lesssim (\text{c}, |\beta|(\text{lab} \mapsto \text{b-a}), ve-a \cup. (\bigcup. (\text{map } (\lambda(v, y). \{(v, b-a) := y\}) (\text{zip vs ds-a}))), \text{b-a})$
using *abs-b*
by(*auto intro:cstate-approx.intros simp add: abs-cstate.simps*)

show $|evalC \cdot (\text{Discr } (\text{c}, \beta(\text{lab} \mapsto \text{b}), ve(\text{map } (\lambda v. (v, b)) \text{ vs } [\mapsto] \text{ ds}), \text{b}))|$
 $\lesssim \widehat{\mathcal{F}} \cdot (\text{Discr } (\text{PC } (\text{Lambda } \text{lab vs c}, |\beta|), \text{ds-a}, \text{ve-a}, \text{b-a}))$
using *Next.hyps(2)[OF prem] ds-a-length*
by (*subst a-evalF.simps, simp del:a-evalF.simps a-evalC.simps*)

next

Case Plus

fix *lab a1 a2 cnt*
assume *isProc cnt*
assume *abs-ds'*: $\{ \{\}, \{\}, |cnt| \} \lesssim ds-a$
then obtain *a1-a a2-a cnt-a* **where** *ds-a*: $ds-a = [a1-a, a2-a, cnt-a]$ **and** *abs-cnt*: $|cnt| \lesssim cnt-a$
unfolding *ds-approx-def*
by (*cases ds-a rule:list.exhaust[OF - list.exhaust[OF - list.exhaust, of - - $\lambda x. x$], of - - $\lambda x. x$]*)
 (*auto simp add:ds-approx-def*)

have *new-elem*: $| \{ ((\text{lab}, [\text{lab} \mapsto \text{b}]), \text{cnt}) \} | \lesssim \{ ((\text{lab}, [\text{lab} \mapsto \text{b-a}]), \text{cont}) \mid \text{cont. cont} \in \text{cnt-a} \}$
using *abs-cnt and abs-b*
by (*auto simp add:ccache-approx-def d-approx-def*)

have *prem*: $|(\text{cnt}, [DI (a1 + a2)], ve, \widehat{nb} \text{ b lab})| \lesssim$
 (*the-elem |cnt|, [\{\}], ve-a, \widehat{nb} b-a lab)
using *abs-ve and abs-b*
by (*auto intro:fstate-approx.intros simp add:ds-approx-def*)

have $|(\text{evalF} \cdot (\text{Discr } (\text{cnt}, [DI (a1 + a2)], ve, \widehat{nb} \text{ b lab})))|$
 $\lesssim \widehat{\mathcal{F}} \cdot (\text{Discr } (\text{the-elem } |cnt|, [\{\}], ve-a, \widehat{nb} \text{ b-a lab}))$
by (*rule Next.hyps(1)[OF prem]*)*


```

also have ...  $\lesssim$  ( $\bigcup cnt \in cnt\text{-}a. \widehat{\mathcal{F}}.(Discr (cnt, [\{\}], ve\text{-}a, \widehat{nb} b\text{-}a lab))$ )
  using abs-cnt
  by (auto intro: the-elem-is-Proc[OF <isProc cnt>] simp del: a-evalF.simps simp add:ccache-approx-def
d-approx-def)
  finally
  have old-elems:  $|(\text{evalF}.(Discr (cnt, [DI (a1 + a2)], ve, nb b lab)))|$ 
     $\lesssim$  ( $\bigcup cnt \in cnt\text{-}a. \widehat{\mathcal{F}}.(Discr (cnt, [\{\}], ve\text{-}a, \widehat{nb} b\text{-}a lab))$ ).

  have  $|(\text{evalF}.(Discr (cnt, [DI (a1 + a2)], ve, nb b lab)))|$ 
     $\cup \{|((lab, [lab \mapsto b]), cnt)\}|$ 
     $\lesssim$   $|(\text{evalF}.(Discr (cnt, [DI (a1 + a2)], ve, nb b lab)))|$ 
     $\cup \{|((lab, [lab \mapsto b]), cnt)\}|$ 
    by (rule abs-ccache-union)
  also
  have ...  $\lesssim$ 
    ( $\bigcup cnt \in cnt\text{-}a. \widehat{\mathcal{F}}.(Discr (cnt, [\{\}], ve\text{-}a, \widehat{nb} b\text{-}a lab))$ )
     $\cup \{|((lab, [lab \mapsto b\text{-}a]), cont) | cont. cont \in cnt\text{-}a\}$ 
    by (rule Un-mono-approx[OF old-elems new-elem])
  finally
  show  $|insert ((lab, [lab \mapsto b]), cnt)$ 
     $(\text{evalF}.(Discr (cnt, [DI (a1 + a2)], ve, nb b lab)))|$ 
     $\lesssim$   $\widehat{\mathcal{F}}.(Discr (PP (prim.Plus lab), ds\text{-}a, ve\text{-}a, b\text{-}a))$ 
    using ds-a by (subst a-evalF.simps)(auto simp del:a-evalF.simps)
  next

```

Case If (true branch)

```

fix ct cf v cntt cntf
assume isProc cntt
assume isProc cntf
assume abs-ds':  $[\{\}, |cntt|, |cntf|] \lesssim ds\text{-}a$ 
then obtain v-a cntt-a cntf-a where ds-a:  $ds\text{-}a = [v\text{-}a, cntt\text{-}a, cntf\text{-}a]$ 
  and abs-cntt:  $|cntt| \lesssim cntt\text{-}a$ 
  and abs-cntf:  $|cntf| \lesssim cntf\text{-}a$ 
  by (cases ds-a rule:list.exhaust[OF - list.exhaust[OF - list.exhaust, of - -  $\lambda$ . x], of - -  $\lambda$ .
x. x])
    (auto simp add:ds-approx-def)

```

let *?c* = *ct::label* **and** *?cnt* = *cntt* **and** *?cnt-a* = *cntt-a*

```

have new-elem:  $|\{((?c, [?c \mapsto b]), ?cnt)\}| \lesssim \{|((?c, [?c \mapsto b\text{-}a]), cont) | cont. cont \in ?cnt\text{-}a\}$ 
  using abs-cntt and abs-cntf and abs-b
  by (auto simp add:ccache-approx-def d-approx-def)

```

```

have prem:  $|(?cnt, [], ve, nb b ?c)| \lesssim$ 
  (the-elem  $|?cnt|, [], ve\text{-}a, \widehat{nb} b\text{-}a ?c$ )
  using abs-ve and abs-b
  by (auto intro:fstate-approx.intros)

```

have $|evalF \cdot (Discr \ (\?cnt, [], ve, nb \ b \ ?c))|$
 $\lesssim \widehat{\mathcal{F}} \cdot (Discr \ (the\text{-}elem \ |\?cnt|, [], ve\text{-}a, \widehat{nb} \ b\text{-}a \ ?c))$
by (*rule Next.hyps(1)[OF prem]*)
also have $\dots \lesssim (\bigcup cnt \in \?cnt\text{-}a. \widehat{\mathcal{F}} \cdot (Discr \ (cnt, [], ve\text{-}a, \widehat{nb} \ b\text{-}a \ ?c)))$
using *abs-cntt and abs-cntf*
by (*auto intro: the-elem-is-Proc[OF <isProc ?cnt>] simp del: a-evalF.simps simp add:ccache-approx-def d-approx-def*)

finally

have *old-elems*: $|evalF \cdot (Discr \ (\?cnt, [], ve, nb \ b \ ?c))|$
 $\lesssim (\bigcup cnt \in \?cnt\text{-}a. \widehat{\mathcal{F}} \cdot (Discr \ (cnt, [], ve\text{-}a, \widehat{nb} \ b\text{-}a \ ?c)))$.

have $|evalF \cdot (Discr \ (\?cnt, [], ve, nb \ b \ ?c))$
 $\cup \{((?c, [?c \mapsto b]), ?cnt)\}$
 $\lesssim |evalF \cdot (Discr \ (\?cnt, [], ve, nb \ b \ ?c))|$
 $\cup \{((?c, [?c \mapsto b]), ?cnt)\}$
by (*rule abs-ccache-union*)

also

have $\dots \lesssim$
 $(\bigcup cnt \in \?cnt\text{-}a. \widehat{\mathcal{F}} \cdot (Discr \ (cnt, [], ve\text{-}a, \widehat{nb} \ b\text{-}a \ ?c)))$
 $\cup \{((?c, [?c \mapsto b\text{-}a]), cont) \mid cont. cont \in \?cnt\text{-}a\}$
by (*rule Un-mono-approx[OF old-elems new-elem]*)

also

have $\dots \lesssim$
 $(\bigcup cnt \in cntt\text{-}a. \widehat{\mathcal{F}} \cdot (Discr \ (cnt, [], ve\text{-}a, \widehat{nb} \ b\text{-}a \ ct)))$
 $\cup \{((ct, [ct \mapsto b\text{-}a]), cont) \mid cont. cont \in cntt\text{-}a\}$
 $\cup (\bigcup cnt \in cntf\text{-}a. \widehat{\mathcal{F}} \cdot (Discr \ (cnt, [], ve\text{-}a, \widehat{nb} \ b\text{-}a \ cf)))$
 $\cup \{((cf, [cf \mapsto b\text{-}a]), cont) \mid cont. cont \in cntf\text{-}a\}$
by (*rule Un-upper1-approx|rule Un-upper2-approx*)

finally

show $|insert \ ((?c, [?c \mapsto b]), ?cnt)$
 $(evalF \cdot (Discr \ (\?cnt, [], ve, nb \ b \ ?c)))| \lesssim$
 $\widehat{\mathcal{F}} \cdot (Discr \ (PP \ (prim.If \ ct \ cf), ds\text{-}a, ve\text{-}a, b\text{-}a))$
using *ds-a by (subst a-evalF.simps)(auto simp del:a-evalF.simps)*
next

Case If (false branch). We use schematic variable to keep this similar to the true branch.

fix *ct cf v cntt cntf*
assume *isProc cntt*
assume *isProc cntf*
assume *abs-ds'*: $[{\}, |cntt|, |cntf|] \lesssim ds\text{-}a$
then obtain *v-a cntt-a cntf-a* **where** *ds-a*: $ds\text{-}a = [v\text{-}a, cntt\text{-}a, cntf\text{-}a]$
and *abs-cntt*: $|cntt| \lesssim cntt\text{-}a$
and *abs-cntf*: $|cntf| \lesssim cntf\text{-}a$
by (*cases ds-a rule:list.exhaust[OF - list.exhaust[OF - list.exhaust, of - - λ- x], of - - λ-*
x. x])
(auto simp add:ds-approx-def)

let $?c = cf::label$ **and** $?cnt = cntf$ **and** $?cnt-a = cntf-a$

have $new\text{-}elem: |\{((?c, [?c \mapsto b]), ?cnt)\}| \lesssim \{((?c, [?c \mapsto b-a]), cont) \mid cont. cont \in ?cnt-a\}$
using $abs\text{-}cntt$ **and** $abs\text{-}cntf$ **and** $abs\text{-}b$
by $(auto\ simp\ add:ccache\text{-}approx\text{-}def\ d\text{-}approx\text{-}def)$

have $prem: |(?cnt, [], ve, nb\ b\ ?c)| \lesssim$
 $(the\text{-}elem\ |?cnt|, [], ve\text{-}a, nb\ b\text{-}a\ ?c)$
using $abs\text{-}ve$ **and** $abs\text{-}b$
by $(auto\ intro:fstate\text{-}approx.intros)$

have $|evalF.(Discr\ (?cnt, [], ve, nb\ b\ ?c))|$
 $\lesssim \widehat{\mathcal{F}}.(Discr\ (the\text{-}elem\ |?cnt|, [], ve\text{-}a, \widehat{nb}\ b\text{-}a\ ?c))$
by $(rule\ Next.hyps(1)[OF\ prem])$
also have $\dots \lesssim (\bigcup cnt \in ?cnt\text{-}a. \widehat{\mathcal{F}}.(Discr\ (cnt, [], ve\text{-}a, \widehat{nb}\ b\text{-}a\ ?c)))$
using $abs\text{-}cntt$ **and** $abs\text{-}cntf$
by $(auto\ intro: the\text{-}elem\text{-}is\text{-}Proc[OF\ \langle isProc\ ?cnt \rangle] simp\ del: a\text{-}evalF.simps\ simp\ add:ccache\text{-}approx\text{-}def\ d\text{-}approx\text{-}def)$

finally
have $old\text{-}elems: |evalF.(Discr\ (?cnt, [], ve, nb\ b\ ?c))|$
 $\lesssim (\bigcup cnt \in ?cnt\text{-}a. \widehat{\mathcal{F}}.(Discr\ (cnt, [], ve\text{-}a, \widehat{nb}\ b\text{-}a\ ?c)))$.

have $|evalF.(Discr\ (?cnt, [], ve, nb\ b\ ?c))$
 $\cup \{((?c, [?c \mapsto b]), ?cnt)\}|$
 $\lesssim |evalF.(Discr\ (?cnt, [], ve, nb\ b\ ?c))|$
 $\cup \{((?c, [?c \mapsto b]), ?cnt)\}|$
by $(rule\ abs\text{-}ccache\text{-}union)$

also
have $\dots \lesssim$
 $(\bigcup cnt \in ?cnt\text{-}a. \widehat{\mathcal{F}}.(Discr\ (cnt, [], ve\text{-}a, \widehat{nb}\ b\text{-}a\ ?c)))$
 $\cup \{((?c, [?c \mapsto b-a]), cont) \mid cont. cont \in ?cnt\text{-}a\}$
by $(rule\ Un\text{-}mono\text{-}approx[OF\ old\text{-}elems\ new\text{-}elem])$

also
have $\dots \lesssim$
 $(\bigcup cnt \in cntt\text{-}a. \widehat{\mathcal{F}}.(Discr\ (cnt, [], ve\text{-}a, \widehat{nb}\ b\text{-}a\ ct)))$
 $\cup \{((ct, [ct \mapsto b-a]), cont) \mid cont. cont \in cntt\text{-}a\}$
 $\cup (\bigcup cnt \in cntf\text{-}a. \widehat{\mathcal{F}}.(Discr\ (cnt, [], ve\text{-}a, \widehat{nb}\ b\text{-}a\ cf)))$
 $\cup \{((cf, [cf \mapsto b-a]), cont) \mid cont. cont \in cntf\text{-}a\}$
by $(rule\ Un\text{-}upper1\text{-}approx|rule\ Un\text{-}upper2\text{-}approx)$

finally
show $|insert\ ((?c, [?c \mapsto b]), ?cnt)$
 $(evalF.(Discr\ (?cnt, [], ve, nb\ b\ ?c)))| \lesssim$
 $\widehat{\mathcal{F}}.(Discr\ (PP\ (prim.If\ ct\ cf), ds\text{-}a, ve\text{-}a, b\text{-}a))$
using $ds\text{-}a$ **by** $(subst\ a\text{-}evalF.simps)(auto\ simp\ del:a\text{-}evalF.simps)$

qed
next

case 2

obtain $c \beta ve b$ **where** $cstate: cstate = (c, \beta, ve, b)$
by $(cases\ cstate, auto)$
moreover
obtain $c-a \beta-a ds-a ve-a b-a$ **where** $cstate-a: cstate-a = (c-a, \beta-a, ve-a, b-a)$
by $(cases\ cstate-a, auto)$
ultimately
have $abs-c: c = c-a$
and $abs-\beta: |\beta| = \beta-a$
and $abs-ve: |ve| \lesssim ve-a$
and $abs-b: |b| = b-a$
using 2 **by** $(auto\ elim:cstate-approx.cases)$

from $cstate\ cstate-a\ abs-c\ abs-\beta\ abs-b$
show $?case$
proof $(cases\ c, auto\ simp\ add:HOL.Let-def\ simp\ del:a-evalF.simps\ a-evalC.simps\ set-map\ evalV.simps)$

Case App

fix $lab\ f\ vs$
let $?d = \mathcal{A}\ f\ \beta\ ve$
assume $isProc\ ?d$

have $map\ (abs-d \circ (\lambda v. \mathcal{A}\ v\ \beta\ ve))\ vs \lesssim map\ (\lambda v. \widehat{\mathcal{A}}\ v\ \beta-a\ ve-a)\ vs$
using $abs-\beta$ **and** $lemma7[OF\ abs-ve, of - \beta]$
by $(auto\ intro!: list-all2I\ simp\ add:set-zip\ ds-approx-def)$

hence $|evalF.(Discr\ (?d, map\ (\lambda v. \mathcal{A}\ v\ \beta\ ve)\ vs, ve, nb\ b\ lab))|$
 $\lesssim \widehat{\mathcal{F}}.(Discr(the-elem\ |?d|, map\ (\lambda v. \widehat{\mathcal{A}}\ v\ \beta-a\ ve-a)\ vs, ve-a, \widehat{nb}\ |b|\ lab))$
using $abs-ve$ **and** $abs-cnt-nb$ **and** $abs-b$
by $-(rule\ Next.hyps(1), auto\ intro:fstate-approx.intros)$
also have $\dots \lesssim (\bigcup_{f' \in \widehat{\mathcal{A}}} f\ \beta-a\ ve-a.$
 $\widehat{\mathcal{F}}.(Discr(f', map\ (\lambda v. \widehat{\mathcal{A}}\ v\ \beta-a\ ve-a)\ vs, ve-a, \widehat{nb}\ |b|\ lab)))$
using $lemma7[OF\ abs-ve]\ the-elem-is-Proc[OF\ isProc\ ?d]\ abs-\beta$
by $(auto\ simp\ del: a-evalF.simps\ simp\ add:d-approx-def\ ccache-approx-def)$
finally
have $old-elems:$
 $|evalF.(Discr\ (\mathcal{A}\ f\ \beta\ ve, map\ (\lambda v. \mathcal{A}\ v\ \beta\ ve)\ vs, ve, nb\ b\ lab))|$
 $\lesssim (\bigcup_{f' \in \widehat{\mathcal{A}}} f\ \beta-a\ ve-a.$
 $\widehat{\mathcal{F}}.(Discr(f', map\ (\lambda v. \widehat{\mathcal{A}}\ v\ \beta-a\ ve-a)\ vs, ve-a, \widehat{nb}\ |b|\ lab)))$
by $auto$

have $new-elem: \{|((lab, \beta), \mathcal{A}\ f\ \beta\ ve)\}$
 $\lesssim \{|((lab, \beta-a), f')\ | f'. f' \in \widehat{\mathcal{A}}\ f\ \beta-a\ ve-a\}$
using $abs-\beta$ **and** $lemma7[OF\ abs-ve]$
by $(auto\ simp\ add:ccache-approx-def\ d-approx-def)$

have $|evalF.(Discr\ (\mathcal{A}\ f\ \beta\ ve, map\ (\lambda v. \mathcal{A}\ v\ \beta\ ve)\ vs, ve, nb\ b\ lab))$

$\cup \{|(lab, \beta), \mathcal{A} f \beta ve|\}$
 $\approx \{|evalF \cdot (Discr (\mathcal{A} f \beta ve, map (\lambda v. \mathcal{A} v \beta ve) vs, ve, nb b lab))|\}$
 $\cup \{|(lab, \beta), \mathcal{A} f \beta ve|\}$
by (*rule abs-ccache-union*)
also have ...
 $\approx (\bigcup_{f' \in \widehat{\mathcal{A}}} f \beta\text{-}a \text{ ve-}a.$
 $\quad \widehat{F} \cdot (Discr (f', map (\lambda v. \widehat{\mathcal{A}} v \beta\text{-}a \text{ ve-}a) vs, ve\text{-}a, \widehat{nb} |b| lab)))$
 $\cup \{|(lab, \beta\text{-}a), f') \mid f' \in \widehat{\mathcal{A}} f \beta\text{-}a \text{ ve-}a\}$
by (*rule Un-mono-approx[OF old-elems new-elem]*)
finally
show $|insert ((lab, \beta), \mathcal{A} f \beta ve)$
 $(evalF \cdot (Discr (\mathcal{A} f \beta ve, map (\lambda v. \mathcal{A} v \beta ve) vs, ve, nb b lab)))|$
 $\approx \widehat{C} \cdot (Discr (App lab f vs, |\beta|, ve\text{-}a, |b|))$
using *abs- β*
by (*subst a-evalC.simps*)(*auto simp add: HOL.Let-def simp del:a-evalF.simps*)
next

Case Let

fix *lab binds c'*

have $|\beta(lab \mapsto nb b lab)| =$
 $\beta\text{-}a(lab \mapsto \widehat{nb} |b| lab)$
using *abs- β* **and** *abs-b*
by *simp*
moreover
have $|map\text{-of} (map (\lambda(v, l). ((v, nb b lab),$
 $DC (l, \beta(lab \mapsto nb b lab))))$
 $binds)|$
 $\approx \bigcup. (map (\lambda(v, l).$
 $\{(v, \widehat{nb} |b| lab) := \{PC (l, \beta\text{-}a(lab \mapsto \widehat{nb} |b| lab))\}\}.)$
 $binds)$
using *abs-b* **and** *abs- β*
apply $-$
apply (*rule venv-approx-trans[OF abs-venv-map-of]*)
apply (*auto intro:smap-union-mono list-all2I*
 $simp add:venv-approx-def o-def set-zip abs-venv-singleton split-def smap-less-refl$)
done
hence $|ve ++ map\text{-of}$
 $(map (\lambda(v, l).$
 $((v, nb b lab),$
 $DC (l, \beta(lab \mapsto nb b lab))))$
 $binds)| \approx$
 $ve\text{-}a \cup.$
 $(\bigcup.$
 $(map (\lambda(v, l).$
 $\{(v, \widehat{nb} |b| lab) := \{PC (l, \beta\text{-}a(lab \mapsto \widehat{nb} |b| lab))\}\}.)$
 $binds)$
by (*rule venv-approx-trans[OF abs-venv-union*

```

      smap-union-mono[OF abs-ve[unfolded venv-approx-def], folded venv-approx-def]])
ultimately
have |evalC·(Discr(c', β(lab ↦ nb b lab),
  ve ++ map-of
    (map (λ(v, l). ((v, nb b lab), DC (l, β(lab ↦ nb b lab)))) binds),
  nb b lab)|
  ≲  $\widehat{\mathcal{C}}$ ·(Discr (c', β-a(lab ↦  $\widehat{nb}$  |b| lab),
  ve-a  $\sqcup$ .
    ( $\sqcup$ . (map (λ(v, l).
      {(v,  $\widehat{nb}$  |b| lab) := {PC (l, β-a(lab ↦  $\widehat{nb}$  |b| lab))}}.)
    binds)),
   $\widehat{nb}$  |b| lab))
using abs-cnt-nb and abs-b
by -(rule Next.hyps(2), auto intro: cstate-approx.intros)

thus |evalC·(Discr (c', β(lab ↦ nb b lab),
  ve ++ map-of (map (λ(v, l).((v, nb b lab),  $\mathcal{A}$  (L l) (β(lab ↦ nb b lab)) ve))
binds),
  nb b lab)| ≲
   $\widehat{\mathcal{C}}$ ·(Discr (call.Let lab binds c', |β|, ve-a, |b| ))
using abs-β
by (subst a-evalC.simps)(auto simp add: HOL.Let-def simp del:a-evalC.simps)
qed
}
qed

```

And finally, we lift this result to $\widehat{\mathcal{PR}}$ and \mathcal{PR} .

```

lemma lemma6: | $\mathcal{PR}$  l| ≲  $\widehat{\mathcal{PR}}$  l
  unfolding evalCPS-def evalCPS-a-def
  by (auto intro!: lemma89 fstate-approx.intros simp del:evalF.simps a-evalF.simps
    simp add: ds-approx-def d-approx-def venv-approx-def)
end

```

7. Generic Computability

```

theory Computability
imports HOLCF HOLCFUtils
begin

```

Shivers proves the computability of the abstract semantics functions only by generic and slightly simplified example. This theory contains the abstract treatment in Section 4.4.3. Later, we will work out the details apply this to $\widehat{\mathcal{PR}}$.

7.1. Non-branching case

After the following lemma (which could go into *HOL.Set-Interval*), we show Shivers' Theorem 10. This says that the least fixed point of the equation

$$f\ x = g\ x \cup f\ (r\ x)$$

is given by

$$f\ x = \bigcup_{i \geq 0} g\ (r^i\ x).$$

The proof follows the standard proof of showing an equality involving a fixed point: First we show that the right hand side fulfills the above equation and then show that our solution is less than any other solution to that equation.

lemma *insert-greaterThan*:

insert ($n::nat$) $\{n<..\}$ = $\{n..\}$

by *auto*

lemma *theorem10*:

fixes $g :: 'a::cpo \rightarrow 'b::type\ set$ **and** $r :: 'a \rightarrow 'a$

shows $fix\ (\lambda f\ x. g\ x \cup f\ (r\ x)) = (\lambda x. (\bigcup i. g\ (r^i\ x)))$

proof (*induct rule:fix-eqI[OF cfun-eqI cfun-belowI, case-names fp least]*)

case (*fp* x)

have $g\ x \cup (\bigcup i. g\ (r^i\ (r\ x))) = g\ (r^0\ x) \cup (\bigcup i. g\ (r^{Suc\ i}\ x))$

by (*simp add: iterate-Suc2 del: iterate-Suc*)

also have $\dots = g\ (r^0\ x) \cup (\bigcup i \in \{0<..\}. g\ (r^i\ x))$

using *less-iff-Suc-add* by *auto*

also have $\dots = (\bigcup i \in insert\ 0\ \{0<..\}. g\ (r^i\ x))$

by *simp*

also have $\dots = (\bigcup i. g\ (r^i\ x))$

by (*simp only: insert-greaterThan atLeast-0*)

finally

show *?case* by *auto*

next

case (*least* $f\ x$)

hence *expand*: $\bigwedge x. f\ x = (g\ x \cup f\ (r\ x))$ by (*auto simp: cfun-eq-iff*)

{ fix n

have $f\ x = (\bigcup i \in \{..n\}. g\ (r^i\ x)) \cup f\ (r^{Suc\ n}\ x)$

proof (*induct* n)

case 0 **thus** *?case* by (*auto simp add: expand[of x]*)

case (*Suc* n)

then have $f\ x = (\bigcup i \in \{..n\}. g\ (r^i\ x)) \cup f\ (r^{Suc\ n}\ x)$ by *simp*

also have $\dots = (\bigcup i \in \{..n\}. g\ (r^i\ x))$

$\cup g\ (r^{Suc\ n}\ x) \cup f\ (r^{Suc\ (Suc\ n)}\ x)$

by (*subst expand[of r^{Suc\ n}\ x], auto*)

also have $\dots = (\bigcup i \in insert\ (Suc\ n)\ \{..n\}. g\ (r^i\ x)) \cup f\ (r^{Suc\ (Suc\ n)}\ x)$

```

    by auto
  also have ... = (⋃ i ∈ {..Suc n}. g.(ri.x)) ∪ f.(rSuc (Suc n).x)
    by (simp add:atMost-Suc)
  finally show ?case .
qed
} note fin = this
have (⋃ i. g.(ri.x)) ⊆ f.x
proof(rule UN-least)
  fix i
  show g.(ri.x) ⊆ f.x
  using fin[of i] by auto
qed
thus ?case
  apply (subst sqsubset-is-subset) by auto
qed

```

7.2. Branching case

Actually, our functions are more complicated than the one above: The abstract semantics functions recurse with multiple arguments. So we have to handle a recursive equation of the kind

$$f\ x = g\ x \cup \bigcup_{a \in R\ x} f\ r.$$

By moving to the power-set relatives of our function, e.g.

$$\underline{g}Y = \bigcup_{a \in A} g\ a \quad \text{and} \quad \underline{R}Y = \bigcup_{a \in R} R\ a$$

the equation becomes

$$\underline{f}Y = \underline{g}Y \cup \underline{f}(\underline{R}Y)$$

(which is shown in Lemma 11) and we can apply Theorem 10 to obtain Theorem 12.

We define the power-set relative for a function together with some properties.

definition *powerset-lift* :: ('a::cpo → 'b::type set) ⇒ 'a set → 'b set ()
where $\underline{f} = (\lambda S. (\bigcup y \in S. f.y))$

lemma *powerset-lift-singleton*[simp]:

$$\underline{f}\{x\} = f.x$$

unfolding *powerset-lift-def* **by** *simp*

lemma *powerset-lift-union*[simp]:

$$\underline{f}(A \cup B) = \underline{f}A \cup \underline{f}B$$

unfolding *powerset-lift-def* **by** *auto*

lemma *UNION-commute*: $(\bigcup x \in A. \bigcup y \in B. P\ x\ y) = (\bigcup y \in B. \bigcup x \in A. P\ x\ y)$
by *auto*

lemma *powerset-lift-UNION*:
 $(\bigcup_{x \in S}. \underline{g} \cdot (A \ x)) = \underline{g} \cdot (\bigcup_{x \in S}. A \ x)$
unfolding *powerset-lift-def* **by** *auto*

lemma *powerset-lift-iterate-UNION*:
 $(\bigcup_{x \in S}. (\underline{g})^i \cdot (A \ x)) = (\underline{g})^i \cdot (\bigcup_{x \in S}. A \ x)$
by (*induct* i , *auto simp add:powerset-lift-UNION*)

lemmas *powerset-distr = powerset-lift-UNION powerset-lift-iterate-UNION*

Lemma 11 shows that if a function satisfies the relation with the branching R , its powerset function satisfies the powerset variant of the equation.

lemma *lemma11*:
fixes $f :: 'a \rightarrow 'b \text{ set}$ **and** $g :: 'a \rightarrow 'b \text{ set}$ **and** $R :: 'a \rightarrow 'a \text{ set}$
assumes $\bigwedge x. f \cdot x = g \cdot x \cup (\bigcup_{y \in R \cdot x}. f \cdot y)$
shows $\underline{f} \cdot S = \underline{g} \cdot S \cup \underline{f} \cdot (\underline{R} \cdot S)$
proof –
have $\underline{f} \cdot S = (\bigcup_{x \in S}. \underline{f} \cdot x)$ **unfolding** *powerset-lift-def* **by** *auto*
also have $\dots = (\bigcup_{x \in S}. g \cdot x \cup (\bigcup_{y \in R \cdot x}. f \cdot y))$ **apply** (*subst assms*) **by** *simp*
also have $\dots = \underline{g} \cdot S \cup \underline{f} \cdot (\underline{R} \cdot S)$ **by** (*auto simp add:powerset-lift-def*)
finally
show *?thesis* .
qed

Theorem 10 as it will be used in Theorem 12.

lemmas *theorem10ps = theorem10[of g r]* **for** $g \ r$

Now we can show Lemma 12: If F is the least solution to the recursive power-set equation, then $x \mapsto F \ x$ is the least solution to the equation with branching R .

We fix the type variable $'a$ to be a discrete cpo, as otherwise $x \mapsto \{x\}$ is not continuous.

lemma *theorem12'*:
fixes $g :: 'a::\text{discrete-cpo} \rightarrow 'b::\text{type set}$ **and** $R :: 'a \rightarrow 'a \text{ set}$
assumes $F\text{-fix}: F = \text{fix} \cdot (\lambda F \ x. \underline{g} \cdot x \cup F \cdot (\underline{R} \cdot x))$
shows $\text{fix} \cdot (\lambda f \ x. g \cdot x \cup (\bigcup_{y \in R \cdot x}. f \cdot y)) = (\lambda x. F \cdot \{x\})$
proof (*induct rule:fix-eqI[OF cfun-eqI cfun-belowI, case-names fp least]*)
have $F\text{-union}: F = (\lambda x. \bigcup i. \underline{g} \cdot ((\underline{R})^i \cdot x))$
using $F\text{-fix}$ **by** (*simp*)(*rule theorem10ps*)
case (*fp* x)
have $g \cdot x \cup (\bigcup_{x' \in R \cdot x}. F \cdot \{x'\}) = \underline{g} \cdot \{x\} \cup F \cdot (\underline{R} \cdot \{x\})$
unfolding *powerset-lift-singleton*
by (*auto simp add: powerset-distr UNION-commute F-union*)
also have $\dots = F \cdot \{x\}$
by (*subst* (2) *fix-eq4*[*OF F-fix*], *auto*)

finally show *?case by simp*
next
case (*least f' x*)
hence *expand: f' = (Λ x. g.x ∪ (∪ y∈R.x. f'.y)) by simp*
have *f' = (Λ S. g.S ∪ f'.(R.S))*
by (*subst expand, rule cfun-eqI, auto simp add:powerset-lift-def*)
hence (*Λ F. Λ x. g.x ∪ F.(R.x).(f') = f' by simp*)
from *fix-least[OF this] and F-fix*
have *F ⊆ f' by simp*
hence *F.{x} ⊆ f'.{x}*
by (*subst (asm)cfun-below-iff, auto simp del:powerset-lift-singleton*)
thus *?case by (auto simp add:sqsubset-is-subset)*
qed

lemma *theorem12:*

fixes *g :: 'a::discrete-cpo → 'b::type set and R :: 'a → 'a set*
shows *fix.(Λ f x. g.x ∪ (∪ y∈R.x. f.y)).x = g.(∪ i.((R)ⁱ.{x}))*
by(*subst theorem12'[OF theorem10ps[THEN sym]], auto simp add:powerset-distr*)

end

8. The abstract semantics is computable

theory *AbsCFComp*

imports *AbsCF Computability FixTransform CPSUtils MapSets*

begin

default-sort *type*

The point of the abstract semantics is that it is computable. To show this, we exploit the special structure of $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$: Each call adds some elements to the result set and joins this with the results from a number of recursive calls. So we separate these two actions into separate functions. These take as arguments the direct sum of \widehat{fstate} and \widehat{cstate} , i.e. we treat the two mutually recursive functions now as one.

abs-g gives the local result for the given argument.

fixrec *abs-g :: ('c::contour \widehat{fstate} + 'c \widehat{cstate}) discr → 'c \widehat{ans}*
where *abs-g.x = (case undiscr x of*
(Inl (PC (Lambda lab vs c, β), as, ve, b)) ⇒ {}
| (Inl (PP (Plus c),[-,-,cnts],ve,b)) ⇒
let b' = \widehat{nb} b c;
β = [c ↦ b]
in {(c, β), cont} | cont . cont ∈ cnts}
| (Inl (PP (prim.If ct cf),[-, cntts, cntfs],ve,b)) ⇒
((let b' = \widehat{nb} b ct;
β = [ct ↦ b]

$$\begin{aligned}
& \text{in } \{((ct, \beta), cnt) \mid cnt . cnt \in cntts\} \\
&) \cup (\\
& \quad \text{let } b' = \widehat{nb} \ b \ cf; \\
& \quad \quad \beta = [cf \mapsto b] \\
& \quad \text{in } \{((cf, \beta), cnt) \mid cnt . cnt \in cntfs\} \\
&) \\
& | (Inl (AStop,[-,-,-]) \Rightarrow \{\}) \\
& | (Inl -) \Rightarrow \perp \\
& | (Inr (App lab f vs,\beta,ve,b)) \Rightarrow \\
& \quad \text{let } fs = \widehat{A} \ f \ \beta \ ve; \\
& \quad \quad as = \text{map } (\lambda v. \widehat{A} \ v \ \beta \ ve) \ vs; \\
& \quad \quad b' = \widehat{nb} \ b \ lab \\
& \quad \text{in } \{((lab, \beta), f') \mid f' . f' \in fs\} \\
& | (Inr (Let lab ls c',\beta,ve,b)) \Rightarrow \{\}) \\
&)
\end{aligned}$$

abs-R gives the set of arguments passed to the recursive calls.

fixrec *abs-R* :: ('c::contour \widehat{fstate} + 'c \widehat{cstate}) *discr* \rightarrow ('c::contour \widehat{fstate} + 'c \widehat{cstate}) *discr set*

where *abs-R*.*x* = (case *undiscr x* of

$$\begin{aligned}
& (Inl (PC (Lambda lab vs c, \beta), as, ve, b)) \Rightarrow \\
& \quad (\text{if } \text{length } vs = \text{length } as \\
& \quad \text{then let } \beta' = \beta \ (lab \mapsto b); \\
& \quad \quad ve' = ve \cup. (\bigcup. (\text{map } (\lambda(v,a). \{(v,b) := a\}.)) \ (\text{zip } vs \ as))) \\
& \quad \quad \text{in } \{Discr (Inr (c,\beta',ve',b))\} \\
& \quad \text{else } \perp) \\
& | (Inl (PP (Plus c),[-,-, cnts],ve,b)) \Rightarrow \\
& \quad \text{let } b' = \widehat{nb} \ b \ c; \\
& \quad \quad \beta = [c \mapsto b] \\
& \quad \quad \text{in } (\bigcup cnt \in cnts. \{Discr (Inl (cnt, [\{\}], ve, b')\}) \\
& | (Inl (PP (prim.If ct cf),[-, cntts, cntfs],ve,b)) \Rightarrow \\
& \quad ((\text{let } b' = \widehat{nb} \ b \ ct; \\
& \quad \quad \beta = [ct \mapsto b] \\
& \quad \quad \text{in } (\bigcup cnt \in cntts . \{Discr (Inl (cnt, [], ve, b')\}) \\
& \quad) \cup (\\
& \quad \quad \text{let } b' = \widehat{nb} \ b \ cf; \\
& \quad \quad \quad \beta = [cf \mapsto b] \\
& \quad \quad \quad \text{in } (\bigcup cnt \in cntfs . \{Discr (Inl (cnt, [], ve, b')\}) \\
& \quad) \\
& | (Inl (AStop,[-,-,-]) \Rightarrow \{\}) \\
& | (Inl -) \Rightarrow \perp \\
& | (Inr (App lab f vs,\beta,ve,b)) \Rightarrow \\
& \quad \text{let } fs = \widehat{A} \ f \ \beta \ ve; \\
& \quad \quad as = \text{map } (\lambda v. \widehat{A} \ v \ \beta \ ve) \ vs; \\
& \quad \quad b' = \widehat{nb} \ b \ lab \\
& \quad \quad \text{in } (\bigcup f' \in fs. \{Discr (Inl (f', as, ve, b')\}) \\
& | (Inr (Let lab ls c',\beta,ve,b)) \Rightarrow
\end{aligned}$$

$$\begin{aligned}
& \text{let } b' = \widehat{nb} \ b \ \text{lab}; \\
& \quad \beta' = \beta \ (\text{lab} \mapsto b'); \\
& \quad ve' = ve \cup. (\cup. (\text{map} \ (\lambda(v,l). \{(v,b') := (\widehat{\mathcal{A}} \ (L \ l) \ \beta' \ ve)\}.) \ ls)) \\
& \text{in } \{\text{Discr} \ (\text{Inr} \ (c', \beta', ve', b'))\} \\
&)
\end{aligned}$$

The initial argument vector, as created by $\widehat{\mathcal{PR}}$.

definition $\text{initial-r} :: \text{prog} \Rightarrow ('c::\text{contour} \ \widehat{fstate} + 'c \ \widehat{cstate}) \ \text{discr}$
where $\text{initial-r} \ \text{prog} = \text{Discr} \ (\text{Inl} \ (\text{the-elem} \ (\widehat{\mathcal{A}} \ (L \ \text{prog}) \ \text{Map.empty} \ \{\}.), [\{AStop\}], \{\}., \widehat{b_0}))$

8.1. Towards finiteness

We need to show that the set of possible arguments for a given program p is finite. Therefore, we define the set of possible procedures, of possible arguments to $\widehat{\mathcal{F}}$, or possible arguments to $\widehat{\mathcal{C}}$ and of possible arguments.

definition $\text{proc-poss} :: \text{prog} \Rightarrow 'c::\text{contour} \ \text{proc} \ \text{set}$
where $\text{proc-poss} \ p = \text{PC} \ ' (\text{lambdas} \ p \times \text{maps-over} \ (\text{labels} \ p) \ \text{UNIV}) \cup \text{PP} \ ' \ \text{prims} \ p \cup \{AStop\}$

definition $\text{fstate-poss} :: \text{prog} \Rightarrow 'c::\text{contour} \ \text{a-fstate} \ \text{set}$
where $\text{fstate-poss} \ p = (\text{proc-poss} \ p \times \text{NList} \ (\text{Pow} \ (\text{proc-poss} \ p))) \ (\text{call-list-lengths} \ p) \times \text{smaps-over} \ (\text{vars} \ p \times \text{UNIV}) \ (\text{proc-poss} \ p) \times \text{UNIV}$

definition $\text{cstate-poss} :: \text{prog} \Rightarrow 'c::\text{contour} \ \text{a-cstate} \ \text{set}$
where $\text{cstate-poss} \ p = (\text{calls} \ p \times \text{maps-over} \ (\text{labels} \ p) \ \text{UNIV}) \times \text{smaps-over} \ (\text{vars} \ p \times \text{UNIV}) \ (\text{proc-poss} \ p) \times \text{UNIV}$

definition $\text{arg-poss} :: \text{prog} \Rightarrow ('c::\text{contour} \ \text{a-fstate} + 'c \ \text{a-cstate}) \ \text{discr} \ \text{set}$
where $\text{arg-poss} \ p = \text{Discr} \ ' (\text{fstate-poss} \ p \lt+\gt \ \text{cstate-poss} \ p)$

Using the auxiliary results from *Shivers-CFA.CPSUtils*, we see that the argument space as defined here is finite.

lemma $\text{finite-arg-space: finite} \ (\text{arg-poss} \ p)$
unfolding arg-poss-def **and** cstate-poss-def **and** fstate-poss-def **and** proc-poss-def
by $(\text{auto} \ \text{intro!}: \text{finite-cartesian-product} \ \text{finite-imageI} \ \text{maps-over-finite} \ \text{smaps-over-finite} \ \text{finite-UNIV} \ \text{finite-Nlist})$

But is it closed? I.e. if we pass a member of arg-poss to abs-R , are the generated recursive call arguments also in arg-poss ? This is shown in $\text{arg-space-complete}$, after proving an auxiliary result about the possible outcome of a call to $\widehat{\mathcal{A}}$ and an admissibility lemma.

lemma evalV-possible :
assumes $f: f \in \widehat{\mathcal{A}} \ d \ \beta \ ve$

```

and  $d$ :  $d \in \text{vals } p$ 
and  $ve$ :  $ve \in \text{smaps-over } (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)$ 
and  $\beta$ :  $\beta \in \text{maps-over } (\text{labels } p) \text{ UNIV}$ 
shows  $f \in \text{proc-poss } p$ 
proof (cases ( $d, \beta, ve$ ) rule: evalV-a.cases)
case ( $1 \text{ cl } \beta' ve'$ )
  thus ?thesis using  $f$  by auto next
case ( $2 \text{ prim } \beta' ve'$ )
  thus ?thesis using  $d f$ 
  by (auto dest: vals1 simp add:proc-poss-def)
next
case ( $3 \text{ l var } \beta' ve'$ )
  thus ?thesis using  $f d \text{ smaps-over-im}[OF - ve]$ 
  by (auto split:option.split-asm dest: vals2)
next
case ( $4 \text{ l } \beta ve$ )
  thus ?thesis using  $f d \beta$ 
  by (auto dest!: vals3 simp add:proc-poss-def)
qed

lemma adm-subset: cont ( $\lambda x. f x \implies \text{adm } (\lambda x. f x \subseteq S)$ )
by (subst sqsubset-is-subset[THEN sym], intro adm-lemmas cont2cont)

```

```

lemma arg-space-complete:
   $state \in \text{arg-poss } p \implies \text{abs-R.state} \subseteq \text{arg-poss } p$ 
proof (induct rule: abs-R.induct[case-names Admissibility Bot Step])
case Admissibility show ?case
  by (intro adm-lemmas adm-subset cont2cont)
next
case Bot show ?case by simp next
case (Step abs-R)
  note  $state = \text{Step}(2)$ 
  show ?case
  proof (cases state)
  case (Discr state') show ?thesis
  proof (cases state')
  case (Inl fstate) show ?thesis
  using Inl Discr state
  proof (cases fstate rule: a-fstate-case, auto)

```

Case Lambda

```

fix  $l \text{ vs } c \beta \text{ as } ve \text{ b}$ 
assume Discr (Inl (PC (Lambda  $l \text{ vs } c, \beta$ ),  $as, ve, b$ ))  $\in \text{arg-poss } p$ 
  hence  $lam$ : Lambda  $l \text{ vs } c \in \text{lambdas } p$ 
  and  $beta$ :  $\beta \in \text{maps-over } (\text{labels } p) \text{ UNIV}$ 
  and  $ve$ :  $ve \in \text{smaps-over } (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)$ 
  and  $as$ :  $as \in \text{NList } (\text{Pow } (\text{proc-poss } p)) (\text{call-list-lengths } p)$ 

```

unfolding *arg-poss-def fstate-poss-def proc-poss-def* **by** *auto*
from *lam* **have** $c \in \text{calls } p$
by (*rule lambdas1*)
moreover
from *lam* **have** $l \in \text{labels } p$
by (*rule lambdas2*)
with *beta* **have** $\beta(l \mapsto b) \in \text{maps-over } (\text{labels } p) \text{ UNIV}$
by (*rule maps-over-upd, auto*)
moreover
from *lam* **have** $vs: \text{set } vs \subseteq \text{vars } p$ **by** (*rule lambdas3*)
from *as* **have** $\forall x \in \text{set } as. x \in \text{Pow } (\text{proc-poss } p)$
unfolding *NList-def nList-def* **by** *auto*
with *vs* **have** $ve \cup. \bigcup. \text{map } (\lambda(v, y). \{ (v, b) := y \}.) (\text{zip } vs \text{ as})$
 $\in \text{smaps-over } (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)$ (**is** $?ve' \in -$)
by (*auto intro!: smaps-over-un[OF ve] smaps-over-Union smaps-over-singleton*)
(*auto simp add:set-zip*)
ultimately
have $(c, \beta(l \mapsto b), ?ve', b) \in \text{cstate-poss } p$ (**is** $?cstate \in -$)
unfolding *cstate-poss-def* **by** *simp*
thus *Discr* (*Inr* $?cstate$) $\in \text{arg-poss } p$
unfolding *arg-poss-def* **by** *auto*
next

Case Plus

fix *ve b l v1 v2 cnts cnt*
assume *Discr* (*Inl* (*PP* (*prim.Plus* *l*), [*v1*, *v2*, *cnts*], *ve*, *b*)) $\in \text{arg-poss } p$
and $cnt \in \text{cnts}$
hence $cnt \in \text{proc-poss } p$
and $ve \in \text{smaps-over } (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)$
unfolding *arg-poss-def fstate-poss-def NList-def nList-def*
by *auto*
moreover
have $\{\{\}\} \in \text{NList } (\text{Pow } (\text{proc-poss } p)) (\text{call-list-lengths } p)$
unfolding *call-list-lengths-def NList-def nList-def* **by** *auto*
ultimately
have $(cnt, \{\{\}\}, ve, \widehat{nb} \text{ } b \text{ } l) \in \text{fstate-poss } p$
unfolding *fstate-poss-def* **by** *auto*
thus *Discr* (*Inl* ($cnt, \{\{\}\}, ve, \widehat{nb} \text{ } b \text{ } l$)) $\in \text{arg-poss } p$
unfolding *arg-poss-def* **by** *auto*
next

Case If (true case)

fix *ve b l1 l2 v cntst cntsf cnt*

assume $Discr (Inl (PP (prim.If l1 l2), [v, cntst, cntsf], ve, b)) \in arg\text{-}poss\ p$
and $cnt \in cntst$
hence $cnt \in proc\text{-}poss\ p$
and $ve \in smaps\text{-}over (vars\ p \times UNIV) (proc\text{-}poss\ p)$
unfolding $arg\text{-}poss\text{-}def\ fstate\text{-}poss\text{-}def\ NList\text{-}def\ nList\text{-}def$
by $auto$
moreover
have $\square \in NList (Pow (proc\text{-}poss\ p)) (call\text{-}list\text{-}lengths\ p)$
unfolding $call\text{-}list\text{-}lengths\text{-}def\ NList\text{-}def\ nList\text{-}def$ **by** $auto$
ultimately
have $(cnt, \square, ve, \widehat{nb}\ b\ l1) \in fstate\text{-}poss\ p$
unfolding $fstate\text{-}poss\text{-}def$ **by** $auto$
thus $Discr (Inl (cnt, \square, ve, \widehat{nb}\ b\ l1)) \in arg\text{-}poss\ p$
unfolding $arg\text{-}poss\text{-}def$ **by** $auto$
next

Case If (false case)

fix $ve\ b\ l1\ l2\ v\ cntst\ cntsf\ cnt$
assume $Discr (Inl (PP (prim.If l1 l2), [v, cntst, cntsf], ve, b)) \in arg\text{-}poss\ p$
and $cnt \in cntsf$
hence $cnt \in proc\text{-}poss\ p$
and $ve \in smaps\text{-}over (vars\ p \times UNIV) (proc\text{-}poss\ p)$
unfolding $arg\text{-}poss\text{-}def\ fstate\text{-}poss\text{-}def\ NList\text{-}def\ nList\text{-}def$
by $auto$
moreover
have $\square \in NList (Pow (proc\text{-}poss\ p)) (call\text{-}list\text{-}lengths\ p)$
unfolding $call\text{-}list\text{-}lengths\text{-}def\ NList\text{-}def\ nList\text{-}def$ **by** $auto$
ultimately
have $(cnt, \square, ve, \widehat{nb}\ b\ l2) \in fstate\text{-}poss\ p$
unfolding $fstate\text{-}poss\text{-}def$ **by** $auto$
thus $Discr (Inl (cnt, \square, ve, \widehat{nb}\ b\ l2)) \in arg\text{-}poss\ p$
unfolding $arg\text{-}poss\text{-}def$ **by** $auto$
qed
next
case $(Inr\ cstate)$
show $?thesis\ \mathbf{proof}(cases\ cstate\ rule:\ prod\text{-}cases4)$
case $(fields\ c\ \beta\ ve\ b)$
show $?thesis\ \mathbf{using}\ Discr\ Inr\ fields\ state\ \mathbf{proof}(cases\ c,\ auto\ simp\ add:HOL.Let\text{-}def\ simp\ del:evalV\text{-}a.simps)$

Case App

fix $l\ d\ ds\ f$
assume $arg:\ Discr (Inr (App\ l\ d\ ds,\ \beta,\ ve,\ b)) \in arg\text{-}poss\ p$
and $f: f \in \widehat{A}\ d\ \beta\ ve$
hence $c: App\ l\ d\ ds \in calls\ p$
and $d: d \in vals\ p$
and $ds: set\ ds \subseteq vals\ p$

and β : $\beta \in \text{maps-over } (\text{labels } p) \text{ UNIV}$
and ve : $ve \in \text{smaps-over } (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)$
by (*auto simp add: arg-poss-def cstate-poss-def call-list-lengths-def dest: app1 app2*)

have len : $length \ ds \in \text{call-list-lengths } p$
by (*auto intro: rev-image-eqI[OF c] simp add: call-list-lengths-def*)

have $f \in \text{proc-poss } p$
using $f \ d \ ve \ \beta$ **by** (*rule evalV-possible*)
moreover
have $map \ (\lambda v. \widehat{\mathcal{A}} \ v \ \beta \ ve) \ ds \in \text{NList } (\text{Pow } (\text{proc-poss } p)) (\text{call-list-lengths } p)$
using $ds \ len$
unfolding *NList-def* **by** (*auto simp add:nList-def intro!: evalV-possible[OF - - ve beta]*)
ultimately
have $(f, \text{map } (\lambda v. \widehat{\mathcal{A}} \ v \ \beta \ ve) \ ds, ve, \widehat{nb} \ b \ l) \in \text{fstate-poss } p$ (**is** $?fstate \in -$)
using ve
unfolding *fstate-poss-def* **by** *simp*
thus $\text{Discr } (\text{Inl } ?fstate) \in \text{arg-poss } p$
unfolding *arg-poss-def* **by** *auto*

next

Case Let

fix $l \text{ binds } c'$
assume arg : $\text{Discr } (\text{Inr } (\text{Let } l \text{ binds } c', \beta, ve, b)) \in \text{arg-poss } p$
hence l : $l \in \text{labels } p$
and c' : $c' \in \text{calls } p$
and vars : $\text{fst } ' \text{ set binds } \subseteq \text{vars } p$
and ls : $\text{snd } ' \text{ set binds } \subseteq \text{lambdas } p$
and β : $\beta \in \text{maps-over } (\text{labels } p) \text{ UNIV}$
and ve : $ve \in \text{smaps-over } (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)$
by (*auto simp add: arg-poss-def cstate-poss-def call-list-lengths-def dest:let1 let2 let3 let4*)

have β' : $\beta(l \mapsto \widehat{nb} \ b \ l) \in \text{maps-over } (\text{labels } p) \text{ UNIV}$ (**is** $? \beta' \in -$)
by (*auto intro: maps-over-upd[OF beta l]*)

moreover
have $ve \cup. \bigcup_{\text{binds}} \text{map } (\lambda(v, lam). \{ (v, \widehat{nb} \ b \ l) := \widehat{\mathcal{A}} \ (L \ lam) \ (\beta(l \mapsto \widehat{nb} \ b \ l)) \ ve \}.)$
 $\in \text{smaps-over } (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)$ (**is** $?ve' \in -$)
using $\text{vars } ls \ \beta'$
by (*auto intro!: smaps-over-un[OF ve] smaps-over-Union*)
(auto intro!:smaps-over-singleton simp add: proc-poss-def)

ultimately
have $(c', ? \beta', ?ve', \widehat{nb} \ b \ l) \in \text{cstate-poss } p$ (**is** $?cstate \in -$)
using c' **unfolding** *cstate-poss-def* **by** *simp*


```

thus Discr (Inr ?cstate) ∈ arg-poss p
  unfolding arg-poss-def by auto
qed
qed
qed
qed
qed

```

This result is now lifted to the powerset of *abs-R*.

```

lemma arg-space-complete-ps: states ⊆ arg-poss p ⇒ (abs-R).states ⊆ arg-poss p
using arg-space-complete unfolding powerset-lift-def by auto

```

We are not so much interested in the finiteness of the set of possible arguments but rather of the the set of occurring arguments, when we start with the initial argument. But as this is of course a subset of the set of possible arguments, this is not hard to show.

```

lemma UN-iterate-less:
  assumes start: x ∈ S
  and step: ⋀y. y ⊆ S ⇒ (f.y) ⊆ S
  shows  $(\bigcup i. \text{iterate } i.f.\{x\}) \subseteq S$ 
proof - {
  fix i
  have iterate i.f.\{x\} ⊆ S
  proof (induct i)
    case 0 show ?case using  $\langle x \in S \rangle$  by simp next
    case (Suc i) thus ?case using step[of iterate i.f.\{x\}] by simp
  qed
  } thus ?thesis by auto
qed

```

```

lemma args-finite: finite (⋃ i. iterate i.(abs-R).\{initial-r p\}) (is finite ?S)
proof (rule finite-subset[OF -finite-arg-space])
  have [simp]: p ∈ lambdas p by (cases p, simp)
  show ?S ⊆ arg-poss p
  unfolding initial-r-def
  by (rule UN-iterate-less[OF -arg-space-complete-ps])
    (auto simp add:arg-poss-def fstate-poss-def proc-poss-def call-list-lengths-def NList-def nList-def
    intro!: imageI)
qed

```

8.2. A decomposition

The functions *abs-g* and *abs-R* are derived from $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$. This connection has yet to be expressed explicitly.

lemma *Un-commute-helper*: $(a \cup b) \cup (c \cup d) = (a \cup c) \cup (b \cup d)$
by *auto*

lemma *a-evalF-decomp*:

$\widehat{F} = \text{fst } (\text{sum-to-tup} \cdot (\text{fix} \cdot (\Lambda f x. (\bigcup y \in \text{abs-R} \cdot x. f \cdot y) \cup \text{abs-g} \cdot x)))$
apply (*subst a-evalF-def*)
apply (*subst fix-transform-pair-sum*)
apply (*rule arg-cong [of - - $\lambda x. \text{fst } (\text{sum-to-tup} \cdot (\text{fix} \cdot x))$]*)
apply (*simp*)
apply (*simp only: discr-app undiscr-Discr*)
apply (*rule cfun-eqI, rule cfun-eqI, simp*)
apply (*case-tac xa, rename-tac a, case-tac a, simp*)
apply (*case-tac aa rule:a-fstate-case, simp-all add: Un-commute-helper*)
apply (*case-tac b rule:prod-cases4*)
apply (*case-tac aa*)
apply (*simp-all add:HOL.Let-def*)
done

8.3. The iterative equation

Because of the special form of \widehat{F} (and thus $\widehat{\mathcal{PR}}$) derived in the previous lemma, we can apply our generic results from *Shivers-CFA.Computability* and express the abstract semantics as the image of a finite set under a computable function.

lemma *a-evalF-iterative*:

$\widehat{F} \cdot (\text{Discr } x) = \underline{\text{abs-g}} \cdot (\bigcup i. \text{iterate } i \cdot (\underline{\text{abs-R}}) \cdot \{\text{Discr } (\text{Inl } x)\})$
by (*simp del:abs-R.simps abs-g.simps add: theorem12 Un-commute a-evalF-decomp*)

lemma *a-evalCPS-iterative*:

$\widehat{\mathcal{PR}} \text{ prog} = \underline{\text{abs-g}} \cdot (\bigcup i. \text{iterate } i \cdot (\underline{\text{abs-R}}) \cdot \{\text{initial-r prog}\})$
unfolding *evalCPS-a-def* **and** *initial-r-def*
by (*subst a-evalF-iterative, simp del:abs-R.simps abs-g.simps evalV-a.simps*)

end

Part III.

The auxiliary theories

9. Syntax tree helpers

theory *CPSUtils*
imports *CPSScheme*
begin

This theory defines the sets *lambdas p*, *calls p*, *calls p*, *vars p*, *labels p* and *prims p* as the subexpressions of the program *p*. Finiteness is shown for each of these sets, and some rules about how these sets relate. All these rules are proven more or less the same ways, which is very inelegant due to the nesting of the type and the shape of the derived induction rule.

It would be much nicer to start with these rules and define the set inductively. Unfortunately, that approach would make it very hard to show the finiteness of the sets in question.

```

fun lambdas :: lambda  $\Rightarrow$  lambda set
and lambdasC :: call  $\Rightarrow$  lambda set
and lambdasV :: val  $\Rightarrow$  lambda set
where lambdas (Lambda l vs c) = ({Lambda l vs c}  $\cup$  lambdasC c)
      | lambdasC (App l d ds) = lambdasV d  $\cup$   $\bigcup$  (lambdasV ' set ds)
      | lambdasC (Let l binds c') = ( $\bigcup_{(-, y) \in \text{set binds.}} \text{lambdas } y$ )  $\cup$  lambdasC c'
      | lambdasV (L l) = lambdas l
      | lambdasV - = {}

```

```

fun calls :: lambda  $\Rightarrow$  call set
and callsC :: call  $\Rightarrow$  call set
and callsV :: val  $\Rightarrow$  call set
where calls (Lambda l vs c) = callsC c
      | callsC (App l d ds) = {App l d ds}  $\cup$  callsV d  $\cup$  ( $\bigcup$  (callsV ' (set ds)))
      | callsC (Let l binds c') = {call.Let l binds c'}  $\cup$  ( $\bigcup_{(-, y) \in \text{set binds.}} \text{calls } y$ )  $\cup$  callsC c'
      | callsV (L l) = calls l
      | callsV - = {}

```

lemma *finite-lambdas[simp]*: *finite (lambdas l) and finite (lambdasC c) finite (lambdasV v)*
by (*induct rule: lambdas-lambdasC-lambdasV.induct, auto*)

lemma *finite-calls[simp]*: *finite (calls l) and finite (callsC c) finite (callsV v)*
by (*induct rule: calls-callsC-callsV.induct, auto*)

```

fun vars :: lambda  $\Rightarrow$  var set
and varsC :: call  $\Rightarrow$  var set
and varsV :: val  $\Rightarrow$  var set
where vars (Lambda - vs c) = set vs  $\cup$  varsC c
      | varsC (App - a as) = varsV a  $\cup$   $\bigcup$  (varsV ' (set as))
      | varsC (Let - binds c') = ( $\bigcup_{(v, l) \in \text{set binds.}} \{v\} \cup \text{vars } l$ )  $\cup$  varsC c'
      | varsV (L l) = vars l
      | varsV (R - v) = {v}
      | varsV - = {}

```

lemma *finite-vars[simp]*: *finite (vars l) and finite (varsC c) finite (varsV v)*
by (*induct rule: vars-varsC-varsV.induct, auto*)

```

fun label :: lambda + call  $\Rightarrow$  label
where label (Inl (Lambda l -)) = l

```

| $label (Inr (App l -)) = l$
| $label (Inr (Let l -)) = l$

fun $labels :: lambda \Rightarrow label\ set$
and $labelsC :: call \Rightarrow label\ set$
and $labelsV :: val \Rightarrow label\ set$
where $labels (Lambda\ l\ vs\ c) = \{l\} \cup labelsC\ c$
| $labelsC (App\ l\ a\ as) = \{l\} \cup labelsV\ a \cup \bigcup (labelsV\ ' (set\ as))$
| $labelsC (Let\ l\ binds\ c') = \{l\} \cup (\bigcup_{(v, y) \in set\ binds.} labels\ y) \cup labelsC\ c'$
| $labelsV (L\ l) = labels\ l$
| $labelsV (R\ l\ -) = \{l\}$
| $labelsV - = \{\}$

lemma $finite-labels[simp]$: $finite (labels\ l)$ **and** $finite (labelsC\ c)$ $finite (labelsV\ v)$
by (*induct rule: labels-labelsC-labelsV.induct, auto*)

fun $prims :: lambda \Rightarrow prim\ set$
and $primsC :: call \Rightarrow prim\ set$
and $primsV :: val \Rightarrow prim\ set$
where $prims (Lambda\ -\ vs\ c) = primsC\ c$
| $primsC (App\ -\ a\ as) = primsV\ a \cup \bigcup (primsV\ ' (set\ as))$
| $primsC (Let\ -\ binds\ c') = (\bigcup_{(-, y) \in set\ binds.} prims\ y) \cup primsC\ c'$
| $primsV (L\ l) = prims\ l$
| $primsV (R\ l\ v) = \{\}$
| $primsV (P\ prim) = \{prim\}$
| $primsV (C\ l\ v) = \{\}$

lemma $finite-prims[simp]$: $finite (prims\ l)$ **and** $finite (primsC\ c)$ $finite (primsV\ v)$
by (*induct rule: labels-labelsC-labelsV.induct, auto*)

fun $vals :: lambda \Rightarrow val\ set$
and $valsC :: call \Rightarrow val\ set$
and $valsV :: val \Rightarrow val\ set$
where $vals (Lambda\ -\ vs\ c) = valsC\ c$
| $valsC (App\ -\ a\ as) = valsV\ a \cup \bigcup (valsV\ ' (set\ as))$
| $valsC (Let\ -\ binds\ c') = (\bigcup_{(-, y) \in set\ binds.} vals\ y) \cup valsC\ c'$
| $valsV (L\ l) = \{L\ l\} \cup vals\ l$
| $valsV (R\ l\ v) = \{R\ l\ v\}$
| $valsV (P\ prim) = \{P\ prim\}$
| $valsV (C\ l\ v) = \{C\ l\ v\}$

lemma

fixes $list2 :: (var \times lambda)\ list$ **and** $t :: var \times lambda$
shows $lambdas1: Lambda\ l\ vs\ c \in lambdas\ x \Longrightarrow c \in calls\ x$
and $Lambda\ l\ vs\ c \in lambdasC\ y \Longrightarrow c \in callsC\ y$
and $Lambda\ l\ vs\ c \in lambdasV\ z \Longrightarrow c \in callsV\ z$
and $\forall z \in set\ list. Lambda\ l\ vs\ c \in lambdasV\ z \longrightarrow c \in callsV\ z$
and $\forall x \in set\ list2. Lambda\ l\ vs\ c \in lambdas\ (snd\ x) \longrightarrow c \in calls\ (snd\ x)$
and $Lambda\ l\ vs\ c \in lambdas\ (snd\ t) \Longrightarrow c \in calls\ (snd\ t)$

apply (*induct rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*case-tac c, auto*)[1]
apply (*rule-tac x=((a, b), ba) in beXI, auto*)
done

lemma

shows *lambdas2: Lambda l vs c ∈ lambdas x ⇒ l ∈ labels x*
and *Lambda l vs c ∈ lambdasC y ⇒ l ∈ labelsC y*
and *Lambda l vs c ∈ lambdasV z ⇒ l ∈ labelsV z*
and $\forall z \in \text{set list. } \text{Lambda } l \text{ vs } c \in \text{lambdasV } z \longrightarrow l \in \text{labelsV } z$
and $\forall x \in \text{set (list2 :: (var } \times \text{ lambda) list) . } \text{Lambda } l \text{ vs } c \in \text{lambdas (snd } x) \longrightarrow l \in \text{labels (snd } x)$
and *Lambda l vs c ∈ lambdas (snd (t:: var×lambda)) ⇒ l ∈ labels (snd t)*
apply (*induct rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*rule-tac x=((a, b), ba) in beXI, auto*)
done

lemma

shows *lambdas3: Lambda l vs c ∈ lambdas x ⇒ set vs ⊆ vars x*
and *Lambda l vs c ∈ lambdasC y ⇒ set vs ⊆ varsC y*
and *Lambda l vs c ∈ lambdasV z ⇒ set vs ⊆ varsV z*
and $\forall z \in \text{set list. } \text{Lambda } l \text{ vs } c \in \text{lambdasV } z \longrightarrow \text{set } vs \subseteq \text{varsV } z$
and $\forall x \in \text{set (list2 :: (var } \times \text{ lambda) list) . } \text{Lambda } l \text{ vs } c \in \text{lambdas (snd } x) \longrightarrow \text{set } vs \subseteq \text{vars (snd } x)$
and *Lambda l vs c ∈ lambdas (snd (t:: var×lambda)) ⇒ set vs ⊆ vars (snd t)*
apply (*induct x and y and z and list and list2 and t rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*erule-tac x=((aa, ba), bb) in ballE*)
apply (*rule-tac x=((aa, ba), bb) in beXI, auto*)
done

lemma

shows *app1: App l d ds ∈ calls x ⇒ d ∈ vals x*
and *App l d ds ∈ callsC y ⇒ d ∈ valsC y*
and *App l d ds ∈ callsV z ⇒ d ∈ valsV z*
and $\forall z \in \text{set list. } \text{App } l \text{ d } ds \in \text{callsV } z \longrightarrow d \in \text{valsV } z$
and $\forall x \in \text{set (list2 :: (var } \times \text{ lambda) list) . } \text{App } l \text{ d } ds \in \text{calls (snd } x) \longrightarrow d \in \text{vals (snd } x)$
and *App l d ds ∈ calls (snd (t:: var×lambda)) ⇒ d ∈ vals (snd t)*
apply (*induct x and y and z and list and list2 and t rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*case-tac d, auto*)
apply (*erule-tac x=((a, b), ba) in ballE*)
apply (*rule-tac x=((a, b), ba) in beXI, auto*)
done

lemma

shows *app2: App l d ds ∈ calls x ⇒ set ds ⊆ vals x*

and $App\ l\ d\ ds \in callsC\ y \implies set\ ds \subseteq valsC\ y$
and $App\ l\ d\ ds \in callsV\ z \implies set\ ds \subseteq valsV\ z$
and $\forall z \in set\ list. App\ l\ d\ ds \in callsV\ z \longrightarrow set\ ds \subseteq valsV\ z$
and $\forall x \in set\ (list2 :: (var \times lambda)\ list) . App\ l\ d\ ds \in calls\ (snd\ x) \longrightarrow set\ ds \subseteq vals\ (snd\ x)$
and $App\ l\ d\ ds \in calls\ (snd\ (t :: var \times lambda)) \implies set\ ds \subseteq vals\ (snd\ t)$
apply (*induct* x **and** y **and** z **and** $list$ **and** $list2$ **and** t *rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*case-tac* x , *auto*)
apply (*erule-tac* $x=((a, b), ba)$ **in** *ballE*)
apply (*rule-tac* $x=((a, b), ba)$ **in** *beXI*, *auto*)
done

lemma

shows *let1: Let* l *binds* $c' \in calls\ x \implies l \in labels\ x$
and *Let* l *binds* $c' \in callsC\ y \implies l \in labelsC\ y$
and *Let* l *binds* $c' \in callsV\ z \implies l \in labelsV\ z$
and $\forall z \in set\ list. Let\ l\ binds\ c' \in callsV\ z \longrightarrow l \in labelsV\ z$
and $\forall x \in set\ (list2 :: (var \times lambda)\ list) . Let\ l\ binds\ c' \in calls\ (snd\ x) \longrightarrow l \in labels\ (snd\ x)$
and *Let* l *binds* $c' \in calls\ (snd\ (t :: var \times lambda)) \implies l \in labels\ (snd\ t)$
apply (*induct* x **and** y **and** z **and** $list$ **and** $list2$ **and** t *rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*erule-tac* $x=((a, b), ba)$ **in** *ballE*)
apply (*rule-tac* $x=((a, b), ba)$ **in** *beXI*, *auto*)
done

lemma

shows *let2: Let* l *binds* $c' \in calls\ x \implies c' \in calls\ x$
and *Let* l *binds* $c' \in callsC\ y \implies c' \in callsC\ y$
and *Let* l *binds* $c' \in callsV\ z \implies c' \in callsV\ z$
and $\forall z \in set\ list. Let\ l\ binds\ c' \in callsV\ z \longrightarrow c' \in callsV\ z$
and $\forall x \in set\ (list2 :: (var \times lambda)\ list) . Let\ l\ binds\ c' \in calls\ (snd\ x) \longrightarrow c' \in calls\ (snd\ x)$
and *Let* l *binds* $c' \in calls\ (snd\ (t :: var \times lambda)) \implies c' \in calls\ (snd\ t)$
apply (*induct* x **and** y **and** z **and** $list$ **and** $list2$ **and** t *rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*case-tac* c' , *auto*)
apply (*erule-tac* $x=((a, b), ba)$ **in** *ballE*)
apply (*rule-tac* $x=((a, b), ba)$ **in** *beXI*, *auto*)
done

lemma

shows *let3: Let* l *binds* $c' \in calls\ x \implies fst\ 'set\ binds \subseteq vars\ x$
and *Let* l *binds* $c' \in callsC\ y \implies fst\ 'set\ binds \subseteq varsC\ y$
and *Let* l *binds* $c' \in callsV\ z \implies fst\ 'set\ binds \subseteq varsV\ z$
and $\forall z \in set\ list. Let\ l\ binds\ c' \in callsV\ z \longrightarrow fst\ 'set\ binds \subseteq varsV\ z$
and $\forall x \in set\ (list2 :: (var \times lambda)\ list) . Let\ l\ binds\ c' \in calls\ (snd\ x) \longrightarrow fst\ 'set\ binds \subseteq vars\ (snd\ x)$

and *Let l binds $c' \in \text{calls} (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{fst } ' \text{ set binds } \subseteq \text{vars} (\text{snd } t)$*
apply (*induct x and y and z and $list$ and $list2$ and t rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply *fastforce*
done

lemma

shows *let4: Let l binds $c' \in \text{calls } x \implies \text{snd } ' \text{ set binds } \subseteq \text{lambdas } x$*
and *Let l binds $c' \in \text{calls} C y \implies \text{snd } ' \text{ set binds } \subseteq \text{lambdas} C y$*
and *Let l binds $c' \in \text{calls} V z \implies \text{snd } ' \text{ set binds } \subseteq \text{lambdas} V z$*
and $\forall z \in \text{set list. Let } l \text{ binds } c' \in \text{calls} V z \longrightarrow \text{snd } ' \text{ set binds } \subseteq \text{lambdas} V z$
and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . \text{Let } l \text{ binds } c' \in \text{calls} (\text{snd } x) \longrightarrow \text{snd } ' \text{ set binds } \subseteq \text{lambdas } (\text{snd } x)$
and *Let l binds $c' \in \text{calls} (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{snd } ' \text{ set binds } \subseteq \text{lambdas} (\text{snd } t)$*
apply (*induct x and y and z and $list$ and $list2$ and t rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*rule-tac $x=((a, b), ba)$ in bexI , auto*)
apply (*case-tac ba , auto*)
apply (*erule-tac $x=((aa, bb), bc)$ in ballE*)
apply (*rule-tac $x=((aa, bb), bc)$ in bexI , auto*)
done

lemma

shows *vals1: $P \text{ prim} \in \text{vals } p \implies \text{prim} \in \text{prims } p$*
and *$P \text{ prim} \in \text{vals} C y \implies \text{prim} \in \text{prims} C y$*
and *$P \text{ prim} \in \text{vals} V z \implies \text{prim} \in \text{prims} V z$*
and $\forall z \in \text{set list. } P \text{ prim} \in \text{vals} V z \longrightarrow \text{prim} \in \text{prims} V z$
and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . P \text{ prim} \in \text{vals} (\text{snd } x) \longrightarrow \text{prim} \in \text{prims} (\text{snd } x)$
and *$P \text{ prim} \in \text{vals} (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{prim} \in \text{prims} (\text{snd } t)$*
apply (*induct rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*erule-tac $x=((a, b), ba)$ in ballE*)
apply (*rule-tac $x=((a, b), ba)$ in bexI , auto*)
done

lemma

shows *vals2: $R l \text{ var} \in \text{vals } p \implies \text{var} \in \text{vars } p$*
and *$R l \text{ var} \in \text{vals} C y \implies \text{var} \in \text{vars} C y$*
and *$R l \text{ var} \in \text{vals} V z \implies \text{var} \in \text{vars} V z$*
and $\forall z \in \text{set list. } R l \text{ var} \in \text{vals} V z \longrightarrow \text{var} \in \text{vars} V z$
and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . R l \text{ var} \in \text{vals} (\text{snd } x) \longrightarrow \text{var} \in \text{vars} (\text{snd } x)$
and *$R l \text{ var} \in \text{vals} (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{var} \in \text{vars} (\text{snd } t)$*
apply (*induct rule:mutual-lambda-call-var-inducts*)
apply *auto*
apply (*erule-tac $x=((a, b), ba)$ in ballE*)
apply (*rule-tac $x=((a, b), ba)$ in bexI , auto*)
done

```

lemma
shows vals3:  $L\ l \in \text{vals}\ p \implies l \in \text{lambdas}\ p$ 
  and  $L\ l \in \text{valsC}\ y \implies l \in \text{lambdasC}\ y$ 
  and  $L\ l \in \text{valsV}\ z \implies l \in \text{lambdasV}\ z$ 
  and  $\forall z \in \text{set list. } L\ l \in \text{valsV}\ z \longrightarrow l \in \text{lambdasV}\ z$ 
  and  $\forall x \in \text{set (list2 :: (var } \times \text{ lambda) list) . } L\ l \in \text{vals (snd } x) \longrightarrow l \in \text{lambdas (snd } x)$ 
  and  $L\ l \in \text{vals (snd (t:: var} \times \text{ lambda))} \implies l \in \text{lambdas (snd } t)$ 
apply (induct rule:mutual-lambda-call-var-inducts)
apply auto
apply (erule-tac x=((a, b), ba) in ballE)
apply (rule-tac x=((a, b), ba) in beXI, auto)
apply (case-tac l, auto)
done

```

```

definition nList :: 'a set => nat => 'a list set
where  $nList\ A\ n \equiv \{l. \text{set } l \leq A \wedge \text{length } l = n\}$ 

```

```

lemma finite-nList[intro]:
  assumes fnA: finite A
  shows finite (nList A n)
proof (induct n)
case 0 thus ?case by (simp add:nList-def) next
case (Suc n) hence finn: finite (nList (A) n) by simp
  have  $nList\ A\ (Suc\ n) = (\text{case-prod } (\#))\ ' (A \times nList\ A\ n)\ (\text{is } ?lhs = ?rhs)$ 
  proof (rule subset-antisym[OF subsetI subsetI])
  fix l assume  $l \in ?lhs$  thus  $l \in ?rhs$ 
    by (cases l, auto simp add:nList-def)
  next
  fix l assume  $l \in ?rhs$  thus  $l \in ?lhs$ 
    by (auto simp add:nList-def)
  qed
thus finite ?lhs using fnA and finn
  by auto
qed

```

```

definition NList :: 'a set => nat set => 'a list set
where  $NList\ A\ N \equiv \bigcup_{n \in N. nList\ A\ n}$ 

```

```

lemma finite-Nlist[intro]:
   $\llbracket \text{finite } A; \text{finite } N \rrbracket \implies \text{finite } (NList\ A\ N)$ 
unfolding NList-def by auto

```

```

definition call-list-lengths
  where  $\text{call-list-lengths}\ p = \{0,1,2,3\} \cup (\lambda c. \text{case } c \text{ of (App } - - \text{ ds)} \Rightarrow \text{length } ds \mid - \Rightarrow 0)$  '
  calls p

```

```

lemma finite-call-list-lengths[simp]: finite (call-list-lengths p)
  unfolding call-list-lengths-def by auto

```


end

10. General utility lemmas

theory *Utils* imports *Main*
begin

This is a potpourri of various lemmas not specific to our project. Some of them could very well be included in the default Isabelle library.

Lemmas about the *single-valued* predicate.

lemma *single-valued-empty*[*simp*]:*single-valued* {}
by (*rule single-valuedI*) *auto*

lemma *single-valued-insert*:
 assumes *single-valued rel*
 and $\bigwedge x y . \llbracket (x,y) \in \text{rel}; x=a \rrbracket \implies y = b$
 shows *single-valued* (*insert* (*a,b*) *rel*)
using *assms*
by (*auto intro:single-valuedI dest:single-valuedD*)

Lemmas about *ran*, the range of a finite map.

lemma *ran-upd*: $\text{ran } (m (k \mapsto v)) \subseteq \text{ran } m \cup \{v\}$
unfolding *ran-def* **by** *auto*

lemma *ran-map-of*: $\text{ran } (\text{map-of } xs) \subseteq \text{snd } ` \text{set } xs$
by (*induct xs*)(*auto simp del:fun-upd-apply dest: ran-upd[THEN subsetD]*)

lemma *ran-concat*: $\text{ran } (m1 ++ m2) \subseteq \text{ran } m1 \cup \text{ran } m2$
unfolding *ran-def*
by *auto*

lemma *ran-upds*:
 assumes *eq-length: length ks = length vs*
 shows $\text{ran } (\text{map-upds } m \text{ ks } vs) \subseteq \text{ran } m \cup \text{set } vs$
proof–
 have $\text{ran } (\text{map-upds } m \text{ ks } vs) \subseteq \text{ran } (m ++ \text{map-of } (\text{rev } (\text{zip } \text{ks } \text{vs})))$
 unfolding *map-upds-def* **by** *simp*
 also have $\dots \subseteq \text{ran } m \cup \text{ran } (\text{map-of } (\text{rev } (\text{zip } \text{ks } \text{vs})))$ **by** (*rule ran-concat*)
 also have $\dots \subseteq \text{ran } m \cup \text{snd } ` \text{set } (\text{rev } (\text{zip } \text{ks } \text{vs}))$
 by (*intro Un-mono[of ran m ran m] subset-refl ran-map-of*)
 also have $\dots \subseteq \text{ran } m \cup \text{set } vs$
 by (*auto intro:Un-mono[of ran m ran m] subset-refl simp del:set-map simp add:set-map[THEN sym] map-snd-zip[OF eq-length]*)
 finally show *?thesis* .

qed

lemma *ran-upd-mem*[simp]: $v \in \text{ran } (m (k \mapsto v))$
unfolding *ran-def* **by** *auto*

Lemmas about *map*, *zip* and *fst/snd*

lemma *map-fst-zip*: $\text{length } xs = \text{length } ys \implies \text{map } \text{fst } (\text{zip } xs \ ys) = xs$
apply (*induct xs ys rule:list-induct2*) **by** *auto*

lemma *map-snd-zip*: $\text{length } xs = \text{length } ys \implies \text{map } \text{snd } (\text{zip } xs \ ys) = ys$
apply (*induct xs ys rule:list-induct2*) **by** *auto*

end

11. Set-valued maps

theory *SetMap*
imports *Main*
begin

For the abstract semantics, we need methods to work with set-valued maps, i.e. functions from a key type to sets of values. For this type, some well known operations are introduced and properties shown, either borrowing the nomenclature from finite maps (*sdom*, *sran*,...) or of sets (*{}*., \cup ,...).

definition

sdom :: $('a \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ set}$ **where**
sdom *m* = $\{a. m \ a \ \sim = \{\}\}$

definition

sran :: $('a \Rightarrow 'b \text{ set}) \Rightarrow 'b \text{ set}$ **where**
sran *m* = $\{b. \exists a. b \in m \ a\}$

lemma *sranI*: $b \in m \ a \implies b \in \text{sran } m$
by(*auto simp: sran-def*)

lemma *sdom-not-mem*[*elim*]: $a \notin \text{sdom } m \implies m \ a = \{\}$
by (*auto simp: sdom-def*)

definition *smap-empty* (*{}*.)
where *{}*. *k* = $\{\}$

definition *smap-union* :: $('a::\text{type} \Rightarrow 'b::\text{type} \text{ set}) \Rightarrow ('a \Rightarrow 'b \text{ set}) \Rightarrow ('a \Rightarrow 'b \text{ set}) \ (- \cup. -)$
where *smap1* $\cup.$ *smap2* *k* = *smap1* *k* \cup *smap2* *k*

primrec *smap-Union* :: $('a::\text{type} \Rightarrow 'b::\text{type} \text{ set}) \ \text{list} \Rightarrow 'a \Rightarrow 'b \text{ set} \ (\cup.-)$
where [*simp*]: $\cup. [] = \{\}$.

$$| \bigcup. (m \# ms) = m \cup. \bigcup. ms$$

definition *smap-singleton* :: 'a::type \Rightarrow 'b::type set \Rightarrow 'a \Rightarrow 'b set ($\{ - := - \}$.)
where $\{k := vs\}. = \{ \}$. ($k := vs$)

definition *smap-less* :: ('a \Rightarrow 'b set) \Rightarrow ('a \Rightarrow 'b set) \Rightarrow bool (-/ \subseteq . - [50, 51] 50)
where *smap-less* m1 m2 = ($\forall k. m1 k \subseteq m2 k$)

lemma *sdom-empty[simp]*: *sdom* $\{ \}$. = $\{ \}$
unfolding *sdom-def smap-empty-def* **by** *auto*

lemma *sdom-singleton[simp]*: *sdom* $\{k := vs\}. \subseteq \{k\}$
by (*auto simp add:sdom-def smap-singleton-def smap-empty-def*)

lemma *sran-singleton[simp]*: *sran* $\{k := vs\}. = vs$
by (*auto simp add:sran-def smap-singleton-def smap-empty-def*)

lemma *sran-empty[simp]*: *sran* $\{ \}$. = $\{ \}$
unfolding *sran-def smap-empty-def* **by** *auto*

lemma *sdom-union[simp]*: *sdom* ($m \cup. n$) = *sdom* $m \cup$ *sdom* n
by(*auto simp add:smap-union-def sdom-def*)

lemma *sran-union[simp]*: *sran* ($m \cup. n$) = *sran* $m \cup$ *sran* n
by(*auto simp add:smap-union-def sran-def*)

lemma *smap-empty[simp]*: $\{ \}$. \subseteq . $\{ \}$.
unfolding *smap-less-def* **by** *auto*

lemma *smap-less-refl*: $m \subseteq. m$
unfolding *smap-less-def* **by** *simp*

lemma *smap-less-trans[trans]*: $\llbracket m1 \subseteq. m2; m2 \subseteq. m3 \rrbracket \Longrightarrow m1 \subseteq. m3$
unfolding *smap-less-def* **by** *auto*

lemma *smap-union-mono*: $\llbracket ve1 \subseteq. ve1'; ve2 \subseteq. ve2' \rrbracket \Longrightarrow ve1 \cup. ve2 \subseteq. ve1' \cup. ve2'$
by (*auto simp add:smap-less-def smap-union-def*)

lemma *smap-Union-union*: $m1 \cup. \bigcup. ms = \bigcup.(m1 \# ms)$
by (*rule ext, auto simp add: smap-union-def smap-Union-def*)

lemma *smap-Union-mono*:
assumes *list-all2 smap-less ms1 ms2*
shows $\bigcup. ms1 \subseteq. \bigcup. ms2$
using *assms*
by(*induct rule:list-induct2[OF list-all2-lengthD[OF assms]]*)
(*auto intro:smap-union-mono*)

lemma *smap-singleton-mono*: $v \subseteq v' \Longrightarrow \{k := v\}. \subseteq. \{k := v'\}$.

by (*auto simp add: smap-singleton-def smap-less-def*)

lemma *smap-union-comm*: $m1 \cup. m2 = m2 \cup. m1$

by (*rule ext, auto simp add: smap-union-def*)

lemma *smap-union-empty1*[*simp*]: $\{\}. \cup. m = m$

by(*rule ext, auto simp add: smap-union-def smap-empty-def*)

lemma *smap-union-empty2*[*simp*]: $m \cup. \{\}. = m$

by(*rule ext, auto simp add: smap-union-def smap-empty-def*)

lemma *smap-union-assoc* [*simp*]: $(m1 \cup. m2) \cup. m3 = m1 \cup. (m2 \cup. m3)$

by (*rule ext, auto simp add: smap-union-def*)

lemma *smap-Union-append*[*simp*]: $\bigcup. (m1 @ m2) = (\bigcup. m1) \cup. (\bigcup. m2)$

by (*induct m1*) *auto*

lemma *smap-Union-rev*[*simp*]: $\bigcup. (\text{rev } l) = \bigcup. l$

by(*induct l*)(*auto simp add: smap-union-comm*)

lemma *smap-Union-map-rev*[*simp*]: $\bigcup. (\text{map } f (\text{rev } l)) = \bigcup. (\text{map } f l)$

by(*subst rev-map[THEN sym], subst smap-Union-rev, rule refl*)

end

12. Sets of maps

theory *MapSets*

imports *SetMap Utils*

begin

In the section about the finiteness of the argument space, we need the fact that the set of maps from a finite domain to a finite range is finite, and the same for the set-valued maps defined in *Shivers-CFA.SetMap*. Both these sets are defined (*maps-over*, *smaps-over*) and the finiteness is shown.

definition *maps-over* :: $'a::\text{type set} \Rightarrow 'b::\text{type set} \Rightarrow ('a \rightarrow 'b) \text{ set}$

where *maps-over* $A B = \{m. \text{dom } m \subseteq A \wedge \text{ran } m \subseteq B\}$

lemma *maps-over-empty*[*simp*]:

Map.empty \in *maps-over* $A B$

unfolding *maps-over-def* **by** *simp*

lemma *maps-over-upd*:

assumes $m \in$ *maps-over* $A B$

and $v \in A$ **and** $k \in B$

shows $m(v \mapsto k) \in$ *maps-over* $A B$

using *assms* **unfolding** *maps-over-def*

by (auto dest: subsetD[OF ran-upd])

lemma *maps-over-finite*[intro]:

assumes *finite A* and *finite B* shows *finite (maps-over A B)*

proof–

have *inj-map-graph*: *inj* ($\lambda f. \{(x, y). \text{Some } y = f x\}$)

proof (*induct rule: inj-onI*)

case (1 *x y*)

from 1.hyps(3) have *hyp*: $\bigwedge a b. (\text{Some } b = x a) \longleftrightarrow (\text{Some } b = y a)$

by (*simp add: set-eq-iff*)

show *?case*

proof (*rule ext*)

fix *z* **show** $x z = y z$

using *hyp*[*of - z*]

by (*cases x z, cases y z, auto*)

qed

qed

have ($\lambda f. \{(x, y). \text{Some } y = f x\}$) ‘ *maps-over A B* \subseteq *Pow(A \times B)* (**is** *?graph* \subseteq -)

unfolding *maps-over-def*

by (*auto dest!: subsetD[of - A] subsetD[of - B] intro: ranI*)

moreover

have *finite (Pow(A \times B))* **using** *assms* **by** *auto*

ultimately

have *finite ?graph* **by** (*rule finite-subset*)

thus *?thesis*

by (*rule finite-imageD[OF - subset-inj-on[OF inj-map-graph subset-UNIV]]*)

qed

definition *smaps-over* :: ‘*a*::*type set* \Rightarrow ‘*b*::*type set* \Rightarrow (‘*a* \Rightarrow ‘*b set*) *set*

where *smaps-over A B* = $\{m. \text{sdom } m \subseteq A \wedge \text{sran } m \subseteq B\}$

lemma *smaps-over-empty*[*simp*]:

$\{\}. \in \text{smaps-over } A B$

unfolding *smaps-over-def* **by** *simp*

lemma *smaps-over-singleton*:

assumes $k \in A$ and $vs \subseteq B$

shows $\{k := vs\}. \in \text{smaps-over } A B$

using *assms* **unfolding** *smaps-over-def*

by(*auto dest: subsetD[OF sdom-singleton]*)

lemma *smaps-over-un*:

assumes $m1 \in \text{smaps-over } A B$ and $m2 \in \text{smaps-over } A B$

shows $m1 \cup. m2 \in \text{smaps-over } A B$

using *assms* **unfolding** *smaps-over-def*

by (*auto simp add: smap-union-def*)

lemma *smaps-over-Union*:

```

assumes set ms  $\subseteq$  smaps-over A B
shows  $\bigcup$ .ms  $\in$  smaps-over A B
using assms
by (induct ms)(auto intro: smaps-over-un)

```

```

lemma smaps-over-im:
   $\llbracket f \in m\ a ; m \in \text{smaps-over}\ A\ B \rrbracket \implies f \in B$ 
unfolding smaps-over-def by (auto simp add: sran-def)

```

```

lemma smaps-over-finite[intro]:
  assumes finite A and finite B shows finite (smaps-over A B)
proof-
  have inj-smap-graph: inj ( $\lambda f. \{(x, y). y = f\ x \wedge y \neq \{\}\}$ ) (is inj ?gr)
  proof (induct rule: inj-onI)
    case (1 x y)
    from 1.hyps(3) have hyp:  $\bigwedge a\ b. (b = x\ a \wedge b \neq \{\}) = (b = y\ a \wedge b \neq \{\})$ 
      by -(subst (asm) (3) set-eq-iff, simp)
    show ?case
    proof (rule ext)
      fix z show x z = y z
      using hyp[of - z]
      by (cases x z  $\neq$  {}, cases y z  $\neq$  {}, auto)
    qed
  qed

```

```

have ?gr ' smaps-over A B  $\subseteq$  Pow( A  $\times$  Pow B ) (is ?graph  $\subseteq$  -)
  unfolding smaps-over-def
  by (auto dest!: subsetD[of - A] subsetD[of - Pow B] sdom-not-mem intro: sranI)
moreover
have finite (Pow( A  $\times$  Pow B )) using assms by auto
ultimately
have finite ?graph by (rule finite-subset)
thus ?thesis
  by (rule finite-imageD[OF - subset-inj-on[OF inj-smap-graph subset-UNIV]])
qed

```

end

13. HOLCF Utility lemmas

```

theory HOLCFUtils
imports HOLCF
begin

```

We use *HOLCF* to define the denotational semantics. By default, *HOLCF* does not turn the regular *set* type into a partial order, so this is done here. Some of the lemmas here are contributed by Brian Huffman.

We start by making the type *bool* a pointed chain-complete partial order.

```

instantiation bool :: po
begin
definition
   $x \sqsubseteq y \iff (x \longrightarrow y)$ 
instance by standard (unfold below-bool-def, fast+)
end

instance bool :: chfn
apply standard
apply (drule finite-range-imp-finch)
apply (rule finite)
apply (simp add: finite-chain-def)
done

instance bool :: pcpo
proof
  have  $\forall y. \text{False} \sqsubseteq y$  by (simp add: below-bool-def)
  thus  $\exists x::\text{bool}. \forall y. x \sqsubseteq y$  ..
qed

lemma is-lub-bool:  $S \ll\mid (\text{True} \in S)$ 
  unfolding is-lub-def is-ub-def below-bool-def by auto

lemma lub-bool:  $\text{lub } S = (\text{True} \in S)$ 
  using is-lub-bool by (rule lub-eqI)

lemma bottom-eq-False[simp]:  $\perp = \text{False}$ 
by (rule below-antisym [OF minimal], simp add: below-bool-def)

```

To convert between the squared syntax used by *HOLCF* and the regular, round syntax for sets, we state some of the equivalencies.

```

instantiation set :: (type) po
begin
definition
   $A \sqsubseteq B \iff A \subseteq B$ 
instance by standard (unfold below-set-def, fast+)
end

lemma sqsubset-is-subset:  $A \sqsubseteq B \iff A \subseteq B$ 
  by (fact below-set-def)

lemma is-lub-set:  $S \ll\mid \bigcup S$ 
  unfolding is-lub-def is-ub-def below-set-def by fast

lemma lub-is-union:  $\text{lub } S = \bigcup S$ 
  using is-lub-set by (rule lub-eqI)

```

instance *set* :: (type) *cpo*
 by *standard* (*fast intro: is-lub-set*)

lemma *emptyset-is-bot*[*simp*]: $\{\} \sqsubseteq S$
 by (*simp add: sqsubset-is-subset*)

instance *set* :: (type) *pcpo*
 by *standard* (*fast intro: emptyset-is-bot*)

lemma *bot-bool-is-emptyset*[*simp*]: $\perp = \{\}$
 using *emptyset-is-bot* by (*rule bottomI [symmetric]*)

To actually use these instance in *fixrec* definitions or fixed-point inductions, we need continuity requirements for various boolean and set operations.

lemma *cont2cont-disj* [*simp*, *cont2cont*]:
 assumes *f*: *cont* $(\lambda x. f x)$ and *g*: *cont* $(\lambda x. g x)$
 shows *cont* $(\lambda x. f x \vee g x)$
 apply (*rule cont-apply [OF f]*)
 apply (*rule chfindom-monofun2cont*)
 apply (*rule monofunI, simp add: below-bool-def*)
 apply (*rule cont-compose [OF - g]*)
 apply (*rule chfindom-monofun2cont*)
 apply (*rule monofunI, simp add: below-bool-def*)
 done

lemma *cont2cont-imp*[*simp*, *cont2cont*]:
 assumes *f*: *cont* $(\lambda x. \neg f x)$ and *g*: *cont* $(\lambda x. g x)$
 shows *cont* $(\lambda x. f x \longrightarrow g x)$
 unfolding *imp-conv-disj* by (*rule cont2cont-disj[OF f g]*)

lemma *cont2cont-Collect* [*simp*, *cont2cont*]:
 assumes $\bigwedge y. \text{cont } (\lambda x. f x y)$
 shows *cont* $(\lambda x. \{y. f x y\})$
 apply (*rule contI*)
 apply (*subst cont2contlubE [OF assms], assumption*)
 apply (*auto simp add: is-lub-def is-ub-def below-set-def lub-bool*)
 done

lemma *cont2cont-mem* [*simp*, *cont2cont*]:
 assumes *cont* $(\lambda x. f x)$
 shows *cont* $(\lambda x. y \in f x)$
 apply (*rule cont-compose [OF - assms]*)
 apply (*rule contI*)
 apply (*auto simp add: is-lub-def is-ub-def below-bool-def lub-is-union*)
 done

lemma *cont2cont-union* [*simp*, *cont2cont*]:
cont $(\lambda x. f x) \implies \text{cont } (\lambda x. g x)$

$\implies \text{cont } (\lambda x. f x \cup g x)$
unfolding *Un-def* **by** *simp*

lemma *cont2cont-insert* [*simp, cont2cont*]:
assumes $\text{cont } (\lambda x. f x)$
shows $\text{cont } (\lambda x. \text{insert } y (f x))$
unfolding *insert-def* **using** *assms*
by (*intro cont2cont*)

lemmas *adm-subset = adm-below*[**where** *?'b = 'a::type set, unfolded sqsubset-is-subset*]

lemma *cont2cont-UNION*[*cont2cont, simp*]:
assumes $\text{cont } f$
and $\bigwedge y. \text{cont } (\lambda x. g x y)$
shows $\text{cont } (\lambda x. \bigcup_{y \in f x} g x y)$
proof (*induct rule: contI2*[*case-names Mono Limit*])
case *Mono*
show *monofun* $(\lambda x. \bigcup_{y \in f x} g x y)$
by (*rule monofunI*)(*auto iff:sqsubset-is-subset dest: monofunE[OF assms(1)][THEN cont2mono]*)
monofunE[OF assms(2)][THEN cont2mono])
next
case (*Limit Y*)
have $(\bigcup_{y \in f} (\bigcup i. Y i). g (\bigcup j. Y j) y) \subseteq (\bigcup k. \bigcup_{y \in f} (Y k). g (Y k) y)$
proof
fix x **assume** $x \in (\bigcup_{y \in f} (\bigcup i. Y i). g (\bigcup j. Y j) y)$
then obtain y **where** $y \in f (\bigcup i. Y i)$ **and** $x \in g (\bigcup j. Y j) y$ **by** *auto*
hence $y \in (\bigcup i. f (Y i))$ **and** $x \in (\bigcup j. g (Y j) y)$ **by** (*auto simp add: cont2contlubE[OF assms(1) Limit(1)] cont2contlubE[OF assms(2) Limit(1)]*)
then obtain i **and** j **where** $y_i: y \in f (Y i)$ **and** $x_j: x \in g (Y j) y$ **by** (*auto simp add:lub-is-union*)
obtain k **where** $i \leq k$ **and** $j \leq k$ **by** (*erule-tac x = max i j in meta-allE*)*auto*
from y_i **and** x_j **have** $y \in f (Y k)$ **and** $x \in g (Y k) y$
using *monofunE[OF assms(1)][THEN cont2mono]*, *OF chain-mono[OF Limit(1) <i≤k>]*
and *monofunE[OF assms(2)][THEN cont2mono]*, *OF chain-mono[OF Limit(1) <j≤k>]*
by (*auto simp add:sqsubset-is-subset*)
hence $x \in (\bigcup_{y \in f} (Y k). g (Y k) y)$ **by** *auto*
thus $x \in (\bigcup k. \bigcup_{y \in f} (Y k). g (Y k) y)$ **by** (*auto simp add:lub-is-union*)
qed
thus *?case* **by** (*simp add:sqsubset-is-subset*)
qed

lemma *cont2cont-Let-simple*[*simp, cont2cont*]:
assumes $\text{cont } (\lambda x. g x t)$
shows $\text{cont } (\lambda x. \text{let } y = t \text{ in } g x y)$
unfolding *Let-def* **using** *assms* .

lemma *cont2cont-case-list* [*simp, cont2cont*]:
assumes $\bigwedge y. \text{cont } (\lambda x. f1 x)$
and $\bigwedge y z. \text{cont } (\lambda x. f2 x y z)$

```

  shows cont ( $\lambda x. \text{case-list } (f1\ x) (f2\ x)\ l$ )
using assms
by (cases l) auto

```

As with the continuity lemmas, we need admissibility lemmas.

```

lemma adm-not-mem:
  assumes cont ( $\lambda x. f\ x$ )
  shows adm ( $\lambda x. y \notin f\ x$ )
using assms
apply (erule-tac t = f in adm-subst)
proof (rule admI)
fix Y :: nat  $\Rightarrow$  'b set
assume chain: chain Y
assume  $\forall i. y \notin Y\ i$  hence ( $\bigsqcup i. y \in Y\ i$ ) = False
  by auto
thus  $y \notin (\bigsqcup i. Y\ i)$ 
  using chain unfolding lub-bool lub-is-union by auto
qed

```

```

lemma adm-id[simp]: adm ( $\lambda x. x$ )
by (rule adm-chfin)

```

```

lemma adm-Not[simp]: adm Not
by (rule adm-chfin)

```

```

lemma adm-prod-split:
  assumes adm ( $\lambda p. f\ (fst\ p)\ (snd\ p)$ )
  shows adm ( $\lambda(x,y). f\ x\ y$ )
using assms unfolding split-def .

```

```

lemma adm-ball':
  assumes  $\bigwedge y. adm\ (\lambda x. y \in A\ x \longrightarrow P\ x\ y)$ 
  shows adm ( $\lambda x. \forall y \in A\ x. P\ x\ y$ )
by (subst Ball-def, rule adm-all[OF assms])

```

```

lemma adm-not-conj:
   $\llbracket adm\ (\lambda x. \neg P\ x); adm\ (\lambda x. \neg Q\ x) \rrbracket \Longrightarrow adm\ (\lambda x. \neg (P\ x \wedge Q\ x))$ 
by simp

```

```

lemma adm-single-valued:
  assumes cont ( $\lambda x. f\ x$ )
  shows adm ( $\lambda x. \text{single-valued } (f\ x)$ )
using assms
unfolding single-valued-def
by (intro adm-lemmas adm-not-mem cont2cont adm-subst[of f])

```

To match Shivers' syntax we introduce the power-syntax for iterated function application.

abbreviation *niceiterate* ((-') [1000] 1000)
where *niceiterate* *f* *i* \equiv *iterate* *i*·*f*

end

14. Fixed point transformations

theory *FixTransform*
imports *HOLCF*
begin

default-sort *type*

In his treatment of the computability, Shivers gives proofs only for a generic example and leaves it to the reader to apply this to the mutually recursive functions used for the semantics. As we carry this out, we need to transform a fixed point for two functions (implemented in *HOLCF* as a fixed point over a tuple) to a simple fixed point equation. The approach here works as long as both functions in the tuple have the same return type, using the equation

$$X^A \cdot X^B = X^{A+B}.$$

Generally, a fixed point can be transformed using any retractable continuous function:

lemma *fix-transform*:
assumes $\bigwedge x. g \cdot (f \cdot x) = x$
shows $\text{fix} \cdot F = g \cdot (\text{fix} \cdot (f \text{ oo } F \text{ oo } g))$
using *assms* **apply** –
apply (*rule parallel-fix-ind*)
apply (*rule adm-eq*)
apply *auto*
apply (*erule retraction-strict*[*of* *g* *f*,*rule-format*])
done

The functions we use here convert a tuple of functions to a function taking a direct sum as parameters and back. We only care about discrete arguments here.

definition *tup-to-sum* :: (*'a* *discr* \rightarrow *'c*) \times (*'b* *discr* \rightarrow *'c*) \rightarrow (*'a* + *'b*) *discr* \rightarrow *'c*::*cpo*
where *tup-to-sum* = (Λ *p* *s*. (λ (*f*,*g*).
 case undiscr *s* *of* *Inl* *x* \Rightarrow *f*·(*Discr* *x*)
 | *Inr* *x* \Rightarrow *g*·(*Discr* *x*) *p*)

definition *sum-to-tup* :: ((*'a* + *'b*) *discr* \rightarrow *'c*) \rightarrow (*'a* *discr* \rightarrow *'c*) \times (*'b* *discr* \rightarrow *'c*::*cpo*)
where *sum-to-tup* = (Λ *f*. (Λ *x*. *f*·(*Discr* (*Inl* (*undiscr* *x*))),
 Λ *x*. *f*·(*Discr* (*Inr* (*undiscr* *x*))))

As so often when working with *HOLCF*, some continuity lemmas are required.

```

lemma cont2cont-case-sum[simp, cont2cont]:
  assumes cont f and cont g
  shows cont (λx. case-sum (f x) (g x) s)
using assms
by (cases s, auto intro: cont2cont-fun)

lemma cont2cont-circ[simp, cont2cont]:
  cont (λf. f ∘ g)
apply (rule cont2cont-lambda)
apply (subst comp-def)
apply (rule cont2cont-fun[of λx. x, OF cont-id])
done

lemma cont2cont-split-pair[cont2cont, simp]:
  assumes f1: cont f
    and f2: ∧ x. cont (f x)
    and g1: cont g
    and g2: ∧ x. cont (g x)
  shows cont (λ(a, b). (f a b, g a b))
apply (intro cont2cont)
apply (rule cont-apply[OF cont-snd - cont-const])
apply (rule cont-apply[OF cont-snd f2])
apply (rule cont-apply[OF cont-fst cont2cont-fun[OF f1] cont-const])

apply (rule cont-apply[OF cont-snd - cont-const])
apply (rule cont-apply[OF cont-snd g2])
apply (rule cont-apply[OF cont-fst cont2cont-fun[OF g1] cont-const])
done

```

Using these continuity lemmas, we can show that our function are actually continuous and thus allow us to apply them to a value.

```

lemma sum-to-tup-app:
  sum-to-tup.f = (λ x. f.(Discr (Inl (undiscr x))), λ x. f.(Discr (Inr (undiscr x))))
unfolding sum-to-tup-def by simp

```

```

lemma tup-to-sum-app:
  tup-to-sum.p = (λ s. (λ(f,g).
    case undiscr s of Inl x ⇒ f.(Discr x)
    | Inr x ⇒ g.(Discr x)) p)
unfolding tup-to-sum-def by simp

```

Generally, lambda abstractions with discrete domain are continuous and can be resolved immediately.

```

lemma discr-app[simp]:
   $(\lambda s. f s).(Discr x) = f (Discr x)$ 
by simp

```

Our transformation functions are inverse to each other, so we can use them to transform a fixed point.

lemma *tup-to-sum-to-tup[simp]*:
shows $sum\text{-to-tup} \cdot (tup\text{-to-sum} \cdot F) = F$
unfolding *sum-to-tup-app* **and** *tup-to-sum-app*
by (*cases F, auto intro:cfun-eqI*)

lemma *fix-transform-pair-sum*:
shows $fix \cdot F = sum\text{-to-tup} \cdot (fix \cdot (tup\text{-to-sum} \circ F \circ sum\text{-to-tup}))$
by (*rule fix-transform[OF tup-to-sum-to-tup]*)

After such a transformation, we want to get rid of these helper functions again. This is done by the next two simplification lemmas.

lemma *tup-sum-oo[simp]*:
assumes *f1: cont F*
and *f2: $\bigwedge x. cont (F x)$*
and *g1: cont G*
and *g2: $\bigwedge x. cont (G x)$*
shows $tup\text{-to-sum} \circ (\Lambda p. (\lambda(a, b). (F a b, G a b)) p) \circ sum\text{-to-tup}$
 $= (\Lambda f s. (case\ undiscr\ s\ of$
 $\quad Inl\ x \Rightarrow$
 $\quad\quad F\ (\Lambda s. f \cdot (Discr\ (Inl\ (undiscr\ s))))$
 $\quad\quad (\Lambda s. f \cdot (Discr\ (Inr\ (undiscr\ s)))) \cdot$
 $\quad\quad (Discr\ x)$
 $\quad | Inr\ x \Rightarrow$
 $\quad\quad G\ (\Lambda s. f \cdot (Discr\ (Inl\ (undiscr\ s))))$
 $\quad\quad (\Lambda s. f \cdot (Discr\ (Inr\ (undiscr\ s)))) \cdot$
 $\quad\quad (Discr\ x))$
by (*rule cfun-eqI, rule cfun-eqI,*
simp add: sum-to-tup-app tup-to-sum-app
cont2cont-split-pair[OF f1 f2 g1 g2]
cont2cont-lambda
cont-apply[OF - f2 cont2cont-fun[OF cont-compose[OF f1]]]
cont-apply[OF - g2 cont2cont-fun[OF cont-compose[OF g1]]])

lemma *fst-sum-to-tup[simp]*:
 $fst\ (sum\text{-to-tup} \cdot x) = (\Lambda xa. x \cdot (Discr\ (Inl\ (undiscr\ xa))))$
by (*simp add: sum-to-tup-app*)

end

References

- [1] J. Breitner. Control flow in functional languages. Student research project, Karlsruhe Institut für Technologie (KIT), November 2010.
- [2] J. Breitner. Implementation of Shivers' Control-Flow Analysis. <http://hackage.haskell.org/package/shivers-cfg-0.1>, November 2010.
- [3] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, 1991.