

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	12
II. The main results	16
5. The exact call cache is a map	16
5.1. Preparations	16
5.2. The proof	18
6. The abstract semantics is correct	19
6.1. Abstraction functions	19
6.2. Lemmas about abstraction functions	21
6.3. Approximation relation	21
6.4. Lemmas about the approximation relation	22

6.5. Lemma 7	23
6.6. Lemmas 8 and 9	23
7. Generic Computability	24
7.1. Non-branching case	24
7.2. Branching case	25
8. The abstract semantics is computable	26
8.1. Towards finiteness	28
8.2. A decomposition	30
8.3. The iterative equation	30
III. The auxiliary theories	30
9. Syntax tree helpers	30
10. General utility lemmas	35
11. Set-valued maps	36
12. Sets of maps	38
13. HOLCF Utility lemmas	40
14. Fixed point transformations	43

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

```

```

  <proof>

```

```

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 ( $\lambda x. case-d (f1 x) (f2 x) (f3 x) (f4 x) d$ )

```

```

  <proof>

```

```

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 ( $\lambda x. case-call (f1 x) (f2 x) c$ )

```

```

  <proof>

```

```

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 ( $\lambda x. case-prim (f1 x) (f2 x) p$ )

```

```

  <proof>

```

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],-,)  $\Rightarrow$  Def i
  | -  $\Rightarrow \perp$ 
    )
  |  $\mathcal{C} \cdot$  cstate = (case undiscr cstate of
    (App lab f vs, $\beta$ ,ve,b)  $\Rightarrow$ 

```

```

    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
 ⟨proof⟩

lemma *correct-ex2*: PR ex2 = Def 2
 ⟨proof⟩

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 $contour = contour$
 $\langle proof \rangle$

definition $initial-contour (b_0)$
where $b_0 = Abs-contour []$

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

instantiation $contour :: preorder$

begin

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

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

instance $\langle proof \rangle$

end

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

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

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

declare $less-imp-le[\mathbf{where}\ 'a = contour, intro]$

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  $\langle$ proof $\rangle$ 
end
instantiation d :: discrete-cpo begin
definition [simp]: (x::d)  $\sqsubseteq$  y  $\longleftrightarrow$  x = y
instance  $\langle$ proof $\rangle$ 
end

instantiation call :: discrete-cpo begin
definition [simp]: (x::call)  $\sqsubseteq$  y  $\longleftrightarrow$  x = y
instance  $\langle$ proof $\rangle$ 
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)  $\beta$  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 | None  $\Rightarrow$  DI 0)
            | None  $\Rightarrow$  DI 0)
      | A (L lam)  $\beta$  ve = DC (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  $\text{cont } (\lambda x. \text{case-lambda } (f x) l)$ 
 $\langle$ proof $\rangle$ 

```

```

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 $\bigwedge y. \text{cont} (\lambda x. f_3 x y)$
and $\text{cont} (\lambda x. f_4 x)$
shows $\text{cont} (\lambda x. \text{case-d} (f_1 x) (f_2 x) (f_3 x) (f_4 x) d)$
 $\langle \text{proof} \rangle$

lemma *cont2cont-case-call* [*simp*, *cont2cont*]:
assumes $\bigwedge a b c. \text{cont} (\lambda x. f_1 x a b c)$
and $\bigwedge a b c. \text{cont} (\lambda x. f_2 x a b c)$
shows $\text{cont} (\lambda x. \text{case-call} (f_1 x) (f_2 x) c)$
 $\langle \text{proof} \rangle$

lemma *cont2cont-case-prim* [*simp*, *cont2cont*]:
assumes $\bigwedge y. \text{cont} (\lambda x. f_1 x y)$
and $\bigwedge y z. \text{cont} (\lambda x. f_2 x y z)$
shows $\text{cont} (\lambda x. \text{case-prim} (f_1 x) (f_2 x) p)$
 $\langle \text{proof} \rangle$

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

```

    (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 *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 - - λa - . 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 - - λa - . 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 - - λa - . a,*

OF - list.exhaust, of - - λ- x - . x]

```

lemmas fstate-case = prod-cases4 [OF d.exhaust, of - λx - - - . 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 x Plus x 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 ⇒ ans (PR)
  where PR l = (let ve = Map.empty;
                β = Map.empty;
                f = A (L l) β ve
                in F.(Discr (f,[Stop],ve,b0)))

```

```

lemma correct-ex1: PR ex1 = {((2,[1 ↦ b0]), Stop)}
<proof>

```

```

lemma correct-ex2: PR ex2 = {((2, [1 ↦ b0]), DP (Plus 3)),
  ((3, [3 ↦ nb b0 2]), Stop)}
<proof>

```

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 ( $\widehat{nb}$ )
  and a-initial-contour :: 'a ( $\widehat{b_0}$ )
```

```
instantiation unit :: contour
```

```
begin
```

```
definition  $\widehat{nb}$  - - = ()
```

```
definition  $\widehat{b_0}$  = ()
```

```
instance ⟨proof⟩
```

```
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 (-  $\widehat{benv}$  [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 ⇒ {})
```

| $\widehat{A} (L \text{ lam}) \beta \text{ ve} = \{PC (lam, \beta)\}$

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

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

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

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

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

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

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

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

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

<proof>

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

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

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

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

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

<proof>

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

<proof>

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

<proof>

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` :: $'c::\text{contour } \widehat{\text{fstate}} \text{ discr} \rightarrow 'c \widehat{\text{ans}} (\widehat{\mathcal{F}})$

and `a-evalC` :: $'c::\text{contour } \widehat{\text{cstate}} \text{ discr} \rightarrow 'c \widehat{\text{ans}} (\widehat{\mathcal{C}})$

where $\widehat{\mathcal{F}}.\text{fstate} = (\text{case undiscr fstate of } (PC (Lambda \text{ lab vs } c, \beta), \text{as}, \text{ve}, b) \Rightarrow$
 $(\text{if length vs} = \text{length as}$
 $\text{then let } \beta' = \beta (\text{lab} \mapsto b);$

$$\begin{aligned}
& ve' = ve \cup. (\bigcup. (\text{map } (\lambda(v,a). \{(v,b) := a\}.) (\text{zip } vs \ as))) \\
& \text{in } \widehat{\mathcal{C}} \cdot (\text{Discr } (c, \beta', ve', b)) \\
& \text{else } \perp \\
| (PP \ (\text{Plus } c), [-, -, cnts], ve, b) \Rightarrow \\
& \quad \text{let } b' = \widehat{nb} \ b \ c; \\
& \quad \quad \beta = [c \mapsto b] \\
& \quad \text{in } (\bigcup_{cnt \in cnts}. \widehat{\mathcal{F}} \cdot (\text{Discr } (cnt, [\{\}], ve, b'))) \\
& \quad \cup \\
& \quad \{(c, \beta), cont \mid cont \cdot cont \in cnts\} \\
| (PP \ (\text{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}. \widehat{\mathcal{F}} \cdot (\text{Discr } (cnt, [], ve, b'))) \\
& \quad \quad \cup \{(ct, \beta), cnt \mid cnt \cdot cnt \in cntts\} \\
& \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}. \widehat{\mathcal{F}} \cdot (\text{Discr } (cnt, [], ve, b'))) \\
& \quad \quad \quad \cup \{(cf, \beta), cnt \mid cnt \cdot cnt \in cntfs\} \\
& \quad) \\
| (AStop, [-, -, -] \Rightarrow \{\}) \\
| - \Rightarrow \perp \\
) \\
| \widehat{\mathcal{C}} \cdot \text{cstate} = (\text{case undiscr cstate of} \\
& \quad (\text{App } lab \ f \ vs, \beta, ve, b) \Rightarrow \\
& \quad \quad \text{let } fs = \widehat{\mathcal{A}} \ f \ \beta \ ve; \\
& \quad \quad \quad as = \text{map } (\lambda v. \widehat{\mathcal{A}} \ v \ \beta \ ve) \ vs; \\
& \quad \quad \quad b' = \widehat{nb} \ b \ lab \\
& \quad \quad \quad \text{in } (\bigcup_{f' \in fs}. \widehat{\mathcal{F}} \cdot (\text{Discr } (f', as, ve, b'))) \\
& \quad \quad \quad \cup \{(lab, \beta), f' \mid f' \cdot f' \in fs\} \\
& \quad (\text{Let } lab \ ls \ c', \beta, ve, b) \Rightarrow \\
& \quad \quad \text{let } b' = \widehat{nb} \ b \ lab; \\
& \quad \quad \quad \beta' = \beta \ (lab \mapsto b'); \\
& \quad \quad \quad ve' = ve \cup. (\bigcup. (\text{map } (\lambda(v,l). \{(v,b') := (\widehat{\mathcal{A}} \ (L \ l) \ \beta' \ ve)\}.) \ ls)) \\
& \quad \quad \text{in } \widehat{\mathcal{C}} \cdot (\text{Discr } (c', \beta', ve', b')) \\
& \quad) \\
)
\end{aligned}$$

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

lemmas $a\text{-evalF}\text{-evalC}\text{-induct} = a\text{-evalF}\text{-a}\text{-evalC}\text{-induct}[\text{case-names Admissibility Bottom Next}]$

fun $a\text{-evalF}\text{-cases}$

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

```

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 - λx - - - . 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 $\widehat{\mathcal{F}}$ 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)  $\widehat{ans}$  ( $\widehat{\mathcal{P}\mathcal{R}}$ )
  where  $\widehat{\mathcal{P}\mathcal{R}}$  l = (let ve = {};
    β = Map.empty;
    f =  $\widehat{A}$  (L l) β ve
  in  $\widehat{\mathcal{F}}$ .(Discr (the-elm f, [{AStop}], ve,  $\widehat{b}_0$ )))

```

end

Part II.

The main results

5. The exact call cache is a map

```

theory ExCFVS
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* :: $d \Rightarrow \text{benv set}$
where *benv-in-d* ($DC\ (l, \beta)$) = $\{\beta\}$
| *benv-in-d* - = $\{\}$

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

fun *contours-in-d* :: $d \Rightarrow \text{contour set}$
where *contours-in-d* ($DC\ (l, \beta)$) = $\text{ran } \beta$
| *contours-in-d* - = $\{\}$

definition *contours-in-ve* :: $\text{venv} \Rightarrow \text{contour set}$
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 $\forall \beta \in \text{benv-in-ve } (ve(\text{map } (\lambda v. (v, b''))\ vs\ [\mapsto]\ ds)). Q\ \beta$
 $\langle \text{proof} \rangle$

lemma *benv-in-eval*:
assumes $\forall \beta' \in \text{benv-in-ve } ve. Q\ \beta'$
and $Q\ \beta$
shows $\forall \beta \in \text{benv-in-d } (\mathcal{A}\ v\ \beta\ ve). Q\ \beta$
 $\langle \text{proof} \rangle$

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

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 $\forall b' \in \text{contours-in-ve } (ve(\text{map } (\lambda v. (v, b''))\ vs\ [\mapsto]\ ds)). Q\ b'$
 $\langle \text{proof} \rangle$

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>

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 $\forall b' \in \text{contours-in-d } (\mathcal{A} f \beta ve). Q b'$
<proof>

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'*:

$\llbracket \forall b' \in \text{contours-in-ve } ve. b' < b$

$; \forall b' \in \text{contours-in-d } d. b' < b$

$; \forall d' \in \text{set } ds. \forall b' \in \text{contours-in-d } d'. b' < b$

\rrbracket

\implies

$(\text{single-valued } (\mathcal{F} \cdot (\text{Discr } (d, ds, ve, b)))$

$\wedge (\forall ((lab, \beta), t) \in \mathcal{F} \cdot (\text{Discr } (d, ds, ve, b)). \exists b'. b' \in \text{ran } \beta \wedge b \leq b')$

$)$

and $\llbracket b \in \text{ran } \beta'$

```

;  $\forall b' \in \text{ran } \beta'. b' \leq b$ 
;  $\forall b' \in \text{contours-in-ve } ve. b' \leq b$ 
 $\parallel$ 
 $\implies$ 
( single-valued ( $\mathcal{C}(\text{Discr } (c, \beta', ve, b))$ )
 $\wedge (\forall ((lab, \beta), t) \in \mathcal{C}(\text{Discr } (c, \beta', ve, b)). \exists b'. b' \in \text{ran } \beta \wedge b \leq b')$ 
)
<proof>

lemma single-valued ( $\mathcal{PR}$  prog)
<proof>
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 $abs\text{-}\langle type \rangle$, 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 *b₀* symbols.

```

class contour-a = contour +
fixes abs-cnt :: contour  $\Rightarrow$  'a
assumes abs-cnt-nb[simp]:  $abs\text{-}cnt \ (nb \ b \ lab) = \widehat{nb} \ (abs\text{-}cnt \ b) \ lab$ 
and abs-cnt-initial[simp]:  $abs\text{-}cnt(b_0) = \widehat{b_0}$ 

```

```

instantiation unit :: contour-a

```

```

begin

```

```

definition abs-cnt - = ()

```

```

instance <proof>

```

end

It would be unwieldy to always write out $abs\langle type \rangle x$. 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 $abs\langle type \rangle$ 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 $| \cdot |$ for the family of abstraction functions.

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

adhoc-overloading

abs abs-cnt

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

adhoc-overloading

abs abs-benv

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

adhoc-overloading

abs abs-closure

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

adhoc-overloading

abs abs-d

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

adhoc-overloading

abs abs-venv

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

adhoc-overloading

$abs\ abs\text{-}ccache$

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

adhoc-overloading

$abs\ abs\text{-}fstate$

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

adhoc-overloading

$abs\ abs\text{-}cstate$

6.2. Lemmas about abstraction functions

Some results about the abstractions functions.

lemma $abs\text{-}benv\text{-}empty[simp]: |Map.empty| = Map.empty$
 $\langle proof \rangle$

lemma $abs\text{-}benv\text{-}upd[simp]: |\beta(c \mapsto b)| = |\beta|\ (c \mapsto |b|)$
 $\langle proof \rangle$

lemma $the\text{-}elem\text{-}is\text{-}Proc$:
assumes $isProc\ cnt$
shows $the\text{-}elem\ |cnt| \in |cnt|$
 $\langle proof \rangle$

lemma $[simp]: |\{\}| = \{\}$ $\langle proof \rangle$

lemma $abs\text{-}cache\text{-}singleton\ [simp]: |\{(c,\beta),d\}| = \{(c, |\beta|), p) \mid p . p \in |d|\}$
 $\langle proof \rangle$

lemma $abs\text{-}venv\text{-}empty[simp]: |Map.empty| = \{\}$.
 $\langle proof \rangle$

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

ad hoc overloading
 $approx\ venv\text{-}approx$

definition $ccache\text{-}approx :: 'c \widehat{ccache} \Rightarrow 'c \widehat{ccache} \Rightarrow bool$
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 $\llbracket ve1 \lesssim ve2 ; ve2 \lesssim ve3 \rrbracket \implies (ve1 \lesssim ve3)$
 $\langle proof \rangle$

lemma *abs-venv-union*: $|ve1 ++ ve2| \lesssim |ve1| \cup |ve2|$
 ⟨proof⟩

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

lemma *abs-venv-map-of*: $|map-of l| \lesssim \bigcup. (map (\lambda(v,k). |[v \mapsto k]|) l)$
 ⟨proof⟩

lemma *abs-venv-singleton*: $|[(v,b) \mapsto d]| = \{(v,|b|) := |d|\}$.
 ⟨proof⟩

lemma *ccache-approx-empty[simp]*:
 fixes $x :: 'c \widehat{ccache}$
 shows $\{\} \lesssim x$
 ⟨proof⟩

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*: $|c1 \cup c2| \lesssim |c1| \cup |c2|$
 ⟨proof⟩

lemma *d-approx-empty[simp]*: $\{\} \lesssim (d::'c \widehat{d})$
 ⟨proof⟩

lemma *ds-approx-empty[simp]*: $\square \lesssim \square$
 ⟨proof⟩

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⟩

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 (λx. abs-ccache(f x))
  ⟨proof⟩

```

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 fstate and cstate-a :: 'c::contour-a cstate
  shows  $|fstate| \lesssim fstate-a \implies |\mathcal{F} \cdot (Discr\ fstate)| \lesssim \widehat{\mathcal{F}} \cdot (Discr\ fstate-a)$ 
  and  $|cstate| \lesssim cstate-a \implies |\mathcal{C} \cdot (Discr\ cstate)| \lesssim \widehat{\mathcal{C}} \cdot (Discr\ cstate-a)$ 
  ⟨proof⟩

```

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

```

lemma lemma6:  $|\mathcal{PR}\ l| \lesssim \widehat{\mathcal{PR}}\ l$ 
  ⟨proof⟩
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*:
 $insert\ (n::nat)\ \{n<..\} = \{n..\}$
 $\langle proof \rangle$

lemma *theorem10*:
fixes $g :: 'a::cpo \rightarrow 'b::type\ set$ **and** $r :: 'a \rightarrow 'a$
shows $fix.\ (\Lambda\ f\ x.\ g.x \cup f.(r.x)) = (\Lambda\ x.\ (\bigcup i.\ g.(r^i.x)))$
 $\langle proof \rangle$

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* :: $('a::cpo \rightarrow 'b::type\ set) \Rightarrow 'a\ set \rightarrow 'b\ set\ ()$
where $\underline{f} = (\Lambda\ S.\ (\bigcup y \in S.\ f.y))$

lemma *powerset-lift-singleton[simp]*:
 $\underline{f}\{x\} = f.x$
 $\langle proof \rangle$

lemma *powerset-lift-union[simp]*:
 $\underline{f}(A \cup B) = \underline{f}A \cup \underline{f}B$
 $\langle proof \rangle$

lemma *UNION-commute*: $(\bigcup x \in A.\ \bigcup y \in B.\ P\ x\ y) = (\bigcup y \in B.\ \bigcup x \in A.\ P\ x\ y)$
 $\langle proof \rangle$

lemma *powerset-lift-UNION*:
 $(\bigcup x \in S.\ \underline{g}(A\ x)) = \underline{g}(\bigcup x \in S.\ A\ x)$

<proof>

lemma *powerset-lift-iterate-UNION*:

$$(\bigcup_{x \in S}. (\underline{g})^i.(A\ x)) = (\underline{g})^i.(\bigcup_{x \in S}. A\ x)$$

<proof>

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 $f :: 'a \rightarrow 'b$ set **and** $g :: 'a \rightarrow 'b$ set **and** $R :: 'a \rightarrow 'a$ set

assumes $\bigwedge x. f \cdot x = g \cdot x \cup (\bigcup_{y \in R \cdot x}. f \cdot y)$

shows $\underline{f} \cdot S = \underline{g} \cdot S \cup \underline{f} \cdot (R \cdot S)$

<proof>

Theorem 10 as it will be used in Theorem 12.

lemmas *theorem10ps = theorem10[of \underline{g} \underline{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** $R :: 'a \rightarrow 'a$ set

assumes $F \cdot \text{fix}. F = \text{fix}. (\bigwedge F\ x. g \cdot x \cup F \cdot (R \cdot x))$

shows $\text{fix}. (\bigwedge f\ x. g \cdot x \cup (\bigcup_{y \in R \cdot x}. f \cdot y)) = (\bigwedge x. F \cdot \{x\})$

<proof>

lemma *theorem12*:

fixes $g :: 'a :: \text{discrete-cpo} \rightarrow 'b :: \text{type set}$ **and** $R :: 'a \rightarrow 'a$ set

shows $\text{fix}. (\bigwedge f\ x. g \cdot x \cup (\bigcup_{y \in R \cdot x}. f \cdot y)) \cdot x = \underline{g} \cdot (\bigcup i. ((R)^i \cdot \{x\}))$

<proof>

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  $abs-g :: ('c::contour \widehat{fstate} + 'c \widehat{cstate}) \text{discr} \rightarrow 'c \widehat{ans}$ 
  where  $abs-g.x = (\text{case undiscr } x \text{ of}$ 
     $(\text{Inl } (PC \ (Lambda \ lab \ vs \ c, \ \beta), \ as, \ ve, \ b)) \Rightarrow \{\}$ 
     $| (\text{Inl } (PP \ (Plus \ c), [-, -, cnts], ve, b)) \Rightarrow$ 
       $\text{let } b' = \widehat{nb} \ b \ c;$ 
       $\beta = [c \mapsto b]$ 
       $\text{in } \{((c, \beta), cont) \mid cont . cont \in cnts\}$ 
     $| (\text{Inl } (PP \ (prim.If \ ct \ cf), [-, cntts, cntfs], ve, b)) \Rightarrow$ 
       $(( \text{let } b' = \widehat{nb} \ b \ ct;$ 
       $\beta = [ct \mapsto b]$ 
       $\text{in } \{((ct, \beta), cnt) \mid cnt . cnt \in cntts\}$ 
       $) \cup$ 
       $( \text{let } b' = \widehat{nb} \ b \ cf;$ 
       $\beta = [cf \mapsto b]$ 
       $\text{in } \{((cf, \beta), cnt) \mid cnt . cnt \in cntfs\}$ 
       $))$ 
     $| (\text{Inl } (AStop, [-, -, -])) \Rightarrow \{\}$ 
     $| (\text{Inl } -) \Rightarrow \perp$ 
     $| (\text{Inr } (App \ lab \ f \ vs, \beta, ve, b)) \Rightarrow$ 
       $\text{let } fs = \widehat{A} \ f \ \beta \ ve;$ 
       $as = \text{map } (\lambda v. \widehat{A} \ v \ \beta \ ve) \ vs;$ 
       $b' = \widehat{nb} \ b \ lab$ 
       $\text{in } \{((lab, \beta), f') \mid f' . f' \in fs\}$ 
     $| (\text{Inr } (Let \ lab \ ls \ c', \beta, ve, b)) \Rightarrow \{\}$ 
   $)$ 

```

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

```

fixrec  $abs-R :: ('c::contour \widehat{fstate} + 'c \widehat{cstate}) \text{discr} \rightarrow ('c::contour \widehat{fstate} + 'c \widehat{cstate}) \text{discr}$ 
  set
  where  $abs-R.x = (\text{case undiscr } x \text{ of}$ 
     $(\text{Inl } (PC \ (Lambda \ lab \ vs \ c, \ \beta), \ as, \ ve, \ b)) \Rightarrow$ 
       $(\text{if } \text{length } vs = \text{length } as$ 
       $\text{then let } \beta' = \beta \ (lab \mapsto b);$ 
       $ve' = ve \cup. (\bigcup. (\text{map } (\lambda(v,a). \{(v,b) := a\}.)) \ (\text{zip } vs \ as)))$ 
       $\text{in } \{Discr \ (\text{Inr } (c, \beta', ve', b))\}$ 
       $\text{else } \perp)$ 
     $| (\text{Inl } (PP \ (Plus \ c), [-, -, cnts], ve, b)) \Rightarrow$ 
       $\text{let } b' = \widehat{nb} \ b \ c;$ 
       $\beta = [c \mapsto b]$ 
       $\text{in } (\bigcup_{cnt \in cnts}. \{Discr \ (\text{Inl } (cnt, [\{\}], ve, b'))\})$ 

```

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

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

definition $initial-r :: prog \Rightarrow 'c::contour \widehat{fstate} + 'c \widehat{cstate} \text{ discr}$
where $initial-r \text{ prog} = Discr (Inl$
 $(\text{the-elem } (\widehat{A} \ (L \ prog) \ \text{Map.empty } \{\}.), [\{AStop\}], \{\}., \widehat{b}_0))$

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

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

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

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

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

lemma $finite\text{-}arg\text{-}space: finite\ (arg\text{-}poss\ p)$
 $\langle proof \rangle$

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

lemma $evalV\text{-}possible:$
assumes $f: f \in \hat{A}\ d\ \beta\ ve$
and $d: d \in vals\ p$
and $ve: ve \in smaps\text{-}over\ (vars\ p \times UNIV)\ (proc\text{-}poss\ p)$
and $\beta: \beta \in maps\text{-}over\ (labels\ p)\ UNIV$
shows $f \in proc\text{-}poss\ p$
 $\langle proof \rangle$

lemma $adm\text{-}subset: cont\ (\lambda x. f\ x) \implies adm\ (\lambda x. f\ x \subseteq S)$
 $\langle proof \rangle$

lemma $arg\text{-}space\text{-}complete:$
 $state \in arg\text{-}poss\ p \implies abs\text{-}R.state \subseteq arg\text{-}poss\ p$
 $\langle proof \rangle$

This result is now lifted to the powerset of $abs\text{-}R$.

lemma $arg\text{-}space\text{-}complete\text{-}ps: states \subseteq arg\text{-}poss\ p \implies (\underline{abs}\text{-}R).states \subseteq arg\text{-}poss\ p$
 $\langle proof \rangle$

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 $UN\text{-}iterate\text{-}less:$
assumes $start: x \in S$
and $step: \bigwedge y. y \subseteq S \implies (f.y) \subseteq S$
shows $(\bigcup i. iterate\ i.f.\{x\}) \subseteq S$
 $\langle proof \rangle$

lemma $args\text{-}finite: finite\ (\bigcup i. iterate\ i.(\underline{abs}\text{-}R).\{initial\text{-}r\ p\})\ (\text{is}\ finite\ ?S)$
 $\langle proof \rangle$

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*: $(a \cup b) \cup (c \cup d) = (a \cup c) \cup (b \cup d)$
<proof>

lemma *a-evalF-decomp*:

$\widehat{\mathcal{F}} = fst (sum-to-tup \cdot (fix \cdot (\lambda f x. (\bigcup_{y \in abs-R \cdot x} f \cdot y) \cup abs-g \cdot x)))$
<proof>

8.3. The iterative equation

Because of the special form of $\widehat{\mathcal{F}}$ (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*:

$\widehat{\mathcal{F}} \cdot (Discr\ x) = \underline{abs-g} \cdot (\bigcup i. iterate\ i \cdot (\underline{abs-R}) \cdot \{Discr\ (Inl\ x)\})$
<proof>

lemma *a-evalCPS-iterative*:

$\widehat{\mathcal{PR}}\ prog = \underline{abs-g} \cdot (\bigcup i. iterate\ i \cdot (\underline{abs-R}) \cdot \{initial-r\ prog\})$
<proof>

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') = ( $\bigcup_{(-, y) \in \text{set binds.}} \textit{lambdas y}$ )  $\cup$  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'}  $\cup$  ( $\bigcup_{(-, y) \in \text{set binds.}} \textit{calls y}$ )  $\cup$  callsC c'
      | callsV (L l) = calls l
      | callsV - = {}

```

lemma *finite-lambdas[simp]*: *finite (lambdas l)* **and** *finite (lambdasC c)* *finite (lambdasV v)*
<proof>

lemma *finite-calls[simp]*: *finite (calls l)* **and** *finite (callsC c)* *finite (callsV v)*
<proof>

```

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') = ( $\bigcup_{(v, l) \in \text{set binds.}} \{v\} \cup \textit{vars l}$ )  $\cup$  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)*
<proof>

```

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} ∪ labelsV a ∪ ⋃(labelsV ‘ (set as))
| labelsC (Let l binds c') = {l} ∪ (⋃(v, y)∈set binds. labels y) ∪ 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)*
<proof>

```

fun prims :: lambda ⇒ prim set
and primsC :: call ⇒ prim set
and primsV :: val ⇒ prim set
where prims (Lambda - vs c) = primsC c
  | primsC (App - a as) = primsV a ∪ ⋃(primsV ‘ (set as))
  | primsC (Let - binds c') = (⋃(-, y)∈set binds. prims y) ∪ 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)*
<proof>

```

fun vals :: lambda ⇒ val set
and valsC :: call ⇒ val set
and valsV :: val ⇒ val set
where vals (Lambda - vs c) = valsC c
  | valsC (App - a as) = valsV a ∪ ⋃(valsV ‘ (set as))
  | valsC (Let - binds c') = (⋃(-, y)∈set binds. vals y) ∪ valsC c'
  | valsV (L l) = {L l} ∪ 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 × lambda) list and t :: var × lambda
shows lambdas1: Lambda l vs c ∈ lambdas x ⇒ c ∈ calls x
and Lambda l vs c ∈ lambdasC y ⇒ c ∈ callsC y
and Lambda l vs c ∈ lambdasV z ⇒ c ∈ callsV z
and ∀ z ∈ set list. Lambda l vs c ∈ lambdasV z → c ∈ callsV z
and ∀ x ∈ set list2. Lambda l vs c ∈ lambdas (snd x) → c ∈ calls (snd x)
and Lambda l vs c ∈ lambdas (snd t) ⇒ c ∈ calls (snd t)
<proof>

```

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 ∀ z ∈ set list. Lambda l vs c ∈ lambdasV z → l ∈ labelsV z

```


and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . \text{Lambda } l \text{ vs } c \in \text{lambdas } (\text{snd } x) \longrightarrow l \in \text{labels } (\text{snd } x)$
and $\text{Lambda } l \text{ vs } c \in \text{lambdas } (\text{snd } (t :: \text{var} \times \text{lambda})) \Longrightarrow l \in \text{labels } (\text{snd } t)$
<proof>

lemma

shows $\text{lambdas3}: \text{Lambda } l \text{ vs } c \in \text{lambdas } x \Longrightarrow \text{set } vs \subseteq \text{vars } x$
and $\text{Lambda } l \text{ vs } c \in \text{lambdasC } y \Longrightarrow \text{set } vs \subseteq \text{varsC } y$
and $\text{Lambda } l \text{ vs } c \in \text{lambdasV } z \Longrightarrow \text{set } vs \subseteq \text{varsV } z$
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 $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . \text{Lambda } l \text{ vs } c \in \text{lambdas } (\text{snd } x) \longrightarrow \text{set } vs \subseteq \text{vars } (\text{snd } x)$
and $\text{Lambda } l \text{ vs } c \in \text{lambdas } (\text{snd } (t :: \text{var} \times \text{lambda})) \Longrightarrow \text{set } vs \subseteq \text{vars } (\text{snd } t)$
<proof>

lemma

shows $\text{app1}: \text{App } l \text{ d } ds \in \text{calls } x \Longrightarrow d \in \text{vals } x$
and $\text{App } l \text{ d } ds \in \text{callsC } y \Longrightarrow d \in \text{valsC } y$
and $\text{App } l \text{ d } ds \in \text{callsV } z \Longrightarrow d \in \text{valsV } z$
and $\forall z \in \text{set list} . \text{App } l \text{ d } ds \in \text{callsV } z \longrightarrow d \in \text{valsV } z$
and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . \text{App } l \text{ d } ds \in \text{calls } (\text{snd } x) \longrightarrow d \in \text{vals } (\text{snd } x)$
and $\text{App } l \text{ d } ds \in \text{calls } (\text{snd } (t :: \text{var} \times \text{lambda})) \Longrightarrow d \in \text{vals } (\text{snd } t)$
<proof>

lemma

shows $\text{app2}: \text{App } l \text{ d } ds \in \text{calls } x \Longrightarrow \text{set } ds \subseteq \text{vals } x$
and $\text{App } l \text{ d } ds \in \text{callsC } y \Longrightarrow \text{set } ds \subseteq \text{valsC } y$
and $\text{App } l \text{ d } ds \in \text{callsV } z \Longrightarrow \text{set } ds \subseteq \text{valsV } z$
and $\forall z \in \text{set list} . \text{App } l \text{ d } ds \in \text{callsV } z \longrightarrow \text{set } ds \subseteq \text{valsV } z$
and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . \text{App } l \text{ d } ds \in \text{calls } (\text{snd } x) \longrightarrow \text{set } ds \subseteq \text{vals } (\text{snd } x)$
and $\text{App } l \text{ d } ds \in \text{calls } (\text{snd } (t :: \text{var} \times \text{lambda})) \Longrightarrow \text{set } ds \subseteq \text{vals } (\text{snd } t)$
<proof>

lemma

shows $\text{let1}: \text{Let } l \text{ binds } c' \in \text{calls } x \Longrightarrow l \in \text{labels } x$
and $\text{Let } l \text{ binds } c' \in \text{callsC } y \Longrightarrow l \in \text{labelsC } y$
and $\text{Let } l \text{ binds } c' \in \text{callsV } z \Longrightarrow l \in \text{labelsV } z$
and $\forall z \in \text{set list} . \text{Let } l \text{ binds } c' \in \text{callsV } z \longrightarrow l \in \text{labelsV } z$
and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . \text{Let } l \text{ binds } c' \in \text{calls } (\text{snd } x) \longrightarrow l \in \text{labels } (\text{snd } x)$
and $\text{Let } l \text{ binds } c' \in \text{calls } (\text{snd } (t :: \text{var} \times \text{lambda})) \Longrightarrow l \in \text{labels } (\text{snd } t)$
<proof>

lemma

shows $\text{let2}: \text{Let } l \text{ binds } c' \in \text{calls } x \Longrightarrow c' \in \text{calls } x$
and $\text{Let } l \text{ binds } c' \in \text{callsC } y \Longrightarrow c' \in \text{callsC } y$
and $\text{Let } l \text{ binds } c' \in \text{callsV } z \Longrightarrow c' \in \text{callsV } z$
and $\forall z \in \text{set list} . \text{Let } l \text{ binds } c' \in \text{callsV } z \longrightarrow c' \in \text{callsV } z$

and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list})$. Let l binds $c' \in \text{calls } (\text{snd } x) \longrightarrow c' \in \text{calls } (\text{snd } x)$

and Let l binds $c' \in \text{calls } (\text{snd } (t :: \text{var} \times \text{lambda})) \implies c' \in \text{calls } (\text{snd } t)$
 ⟨proof⟩

lemma

shows *let3*: Let l binds $c' \in \text{calls } x \implies \text{fst } ' \text{ set binds } \subseteq \text{vars } x$

and Let l binds $c' \in \text{callsC } y \implies \text{fst } ' \text{ set binds } \subseteq \text{varsC } y$

and Let l binds $c' \in \text{callsV } z \implies \text{fst } ' \text{ set binds } \subseteq \text{varsV } z$

and $\forall z \in \text{set list}$. Let l binds $c' \in \text{callsV } z \longrightarrow \text{fst } ' \text{ set binds } \subseteq \text{varsV } z$

and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list})$. Let l binds $c' \in \text{calls } (\text{snd } x) \longrightarrow \text{fst } ' \text{ set binds } \subseteq \text{vars } (\text{snd } x)$

and Let l binds $c' \in \text{calls } (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{fst } ' \text{ set binds } \subseteq \text{vars } (\text{snd } t)$

⟨proof⟩

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 $c' \in \text{callsC } y \implies \text{snd } ' \text{ set binds } \subseteq \text{lambdasC } y$

and Let l binds $c' \in \text{callsV } z \implies \text{snd } ' \text{ set binds } \subseteq \text{lambdasV } z$

and $\forall z \in \text{set list}$. Let l binds $c' \in \text{callsV } z \longrightarrow \text{snd } ' \text{ set binds } \subseteq \text{lambdasV } z$

and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list})$. Let l binds $c' \in \text{calls } (\text{snd } x) \longrightarrow \text{snd } ' \text{ set binds } \subseteq \text{lambdas } (\text{snd } x)$

and Let l binds $c' \in \text{calls } (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{snd } ' \text{ set binds } \subseteq \text{lambdas } (\text{snd } t)$

⟨proof⟩

lemma

shows *vals1*: $P \text{ prim} \in \text{vals } p \implies \text{prim} \in \text{prims } p$

and $P \text{ prim} \in \text{valsC } y \implies \text{prim} \in \text{primsC } y$

and $P \text{ prim} \in \text{valsV } z \implies \text{prim} \in \text{primsV } z$

and $\forall z \in \text{set list}$. $P \text{ prim} \in \text{valsV } z \longrightarrow \text{prim} \in \text{primsV } z$

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 $P \text{ prim} \in \text{vals } (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{prim} \in \text{prims } (\text{snd } t)$

⟨proof⟩

lemma

shows *vals2*: $R \text{ l var} \in \text{vals } p \implies \text{var} \in \text{vars } p$

and $R \text{ l var} \in \text{valsC } y \implies \text{var} \in \text{varsC } y$

and $R \text{ l var} \in \text{valsV } z \implies \text{var} \in \text{varsV } z$

and $\forall z \in \text{set list}$. $R \text{ l var} \in \text{valsV } z \longrightarrow \text{var} \in \text{varsV } z$

and $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list})$. $R \text{ l var} \in \text{vals } (\text{snd } x) \longrightarrow \text{var} \in \text{vars } (\text{snd } x)$

and $R \text{ l var} \in \text{vals } (\text{snd } (t :: \text{var} \times \text{lambda})) \implies \text{var} \in \text{vars } (\text{snd } t)$

⟨proof⟩

lemma

shows *vals3*: $L \text{ l} \in \text{vals } p \implies \text{l} \in \text{lambdas } p$

and $L \text{ l} \in \text{valsC } y \implies \text{l} \in \text{lambdasC } y$

and $L \text{ l} \in \text{valsV } z \implies \text{l} \in \text{lambdasV } z$

and $\forall z \in \text{set list}$. $L \text{ l} \in \text{valsV } z \longrightarrow \text{l} \in \text{lambdasV } z$

```

and  $\forall x \in \text{set } (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) . L l \in \text{vals } (\text{snd } x) \longrightarrow l \in \text{lambdas } (\text{snd } x)$ 
and  $L l \in \text{vals } (\text{snd } (t :: \text{var} \times \text{lambda})) \Longrightarrow l \in \text{lambdas } (\text{snd } t)$ 
<proof>

```

```

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 finA: finite A
  shows finite (nList A n)
<proof>

```

```

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

```

```

lemma finite-Nlist[intro]:
   $\llbracket \text{finite } A; \text{finite } N \rrbracket \Longrightarrow \text{finite } (\text{NList } A N)$ 
<proof>

```

```

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

```

```

lemma finite-call-list-lengths[simp]: finite (call-list-lengths p)
<proof>

```

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 {}
<proof>

```

```

lemma single-valued-insert:
  assumes single-valued rel
  and  $\bigwedge x y . \llbracket (x,y) \in \text{rel}; x=a \rrbracket \Longrightarrow y = b$ 
  shows single-valued (insert (a,b) rel)
<proof>

```

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

lemma *ran-upd*: $\text{ran } (m (k \mapsto v)) \subseteq \text{ran } m \cup \{v\}$
 ⟨proof⟩

lemma *ran-map-of*: $\text{ran } (\text{map-of } xs) \subseteq \text{snd } ` \text{set } xs$
 ⟨proof⟩

lemma *ran-concat*: $\text{ran } (m1 ++ m2) \subseteq \text{ran } m1 \cup \text{ran } m2$
 ⟨proof⟩

lemma *ran-upds*:
assumes *eq-length*: $\text{length } ks = \text{length } vs$
shows $\text{ran } (\text{map-upds } m \ ks \ vs) \subseteq \text{ran } m \cup \text{set } vs$
 ⟨proof⟩

lemma *ran-upd-mem[simp]*: $v \in \text{ran } (m (k \mapsto v))$
 ⟨proof⟩

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

lemma *map-fst-zip*: $\text{length } xs = \text{length } ys \implies \text{map } \text{fst } (\text{zip } xs \ ys) = xs$
 ⟨proof⟩

lemma *map-snd-zip*: $\text{length } xs = \text{length } ys \implies \text{map } \text{snd } (\text{zip } xs \ ys) = ys$
 ⟨proof⟩

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 (*{}*., *∪*.,...).

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*: $b \in m \ a \implies b \in \text{sran } m$

<proof>

lemma *sdom-not-mem[elim]*: $a \notin \text{sdom } m \implies m a = \{\}$
<proof>

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

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

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

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

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

lemma *sdom-empty[simp]*: $\text{sdom } \{\}. = \{\}$
<proof>

lemma *sdom-singleton[simp]*: $\text{sdom } \{k := vs\}. \subseteq \{k\}$
<proof>

lemma *sran-singleton[simp]*: $\text{sran } \{k := vs\}. = vs$
<proof>

lemma *sran-empty[simp]*: $\text{sran } \{\}. = \{\}$
<proof>

lemma *sdom-union[simp]*: $\text{sdom } (m \cup. n) = \text{sdom } m \cup \text{sdom } n$
<proof>

lemma *sran-union[simp]*: $\text{sran } (m \cup. n) = \text{sran } m \cup \text{sran } n$
<proof>

lemma *smap-empty[simp]*: $\{\}. \subseteq. \{\}$.
<proof>

lemma *smap-less-refl*: $m \subseteq. m$
<proof>

lemma *smap-less-trans[trans]*: $\llbracket m1 \subseteq. m2; m2 \subseteq. m3 \rrbracket \implies m1 \subseteq. m3$
<proof>

lemma *smap-union-mono*: $\llbracket ve1 \subseteq. ve1'; ve2 \subseteq. ve2' \rrbracket \implies ve1 \cup. ve2 \subseteq. ve1' \cup. ve2'$

<proof>

lemma *smap-Union-union*: $m1 \cup. \bigcup. ms = \bigcup. (m1 \# ms)$

<proof>

lemma *smap-Union-mono*:

assumes *list-all2 smap-less ms1 ms2*

shows $\bigcup. ms1 \subseteq. \bigcup. ms2$

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

lemma $\text{maps-over-empty}[\text{simp}]$:

$\text{Map.empty} \in \text{maps-over } A B$

$\langle \text{proof} \rangle$

lemma maps-over-upd :

assumes $m \in \text{maps-over } A B$

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

shows $m(v \mapsto k) \in \text{maps-over } A B$

$\langle \text{proof} \rangle$

lemma $\text{maps-over-finite}[\text{intro}]$:

assumes $\text{finite } A$ **and** $\text{finite } B$ **shows** $\text{finite } (\text{maps-over } A B)$

$\langle \text{proof} \rangle$

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

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

lemma $\text{smaps-over-empty}[\text{simp}]$:

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

$\langle \text{proof} \rangle$

lemma $\text{smaps-over-singleton}$:

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

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

$\langle \text{proof} \rangle$

lemma smaps-over-un :

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

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

$\langle \text{proof} \rangle$

lemma smaps-over-Union :

assumes $\text{set } ms \subseteq \text{smaps-over } A B$

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

$\langle \text{proof} \rangle$

lemma smaps-over-im :

$\llbracket f \in m \ a ; m \in \text{smaps-over } A B \rrbracket \Longrightarrow f \in B$

$\langle \text{proof} \rangle$

lemma $\text{smaps-over-finite}[\text{intro}]$:

assumes $\text{finite } A$ **and** $\text{finite } B$ **shows** $\text{finite } (\text{smaps-over } A B)$

$\langle \text{proof} \rangle$

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
   $x \sqsubseteq y \longleftrightarrow (x \longrightarrow y)$ 
instance <proof>
end
```

```
instance bool :: chfin
<proof>
```

```
instance bool :: pcpo
<proof>
```

```
lemma is-lub-bool:  $S \ll\mid (True \in S)$ 
<proof>
```

```
lemma lub-bool:  $\text{lub } S = (True \in S)$ 
<proof>
```

```
lemma bottom-eq-False[simp]:  $\perp = False$ 
<proof>
```

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
   $A \sqsubseteq B \longleftrightarrow A \subseteq B$ 
instance <proof>
end
```

```
lemma sqsubset-is-subset:  $A \sqsubseteq B \longleftrightarrow A \subseteq B$ 
<proof>
```

```
lemma is-lub-set:  $S \ll\mid \bigcup S$ 
<proof>
```


lemma *lub-is-union*: $\text{lub } S = \bigcup S$
⟨*proof*⟩

instance *set* :: (*type*) *cpo*
⟨*proof*⟩

lemma *emptyset-is-bot*[*simp*]: $\{\} \sqsubseteq S$
⟨*proof*⟩

instance *set* :: (*type*) *pcpo*
⟨*proof*⟩

lemma *bot-bool-is-emptyset*[*simp*]: $\perp = \{\}$
⟨*proof*⟩

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 $f: \text{cont } (\lambda x. f x)$ **and** $g: \text{cont } (\lambda x. g x)$
 shows $\text{cont } (\lambda x. f x \vee g x)$
⟨*proof*⟩

lemma *cont2cont-imp*[*simp*, *cont2cont*]:
 assumes $f: \text{cont } (\lambda x. \neg f x)$ **and** $g: \text{cont } (\lambda x. g x)$
 shows $\text{cont } (\lambda x. f x \longrightarrow g x)$
⟨*proof*⟩

lemma *cont2cont-Collect* [*simp*, *cont2cont*]:
 assumes $\bigwedge y. \text{cont } (\lambda x. f x y)$
 shows $\text{cont } (\lambda x. \{y. f x y\})$
⟨*proof*⟩

lemma *cont2cont-mem* [*simp*, *cont2cont*]:
 assumes $\text{cont } (\lambda x. f x)$
 shows $\text{cont } (\lambda x. y \in f x)$
⟨*proof*⟩

lemma *cont2cont-union* [*simp*, *cont2cont*]:
 $\text{cont } (\lambda x. f x) \Longrightarrow \text{cont } (\lambda x. g x)$
 $\Longrightarrow \text{cont } (\lambda x. f x \cup g x)$
⟨*proof*⟩

lemma *cont2cont-insert* [*simp*, *cont2cont*]:
 assumes $\text{cont } (\lambda x. f x)$
 shows $\text{cont } (\lambda x. \text{insert } y (f x))$
⟨*proof*⟩

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

lemma *cont2cont-UNION*[*cont2cont, simp*]:

assumes *cont f*
and $\bigwedge y. \text{cont } (\lambda x. g \ x \ y)$
shows *cont* $(\lambda x. \bigcup_{y \in f \ x. g \ x \ y}$
{*proof*}

lemma *cont2cont-Let-simple*[*simp, cont2cont*]:

assumes *cont* $(\lambda x. g \ x \ t)$
shows *cont* $(\lambda x. \text{let } y = t \ \text{in } g \ x \ y)$
{*proof*}

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* $(\lambda x. \text{case-list } (f1 \ x) \ (f2 \ x) \ l)$
{*proof*}

As with the continuity lemmas, we need admissibility lemmas.

lemma *adm-not-mem*:

assumes *cont* $(\lambda x. f \ x)$
shows *adm* $(\lambda x. y \notin f \ x)$
{*proof*}

lemma *adm-id*[*simp*]: *adm* $(\lambda x . x)$

{*proof*}

lemma *adm-Not*[*simp*]: *adm Not*

{*proof*}

lemma *adm-prod-split*:

assumes *adm* $(\lambda p. f \ (fst \ p) \ (snd \ p))$
shows *adm* $(\lambda(x,y). f \ x \ y)$
{*proof*}

lemma *adm-ball'*:

assumes $\bigwedge y. \text{adm } (\lambda x. y \in A \ x \longrightarrow P \ x \ y)$
shows *adm* $(\lambda x. \forall y \in A \ x . P \ x \ y)$
{*proof*}

lemma *adm-not-conj*:

$\llbracket \text{adm } (\lambda x. \neg P \ x); \text{adm } (\lambda x. \neg Q \ x) \rrbracket \Longrightarrow \text{adm } (\lambda x. \neg (P \ x \wedge Q \ x))$
{*proof*}

lemma *adm-single-valued*:

assumes *cont* $(\lambda x. f \ x)$

shows *adm* ($\lambda x. \text{single-valued } (f\ x)$)
 <proof>

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 $\text{fix} \cdot F = g \cdot (\text{fix} \cdot (f \text{ oo } F \text{ oo } g))$
 <proof>

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* = (Λp s. ($\lambda(f,g).$
 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* = ($\Lambda f. (\Lambda x. f \cdot (\text{Discr } (\text{Inl } (\text{undiscr } x))),$
 $\Lambda x. f \cdot (\text{Discr } (\text{Inr } (\text{undiscr } x))))$)

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)*
<proof>

lemma *cont2cont-circ*[*simp,cont2cont*]:
cont (λf. f ∘ g)
<proof>

lemma *cont2cont-split-pair*[*cont2cont,simp*]:
assumes *f1: cont f*
and *f2: λ x. cont (f x)*
and *g1: cont g*
and *g2: λ x. cont (g x)*
shows *cont (λ(a, b). (f a b, g a b))*
<proof>

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))))
<proof>

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)*
<proof>

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

lemma *discr-app*[*simp*]:
(λ s. f s).(Discr x) = f (Discr x)
<proof>

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-to-tup.(tup-to-sum.F) = F*
<proof>

lemma *fix-transform-pair-sum*:
shows *fix.F = sum-to-tup.(fix.(tup-to-sum oo F oo sum-to-tup))*
<proof>

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:  $\bigwedge x. \text{cont } (F x)$ 
    and g1: cont G
    and g2:  $\bigwedge x. \text{cont } (G x)$ 
shows tup-to-sum oo ( $\Lambda p. (\lambda(a, b). (F a b, G a b)) p$ ) oo sum-to-tup
  = ( $\Lambda f s. (\text{case undisc } s \text{ of}$ 
     $\text{Inl } x \Rightarrow$ 
       $F (\Lambda s. f \cdot (\text{Disc } (\text{Inl } (\text{undisc } s))))$ 
       $(\Lambda s. f \cdot (\text{Disc } (\text{Inr } (\text{undisc } s)))) \cdot$ 
       $(\text{Disc } x)$ 
    |  $\text{Inr } x \Rightarrow$ 
       $G (\Lambda s. f \cdot (\text{Disc } (\text{Inl } (\text{undisc } s))))$ 
       $(\Lambda s. f \cdot (\text{Disc } (\text{Inr } (\text{undisc } s)))) \cdot$ 
       $(\text{Disc } x))$ )
  <proof>

```

```

lemma fst-sum-to-tup[simp]:
   $\text{fst } (\text{sum-to-tup } x) = (\Lambda xa. x \cdot (\text{Disc } (\text{Inl } (\text{undisc } xa))))$ 
  <proof>

```

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.