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
Part I.
The definitions

1. Syntax

theory CPSScheme
  imports Main
begin

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

We assume that the program to be investigated is already parsed into a syntax tree.
Furthermore, we assume that distinct labels were added to distinguish different code
categories 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.

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

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

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

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

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

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

**datatype-compat** lambda call val

**type-synonym** prog = lambda

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

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

**abbreviation** ex1 == (Lambda 1 [(1,"cont")]) (App 2 (R 3 (1,"cont")) [(C 4 0)])
**abbreviation** ex2 == (Lambda 1 [(1,"cont")]) (App 2 (P (Plus 3)) [(C 4 1), (C 5 1), (R 6 (1,"cont"))])
**abbreviation** ex3 == (Lambda 1 [(1,"cont")]) (Let 2 [(2,"rec"),(Lambda 3 [(3,"p"), (3,"i"), (3,"c")]) (App 3 (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 clarifications from Section 4.1. As explained previously, our set of values encompasses just the integers, there is no separate value for false. Also, values and procedures are not distinguished by the type system.

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

type-synonym contour = nat
type-synonym benv = label → 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

type-synonym venv = var × contour → d
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.

\[
\text{fun evalV :: val } \Rightarrow \text{ benv } \Rightarrow \text{ venv } \Rightarrow d (A)
\]

\[
\begin{align*}
\text{where } A (C - i) \beta ve &= \text{ DI } i \\
A (P \text{ prim}) \beta ve &= \text{ DP prim } \\
A (R - \text{ var}) \beta ve &= \\
&\quad \text{(case } \beta \text{ (binder var) of} \\
&\quad \text{Some } l \Rightarrow \text{(case } ve \text{ (var}, l) \text{ of Some } d \Rightarrow d) \\
A (L \text{ lam}) \beta ve &= \text{ DC } (\text{ lam}, \beta)
\end{align*}
\]

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

**type-synonym ans = int lift**

To be able to do case analysis on the custom datatypes \( \text{ lambda, d, call and prim } \) 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.

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

- assumes \( \forall a b c. \text{ cont } (\lambda x. f x a b c) \)
- shows \( \text{ cont } (\lambda x. \text{ case-lambda } (f x) l) \)
  by \( (\text{ cases } l) \) auto

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

- assumes \( \forall y. \text{ cont } (\lambda x. f1 x y) \)
  and \( \forall y. \text{ cont } (\lambda x. f2 x y) \)
  and \( \forall y. \text{ cont } (\lambda x. f3 x y) \)
  and \( \text{ cont } (\lambda x. f4 x) \)
- shows \( \text{ cont } (\lambda x. \text{ case-d } (f1 x) (f2 x) (f3 x) (f4 x) d) \)
  using \( \text{ assms} \)
  by \( (\text{ cases } d) \) auto

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

- assumes \( \forall a b c. \text{ cont } (\lambda x. f1 x a b c) \)
  and \( \forall a b c. \text{ cont } (\lambda x. f2 x a b c) \)
- shows \( \text{ cont } (\lambda x. \text{ case-call } (f1 x) (f2 x) c) \)
  using \( \text{ assms} \)
  by \( (\text{ cases } c) \) auto

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

- assumes \( \forall y. \text{ cont } (\lambda x. f1 x y) \)
  and \( \forall y z. \text{ cont } (\lambda x. f2 x y z) \)
shows cont (λx. case-prim (f1 x) (f2 x) p)
using assms
by (cases p) auto

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

Note how the contour counter is incremented before each call to \( \mathcal{F} \) or when a \textit{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 \textit{fixrec} command from the HOPLCF package is used. This takes a set of equations and builds a functional from that. It mechanically proofs that this functional is continuous and thus a least fixed point exists. This is then used to define \( \mathcal{F} \) and \( \mathcal{C} \) and proof the equations given here. To use the HOPLCF setup, the continuous function arrow \( \to \) with application operator \( \cdot \) is used and our types are wrapped in \textit{discr} and \textit{lift} to indicate which partial order is to be used.

type-synonym fstate = (d × d list × venv × contour)
type-synonym cstate = (call × benv × venv × contour)

fixrec evalF :: fstate discr → ans (F)
and evalC :: cstate discr → ans (C)
where evalF :: fstate = (case undiscr fstate of
  (DC (Lambda lab vs c, β), as, ve, b) ⇒
   (if length vs = length as
     then let β' = β (lab ↦ b);
        ve' = map-upds ve (map (λv, (v, b)) vs) as
        in C·(Discr (c, β', ve', b))
     else ⊥)
  | (DP (Plus c), [DI a1, DI a2], cnt, ve, b) ⇒
     let b' = Suc b;
     β = [c ↦ b]
     in F·(Discr (cnt, [DI (a1 + a2)], ve, b'))
  | (DP (prim.If ct cf), [DI v, contt, contf], ve, b) ⇒
     (if v ≠ 0
      then let b' = Suc b;
      β = [ct ↦ b]
      in F·(Discr (contt, [], ve, b'))
     else let b' = Suc b;
      β = [cf ↦ b]
      in F·(Discr (contf, [], ve, b'))
)
To evaluate a full program, it is passed to $\mathcal{F}$ with proper initializations of the other arguments. We test our semantics function against two example programs and observe that the expected value is returned.

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

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

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

**end**

3. Exact nonstandard semantics

**theory** \( ExCF \)

**imports** HOLCF HOLCFUtils CPSScheme Utils

**begin**

We now alter the standard semantics given in the previous section to calculate a control flow graph instead of the return value. At this point, we still “run” the program in full, so this is not yet the static analysis that we aim for. Instead, this is the reference for
the correctness proof of the static analysis: If an edge is recorded here, we expect it to be found by the static analysis as well.

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

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

definition contour = (UNIV::label list set)

typedef contour = contour
  unfolding contour-def by auto

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

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

instantiation contour :: preorder
begin
  definition le-contour-def: b ≤ 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 \times benv

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

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

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

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

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

instantiation contour :: discrete-cpo
begin
definition [simp]: \( (x :: \text{contour}) \sqsubseteq y \iff x = y \)
instance by standard simp
end

instantiation d :: discrete-cpo begin
definition [simp]: \( (x :: d) \sqsubseteq y \iff x = y \)
instance by standard simp
end

instantiation call :: discrete-cpo begin
definition [simp]: \( (x :: \text{call}) \sqsubseteq y \iff x = y \)
instance by standard simp
end

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

fun evalV :: val => benv => venv => d (A)
where A (C - i) \beta ve = DI i
  | A (P prim) \beta ve = DP prim
  | A (R - var) \beta ve =
    (case \beta (binder var) of
      Some l \Rightarrow (case ve (var,l) of Some d \Rightarrow d | None \Rightarrow DI 0)
      | None \Rightarrow DI 0)
  | A (L lam) \beta ve = DC (lam, \beta)
To be able to do case analysis on the custom datatypes \textit{lambda}, \textit{d}, \textit{call} and \textit{prim} inside
a function defined with \texttt{fixrec}, we need continuity results for them. These are all of the
same shape and proven by case analysis on the discriminator.

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

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

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

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

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

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

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

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

\texttt{fixrec} \texttt{evalF :: fstate discr \rightarrow ans(\mathcal{F})}
\textbf{and} evalC := \text{estate discr} \rightarrow \text{ans (C)}

\textbf{where}\ F\text{-fstate} = (\text{case undiscr fstate of}
\quad (\text{DC (Lambda lab vs c, } \beta, \text{ as, ve, b)} \Rightarrow
\quad \text{(if length vs = length as}
\quad \text{then let } \beta' = \beta (\text{lab }\mapsto\text{b});
\quad \text{ve'} = \text{map-upds ve (map (} \lambda v. (v, b) \text{) vs) as}
\quad \text{in } \mathcal{C} (\text{Discr } (c, \beta', \text{ve', b})))
\quad \text{else } \bot)
\quad | (\text{DP (Plus c)}),[\text{DI a1, DI a2, cnt}], \text{ve, b)} \Rightarrow
\quad \text{(if } \text{isProc cnt}
\quad \text{then let } b' = \text{nb b c};
\quad \beta = [c \mapsto \text{b}]
\quad \text{in } F (\text{Discr (cnt},[\text{DI (a1 + a2)}], \text{ve, b'}))
\quad \cup \{((c, \beta),\text{cnt})\}
\quad \text{else let } b' = \text{nb b cf};
\quad \beta = [\text{cf }\mapsto\text{b}]
\quad \text{in } F (\text{Discr (cnt},[\text{DI }\text{}], \text{ve, b'}))
\quad \cup \{((\text{cf}, \beta),\text{cnt})\}
\quad \text{else } \bot)
\quad | (\text{DP (prim.If ct cf)}),[\text{DI v, contt, contf}], \text{ve, b)} \Rightarrow
\quad \text{(if } \text{isProc contt }\land \text{isProc contf}
\quad \text{then}
\quad \text{(if } v \neq 0
\quad \text{then let } b' = \text{nb b ct};
\quad \beta = [\text{ct }\mapsto\text{b}]
\quad \text{in } (F\text{-Discr (contt},[\text{}], \text{ve, b'}))
\quad \cup \{((\text{ct}, \beta),\text{contt})\})
\quad \text{else let } b' = \text{nb b cf};
\quad \beta = [\text{cf }\mapsto\text{b}]
\quad \text{in } (F\text{-Discr (contf},[\text{}], \text{ve, b'}))
\quad \cup \{((\text{cf}, \beta),\text{contf})\})
\quad \text{else } \bot)
\quad | (\text{Stop},[\text{DI i}],\text{-,-}) \Rightarrow \{\}
\quad | \text{- }\Rightarrow \bot)
\quad )
\quad | \mathcal{C}\text{-estate} = (\text{case undiscr estate of}
\quad (\text{App lab f vs,} \beta, \text{ve, b)} \Rightarrow
\quad \text{let } f' = A f \beta \text{ ve;}
\quad \text{as }= \text{map } (\lambda v. A v \beta \text{ ve}) \text{ vs;}
\quad b' = \text{nb b lab}
\quad \text{in if isProc f'}
\quad \text{then } F (\text{Discr } (f', \text{as}, \text{ve, b'})) \cup \{((\text{lab, } \beta),f')\}
\quad \text{else } \bot)
\quad | (\text{Let lab ls c',} \beta, \text{ve, b)} \Rightarrow
\quad \text{let } b' = \text{nb b lab;}
\quad \beta' = \beta (\text{lab }\mapsto\text{b'});
\quad \text{ve'} = \text{ve }++\text{ map-of (map } (\lambda v.l). (v, b')) \text{ ls})
\quad \text{in } \mathcal{C} (\text{Discr } (c', \beta', \text{ve', b'}))
\quad )

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

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

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

lemma ds-cases-plus = list.exhaust
  OF - d.exhaust, of - - λ a - . a,
  OF - list.exhaust, of - - λ- x . x,
  OF - - d.exhaust, of - - λ- - - a - . a,
  OF - - list.exhaust, of - - λ- - - x . x,
  OF - - - list.exhaust, of - - λ- - - - . x . x
]

lemma ds-cases-if = list.exhaust[OF - d.exhaust, of - - λ a - . a,
  OF - list.exhaust[OF - list.exhaust[OF - list.exhaust, of - - λ- x . x], of - - λ- x . x], of - - λ- x . x
- . x]

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

lemma fstate-case = prod-cases4[OF d.exhaust, of - λx - - . x,
  OF - cl-cases prim.exhaust, of - - λ- - - a . a λ - - - - a . a,
  OF - case-split ds-cases-plus ds-cases-if ds-cases-stop,
  of - - λ- as - - - - vs -. length vs = length as λ - ds - - - - . ds λ - ds - - - - . ds λ - ds - - . ds,
  case-names x Closure x x x x Plus x x x x x x x x x x x x x x x x x If-True If-False x x x x x Stop x x x x x x
]

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

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

lemma correct-ex1: PR ex1 = {{(2,[[1 → b0]], Stop)}}
unfolding evalCPS-def
by simp

lemma correct-ex2: PR ex2 = {{(2, [[1 → b0]], DP (Plus 3)),
                           ((3, [3 → nb b0 2]], Stop)}}
unfolding evalCPS-def
by simp

end

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.

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

instantiation unit :: contour
begin
  definition b - = ()
  definition b0 = ()
  instance by standard auto
end
```

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

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

```isar
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 'venv ⇒ 'c 'd (¬ 'A)
where 'A (C - i) β ve = {}  
  | 'A (P prim) β ve = {PP prim}  
  | 'A (R - var) β ve =  
  case β (binder var) of
  Some l ⇒ ve (var,l)  
  | None ⇒ {})  
  | 'A (L lam) β ve = {PC (lam, β)}

The types of the calculated graph, the arguments to 'F and '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 ⋀ 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 ⋀ y. cont (λx. f1 x y)
and ⋀ 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]:

14
assumes $\bigwedge a b c. \text{cont} (\lambda x. f1 x a b c)$  
and $\bigwedge a b c. \text{cont} (\lambda x. f2 x a b c)$  
shows $\text{cont} (\lambda x. \text{case-call} (f1 x) (f2 x) c)$  
using \text{assms}  
by (cases c) auto

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

We can now define the abstract nonstandard semantics, based on the equations in Figure 4.5 and 4.6 of Shivers’ dissertation. In the $A\text{Stop}$ 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.

fixrec $a\text{-evalF} :: \text{'c::contour} \to \text{fstate discr} \to \text{'c ans (F)}$  
and $a\text{-evalC} :: \text{'c::contour} \to \text{cstate discr} \to \text{'c ans (C)}$

where $F\text{-fstate} = (\text{case undiscr fstate of}$

  $(PC \text{ (Lambda lab vs c, \beta}, as, ve, b) \Rightarrow$
  (if length vs = length as
    then let $\beta' = \beta (\text{lab} \mapsto b)$;
    $ve' = ve \cup (\bigcup (map (\lambda (v,a). \{(v,b) := a\}) (\text{zip vs as})))$
    in $C\text{-Discr (c,$\beta'$,ve',b))$
  else $\bot$)

  $(PP \text{ (Plus c),[-,-,cnts],ve,b) \Rightarrow$
  let $b' = \text{nb b c}$;
  $\beta = [c \mapsto b]$
  in $(\bigcup cnt\in\text{cnts}. F\text{-Discr (cnt,[]},ve,b'))$

  $\bigcup$

  $\{(c, \beta, \text{cont} | \text{cont . cont} \in \text{cnts}\}$

  $(PP \text{ (prim.If ct cf,)[-,-,cntts, cntfs],ve,b) \Rightarrow$
  $(\text{let } b' = \text{nb b ct};$
  $\beta = [ct \mapsto b]$
  in $(\bigcup cnt\in\text{cntts . F-Discr (cnt,[]},ve,b'))$

  $\bigcup$

  $\{(ct, \beta, \text{cnt} | \text{cnt . cnt} \in \text{cntts}\}$

  $\bigcup$

  $\{(ct, \beta, \text{cnt} | \text{cnt . cnt} \in \text{cntfs}\}$

  $)$

  $(A\text{Stop,[-,-,-}) \Rightarrow \{\}$

  $-$ \Rightarrow $\bot$

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

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

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

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

\textbf{lemmas} a-fstate-case-x = a-evalF-cases.cases
\[\OF\ case-split, of - \lambda- vs - - as - - . length vs = length as,\]
\[case-names Closure Closure-inv Plus If Stop]\]

\textbf{lemmas} a-cl-cases = prod.exhaust[\OF lambda.exhaust, of - \lambda a - . a]

\textbf{lemmas} a-ds-cases = list.exhaust[\OF - list.exhaust, of - - \lambda- x. x,\]
\[OF - - - list.exhaust ,of - - \lambda- - - x. x ,\]
\[OF - - - list.exhaust,of - - \lambda- - - - x. x\]
\]
\textbf{lemmas} a-ds-cases-stop = list.exhaust[\OF - list.exhaust, of - - \lambda- x. x]\]
\textbf{lemmas} a-fstate-case = prod.cases[\OF proc.exhaust, of - \lambda x - - - x,\]
\[OF a-cl-cases prim.exhaust, of - \lambda - - - a . a - \lambda - - - a, a,\]
\[OF case-split a-ds-cases a-ds-cases a-ds-cases-stop,\]
\[of - \lambda- as - - - - vs - - . length vs = length as - \lambda - ds - - - - . ds \lambda - ds - - - - , ds \lambda - ds - - - - . ds\]

Not surprisingly, the abstract semantics of a whole program is defined using \(\hat{F}\) with suitably initialized arguments. The function \textit{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.

\textbf{definition} evalCPS-a :: prog \Rightarrow ('c::contour) \\\tilde{\\alpha}ns (\\\tilde{\\alpha}R)
\textbf{where} \\\tilde{\\alpha}R l = (\text{let } ve = \{\};
\[ \beta = \text{empty}; \]
\[ f = \hat{A}(L\,l)\,\beta\,ve \]
\[ \text{in } \hat{F} \cdot (\text{Discr (the-elem } f, ,\{\text{AStop}\}, ve, \hat{b}_0)) \]

end

Part II.
The main results

5. The exact call cache is a map

theory ExCFSV
imports ExCF
begin

5.1. Preparations

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

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

fun benv-in-d :: \( d \Rightarrow \) benv set
  where benv-in-d (DC (l,\beta)) = \{\beta\}
  | benv-in-d - = {} | 

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

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

definition contours-in-ve :: venv \Rightarrow contour set
  where contours-in-ve ve = \bigcup \{contours-in-d \, d \mid 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`
- `∀ β ∈ benv-in-ve. Q β`
- `∀ d' ∈ set ds. ∀ β ∈ benv-in-d d'. Q β`
- **Shows** `∀ β ∈ benv-in-ve (ve (map (λ v. (v, b')) vs |-> ds)). Q β`

**Proof**

1. Fix `β`
2. Assume `ass: β ∈ benv-in-ve (ve (map (λ v. (v, b')) vs |-> ds))`
3. Then obtain `d` where `β ∈ benv-in-d d` and `d ∈ ran (ve (map (λ v. (v, b')) vs |-> ds))`
4. Unfolding `benv-in-ve-def` by `auto`
5. Moreover have `ran (ve (map (λ v. (v, b')) vs |-> ds)) ⊆ ran ve ∪ set ds` using `eq-length`
6. Ultimately have `d ∈ ran ve ∨ d ∈ set ds` by `auto`
7. Thus `Q β` using `assms(2,3) (β ∈ benv-in-d d) unfolding benv-in-ve-def by auto`

**Qed**

**Lemma** `benv-in-eval`:

- **Assumes** `∀ β' ∈ benv-in-ve. Q β'`
- `Q β`
- **Shows** `∀ β ∈ benv-in-d (A v β ve). Q β`

**Proof**

1. (Cases `v`)
   - Case `R - var`
     - Thus `?thesis`
   - Proof (Cases `β (fst var)`)
     - Case `None with R` show `?thesis` by `simp` next
     - Case `(Some cnt)` show `?thesis`
     - Proof (Cases `ve (var,cnt)`)
       - Case `None with Some R` show `?thesis` by `simp` next
       - Case `(Some d)`
         - Hence `d ∈ ran ve unfolding ran-def by blast`
         - Thus `?thesis` using `Some (β (fst var) = Some cnt) R assms(1)`
         - Unfolding `benv-in-ve-def` by `auto`
   - Qed
2. Qed next
   - Case `(L l)` thus `?thesis` using `assms(2)` by `simp` next
   - Case `C` thus `?thesis` by `simp` next
   - Case `P` thus `?thesis` by `simp`

**Qed**

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

**Unfolding** `contours-in-ve-def` by `auto`

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

- **Assumes** `eq-length: length vs = length ds`
- `∀ b' ∈ contours-in-ve. Q b'`
- `∀ d' ∈ set ds. ∀ b' ∈ contours-in-d d'. Q b'`

18
proof
have ran \((ve(map \lambda(v. (v, b''))) \rightarrow ds)\) \(\subseteq\) ran ve \(\cup\) set ds using eq-length by(auto intro!:ran-upds)
thus ?thesis using assms(2,3) unfolding contours-in-ve-def by blast
qed

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

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

have ran \((ve \rightarrow\map\map\map (\lambda(v,l). ((v,b''), (A (L l) \beta' ve)) ls)). Q b'
by(auto intro!:ran-concat)
also
have \(\ldots \subseteq\) ran ve \(\cup\) snd ' set \((\map\map\map (\lambda(v,l). ((v,b''), (A (L l) \beta' ve)) ls))\)
by (rule Un-mono[of ran ve ran ve, OF subset-refl ran-map-of])
also
have \(\ldots \subseteq\) ran ve \(\cup\) set \((\map\map\map (\lambda(v,l). (A (L l) \beta' ve)) ls)\)
by (rule Un-mono[of ran ve ran ve, OF subset-refl]);auto
finally
have d \(\in\) ran ve \(\cup\) set \((\map\map\map (\lambda(v,l). (A (L l) \beta' ve)) ls)\) using d by auto
thus Q b' using assms b unfolding contours-in-ve-def by auto
qed

lemma contours-in-eval:
assumes \(\forall b' \in\) contours-in-ve ve. Q b'
and \(\forall b' \in\) ran \(\beta\), Q b'
shows \(\forall b' \in\) contours-in-d \((A f \beta ve). Q b'
unfolding contours-in-ve-def

proof(cases f)
  case \((R - var)\)
  thus ?thesis
  proof (cases \(\beta \ (f \var)\))
    case None with \(R\) show ?thesis by simp next
    case (Some cnt) show ?thesis
    proof (cases ve (var,cnt))
      case None with Some \(R\) show ?thesis by simp next
      case (Some d)
      hence d \(\in\) ran ve unfolding ran-def by blast
      thus ?thesis using Some (β (f var) = Some cnt) R \(\forall b' \in contours-in-ve ve). Q b'
unfolding contours-in-ve-def
      by auto
5.2. The proof

The set returned by $\mathcal{F}$ and $\mathcal{C}$ is actually a partial map from callsite/binding environment pairs to called values. The corresponding predicate in Isabelle is \textit{single-valued}. We would like to show an auxiliary result about the contour counter passed to $\mathcal{F}$ and $\mathcal{C}$ (such that it is an unused counter when passed to $\mathcal{F}$ and others) first. Unfortunately, this is not possible with induction proofs over fixed points: While proving the inductive case, one does not show results for the function in question, but for an information-theoretical approximation. Thus, any previously shown results are not available. We therefore intertwine the two inductions in one large proof.

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

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

\begin{verbatim}
lemma cc-single-valued':
  \[ \forall b' \in contours-in-ve.\ b' < b \\
  \quad ; \forall b' \in contours-in-d.\ b' < b \\
  \quad ; \forall d' \in set ds.\ \forall b' \in contours-in-d.\ b' < b \\
  \quad \Rightarrow \\
  \quad (\single-valued (\mathcal{F}(\text{Discr (d,ds,ve,b)))) \\
  \quad \land (\forall ((lab,\beta),t) \in \mathcal{F}(\text{Discr (d,ds,ve,b)}).\ \exists b'.\ b' \in \text{ran } \beta \land b \leq b') \\
  \quad ) \]
and \[ \forall b' \in \text{ran } \beta'.\ b' \leq b \\
\quad ; \forall b' \in contours-in-ve.\ b' \leq b \\
\quad \Rightarrow \\
\quad (\single-valued (\mathcal{C}(\text{Discr (c,\beta',ve,b)})) \\
\quad \land (\forall ((lab,\beta),t) \in \mathcal{C}(\text{Discr (c,\beta',ve,b)}).\ \exists b'.\ b' \in \text{ran } \beta \land b \leq b') \]
\end{verbatim}

20
proof \( (\text{induct arbitrary}; d \; ds \; ve \; b \; c \; \beta' \; \text{rule: evalF-evalC-induct}) \)
case Admissibility show \( \text{?case} \)
   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 \( \text{?case} \) by auto next 
   case 2 thus \( \text{?case} \) by auto
\}
next
case \((\text{Next evalF evalC})\)

Nicer names for the hypotheses:

\begin{itemize}
\item note \( \text{hyps-F-sv} = \text{Next.hyps(1)}\)[THEN conjunct1]
\item note \( \text{hyps-F-b} = \text{Next.hyps(1)}\)[THEN conjunct2, THEN bspec]
\item note \( \text{hyps-C-sv} = \text{Next.hyps(2)}\)[THEN conjunct1]
\item note \( \text{hyps-C-b} = \text{Next.hyps(2)}\)[THEN conjunct2, THEN bspec]
\end{itemize}

\{ 
   case \((1 \; d \; ds \; ve \; b)\) thus \( \text{?case} \) proof \( \text{cases (d,ds,ve,b) rule:fstate-case, auto simp del:Un-insert-left Un-insert-right}) \)

Case Closure

\begin{itemize}
\item fix \( \text{lab'} \) and \( \text{vs :: var list and c and } \beta' :: \text{benv} \)
\item assume \( \text{prem-d: } \forall \beta' \in \text{ran } (\beta'(\text{lab'} \rightarrow b)) \) by simp
\item have new: \( b \in \text{ran } (\beta'(\text{lab'} \rightarrow b)) \)
\item proof fix \( b' \) assume \( b' \in \text{ran } (\beta'(\text{lab'} \rightarrow b)) \)
\item hence \( b' \in \text{ran } (\beta' \lor b' \leq b) \) by (auto dest: ran-upd\[THEN\subsetsetD])
\item thus \( b' \leq b \) using \( \text{prem-d by auto} \)
\item qed
\end{itemize}

from \( \text{contours-in-ve-apds[OF eq-length 1.prems(1) 1.prems(3)]} \)

\begin{itemize}
\item have \( b\text{-dom-ve: } \forall b' \in \text{contours-in-ve (ve(map (}\lambda v. (v, b)) \text{ vs } \rightarrow ds)), b' \leq b} \)
\item by auto
\end{itemize}

show single-valued \( (\text{evalC}(\text{Discr } (c, \beta'(\text{lab'} \rightarrow b)), \text{ve}(\text{map (}\lambda v. (v, b)) \text{ vs } \rightarrow ds)), b')) \)

by \( \text{(rule hyps-C-sv[OF new b-dom-beta b-dom-ve, of c])} \)

\begin{itemize}
\item fix \( \text{lab } \beta \) and \( t \)
\item assume \( ((\text{lab}, \beta), t) \in \text{evalC}(\text{Discr } (c, \beta'(\text{lab'} \rightarrow b)), \text{ve}(\text{map (}\lambda v. (v, b)) \text{ vs } \rightarrow ds)), b')) \)
\item thus \( \exists b', b' \in \text{ran } \beta \land b \leq b' \)
\item by \( \text{(auto dest: hyps-C-b[OF new b-dom-beta b-dom-ve])} \)
\end{itemize}

next
Case Plus

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

Case If (true branch)

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

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

fix \( cp2 \ cp1 \ i \ \text{cntt cntf} \)
assume \( \forall b' \in \text{contours-in-d} \ \text{cntt.} \ b' < b \)
hence $b$-dom-$d$: $\forall b' \in \text{contours-in-d cntt. } b' < nb b \ cp1$ by auto
have $b$-dom-$ds$: $\forall d' \in \text{set } [], \forall b' \in \text{contours-in-d } d', b' < nb b \ cp1$ by auto
have $b$-dom-$ve$: $\forall b' \in \text{contours-in-ve ve. } b' < nb b \ cp1$ using 1.prems(1) by auto

\begin{align*}
\text{prove} & \quad \text{thus} \quad \text{case} \\
\text{rewrite} & \quad \text{have} \quad \text{b-dom-ve}\quad \forall b' \in \text{contours-in-ve ve. } b' < nb b \ cp1 \\
\text{have} & \quad \text{b-dom-ds} \\
\text{hence} & \quad \text{False} \quad \text{by} \quad \text{auto intro: single-valued-insert}
\end{align*}

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

Case App

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

\begin{align*}
\text{qed} \quad \text{next} \\
\text{case} & \quad (2 \ ve b c \beta') \\
\text{thus} & \quad ?\text{case}
\end{align*}

\begin{align*}
\text{proof} & \quad \text{(cases c, auto simp add: HOL.Let-def simp del: Un-insert-left Un-insert-right evalV.simps)}
\end{align*}
l
lab')

∪ \{(lab', β'), evalV f β' ve )\}

by(auto intro:single-valued-insert)

fix lab β t
assume (lab, β), t ∈ (evalF·(Discr (evalV f β' ve, map (λv. evalV v β' ve) vs, ve, nb b lab')))
thus ∃ b', b' ∈ ran β ∧ b ≤ b'
by (auto dest: hyps-C-sv[OF b-dom-ve b-dom-d b-dom-ds])
next

Case Let

fix lab ls c'
have prem2: ∀ b'∈ ran (β'(lab' → nb b lab')). b' ≤ nb b lab'
proof
  fix b' assume b'∈ran (β'(lab' → nb b lab'))
hence b' ∈ ran β' v b' = nb b lab' by (auto dest:ran-upd[THEN subsetD])
thus b' ≤ nb b lab' using 2.prems(2) by auto
qed
have prem3: ∀ b'∈contours-in-ve. b' ≤ nb b lab' using 2.prems(3)
by auto

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

have b-dom-ve: ∀ b'∈ contours-in-ve (ve ++ map-of (map (λv.l), ((v,nb b lab'), evalV (l l)) (β'(lab' → nb b lab') ve))) ls). b' ≤ nb b lab'
by (rule c-in-ve')
have b-dom-beta: ∀ b'∈ ran (β'(lab' → nb b lab')). b' ≤ nb b lab' by (rule prem2')
have new: nb b lab' ∈ ran (β'(lab' → nb b lab')) by simp
from hyps-C-so[OF new b-dom-beta b-dom-ve, of c']
show single-valued (evalC·(Discr (c', β'(lab' → nb b lab'),
  ve ++ map-of (map (λv.l),((v, nb b lab'), evalV (l l)) (β'(lab' → nb b lab') ve)))ls),
  nb b lab')).

fix lab β t
assume (lab, β), t ∈ evalC·(Discr (c', β'(lab' → nb b lab'),
  ve ++ map-of (map (λv.l),((v, nb b lab'), A (l l)) (β'(lab' → nb b lab') ve)))ls),
  nb b lab'))
thus ∃ b', b' ∈ ran β ∧ b ≤ b'
by (drule hyps-C-b[OF new b-dom-beta b-dom-ve], auto)
qed
}
}

lemma single-valued (PR prog)
unfolding evalCPS-def
by ((subst HOL.let_def)+, rule cc-single-valued ![THEN conjunct1], auto)
end

6. The abstract semantics is correct

theory AbsCFCorrect
  imports AbsCF ExCF
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 \( \text{abs-}<\text{type}> \), 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) = \hat{b}_0

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

It would be unwieldly to always write out \( \text{abs-}<\text{type}> \) \( x \). 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 \( \text{abs-}<\text{type}> \) 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.

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

**adhoc-overloading**

**definition** $abs-cnt :: abs$

**adhoc-overloading**

**definition** $abs-benv :: benv \Rightarrow 'c::contour-a \\langle \text{benv} \rangle$

**adhoc-overloading**

**primrec** $abs-closure :: closure \Rightarrow 'c::contour-a \\langle \text{closure} \rangle$

**adhoc-overloading**

**primrec** $abs-d :: d \Rightarrow 'c::contour-a \\langle d \rangle$

**adhoc-overloading**

**definition** $abs-venv :: venv \Rightarrow 'c::contour-a \\langle \text{venv} \rangle$

**adhoc-overloading**

**definition** $abs-ccache :: ccache \Rightarrow 'c::contour-a \\langle \text{ccache} \rangle$

**adhoc-overloading**

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

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

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

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

6.2. Lemmas about abstraction functions

Some results about the abstractions functions.

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

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

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 \( \text{abs-ccache-def} \text{ by auto} \)

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

lemma \( \text{abs-venv-empty} \text{[simp]}: |\text{empty}| = \{} \).
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 \( \text{venv-approx} = \text{smap-less} \)

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

definition \( \text{ccache-approx} :: 'c \text{ ccache} \Rightarrow 'c \text{ ccache} \Rightarrow \text{bool} \)
where \( \text{ccache-approx} = \text{less-eq} \)
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]:**

defines 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-asn, auto)

**Lemma abs-venv-map-of-rev:** |map-of (rev l)| ⪅ U. (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 \( \text{rule \ prod.exhaust} \)

hence \( \{ \text{map-of \ (rev \ (a\#l))} \} \subseteq \{ \int \text{map-of \ (rev \ l)} \} \\ \text{by \ (auto \ intro: \ abs-venv-union) \}.

also

have \( \ldots \subseteq \{ \int v \mapsto k \} \cup (\bigcup \{ \text{map} (\lambda(v, k). \int v \mapsto k) \}) l \) \text{ by \ (auto \ intro!: \ smap-union-mono \ OF \ smap-less-refl \ Cons[unfolded \ venv-approx-def]] \ simp:venv-approx-def) \}

also

have \( \ldots = \bigcup \{ \text{map} (\lambda(v, k). \int v \mapsto k) \}) l \) \\ using \( (a = (v, k)) \); \\ by \( \text{auto} \) \\
finally
show \(?\text{case}\).

qed

lemma \( \text{abs-venv-map-of} \): \( \{ \text{map-of \ l} \} \subseteq \{ \text{map} (\lambda(v, k). \int v \mapsto k) \}) l \) \\ using \( \text{abs-venv-map-of-of \ rev \ l} \) \text{ by \ simp} 

lemma \( \text{abs-venv-singleton} \): \( \{(v, b) \mapsto d\} = \{(v, b ) := \int d\} \).

by \( \text{rule \ ext, \ auto \ simp \ add: \ abs-venv-def \ smap-singleton-def \ smap-empty-def} \)

lemma \( \text{ccache-approx-empty}[simp] \): \\
fixes \( x \) :: `'c ccache \\
shows \( \{\} \subseteq x \) \\
unfolding \( \{\} \subseteq x \) \text{ by simp} 

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

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

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

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

lemma \( \text{abs-ccache-union} \): \( \{c1 \cup c2\} \subseteq \{c1\} \cup \{c2\} \) \\
unfolding \( \text{ccache-approx-def \ abs-ccache-def \ by \ simp} \)

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

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

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

**lemma lemma7:**

assumes $|\mathbf{v}:\mathbf{venv}| \preceq \mathbf{v}-a$  
shows $|A \beta \mathbf{v}| \preceq \hat{A} f |\beta| \mathbf{v}-a$

**proof (cases f)**

case $(R - v)$

from assms have assm': $\land v, \text{case-option } \{\} \text{ abs-d (ve (v,b)) } \preceq \mathbf{v}-a (v,|b|)$

by (auto simp add:d-approx-def abs-venv-def venv-approx-def smap-less-def elim!:allE)

show $?\text{thesis}$

proof (cases $\beta$ (binder v))

case None thus $?\text{thesis using } R$ by auto

case (Some b)

thus $?\text{thesis using } R \text{ assm' } of v b$

by (auto simp add:abs-benv-def split:option.split)

done

qed

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 $f$  
shows $\lambda x. \text{abs-ccache}(f x)$

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 $\mathbf{fstate-a} :: 'c::\text{contour-a} \mathbf{fstate} \text{ and } \mathbf{cstate-a} :: 'c::\text{contour-a} \mathbf{cstate}$

shows $|\mathbf{fstate}| \preceq |\mathbf{fstate-a} \rightarrow |F\cdot (\text{Discr } \mathbf{fstate})| \preceq \hat{F} \cdot (\text{Discr } \mathbf{fstate-a})$

and $|\mathbf{cstate}| \preceq |\mathbf{cstate-a} \rightarrow |C\cdot (\text{Discr } \mathbf{cstate})| \preceq \hat{C} \cdot (\text{Discr } \mathbf{cstate-a})$

proof (induct arbitrary: $\mathbf{fstate} \mathbf{fstate-a} \mathbf{cstate} \mathbf{cstate-a}$ rule: evalF-evalC-induct)

case Admissibility show $?\text{case}$

unfolding ccache-approx-def

by (intro adm-lemmas adm-subset adm-prod-split adm-not-conj adm-not-mem adm-single-valued cont2cont)

30
next
  case Bottom {
    next
    case 1 show ?case by simp next
    case 2 show ?case by simp next
  }
  next
  case (Next evalF evalC) {
    case 1
    obtain d ds ve b where fstate: fstate = (d,ds,ve,b)
      by (cases fstate, auto)
    moreover
    obtain proc ds-a ve-a b-a where fstate-a: fstate-a = (proc,ds-a,ve-a,b-a)
      by (cases fstate-a, auto)
    ultimately
    have abs-d: the-elem |d| = proc
    and abs-ds: map abs-d ds ≦ ds-a
    and abs-ve: |ve| ≦ ve-a
    and abs-b: |b| = b-a
    using 1 by (auto elim:fstate-approx.cases)
  }
  from abs-ds have dslength: length ds = length ds-a
    by (auto simp add:ds-approx-def dest:!:list-all2-lengthD)
  from fstate fstate-a abs-d abs-ds abs-ve abs-ds dslength
  show ?case
  proof (cases fstate rule:fstate-case, auto simp del:a-evalF.simps a-evalC.simps set-map)

Case Lambda

  fix β and lab and vs:: var list and c
  assume ds-a-length: length vs = length ds-a

  have |β|lab b| = |β| (lab b-a)
    unfolding below-fun-def using abs-b by simp
  moreover
    { have |ve(map (λv. (v, b)) vs |→| ds)|
      ≦ |ve| ∪ |map-of (rev (zip (map (λv. (v, b)) vs) ds))|
    }
    unfolding map-upds-def by (intro abs-venv-union)
    also
    have ... ≦ ve-a ∪ (⊔ (map (λv. k) {k|} (zip (map (λv. (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 ... = ve-a ∪ (⊔ (map (λv. y) {(v,b) |→ y} (zip vs ds)))
    by (auto simp add: zip-map1 o-def split-def)
    also
    have ... ≦ ve-a ∪ (⊔ (map (λv. y) {(v,b-a) := y} (zip vs ds-a)))
    proof
...
also have \( \subseteq (\bigcup \text{cnt} \in \text{cnt-a} . \hat{F} \cdot (\text{Discr} (\text{cnt}, [\{\}], \text{ve-a}, \text{nb b-a lab}))) \)
using \( \text{abs-cnt} \)
by (auto intro: the-elem-is-Proc[OF \( \text{isProc cntt} \)] simp del: a-evalF.simps simp add: ccache-approx-def d-approx-def)

finally

have \( \text{old-elems: } \{(\text{evalF} \cdot (\text{Discr} (\text{cnt}, [\{\}], \text{ve}, \text{nb b lab})))\} \)
\( \subseteq (\bigcup \text{cnt} \in \text{cnt-a} . \hat{F} \cdot (\text{Discr} (\text{cnt}, [\{\}], \text{ve-a}, \text{nb b-a lab}))) \).

have \( \{(\text{evalF} \cdot (\text{Discr} (\text{cnt}, [\{\}], \text{ve}, \text{nb b lab}))) \)
\( \cup \{((\text{lab}, [\text{lab} \mapsto b]), \text{cnt})\} \)
\( \subseteq (\text{evalF} \cdot (\text{Discr} (\text{cnt}, [\{\}], \text{ve}, \text{nb b lab}))) \)
\( \cup \{((\text{lab}, [\text{lab} \mapsto b]), \text{cnt})\} \)
by (rule abs-ccache-union)

also

have \( \subseteq \)
\( (\bigcup \text{cnt} \in \text{cnt-a} . \hat{F} \cdot (\text{Discr} (\text{cnt}, [\{\}], \text{ve-a}, \text{nb b-a lab}))) \)
\( \cup \{((\text{lab}, [\text{lab} \mapsto b]), \text{cnt}) \text{ cont. cont } \in \text{cnt-a} \} \)
by (rule Un-mono-approx[OF \( \text{old-elems new-elem} \)]

finally

show \( \{\text{insert } ((\text{lab}, [\text{lab} \mapsto b]), \text{cnt}) \}
\( \text{evalF} \cdot (\text{Discr} (\text{cnt}, [\{\}], \text{ve}, \text{nb b lab}))) \)
\( \subseteq \hat{F} \cdot (\text{Discr} (\text{PP} (\text{prim Plus lab}), \text{ds-a}, \text{ve-a}, \text{b-a})) \)
using \( \text{ds-a} \) by (subst a-evalF.simps)(auto simp del:a-evalF.simps)
next

Case If (true branch)

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

let \( ?c = ct::\text{label} \) and \( ?cnt = \text{cntt} \) and \( ?\text{cnt-a} = \text{cntt-a} \)

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

have \( \text{prem: } |\{?\text{cnt}, [], \text{ve}, \text{nb b } ?c\}| \subseteq \hat{F} \cdot (\text{the-elem } |\text{cnt}|, [], \text{ve-a, nb b-a } ?c) \)
using \( \text{abs-ve and abs-b} \)
by (auto intro:fstate-approx.intros)
have \[\text{evalF}(\text{Discr}\ (?\text{cnt}, [\ldots, \text{ve}, \hat{n}b \ b ?c]))\]

\[\overset{\leq}{\sim} \hat{F}(\text{Discr}\ (\text{the-elm} \ |？\text{cnt|}, [\ldots, \text{ve-a}, \hat{n}b b-a ?c]))\]

by (rule Next.hyps(1)|OF prem)

also have \[\ldots \overset{\leq}{\sim} (\bigcup \text{cnt} \in \?\text{cnt-a} \cdot \hat{F}(\text{Discr}\ (\text{cnt}, [\ldots, \text{ve-a}, \hat{n}b b-a ?c]))\]

using abs-cntt and abs-cntf


finally have old-elems: \[\text{evalF}(\text{Discr}\ (?\text{cnt}, [\ldots, \text{ve}, \hat{n}b \ b ?c]))\]

\[\overset{\leq}{\sim} (\bigcup \text{cnt} \in \?\text{cnt-a} \cdot \hat{F}(\text{Discr}\ (\text{cnt}, [\ldots, \text{ve-a}, \hat{n}b b-a ?c])))\]

have \[\text{evalF}(\text{Discr}\ (?\text{cnt}, [\ldots, \text{ve}, \hat{n}b \ b ?c]))\]

\[\bigcup \{|\{?\text{cnt, ?c} \mapsto \text{b} \}|, \?\text{cnt}\}\] \[\overset{\leq}{\sim} \text{evalF}(\text{Discr}\ (?\text{cnt}, [\ldots, \text{ve}, \hat{n}b b ?c]))\]

\[\bigcup \{|\{?\text{cnt, ?c} \mapsto \text{b} \}|, \?\text{cnt}\}\]

by (rule abs-ccache-anion)

also have \[\ldots \overset{\leq}{\sim} (\bigcup \text{cnt} \in \?\text{cnt-a} \cdot \hat{F}(\text{Discr}\ (\text{cnt}, [\ldots, \text{ve-a}, \hat{n}b b-a ?c])))\]

\[\bigcup \{|\{?\text{cnt, ?c} \mapsto \text{b} \}|, \text{cont}\} | \text{cont. cont} \in \?\text{cnt-a}\}\]

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

also have \[\ldots \overset{\leq}{\sim} (\bigcup \text{cnt} \in \?\text{cnt-a} \cdot \hat{F}(\text{Discr}\ (\text{cnt}, [\ldots, \text{ve-a}, \hat{n}b b-a \text{ ct}])))\]

\[\bigcup \{|\{?\text{cnt, ?c} \mapsto \text{b} \}|, \text{cont}\} | \text{cont. cont} \in \?\text{cnt-a}\}\]

\[\bigcup \{|\{\text{cf, ?c} \mapsto \text{b} \}|, \text{cont}\} | \text{cont. cont} \in \?\text{cntf-a}\}\]

by (rule Un-upper1-approx|rule Un-upper2-approx)

finally show \[\text{insert}\ (\{|?\text{c, ?c} \mapsto \text{b} \}|, \?\text{cnt})\]

\[\text{evalF}(\text{Discr}\ (?\text{cnt}, [\ldots, \text{ve}, \hat{n}b b ?c]))\] \[\overset{\leq}{\sim} \hat{F}(\text{Discr}\ (\text{PP (prim. If ct cf)}, \text{ds-a, ve-a, b-a}))\]

using ds-a by (subst a-evalF.simps)(auto simp del:a-evalF.simps)

next

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

fix ct cf v cntt cntf
assume isProc cntt
assume isProc cntf
assume abs-ds': [\{\}, |cntt|, |cntf|] \[\overset{\leq}{\sim} \text{ds-a}\]
then obtain v-a cntt-a cntf-a where ds-a: ds-a = \[v-a, \text{cntt-a, cntf-a}\]

and abs-cntt: |cntt| \[\overset{\leq}{\sim} \text{cntt-a}\]

and abs-cntf: |cntf| \[\overset{\leq}{\sim} \text{cntf-a}\]

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)

34
let \( \text{cf}:\text{label} \) and \( \text{cnt} = \text{cntf} \) and \( \text{cnt-a} = \text{cntf-a} \)

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

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

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

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

have \[\text{evalF}(\text{Discr (?cnt, [\text{cont}], \text{ve}, \text{nb b ?c}))\]
\[\subseteq \hat{\text{F}}(\text{Discr (\text{the-elem} | ?cnt|, [\text{cont}], \text{ve-a}, \text{nb b-a ?c}))\]
\[\cup \{((?c, [?c \mapsto b]), (?cnt))\}\]
\[\subseteq \text{evalF}(\text{Discr (?cnt, [\text{cont}], \text{ve}, \text{nb b ?c}))\]
\[\cup \{((?c, [?c \mapsto b-a]), (?cnt))\}\]
by (rule \text{abs-ccache-union})
also
have \[\ldots \subseteq \]
\[\bigcup \text{cnt}\in?=\text{cnt-a}. \hat{\text{F}}(\text{Discr (\text{cnt}, [\text{cont}], \text{ve-a}, \text{nb b-a ?c})))\]
\[\cup \{((?c, [?c \mapsto b-a]), \text{cont}) | \text{cont. cont} \in \text{cnt-a}\}\]
by (rule Un-mono-approx(\text{OF old-elems new-elem}))
also
have \[\ldots \subseteq \]
\[\bigcup \text{cnt}\in?=\text{cntt-a}. \hat{\text{F}}(\text{Discr (\text{cnt}, [\text{cont}], \text{ve-a}, \text{nb b-a ct}))}\]
\[\cup \{((\text{ct}, [\text{ct} \mapsto b-a]), \text{cont}) | \text{cont. cont} \in \text{cntt-a}\}\]
\[\cup \{((\text{cf}, [\text{cf} \mapsto b-a]), \text{cont}) | \text{cont. cont} \in \text{cntf-a}\}\]
by (rule Un-upper1-approx\text{rule Un-upper2-approx})
finally
show \[\text{insert} ((?c, [?c \mapsto b]), \text{cnt})\]
\[\text{evalF}(\text{Discr (?cnt, [\text{cont}], \text{ve}, \text{nb b ?c})))\] \[\subseteq \]
\[\hat{\text{F}}(\text{Discr (\text{Prim.If ct cf}, \text{ds-a}, \text{ve-a}, \text{b-a}))}\]
using \( \text{ds-a} \) by (subst a-evalF.simps)(auto simp del:a-evalF.simps)
qed
next

35
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: \( |\beta| \leq |ve| \)
  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 (\( \lambda v. A \) v \( \beta \) ve)) vs \( \leq \) map (\( \lambda v. A \) 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 \[\text{evalF}(\text{Discr} (?d, \text{map} (\{\lambda v. A \beta \}\text{ve}) vs, ve, nb b lab))] \leq \text{F}(\text{Discr}(\text{the-elem} |?d|, \text{map} (\{\lambda v. A \beta \}\text{ve-a}) vs, ve-a, \text{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\tilde{A} f \beta-a ve-a. \text{F}(\text{Discr}(f', \text{map} (\{\lambda v. A \beta \}\text{ve-a}) vs, ve-a, \text{nb} |b| lab))) \leq \text{F}(\text{Discr}(f', \text{map} (\{\lambda v. A \beta \}\text{ve-a}) vs, ve-a, \text{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:
\[\text{evalF}(\text{Discr} (\{\lambda v. A \beta \}\text{ve}) vs, ve, nb b lab)) \leq (\bigcup f'\in\tilde{A} f \beta-a ve-a. \text{F}(\text{Discr}(f', \text{map} (\{\lambda v. A \beta \}\text{ve-a}) vs, ve-a, \text{nb} |b| lab)))
by auto

have new-elem: \[\text{evalF}(\text{Discr} \{((\text{lab}, \beta), A f \beta ve)) |\}
\leq \text{evalF}(\{((\text{lab}, \beta-a), f'), f' \in \tilde{A} f \beta-a ve-a\}
using abs-\( \beta \) and lemma7[OF abs-ve]
by(auto simp add:ccache-approx-def d-approx-def)

have \[\text{evalF}(\text{Discr} (\{\lambda v. A \beta \}\text{ve}) vs, ve, nb b lab))

36
\[ \cup \{ ((l, \beta), A f \beta v) \} \]
\[ \subseteq \{ \text{evalF}(\text{Discr}(A f \beta v, \text{map(\lambda \nu \ A v \beta ve vs, ve, nb b lab})) \}
\cup \{ ((l, \beta), A f \beta v) \} \]
by (rule abs-ccache-union)
also have 
\[ \subseteq \{ ((l, \beta), A f \beta v) \}
\cup \{ ((l, \beta), \hat{f} f' \in \hat{A} f \beta v-a) \}
\]
by (rule Un-mono-approx[OF old-elems new-elem])
finally
show \[ \text{insert} ((l, \beta), A f \beta v) \]
\[ \subseteq \text{C} \cdot \text{Discr}(\text{App lab f vs, } | \beta |, ve-a, | b |) \]
using abs-\beta
by (subst a-evalC].simp] (auto simp add: HOL.let-def simp del: a-evalF.simps)
next

Case Let

fix lab binds c'

have \[ | \beta (l \mapsto nb b lab) | = \beta -a(l \mapsto \tilde{nb} | b | lab) \]
using abs-\beta and abs-b
by simp
moreover
have \[ | \text{map-of} (\text{map(\lambda v \ l, ((v, nb b lab), DC (l, \beta (lab \mapsto nb b lab))})) \]
\[ \subseteq \{ (\lambda v \ l). \{ (v, nb b lab), DC (l, \beta (lab \mapsto nb b lab)) \} \} \]
\[ \}
\]
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 \[ | ve ++ map-of \]
\[ (\text{map(\lambda v \ l, ((v, nb b lab), DC (l, \beta (lab \mapsto nb b lab))})) \]
binds]\[ \subseteq \]
\[ ve-a \cup \]
\[ \}
\[ \}
\[ \}
\]
by (rule venv-approx-trans[OF abs-venv-union]
ultimately have
\[
\text{evalC}(\text{Discr}(c'), \beta(\text{lab} \mapsto nb \ \text{lab}), \ve \mapsto \map-of(\map{\lambda(v, l). ((v, nb \ \text{lab}), DC (l, \beta(\text{lab} \mapsto nb \ \text{lab})))) \binds, nb \ \text{lab}))]
\]
\[\subseteq \hat{C}(\text{Discr}(c', \beta-a(\text{lab} \mapsto \hat{nb} \ | b) \ \text{lab}), \ve-a \uplus,
\{ (v, \hat{nb} \ | b) \ \text{lab} ) := \{ \text{PC} (l, \beta-a(\text{lab} \mapsto \hat{nb} \ | b) \ \text{lab})) \} \}) \)
\]
\[\text{using abs-cnt-nb and abs-b by \ -(rule \ \text{Next.hyps}(2), auto intro: \text{cstate-approx.intros})} \]
\]
\[\text{thus \evalC}(\text{Discr}(c', \beta(\text{lab} \mapsto nb \ \text{lab}), \ve \mapsto \map-of(\map{\lambda(v, l). ((v, nb \ \text{lab}), A (L l) (\beta(\text{lab} \mapsto nb \ \text{lab})))) \ve})) \]
\[\hat{C}(\text{Discr}(\text{call.Let } \text{lab binds c' \mapsto \hat{nb} \ | \ b}, \ve-a \mapsto \hat{nb} \ | b) \ \text{lab})) \]
\[\text{using abs-\beta by \ (subst \text{a-evalC.simps}(\text{auto simp add: HOL.Let-def simp del:a-evalC.simps}))} \]
\]
\]
And finally, we lift this result to $\hat{\mathcal{P}} \mathcal{R}$ and $\mathcal{P} \mathcal{R}$.

\[
\text{lemma lemma6: } |\mathcal{P} \mathcal{R} | \subseteq \hat{\mathcal{P}} \mathcal{R} \ l
\]
\[
\text{unfolding evalCPS-def evalCPS-a-def by \ (auto intro!:lemma89 \text{fstate-approx.intros simp del:evalF.simps a-evalF.simps simp add: ds-approx-def d-approx-def venv-approx-def})} \]
\]
\]
7. Generic Computability

\[
\text{theory \ Computability imports HOLCF HOLCFUtils begin} \]
\[
\text{Shivers proves the computability of the abstract semantics functions only by generic and slightly simplified example. This theory contains the abstract treatment in Section 4.4.3. Later, we will work out the details apply this to $\hat{\mathcal{P}} \mathcal{R}$.} \]
7.1. Non-branching case

After the following lemma (which could go into \textit{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.

\textbf{lemma insert-greaterThan:}
\begin{verbatim}
insert (n::nat) {n<..} = {n..}
\end{verbatim}
\textbf{by auto}

\textbf{lemma theorem10:}
\begin{verbatim}
fixes g :: 'a::cpo \to 'b::type set and r :: 'a \to 'a
shows 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-equil[OF cfun-eq1 cfun-belou1, case-names fp least!])
\end{verbatim}
\textbf{case} (fp x)
\begin{verbatim}
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\cdot \text{Suc } i\cdot x))
by (simp add: iterate-Suc2 del: iterate-Suc)
also have \ldots = g\cdot (r^0\cdot x) \cup (\bigcup i\in\{0<..\}. g\cdot (r^i\cdot x))
by auto
also have \ldots = (\bigcup i\in insert 0 \{0<..\}. g\cdot (r^i\cdot x))
by simp
also have \ldots = (\bigcup i. g\cdot (r^i\cdot x))
by (simp only: insert-greaterThan atLeast-0 )
finally
show \ldots by auto
\end{verbatim}
\textbf{next}
\begin{verbatim}
\textbf{case} (least f x)
\end{verbatim}
\begin{verbatim}
\textbf{hence} expand: \(\forall x. f\cdot x = (g\cdot x \cup f\cdot (r\cdot x))\) \textbf{by} (auto simp:cfun-eq-iff)
\end{verbatim}
\begin{verbatim}
\{ fix n
have f\cdot x = (\bigcup i\in\{..n\}. g\cdot (r^i\cdot x)) \cup f\cdot (r\cdot \text{Suc } n\cdot x)
proof(induct n)
\textbf{case} 0 \textbf{thus} \ldots \textbf{by} (auto simp add:expand[of x])
\textbf{case} (Suc n)
\begin{verbatim}
then have f\cdot x = (\bigcup i\in\{..n\}. g\cdot (r^i\cdot x)) \cup f\cdot (r\cdot \text{Suc } n\cdot x) \textbf{by} simp
also have \ldots = (\bigcup i\in\{..n\}. g\cdot (r^i\cdot x))
\begin{verbatim}
\cup g\cdot (r\cdot \text{Suc } n\cdot x) \cup f\cdot (r\cdot \text{Suc } (\text{Suc } n)\cdot x)
\end{verbatim}
\textbf{by} (subst expand[of r\cdot \text{Suc } n\cdot x], auto)
also have \ldots = (\bigcup i\in insert \text{Suc } n\cdot \{..n\}. g\cdot (r^i\cdot x)) \cup f\cdot (r\cdot \text{Suc } (\text{Suc } n)\cdot x)
\end{verbatim}
\end{verbatim}

39
by auto
also have \( (\bigcup_{i} g(r^i.x)) \cup f(r^{Suc\ Suc\ n}.x) \)
by (simp add:atMost-Suc)
finally show \( ?case \).
qed

\[
\text{have } (\bigcup_{i} g(r^i.x)) \subseteq f.x \\
\text{proof (rule UN-least)} \setcounter{equation}{0} \\
\text{fix } i \\
\text{show } g(r^i.x) \subseteq f.x \\
\text{using fin[of i] by auto} \\
\text{qed} \\
\text{thus } ?case \\
\text{apply (subst sqsubset-is-subset) by auto} \\
\text{qed}
\]

7.2. Branching case

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

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

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

\[
g Y = \bigcup_{a \in A} g a \quad \text{and} \quad R Y = \bigcup_{a \in R} R a
\]

the equation becomes

\[
f Y = g Y \cup f (R Y)
\]

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

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

definition powerset-lift :: \( (\'a::cpo \to \'b::type set) \Rightarrow \'a set \to \'b set () \)
where \( \_ \equiv (\Lambda S. (\bigcup y \in S. f.y)) \)

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

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

lemma UNION-commute: \( (\bigcup x \in A. \bigcup y \in B. P x y) = (\bigcup y \in B. \bigcup x \in A . P x y) \)
by auto
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** $f :: 'a \to 'b \\text{set}$ and $g :: 'a \to 'b \\text{set}$ and $R :: 'a \to 'a \\text{set}$
- **assumes** $\forall x. f \cdot x = g \cdot x \cup (\bigcup x \in R \cdot x. f \cdot y)$
- **shows** $f \cdot S = g \cdot S \cup \{ f(R \cdot S) \}$

**proof**

- **have** $f \cdot S = (\bigcup x \in S. f \cdot x)$ **unfolding** powerset-lift-def **by** auto
- **also have** $\ldots = (\bigcup x \in S. f \cdot x \cup (\bigcup y \in R \cdot x. f \cdot y))$ **apply** (subst assms) **by** simp
- **also have** $\ldots = g \cdot S \cup f(R \cdot S)$ **by** (auto simp add: powerset-lift-def)
- **finally**
  - **show** ?thesis .

qed

Theorem 10 as it will be used in Theorem 12.

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

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

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

**lemma theorem12':**

- **fixes** $g :: 'a::\text{discrete-cpo} \to 'b::\text{type set}$ and $R :: 'a \to 'a \text{set}$
- **assumes** $F\cdot x = \text{fix}(\Lambda x. g \cdot x \cup F \cdot (R \cdot x))$
- **shows** $\text{fix}(\Lambda x. g \cdot x \cup (\bigcup y \in R \cdot x. f \cdot y)) = (\Lambda x. F \cdot \{ x \})$

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

- **have** $F\cdot x = (\Lambda x. \bigcup y \in R \cdot x. f \cdot y)$ **using** F-fix **by** (simp)(rule theorem10ps)
- **case** $(F \cdot x)$
  - **have** $g \cdot x \cup (\bigcup y \in R \cdot x. F \cdot \{ x \}) = g \cdot \{ x \} \cup F \cdot (R \cdot \{ x \})$
    - **unfolding** powerset-lift-singleton
    - **by** (auto simp add: powerset-distr UNION-commute F-union)
  - **also have** $\ldots = F \cdot \{ x \}$ **by** (subst (2) fix-eq4[OF F-fix], auto)

41
finally show \(?\)case by simp
next
case (\(\text{least } f' x\))
  hence expand: \(f' = (\Lambda S. g S \cup f'(R S))\) by simp
  have \(f' = (\Lambda x. g x \cup (\bigcup y \in R x. f' y))\) by simp
  by (\(\text{subst expand, rule cfun-eqI, auto simp add: powerset-lift-def}\))
  hence \((\Lambda F. \Lambda x. g x \cup F(R x)) \cdot (f') = f'\) by simp
  from fix-least[OF this and F-fix]
  have \(F \subseteq f'\) by simp
  hence \(F \cdot \{x\} \subseteq f' \cdot \{x\}\) by (\(\text{subst (asm)cfun-below-iff, auto simp del: powerset-lift-singleton}\))
  thus \(?\)case by (auto simp add: sqsubset-is-subset)
qed

lemma theorem12:
fixes \(g :: 'a::discrete-cpo \Rightarrow 'b::type set\) and \(R :: 'a \Rightarrow 'a set\)
shows \(\text{fix} (\Lambda f x. g x \cup (\bigcup y \in R x. f y)) \cd x = g(\bigcup i. ((R)^i \cdot \{x\}))\)
by (\(\text{subst theorem12'[OF theorem10ps[THEN sgn]], auto simp add: powerset-distr}\))

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 \(\hat{F}\) and \(\hat{C}\): Each call adds some elements to the result set
and joins this with the results from a number of recursive calls. So we separate these
two actions into separate functions. These take as arguments the direct sum of \(fstate\) and
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 :: ('a::contour fstate + 'c cstate) discr \Rightarrow 'c ans
where abs-\(\_\)\(\_\) = (case undiscr \(\_\) of
  (Inl (PC (Lambda lab vs c, \(\beta\), as, ve, b)) \Rightarrow \{\})
| (Inl (PP (Plus c),[\(\cdot\),cnts],ve,b)) \Rightarrow
  let \(b' = \hat{nb} b c\);
  \(\beta = [c \mapsto b]\)
in \{(\(c, \beta\), cont) | cont . cont \in cnts\}
| (Inl (PP (prim.If ct cf),[,\(\cdot\),cntts,cntfs],ve,b)) \Rightarrow
  ((
    let \(b' = \hat{nb} b ct\);
    \(\beta = [ct \mapsto b]\)

42
\begin{align*}
\text{in } \{(c, \beta), \text{cnt} \mid \text{cnt} \in \text{cnts}\} \cup \\
\quad \left(\begin{array}{l}
\text{let } b' = \overline{nb} b \ c; \\
\quad \beta = [c f \mapsto b] \\
\quad \text{in } \{(c f, \beta), \text{cnt} \mid \text{cnt} \in \text{cntfs}\}
\end{array}\right)
\end{align*}

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

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

\textbf{fixrec abs-R} := \langle\text{c::contour fsate} + \text{c estate}\rangle \ \text{discr} \rightarrow \langle\text{c::contour fsate} + \text{c estate}\rangle \ \text{discr set}

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

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

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

**Definition** initial-r :: $\text{prog} \Rightarrow (\text{c::contour \hat{fstate} + c\, \text{cstate}}) \, \text{discr}$

where

initial-r $\text{prog} = \text{Discr} (\text{Inl} (\text{the-elem} (\hat{A} (L prog) \, \text{empty} \{\}, \{\}, \{\}, \hat{b}_0)))$

**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 $\hat{F}$, or possible arguments to $\hat{C}$ and of possible arguments.

**Definition** proc-poss :: $\text{prog} \Rightarrow (\text{c::contour proc set}$

where

proc-poss $\text{p} = PC' \times (\text{labels p} \times \text{maps-over} \, (\text{labels p}) \, \text{UNIV}) \cup PP' \times \text{prims p} \cup \{\text{AStop}\}$

**Definition** fstate-poss :: $\text{prog} \Rightarrow (\text{c::contour a-fstate set}$

where

fstate-poss $\text{p} = (\text{proc-poss p} \times \text{NList (Pow (proc-poss p))}) \times \text{call-list-lengths p} \times \text{NList (Pow (proc-poss p))}$

**Definition** cstate-poss :: $\text{prog} \Rightarrow (\text{c::contour a-cstate set}$

where

cstate-poss $\text{p} = (\text{calls p} \times \text{maps-over} \, (\text{labels p}) \, \text{UNIV} \times \text{maps-over} \, (\text{vars p} \times \text{UNIV}) \times (\text{proc-poss p}) \times \text{UNIV})$

**Definition** arg-poss :: $\text{prog} \Rightarrow (\text{c::contour a-fstate + c a-cstate}) \, \text{discr set}$

where

arg-poss $\text{p} = \text{Discr} \times (\text{fstate-poss p} <\Rightarrow c\text{state-poss p})$

Using the auxiliary results from $\text{CPSUtils}$, we see that the argument space as defined here is finite.

**Lemma** finite-arg-space: finite (arg-poss $\text{p}$)

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

**Lemma** evalV-possible:

assumes $f : f \in \hat{A} \, d \, \beta \, ve$

44
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 \(?\text{thesis using} \ f \ \text{by} \ \text{auto} \ \text{next}\)

**case** \((2 \ \text{prim} \ \beta' \ \text{ve}')\)

thus \(?\text{thesis using} \ d \ f \ \text{by} \ (\text{auto} \ \text{dest:} \ \text{vals1 simp add:} \ \text{proc-poss-def})\)

next

**case** \((3 \ \text{l} \ \text{var} \ \beta' \ \text{ve}')\)

thus \(?\text{thesis using} \ f \ d \ \text{smaps-over-im}[OF - \ \text{ve}] \ \text{by} \ (\text{auto} \ \text{split:} \ \text{option.split-asm dest:} \ \text{vals2})\)

next

**case** \((4 \ \text{l} \ \beta \ \text{ve})\)

thus \(?\text{thesis using} \ f \ \beta \ \text{by} \ (\text{auto} \ \text{dest!:} \ \text{vals3 simp add:} \ \text{proc-poss-def})\)

qed

**lemma** \(\text{adm-subset}: \ \text{cont} (\lambda x. f x) \implies \ \text{adm} (\lambda x. f x \subseteq S)\)

by \((\text{subst sqsubset-is-subset}[THEN sym], \ \text{intro adm-lemmas cont2cont})\)

**lemma** \(\text{arg-space-complete}:\)

\( \text{state} \in \ \text{arg-poss} p \implies \ \text{abs-R} \cdot \text{state} \subseteq \ \text{arg-poss} p \)

**proof**(\(\text{induct rule:} \ \text{abs-R.induct}[\text{case-names Admissibility Bot Step}]\))

**case Admissibility** \(?\text{case}\)

by \((\text{intro adm-lemmas adm-subset cont2cont})\)

next

**case Bot** \(?\text{case by} \ \text{simp} \ \text{next}\)

**case** \((\text{Step} \ \text{abs-R})\)

**note** \(\text{state} = \ \text{Step}(2)\)

**show** \(?\text{case}\)

**proof** (cases state)

**case** \((\text{Discr} \ \text{state'}\) \ (?\text{thesis}\)

**proof** (cases state')

**case** \((\text{Inl fstate})\) \ (?\text{thesis}\)

**using** \(\text{Inl Discr state}\)

**proof** (cases fstate rule: a-fstate-case, auto)

**Case Lambda**

fix \( l \ \text{vs} \ c \ \beta \ \text{as} \ \text{ve} \ b\)

**assume** \(\text{Discr} (\text{Inl} (\text{PC} (\text{Lambda} l \ \text{vs} \ c, \ \beta), \ \text{as}, \ \text{ve}, \ b)) \in \ \text{arg-poss} p\)

**hence lam: Lambda l \ \text{vs} \ c \ \in \ \text{lamdas} 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 \(\text{as}: \text{as} \in \text{NList} (\text{Pow} (\text{proc-poss} p)) (\text{call-list-lengths} p)\)
\textbf{unfolding} \texttt{arg-poss-def fstate-poss-def proc-poss-def by auto}

\texttt{from lam have} \( c \in \text{calls} \) \[p\]  
\texttt{by (rule lambdas1)}

\textbf{moreover}
\texttt{from lam have} \( l \in \text{labels} \) \[p\]  
\texttt{by (rule lambdas2)}
\texttt{with beta have} \( \beta(l \mapsto b) \in \text{maps-over (labels} \) \[p\] \text{UNIV} \)  
\texttt{by (rule maps-over-upd, auto)}

\textbf{moreover}
\texttt{from lam have} \( \text{vs: set \text{vs} \subseteq \text{vars} \) \[p\] \)  
\texttt{by (rule lambdas3)}
\texttt{from as have} \( \forall \ x \in \text{set as}. \ x \in \text{Pow (proc-poss} \) \[p\] \)  
\texttt{unfolding NList-def nList-def by auto}
\texttt{ultimately have} \( \{\} \in \text{NList (Pow (proc-poss} \) \[p\] \text{)) (call-list-lengths} \[p\] \)  
\texttt{by auto}

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

\textbf{Case If (true case)}
\texttt{fix ve b l1 l2 v cntst cntsf cnt}
assume \( \text{Discr} \ (\text{Inl\ (PP\ (prim\cdot\text{If\ l1\ l2})}, [v,\ \text{cntst},\ \text{cntsf}],\ ve,\ b]) \in \text{arg-poss\ p} \)
and \( \text{cnt} \in \text{cntst} \)

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

unfolding \( \text{arg-poss-def}\ \text{fstate-poss-def}\ \text{NList-def}\ \text{nList-def} \) by auto

moreover
have \( [] \in \text{NList\ (Pow\ (proc-poss\ p))\ (call-list-lengths\ p)} \)

unfolding \( \text{call-list-lengths-def}\ \text{NList-def}\ \text{nList-def} \) by auto
ultimately
have \( (\text{cnt},\ [],\ \text{ve},\ \text{nb\ l1}) \in \text{fstate-poss\ p} \)

unfolding \( \text{fstate-poss-def} \) by auto

thus \( \text{Discr}\ (\text{Inl}\ (\text{cnt},\ []),\ \text{ve},\ \text{nb\ l1})) \in \text{arg-poss\ p} \)

unfolding \( \text{arg-poss-def} \) by auto

next

Case If (false case)

fix \( ve,\ b,\ l_1,\ l_2,\ v,\ \text{cntst},\ \text{cntsf},\ \text{cnt} \)
assume \( \text{Discr} \ (\text{Inl\ (PP\ (prim\cdot\text{If\ l1\ l2})}, [v,\ \text{cntst},\ \text{cntsf}],\ ve,\ b]) \in \text{arg-poss\ p} \)
and \( \text{cnt} \in \text{cntst} \)

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

unfolding \( \text{arg-poss-def}\ \text{fstate-poss-def}\ \text{NList-def}\ \text{nList-def} \) by auto
moreover
have \( [] \in \text{NList\ (Pow\ (proc-poss\ p))\ (call-list-lengths\ p)} \)

unfolding \( \text{call-list-lengths-def}\ \text{NList-def}\ \text{nList-def} \) by auto
ultimately
have \( (\text{cnt},\ [],\ \text{ve},\ \text{nb\ l1}) \in \text{fstate-poss\ p} \)

unfolding \( \text{fstate-poss-def} \) by auto

thus \( \text{Discr}\ (\text{Inl}\ (\text{cnt},\ []),\ \text{ve},\ \text{nb\ l1})) \in \text{arg-poss\ p} \)

unfolding \( \text{arg-poss-def} \) by auto

qed
next

Case \( \text{(Inr\ cstate)} \)

show \(?\text{thesis}\) proof\((\text{cases\ cstate\ rule:\ prod-cases4})\)

 case \( \text{(fields\ c\ \beta\ ve\ b)} \)

show \(?\text{thesis}\) using \( \text{Discr}\ \text{Inr}\ \text{fields}\ \text{state}\) proof\((\text{cases\ c,\ auto\ simp\ add:\HOL.\Let-def}\ \text{smp}\ del:\evalV-a.simps})\)

Case App

fix \( l,\ d,\ ds,\ f \)
assume \( \text{arg:\ Discr}\ (\text{Inr\ (App\ l\ d\ ds,\ \beta,\ ve,\ b)}) \in \text{arg-poss\ p} \)
and \( f: f \in \text{\u00e5\ d\ \beta\ ve} \)

hence \( c: \text{App\ l\ d\ ds} \in \text{calls\ p} \)
and \( d: d \in \text{vals\ p} \)
and \( ds: \text{set\ ds} \subseteq\ \text{vals\ p} \)

47
and beta: β ∈ maps-over (labels p) UNIV
and ve: ve ∈ smaps-over (vars p × UNIV) (proc-poss p)
by (auto simp add: arg-poss-def cstate-poss-def call-list-lengths-def dest: app1 app2)

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

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

next

Case Let

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

have beta': β(l → nb b l) ∈ maps-over (labels p) UNIV (is ?β' ∈ -)
by (auto intro: maps-over-upd[OF beta l])

moreover
have ve ∪. ∪. map (λv, lam). { (v, nb b l) := A (L lam) (β(l → nb b l)) ve }.)
binds
∈ smaps-over (vars p × UNIV) (proc-poss p) (is ?ve' ∈ -)
using vars ls beta'
by (auto intro!: smaps-over-un[OF ve] smaps-over-Union)
(auto intro!:smaps-over-singleton simp add: proc-poss-def)
ultimately
have (c', ?β', ?ve', nb b l) ∈ cstate-poss p (is ?cstate ∈ -)
using c' unfolding cstate-poss-def by simp
This result is now lifted to the powerset of $\text{abs-R}$.

**lemma** arg-space-complete-ps: $\text{states} \subseteq \arg-poss \ p \Longrightarrow (\text{abs-R})\text{-states} \subseteq \arg-poss \ p$

**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 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: $x \in S$

**and** step: $\forall y, y \subseteq S \Longrightarrow (f \cdot y) \subseteq S$

**shows** $(\bigcup i. \text{iterate } i \cdot f \cdot \{x\}) \subseteq S$

**proof**

- **fix** $i$
- **have** iterate $i \cdot f \cdot \{x\} \subseteq S$
- **proof**(induct $i$)
  - **case** 0 show $?case$ using $(x \in S)$ by simp
  - **case** (Suc $i$) thus $?case$ using step[of iterate $i \cdot f \cdot \{x\}$] by simp
- **qed**
- **thus** $?thesis$ by auto
- **qed**

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

**proof**

- **rule** finite-subset[OF -finite-arg-space]
- **have** [simp]: $p \in \text{lambdas } p$ by (cases $p$, simp)
- **show** $?S \subseteq \arg-poss \ p$
- **unfolding** initial-r-def
- **by** (rule UN-iterate-less[OF -arg-space-complete-ps])
  - (auto simp add: arg-poss-def fstate-poss-def proc-poss-def call-list-lengths-def NList-def nList-def
    intro!: imageI)
- **qed**

8.2. A decomposition

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

lemma a-evalF-decomp:
\[ \hat F = \text{fst} (\text{sum-to-tup}(\text{fix}(\Lambda f x. (\bigcup y \in \text{abs-R} x. f \cdot y) \cup \text{abs-g} x))) \]
apply (subst a-evalF-def)
apply (subst fix-transform-pair-sum)
apply (rule arg-cong [of - - \lambda x. \text{fst} (\text{sum-to-tup}(\text{fix}-x))])
apply (simp)
apply (simp only: discr-app undiscr-Discr)
apply (rule cfun-eqI, rule cfun-eqI, simp)
apply (rule case-tac xa, rename-tac a, case-tac a, simp)
apply (rule case-tac aa rule: a-fstate-case, simp-all add: Un-commute-helper)
apply (case-tac b rule: prod-cases4)
apply (simp)
apply (simp-all add: HOL.Let-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 Computability and express the abstract semantics as the image of a finite set under a computable function.

lemma a-evalF-iterative:
\[ \hat F \cdot (\text{Discr} x) = \text{abs-g} \cdot \bigcup i. \text{iterate i} \cdot (\text{abs-R}) \cdot \{\text{Discr} (\text{Inl} x)\} \]
by (simp del: abs-R.simps abs-g.simps add: theorem12 Un-commute a-evalF-decomp)

lemma a-evalCPS-iterative:
\[ \hat PR \ prog = \text{abs-g} \cdot \bigcup i. \text{iterate i} \cdot (\text{abs-R}) \cdot \{\text{initial-r prog}\} \]
unfolding evalCPS-a-def and initial-r-def
by (subst a-evalF-iterative, simp del: abs-R.simps abs-g.simps evalV-a.simps)

done

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.

\[
\begin{align*}
\text{fun} & \quad \text{lambdas} :: \lambda \Rightarrow \text{lambda set} \\
\text{and} & \quad \text{lambdasC} :: \text{call} \Rightarrow \text{lambda set} \\
\text{and} & \quad \text{lambdasV} :: \text{val} \Rightarrow \text{lambda set} \\
\text{where} & \quad \text{lambdas} \ (\Lambda l \ vs \ c) = (\{\Lambda l \ vs \ c\} \cup \text{lambdasC} \ c) \\
& \quad \text{lambdasC} \ (\text{App} \ l \ d \ ds) = \text{lambdasV} \ d \cup (\text{UNION} \ (\text{set} \ ds) \ \text{lambdasV}) \\
& \quad \text{lambdasC} \ (\text{Let} \ l \ binds \ c') = (\text{UNION} \ (\text{set} \ binds) \ (\lambda(-,l). \ \text{lambdas} \ l) \cup \text{lambdasC} \ c') \\
& \quad \text{lambdasV} \ (L \ l) = \text{lambdas} \ l \\
& \quad \text{lambdasV} - = \{\}
\end{align*}
\]

\[
\begin{align*}
\text{fun} & \quad \text{calls} :: \lambda \Rightarrow \text{call set} \\
\text{and} & \quad \text{callsC} :: \text{call} \Rightarrow \text{call set} \\
\text{and} & \quad \text{callsV} :: \text{val} \Rightarrow \text{call set} \\
\text{where} & \quad \text{calls} \ (\Lambda l \ vs \ c) = \text{callsC} \ c \\
& \quad \text{callsC} \ (\text{App} \ l \ d \ ds) = \{\text{App} \ l \ d \ ds\} \cup \text{callsV} \ d \cup (\text{UNION} \ (\text{set} \ ds) \ \text{callsV}) \\
& \quad \text{callsC} \ (\text{Let} \ l \ binds \ c') = \{\text{Let} \ l \ binds \ c'\} \cup (\text{UNION} \ (\text{set} \ binds) \ (\lambda(-,l). \ \text{calls} \ l) \cup \text{callsC} \ c') \\
& \quad \text{callsV} \ (L \ l) = \text{calls} \ l \\
& \quad \text{callsV} - = \{\}
\end{align*}
\]

\[
\begin{align*}
\text{fun} & \quad \text{vars} :: \lambda \Rightarrow \text{var set} \\
\text{and} & \quad \text{varsC} :: \text{call} \Rightarrow \text{var set} \\
\text{and} & \quad \text{varsV} :: \text{val} \Rightarrow \text{var set} \\
\text{where} & \quad \text{vars} \ (\Lambda l \ - \ vs \ c) = \text{set} \ vs \cup \text{varsC} \ c \\
& \quad \text{varsC} \ (\text{App} \ - \ a \ as) = \text{varsV} \ a \cup (\text{UNION} \ (\text{set} \ as) \ \text{varsV}) \\
& \quad \text{varsC} \ (\text{Let} \ - \ binds \ c') = (\text{UNION} \ (\text{set} \ binds) \ (\lambda(v,l). \ \{v\} \cup \text{vars} \ l) \cup \text{varsC} \ c') \\
& \quad \text{varsV} \ (L \ l) = \text{vars} \ l \\
& \quad \text{varsV} \ (R \ - \ v) = \{v\} \\
& \quad \text{varsV} - = \{\}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} & \quad \text{finite-lambdas[simp]}: \text{finite (lambdas} \ l) \ \text{and finite (lambdasC} \ c) \ \text{finite (lambdasV} \ v) \\
\text{by} & \quad (\text{induct rule}: \ \text{lambdas-lambdasC-lambdasV.induct, auto})
\end{align*}
\]

\[
\begin{align*}
\text{lemma} & \quad \text{finite-calls[simp]}: \text{finite (calls} \ l) \ \text{and finite (callsC} \ c) \ \text{finite (callsV} \ v) \\
\text{by} & \quad (\text{induct rule}: \ \text{calls-callsC-callsV.induct, auto})
\end{align*}
\]

\[
\begin{align*}
\text{fun} & \quad \text{vars} :: \lambda \Rightarrow \text{var set} \\
\text{and} & \quad \text{varsC} :: \text{call} \Rightarrow \text{var set} \\
\text{and} & \quad \text{varsV} :: \text{val} \Rightarrow \text{var set} \\
\text{where} & \quad \text{vars} \ (\Lambda l \ - \ vs \ c) = \text{set} \ vs \cup \text{varsC} \ c \\
& \quad \text{varsC} \ (\text{App} \ - \ a \ as) = \text{varsV} \ a \cup (\text{UNION} \ (\text{set} \ as) \ \text{varsV}) \\
& \quad \text{varsC} \ (\text{Let} \ - \ binds \ c') = (\text{UNION} \ (\text{set} \ binds) \ (\lambda(v,l). \ \{v\} \cup \text{vars} \ l) \cup \text{varsC} \ c') \\
& \quad \text{varsV} \ (L \ l) = \text{vars} \ l \\
& \quad \text{varsV} \ (R \ - \ v) = \{v\} \\
& \quad \text{varsV} - = \{\}
\end{align*}
\]

\[
\begin{align*}
\text{lemma} & \quad \text{finite-vars[simp]}: \text{finite (vars} \ l) \ \text{and finite (varsC} \ c) \ \text{finite (varsV} \ v)
\end{align*}
\]

51
by (induct rule: vars-vars-varsV.induct, auto)

fun label :: lambda + call ⇒ label
where label (Inl (Lambda l - -)) = l
  | label (Inr (App l - -)) = l
  | label (Inr (Let l - -)) = l

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

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

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

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

fun vals :: lambda ⇒ val set
and valsC :: call ⇒ val set
and valsV :: val ⇒ val set
where vals (Lambda - vs c) = valsC c
  | valsC (App - a as) = valsV a ∪ UNION (set as) valsV
  | valsC (Let - binds c') = UNION (set binds) (λ(-,l). vals l) ∪ valsC c'
  | valsV (L l) = {L l} ∪ vals l
  | valsV (R l v) = {R l v}
  | valsV (P prim) = {P prim}
  | valsV (C l v) = {C l v}

lemma fixes list2 :: (var × lambda) list and t :: var×lambda
shows lambdas1: Lambda l vs c ∈ lambdas x ⇒ c ∈ calls x
and Lambda l vs c ∈ lambdasC y ⇒ c ∈ callsC y

52
apply

and \( \lambda l vs c \in \text{lambdasV } z \rightarrow c \in \text{callsV } z \)
and \( \forall z \in \text{set list. } \lambda l vs c \in \text{lambdasV } z \rightarrow \lambda l \in \text{labelsV } z \)
and \( \lambda l vs c \in \text{lambdasV } z \rightarrow \lambda l \in \text{labelsV } z \)
and \( \lambda x \in \text{set } (\text{list2 } :: (\text{var } \times \text{lambda}) \text{ list}) . \lambda l vs c \in \text{lambdasV } z \rightarrow \lambda l \in \text{labelsV } z \)

apply \((\text{induct rule:mutual-lambda-call-var-inducts})\)
apply \(\text{auto}\)
apply \((\text{case-tac } c, \text{ auto})[1]\)
apply \((\text{rule-tac } x=\langle a, b \rangle, \text{ ba } \rangle \in \text{bexI, auto})\)
done

lemma

shows lambdas2: \( \lambda l vs c \in \text{lambdas } x \rightarrow \lambda l \in \text{labels } x \)
and \( \lambda l vs c \in \text{lambdasC } y \rightarrow \lambda l \in \text{labelsC } y \)
and \( \lambda l vs c \in \text{lambdasV } z \rightarrow \lambda l \in \text{labelsV } z \)
and \( \lambda x \in \text{set } (\text{list2 } :: (\text{var } \times \text{lambda}) \text{ list}) . \lambda l vs c \in \text{lambdasV } z \rightarrow \lambda l \in \text{labelsV } z \)

apply \((\text{induct rule:mutual-lambda-call-var-inducts})\)
apply \(\text{auto}\)
apply \((\text{rule-tac } x=\langle a, b \rangle, \text{ ba } \rangle \in \text{bexI, auto})\)
done

lemma

shows lambdas3: \( \lambda l vs c \in \text{lambdas } x \rightarrow \text{set } vs \subseteq \text{vars } x \)
and \( \lambda l vs c \in \text{lambdasC } y \rightarrow \text{set } vs \subseteq \text{varsC } y \)
and \( \lambda l vs c \in \text{lambdasV } z \rightarrow \text{set } vs \subseteq \text{varsV } z \)
and \( \lambda x \in \text{set } (\text{list2 } :: (\text{var } \times \text{lambda}) \text{ list}) . \lambda l vs c \in \text{lambdasV } z \rightarrow \text{set } vs \subseteq \text{varsV } z \)

apply \((\text{induct } x \text{ and } y \text{ and } z \text{ and list and list2 and t rule:mutual-lambda-call-var-inducts})\)
apply \(\text{auto}\)
apply \((\text{erule-tac } x=\langle a a, ba, bb \rangle \in \text{ballE})\)
apply \((\text{rule-tac } x=\langle a a, ba, bb \rangle \in \text{bexI, auto})\)
done

lemma

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

apply \((\text{induct } x \text{ and } y \text{ and } z \text{ and list and list2 and t rule:mutual-lambda-call-var-inducts})\)
apply \(\text{auto}\)
apply \((\text{case-tac } d, \text{ auto})\)
apply \((\text{erule-tac } x=\langle a, b \rangle, \text{ ba } \rangle \in \text{ballE})\)
apply \((\text{rule-tac } x=\langle a, b \rangle, \text{ ba } \rangle \in \text{bexI, auto})\)
done

lemma
  shows 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
  shows 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
  shows 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
  shows 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 Let l binds c' ∈ callsV z ⇒ fst ' set binds ⊆ varsV z
and ∀ z ∈ set list. Let l binds c' ∈ callsV z ⇒ fst ' set binds ⊆ varsV z
and ∀ x ∈ set (list2 :: (var × lambda) list). Let l binds c' ∈ calls (snd x) ⇒ fst ' set binds ⊆ vars (snd x)
and Let l binds c' ∈ calls (snd (t:: var×lambda)) ⇒ fst ' set binds ⊆ vars (snd t)
apply (induct x and y and z and list and list2 and t rule: mutual-lambda-call-var-inducts)
apPLY auto
apply (erule-tac x=((ab, bc), bd) in ballE)
apPLY (rule-tac x=((ab, bc), bd) in bexI, auto)
done

lemma
shows let4: Let l binds c' ∈ calls x ⇒ snd ' set binds ⊆ lambdas x
and Let l binds c' ∈ callsC y ⇒ snd ' set binds ⊆ lambdasC y
and Let l binds c' ∈ callsV z ⇒ snd ' set binds ⊆ lambdasV z
and ∀ z ∈ set list. Let l binds c' ∈ callsV z ⇒ snd ' set binds ⊆ lambdasV z
and ∀ x ∈ set (list2 :: (var × lambda) list). Let l binds c' ∈ calls (snd x) ⇒ snd ' set binds ⊆ lambdas (snd x)
and Let l binds c' ∈ calls (snd (t:: var×lambda)) ⇒ snd ' set binds ⊆ lambdas (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=((ab, bc), bd) in ballE)
apPLY (case-tac ba, auto)
apPLY (erule-tac x=((aa, bb), bc) in ballE)
apPLY (rule-tac x=((aa, bb), bc) in bexI, auto)
done

lemma
shows vals1: P prim ∈ vals p ⇒ prim ∈ prims p
and P prim ∈ valsC y ⇒ prim ∈ primsC y
and P prim ∈ valsV z ⇒ prim ∈ primsV z
and ∀ z ∈ set list. P prim ∈ valsV z ⇒ prim ∈ primsV z
and ∀ x ∈ set (list2 :: (var × lambda) list). P prim ∈ vals (snd x) ⇒ prim ∈ prims (snd x)
and P prim ∈ vals (snd (t:: var×lambda)) ⇒ prim ∈ prims (snd t)
apPLY (induct rule: mutual-lambda-call-var-inducts)
apPLY auto
apply (erule-tac x=((ab, bc), bd) in ballE)
apPLY (rule-tac x=((ab, bc), bd) in bexI, auto)
done

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

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

definition nList :: 'a set \Rightarrow nat \Rightarrow 'a list set
where \( nList \ A \ n \equiv \{l. \ \text{set} \ l \leq A \land \text{length} \ l = n\} \)
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 \#) \((A \times \text{nList} \ A \ n)\) (is \(?lhs = ?rhs\) )
proof(rule subset-antisym[OF subsetI subsetI])
fix l assume \(?lhs \ thus \ l \in \?rhs\)
by (cases l, auto simp add:nList-def)
next
fix l assume \(?rhs \ thus \ l \in \?lhs\)
by (auto simp add:nList-def)
qed
thus finite \(?lhs\) using finA and finn
by auto
qed

definition NList :: 'a set \Rightarrow nat set \Rightarrow 'a list set
where \( NList \ A \ N \equiv \bigcup \ n \in \ N. \ nList \ A \ n \)
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 \( \text{call-list-lengths} \ p = \{0, 1, 2, 3\} \cup (\lambda c. \text{case } c \text{ of } (\text{App } a \ ds) \Rightarrow \text{length } ds \mid \_ \Rightarrow 0) \)'

\( \text{calls } p \)

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

end

10. General utility lemmas

theory Utils imports Main begin

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

Lemmas about the single-valued predicate.

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

lemma single-valued-insert:
  assumes single-valued rel
  and \( \land x y . [(x, y) \in \text{rel}; x = a] \Rightarrow y = b \)
  shows single-valued (insert (a, b) rel)
  using assms
  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 ran m \cup \{v\}
  unfolding ran-def by auto

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

lemma ran-concat: ran (m1 ++ m2) \subseteq ran m1 \cup 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 ran m \cup \text{set} vs
  proof
    have ran (map-upds m ks vs) \subseteq ran (m++map-of (rev (zip ks vs))]
      unfolding map-upds-def by simp
    also have \_ \subseteq ran m \cup ran (map-of (rev (zip ks vs))) by (rule ran-concat)
    also have \_ \subseteq ran m \cup \text{snd } \text{set} (rev (zip ks vs))
  end

57
also have \( \subseteq \text{ran } m \cup \text{set vs} \)
by (auto intro:Un-mono[of ran m ran m] subset-refl simp del:set-map simp add:set-map[THEN sym] map-snd-zip[OF eq-length])
finally show \( \text{thesis} \).
qed

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

Lemmas about \( \text{map} \), \( \text{zip} \) and \( \text{fst/snd} \)

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

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

end

11. Set-valued maps

theory SetMap
  imports Main
begin

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

definition
  \( \text{sdom} :: (\tau \Rightarrow \tau \text{ set}) \Rightarrow \tau \text{ set} \) where
  \( \text{sdom } m = \{ a. \ m a \sim = \{\}\} \)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma \textit{smap-less-refl}: m \subseteq m
    unfolding \textit{smap-less-def} by simp

lemma \textit{smap-less-trans}[trans]: \{ m1 \subseteq m2; m2 \subseteq m3 \} \Longrightarrow m1 \subseteq m3
    unfolding \textit{smap-less-def} by auto

lemma \textit{smap-union-mono}: \{ \ve1 \subseteq \ve1'; \ve2 \subseteq \ve2' \} \Longrightarrow \ve1 \cup. \ve2 \subseteq \ve1' \cup. \ve2'
    by \{ auto simp add: \textit{smap-less-def} \textit{smap-union-def} \}

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

lemma \textit{smap-Union-mono}:
    assumes \textit{list-all2} \textit{smap-less} \textit{ms1} \textit{ms2}
    shows \cup. \textit{ms1} \subseteq \cup. \textit{ms2}

59
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: \( m1 \cup m2 = m2 \cup m1 \)
  by (rule ext, auto simp add: smap-union-def)

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

lemma smap-union-empty2 [simp]: \( m \cup \{ \}. = m \)
  by (rule ext, auto simp add: smap-union-def)

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

lemma smap-Union-append [simp]: \( \bigcup (m1@m2) = (\bigcup \. m1) \cup (\bigcup \. m2) \)
  by (induct m1) auto

lemma smap-Union-rev [simp]: \( \bigcup (\text{rev } l) = \bigcup \. l \)
  by (induct l) (auto simp add: smap-union-comm)

lemma smap-Union-map-rev [simp]: \( \bigcup (\text{map } f \ (\text{rev } l)) = \bigcup (\text{map } f \ l) \)
  by (subst rev-map [THEN sym], subst smap-Union-rev, rule refl)

end

12. Sets of maps

theory MapSets
imports SetMap Utils
begin

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

definition maps-over :: \('a::type set \Rightarrow 'b::type set \Rightarrow ('a \to 'b) set\)
  where maps-over A B = \{ m. dom m \subseteq A \land ran m \subseteq B \}\n
lemma maps-over-empty [simp]:
  empty \in maps-over A B

unfolding maps-over-def by simp
lemma maps-over-upd:
  assumes m ∈ maps-over A B
  and v ∈ A and k ∈ B
shows m(v → k) ∈ maps-over A B
  using assms unfolding maps-over-def
  by (auto dest: subsetD[of ran-upd])

lemma maps-over-finite[intro]:
  assumes finite A and finite B shows finite (maps-over A B)
proof
  have inj-map-graph: inj (λf. {(x, y). Some y = f x})
  proof (induct rule: inj-onI)
    case (1 x y)
    from 1.hyps(3) have hyp: ∃ a b. (Some b = x a) ↔ (Some b = y a)
      by (simp add: set-eq_iff)
    show ?case
      proof (rule ext)
        fix z
        show x z = y z
          using hyp[of - z]
          by (cases x z, cases y z, auto)
      qed
  qed
  have (λf. {(x, y). Some y = f x}) ' maps-over A B ⊆ Pow( A × B ) (is ?graph ⊆ -)
    unfolding maps-over-def
    by (auto dest!: subsetD[of - A] subsetD[of - B] intro:ranI)
moreover
  have finite (Pow( A × B )) using assms by auto
ultimately
  have finite ?graph by (rule finite-subset)
thus ?thesis
  by (rule finite-imageD[OF - subset-inj-on[OF inj-map-graph subset-UNIV]])
qed

definition smaps-over :: 'a::type set ⇒ 'b::type set ⇒ ('a ⇒ 'b set) set
  where smaps-over A B = {m. sdom m ⊆ A ∧ sran m ⊆ B}

lemma smaps-over-empty[simp]:
  { }. ∈ smaps-over A B
unfolding smaps-over-def by simp

lemma smaps-over-singleton:
  assumes k ∈ A and vs ⊆ B
shows {k := vs}. ∈ smaps-over A B
  using assms unfolding smaps-over-def
  by(auto dest: subsetD[OF sdom-singleton])

lemma smaps-over-un:
  assumes m1 ∈ smaps-over A B and m2 ∈ smaps-over A B
\textbf{13. HOLCF Utility lemmas}

theory HOLCFUtils
imports HOLCF
begin

shows $m_1 \cup m_2 \in \text{smaps-over } A B$
using assms unfolding smaps-over-def
by (auto simp add: smap-union-def)

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

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

\textbf{lemma smaps-over-finite[intro]:}
assumes finite $A$ and finite $B$ shows 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: $a \land b. (b = x a \land b \neq \{} = (b = y a \land b \neq \{}$
    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 \mathit{Pow}(A \times \mathit{Pow} B)$ (is ?graph \subseteq -)
  unfolding smaps-over-def
  moreover
  have finite (\mathit{Pow}(A \times \mathit{Pow} B)) using assms by auto
  ultimately
  have finite ?graph by (rule finite-subset)
  thus ?thesis
  by (rule finite-imageD[OF - subset-inj-on[OF inj-smap-graph subset-UNIV]])
  qed

end
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.

\[
\text{instantiation bool :: po}
\begin{align*}
\text{definition} & \quad x \sqsubseteq y \leftrightarrow (x \rightarrow y) \\
\text{instance by} & \quad \text{standard (unfold below-bool-def, fast+)} \\
\text{end}
\end{align*}
\]

\[
\text{instance bool :: chfin}
\begin{align*}
\text{apply standard} \\
\text{apply (drule finite-range-imp-finch)} \\
\text{apply (rule finite)} \\
\text{apply (simp add: finite-chain-def)} \\
\text{done}
\end{align*}
\]

\[
\text{instance bool :: pcpo}
\begin{align*}
\text{proof} \\
\text{have } \forall y. \text{False } \subseteq y \text{ by (simp add: below-bool-def)} \\
\text{thus } \exists x::\text{bool}. \forall y. x \sqsubseteq y .. \\
\text{qed}
\end{align*}
\]

\[
\text{lemma is-lub-bool: } S << | (\text{True } \in S)
\begin{align*}
\text{unfolding} & \quad \text{is-lub-def is-ub-def below-bool-def by auto} \\
\text{lemma lab-bool: } & \quad \text{lab } S = (\text{True } \in S) \\
\text{using} & \quad \text{is-lub-bool by (rule lab-eqf)} \\
\text{lemma bottom-eq-False[simp]:} & \quad \bot = \text{False} \\
\text{by (rule below-antisym [OF minimal], simp add: below-bool-def)}
\end{align*}
\]

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

\[
\text{instantiation set :: (type) po}
\begin{align*}
\text{begin} \\
\text{definition} & \quad A \subseteq B \leftrightarrow A \subseteq B \\
\text{instance by} & \quad \text{standard (unfold below-set-def, fast+)} \\
\text{end}
\end{align*}
\]

\[
\text{lemma sqsubset-is-subset: } A \subseteq B \leftrightarrow A \subseteq B \\
\text{by (fact below-set-def)}
\]

\[
\text{lemma is-lub-set: } S << | \bigcup S
\]

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 standard (fast intro: is-lub-set)

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

instance set :: (type) pcpo
  by standard (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)
  apply (rule cont-apply [OF f])
  apply (rule chfindom-monofun2cont)
  apply (rule monofunI, simp add: below-bool-def)
  apply (rule cont-compose [OF - g])
  apply (rule chfindom-monofun2cont)
  apply (rule monofunI, simp add: below-bool-def)
  done

lemma cont2cont-imp [simp, cont2cont]:
  assumes f: cont (\lambda x. \neg f x) and g: cont (\lambda x. g x)
  shows cont (\lambda x. f x \rightarrow 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)

64
apply (auto simp add: is-lab-def is-ub-def below-bool-def lub-is-union)
done

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

unfolding Un-def by simp

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

unfolding insert-def using assms
by (intro cont2cont)

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

lemma cont2cont-UNION[cont2cont,simp]:
  assumes \[ \text{cont } f \]
  and \[ \land y. \text{cont } (\lambda x. g x y) \]
  shows \[ \text{cont } (\lambda x. \bigcup y\in f \ x. g x y) \]

proof (induct rule: contI2-case-names Mono Limit)
  case Mono
  show \[ \text{monofun } (\lambda x. \bigcup y\in f \ x. g x y) \]
  by (rule monofunI)(auto iff: sqsubset-is-subset dest: monofunE[OF assms(1)][THEN cont2mono]
      monofunE[OF assms(2)][THEN cont2mono])
  next
  case (Limit Y)
  have \[ (\bigcup y\in f \ Y i. g \ (\bigcup j. Y j) \ y) \subseteq (\bigcup \ k. \bigcup y\in f \ Y k. g \ Y k \ y) \]
  proof
    fix \[ x \]
    assume \[ x \in (\bigcup y\in f \ (\bigcup i. Y i). g \ (\bigcup j. Y j) \ y) \]
    then obtain \[ y \] where \[ y\in f \ Y i. g \ (\bigcup j. Y j) \ y \] by auto
    hence \[ y \in (\bigcup i. f \ Y i) \ \text{and } x\in (\bigcup j. g \ Y j) \ y \] by (auto simp add: cont2contlubE[OF assms(1) Limit(1)] cont2contlubE[OF assms(2) Limit(1)])
    then obtain \[ i \text{ and } j \\] where \[ yi: y\in f \ Y i \ \text{and } xj: x\in g \ Y j \ y \] by (auto simp add: lub-is-union)
    obtain \[ k \] where \[ i\leq k \ \text{and } j\leq k \] by (rule_tac x = \(i\) in max j in meta-allE)auto
    from \[ yi \text{ and } xj \]
    have \[ y\in f \ Y k \ \text{and } x\in g \ Y k \ y \]
    using monofunE[OF assms(1)][THEN cont2mono], OF chain-lubE[OF Limit(1) \(i\leq k\)]
    and monofunE[OF assms(2)][THEN cont2mono], OF chain-lubE[OF Limit(1) \(j\leq k\)]
    by (auto simp add:sqsubset-is-subset)
    hence \[ x\in (\bigcup y\in f \ Y k. g \ Y k \ y) \] by auto
    thus \[ x\in (\bigcup k. \bigcup y\in f \ Y k. g \ Y k \ y) \] by (auto simp add:lub-is-union)
  qed
  thus \(?\) case by (simp add:sqsubset-is-subset)
  qed

lemma cont2cont-Let-simple[simp,cont2cont]:
  assumes \[ \text{cont } \lambda x. g x t \]
  shows \[ \text{cont } \lambda x. \text{let } y = t \in g x y \]

65
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 :: '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]: adm (\lambda x . x)
by (rule adm-chfin)

lemma adm-Not[simp]: adm 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 computably, 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

\[ \mathcal{X}^A \cdot \mathcal{X}^B = \mathcal{X}^{A+B}. \]

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

lemma fix-transform:
  assumes \( \forall x. \ g \cdot (f \cdot x) = x \)
  shows \( \text{fix } F = g \cdot (\text{fix } (f \circ F \circ g)) \)
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 ⇒ 'c) × ('b discr ⇒ 'c) ⇒ ('a + 'b) discr ⇒ 'c::cpo
  where tup-to-sum = (λ p s. (λ(f,g)).
    case undiscr s of Inl x ⇒ f·(Discr x)
Inr x ⇒ g(Discr x) p)

**definition** sum-to-tup :: ((′a + ′b) discr → ′c) → (′a discr → ′c) × (′b discr → ′c::cpo)

**where** sum-to-tup = (Λ f. (Λ x. f·(Discr (Inl (undiscr x)))),

Λ x. f·(Discr (Inr (undiscr x))))

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

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

assumes cont f and cont g

shows cont (λx. case-sum (f x) (g x) s)

using assms by (cases s, auto intro:cont2cont-fun)

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

cont (λf. f ◦ g)

apply (rule cont2cont-lambda)

apply (subst comp-def)

apply (rule cont2cont-fun [of λx. x, OF cont-id])

done

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

assumes f1: cont f

and f2: Λ x. cont (f x)

and g1: cont g

and g2: Λ x. cont (g x)

shows cont (λ(a, b). (f a b, g a b))

apply (intro cont2cont)

apply (rule cont-apply [OF cont-snd - cont-const])

apply (rule cont-apply [OF cont-snd f2])

apply (rule cont-apply [OF cont-fst cont2cont-fun [OF f1] cont-const])

apply (rule cont-apply [OF cont-snd - cont-const])

apply (rule cont-apply [OF cont-snd g2])

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

done

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

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

sum-to-tup·f = (Λ x. f·(Discr (Inl (undiscr x)))), Λ x. f·(Discr (Inr (undiscr x))))

unfolding sum-to-tup-def by simp

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

tup-to-sum·p = (Λ s. (λ(f,g).

    case undiscr s of Inl x ⇒ f·(Discr x)
    | Inr x ⇒ g·(Discr x)) p)

unfolding tup-to-sum-def by simp
Generally, lambda abstractions with discrete domain are continuous and can be resolved immediately.

**lemma** discr-app[simp];

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

**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** sum-to-tup\cdot (tup-to-sum \cdot F) = F

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

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

**lemma** fix-transform-pair-sum;

**shows** fix\cdot F = sum-to-tup\cdot (fix\cdot (tup-to-sum oo F oo 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** f1: cont F

and f2: \( \forall x. \text{cont } (F x) \)

and g1: cont G

and g2: \( \forall x. \text{cont } (G x) \)

**shows** tup-to-sum oo (\( \Lambda p. (\lambda (a, b). (F a b, G a b)) \) \( p \)) oo sum-to-tup

\( = (\Lambda f s. \text{(case } \text{undiscr } s \text{ of } \text{Inl } x \Rightarrow F \cdot (\Lambda s. f \cdot (\text{Discr } (\text{Inl } \text{undiscr } s))))\)

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

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

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

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

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

**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];

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

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

end
References

