Shivers’ Control Flow Analysis

Joachim Breitner

May 28, 2015

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 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
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 alphatised, i.e. that each variable name is only
bound once. This binding position is, as a convenience, considered part of the variable name.

\textbf{type-synonym} label = nat
\textbf{type-synonym} var = label × string

\textbf{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 \textit{false}. We simplified this to just the set of integers. The conditional \textit{If} considers zero as false and any other number as true.

Shivers also restricts the values in a call expression: No constant maybe 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.

\textbf{datatype} prim = Plus label | If label label
\textbf{datatype} lambda = Lambda label var list call
\hspace{1em} \textbf{and} call = App label val val list
\hspace{1em} | Let label (var × lambda) list call
\hspace{1em} \textbf{and} val = L lambda | R label var | C label int | P prim

\textbf{datatype-compat} lambda call val

\textbf{type-synonym} prog = lambda

\textbf{lemmas} mutual-lambda-call-var-inducts =
\hspace{1em} compat-lambda.induct
\hspace{1em} compat-call.induct
\hspace{1em} compat-val.induct
\hspace{1em} compat-val-list.induct
\hspace{1em} compat-nat-char-list-prod-lambda-prod-list.induct
\hspace{1em} 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].

\textbf{abbreviation} ex1 == (Lambda 1 [(1,"cont'")]) (App 2 (R 3 (1,"cont'"))) [(C 4 0)])
\textbf{abbreviation} ex2 == (Lambda 1 [(1,"cont'")]) (App 2 (P (Plus 3)) [(C 4 1), (C 5 1), (R 6 (1,"cont'"))])
\textbf{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-'")])))])]]))]

3
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 clarifi-
cations 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 ⇒ contour
type-synonym closure = lambda × 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

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.
The function $A$ 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 $\beta$, 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.

```latex
fun evalV :: val ⇒ benv ⇒ venv ⇒ 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 ⇒ (case ve (var,l) of Some d ⇒ d))
| A (L lam) β ve = DC (lam, β)
```

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 $\bot$.

```latex
type-synonym ans = int lift
```

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

```latex
lemma cont2cont-case-lambda [simp, cont2cont]:
  assumes ∀a b c. cont (λx. f x a b c)
  shows cont (λx. case-lambda (f x) l)
using assms
by (cases l) auto
```

```latex
lemma cont2cont-case-d [simp, cont2cont]:
  assumes ∀y. cont (λx. f1 x y)
  and ∀y. cont (λx. f2 x y)
  and ∀y. cont (λx. f3 x y)
  and cont (λx. f4 x)
  shows cont (λx. case-d (f1 x) (f2 x) (f3 x) (f4 x) d)
using assms
by (cases d) auto
```

```latex
lemma cont2cont-case-call [simp, cont2cont]:
  assumes ∀a b c. cont (λx. f1 x a b c)
  and ∀a b c. cont (λx. f2 x a b c)
  shows cont (λx. case-call (f1 x) (f2 x) c)
using assms
by (cases c) auto
```

```latex
lemma cont2cont-case-prim [simp, cont2cont]:
  assumes ∀y. cont (λx. f1 x y)
  and ∀y z. cont (λx. f2 x y z)
```
shows $\text{cont} \ (\lambda x. \ \text{case-prim} \ (f1 \ x) \ (f2 \ x) \ p)$
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: $F$ applies a procedure to a list of arguments. Here closures are unwrapped, the primitive operations are implemented and the terminal continuation $\text{Stop}$ is handled. $C$ evaluates a call expression, either by evaluating procedure and arguments and passing them to $F$, or by adding the bindings of a $\text{Let}$ expression to the environment.

Note how the contour counter is incremented before each call to $F$ or when a $\text{Let}$ 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 $F$ and $C$ 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 $\text{discr}$ and $\text{lift}$ to indicate which partial order is to be used.

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

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

\textbf{definition evalCPS :: prog ⇒ ans (PR)}
\begin{align*}
\text{where PR l} &= \text{(let ve = empty; } \\
\beta &= \text{empty; } \\
f &= A (L l) \beta ve \]  
\text{in } \mathcal{F} (\text{Discr } (f,[\text{Stop}],ve,0)))
\end{align*}

\textbf{lemma correct-ex1: PR ex1 = Def 0}
\text{unfolding evalCPS-def by simp}

\textbf{lemma correct-ex2: PR ex2 = Def 2}
\text{unfolding evalCPS-def by simp}

end

\section{3. Exact nonstandard semantics}

\textbf{theory ExCF}
\textbf{imports HOLCF HOLCFUtils CPSScheme Utils}
\begin{align*}
\text{begin} \hspace{1cm}
\end{align*}

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, re-
membering at each step which part of the program was just evaluated.

Note that for the exact semantics, this is 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 ≤ b' ←→ length (Rep-contour b) ≤ length (Rep-contour b')
**definition** less-contour-def: b < b' ←→ length (Rep-contour b) < length (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 ≤ b' ←→ 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 ←→ b' ≤ 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 → contour
type-synonym closure = lambda × benv

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

type-synonym venv = var × contour → 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) ⊑ y ←→ x = y
  instance by default simp
end

instantiation d :: discrete-cpo begin
  definition [simp]: (x::d) ⊑ y ←→ x = y
  instance by default simp
end

instantiation call :: discrete-cpo begin
  definition [simp]: (x::call) ⊑ y ←→ x = y
  instance by default 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 ⇒ benv ⇒ venv ⇒ 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 ⇒ (case ve (var,l) of Some d ⇒ d | None ⇒ DI 0)
      | None ⇒ DI 0)
  | A (L lam) β ve = DC (lam, β)
To be able to do case analysis on the custom datatypes \( \lambda \), \( d \), \( call \) and \( prim \) inside a function defined with \( \text{fixrec} \), we need continuity results for them. These are all of the same shape and proven by case analysis on the discriminator.

\[ \text{lemma cont2cont-case-lambda} \ [\text{simp, cont2cont}]: \]
\[ \quad \text{assumes } \lambda a b c. \cont (\lambda x. f x a b c) \]
\[ \quad \text{shows } \cont (\lambda x. \text{case-lambda} (f x) l) \]
\[ \quad \text{using } \text{assms} \]
\[ \quad \text{by } (\text{cases } l) \text{ auto} \]

\[ \text{lemma cont2cont-case-d} \ [\text{simp, cont2cont}]: \]
\[ \quad \text{assumes } \lambda y. \cont (\lambda x. f1 x y) \]
\[ \quad \text{and } \lambda y. \cont (\lambda x. f2 x y) \]
\[ \quad \text{and } \lambda y. \cont (\lambda x. f3 x y) \]
\[ \quad \text{and } \cont (\lambda x. f4 x) \]
\[ \quad \text{shows } \cont (\lambda x. \text{case-d} (f1 x) (f2 x) (f3 x) (f4 x) d) \]
\[ \quad \text{using } \text{assms} \]
\[ \quad \text{by } (\text{cases } d) \text{ auto} \]

\[ \text{lemma cont2cont-case-call} \ [\text{simp, cont2cont}]: \]
\[ \quad \text{assumes } \lambda a b c. \cont (\lambda x. f1 x a b c) \]
\[ \quad \text{and } \lambda a b c. \cont (\lambda x. f2 x a b c) \]
\[ \quad \text{shows } \cont (\lambda x. \text{case-call} (f1 x) (f2 x) c) \]
\[ \quad \text{using } \text{assms} \]
\[ \quad \text{by } (\text{cases } c) \text{ auto} \]

\[ \text{lemma cont2cont-case-prim} \ [\text{simp, cont2cont}]: \]
\[ \quad \text{assumes } \lambda y. \cont (\lambda x. f1 x y) \]
\[ \quad \text{and } \lambda y z. \cont (\lambda x. f2 x y z) \]
\[ \quad \text{shows } \cont (\lambda x. \text{case-prim} (f1 x) (f2 x) p) \]
\[ \quad \text{using } \text{assms} \]
\[ \quad \text{by } (\text{cases } p) \text{ auto} \]

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 \( F \) and \( C \), 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.

\[ \text{type-synonym ccache} = ((\text{label } \times \text{ benv}) \times d) \text{ set} \]
\[ \text{type-synonym ans} = \text{ccache} \]
\[ \text{type-synonym fstate} = (d \times d \text{ list } \times \text{ venv } \times \text{ contour}) \]
\[ \text{type-synonym cstate} = (\text{call } \times \text{ benv } \times \text{ venv } \times \text{ contour}) \]

\[ \text{fixrec \ \text{evalF} :: fstate discr } \rightarrow \text{ ans } (F) \]
and evalC :: cstate discr \to ans (C)

where \mathcal{F} \cdot 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 ⊥)
  | (DP (Plus c), [DI a1, DI a2, cnt], ve, b) \Rightarrow
    (if isProc cnt
      then let b' = nb b c;
      \beta = [c \mapsto b]
      in \mathcal{F} \cdot (Discr (cnt, [DI (a1 + a2)], ve, b'))
      \cup \{(c, \beta, cnt)\}
    else let b' = nb b cf;
     \beta = [cf \mapsto b]
      in \mathcal{F} \cdot (Discr (cnt, [ve, b']))
      \cup \{(cf, \beta, cnt)\}
    else ⊥)
  | (DP (prim.If ct cf), [DI v, contt, contf], ve, b) \Rightarrow
    (if isProc contt \land isProc contf
      then (if v \neq 0
        then let b' = nb b ct;
        \beta = [ct \mapsto b]
        in (\mathcal{F} \cdot (Discr (contt, [ve, b'])))
        \cup \{(ct, \beta, contt)\})
      else let b' = nb b cf;
     \beta = [cf \mapsto b]
      in (\mathcal{F} \cdot (Discr (contf, [ve, b'])))
      \cup \{(cf, \beta, contf)\})
    else ⊥)
  | (Stop, [DI i], -) \Rightarrow \{}
  | - \Rightarrow ⊥)

| \mathcal{C} \cdot cstate = (case undiscr cstate of
  (App lab f vs, \beta, ve, b) \Rightarrow
    let f' = A f \beta ve;
    as = map (\lambda v. A v \beta ve) vs;
    b' = nb b lab
    in if isProc f'
      then \mathcal{F} \cdot (Discr (f', as, ve, b')) \cup \{(\beta, f')\}
      else ⊥
  | (Let lab ls c', \beta, ve, b) \Rightarrow
    let b' = nb b lab;
    \beta' = \beta (lab \mapsto b');
    ve' = ve ++ map-of (map (\lambda (v, l). (\lambda (v, b') \cdot A (L l) \beta' ve)) ls)
    in \mathcal{C} \cdot (Discr (c', \beta', 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 - $\lambda$ a . a]

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

- $OF$ - d.exhaust, of - - $\lambda$ a . a,
- $OF$ - list.exhaust, of - - $\lambda$- x -. x,
- $OF$ - d.exhaust, of - - $\lambda$- - a -. a,
- $OF$ - list.exhaust, of - - $\lambda$- - x -. x,
- $OF$ - - list.exhaust, of - - $\lambda$- - - x -. x.

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

- $OF$ - list.exhaust[OF - list.exhaust, of - - $\lambda$- x . x], of - - $\lambda$- x . x], of - - $\lambda$- x -. x]

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

- $OF$ - list.exhaust, of - - $\lambda$- x -. x]

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

- $OF$ - cl-cases prim.exhaust, of - - $\lambda$- - - a . a $\lambda$ - - - - a . a,
- $OF$ - case-split ds-cases-plus ds-cases-if ds-cases-stop,

- of - - $\lambda$- as - - - - - vs - . length vs = length as $\lambda$ - ds - - - . ds $\lambda$ - ds - - - . ds $\lambda$ - ds - - . ds,

- case-names x Closure x x x x x x x x Plus x x x x x x x x x x If-True If-False x x x x x x x x x x Stop x x x x x 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 = empty;

  $\beta$ = empty;

  $f$ = $A$ ($L$ $l$) $\beta$ ve

  in $\mathcal{F}$:(Discr ($f$,Stop ve,b_0)))

**lemma** correct-ex1: $\mathcal{PR}$ ex1 = $\{(2,[1 \mapsto b_0]), \ Stop\}$

unfolding evalCPS-def

by simp

**lemma** correct-ex2: $\mathcal{PR}$ ex2 = $\{(2, [1 \mapsto b_0]), DP \ (Plus \ 3)),

((3, [3 \mapsto nb b_0 2]), \ Stop)\}$

unfolding evalCPS-def

by simp


4. Abstract nonstandard semantics

**theory** AbsCF
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 ( hat nb)
  and a-initial-contour :: 'a ( hat b_0)

instantiation unit :: contour
begin
  definition hat nb - = ()
  definition hat b_0 = ()
  instance by (default 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 SetMap contains functions and lemmas to work with such maps, symbolized by an appended dot (e.g. {}. ⊔).

```
type-synonym 'c a-benv = label ⇒ 'c (' - benv [1000])
type-synonym 'c a-closure = lambda × 'c benv (' - closure [1000])
```
datatype 'c proc (- proc [1000])
    = PC 'c closure
    | PP prim
    | AStop

type-synonym 'c a-d = 'c proc set (- d [1000])

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

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

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

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

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

type-synonym 'c a-ans = 'c ccache (- ans [1000])

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

type-synonym 'c a-cstate = (call × 'c benv × 'c venv × 'c) (- 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 (λx. f x a b c)
    shows cont (λx. case-lambda (f x) l)
    using assms
    by (cases l) auto

lemma cont2cont-case-proc [simp, cont2cont]:
    assumes \( \bigwedge y. \) cont (λx. f1 x y)
        and \( \bigwedge y. \) cont (λx. f2 x y)
        and cont (λx. f3 x)
    shows cont (λx. case-proc (f1 x) (f2 x) (f3 x) d)
    using assms
    by (cases d) auto

lemma cont2cont-case-call [simp, cont2cont]:
assumes \( \bigwedge a \ b \ c. \ cont (\lambda x. f1 \ x \ a \ b \ c) \)
and \( \bigwedge a \ b \ c. \ cont (\lambda x. f2 \ x \ a \ b \ c) \)
shows \( \cont (\lambda x. \text{case-call} \ (f1 \ x) \ (f2 \ x) \ c) \)
using \( \text{assms} \)
by \( \text{(cases c) auto} \)

lemma \( \text{cont2cont-case-prim} \ [\text{simp, cont2cont}] \):
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 \( \text{assms} \)
by \( \text{(cases p) auto} \)

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

\[
A\text{evalF} :: 'c::contour \ f\text{state} \ \text{discr} \ \rightarrow '\ 'c \ \text{ans} \ (\hat{\cal F})
\]
\[
A\text{evalC} :: 'c::contour \ c\text{state} \ \text{discr} \ \rightarrow '\ 'c \ \text{ans} \ (\hat{\cal C})
\]
where \( \hat{\cal F} \cdot \text{fstate} = (\text{case undiscr} \ f\text{state} \ of\n\ ) \ )
(P\ C \ (\text{Lambda} \ \text{lab} \ vs \ c, \ \beta, \ as, \ ve, \ b) \ \Rightarrow
\ )
\[
(\text{if} \ \text{length} \ vs = \ \text{length} \ as \ \text{then} \ \text{let} \ \beta' = \beta (\text{lab} \mapsto b); \ \text{ve}' = \text{ve} \cup \ (\bigcup \ (\map\ (\lambda (v,a). \ {(v,b) := a}). \ (\text{zip} \ vs \ as))) \ )\text{in} \ \hat{\cal C} \cdot \text{Discr} \ (c,\beta',\text{ve}',\text{b})\])
\]
else \( \bot \ )
\[
| (PP \ (\text{Plus} \ c),[-,-,\text{cnts}],\text{ve},\text{b}) \ \Rightarrow
\ )
\[
(\text{let} \ \beta' = \hat{\text{nb}} \ b \ c; \ \beta = [c \mapsto b]; \ \text{in} \ (\bigcup \ \text{cnt} \in \text{cnts} \ . \ \hat{\cal F} \cdot \text{Discr} \ ([\{}],\text{ve},\text{b}')\)) \bigcup\{(c, \beta, \ text{cont}) | \ text{cont} \in \text{cnts}\}
\]
| (PP \ (\text{prim.If} \ ct \ cf),[-,-,\text{cnts},\text{cntfs}],\text{ve},\text{b}) \ \Rightarrow
\ )
\[
(\text{let} \ \beta' = \hat{\text{nb}} \ b \ ct; \ \beta = [ct \mapsto b]; \ \text{in} \ (\bigcup \ \text{cnt} \in \text{cnts} \ . \ \hat{\cal F} \cdot \text{Discr} \ ([\{}],\text{ve},\text{b}')\)) \bigcup\{(ct, \beta, \ text{cnt}) | \ text{cnt} \in \text{cntfs}\}
\]
\)
| (A\text{Stop},[-,-,-] \ \Rightarrow \ \{\})
| (\text{-} \ \Rightarrow \ \bot)
\)
| \( \hat{\cal C} \cdot \text{cstate} = (\text{case undiscr} \ \text{cstate} \ of\n\ ) \ )
\[(\text{App lab } f \text{ vs} \beta, \text{ve}, b) \Rightarrow\]
\[
\text{let } fs = \hat{A} f \beta \text{ ve};
\text{as} = \text{map } (\lambda v. \hat{A} v \beta \text{ ve}) \text{ vs};
\text{b'} = \overline{nb} b \text{ lab}
\text{in } (\bigcup f' \in fs. \hat{F}.(\text{Discr } (f', \text{as}, \text{ve}, b')))
\cup\{( (\text{lab}, \beta), f') | f' \cdot f' \in fs\}
\]
\[
| (\text{Let lab } ls c', \beta, \text{ve}, b, b) \Rightarrow\]
\[
\text{let } b' = \overline{nb} b \text{ lab};
\beta' = \beta (\text{lab} \mapsto b');
\text{ve'} = \text{ve} \cup (\bigcup (\text{map } (\lambda (v, l). \{(v, b') := (\hat{A} (L l) \beta' \text{ ve})\}). \text{ls})
\text{in } \hat{C}.(\text{Discr } (c', \beta', \text{ve'}, b'))
\]

Again, we name the cases of the induction rule and build a nicer case analysis rule for arguments of type \(\hat{f} \text{state}\).

**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 - \lambda- vs - . length vs = length as,
| case-names Closure Closure-inv Plus If Stop

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

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

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

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

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

**definition** evalCPS-a :: prog \Rightarrow (\langle c::contour \rangle \overline{an}\ (\overline{\text{PR}}))

**where** \(\overline{\text{PR}} l = (\text{let ve} = \{\};

16
\[ \beta = \text{empty}; \\
f = \tilde{A}(L\,l)\,\beta\,ve \\
in \tilde{F}((\text{Discr (the-elem } f,\{\text{AStop}\},ve,\hat{b}_0))) \\
\]

**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 = \( \bigcup \{ \text{benv-in-d } d \mid d . d \in \text{ran } ve \} \)

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: length vs = length ds

and \( \forall \beta \in \text{benv-in-ve} \ \text{ve} \ \ Q \ \beta \)

and \( \forall d' \in \text{set} \ ds \ \forall \beta \in \text{benv-in-d} \ d' \ \text{ve} \ \ Q \ \beta \)

**Shows** \( \forall \beta \in \text{benv-in-ve} \ (\text{ve(map} \ (\lambda v. \ (v, \ b'')) \ vs \ \rightarrow \ ds)) \ \ Q \ \beta \)

**Proof**

fix \( \beta \)

assume \( \beta \in \text{benv-in-ve} \ (\text{ve(map} \ (\lambda v. \ (v, \ b'')) \ vs \ \rightarrow \ ds)) \)

then obtain \( d \) where \( \beta \in \text{benv-in-d} \ d \) and \( d \in \text{ran} \ (\text{ve(map} \ (\lambda v. \ (v, \ b'')) \ vs \ \rightarrow \ ds)) \)

unfolding benv-in-def by auto

moreover have \( \text{ran} \ (\text{ve(map} \ (\lambda v. \ (v, \ b'')) \ vs \ \rightarrow \ ds)) \subseteq \text{ran} \ \text{ve} \cup \text{set} \ ds \) using eq-length

ultimately have \( d \in \text{ran} \ \text{ve} \lor d \in \text{set} \ ds \) by auto

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

qed

**Lemma benv-in-eval:**

**Assumes** \( \forall \beta' \in \text{benv-in-ve} \ \text{ve} \ \ Q \ \beta' \)

and \( Q \ \beta \)

**Shows** \( \forall \beta \in \text{benv-in-d} \ (A \ v \ \beta \ \text{ve}) \ \ Q \ \beta \)

**Proof**

(cases \( v \))

**Case** \( R - \text{var} \)

thus \( \langle \text{thesis} \rangle \)

**Proof**

(cases \( \beta \) \( \text{fst} \ \text{var} \))

**Case** None with \( R \) show \( \langle \text{thesis} \rangle \) by simp next

**Case** (Some \( \text{cnt} \)) show \( \langle \text{thesis} \rangle \)

**Proof**

(cases \( \text{ve} \ (\text{var},\text{cnt}) \))

**Case** None with \( \text{Some} \ R \) show \( \langle \text{thesis} \rangle \) by simp next

**Case** (Some \( d \))

hence \( d \in \text{ran} \ \text{ve} \) unfolding ran-def by blast

thus \( \langle \text{thesis} \rangle \) using \( \text{Some} \ (\beta \ (\text{fst} \ \text{var}) = \text{Some} \ \text{cnt}) \) \( R \) assms (1)

unfolding benv-in-def by auto

qed

**Qed**

**Qed next**

**Case** \( L \ \text{l} \) thus \( \langle \text{thesis} \rangle \) using assms (2) by simp next

**Case** \( C \) thus \( \langle \text{thesis} \rangle \) by simp next

**Case** \( P \) thus \( \langle \text{thesis} \rangle \) by simp

**Qed**

**Lemma contours-in-ve-empty[simp]:** contours-in-ve empty = {}

unfolding contours-in-ve-def by auto

**Lemma contours-in-ve-upds:**

**Assumes** eq-length: length vs = length ds

and \( \forall b' \in \text{contours-in-ve} \ \text{ve} \ \ Q \ b' \)

and \( \forall d' \in \text{set} \ ds \ \forall b' \in \text{contours-in-d} \ d' \ \text{ve} \ \ Q \ b' \)

18
shows \( \forall b' \in \text{contours-in-ve} \ (ve(map \ (\lambda v. (v, b'')) vs \mapsto ds)). \ Q b' \)

proof=
  have ran (ve(map \ (\lambda v. (v, b'')) vs \mapsto ds)) \subseteq ran ve \cup set ds using eq-length by(auto intro!:ran-upsds)
  thus \( ?\text{thesis using assms(2,3)} \)

qed

lemma contours-in-upds-binds:
  assumes \( \forall b' \in \text{contours-in-ve} \ ve. \ Q b' \)
  and \( \forall b' \in \text{ran} \ b'. \ Q b' \)
  shows \( \forall b' \in \text{contours-in-ve} \ (ve++map-of \ (map \ (\lambda v.l). ((v, b''), A (L l) \beta' ve)) ls)). \ Q b' \)

proof
  fix \( b' \) assume \( b' \in \text{contours-in-ve} \ (ve++map-of \ (map \ (\lambda v.l). ((v, b''), A (L l) \beta' ve)) ls)) \)
  then obtain \( d \) where \( d:d \in \text{ran} \ (ve++map-of \ (map \ (\lambda v.l). ((v, b''), A (L l) \beta' ve)) ls)) \)
  and \( b:b' \in \text{contours-in-d} \ d \)
  unfolding contours-in-ve-def by auto

  have ran (ve++map-of \ (map \ (\lambda v.l). ((v, b''), A (L l) \beta' ve)) ls)) \subseteq ran ve \cup ran \ (map-of \ (map \ (\lambda v.l). ((v, b''), A (L l) \beta' ve)) ls))
  by(auto intro!:ran-concat)
  also
  have \( \ldots \subseteq \text{ran} \ ve \cup \text{snd} \ \text{set} \ (map \ (\lambda v.l). ((v, b''), A (L l) \beta' ve)) ls) \)
  by (rule Un-mono[of \text{ran} \ve \text{ran} \ve, OF \text{subset-refl} \text{ran-map-of}])
  also
  have \( \ldots \subseteq \text{ran} \ ve \cup \text{set} \ (map \ (\lambda v.l). (A (L l) \beta' ve)) ls) \)
  by (rule Un-mono[of \text{ran} \ve \text{ran} \ve, OF \text{subset-refl}] auto)
  finally
  have \( d \in \text{ran} \ ve \cup \text{set} \ (map \ (\lambda v.l). (A (L l) \beta' ve)) ls) \) 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 \( \forall b' \in \text{ran} \ b. \ Q b' \)
  shows \( \forall b' \in \text{contours-in-d} \ (A f \beta ve). \ Q b' \)

unfolding contours-in-ve-def

proof(cases \( f \))
  case (\( R - \text{var} \))
  thus \( ?\text{thesis} \)

proof (cases \( \beta \) (\( \text{fst} \) \( \text{var} \))
  case None with \( R \) show \( ?\text{thesis} \) by simp next
  case (Some cnt) show \( ?\text{thesis} \)

proof (cases \( ve \) (\( \text{var}, \text{cnt} \))
  case None with Some \( R \) show \( ?\text{thesis} \) by simp next
  case (Some \( d \))
  hence \( d \in \text{ran} \ ve \) unfolding ran-def by blast
  thus \( ?\text{thesis} \) by blast

unfolding contours-in-ve-def

by auto
qed
qed next

\begin{itemize}
\item \texttt{case} \texttt{(L l)} \texttt{thus }\texttt{thesis using }\forall b' \in \texttt{ran }\beta.\ Q\ b'\ \texttt{by simp next}
\item \texttt{case} \texttt{C} \texttt{thus }\texttt{thesis by simp next}
\item \texttt{case} \texttt{P} \texttt{thus }\texttt{thesis by simp}
\end{itemize}

\section{The Proof}

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

We would like to show an auxiliary result about the contour counter passed to $F$ and $C$ (such that it is an unused counter when passed to $F$ 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, \texttt{HOLCF} provides a number of introduction lemmas that, together with some additions in \texttt{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 the desired property. Here, we break the proof into the various cases that occur in the definitions of $F$ and $C$ and use the induction hypotheses.

\begin{lemma}
\texttt{cc-single-valued'}:
\begin{align*}
& \forall b' \in \texttt{contours-in-ve} \texttt{ ve}.\ b' < b \\
& \land \forall b' \in \texttt{contours-in-d} \texttt{ d}.\ b' < b \\
& \land \forall d' \in \texttt{set ds}.\ \forall b' \in \texttt{contours-in-d} \texttt{ d'.}\ b' < b \\
& \Rightarrow \texttt{single-valued} (F· (\texttt{Discr} (d,ds,ve,b))) \\
& \land (\forall ((\texttt{lab},\beta),t) \in F· (\texttt{Discr} (d,ds,ve, b)).\ \exists b'.\ b' \in \texttt{ran }\beta \land b \leq b')
\end{align*}
\end{lemma}

\begin{lemma}
\texttt{and'}:
\begin{align*}
& b \in \texttt{ran }\beta' \\
& \land \forall b' \in \texttt{ran }\beta'.\ b' \leq b \\
& \land \forall b' \in \texttt{contours-in-ve} \texttt{ ve}.\ b' \leq b \\
& \Rightarrow \texttt{single-valued} (C· (\texttt{Discr} (e,\beta',ve,b))) \\
& \land (\forall ((\texttt{lab},\beta),t) \in C· (\texttt{Discr} (e,\beta',ve, b)).\ \exists b'.\ b' \in \texttt{ran }\beta \land b \leq b')
\end{align*}
\end{lemma}
proof (induct arbitrary; d ds ve b ∈ β' rule: evalF-evalC-induct)

next

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 }

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 Next.prems(1) Next.prems(β)]
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)))
proof (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

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

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

Case If (true branch)

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

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

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

\(\text{fix } cp2 \text{ cp1 i cntf cntt}\)
\(\text{assume } \forall b' \in \text{contours-in-d cntt. } b' < b\)
hence \( b\text{-dom-d} : \forall b' \in \text{contours-in-d} \) \( \text{cntt} \cdot b' < nb b \) \( cp1 \) \text{ by auto}

have \( b\text{-dom-ds} : \forall d' \in \text{set}[], \forall b' \in \text{contours-in-d} d' \cdot b' < nb b \) \( cp1 \) \text{ by auto}

have \( b\text{-dom-ve} : \forall b' \in \text{contours-in-ve} \cdot b' < nb b \) \( cp1 \) \text{ using Next.prems(1) by auto}

\{
  \text{fix} t
  \begin{align*}
  \text{assume} & \quad ((cp1, [cp1 \mapsto b]), t) \in \text{evalF} \cdot (\text{Discr} \ (\text{cnttt}, [], ve, nb b \) \( cp1 \)) \\
  \text{hence} & \quad \text{False by} \quad (\text{auto dest:hyps-F-b}[OF b\text{-dom-ve} b\text{-dom-d} b\text{-dom-ds}])
  \end{align*}
\}

with \( \text{Next.prems(1)}[OF b\text{-dom-ve} b\text{-dom-d} \) \( b\text{-dom-ds}, \text{THEN conjunct1}] \)

show single-valued \((\text{evalF} \cdot (\text{Discr} \ (\text{cnttt}, [], ve, nb b \) \( cp1 \)))

\quad \cup \quad \{((cp1, [cp1 \mapsto b]), cntt]\}

\text{by} \quad (\text{auto intro:single-valued-insert})

\text{fix} \( lab \beta t \)

\text{assume} \quad ((\text{lab, } \beta), t) \in \text{evalF} \cdot (\text{Discr} \ (\text{cnttt}, [], ve, nb b \) \( cp1 \))

thus \( \exists b', b' \in \text{ran} \beta \land b \leq b' \)

\text{by} \quad (\text{auto dest: hyps-F-b}[OF b\text{-dom-ve} b\text{-dom-d} b\text{-dom-ds}])

\text{qed}

\text{next}

\text{case} \( (2 \) \( \text{ve} b c \beta') \)

\text{thus} \ ?\text{case}

\text{proof} \quad (\text{cases c, auto simp add:HOL.Let-def simp del:Un-insert-left Un-insert-right evalV.simps})

\text{Case App}

\text{fix} \( \text{lab'} f \) \( vs \)

\text{have} \quad \text{prem}2' : \forall b' \in \text{ran} \beta'. b' < nb b \) \( \text{lab'} \) \text{ using Next.prems(2) by auto}

\text{have} \quad \text{prem}3' : \forall b' \in \text{contours-in-ve} \cdot b' < nb b \) \( \text{lab'} \) \text{ using Next.prems(3) by auto}

\text{note} \text{ c-in-e = contours-in-ve}[OF prem3' prem2']

\text{have} \quad \text{b-dom-d} : \forall b' \in \text{contours-in-d} \cdot (\text{evalV} f \beta' \ve), b' < nb b \) \( \text{lab'} \) \text{ by (rule c-in-e)}

\text{have} \quad \text{b-dom-ds} : \forall d' \in \text{set} \cdot (\text{map} \ (\lambda v . \text{evalV} v \beta' \ve) \ vs), \forall b' \in \text{contours-in-d} d'. b' < nb b \) \( \text{lab'} \)

\text{using} \ text{ c-in-e by auto}

\text{have} \quad \text{b-dom-ve} : \forall b' \in \text{contours-in-ve} \cdot b' < nb b \) \( \text{lab'} \) \text{ by (rule prem3')}

\text{have} \quad \forall y. ((\text{lab'}, \beta'), y) \not\in \text{evalF} \cdot (\text{Discr} \ (\text{evalV} f \beta' \ve, \text{map} \ (\lambda v . \text{evalV} v \beta' \ve) \ vs, ve, nb b \) \( \text{lab'}))

\text{proof}(\text{rule allI, rule notI})

\text{fix} y \text{ assume} \quad ((\text{lab'}, \beta'), y) \in \text{evalF} \cdot (\text{Discr} \ (\text{evalV} f \beta' \ve, \text{map} \ (\lambda v . \text{evalV} v \beta' \ve) \ vs, ve, nb b \) \( \text{lab'}))

\text{hence} \quad \exists b' : b' \in \text{ran} \beta' \land nb b \) \( \text{lab'} \) \leq b'

\text{by} \quad (\text{auto dest: hyps-F-b}[OF b\text{-dom-ve} b\text{-dom-d} b\text{-dom-ds}])

\text{thus} \quad \text{False using prem}2' \text{ by (auto iff:less-le-not-le)}

\text{qed}

with \( \text{hyps-F-sv}[OF b\text{-dom-ve} b\text{-dom-d} b\text{-dom-ds}] \)

\text{show} \quad \text{single-valued} \ ((\text{evalF} \cdot (\text{Discr} \ (\text{evalV} f \beta' \ve, \text{map} \ (\lambda v . \text{evalV} v \beta' \ve) \ vs, ve, nb b \) \( \text{lab'}))}
\(\text{lab}()\) \\
\(\cup \{((\text{lab}', \beta'), \text{evalV f } \beta' \text{ ve})\}\) \\
by(auto intro:single-valued-insert)

fix lab \(\beta\) t 
assume ((lab, \(\beta\)), t) \(\in\) (evalF·(Discr (evalV f \(\beta'\) ve, map (\(\lambda v.\) evalV v \(\beta'\) ve) vs, ve, nb b lab'))) 
thus \(\exists b', b' \in\) ran \(\beta \land b \leq b'\) 
by (auto dest: hyps-C-sv OF b-dom-ve b-dom-d b-dom-ds)
next

Case Let 
fix lab' ls e' 
have prem2': \(\forall b' \in\) ran (\(\beta'(\text{lab}' \mapsto \text{nb b lab}')\)). \(b' \leq\) nb b lab' 
proof 
fix b' assume b' \(\in\) ran (\(\beta'(\text{lab}' \mapsto \text{nb b lab}')\)) 
hence b' \(\in\) ran \(\beta' \land b' =\) nb b lab' by (auto dest: ran-upd[THEN subsetD]) 
thus b' \(\leq\) nb b lab' using Next.prems(2) by auto 
qed 
have prem3': \(\forall b' \in\) contours-in-ve ve. \(b' \leq\) nb b lab' using Next.prems(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) ((\(\beta'(\text{lab}' \mapsto \text{nb b lab}')\)) ve)) ls)). \(b' \leq\) nb b lab' 
by (rule c-in-ve') 
have b-dom-beta: \(\forall b' \in\) ran (\(\beta'(\text{lab}' \mapsto \text{nb b lab}')\)). \(b' \leq\) nb b lab' by (rule prem2') 
have new: nb b lab' \(\in\) ran (\(\beta'(\text{lab}' \mapsto \text{nb b lab}')\)) by simp

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

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

lemma single-valued (\(\mathcal{PR}\) prog) 
unfolding evalCPS-def
by \((\text{subst HOL.Let-def})+, \text{rule cc-single-valued}[\text{THEN conjunct1}], \text{auto})\)
end

6. The abstract semantics is correct

theory AbsCFCorrect
  imports AbsCF ExCF ~~/src/Tools/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 \(\text{abs-<type>},\) 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 \textit{contour} type class by the abstraction functions and two equations involving the \(nb\) and \(b_0\) symbols.

\begin{verbatim}
class contour-a = contour +
  fixes abs-cnt :: contour => 'a
  assumes abs-cnt-nb[simp]: abs-cnt (nb b lab) = \(\hat{nb}\) (abs-cnt b) lab
  and abs-cnt-initial[simp]: abs-cnt(b_0) = \(b_0\)

instantiation unit :: contour-a
begin
definition abs-cnt - = ()
instance by default auto
end
\end{verbatim}

It would be unwieldly to always write out \(\text{abs-<type>}\ x\). 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 \(\text{abs-<type>}\) 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 im-
mediately. This is used in the following to set up the the symbol $|\cdot|$ for the family of
abstraction functions.

\begin{verbatim}
consts abs :: 'a ⇒ 'b (|\cdot|)

adhoc-overloading
  abs abs-cnt

definition abs-benv :: benv ⇒ 'c::contour-a benv
  where abs-benv β = map-option abs-cnt ◦ β

adhoc-overloading
  abs abs-benv

primrec abs-closure :: closure ⇒ 'c::contour-a closure
  where abs-closure (l,β) = (l,|\cdot|β )

adhoc-overloading
  abs abs-closure

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

adhoc-overloading
  abs abs-d

definition abs-venv :: venv ⇒ 'c::contour-a venv
  where abs-venv ve = (λ(v,b-a). ∪{(case ve (v,b) of Some d ⇒ |d| | None ⇒ {})} | b. |b| = b-a )

adhoc-overloading
  abs abs-venv

definition abs-ccache :: ccache ⇒ 'c::contour-a ccache
  where abs-ccache cc = (∪((e,β),d) ∈ cc . {{(e,abs-benv β), p) | p . p∈abs-d d}})

adhoc-overloading
  abs abs-ccache

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

adhoc-overloading
\( \text{abs abs-fstate} \)

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

adhoc-overloading
\( \text{abs abs-cstate} \)

6.2. Lemmas about abstraction functions

Some results about the abstractions functions.

lemma \( \text{abs-benv-empty simp}: |\text{empty}| = \text{empty} \)
unfolding \( \text{abs-benv-def by simp} \)

lemma \( \text{abs-benv-upd simp}: |\beta(c\mapsto b)| = |\beta| (c \mapsto |b|) \)
unfolding \( \text{abs-benv-def by simp} \)

lemma \( \text{the-elem-is-Proc} \):
assumes \( \text{isProc cnt} \)
shows \( \text{the-elem } |\text{cnt}| \in |\text{cnt}| \)
using \( \text{assms by (cases cnt)auto} \)

lemma \( \text{simp}: |\{\}| = \{\} \)
unfolding \( \text{abs-ccache-def by auto} \)

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

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

6.3. Approximation relation

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

consts \( \text{approx :: 'a } \Rightarrow 'a \Rightarrow \text{bool (- \subseteq -)} \)

definition \( \text{venv-approx :: 'c } \text{venv } \Rightarrow 'c } \text{venv } \Rightarrow \text{bool} \)
where \( \text{venv-approx = smap-less} \)

adhoc-overloading
\( \text{approx venv-approx} \)

definition \( \text{ccache-approx :: 'c } \text{ccache } \Rightarrow 'c } \text{ccache } \Rightarrow \text{bool} \)
where \( \text{ccache-approx = less-eq} \)
ad hoc overloading
approx 
cache-approx

definition d-approx :: 'c 'd ⇒'c 'd ⇒ bool
where d-approx = less-eq

ad hoc overloading
approx d-approx

definition ds-approx :: 'c 'd list ⇒'c 'd list ⇒ bool
where ds-approx = list-all2 d-approx

ad hoc overloading
approx ds-approx

inductive fstate-approx :: 'c fstate ⇒'c fstate ⇒ bool
where [ [ ve ⪅ ve' ; ds ⪅ ds' ] ] ⇒ fstate-approx (proc,ds,ve,b) (proc,ds',ve',b)

ad hoc overloading
approx fstate-approx

inductive cstate-approx :: 'c cstate ⇒'c cstate ⇒ bool
where [ [ ve ⪅ ve' ] ] ⇒ cstate-approx (c,β,ve,b) (c,β,ve',b)

ad hoc overloading
approx cstate-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-approx-trans[trans]:
fixes ve1 ve2 ve3 :: 'c venv
shows [ [ ve1 ⪅ ve2 ; ve2 ⪅ ve3 ] ] ⇒ (ve1 ⪅ ve3)
unfolding venv-approx-def by (rule smap-less-trans)

lemma abs-venv-union: |ve1 ++ ve2| ⪅ |ve1| ∪ |ve2|
by (auto simp add: venv-approx-def smap-less-def abs-venv-def smap-union-def, split option.split-asm, auto)

lemma abs-venv-map-of-rev: |map-of (rev l)| ⪅ l. (map (λ(v,k). [[v → k]]) l)
proof (induct l)
case Nil show ?case unfolding abs-venv-def by (auto simp: venv-approx-def smap-less-def)
next
case (Cons a l)
obtain \( v, k \) where \( a = (v, k) \) by (rule prod.exhaust)
hence \(|map-of (rev (a\#1))| \leq (|v \mapsto k|) \cup |map-of (rev l)|) :: 'a \venv
by (auto intro: abs-venv-union)
also
have ... \( \leq |v \mapsto k| \cup \bigcup (\{ \{v, b\} := |d|\}) \)
by (auto intro!: smap-Union-union)
also
have ... \( = \bigcup (\{ \mapsto (\lambda (v,k). [v \mapsto k]) \}) \)
using \( (a = (v,k))\)
by auto
finally
show ?case .
qed

lemma abs-venv-map-of: \(|map-of l| \leq \bigcup \mapsto (\lambda (v,k). [v \mapsto k]) \) \( l \)
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 \( x :: 'c \ccache \)
shows \( \{ \} \leq \bigcup \) \( x \)
unfolding \( \{ \} \leq x \) by simp

lemmas ccache-approx-trans[trans] = subset-trans[where \( a = ((\text{label} \times \ 'c \ \text{benv}) \times 'c \ \text{proc}), \)
folded ccache-approx-def]
lemmas Un-mono-approx = Un-mono[where \( a = ((\text{label} \times 'c \ \text{benv}) \times 'c \ \text{proc}), \)
folded ccache-approx-def]
lemmas Un-upper1-approx = Un-upper1[where \( a = ((\text{label} \times 'c \ \text{benv}) \times 'c \ \text{proc}), \)
folded ccache-approx-def]
lemmas Un-upper2-approx = Un-upper2[where \( a = ((\text{label} \times 'c \ \text{benv}) \times 'c \ \text{proc}), \)
folded ccache-approx-def]

lemma abs-ccache-union: \(|c1 \cup c2| \leq |c1| \cup |c2|
unfolding ccache-approx-def abs-ccache-def by auto

lemma d-approx-empty[simp]: \( \{ \} \leq \{d::'c \ d\}
unfolding d-approx-def by simp

lemma ds-approx-empty[simp]: \( \{ \} \leq \)
unfolding ds-approx-def by simp
6.5. Lemma 7

Shivers’ lemma 7 says that $\hat{A}$ safely approximates $A$.

**lemma lemma7:**
- **assumes** $|vc::venv| \subseteq vc-a$
- **shows** $|A f \beta ve| \subseteq \hat{A} f |\beta| ve-a$

**proof (cases f)**
- **case** $(R - v)$
  - **from** `assms` **have** `assm'` $\forall v b. \text{case-option } \{ \text{abs-d } (ve (v,b)) \subseteq vc-a (v,|b|)$
    - **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
      - **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 $\hat{F}$ safely approximates $F$ and that $\hat{C}$ safely approximates $C$. 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** $cont (\lambda x. \text{abs-ccache}(f x))$

**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** $fstate-a :: \ c::contour-a fstate$ and $cstate-a :: \ c::contour-a cstate$
- **shows** $|fstate| \subseteq fstate-a \Longrightarrow |F-(Discr fstate)| \subseteq \hat{F} (Discr fstate-a)$
- and $|cstate| \subseteq cstate-a \Longrightarrow |C-(Discr cstate)| \subseteq \hat{C} (Discr cstate-a)$

**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 (\texttt{\textit{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 \(\text{abs-d: the-elem} |d| = proc\)
  and \(\text{abs-ds: map} \text{abs-d} ds \subseteq ds-a\)
  and \(\text{abs-ve:} |ve| \subseteq ve-a\)
  and \(\text{abs-b:} |b| = b-a\)
  using \(1\) by (auto elim:\text{fstate-approx.cases})

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

  from \(\text{fstate fstate-a abs-d abs-ds abs-ve abs-ds dslength}\)
  show ?case
  proof (cases \(fstate\) rule:\text{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: \(\text{length} vs = \text{length} ds-a\)

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

  moreover

  \{ have \(|ve(map (\lambda v. (v, b)) vs \mapsto) ds)|
    \(\subseteq |ve| \cup. |map-of (\text{rev} (map (\lambda v. (v, b)) vs) ds))|\)
    unfolding map-upds-def by (intro abs-venv-union)
  also
  have \(\ldots \subseteq ve-a \cup. (\cup. (map (\lambda (v,k). ||(v \mapsto k)||) (zip (map (\lambda v. (v, b)) vs) ds)))|\)
    using abs-ve abs-venv-map-of-rev
    by (auto intro:smap-union-mono simp add:venv-approx-def)
  also
  have \(\ldots = ve-a \cup. (\cup. (map (\lambda (v,y). ||(v,b) \mapsto y||) (zip vs ds)))|\)
    by (auto simp add: zip-map1 o-def split-def)
  also
  have \(\ldots \subseteq ve-a \cup. (\cup. (map (\lambda (v,y). \{(v,b-a) := y\}.) (zip vs ds-a)))|\)
  proof --
from abs-b abs-ds
have list-all2 venv-approx (map (λ(v, y). |[(v, b) → y]|) (zip vs ds))
  (map (λ(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(map (λv. (v, b)) vs [→] ds)]
  ⊆ ve-a ∪ (⋃ (map (λ(v, y). {(v, b-a) := y}) (zip vs ds-a))).
}
ultimately
have prem: [(c, β(lab → b), ve(map (λv. (v, b)) vs [→] ds), b)]
  ⊆ (c, |β|lab → b-a), ve-a ∪ (⋃ (map (λ(v, y). {(v, b-a) := y}) (zip vs ds-a))), b-a)
using abs-b
by(auto intro:cstate-approx.intros simp add: abs-cstate.simps)
show [evalC-(Discr (c, β(lab → b), ve(map (λv. (v, b)) vs [→] ds), b))]
  ⊆ F.(Discr (PC (Lambda lab vs c, |β|), ds-a, ve-a, 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| ⊆ 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| ⊆ cnt-a
  unfolding ds-approx-def
  by (cases ds-a rule:list.exhaust[OF - list.exhaust[OF - list.exhaust, of - λ- x. x], of - λ-x. x])
(auto simp add:ds-approx-def)

have new-elem: |{((lab, [lab → b]), cnt)}| ⊆ |{(lab, [lab → b-a]), cont} |cont. cont ∈ cnt-a|
  using abs-cnt and abs-b
by (auto simp add:ccache-approx-def d-approx-def)

have prem: [(cnt, [DI (a1 + a2)], ve, nb b lab)] ⊆
  (the-elem [cnt], [[]], ve-a, nb b-a lab)
using abs-ve and abs-b
by (auto intro:fstate-approx.intros simp add:ds-approx-def)

have [(evalF-(Discr (cnt, [DI (a1 + a2)], ve, nb b lab)))]
  ⊆ F.(Discr (the-elem [cnt], [[]], ve-a, nb b-a lab))
by (rule Next.hyps(1)[OF prem])

32
also have \( \preceq (\bigcup c\in c\in cnt-a. \hat{\mathcal{F}}(\text{Discr (cnt, [\{\}], ve-a, nb b-a lab))) \]
using abs-cnt
finally
have \((\text{eval}(\text{Discr (cnt, [DI (a1 + a2]), ve, nb b lab}))\]
\( \subseteq (\bigcup c\in c\in cnt-a. \hat{\mathcal{F}}(\text{Discr (cnt, [\{\}], ve-a, nb b-a lab))) \.

have \((\text{eval}(\text{Discr (cnt, [DI (a1 + a2]), ve, nb b lab}))\]
\( \cup \{(\text{lab}, [\text{lab} \mapsto b]), cnt)\}\]
\( \subseteq (\text{eval}(\text{Discr (cnt, [DI (a1 + a2]), ve, nb b lab}))\]
\( \cup \{(\text{lab}, [\text{lab} \mapsto b]), cnt)\}\)
by (rule abs-ccache-union)
also
have \( \subseteq (\bigcup c\in c\in cnt-a. \hat{\mathcal{F}}(\text{Discr (cnt, [\{\}], ve-a, nb b-a lab))) \]
\( \cup \{(\text{lab}, [\text{lab} \mapsto b]), cnt)\}\]
by (rule Un-mono-approx[OF old-elems new-elem])
finally
show \((\text{insert} ([\text{lab}, [\text{lab} \mapsto b]), cnt)\]
\( \text{eval}(\text{Discr (cnt, [DI (a1 + a2]), ve, nb b lab}))\]
\( \subseteq \hat{\mathcal{F}}(\text{Discr (PP (prim.Plus lab), ds-a, ve-a, b-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| ] \leq ds-a
then obtain v-a cntt-a cntf-a where ds-a: ds-a = [v-a, cntt-a, cntf-a] and abs-cntt: \(|cntt| \leq cntt-a
and abs-cntf: \(|cntf| \leq cntf-a
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)

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

have new-elem: \([(\text{the-elem} |\text{cntt}|, [], ve, nb b ?c))\]
\( \subseteq (\text{eval (\text{Discr (cnt, [\{\}], ve-a, nb b-a ?c)\)
using abs-ve and abs-b by (auto intro:fstate-approx-intros)
Case If (false branch). We use schematic variable to keep this similar to the true branch.

| evalF (Discr (?cnt, [], ve, nb b ?c)) |
| \[\approx \hat{F} (Discr (the-elem (?cnt), [], ve-a, \widehat{nb} b-a ?c)) \]
| by (rule Next.hyps(1)[OF prem]) |

also have ... \[\approx (\bigcup cnt \in ?cnt-a. \hat{F} (Discr (cnt, [], ve-a, \widehat{nb} b-a ?c))) \]

using abs-cntt and abs-cntf

finally
have old-elems: | evalF (Discr (?cnt, [], ve, nb b ?c)) |
| \[\approx \bigcup \{\{?c, [?c \mapsto b]\}, ?cnt\}\] |
| \[\approx \bigcup \{\{?c, [?c \mapsto b]\}, ?cnt\}\] |
| by (rule abs-ccache-anion) |

also have ...
| \[\approx \bigcup \{\{?c, [?c \mapsto b]\}, ?cnt\}\] |
| \[\approx \bigcup \{\{?c, [?c \mapsto b]\}, ?cnt\}\] |
| by (rule Un-mono-approx[OF old-elems new-elem]) |

also have ...
| \[\approx \bigcup \{\{?c, [?c \mapsto b]\}, ?cnt\}\] |
| \[\approx \bigcup \{\{?c, [?c \mapsto b]\}, ?cnt\}\] |
| by (rule Un-upper1-approx|rule Un-upper2-approx) |

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

next

| fix ct cf v cntt cntf |
| assume isProc cntt |
| assume isProc cntf |
| assume abs-ds': [{}, |cntt|, |cntf|] \[\approx \] ds-a |
| then obtain v-a cntt-a cntf-a where ds-a: ds-a = [v-a, cntt-a, cntf-a] |
| and abs-cntt: |cntt| \[\approx\] cntt-a |
| and abs-cntf: |cntf| \[\approx\] cntf-a |
| by (cases ds-a rule:list.exhaust[OF - list.exhaust[OF - list.exhaust, of - - λ· x. 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-elem: \( \{((?c, [?c \mapsto b]), ?cnt)\} \subseteq \{((?c, [?c \mapsto b-a]), cont) \mid \text{cont. cont} \in ?cnt-a\} \)
using abs-cntt and abs-cntf and abs-b
by (auto simp add:ccache-approx-def d-approx-def)

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

have \([evalF:(\text{Discr} \mid ?cnt, [], ve, nb b ?c)]) \subseteq (\hat{\mathcal{F}}:(\text{Discr} \mid ?cnt, [], ve-a, nb b-a ?c)) \)
by (rule Next.hyps(1)[OF prem])
also have \( \cdots \subseteq (\bigcup \text{cnt} \in ?cnt-a. \hat{\mathcal{F}}:(\text{Discr} \mid ?cnt, [], ve-a, nb b-a ?c)) \)
using abs-cntt and abs-cntf

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

have \( [evalF:(\text{Discr} \mid ?cnt, [], ve, nb b ?c)) \cup \{((?c, [?c \mapsto b]), ?cnt)\} \subseteq [evalF:(\text{Discr} \mid ?cnt, [], ve, nb b ?c)) \cup \{((?c, [?c \mapsto b]), ?cnt)\} \)
by (rule abs-ccache-union)
also
have \( \cdots \subseteq (\bigcup \text{cnt} \in ?cnt-a. \hat{\mathcal{F}}:(\text{Discr} \mid ?cnt, [], ve-a, nb b-a ?c)) \)
\( \cup \{((?c, [?c \mapsto b-a]), cont) \mid \text{cont. cont} \in ?cnt-a\} \)
by (rule Un-mono-approx[OF old-elems new-elem])
also
have \( \cdots \subseteq (\bigcup \text{cnt} \in ?cnt-a. \hat{\mathcal{F}}:(\text{Discr} \mid ?cnt, [], ve-a, nb b-a ct)) \)
\( \cup \{((ct, [ct \mapsto b-a]), cont) \mid \text{cont. cont} \in ?cntt-a\} \)
\( \cup (\bigcup \text{cnt} \in ?cntt-a. \hat{\mathcal{F}}:(\text{Discr} \mid ?cnt, [], ve-a, nb b-a cf)) \)
\( \cup \{((cf, [cf \mapsto b-a]), cont) \mid \text{cont. cont} \in ?cntf-a\} \)
by (rule Un-upper1-approx)rule Un-upper2-approx)
finally
show \( \text{insert} \{((?c, [?c \mapsto b]), ?cnt) \)
(\( \hat{\mathcal{F}}:(\text{Discr} \mid ?cnt, [], ve, nb b ?c)) \subseteq \)
by (auto simp del:a-evalF.simps)(auto simp del:a-evalF.simps)
qed
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: \( |ve| \leq ve-a \)
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 \( ?d = A f \beta ve \)
assume isProc \( ?d \)

have map \( (abs-d \ o (\lambda v. A v \beta ve)) vs \leq map (\lambda v. \hat{A} v \beta-a ve-a) vs \)
using abs-\( \beta \) and lemma7[OF abs-ve, OF - \( \beta \)]
by (auto intro!: list-all2I simp add:set-zip ds-approx-def)

hence \[ \hat{F}(Discr(\beta, map (\lambda v. \hat{A} v \beta-a ve-a) vs, ve-a, \tilde{nb} |b| lab)) \]
using abs-ve and abs-cnt-nb and abs-b
by -(rule Next.hyps(1),auto intro:fstate-approx.intros)

also have ... \( (\bigcup f'\in A f \beta-a ve-a. \hat{F}(Discr(f', map (\lambda v. \hat{A} v \beta-a ve-a) vs, ve-a, \tilde{nb} |b| lab))) \)
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:
\[ \hat{F}(Discr(\beta, map (\lambda v. \hat{A} v \beta-a ve-a) vs, ve-a, \tilde{nb} |b| lab))) \]
by auto

have new-elem: \( \{(\{lab, \beta\}, A f \beta ve)\} \)
using abs-\( \beta \) and lemma7[OF abs-ve]
by(auto simp add:ccache-approx-def d-approx-def)

have \[ \hat{F}(Discr(\beta, map (\lambda v. \hat{A} v \beta ve) vs, ve, nb b lab))) \]
Also have . . .

\( \bigcup \{ (\text{lab}, \beta, (A, f, \beta ve)) \mid evalF\cdot (\text{Discr}(A, f, \beta ve, \text{map}(\lambda v. A f, \beta ve) vs, ve, nb b lab)) \}
\)

by (rule abs-ccache-union)

by (rule Un-mono-approx[OF old-elems new-elem])

Finally

show \( (\text{insert} ((\text{lab}, \beta), (A f, \beta ve)) \mid evalF\cdot (\text{Discr}(A f, \beta ve, \text{map}(\lambda v. A f, \beta ve) vs, ve, nb b lab)) ) \)

using abs-\( \beta \)

by simp

Moreover

have \( |\beta(\text{lab} \mapsto nb b lab)| = \beta-a(\text{lab} \mapsto nb b lab) \)

using abs-\( \beta \) and abs-b

by simp

Case Let

fix lab binds c'

have \( |\beta(\text{lab} \mapsto nb b lab)| = \beta-a(\text{lab} \mapsto nb b lab) \)

using abs-\( \beta \) and abs-b

by simp

moreover

have \( |\text{map-of}(\text{map}(\lambda (v, l). ((v, nb b lab), DC (l, \beta(\text{lab} \mapsto nb b lab)))) \mid binds) | \leq \bigcup \{ (\text{map}(\lambda (v, l). \{(v, nb b lab) := \{PC (l, \beta-a(\text{lab} \mapsto nb b lab)) \mid binds) \}) \mid binds) \}

using abs-b and abs-\( \beta \)

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 \( ve ++ map-of \)

\( \text{(map}(\lambda (v, l). ((v, nb b lab), DC (l, \beta(\text{lab} \mapsto nb b lab)))) \mid binds) \leq \bigcup \{ (\text{map}(\lambda (v, l). \{(v, nb b lab) := \{PC (l, \beta-a(\text{lab} \mapsto nb b lab)) \mid binds) \}) \mid binds) \}

by (rule venv-approx-trans[OF abs-venv-union]
ultimately have \(\text{eval}_{C} \cdot \text{Discr} (c', \beta(lab \mapsto nb \ lab), \ve \cdot \map{\lambda (v, l). \{((v, nb b lab), DC (l, \beta(lab \mapsto nb b lab)))\}} \text{binds}, nb b lab))\)
\(\subseteq \widehat{C} \cdot \text{Discr} (c', \beta-a(lab \mapsto \widehat{nb} | b| lab), \ve-a \cup (
\map{\lambda (v, l). \{((v, \widehat{nb} | b| lab) := \{PC (l, \beta-a(lab \mapsto \widehat{nb} | b| lab))\}\}} \text{binds}),
\widehat{nb} | b| lab))\)
using abs-cnt-nb and abs-b
by 
\(\text{subst a-evalC} \cdot \text{simps} \cdot \text{auto simp add: HOL.Let-def simp del: a-evalC.simps}\)
qed

And finally, we lift this result to \(\widehat{\mathcal{R}}\) and \(\mathcal{R}\).

lemma lemma6: \(|\mathcal{PR}| \subseteq \widehat{\mathcal{PR}}|\)
unfolding evalCPS-def evalCPS-a-def
by 
\(\text{auto intro: lemma89 fstate-approx.intros simp del: evalF.simps a-evalF.simps simp add: ds-approx-def d-approx-def venve-approx-def}\)

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 \(\widehat{\mathcal{R}}\).
7.1. Non-branching case

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

\[ f \cdot x = g \cdot x \cup f \cdot (r \cdot x) \]

is given by

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

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:

\[ \text{insert} (n::\text{nat}) \{n<..\} = \{n..\} \]

**by** auto

**lemma** theorem10:

**fixes** \( g :: 'a::\text{cpo} \rightarrow 'b::\text{type} \text{ set} \text{ and} \ r :: 'a \rightarrow 'a \)

**shows** \( \text{fix}(\Lambda f \cdot x. \ g \cdot x \cup f \cdot (r \cdot x)) = (\Lambda x. (\bigcup i. g \cdot (r^i \cdot x))) \)

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

**case** \((fp \ x)\)

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

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

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

**by** auto

**also have** ...

**=** \( (\bigcup i\in\text{insert} \ 0 \ \{0<..\}. g \cdot (r^i \cdot x)) \)

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

finally

**show** \(?case\ by\ auto\)

**next**

**case** \((least \ f \ x)\)

**hence** expand: \( \forall x. f \cdot x = (g \cdot x \cup f \cdot (r \cdot x)) \)

**by** (**auto simp:** cfun-eq-iff)

**\{ fix** \( n \)

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

**proof** (**induct** \( n \))

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

**case** \((Suc\ n)\)

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

**by** (**simp**)

**also have** ...

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

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

**also have** ...

**=** \( (\bigcup i\in\text{insert} \ (Suc\ n) \ \{..n\}. g \cdot (r^i \cdot x)) \cup f \cdot (r^{Suc\ (Suc\ n)} \cdot x) \)

39
by auto
also have \( \ldots = (\bigcup i \in \{..Suc n\} . \, g \cdot (r^i \cdot x)) \cup f \cdot (r^Suc (Suc n) \cdot x) \)
by \((simp\ add:atMost-Suc)\)
finally show \(?case\).
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
\[
f \cdot x = g \cdot x \cup \bigcup_{a \in R \cdot x} f \cdot r.
\]
By moving to the power-set relatives of our function, e.g.
\[
g \cdot Y = \bigcup_{a \in A} g \cdot a \quad \text{and} \quad R \cdot Y = \bigcup_{a \in R} R \cdot a
\]
the equation becomes
\[
f \cdot Y = g \cdot Y \cup f \cdot (R \cdot Y)
\]
(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\ \Rightarrow \ 'a\ \Rightarrow \ 'b\\)
where \(\lf = (\Lambda S. \, (\bigcup y \in S . \, f \cdot y))\)

lemma powerset-lift-singleton[simp]:
\(\lf\{x\} = f \cdot x\)
unfolding powerset-lift-def by simp

lemma powerset-lift-union[simp]:
\(\lf(A \cup B) = f \cdot A \cup f \cdot B\)
unfolding powerset-lift-def by auto

lemma UNION-commute:\((\bigcup x \in A. \, \bigcup y \in B . \, P \cdot x \cdot y) = (\bigcup y \in B . \, \bigcup x \in A . \, P \cdot x \cdot y)\)
by auto
lemma powerset-lift-UNION:
\[(\bigcup x \in S. g(A x)) = g(\bigcup x \in S. A x)\]
unfolding powerset-lift-def by auto

lemma powerset-lift-iterate-UNION:
\[(\bigcup x \in S. (g)^i(A x)) = (g)^i(\bigcup x \in S. A x)\]
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 ⇒ 'b set and g :: 'a ⇒ 'b set and R :: 'a ⇒ 'a set
assumes \(\forall x. f x = g x \cup (\bigcup y \in R x. f y)\)
shows \(f S = g S \cup f(R S)\)
proof
  have \(f S = (\bigcup x \in S. f x)\) unfolding powerset-lift-def by auto
  also have \(\ldots = (\bigcup x \in S. g x \cup (\bigcup y \in R x. f y))\) apply (subst assms) by simp
  also have \(\ldots = g S \cup f(R S)\) by (auto simp add: powerset-lift-def)
  finally show ?thesis .

qed

Theorem 10 as it will be used in Theorem 12.

lemmas theorem10ps = theorem10[af g \[ 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::discrete-cpo ⇒ 'b::type set and R :: 'a ⇒ 'a set
assumes F-fix: \(F = \text{fix}(\Lambda x. g x \cup F(R x))\)
shows \(\text{fix}(\Lambda x. g x \cup (\bigcup y \in R x. f y)) = (\Lambda x. F\{x\})\)
proof (induct rule: fix-eqI[OF cfun-eqI cfun-belowI, case-names fp least])
  have \(F\text{-union} = (\Lambda x. \bigcup i. g((R)^i x))\)
    using F-fix by (simp)(rule theorem10ps)
  case (fp x)
    have \(g x \cup (\bigcup x' \in R x. F\{x'\}) = g\{x\} \cup F(R\{x\})\)
      unfolding powerset-lift-singleton
    by (auto simp add: powerset-distr UNION-commute F-fix)
  also have \(\ldots = F\{x\}\)
    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 ∪ (Union y∈R·x. f' y)) by simp
  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 ∪ (Union y∈R·x. f y))·x = g·(Union i.(R·x){i})
  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 \( \hat{\mathcal{F}} \) and \( \hat{\mathcal{C}} \): 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
actions into separate functions. These take as arguments the direct sum of \( \hat{\mathsf{fstate}} \) and
\( \hat{\mathsf{cstate}} \), 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 \( \hat{\mathsf{fstate}} \) + 'c \( \hat{\mathsf{cstate}} \)) discr ⇒ 'c \( \hat{\mathsf{ans}} \)
where abs-g-x = (case undiscr x of
  (Inl (PC (Lambda lab vs c, β), as, ve, b)) ⇒ {}
  | (Inl (PP (Plus c),[r-,cnts],ve,b)) ⇒
    let b' = \h b c;
    β = \[c \mapsto b\]
    in \{(c, β), cont | cont . cont ∈ cnts\}
  | (Inl (PP (prim.If ct cf),[r-, cntts, cntfs],ve,b)) ⇒
    (let b' = \h b ct;
     β = \[ct \mapsto b\]
\[
in \{ ((c, \beta), \text{cnt}) \mid \text{cnt} \in \text{cnts} \}
\]

\[
\bigcup
\]

\[
\begin{align*}
\text{let } b' & = \overline{nb} b \ cf; \\
\beta & = [c f \mapsto b] \\
in \{ ((c, \beta), \text{cnt}) \mid \text{cnt} \in \text{cntfs} \}
\end{align*}
\]

\[
| \text{Inl (AStop,[],-,-)) } & \Rightarrow \{ \} \\
| \text{Inl -) } & \Rightarrow \bot \\
| \text{Inr (App lab f vs,\beta,ve,b)) } & \Rightarrow \\
\text{let } \text{fs} & = \overline{\lambda} f \ \beta \ ve; \\
\text{as} & = \text{map} (\lambda v. \overline{\lambda} v \ \beta \ ve) \ vs; \\
b' & = \overline{nb} b \ \text{lab} \\
in \{ ((\text{lab}, \beta), f') \mid f' \in \text{fs} \} \\
| \text{Inr (Let lab ls c',\beta,ve,b)) } & \Rightarrow \{ \}
\end{align*}
\]

\textit{abs-R} gives the set of arguments passed to the recursive calls.

\textbf{fixrec abs-R} :: \((c:\text{contour} \ \text{fstate} + \ 'c \ \text{estate} \ \text{discr}) \ \text{disr} \rightarrow \ (c:\text{contour} \ \text{fstate} + \ 'c \ \text{estate} \ \text{discr} \ \text{set})\

\textbf{where} \text{abs-R} x = \text{(case undiscr x of} \\
\text{Inl (PC (Lambda lab vs c, \beta), as, ve, b)) } \Rightarrow \\
\text{if length vs = length as} \\
\text{then } \beta' = \beta (\text{lab} \mapsto b); \\
\text{ve}' = \text{ve} \cup (\bigcup (\text{map} (\lambda v. \overline{\lambda} v \ \beta \ ve) \ vs; \\
\text{in } \{ (\text{Discr (Inr (c,\beta',ve',b)))} \\
\text{else } \bot) \\
\text{(Inl (PP (Plus c),[-,-,cnts],ve,b)) } \Rightarrow \\
\text{let } b' & = \overline{nb} b \ c; \\
\beta & = [c \mapsto b] \\
in \{ \bigcup \text{cnts } . \{ (\text{Discr (Inl (\text{cnts},ve,b))}) \\
\text{(Inl (PP (prim.If ct cf),[-,-,cnts],ve,b)) } \Rightarrow \\
\text{let } b' & = \overline{nb} b \ ct; \\
\beta & = [ct \mapsto b] \\
in \{ \bigcup \text{cnts } . \{ (\text{Discr (Inl (\text{cnts},ve,b'))}) \\
\text{(Inl (AStop,[],-,-)) } & \Rightarrow \{ \} \\
| \text{Inl -) } & \Rightarrow \bot \\
\text{Inr (App lab f vs,\beta,ve,b)) } \Rightarrow \\
\text{let } \text{fs} & = \overline{\lambda} f \ \beta \ ve; \\
\text{as} & = \text{map} (\lambda v. \overline{\lambda} v \ \beta \ ve) \ vs; \\
b' & = \overline{nb} b \ \text{lab} \\
in \{ \bigcup f' \in \text{fs} . \{ (\text{Discr (Inl (f',as,ve,b'))}) \\
\text{Inr (Let lab ls c',\beta,ve,b)) } \Rightarrow \\
\text{}}
\]

43
let b' = \overline{nb} \, b \, \text{lab}; \\
\beta' = \beta \, (\text{lab} \mapsto b'); \\
ve' = ve \cup (\bigcup (\text{map} (\lambda (v, l). \{(v, b') := (\overline{A} (L l) \beta' \, ve)\}) \, \text{ls})) \\
in \{\text{Discr (Inr (c', \beta', ve', b'))}\}

The initial argument vector, as created by \overline{PR}.

**definition** initial-r :: prog \Rightarrow ('c::contour fstate + 'c cstate) discr \\
where initial-r prog = Discr (Inl (the-elem (\overline{A} (L prog) empty \{\}, [{AStop}], \{\}, \overline{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 \( \overline{F} \), or possible arguments to \( \overline{C} \) and of possible arguments.

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

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

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

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

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

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

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

**lemma** evalV-possible: \\
assumes f: f \in \overline{A} \, d \, \beta \, ve
and \( d: d \in \text{vals} p \)

and \( \text{ve}: \text{ve} \in \text{smaps-over} (\text{vars} p \times \text{UNIV}) (\text{proc-poss} p) \)

and \( \beta: \beta \in \text{maps-over} (\text{labels} p) \text{UNIV} \)

shows \( f \in \text{proc-poss} p \)

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

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

lemma arg-space-complete:
  \(\text{state} \in \text{arg-poss} p \implies \text{abs-R state} \subseteq \text{arg-poss} p\)
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 \(\text{state} = \text{Step}(2)\)
  show \(?case\)
  proof (cases \(\text{state}\))
  case (Discr \(\text{state}'\)) show \(?thesis\)
    proof (cases \(\text{state}'\))
      case (Inl \(\text{fstate}\)) show \(?thesis\)
        using Inl Discr \(\text{state}\)
        proof (cases \(\text{fstate rule: a-fstate-case, auto}\))

Case Lambda

fix \(l\) \(c\) \(\beta\) as \(\text{ve} b\)
assume Discr \((\text{Inl} (\text{PC} (\text{Lambda} l \text{ vs} c, \beta), \text{as}, \text{ve}, b)) \in \text{arg-poss} p\)
  hence lam: \(\text{Lambda} l \text{ vs} c \in \text{lambdas} p\)
and beta: \(\beta \in \text{maps-over} (\text{labels} p) \text{UNIV}\)
and \(\text{ve}: \text{ve} \in \text{smaps-over} (\text{vars} p \times \text{UNIV}) (\text{proc-poss} p)\)
and \(\text{as}: \text{as} \in \text{NList} (\text{Pow} (\text{proc-poss} p)) (\text{call-list-lengths} p)\)
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 \( \text{vs} : \text{set} \ \text{vs} \subseteq \text{vars} \ p \) 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 \( \text{ve} \cup \bigcup \text{map} \ (\lambda (v, y) . \{ (v, b) := y \}, \text{vs as}) \in \text{smaps-over} \ (\text{vars} \ p \times \text{UNIV}) \ (\text{proc-poss} \ p) \ (\text{is} \ ?\text{ve}' \in \text{-}) \)
  by (auto intro!: smaps-over-un [OF ve] smaps-over-Union smaps-over-singleton)
    (auto simp add: set-zip)

ultimately
have \( (c, \beta(l \mapsto b), \ ?\text{ve}', b) \in \text{cstate-poss} \ p \ (\text{is} \ \text{?cstate} \in \text{-}) \)
  unfolding cstate-poss-def by simp
thus \( \text{Discr} \ (\text{Inr} ?\text{cstate}) \in \text{arg-poss} \ p \)
  unfolding arg-poss-def by auto
next

Case Plus

fix \( \text{ve} \ b \ l \ v1 v2 \text{ cnts} \ \text{cnt} \)
assume Discr (Inl (PP (prim.Plus l), [v1, v2, cnts], ve, b)) \( \in \text{arg-poss} \ p \)
  and \( \text{cnt} \in \text{cnts} \)
hence \( \text{cnt} \in \text{proc-poss} \ p \)
  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 \( (\text{cnt}, [\{\}]), \text{ve}, \text{nb} b l \) \( \in \text{fstate-poss} \ p \)
unfolding fstate-poss-def by auto
thus \( \text{Discr} \ (\text{Inl} \ (\text{cnt}, [\{\}], \text{ve}, \text{nb} b l)) \in \text{arg-poss} \ p \)
  unfolding arg-poss-def by auto
next

Case If (true case)

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

case App

fix l d ds f
assume arg: Discr (Inr (App l d ds, β, ve, b)) ∈ arg-poss p
and f: f ∈ Â d β ve
hence c: App l d ds ∈ calls p
and d: d ∈ vals p
and ds: set ds ⊆ vals p
and \( \beta \in \text{maps-over} \) (labels) \( \text{UNIV} \)
and \( \text{ve} : \text{ve} \in \text{smaps-over} \) (vars \( p \times \text{UNIV} \)) (proc-poss \( p \))
by (auto simp add: \( \text{arg-poss-def} \) \( \text{cstate-poss-def} \) call-list-lengths-def dest: app1 app2)

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

have \( f \in \text{proc-poss} \) \( p \)
using \( f \) \( d \) \( \text{ve} \) \( \beta \) by (rule evalV-possible)

moreover have \( \text{map} (\lambda v. \hat{\beta} \) \( \text{ve} \) \( \beta \) \( \text{ve} \) \) \( ds \in \text{NList} \) (Pow (proc-poss \( p \))) (call-list-lengths \( p \))
using \( ds \) \( \text{len} \)
unfolding \( \text{NList-def} \) by (auto simp add: nList-def)

ultimately have \( f, \) \( \text{map} (\lambda v. \hat{\beta} \) \( \text{ve} \) \( \beta \) \( \text{ve} \) \( ds \) \) \( \in \) \( \text{fstate-poss} \) \( p \) \( (\text{is} \) \( ?\text{fstate} \) \( \in \) -)
using \( \text{ve} \)
unfolding \( \text{fstate-poss-def} \) by simp

thus \( \text{Discr} \) (Inl \( ?\text{fstate} \) \( \in \) \( \text{arg-poss} \) \( p \))
unfolding \( \text{arg-poss-def} \) by auto

next

Case Let

fix \( l \) \( \text{binds} \) \( c' \)
assume arg: \( \text{Discr} \) (Inr (Let \( l \) \( \text{binds} \) \( c' \), \( \beta \), \( \text{ve} \), \( b \)) \) (\( \text{is} \) \( ?\text{fstate} \) \( \in \) -)

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

have \( \beta' : \beta(l \mapsto \hat{b} b l) \) \( \in \text{maps-over} \) (labels) \( \text{UNIV} \) (\( \text{is} \) \( ?\beta' \) \( \in \) -)
by (auto intro: maps-over-upd[\( \text{OF} \] \( c \) \( \beta' \) \( \in \) -])

moreover have \( \text{ve} \cup \bigcup \text{map} (\lambda v, \text{lam}). \{ (v, \hat{b} b l) := \hat{A} (L \text{lam}) (\beta (l \mapsto \hat{b} b l)) \text{ve} \} \}

binds \( \in \text{smaps-over} \) (vars \( p \times \text{UNIV} \)) (proc-poss \( p \)) (\( \text{is} \) \( ?\text{ve}' \) \( \in \) -)
using \( \text{vars} \) \( \text{ls} \) \( \beta \)
by (auto intro!: \( \text{maps-over-un}[\text{OF} \text{ve}] \) \( \text{smaps-over-Union} \)
(auto intro!: \( \text{maps-over-singleton} \) \( \text{simp add: proc-poss-def} \))

ultimately have \( c', ?\beta', ?\text{ve}', \hat{b} b l) \) \( \in \text{cstate-poss} \) \( p \) (\( \text{is} \) \( ?\text{cstate} \) \( \in \) -)
using \( c' \) unfolding \( \text{cstate-poss-def} \) by simp
thus Discr (Inr ?cstate) ∈ arg-poss p
  unfolding arg-poss-def by auto
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 (⋃i. iterate i · f · {x}) ⊆ S
proof−
  fix i
  have iterate i · f · {x} ⊆ S
  proof(induct i)
    case 0 show ?case using ⟨x ∈ S⟩ 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
    intro!: imageI)
qed

8.2. A decomposition

The functions abs-g and abs-R are derived from \( \hat{F} \) and \( \hat{C} \). This connection has yet to expressed explicitly.
lemma \( \text{Un-commute-helper} : (a \cup b) \cup (c \cup d) = (a \cup c) \cup (b \cup d) \)

by auto

lemma \( \text{a-evalF-decomp} : \)

\[ \hat{F} = \text{fst} \left( \sum\text{-}\text{tup} \cdot \text{fix} \cdot (\Lambda f \cdot x. (\bigcup \{ y \in \text{abs-R} : f \cdot y \} \cup \text{abs-g} \cdot x)) \right) \]

apply (subst a-evalF-def)

apply (subst fix\text{-}transform\text{-}pair\text{-}sum)

apply (rule arg\text{-}cong \[ of - \cdot \lambda x. \text{fst} \left( \sum\text{-}\text{tup} \cdot \text{fix} \cdot x \right) \])

apply (simp)

apply (simp only: discr\text{-}app undiscr\text{-}Discr)

apply (rule cfun\text{-}eqI, rule cfun\text{-}eqI, simp)

apply (case\text{-}tac xa, rename\text{-}tac a, case\text{-}tac a, simp)

apply (case\text{-}tac aa rule: a\text{-}fstate\text{-}case)

apply (simp\text{-}all add: Un\text{-}commute\text{-}helper)

apply (case\text{-}tac aa)

apply (simp\text{-}all add: HOL\text{-}Let\text{-}def)

done

8.3. The iterative equation

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

lemma \( \text{a-evalF-iterative} : \)

\[ \hat{F} \cdot \text{Discr} \cdot x = \text{abs-g} \cdot (\bigcup i \cdot \text{iterate} \cdot i \cdot \text{abs-R} \cdot \{ \text{Discr} \cdot (\text{Inl} \cdot x) \}) \]

by (simp del: abs-R.simps abs-g.simps add: theorem12 Un\text{-}commute a\text{-}evalF\text{-}decomp)

lemma \( \text{a-evalCPS-iterative} : \)

\[ \hat{PR} \cdot \text{prog} = \text{abs-g} \cdot (\bigcup i \cdot \text{iterate} \cdot i \cdot \text{abs-R} \cdot \{ \text{initial-r} \cdot \text{prog} \}) \]

unfolding evalCPS-a-def and initial-r-def

by (subst a\text{-}evalF\text{-}iterative, simp del: abs-R.simps abs-g.simps evalV-a.simps)

end

Part III.
The auxiliary theories

9. Syntax tree helpers
This theory defines the sets \( \text{lambdas } p \), \( \text{calls } p \), \( \text{vars } p \), \( \text{labels } p \) and \( \text{prims } p \) 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.

\[
\begin{align*}
\text{fun & lambdas :: lambda } & \Rightarrow \text{ lambda set} \\
\text{and & lambdasC :: call } & \Rightarrow \text{ lambda set} \\
\text{and & lambdasV :: val } & \Rightarrow \text{ lambda set}
\end{align*}
\]

where \( \text{lambdas } (\text{Lambda } l \ vs \ c) = (\text{Lambda } l \ vs \ c) \cup \text{lambdasC } c \)

\[
\begin{align*}
\text{fun & lambdasC :: call } & \Rightarrow \text{ lambda set} \\
\text{and & lambdasV :: val } & \Rightarrow \text{ lambda set}
\end{align*}
\]

\[
\begin{align*}
\text{fun & calls :: lambda } & \Rightarrow \text{ call set} \\
\text{and & callsC :: call } & \Rightarrow \text{ call set} \\
\text{and & callsV :: val } & \Rightarrow \text{ call set}
\end{align*}
\]

where \( \text{calls } (\text{Lambda } l \ vs \ c) = \text{callsC } c \)

\[
\begin{align*}
\text{fun & vars :: lambda } & \Rightarrow \text{ var set} \\
\text{and & varsC :: call } & \Rightarrow \text{ var set} \\
\text{and & varsV :: val } & \Rightarrow \text{ var set}
\end{align*}
\]

where \( \text{vars } (\text{Lambda } - \ vs \ c) = \text{set } vs \cup \text{varsC } c \)

\[\]

\[
\begin{align*}
\text{fun & lambdas :: lambda } & \Rightarrow \text{ lambda set} \\
\text{and & lambdasC :: call } & \Rightarrow \text{ lambda set} \\
\text{and & lambdasV :: val } & \Rightarrow \text{ lambda set}
\end{align*}
\]

where \( \text{lambdas } (\text{Lambda } l \ vs \ c) = (\text{Lambda } l \ vs \ c) \cup \text{lambdasC } c \)

\[
\begin{align*}
\text{funcalls :: lambda } & \Rightarrow \text{ call set} \\
\text{and & callsC :: call } & \Rightarrow \text{ call set} \\
\text{and & callsV :: val } & \Rightarrow \text{ call set}
\end{align*}
\]

where \( \text{calls } (\text{Lambda } l \ vs \ c) = \text{callsC } c \)

\[
\begin{align*}
\text{fun & vars :: lambda } & \Rightarrow \text{ var set} \\
\text{and & varsC :: call } & \Rightarrow \text{ var set} \\
\text{and & varsV :: val } & \Rightarrow \text{ var set}
\end{align*}
\]

where \( \text{vars } (\text{Lambda } - \ vs \ c) = \text{set } vs \cup \text{varsC } c \)

\[\]

51
fun label :: lambda + call ⇒ label
where label (\text{Inl} (\text{Lambda} \ l \ -.)) = l
| label (\text{Inr} (\text{App} \ l \ -.)) = l
| label (\text{Inr} (\text{Let} \ l \ -.)) = l

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

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

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

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

fun vals :: lambda ⇒ val set
and valsC :: call ⇒ val set
and valsV :: val ⇒ val set
where vals (\text{Lambda} - vs \ c) = \text{valsC} \ c
| valsC (\text{App} - a \ as) = \text{valsV} \ a \cup \text{UNION} (\text{set as}) \text{valsV}
| valsC (\text{Let} - binds \ c') = \text{UNION} (\text{set binds}) (\lambda(-,l). \text{vals} \ l) \cup \text{valsC} \ c'
| valsV (L \ l) = \{L \ l\} \cup \text{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: \text{Lambda} \ l \ vs \ c \in \text{lambdas} \ x \Rightarrow \text{c} \in \text{calls} \ x
and \text{Lambda} \ l \ vs \ c \in \text{lambdasC} \ y \Rightarrow \text{c} \in \text{callsC} \ y

52
apply (induct rule: mutual-lambda-call-var-inducts)
apply auto
apply (case-tac c, auto)[I]
apply (rule-tac \(x=\langle a, b, ba\rangle\) in bexI, auto)
done

lemma
shows \(\text{labels}2\): \(\text{Lambda} l \; vs \; c \in \text{labels} \Rightarrow l \in \text{labels} x\)
and \(\text{Lambda} l \; vs \; c \in \text{labels}C \; y \Rightarrow l \in \text{labels}C \; y\)
and \(\text{Lambda} l \; vs \; c \in \text{labels}V \; z \Rightarrow l \in \text{labels}V \; z\)
and \(\forall z \in \text{set list}. \; \text{Lambda} l \; vs \; c \in \text{labels} \Rightarrow l \in \text{labels} \; (\text{snd} \; x)\)
and \(\text{Lambda} l \; vs \; c \in \text{labels} \; (\text{snd} \; t) \Rightarrow l \in \text{labels} \; (\text{snd} \; t)\)
apply (induct rule: mutual-lambda-call-var-inducts)
apply auto
apply (rule-tac \(x=\langle a, b, ba\rangle\) in bexI, auto)
done

lemma
shows \(\text{labels}3\): \(\text{Lambda} l \; vs \; c \in \text{labels} \Rightarrow \text{set} \; vs \subseteq \text{vars} \; x\)
and \(\text{Lambda} l \; vs \; c \in \text{labels}C \; y \Rightarrow \text{set} \; vs \subseteq \text{vars}C \; y\)
and \(\text{Lambda} l \; vs \; c \in \text{labels}V \; z \Rightarrow \text{set} \; vs \subseteq \text{vars}V \; z\)
and \(\forall z \in \text{set list}. \; \text{Lambda} l \; vs \; c \in \text{labels} \Rightarrow \text{set} \; vs \subseteq \text{vars} \; (\text{snd} \; x)\)
and \(\text{Lambda} l \; vs \; c \in \text{labels} \; (\text{snd} \; t) \Rightarrow \text{set} \; vs \subseteq \text{vars} \; (\text{snd} \; t)\)
apply (induct \(x\) and \(y\) and \(z\) and list and list2 and \(t\) rule: mutual-lambda-call-var-inducts)
apply auto
apply (erule-tac \(x=\langle aa, ba, bb\rangle\) in ballE)
apply (rule-tac \(x=\langle aa, ba, bb\rangle\) in bexI, auto)
done

lemma
shows \(\text{app}1\): \(\text{App} l \; d \; ds \in \text{calls} \Rightarrow d \in \text{vals} \; x\)
and \(\text{App} l \; d \; ds \in \text{calls}C \; y \Rightarrow d \in \text{vals}C \; y\)
and \(\text{App} l \; d \; ds \in \text{calls}V \; z \Rightarrow d \in \text{vals}V \; z\)
and \(\forall z \in \text{set list}. \; \text{App} l \; d \; ds \in \text{calls} \Rightarrow d \in \text{vals} \; (\text{snd} \; x)\)
and \(\text{App} l \; d \; ds \in \text{calls} \; (\text{snd} \; t) \Rightarrow d \in \text{vals} \; (\text{snd} \; t)\)
apply (induct \(x\) and \(y\) and \(z\) and list and list2 and \(t\) rule: mutual-lambda-call-var-inducts)
apply auto
apply (case-tac \(d\), auto)
apply (erule-tac \(x=\langle a, b, ba\rangle\) in ballE)
apply (rule-tac \(x=\langle a, b, ba\rangle\) in bexI, auto)
done

lemma

show app2: App l d ds ∈ calls x ⇒ set ds ⊆ vals x
  and App l d ds ∈ callsC y ⇒ set ds ⊆ valsC y
  and App l d ds ∈ callsV z ⇒ set ds ⊆ valsV z
  and ∀ z ∈ set list. App l d ds ∈ callsV z ⇒ set ds ⊆ valsV z
  and ∀ x ∈ set (list2 :: (var × lambda) list) . App l d ds ∈ calls (snd x) ⇒ set ds ⊆ vals (snd x)
  and App l d ds ∈ calls (snd (t :: var × lambda)) ⇒ set ds ⊆ 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

show let1: Let l binds c' ∈ calls x ⇒ l ∈ labels x
  and Let l binds c' ∈ callsC y ⇒ l ∈ labelsC y
  and Let l binds c' ∈ callsV z ⇒ l ∈ labelsV z
  and ∀ z ∈ set list. Let l binds c' ∈ callsV z ⇒ l ∈ labelsV z
  and ∀ x ∈ set (list2 :: (var × lambda) list) . Let l binds c' ∈ calls (snd x) ⇒ l ∈ labels (snd x)
  and Let l binds c' ∈ calls (snd (t :: var × lambda)) ⇒ l ∈ 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

show let2: Let l binds c' ∈ calls x ⇒ c' ∈ calls x
  and Let l binds c' ∈ callsC y ⇒ c' ∈ callsC y
  and Let l binds c' ∈ callsV z ⇒ c' ∈ callsV z
  and ∀ z ∈ set list. Let l binds c' ∈ callsV z ⇒ c' ∈ callsV z
  and ∀ x ∈ set (list2 :: (var × lambda) list) . Let l binds c' ∈ calls (snd x) ⇒ c' ∈ calls (snd x)
  and Let l binds c' ∈ calls (snd (t :: var × lambda)) ⇒ c' ∈ 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

show let3: Let l binds c' ∈ calls x ⇒ fst ' set binds ⊆ vars x
  and Let l binds c' ∈ callsC y ⇒ fst ' set binds ⊆ varsC y
and \( \text{Let } l \text{ binds } c' \in \text{calls} V z \mapsto \text{fst ' set binds } \subseteq \text{vars} V z \)
and \( \forall z \in \text{set list}. \text{Let } l \text{ binds } c' \in \text{calls} V z \mapsto \text{fst ' set binds } \subseteq \text{vars} V z \)
and \( \forall x \in \text{set} \text{ (list2 :: (var \times \text{lambda}) list). Let } l \text{ binds } c' \in \text{calls} (\text{snd } x) \mapsto \text{fst ' set binds } \subseteq \text{vars } (\text{snd } x) \)
and \( \text{Let } l \text{ binds } c' \in \text{calls} (\text{snd } (t:: \text{var} \times \text{lambda})) \mapsto \text{fst ' set binds } \subseteq \text{vars } (\text{snd } t) \)
apply (\text{induct } x \text{ and } y \text{ and } z \text{ and list and list2 and } t \text{ rule:mutual-lambda-call-var-inducts})
apply auto
apply (\text{erule-tac } x=((a, b), b) \text{ in } \text{ballE})
apply (\text{rule-tac } x=((a, b), b) \text{ in } \text{bexI, auto})
done

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

\text{lemma}
\text{shows vals1: } P \text{ prim } \in \text{vals } p \mapsto \text{prim } \in \text{prim } \text{p}
and \( \text{P prim } \in \text{valsC } y \mapsto \text{prim } \in \text{primC } y
and \( \text{P prim } \in \text{valsV } z \mapsto \text{prim } \in \text{primV } z
and \( \forall z \in \text{set list}. \text{P prim } \in \text{valsV } z \mapsto \text{prim } \in \text{primV } z
and \( \forall x \in \text{set} \text{ (list2 :: (var \times \text{lambda}) list). P prim } \in \text{vals } (\text{snd } x) \mapsto \text{prim } \in \text{prim } (\text{snd } x)
and \( \text{P prim } \in \text{vals } (\text{snd } (t:: \text{var} \times \text{lambda})) \mapsto \text{prim } \in \text{prim } (\text{snd } t)
apply (\text{induct } \text{rule:mutual-lambda-call-var-inducts})
apply auto
apply (\text{erule-tac } x=((a, b), b) \text{ in } \text{ballE})
apply (\text{rule-tac } x=((a, b), b) \text{ in } \text{bexI, auto})
done

\text{lemma}
\text{shows vals2: } R \text{ l var } \in \text{vals } p \mapsto \text{var } \in \text{vars } p
and \( \text{R l var } \in \text{valsC } y \mapsto \text{var } \in \text{varsC } y
and \( \text{R l var } \in \text{valsV } z \mapsto \text{var } \in \text{varsV } z
and \( \forall z \in \text{set list}. \text{R l var } \in \text{valsV } z \mapsto \text{var } \in \text{varsV } z
and \( \forall x \in \text{set} \text{ (list2 :: (var \times \text{lambda}) list). R l var } \in \text{vals } (\text{snd } x) \mapsto \text{var } \in \text{vars } (\text{snd } x)
and \( \text{R l var } \in \text{vals } (\text{snd } (t:: \text{var} \times \text{lambda})) \mapsto \text{var } \in \text{vars } (\text{snd } t)
apply (\text{induct } \text{rule:mutual-lambda-call-var-inducts})
apply auto
apply \( \text{erule-tac } x=((a, b), ba) \text{ in } \text{ballE} \)
apply \( \text{rule-tac } x=((a, b), ba) \text{ in } \text{bexI, auto} \)
done

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

definition nList :: 'a set => nat => 'a list set
where nList A n ≡ \{ l. set l ≤ A ∧ length l = n \}

lemma finite-nList[intro]:
  assumes finA: 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) = (split op #) ' (A × nList A n) (is ?lhs = ?rhs)
  proof (rule subset-antisym[OF subsetI subsetI])
  fix l assume l ∈ ?lhs thus l ∈ ?rhs by (cases l, auto simp add:nList-def)
next
fix l assume l ∈ ?rhs thus l ∈ ?lhs by (auto simp add:nList-def)
qed thus finite ?lhs using finA and finn by auto
qed

definition NList :: 'a set => nat set => 'a list set
where NList A N ≡ \{ n ∈ N. nList A n \}

lemma finite-Nlist[intro]:
  [ finite A; finite N ] \Rightarrow finite (NList A N)
unfolding NList-def using assms by auto

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

**lemma** finite-call-list-lengths[simp]: finite (\( \text{call-list-lengths} p \))

**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 \( \forall x \, y . \ [(x,y) \in \text{rel}; x=a] \Rightarrow y = b \)

shows single-valued (insert (a,b) rel)

**using** asms

by (auto intro:single-valuedI dest:single-valuedD)

Lemmas about \( \text{ran} \), the range of a finite map.

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

**unfolding** ran-def by auto

**lemma** ran-map-of: ran (map-of \( xs \)) \( \subseteq \) \( \text{snd } \set \text{xs} \)

by (induct \( xs \))(auto simp del:fun-upd-apply dest: ran-upd[THEN subsetD])

**lemma** ran-concat: 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 ran (map-upds \( m \) \( ks \) \( vs \)) \( \subseteq \) \( \text{ran } m \cup \text{set } vs \)

**proof**

have ran (map-upds \( m \) \( ks \) \( vs \)) \( \subseteq \) ran (\( m++\text{map-of } (\text{rev } \text{zip } ks \text{ vs})) \)

**unfolding** map-upds-def by simp

also have \( \ldots \) \( \subseteq \) ran m \( \cup \) ran (map-of (rev (\( \text{zip } ks \text{ vs})))) by (rule ran-concat)

also have \( \ldots \) \( \subseteq \) ran m \( \cup \) \( \text{snd } \cdot \text{set } (\text{rev } \text{zip } ks \text{ vs}) \)

57
by (intro Un-mono[of ran m ran m] subset-refl ran-map[of])
also have \( \subseteq \) ran m \( \cup \) set vs
by (auto intro:Un-mono[of ran m ran m] subset-refl simp del:subset-refl simp add:map-map)}
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 \( \text{fst} / \text{snd} \)

lemma map-fst-zip: \( \text{length} \ (xs) = \text{length} \ (ys) \implies \text{map} \ \text{fst} \ (\text{zip} \ (xs) \ (ys)) = \ (xs) \)
apply (induct \( \text{xs} \) \( \text{ys} \) 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 \( \text{xs} \) \( \text{ys} \) 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 :: (\( 'a =\Rightarrow 'b \) set) \Rightarrow 'a set where
sdom \( m \) = \( \{a. \text{m} \ (a) \sim \{\}\} \)

definition
sran :: (\( 'a =\Rightarrow 'b \) set) \Rightarrow 'b set where
sran \( m \) = \( \{b. \text{EX} \ a. \ b \in \text{m} \ (a)\} \)

lemma sranI: \( b \in \text{m} \ (a) \implies \ b \in \text{sran} \ (m) \)
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 \texttt{smap-union} :: (\texttt{a::type} \Rightarrow \texttt{b::type set}) \Rightarrow (\texttt{a} \Rightarrow \texttt{b set}) \Rightarrow (\texttt{a} \Rightarrow \texttt{b set}) (\cdot \cup \cdot)
\text{where} \quad \texttt{smap1 \cup \texttt{smap2} k = \texttt{smap1} k \cup \texttt{smap2} k}

primrec \texttt{smap-Union} :: (\texttt{a::type} \Rightarrow \texttt{b::type set}) \texttt{list} \Rightarrow \texttt{a} \Rightarrow \texttt{b set} \{\cup\cdot\}\n\text{where} \quad \{\texttt{simp}\}:: \texttt{\cup} \cdot \texttt{[]} = \{\cdot \}
\quad | \texttt{\cup} \cdot \texttt{\{m#ms\}} = \texttt{m} \cup \texttt{\cup} \cdot \texttt{ms}

definition \texttt{smap-singleton} :: \texttt{a::type} \Rightarrow \texttt{b::type set} \Rightarrow \texttt{a} \Rightarrow \texttt{b set} \{\cdot := \cdot\}\n\text{where} \quad \{\texttt{simp}\}: \texttt{\cup} \cdot \texttt{\{k := vs\}} = \{\cdot \} \cup \texttt{\cup} \cdot \texttt{\{k := vs\}}

definition \texttt{smap-less} :: \texttt{(a \Rightarrow \texttt{b set}) \Rightarrow (a \Rightarrow \texttt{b set}) \Rightarrow \texttt{bool}} (\cdot \subseteq \cdot \}\n\text{where} \quad \texttt{smap-less m1 m2} = (\forall k. m1 k \subseteq m2 k)

lemma \texttt{sdom-empty} [\texttt{simp}]: \texttt{sdom} \{\cdot\} = \{\cdot\}
\text{unfolding} \quad \texttt{sdom-def smap-empty-def by auto}

lemma \texttt{sdom-singleton} [\texttt{simp}]: \texttt{sdom} \{k := vs\} \subseteq \{k\}
\text{by} \quad (\texttt{auto simp add: sdom-def smap-singleton-def smap-empty-def})

lemma \texttt{sran-singleton} [\texttt{simp}]: \texttt{sran} \{k := vs\} = vs
\text{by} \quad (\texttt{auto simp add: sran-def smap-singleton-def smap-empty-def})

lemma \texttt{sran-empty} [\texttt{simp}]: \texttt{sran} \{\cdot\} = \{\cdot\}
\text{unfolding} \quad \texttt{sran-def smap-empty-def by auto}

lemma \texttt{sdom-union} [\texttt{simp}]: \texttt{sdom} (m \cup n) = \texttt{sdom} m \cup \texttt{sdom} n
\text{by} \quad (\texttt{auto simp add: smap-union-def sdom-def})

lemma \texttt{sran-union} [\texttt{simp}]: \texttt{sran} (m \cup n) = \texttt{sran} m \cup \texttt{sran} n
\text{by} \quad (\texttt{auto simp add: smap-union-def sran-def})

lemma \texttt{smap-empty} [\texttt{simp}]: \{\cdot\} \subseteq \{\cdot\}
\text{unfolding} \quad \texttt{smap-less-def by auto}

lemma \texttt{smap-less-refl} : \texttt{m} \subseteq \texttt{m}
\text{unfolding} \quad \texttt{smap-less-def by simp}

lemma \texttt{smap-less-trans} [\texttt{trans}]: \texttt{[m1 \subseteq m2; m2 \subseteq m3] \Rightarrow m1 \subseteq m3}
\text{unfolding} \quad \texttt{smap-less-def by auto}

lemma \texttt{smap-union-mono} [\texttt{simp}]: \texttt{[ve1 \subseteq ve1'; ve2 \subseteq ve2']} \Rightarrow \texttt{ve1 \cup ve2 \subseteq ve1' \cup ve2'}
\text{by} \quad (\texttt{auto simp add: smap-less-def smap-union-def})

lemma \texttt{smap-Union-union} : \texttt{m1 \cup \texttt{\cup ms} = \cup \cdot (m1#ms)}
\text{by} \quad (\texttt{rule ext, auto simp add: smap-union-def smap-Union-def})

lemma \texttt{smap-Union-mono}:
\text{assumes} \quad \texttt{list-all2 smap-less ms1 ms2}
\text{shows} \quad \texttt{\cup \cdot ms1 \subseteq \cup \cdot ms2}
using assms
  by (induct rule: list-induct2 [OF list-all2-lengthD [OF assms]])
    (auto intro: smap-union-mono)

lemma smap-singleton-mono: \( v \subseteq v' \implies \{ k := v \}. \subseteq \{ k := v' \}. \)
  by (auto simp add: smap-singleton-def smap-less-def)

lemma smap-union-comm: \( \textit{m1} \cup \textit{m2} = \textit{m2} \cup \textit{m1} \)
  by (rule ext, auto simp add: smap-union-def)

lemma smap-union-empty1 [simp]: \( \emptyset \cup \textit{m} = \textit{m} \)
  by (rule ext, auto simp add: smap-union-def smap-empty-def)

lemma smap-union-assoc [simp]: \((\textit{m1} \cup \textit{m2}) \cup \textit{m3} = \textit{m1} \cup (\textit{m2} \cup \textit{m3})\)
  by (rule ext, auto simp add: smap-union-def)

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 SetMap. Both these sets are defined (maps-over, smaps-over) and the finiteness is shown.

definition maps-over :: \('a::type set \Rightarrow 'b::type set \Rightarrow ('a \Rightarrow 'b) set\)
  where maps-over A B = \{ \textit{m}. \textit{dom} \textit{m} \subseteq A \land \textit{ran} \textit{m} \subseteq B \}\)

lemma maps-over-empty [simp]:
  \( \emptyset \in \text{maps-over} \ A \ B \)
unfolding maps-over-def by simp

60
lemma maps-over-upd:  
assumes  \( m \in \text{maps-over } A \ B \)  
and  \( v \in A \ \text{and} \ k \in B \)  
shows  \( m(v \mapsto k) \in \text{maps-over } A \ B \)  
using assms unfolding maps-over-def  
by (auto dest: subsetD[OF ran-upd])

lemma maps-over-finite[intro]:  
assumes  \( \text{finite } A \ \text{and} \ \text{finite } B \)  
shows  \( \text{finite } (\text{maps-over } A \ B) \)

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: \(\bigwedge a b. (\text{Some } b = x a) \iff (\text{Some } b = y a)\)
      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
  qed

  have \((\lambda f. \{(x, y). \text{Some } y = f x\}) \cdot \text{maps-over } A \ B \subseteq \text{Pow}( A \times B)\)  
  unfolding maps-over-def  

moreover
  have finite (\text{Pow}( A \times B)) using assms by auto
ultimately
  have finite \(?\text{graph}\) by (rule finite-subset)
thus \(?\text{thesis}\)
  by (rule finite-imageD[OF - subset-inj-on[OF inj-map-graph subset-UNIV]])

qed

definition smaps-over :: \('a::type \Rightarrow \ 'b::type set \Rightarrow \ ('a \Rightarrow \ 'b set) set\)  
where smaps-over A B = {m. sdom m \subseteq A \ \text{and} \ 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 \( k \in A \ \text{and} \ vs \subseteq B \)  
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 \( m1 \in \text{smaps-over } A \ B \ \text{and} \ m2 \in \text{smaps-over } A \ B\)
shows \( m_1 \cup m_2 \in \text{smaps-over} \ A \ B \)
using \( \text{assms unfolding} \ \text{smaps-over-def} \)
by (auto simp add:smap-union-def)

lemma \( \text{smaps-over-Union} \):
  assumes \( \text{set } ms \subseteq \text{smaps-over} \ A \ B \)
  shows \( \bigcup ms \in \text{smaps-over} \ A \ B \)
using \( \text{assms} \)
by (induct ms)(auto intro: smaps-over-un)

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

lemma \( \text{smaps-over-finite[intro]} \):
  assumes \( \text{finite } A \) and \( \text{finite } B \)
  shows \( \text{finite } (\text{smaps-over} \ A \ B) \)
proof –
  have inj-smap-graph: inj \((\lambda f. \{(x, y). y = f x \land y \neq \{}\}) \) (is inj ?gr)
  proof (induct rule: inj-onI)
    case 1 x y
    from 1.hyps(3) have hyp: \( a b. (b = x a \land b \neq \{}\) = \( (b = y a \land b \neq \} \)
    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 \text{Pow}( \ A \times \text{Pow} \ B) \)
  unfolding \( \text{smaps-over-def} \)
moreover
  have finite (\text{Pow}( \ A \times \text{Pow} \ B)) using \( \text{assms} \) by auto
ultimately
  have finite ?graph by (rule finite-subset)
  thus ?thesis
  proof
    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.

```plaintext
instantiation bool :: po
begin
definition x ⊑ y ←→ (x → y)
instantce by (default, unfold below-def, fast+)
end

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

instance bool :: pcpo
proof
  have ∀ y. False ⊑ y by (simp add: below-def)
  thus ∃ x::bool. ∀ y. x ⊑ y ..
qed

lemma is-lub-bool: S <| (True ∈ S)
  unfolding is-lub-def is-ub-def below-def by auto

lemma lub-bool: lub S = (True ∈ S)
  using is-lub-bool by (rule lub-eqf)

lemma bottom-eq-False[simp]: ⊥ = False
  by (rule below-antisym [OF minimal], simp add: below-def)
```

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

```plaintext
instantiation set :: (type) po
begin
definition A ⊑ B ←→ A ⊆ B
instance by (default, unfold below-def, fast+)
end

lemma sqsubset-is-subset: A ⊑ B ←→ A ⊆ B
  by (fact below-def)

lemma is-lub-set: S <| ∪ S
```

63
unfolding is-lub-def is-ub-def below-set-def by fast

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

instance set :: (type) cpo
  by (default, fast intro: is-lub-set)

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

instance set :: (type) pcpo
  by (default, fast intro: emptyset-is-bot)

lemma bot-bool-is-emptyset (simp): \bot = {}
  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 \lor g x)
  unfolding imp-conv-disj by (rule cont2cont-disj [OF f g])

lemma cont2cont-Collect (simp, cont2cont):
  assumes \forall y. 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)
  done
apply (auto simp add: is-lab-def is-ub-def below-bool-def lub-is-union)
done

lemma cont2cont-union [simp, cont2cont]:
  \(\forall x. f\ x \Rightarrow \forall x. g\ x\)
\(\Rightarrow \forall x. f\ x \cup g\ x\)
unfolding Un-ndef by simp

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

lemmas adm-subset = adm-below[where \(?b = \text{a::type set, unfolded sqsubset-is-subset}\\)

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

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

lemma cont2cont-case-list [simp, cont2cont]:
  assumes \( \forall y. \text{cont} (\lambda x. f_1 x) \)
  and \( \forall y z. \text{cont} (\lambda x. f_2 x y z) \)
  shows \( \text{cont} (\lambda x. \text{case-list} (f_1 x) (f_2 x) l) \)
using assms
by (cases l) auto

As with the continuity lemmas, we need admissibility lemmas.

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

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

lemma adm-Not[simp]: \( \text{adm} \text{Not} \)
by (rule adm-chfin)

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

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

lemma adm-not-conj:
[[ \( \text{adm} (\lambda x. \neg P x) ; \text{adm} (\lambda x. \neg Q x) \) ]] \( \Rightarrow \) \( \text{adm} (\lambda x. \neg (P x \land Q x)) \)
by simp

lemma adm-single-valued:
  assumes \( \text{cont} (\lambda x. f x) \)
  shows \( \text{adm} (\lambda x. \text{single-valued} (f x)) \)
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 ≡ iterate i f

end

14. Fixed point transformations

theory FixTransform
imports HOLCF
begin

default-sort type

In his treatment of the computabily, 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 \( \forall x. g(f \cdot x) = x \)
  shows \( \text{fix} F = g(\text{fix}(f \circ F \circ g)) \)
  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 \to \text{'c}) \times ('b discr \to \text{'c}) \to ('a + 'b) discr \to \text{'c::cpo}
  where tup-to-sum = (\Lambda p s. (\lambda(f,g). case undiscr s of Inl x \Rightarrow f(\text{Discr} x))

67
Inr x \Rightarrow g(Discr x) \; p)

**definition** sum-to-tup :: (\((\texttt{'}a + \texttt{'}b) \text{ discr} \rightarrow \texttt{'}c) \rightarrow (\texttt{'}a \text{ discr} \rightarrow \texttt{'}c) \times (\texttt{'}b \text{ discr} \rightarrow \texttt{'}c::\text{cpo})\)

**where** sum-to-tup = (\(\Lambda \; f. \; (\Lambda \; x. \; f-(\text{Discr} \; (\text{Inl} \; (\text{undiscr} \; x))))\),

\(\Lambda \; x. \; f-(\text{Discr} \; (\text{Inr} \; (\text{undiscr} \; x))))\))

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 (\(\lambda x. \; \text{case-sum} \; (f \; x) \; (g \; x) \; s\))
- **using** assms
- **by** (cases s, auto intro: cont2cont-fun)

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

- cont (\(\lambda f. \; (\lambda x. \; f-(\text{Discr} \; (\text{Inl} \; (\text{undiscr} \; x))))\)
- apply (rule cont2cont-lambda)
- apply (subst comp-def)
- apply (rule cont2cont-fun [of \(\lambda x. \; \text{x}\), OF cont-id])
- done

**lemma** cont2cont-split-pair [cont2cont, simp]:

- **assumes** f1: cont f
- **and** f2: \(\lambda x. \; \text{cont} \; (f \; x)\)
- **and** g1: cont g
- **and** g2: \(\lambda x. \; \text{cont} \; (g \; x)\)
- **shows** cont (\(\lambda(a, \; b). \; (f \; a \; b, \; g \; a \; b)\))
- **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** = (\(\Lambda \; x. \; f-(\text{Discr} \; (\text{Inl} \; (\text{undiscr} \; x))))\), \(\Lambda \; x. \; f-(\text{Discr} \; (\text{Inr} \; (\text{undiscr} \; x))))\)
- unfolding sum-to-tup-def by simp

**lemma** tup-to-sum-app:

- **tup-to-sum-p** = (\(\Lambda \; s. \; (\lambda(f, g). \)
  - case undiscr s of Inl x \Rightarrow f-(\text{Discr} x)
  - Inr x \Rightarrow g-(\text{Discr} x) \; p)\)
- unfolding tup-to-sum-def by simp

68
Generally, lambda abstractions with discrete domain are continuous and can be resolved immediately.

**Lemma** `discr-app[simp]:`

\[(\Lambda s. f s) \cdot (\text{Discr } x) = f (\text{Discr } x)\]

*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* \(\text{sum-to-tup} \cdot (\text{tup-to-sum} \circ F) = F\)

*unfolding* `sum-to-tup-app` and `tup-to-sum-app`

*by* `(cases F, auto intro:cfun-eqI)`

**Lemma** `fix-transform-pair-sum`:

*shows* 
\[\text{fix} \cdot F = \text{sum-to-tup} \cdot (\text{fix} \cdot (\text{tup-to-sum} \circ F \circ \text{sum-to-tup}))\]

*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* \(f_1: \text{cont } F\)

and \(f_2: \forall x. \text{cont } (F x)\)

and \(g_1: \text{cont } G\)

and \(g_2: \forall x. \text{cont } (G x)\)

*shows* \(\text{tup-to-sum} \circ (\Lambda p. (\lambda (a, b). (F a \ b, G a \ b)) \ p) \circ \text{sum-to-tup} = (\Lambda f s. (\text{case } \text{undiscr } s \ of\ \text{Inl } x \Rightarrow F (\Lambda s. f \cdot (\text{Discr } (\text{Inl } (\text{undiscr } s))))\)

\(\Lambda s. f \cdot (\text{Discr } (\text{Inr } (\text{undiscr } s))))\)·

\(\text{Discr } x)\)

| \(\text{Inr } x \Rightarrow G (\Lambda s. f \cdot (\text{Discr } (\text{Inl } (\text{undiscr } s))))\)

\(\Lambda s. f \cdot (\text{Discr } (\text{Inr } (\text{undiscr } s))))\)·

\(\text{Discr } x))\)

*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]:`

\(\text{fst} (\text{sum-to-tup} \cdot x) = (\Lambda za. x \cdot (\text{Discr } (\text{Inl } (\text{undiscr } za))))\)

*by* `(simp add: sum-to-tup-app)`

end
References

