The Correctness of Launchbury's Natural Semantics for Lazy Evaluation

Joachim Breitner
Programming Paradigms Group
Karlsruhe Institute for Technology
breitner@kit.edu

October 13, 2025

In his seminal paper "Natural Semantics for Lazy Evaluation" [Lau93], John Launchbury proves his semantics correct with respect to a denotational semantics, and outlines an adequacy proof. We have formalized both semantics and machine-checked the correctness proof, clarifying some details. Furthermore, we provide a new and more direct adequacy proof that does not require intermediate operational semantics.

Contents

1 Introduction

The Natural Semantics for Lazy Evaluation [Lau93] created by John Launchbury in 1992 is often taken as the base for formal treatments of call-by-need evaluation, either to prove properties of lazy evaluation or as a base to describe extensions of the language or the implementation of the language. Therefore, assurance about the correctness and adequacy of the semantics is important in this field of research. Launchbury himself supports his semantics by defining a standard denotational semantics to prove both correctness and adequacy.

Although his proofs are already on the more rigorous side for pen-and-paper proofs, they have not yet been verified by transforming them to machine-checked proofs. The present work fills this gap by formalizing both semantics in the proof assistant Isabelle and proving both correctness and adequacy.

Our correctness formal proof is very close to the original proof. This is possible if the operator $\sqcup$ is understood as a right-sided update. If we were to understand $\sqcup$ as the least upper bound, then Theorem 2 in [Lau93], which is the generalization of the correctness statement used for Launchbury's inductive proof, is wrong. The main correctness result still holds, but needs a different proof; this is discussed in greater detail in [Bre13].

Launchbury outlines an adequacy proof via an intermediate operational semantics and resourced denotational semantics. The alternative operational semantics uses indirection instead of substitution for applications, does not update variable results and does not perform blackholing during evaluation of a variable. The equivalence of these two operational semantics is hard and tricky to prove. We found a direct proof for the adequacy of the original operational semantics and the (slightly modified) resourced denotational semantics. This is, as far as we know, the first complete and rigorous proof of adequacy of Launchbury's semantics.

In this development we extend Launchburys syntax and semantics with boolean values and an if-then-else construct, in order to base a subsequent work [**?**] on this. This extension does not affect the validity of the proven theorems, and the extra cases can simply be ignored if one is interested in the plain semantics. The next introductory section does exactly that. Unfortunately, such meta-level arguments are not easily implemented inside a theorem prover.

Our contributions are:

- We define the natural and denotational semantics given by Launchbury in the theorem prover Isabelle.
- We demonstrate how to use both the Nominal package (to handle name binding) [UK12] and the HOLCF [Huf12] package (for the domain-theoretic aspects) in the same development.
- We verify Launchbury's proof of correctness.

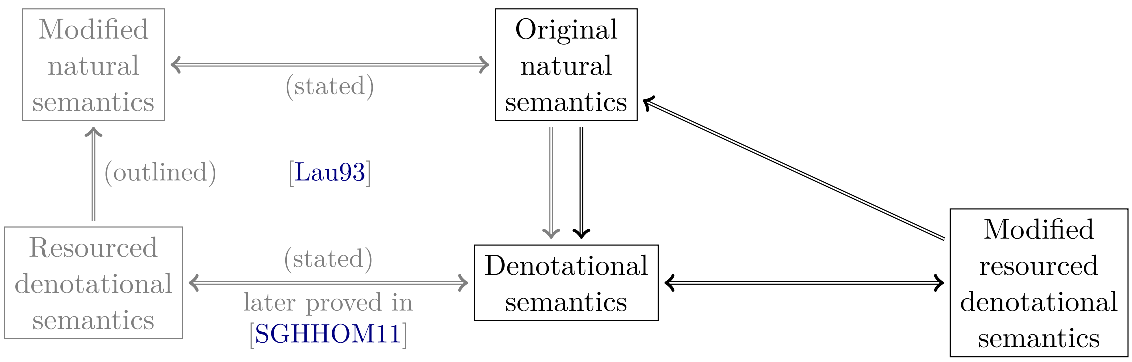
- We provide a new and more direct proof of adequacy.
- In order to do so, we formalize parts of [SGHHOM11], fixing a mistake in the proof.

1.1 Main definitions and theorems

For your convenience, the main definitions and theorems of the present work are assembled in this section. The following formulas are mechanically pretty-printed versions of the statements as defined resp. proven in Isabelle. Free variables are all-quantified. Some type conversion functions (like *set*) are omitted. The relations $\sharp$ and $\sharp*$ come from the Nominal package and express freshness of the variables on the left with regard to the expressions on the right.

1.1.1 The big picture

The following picture gives an overview of the different semantics. Elements printed in black are formally defined and proved in the present work, while the gray square on the left shows the proofs and propositions in Launchbury's original work [Lau93].



1.1.2 Expressions

The type *var* of variables is abstract and provided by the Nominal package. All we know about it is that it is countably infinite. Expressions of type *exp* are given by the following grammar:

$$\begin{array}{rlll} e & ::= & \lambda x.\ e & \text{lambda abstraction} \\ & \mid & e\ x & \text{application} \\ & \mid & x & \text{variable} \\ & \mid & \mathsf{\textit{let}}\ \mathit{as}\ \mathsf{\textit{in}}\ e & \text{recursive let} \end{array}$$

In the introduction we pretty-print expressions to resemble the notation in [Lau93] and omit the constructor names *Var*, *App*, *Lam* and *Let*. In the actual theories, these are visible. These expressions are, due to the machinery of the Nominal package, actually alpha-equivalency classes, so $\lambda x.\ x = \lambda y.\ y$ holds provably. This differs from Launchbury's original definition, which expects distinctly-named expressions and performs explicit alpha-renaming in the semantics.

The type *heap* is an abbreviation for $(var \times exp)$ *list*. These are *not* alpha-equivalency classes, i.e. we manage the bindings in heaps explicitly.

1.1.3 The natural semantics

Launchbury's original semantics, extended with some technical overhead related to name binding (following [Ses97]), is defined as follows:

$$\frac{}{\Gamma : \lambda x.\ e \Downarrow_L \Gamma : \lambda x.\ e} \quad \text{LAMBDA}$$

$$\frac{y \sharp (\Gamma,\ e,\ x,\ L,\ \Delta,\ \Theta,\ z) \qquad \Gamma : e \Downarrow_L \Delta : \lambda y.\ e' \qquad \Delta : e'[y{::=}x] \Downarrow_L \Theta : z}{\Gamma : e\ x \Downarrow_L \Theta : z} \quad \text{APPLICATION}$$

$$\frac{(x,\ e) \in \Gamma \qquad \Gamma \backslash x : e \Downarrow_{x\ \cdot\ L} \Delta : z}{\Gamma : x \Downarrow_L (x,\ z) \cdot \Delta : z} \quad \text{VARIABLE}$$

$$\frac{\mathit{dom}\ \Delta \sharp{*}\ (\Gamma,\ L) \qquad \Delta \mathbin{@} \Gamma : \mathit{body} \Downarrow_L \Theta : z}{\Gamma : \mathit{let}\ \Delta\ \mathit{in}\ \mathit{body} \Downarrow_L \Theta : z} \quad \text{LET}$$

1.1.4 The denotational semantics

The value domain of the denotational semantics is the initial solution to

$$D = [D \to D]_\perp$$

as introduced in [Abr90]. The type *Value*, together with the bottom value $\perp$, the injection *Fn* and the projection $\_ \downarrow Fn\ \_ :: Value \to Value \to Value$, is constructed as a pointed chain-complete partial order from this equation by the HOLCF package. The type of semantic environments is $var \Rightarrow Value$.

The semantics of an expression $e::exp$ in an environment $\varrho::var \Rightarrow Value$ is written $[\![\ e\ ]\!]_\varrho :: Value$ and defined by the following equations:

$$\begin{aligned}
[\![\ \lambda x.\ e\ ]\!]_\varrho &= Fn{\cdot}(\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x\ :=\ v)}) \\
[\![\ e\ x\ ]\!]_\varrho &= [\![\ e\ ]\!]_\varrho \downarrow Fn\ \varrho\ x \\
[\![\ x\ ]\!]_\varrho &= \varrho\ x \\
[\![\ \mathit{let}\ \Gamma\ \mathit{in}\ \mathit{body}\ ]\!]_\varrho &= [\![\ \mathit{body}\ ]\!]_{\{\!|\Gamma|\!\}\varrho}.
\end{aligned}$$

The expression $[\![\ \Gamma\ ]\!]_\varrho$ maps the evaluation function over a heap, returning an environment:

$$
\begin{aligned}
([\![\ \Gamma\ ]\!]_\varrho)\ v &= [\![\ e\ ]\!]_\varrho && \text{if } (v,\ e) \in \Gamma \\
([\![\ \Gamma\ ]\!]_\varrho)\ v &= \bot && \text{if } v \notin \mathsf{dom}\ \Gamma
\end{aligned}
$$

The semantics $\{\!|\Gamma|\!\}\varrho{::}var \Rightarrow Value$ of a heap $\Gamma{::}heap$ in an environment $\varrho{::}var \Rightarrow Value$ is defined by the recursive equation

$$\{\!|\Gamma|\!\}\varrho = \varrho \ {+}{+}_{\mathsf{dom}\ \Gamma}\ [\![\ \Gamma\ ]\!]_{\{\!|\Gamma|\!\}\varrho}$$

where

$$
\begin{aligned}
(f\ {+}{+}_A\ g)\ a &= f\ a && \text{if } a \notin A \\
(f\ {+}{+}_A\ g)\ a &= g\ a && \text{if } a \in A.
\end{aligned}
$$

The semantics of the heap in the empty environment $\bot$ is abbreviated as $\{\!|\Gamma|\!\}$.

1.1.5 Correctness and Adequacy

The statement of correctness reads: If $\Gamma : e \Downarrow_L \Delta : v$ and, as a side condition, $fv\ (\Gamma,\ e) \subseteq L \cup \mathsf{dom}\ \Gamma$ holds, then

$$[\![\ e\ ]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![\ v\ ]\!]_{\{\!|\Delta|\!\}\varrho}.$$

The statement of adequacy reads:

$$\text{If } [\![\ e\ ]\!]_{\{\!|\Gamma|\!\}} \neq \bot \text{ then } \exists \Delta\ v.\ \Gamma : e \Downarrow_S \Delta : v.$$

1.2 Differences to our previous work

We have previously published [Bre13] of which the present work is a continuation. They differ in scope and focus:

1.2.1 The treatment of $\sqcup$

In [Bre13], the question of the precise meaning of $\sqcup$ is discussed in detail. The original paper is not clear about whether this operator denotes the least upper bound, or the right-sided override operator. A lemma stated in [Lau93] only holds if $\sqcup$ is the least upper bound, but with that definition, Launchbury's Theorem 2 – the generalized correctness theorem – is false; a counter-example is given in [Bre13].

We came up with an alternative operational semantics that keeps more of the evaluation context in the judgments and allows the correctness theorem to be proved inductively without the problematic generalization. We proved the two operational semantics equivalent and thus obtained the (non-generalized) correctness of Launchbury's semantics.

We also showed that if one takes $\sqcup$ to be the update operator, Theorem 2 holds and the proof goes through as it is. Furthermore, we showed that the resulting denotational semantics are identical for expressions, and can differ only for heaps. Therefore, the question of the precise meaning of $\sqcup$ can be considered of little importance and for the present work we soley work with right sided updates. We also avoid the ambiguos syntax $\sqcup$ and write $\_ \; {+}{+}_{\_} \; \_$ instead (the index indicates on what set the function on the right overrides the function on the left). The alternative operational semantics is not included in this work.

1.2.2 The types of environments

Another difference is the choice of the type for environments, which map variables to semantics values. A naive choice is $var \Rightarrow Value$, but this causes problems when defining the value semantics, for which

$$[\![\ \lambda x.\ e\ ]\!]_{\varrho} = Fn \cdot (\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x\ :=\ v)})$$

is a defining equation. The argument on the left hand side is the representative of an equivalence class (defined using the Nominal package), so this is only allowed if the right hand side is indeed independent of the actual choice of x. This is shown most commonly and easily if x is fresh in all the other arguments ($x \sharp \varrho$), and indeed the Nominal package allows us to specify this as a side condition to the defining equation, which is what we did in [Bre13].

But this convenience comes as a price: Such side-conditions are only allowed if the argument has finite support (otherwise there might no variable fulfilling $x \sharp \varrho$). More precisely: The type of the argument must be a member of the *fs* typeclass provided by the Nominal package. The type $var \Rightarrow Value$ cannot be made a member of this class, as there obviously are elements that have infinite support. The fix here was to introduce a new type constructor, *fmap*, for partial functions with finite domain. This is fine: Only functions with finite domain matter in our formalisation.

The introduction of *fmap* had further consequences. The main type class of the HOLCF package, which we use to define domains and continuous functions on them, is the class *cpo*, of chain-complete partial orders. With the usual ordering on partial functions, (*var*, *Value*) *fmap* cannot be a member of this class. The fix here is to use a different ordering and only let elements be compareable that have the same domain. In our formalisation, the domain is alway known (e.g. all variables bound on some heap), so this worked out.

But not without causing yet another issue: With this ordering, (*var*, *Value*) *fmap* is a *cpo*, but lacks a bottom element, i.e. now it is no *pcpo*, and HOLCF's built-in operator

$\mu\ x.\ f\ x$ for expressing least fixed-points, as they occur in the semantics of heaps, is not available. Furthermore, $\sqcup$ is not a total function, i.e. defined only on a subset of all possible arguments. The solution was a rather convoluted set of theories that formalize functions that are continuous on a specific set, fixed-points on such sets etc.

In the present work, this problems is solved in a much more elegant way. Using a small trick we defined the semantics functions so that

$$[\![\ \lambda x.\ e\ ]\!]_{\varrho} = \mathit{Fn}\cdot(\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x\ :=\ v)})$$

holds unconditionally. The actual, technical definition is

$$[\![\ \lambda x.\ e\ ]\!]_{\varrho} = \mathit{Fn}\cdot(\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho|_{\mathit{fv}\ (\lambda x.\ e)}(x\ :=\ v)})$$

where the right-hand-side can be shown to be invariant of the choice of x, as $x \notin \mathit{fv}$ $(\lambda x.\ e)$. Once the function is defined, the equality $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho|_{\mathit{fv}\ e}}$ can be proved. With that, the desired equation for $[\![\ \lambda x.\ e\ ]\!]_{\varrho}$ follows. The same trick is applied to the equation for $[\![\ \textsf{let}\ \Gamma\ \textsf{in}\ \mathit{body}\ ]\!]_{\varrho}$.

This allows us to use the type $\mathit{var} \Rightarrow \mathit{Value}$ for the semantic envionments and considerably simplifies the formalization compared to [Bre13].

1.2.3 No type *assn*

The nominal package provides means to define types that are alpha-equivalence classes, and we use that to define our type *exp*, which contains a constructor **let** *binds* **in** *expr*. The desired type of the parameter for the binding is $(\mathit{var} \times \mathit{exp})\ \mathit{list}$, but the Nominal package does not support such nested recursion, and requires a mutual recursive definition with a custom type (*assn*) with constructors *ANil* and *ACons* that is isomorphic to $(\mathit{var} \times \mathit{exp})\ \mathit{list}$. In [Bre13], this type and conversion functions from and to $(\mathit{var} \times \mathit{exp})\ \mathit{list}$ cluttered the whole development. In the present work we improved this by defining the type with a “temporary” constructor *LetA*. Afterwards we define conversions functions and the desired constructor *Let*, and re-state all lemmas produced by the Nominal package (such as type exhaustiveness, distinctiveness of constructors and the induction rules) with that constructor. From that point on, the development is free of the crutch *assn*.

In short, the notable changes in this work over [Bre13] are:

- We consider $\sqcup$ to be a right-sided update and do discuss neither the problem with $\sqcup$ denoting the least uppper bound, nor possible solutions.
- This, a simpler choice for the type of semantic environments and a better definition of the type for terms, considerably simplifies the work.
- Most importantly, this work contains a complete and formal proof of the adequacy of Launchbury’s semantics.

1.3 Related work

Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero and Yolanda Ortega-Mallén have worked on formal aspects of Launchbury's semantics as well.

They identified a step in his adequacy proof relating the standard and the resourced denotational semantics that is not as trivial as it seems at first and worked out a detailed pen-and-paper proof [SGHHOM11], where they first construct a similarity relation $\_ \lhd\rhd \_$ between the standard semantic domain (*Value*) and the resourced domain (*CValue*) and show that the denotation semantics yield similar results ($\varrho \lhd\rhd^* \sigma \Longrightarrow [\![e]\!]_\varrho \lhd\rhd (\mathcal{N}[\![e]\!]_\sigma)\cdot C^\infty$), which is one step in the adequacy proof. We formalized this (Sections 8.1 and 8.2), identifying and fixing a mistake in the paper (Lemma 2.3(3) does not hold; the problem can be fixed by applying an extra round of take-induction in the proof of Proposition 9).

Currently, they are working on completing the adequacy proof as outlined by Launchbury, i.e. by going via the alternative natural semantics given in [Lau93], which differs from the semantics above in that the application rule works with an indirection on the heap instead of a substitution and that the variable rule has no blackholing and no update. In [SGHHOM14], they relate the original semantics with one where indirections have been introduced. The next step, modifying the variable rule, is under development. Once that is done they can close the loop and have completed Launchbury's work.

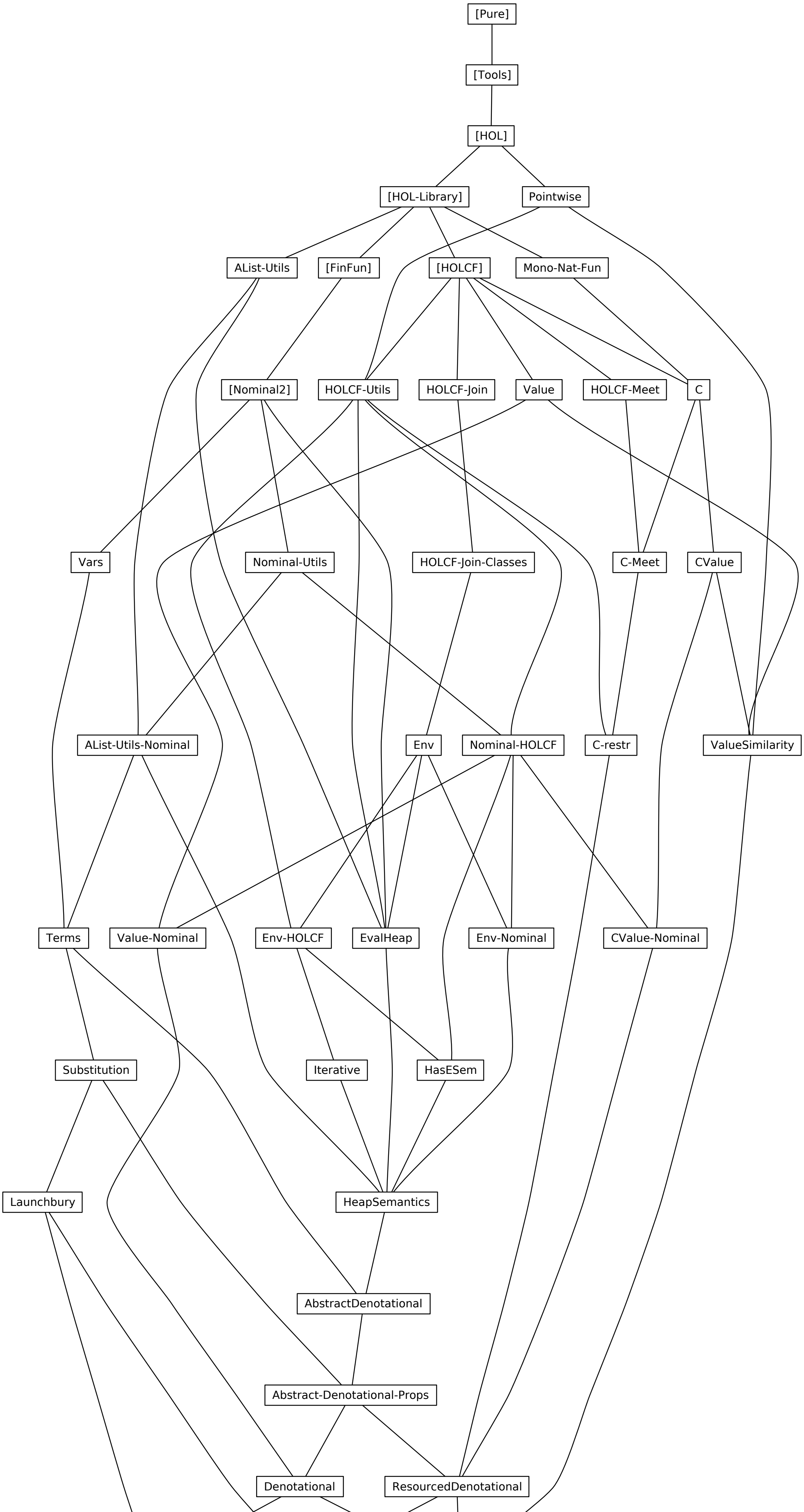
This work proves the adequacy as stated by Launchbury as well, but in contrast to his proof outline no alternative operational semantics is introduced. The problems of indirection vs. substitution and of blackholing is solved on the denotational side instead, which turned out to be much easier than proving the various operational semantics to be equivalent.

1.4 Theory overview

The following chapters contain the complete Isabelle theories, with one section per theory. Their interdependencies are visualized in Figure 1.

Chapter 2 contains auxiliary theories, not necessarily tied to Launchbury's semantics. The base theories are kept independent of Nominal and HOLCF where possible, the lemmas combining them are in theories of their own, creatively named by appending *-Nominal* resp. *-HOLCF*. You will find these theories:

- A definition for lifting a relation point-wise (*Pointwise*).
- A collection of definition related to associative lists (*AList-Utils*, *AList-Utils-Nominal*).
- A characterization of monotonous functions $\mathbb{N} \to \mathbb{N}$ (*Mono-Nat-Fun*).
- General utility functions extending Nominal (*Nominal-Utils*).


[Pure]
[Tools]
[HOL]
[HOL-Library]
Pointwise
AList-Utils
[FinFun]
[HOLCF]
Mono-Nat-Fun
[Nominal2]
HOLCF-Utils
HOLCF-Join
Value
HOLCF-Meet
C
Vars
Nominal-Utils
HOLCF-Join-Classes
C-Meet
CValue
AList-Utils-Nominal
Env
Nominal-HOLCF
C-restr
ValueSimilarity
Terms
Value-Nominal
Env-HOLCF
EvalHeap
Env-Nominal
CValue-Nominal
Substitution
Iterative
HasESem
Launchbury
HeapSemantics
AbstractDenotational
Abstract-Denotational-Props
Denotational
ResourcedDenotational

- General utility functions extending HOLCF (*HOLCF-Utils*).
- Binary meets in the context of HOLCF (*HOLCF-Meet*).
- A theory combining notions from HOLCF and Nominal, e.g. continuity of permutation (*Nominal-HOLCF*).
- A theory for working with pcpo-valued functions as semantic environments (*Env*, *Env-Nominal*, *Env-HOLCF*).
- A function *evalHeap* that converts between associative lists and functions. (*Eval-Heap*)

Chapter 3 defines the syntax and Launchbury's natural semantics.

Chapter 4 sets the stage for the denotational semantics by defining a locale *semantic-domain* for denotational domains, and an instantiation for the standard domain.

Chapter 5 defines the denotational semantics. It also introduces the locale *has-ESem* which abstracts over the value semantics when defining the semantics of heaps.

Chapter 6 defines the resourced denotational semantics.

Chapter 7 proves the correctness of Launchbury's semantics with regard to both denotational semantics. We need the correctness with regard to the resourced semantics in the adequacy proof.

Chapter 8 proves the two denotational semantics related, which is used in

Chapter 9, where finally the adequacy is proved.

1.5 Acknowledgements

I'd like to thank Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero and Yolanda Ortega-Mallén for inviting me to Madrid to discuss our respective approaches.
This work was supported by the Deutsche Telekom Stiftung.

2 Auxiliary theories

2.1 Pointwise

theory *Pointwise* **imports** *Main* **begin**

Lifting a relation to a function.

definition *pointwise* **where** $pointwise\ P\ m\ m' = (\forall\ x.\ P\ (m\ x)\ (m'\ x))$

lemma *pointwiseI*[*intro*]: $(\bigwedge\ x.\ P\ (m\ x)\ (m'\ x)) \Longrightarrow pointwise\ P\ m\ m'$ ⟨*proof*⟩

end

2.2 AList-Utils

theory *AList−Utils*
imports *Main HOL−Library.AList*
begin
declare *implies-True-equals* [*simp*] *False-implies-equals*[*simp*]

We want to have *delete* and *update* back in the namespace.

abbreviation *delete* **where** *delete* ≡ *AList.delete*
abbreviation *update* **where** *update* ≡ *AList.update*
abbreviation *restrictA* **where** *restrictA* ≡ *AList.restrict*
abbreviation *clearjunk* **where** *clearjunk* ≡ *AList.clearjunk*

lemmas *restrict-eq* = *AList.restrict-eq*
  and *delete-eq* = *AList.delete-eq*

lemma *restrictA-append*: *restrictA S* (*a*@*b*) = *restrictA S a* @ *restrictA S b*
  ⟨*proof*⟩

lemma *length-restrictA-le*: *length* (*restrictA S a*) ≤ *length a*
  ⟨*proof*⟩

2.2.1 The domain of an associative list

definition *domA*
  where *domA h* = *fst* ‘ *set h*

lemma *domA-append*[*simp*]:*domA* (*a* @ *b*) = *domA a* ∪ *domA b*
  and [*simp*]:*domA* ((*v*,*e*) # *h*) = *insert v* (*domA h*)
  and [*simp*]:*domA* (*p* # *h*) = *insert* (*fst p*) (*domA h*)
  and [*simp*]:*domA* [] = {}
  ⟨*proof*⟩

lemma *domA-from-set*:

$(x,\ e) \in set\ h \Longrightarrow x \in domA\ h$
$\langle proof \rangle$

lemma *finite-domA[simp]*:
$finite\ (domA\ \Gamma)$
$\langle proof \rangle$

lemma *domA-delete[simp]*:
$domA\ (delete\ x\ \Gamma) = domA\ \Gamma - \{x\}$
$\langle proof \rangle$

lemma *domA-restrictA[simp]*:
$domA\ (restrictA\ S\ \Gamma) = domA\ \Gamma \cap S$
$\langle proof \rangle$

lemma *delete-not-domA[simp]*:
$x \notin domA\ \Gamma \Longrightarrow\ delete\ x\ \Gamma = \Gamma$
$\langle proof \rangle$

lemma *deleted-not-domA*: $x \notin domA\ (delete\ x\ \Gamma)$
$\langle proof \rangle$

lemma *dom-map-of-conv-domA*:
$dom\ (map\text{-}of\ \Gamma) = domA\ \Gamma$
$\langle proof \rangle$

lemma *domA-map-of-Some-the*:
$x \in domA\ \Gamma \Longrightarrow map\text{-}of\ \Gamma\ x = Some\ (the\ (map\text{-}of\ \Gamma\ x))$
$\langle proof \rangle$

lemma *domA-clearjunk[simp]*: $domA\ (clearjunk\ \Gamma) = domA\ \Gamma$
$\langle proof \rangle$

lemma *the-map-option-domA[simp]*: $x \in domA\ \Gamma \Longrightarrow the\ (map\text{-}option\ f\ (map\text{-}of\ \Gamma\ x)) = f\ (the\ (map\text{-}of\ \Gamma\ x))$
$\langle proof \rangle$

lemma *map-of-domAD*: $map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow x \in domA\ \Gamma$
$\langle proof \rangle$

lemma *restrictA-noop*: $domA\ \Gamma \subseteq S \Longrightarrow restrictA\ S\ \Gamma = \Gamma$
$\langle proof \rangle$

lemma *restrictA-cong*:
$(\bigwedge x.\ x \in domA\ m1 \Longrightarrow x \in V \longleftrightarrow x \in V') \Longrightarrow m1 = m2 \Longrightarrow restrictA\ V\ m1 = restrictA\ V'\ m2$
$\langle proof \rangle$

2.2.2 Other lemmas about associative lists

lemma *delete-set-none*: $(\textit{map-of } l)(x := \textit{None}) = \textit{map-of } (\textit{delete } x\ l)$
⟨*proof*⟩

lemma *list-size-delete*[*simp*]: $\textit{size-list size } (\textit{delete } x\ l) < \textit{Suc } (\textit{size-list size } l)$
⟨*proof*⟩

lemma *delete-append*[*simp*]: $\textit{delete } x\ (l1\ @\ l2) = \textit{delete } x\ l1\ @\ \textit{delete } x\ l2$
⟨*proof*⟩

lemma *map-of-delete-insert*:
assumes $\textit{map-of } \Gamma\ x = \textit{Some } e$
shows $\textit{map-of } ((x,e)\ \#\ \textit{delete } x\ \Gamma) = \textit{map-of } \Gamma$
⟨*proof*⟩

lemma *map-of-delete-iff*[*simp*]: $\textit{map-of } (\textit{delete } x\ \Gamma)\ xa = \textit{Some } e \longleftrightarrow (\textit{map-of } \Gamma\ xa = \textit{Some } e) \wedge xa \neq x$
⟨*proof*⟩

lemma *map-add-domA*[*simp*]:
$x \in \textit{domA } \Gamma \Longrightarrow (\textit{map-of } \Delta\ {+}{+}\ \textit{map-of } \Gamma)\ x = \textit{map-of } \Gamma\ x$
$x \notin \textit{domA } \Gamma \Longrightarrow (\textit{map-of } \Delta\ {+}{+}\ \textit{map-of } \Gamma)\ x = \textit{map-of } \Delta\ x$
⟨*proof*⟩

lemma *set-delete-subset*: $\textit{set } (\textit{delete } k\ al) \subseteq \textit{set } al$
⟨*proof*⟩

lemma *dom-delete-subset*: $\textit{snd } \textit{`}\ \textit{set } (\textit{delete } k\ al) \subseteq \textit{snd } \textit{`}\ \textit{set } al$
⟨*proof*⟩

lemma *map-ran-cong*[*fundef-cong*]:
$[\![\bigwedge x\ .\ x \in \textit{set } m1 \Longrightarrow f1\ (\textit{fst } x)\ (\textit{snd } x) = f2\ (\textit{fst } x)\ (\textit{snd } x)\ ;\ m1 = m2\ ]\!]$
$\Longrightarrow \textit{map-ran } f1\ m1 = \textit{map-ran } f2\ m2$
⟨*proof*⟩

lemma *domA-map-ran*[*simp*]: $\textit{domA } (\textit{map-ran } f\ m) = \textit{domA } m$
⟨*proof*⟩

lemma *map-ran-delete*:
$\textit{map-ran } f\ (\textit{delete } x\ \Gamma) = \textit{delete } x\ (\textit{map-ran } f\ \Gamma)$
⟨*proof*⟩

lemma *map-ran-restrictA*:
$\textit{map-ran } f\ (\textit{restrictA } V\ \Gamma) = \textit{restrictA } V\ (\textit{map-ran } f\ \Gamma)$
⟨*proof*⟩

lemma *map-ran-append*:
$\textit{map-ran } f\ (\Gamma @ \Delta) = \textit{map-ran } f\ \Gamma\ @\ \textit{map-ran } f\ \Delta$

$\langle proof \rangle$

2.2.3 Syntax for map comprehensions

definition $mapCollect :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'c\ set$
where $mapCollect\ f\ m = \{f\ k\ v \mid k\ v\ .\ m\ k = Some\ v\}$

syntax
$\text{-}MapCollect :: 'c \Rightarrow pttrn => pttrn \Rightarrow 'a \rightharpoonup 'b => 'c\ set$ $\quad$ (‹(1{- |/-/↦/-/∈/-/})›)
syntax-consts
$\text{-}MapCollect == mapCollect$
translations
$\{e \mid k{\mapsto}v \in m\} == CONST\ mapCollect\ (\lambda k\ v.\ e)\ m$

lemma $mapCollect\text{-}empty[simp]$: $\{f\ k\ v \mid k \mapsto v \in Map.empty\} = \{\}$
$\langle proof \rangle$

lemma $mapCollect\text{-}const[simp]$:
$m \neq Map.empty \Longrightarrow \{e \mid k{\mapsto}v{\in}m\} = \{e\}$
$\langle proof \rangle$

lemma $mapCollect\text{-}cong[fundef\text{-}cong]$:
$(\bigwedge k\ v.\ m1\ k = Some\ v \Longrightarrow f1\ k\ v = f2\ k\ v) \Longrightarrow m1 = m2 \Longrightarrow mapCollect\ f1\ m1 = mapCollect\ f2\ m2$
$\langle proof \rangle$

lemma $mapCollectE[elim!]$:
assumes $x \in \{f\ k\ v \mid k \mapsto v \in m\}$
obtains $k\ v$ **where** $m\ k = Some\ v$ **and** $x = f\ k\ v$
$\langle proof \rangle$

lemma $mapCollectI[intro]$:
assumes $m\ k = Some\ v$
shows $f\ k\ v \in \{f\ k\ v \mid k \mapsto v \in m\}$
$\langle proof \rangle$

lemma $ball\text{-}mapCollect[simp]$:
$(\forall\ x \in \{f\ k\ v \mid k \mapsto v \in m\}.\ P\ x) \longleftrightarrow (\forall\ k\ v.\ m\ k = Some\ v \longrightarrow P\ (f\ k\ v))$
$\langle proof \rangle$

lemma $image\text{-}mapCollect[simp]$:
$g\ `\ \{f\ k\ v \mid k \mapsto v \in m\} = \{\ g\ (f\ k\ v) \mid k \mapsto v \in m\}$
$\langle proof \rangle$

lemma $mapCollect\text{-}map\text{-}upd[simp]$:
$mapCollect\ f\ (m(k{\mapsto}v)) = insert\ (f\ k\ v)\ (mapCollect\ f\ (m(k := None)))$
$\langle proof \rangle$

definition *mapCollectFilter* :: $('a \Rightarrow 'b \Rightarrow (bool \times 'c)) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'c\ set$
  where *mapCollectFilter f m* = $\{snd\ (f\ k\ v) \mid k\ v\ .\ m\ k = Some\ v \wedge fst\ (f\ k\ v)\}$

syntax
  -MapCollectFilter :: $'c \Rightarrow pttrn \Rightarrow pttrn \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool \Rightarrow 'c\ set$ (‹(1{- |/-/↦/-/∈/-/./-})›)
syntax-consts
  -MapCollectFilter == *mapCollectFilter*
translations
  $\{e \mid k{\mapsto}v \in m\ .\ P\ \}$ == *CONST mapCollectFilter* $(\lambda k\ v.\ (P,e))\ m$

lemma *mapCollectFilter-const-False*[*simp*]:
  $\{e \mid k{\mapsto}v \in m\ .\ False\ \} = \{\}$
  ⟨*proof*⟩

lemma *mapCollectFilter-const-True*[*simp*]:
  $\{e \mid k{\mapsto}v \in m\ .\ True\ \} = \{e \mid k{\mapsto}v \in m\}$
  ⟨*proof*⟩

end

2.3 Mono-Nat-Fun

theory *Mono−Nat−Fun*
imports *HOL−Library.Infinite-Set*
begin

The following lemma proves that a monotonous function from and to the natural numbers is either eventually constant or unbounded.

lemma *nat-mono-characterization*:
  fixes f :: *nat* ⇒ *nat*
  assumes *mono f*
  obtains n **where** $\bigwedge m\ .\ n \leq m \Longrightarrow f\ n = f\ m \mid \bigwedge\ m\ .\ \exists\ n\ .\ m \leq f\ n$
⟨*proof*⟩

end

2.4 Nominal-Utils

theory *Nominal−Utils*
imports *Nominal2.Nominal2 HOL−Library.AList*
begin

2.4.1 Lemmas helping with equivariance proofs

lemma *perm-rel-lemma*:
  assumes $\bigwedge \pi\ x\ y.\ r\ (\pi \cdot x)\ (\pi \cdot y) \Longrightarrow r\ x\ y$
  shows $r\ (\pi \cdot x)\ (\pi \cdot y) \longleftrightarrow r\ x\ y$ (**is** $?l \longleftrightarrow ?r$)
$\langle proof \rangle$

lemma *perm-rel-lemma2*:
  assumes $\bigwedge \pi\ x\ y.\ r\ x\ y \Longrightarrow r\ (\pi \cdot x)\ (\pi \cdot y)$
  shows $r\ x\ y \longleftrightarrow r\ (\pi \cdot x)\ (\pi \cdot y)$ (**is** $?l \longleftrightarrow ?r$)
$\langle proof \rangle$

lemma *fun-eqvtI*:
  assumes *f-eqvt*[*eqvt*]: $(\bigwedge\ p\ x.\ p \cdot (f\ x) = f\ (p \cdot x))$
  shows $p \cdot f = f$ $\langle proof \rangle$

lemma *eqvt-at-apply*:
  assumes *eqvt-at f x*
  shows $(p \cdot f)\ x = f\ x$
$\langle proof \rangle$

lemma *eqvt-at-apply′*:
  assumes *eqvt-at f x*
  shows $p \cdot f\ x = f\ (p \cdot x)$
$\langle proof \rangle$

lemma *eqvt-at-apply′′*:
  assumes *eqvt-at f x*
  shows $(p \cdot f)\ (p \cdot x) = f\ (p \cdot x)$
$\langle proof \rangle$

lemma *size-list-eqvt*[*eqvt*]: $p \cdot$ *size-list* $f\ x =$ *size-list* $(p \cdot f)\ (p \cdot x)$
$\langle proof \rangle$

2.4.2 Freshness via equivariance

lemma *eqvt-fresh-cong1*: $(\bigwedge p\ x.\ p \cdot (f\ x) = f\ (p \cdot x)) \Longrightarrow a \sharp x \Longrightarrow a \sharp f\ x$
  $\langle proof \rangle$

lemma *eqvt-fresh-cong2*:
  assumes *eqvt*: $(\bigwedge p\ x\ y.\ p \cdot (f\ x\ y) = f\ (p \cdot x)\ (p \cdot y))$
  and *fresh1*: $a \sharp x$ **and** *fresh2*: $a \sharp y$
  shows $a \sharp f\ x\ y$
$\langle proof \rangle$

lemma *eqvt-fresh-star-cong1*:
  assumes *eqvt*: $(\bigwedge p\ x.\ p \cdot (f\ x) = f\ (p \cdot x))$
  and *fresh1*: $a \sharp* x$
  shows $a \sharp* f\ x$

⟨*proof*⟩

lemma *eqvt-fresh-star-cong2*:
assumes *eqvt*: $(\bigwedge p\ x\ y.\ p \cdot (f\ x\ y) = f\ (p \cdot x)\ (p \cdot y))$
and *fresh1*: $a \sharp* x$ **and** *fresh2*: $a \sharp* y$
shows $a \sharp* f\ x\ y$
⟨*proof*⟩

lemma *eqvt-fresh-cong3*:
assumes *eqvt*: $(\bigwedge p\ x\ y\ z.\ p \cdot (f\ x\ y\ z) = f\ (p \cdot x)\ (p \cdot y)\ (p \cdot z))$
and *fresh1*: $a \sharp x$ **and** *fresh2*: $a \sharp y$ **and** *fresh3*: $a \sharp z$
shows $a \sharp f\ x\ y\ z$
⟨*proof*⟩

lemma *eqvt-fresh-star-cong3*:
assumes *eqvt*: $(\bigwedge p\ x\ y\ z.\ p \cdot (f\ x\ y\ z) = f\ (p \cdot x)\ (p \cdot y)\ (p \cdot z))$
and *fresh1*: $a \sharp* x$ **and** *fresh2*: $a \sharp* y$ **and** *fresh3*: $a \sharp* z$
shows $a \sharp* f\ x\ y\ z$
⟨*proof*⟩

2.4.3 Additional simplification rules

lemma *not-self-fresh*[*simp*]: $atom\ x \sharp x \longleftrightarrow False$
⟨*proof*⟩

lemma *fresh-star-singleton*: $\{\ x\ \} \sharp* e \longleftrightarrow x \sharp e$
⟨*proof*⟩

2.4.4 Additional equivariance lemmas

lemma *eqvt-cases*:
fixes $f\ x\ \pi$
assumes *eqvt*: $\bigwedge x.\ \pi \cdot f\ x = f\ (\pi \cdot x)$
obtains $f\ x\ f\ (\pi \cdot x) \mid \neg\ f\ x \quad \neg\ f\ (\pi \cdot x)$
⟨*proof*⟩

lemma *range-eqvt*: $\pi \cdot range\ Y = range\ (\pi \cdot Y)$
⟨*proof*⟩

lemma *case-option-eqvt*[*eqvt*]:
$\pi \cdot case\text{-}option\ d\ f\ x = case\text{-}option\ (\pi \cdot d)\ (\pi \cdot f)\ (\pi \cdot x)$
⟨*proof*⟩

lemma *supp-option-eqvt*:
$supp\ (case\text{-}option\ d\ f\ x) \subseteq supp\ d \cup supp\ f \cup supp\ x$
⟨*proof*⟩

lemma *funpow-eqvt*[*simp*,*eqvt*]:
$\pi \cdot ((f :: 'a \Rightarrow 'a::pt)\ \widehat{\ }\widehat{\ }\ n) = (\pi \cdot f)\ \widehat{\ }\widehat{\ }\ (\pi \cdot n)$
⟨*proof*⟩

lemma *delete-eqvt*[*eqvt*]:
$\pi \cdot$ *AList.delete* $x\ \Gamma$ = *AList.delete* $(\pi \cdot x)\ (\pi \cdot \Gamma)$
⟨*proof*⟩

lemma *restrict-eqvt*[*eqvt*]:
$\pi \cdot$ *AList.restrict* $S\ \Gamma$ = *AList.restrict* $(\pi \cdot S)\ (\pi \cdot \Gamma)$
⟨*proof*⟩

lemma *supp-restrict*:
supp (*AList.restrict* $S\ \Gamma$) $\subseteq$ *supp* Γ
⟨*proof*⟩

lemma *clearjunk-eqvt*[*eqvt*]:
$\pi \cdot$ *AList.clearjunk* Γ = *AList.clearjunk* $(\pi \cdot \Gamma)$
⟨*proof*⟩

lemma *map-ran-eqvt*[*eqvt*]:
$\pi \cdot$ *map-ran* $f\ \Gamma$ = *map-ran* $(\pi \cdot f)\ (\pi \cdot \Gamma)$
⟨*proof*⟩

lemma *dom-perm*:
dom $(\pi \cdot f) = \pi \cdot$ (*dom* f)
⟨*proof*⟩

lemmas *dom-perm-rev*[*simp*,*eqvt*] = *dom-perm*[*symmetric*]

lemma *ran-perm*[*simp*]:
$\pi \cdot$ (*ran* f) = *ran* $(\pi \cdot f)$
⟨*proof*⟩

lemma *map-add-eqvt*[*eqvt*]:
$\pi \cdot (m1\ {+}{+}\ m2) = (\pi \cdot m1)\ {+}{+}\ (\pi \cdot m2)$
⟨*proof*⟩

lemma *map-of-eqvt*[*eqvt*]:
$\pi \cdot$ *map-of* l = *map-of* $(\pi \cdot l)$
⟨*proof*⟩

lemma *concat-eqvt*[*eqvt*]: $\pi \cdot$ *concat* l = *concat* $(\pi \cdot l)$
⟨*proof*⟩

lemma *tranclp-eqvt*[*eqvt*]: $\pi \cdot$ *tranclp* $P\ v_1\ v_2$ = *tranclp* $(\pi \cdot P)\ (\pi \cdot v_1)\ (\pi \cdot v_2)$
⟨*proof*⟩

lemma *rtranclp-eqvt*[*eqvt*]: $\pi \cdot$ *rtranclp* $P\ v_1\ v_2$ = *rtranclp* $(\pi \cdot P)\ (\pi \cdot v_1)\ (\pi \cdot v_2)$
⟨*proof*⟩

lemma *Set-filter-eqvt*[*eqvt*]: $\pi \cdot$ *Set.filter* $P\ S$ = *Set.filter* $(\pi \cdot P)\ (\pi \cdot S)$

⟨*proof*⟩

lemma *Sigma-eqvt'*[*eqvt*]: $\pi \cdot Sigma = Sigma$
⟨*proof*⟩

lemma *override-on-eqvt*[*eqvt*]:
$\pi \cdot (override\text{-}on\ m1\ m2\ S) = override\text{-}on\ (\pi \cdot m1)\ (\pi \cdot m2)\ (\pi \cdot S)$
⟨*proof*⟩

lemma *card-eqvt*[*eqvt*]:
$\pi \cdot (card\ S) = card\ (\pi \cdot S)$
⟨*proof*⟩

lemma *Projl-permute*:
assumes a: $\exists\, y.\ f = Inl\ y$
shows $(p \cdot (Sum\text{-}Type.projl\ f)) = Sum\text{-}Type.projl\ (p \cdot f)$
⟨*proof*⟩

lemma *Projr-permute*:
assumes a: $\exists\, y.\ f = Inr\ y$
shows $(p \cdot (Sum\text{-}Type.projr\ f)) = Sum\text{-}Type.projr\ (p \cdot f)$
⟨*proof*⟩

2.4.5 Freshness lemmas

lemma *fresh-list-elem*:
assumes $a \sharp \Gamma$
and $e \in set\ \Gamma$
shows $a \sharp e$
⟨*proof*⟩

lemma *set-not-fresh*:
$x \in set\ L \Longrightarrow \neg(atom\ x \sharp L)$
⟨*proof*⟩

lemma *pure-fresh-star*[*simp*]: $a \sharp{*}\ (x :: 'a :: pure)$
⟨*proof*⟩

lemma *supp-set-mem*: $x \in set\ L \Longrightarrow supp\ x \subseteq supp\ L$
⟨*proof*⟩

lemma *set-supp-mono*: $set\ L \subseteq set\ L2 \Longrightarrow supp\ L \subseteq supp\ L2$
⟨*proof*⟩

lemma *fresh-star-at-base*:
fixes $x :: 'a :: at\text{-}base$
shows $S \sharp{*}\ x \longleftrightarrow atom\ x \notin S$

$\langle proof \rangle$

2.4.6 Freshness and support for subsets of variables

lemma *supp-mono*: *finite* $(B::'a::fs\ set) \Longrightarrow A \subseteq B \Longrightarrow supp\ A \subseteq supp\ B$
$\langle proof \rangle$

lemma *fresh-subset*:
finite $B \Longrightarrow x \sharp (B :: 'a::at\text{-}base\ set) \Longrightarrow A \subseteq B \Longrightarrow x \sharp A$
$\langle proof \rangle$

lemma *fresh-star-subset*:
finite $B \Longrightarrow x \sharp{*} (B :: 'a::at\text{-}base\ set) \Longrightarrow A \subseteq B \Longrightarrow x \sharp{*} A$
$\langle proof \rangle$

lemma *fresh-star-set-subset*:
$x \sharp{*} (B :: 'a::at\text{-}base\ list) \Longrightarrow set\ A \subseteq set\ B \Longrightarrow x \sharp{*} A$
$\langle proof \rangle$

2.4.7 The set of free variables of an expression

definition $fv :: 'a::pt \Rightarrow 'b::at\text{-}base\ set$
where $fv\ e = \{v.\ atom\ v \in supp\ e\}$

lemma *fv-eqvt*[*simp*,*eqvt*]: $\pi \cdot (fv\ e) = fv\ (\pi \cdot e)$
$\langle proof \rangle$

lemma *fv-Nil*[*simp*]: $fv\ [] = \{\}$
$\langle proof \rangle$
lemma *fv-Cons*[*simp*]: $fv\ (x\ \#\ xs) = fv\ x \cup fv\ xs$
$\langle proof \rangle$
lemma *fv-Pair*[*simp*]: $fv\ (x,\ y) = fv\ x \cup fv\ y$
$\langle proof \rangle$
lemma *fv-append*[*simp*]: $fv\ (x\ @\ y) = fv\ x \cup fv\ y$
$\langle proof \rangle$
lemma *fv-at-base*[*simp*]: $fv\ a = \{a::'a::at\text{-}base\}$
$\langle proof \rangle$
lemma *fv-pure*[*simp*]: $fv\ (a::'a::pure) = \{\}$
$\langle proof \rangle$

lemma *fv-set-at-base*[*simp*]: $fv\ (l :: ('a :: at\text{-}base)\ list) = set\ l$
$\langle proof \rangle$

lemma *flip-not-fv*: $a \notin fv\ x \Longrightarrow b \notin fv\ x \Longrightarrow (a \leftrightarrow b) \cdot x = x$
$\langle proof \rangle$

lemma *fv-not-fresh*: $atom\ x \sharp e \longleftrightarrow x \notin fv\ e$
$\langle proof \rangle$

lemma *fresh-fv*: *finite* $(fv\ e :: 'a\ set) \Longrightarrow$ *atom* $(x :: ('a::at\text{-}base))\ \sharp\ (fv\ e :: 'a\ set) \longleftrightarrow atom\ x\ \sharp\ e$
⟨*proof*⟩

lemma *finite-fv*[*simp*]: *finite* $(fv\ (e::'a::fs) :: ('b::at\text{-}base)\ set)$
⟨*proof*⟩

definition *fv-list* :: $'a::fs \Rightarrow 'b::at\text{-}base\ list$
where *fv-list* $e = (SOME\ l.\ set\ l = fv\ e)$

lemma *set-fv-list*[*simp*]: $set\ (fv\text{-}list\ e) = (fv\ e :: ('b::at\text{-}base)\ set)$
⟨*proof*⟩

lemma *fresh-fv-list*[*simp*]:
$a\ \sharp\ (fv\text{-}list\ e :: 'b::at\text{-}base\ list) \longleftrightarrow a\ \sharp\ (fv\ e :: 'b::at\text{-}base\ set)$
⟨*proof*⟩

2.4.8 Other useful lemmas

lemma *pure-permute-id*: $permute\ p = (\lambda\ x.\ (x::'a::pure))$
⟨*proof*⟩

lemma *supp-set-elem-finite*:
assumes *finite* S
and $(m::'a::fs) \in S$
and $y \in supp\ m$
shows $y \in supp\ S$
⟨*proof*⟩

lemmas *fresh-star-Cons* = *fresh-star-list*(*2*)

lemma *mem-permute-set*:
shows $x \in p \bullet S \longleftrightarrow (-\ p \bullet x) \in S$
⟨*proof*⟩

lemma *flip-set-both-not-in*:
assumes $x \notin S$ **and** $x' \notin S$
shows $((x' \leftrightarrow x) \bullet S) = S$
⟨*proof*⟩

lemma *inj-atom*: *inj atom* ⟨*proof*⟩

lemmas *image-Int*[*OF inj-atom*, *simp*]

lemma *eqvt-uncurry*: *eqvt* $f \Longrightarrow$ *eqvt* (*case-prod* f)
⟨*proof*⟩

lemma *supp-fun-app-eqvt2*:
assumes *a*: *eqvt* f

shows $supp\ (f\ x\ y) \subseteq supp\ x \cup supp\ y$
⟨*proof*⟩

lemma *supp-fun-app-eqvt3*:
assumes *a*: *eqvt f*
shows $supp\ (f\ x\ y\ z) \subseteq supp\ x \cup supp\ y \cup supp\ z$
⟨*proof*⟩

lemma *permute-0*[*simp*]: $permute\ 0 = (\lambda\ x.\ x)$
⟨*proof*⟩
lemma *permute-comp*[*simp*]: $permute\ x \circ permute\ y = permute\ (x + y)$ ⟨*proof*⟩

lemma *map-permute*: $map\ (permute\ p) = permute\ p$
⟨*proof*⟩

lemma *fresh-star-restrictA*[*intro*]: $a \sharp{*}\ \Gamma \Longrightarrow a \sharp{*}\ AList.restrict\ V\ \Gamma$
⟨*proof*⟩

lemma *Abs-lst-Nil-eq*[*simp*]: $[[]]lst.\ (x{::}'a{::}fs) = [xs]lst.\ x' \longleftrightarrow (([],x) = (xs,\ x'))$
⟨*proof*⟩

lemma *Abs-lst-Nil-eq2*[*simp*]: $[xs]lst.\ (x{::}'a{::}fs) = [[]]lst.\ x' \longleftrightarrow ((xs,x) = ([],\ x'))$
⟨*proof*⟩

end

2.5 AList-Utils-Nominal

theory *AList−Utils−Nominal*
imports *AList−Utils Nominal−Utils*
begin

2.5.1 Freshness lemmas related to associative lists

lemma *domA-not-fresh*:
$x \in domA\ \Gamma \Longrightarrow \neg(atom\ x \sharp\ \Gamma)$
⟨*proof*⟩

lemma *fresh-delete*:
assumes $a \sharp\ \Gamma$
shows $a \sharp\ delete\ v\ \Gamma$
⟨*proof*⟩

lemma *fresh-star-delete*:
assumes $S \sharp* \Gamma$
shows $S \sharp*$ *delete* $v\ \Gamma$
⟨*proof*⟩

lemma *fv-delete-subset*:
fv (*delete* $v\ \Gamma$) $\subseteq$ *fv* Γ
⟨*proof*⟩

lemma *fresh-heap-expr*:
assumes $a \sharp \Gamma$
and $(x,e) \in$ *set* Γ
shows $a \sharp e$
⟨*proof*⟩

lemma *fresh-heap-expr'*:
assumes $a \sharp \Gamma$
and $e \in$ *snd* ‘ *set* Γ
shows $a \sharp e$
⟨*proof*⟩

lemma *fresh-star-heap-expr'*:
assumes $S \sharp* \Gamma$
and $e \in$ *snd* ‘ *set* Γ
shows $S \sharp* e$
⟨*proof*⟩

lemma *fresh-map-of*:
assumes $x \in$ *domA* Γ
assumes $a \sharp \Gamma$
shows $a \sharp$ *the* (*map-of* Γ x)
⟨*proof*⟩

lemma *fresh-star-map-of*:
assumes $x \in$ *domA* Γ
assumes $a \sharp* \Gamma$
shows $a \sharp*$ *the* (*map-of* Γ x)
⟨*proof*⟩

lemma *domA-fv-subset*: *domA* $\Gamma \subseteq$ *fv* Γ
⟨*proof*⟩

lemma *map-of-fv-subset*: $x \in$ *domA* $\Gamma \Longrightarrow$ *fv* (*the* (*map-of* Γ x)) $\subseteq$ *fv* Γ
⟨*proof*⟩

lemma *map-of-Some-fv-subset*: *map-of* Γ x = *Some* $e \Longrightarrow$ *fv* $e \subseteq$ *fv* Γ
⟨*proof*⟩

2.5.2 Equivalariance lemmas

lemma *domA*[*eqvt*]:
$\pi \bullet domA\ \Gamma = domA\ (\pi \bullet \Gamma)$
⟨*proof*⟩

lemma *mapCollect*[*eqvt*]:
$\pi \bullet mapCollect\ f\ m = mapCollect\ (\pi \bullet f)\ (\pi \bullet m)$
⟨*proof*⟩

2.5.3 Freshness and distinctness

lemma *fresh-distinct*:
assumes $atom\ `\ S\ \sharp{*}\ \Gamma$
shows $S \cap domA\ \Gamma = \{\}$
⟨*proof*⟩

lemma *fresh-distinct-list*:
assumes $atom\ `\ S\ \sharp{*}\ l$
shows $S \cap set\ l = \{\}$
⟨*proof*⟩

lemma *fresh-distinct-fv*:
assumes $atom\ `\ S\ \sharp{*}\ l$
shows $S \cap fv\ l = \{\}$
⟨*proof*⟩

2.5.4 Pure codomains

lemma *domA-fv-pure*:
fixes $\Gamma :: ('a::at\text{-}base \times 'b::pure)\ list$
shows $fv\ \Gamma = domA\ \Gamma$
⟨*proof*⟩

lemma *domA-fresh-pure*:
fixes $\Gamma :: ('a::at\text{-}base \times 'b::pure)\ list$
shows $x \in domA\ \Gamma \longleftrightarrow \neg(atom\ x\ \sharp\ \Gamma)$
⟨*proof*⟩

end

2.6 HOLCF-Utils

theory *HOLCF−Utils*
imports *HOLCF Pointwise*
begin

default-sort *type*

lemmas *cont-fun*[*simp*]

lemmas *cont2cont-fun*[*simp*]

lemma *cont-compose2*:
  assumes $\bigwedge$ *y*. *cont* (λ *x*. *c x y*)
  assumes $\bigwedge$ *x*. *cont* (λ *y*. *c x y*)
  assumes *cont f*
  assumes *cont g*
  shows *cont* (λ*x*. *c* (*f x*) (*g x*))
  ⟨*proof*⟩

lemma *pointwise-adm*:
  fixes *P* :: *'a::pcpo* ⇒ *'b::pcpo* ⇒ *bool*
  assumes *adm* (λ *x*. *P* (*fst x*) (*snd x*))
  shows *adm* (λ *m*. *pointwise P* (*fst m*) (*snd m*))
⟨*proof*⟩

lemma *cfun-beta-Pair*:
  assumes *cont* (λ *p*. *f* (*fst p*) (*snd p*))
  shows *csplit*·(Λ *a b* . *f a b*)·(*x*, *y*) = *f x y*
  ⟨*proof*⟩

lemma *fun-upd-mono*:
  $\varrho 1 \sqsubseteq \varrho 2 \Longrightarrow v1 \sqsubseteq v2 \Longrightarrow \varrho 1(x := v1) \sqsubseteq \varrho 2(x := v2)$
  ⟨*proof*⟩

lemma *fun-upd-cont*[*simp*,*cont2cont*]:
  assumes *cont f* **and** *cont h*
  shows *cont* (λ *x*. (*f x*)(*v* := *h x*) :: *'a* ⇒ *'b::pcpo*)
  ⟨*proof*⟩

lemma *fun-upd-belowI*:
  assumes $\bigwedge$ *z* . $z \neq x \Longrightarrow \varrho\ z \sqsubseteq \varrho'\ z$
  assumes $y \sqsubseteq \varrho'\ x$
  shows $\varrho(x := y) \sqsubseteq \varrho'$
  ⟨*proof*⟩

lemma *cont-if-else-above*:
  assumes *cont f*
  assumes *cont g*
  assumes $\bigwedge$ *x*. $f\ x \sqsubseteq g\ x$
  assumes $\bigwedge$ *x y*. $x \sqsubseteq y \Longrightarrow P\ y \Longrightarrow P\ x$
  assumes *adm P*
  shows *cont* (λ*x*. *if P x then f x else g x*) (**is** *cont ?I*)
⟨*proof*⟩

fun *up2option* :: *'a::cpo*$_\bot$ ⇒ *'a option*
  where *up2option Ibottom* = *None*

| $up2option\ (Iup\ a) = Some\ a$

lemma *up2option-simps*[*simp*]:
$up2option\ \bot = None$
$up2option\ (up{\cdot}x) = Some\ x$
⟨*proof*⟩

fun $option2up :: 'a\ option \Rightarrow 'a::cpo_{\bot}$
where $option2up\ None = \bot$
| $option2up\ (Some\ a) = up{\cdot}a$

lemma *option2up-up2option*[*simp*]:
$option2up\ (up2option\ x) = x$
⟨*proof*⟩
lemma *up2option-option2up*[*simp*]:
$up2option\ (option2up\ x) = x$
⟨*proof*⟩

lemma *adm-subst2*: $cont\ f \Longrightarrow cont\ g \Longrightarrow adm\ (\lambda x.\ f\ (fst\ x) = g\ (snd\ x))$
⟨*proof*⟩

2.6.1 Composition of fun and cfun

lemma *cont2cont-comp* [*simp*, *cont2cont*]:
assumes $cont\ f$
assumes $\bigwedge\ x.\ cont\ (f\ x)$
assumes $cont\ g$
shows $cont\ (\lambda\ x.\ (f\ x) \circ (g\ x))$
⟨*proof*⟩

definition $cfun\text{-}comp :: ('a::pcpo \rightarrow 'b::pcpo) \rightarrow ('c::type \Rightarrow 'a) \rightarrow ('c::type \Rightarrow 'b)$
where $cfun\text{-}comp = (\Lambda\ f\ \varrho.\ (\lambda\ x.\ f{\cdot}x) \circ \varrho)$

lemma [*simp*]: $cfun\text{-}comp{\cdot}f{\cdot}(\varrho(x := v)) = (cfun\text{-}comp{\cdot}f{\cdot}\varrho)(x := f{\cdot}v)$
⟨*proof*⟩

lemma *cfun-comp-app*[*simp*]: $(cfun\text{-}comp{\cdot}f{\cdot}\varrho)\ x = f{\cdot}(\varrho\ x)$
⟨*proof*⟩

lemma *fix-eq-fix*:
$f{\cdot}(fix{\cdot}g) \sqsubseteq fix{\cdot}g \Longrightarrow g{\cdot}(fix{\cdot}f) \sqsubseteq fix{\cdot}f \Longrightarrow fix{\cdot}f = fix{\cdot}g$
⟨*proof*⟩

2.6.2 Additional transitivity rules

These collect side-conditions of the form *cont f*, so the usual way to discharge them is to write *by this* (*intro cont2cont*)+ at the end.

lemma *below-trans-cong*[*trans*]:

$a \sqsubseteq f\, x \Longrightarrow x \sqsubseteq y \Longrightarrow cont\ f \Longrightarrow a \sqsubseteq f\, y$
⟨*proof*⟩

lemma *not-bot-below-trans*[*trans*]:
$a \neq \bot \Longrightarrow a \sqsubseteq b \Longrightarrow b \neq \bot$
⟨*proof*⟩

lemma *not-bot-below-trans-cong*[*trans*]:
$f\, a \neq \bot \Longrightarrow a \sqsubseteq b \Longrightarrow cont\ f \Longrightarrow f\, b \neq \bot$
⟨*proof*⟩

end

2.7 HOLCF-Meet

theory *HOLCF−Meet*
imports *HOLCF*
begin

This theory defines the ⊓ operator on HOLCF domains, and introduces a type class for domains where all finite meets exist.

2.7.1 Towards meets: Lower bounds

context *po*
begin
definition *is-lb* :: *'a set* ⇒ *'a* ⇒ *bool* (**infix** ‹>|› *55*) **where**
$S >|\ x \longleftrightarrow (\forall y{\in}S.\ x \sqsubseteq y)$

lemma *is-lbI*: $(!!x.\ x \in S ==> l \sqsubseteq x) ==> S >|\ l$
⟨*proof*⟩

lemma *is-lbD*: $[|S >|\ l;\ x \in S|] ==> l \sqsubseteq x$
⟨*proof*⟩

lemma *is-lb-empty* [*simp*]: $\{\} >|\ l$
⟨*proof*⟩

lemma *is-lb-insert* [*simp*]: $(insert\ x\ A) >|\ y = (y \sqsubseteq x \wedge A >|\ y)$
⟨*proof*⟩

lemma *is-lb-downward*: $[|S >|\ l;\ y \sqsubseteq l|] ==> S >|\ y$
⟨*proof*⟩

2.7.2 Greatest lower bounds

definition *is-glb* :: *'a set* ⇒ *'a* ⇒ *bool* (**infix** ‹>>|› *55*) **where**
$S >>|\ x \longleftrightarrow S >|\ x \wedge (\forall u.\ S >|\ u --> u \sqsubseteq x)$

definition *glb* :: *'a set* ⇒ *'a* (‹⨅ -› [60]60) **where**
glb S = (*THE x. S* >>| *x*)

Access to the definition as inference rule

lemma *is-glbD1*: *S* >>| *x* ==> *S* >| *x*
⟨*proof*⟩

lemma *is-glbD2*: [|*S* >>| *x*; *S* >| *u*|] ==> *u* ⊑ *x*
⟨*proof*⟩

lemma (**in** *po*) *is-glbI*: [|*S* >| *x*; !!*u*. *S* >| *u* ==> *u* ⊑ *x*|] ==> *S* >>| *x*
⟨*proof*⟩

lemma *is-glb-above-iff*: *S* >>| *x* ==> *u* ⊑ *x* ⟷ *S* >| *u*
⟨*proof*⟩

glbs are unique

lemma *is-glb-unique*: [|*S* >>| *x*; *S* >>| *y*|] ==> *x* = *y*
⟨*proof*⟩

technical lemmas about *glb* and (>>|)

lemma *is-glb-glb*: *M* >>| *x* ==> *M* >>| *glb M*
⟨*proof*⟩

lemma *glb-eqI*: *M* >>| *l* ==> *glb M* = *l*
⟨*proof*⟩

lemma *is-glb-singleton*: {*x*} >>| *x*
⟨*proof*⟩

lemma *glb-singleton* [*simp*]: *glb* {*x*} = *x*
⟨*proof*⟩

lemma *is-glb-bin*: *x* ⊑ *y* ==> {*x*, *y*} >>| *x*
⟨*proof*⟩

lemma *glb-bin*: *x* ⊑ *y* ==> *glb* {*x*, *y*} = *x*
⟨*proof*⟩

lemma *is-glb-maximal*: [|*S* >| *x*; *x* ∈ *S*|] ==> *S* >>| *x*
⟨*proof*⟩

lemma *glb-maximal*: [|*S* >| *x*; *x* ∈ *S*|] ==> *glb S* = *x*
⟨*proof*⟩

lemma *glb-above*: *S* >>| *z* ⟹ *x* ⊑ *glb S* ⟷ *S* >| *x*
⟨*proof*⟩

end

lemma (**in** *cpo*) *Meet-insert*: $S >>| \; l \Longrightarrow \{x,\ l\} >>| \; l2 \Longrightarrow insert\ x\ S >>| \; l2$
⟨*proof*⟩

Binary, hence finite meets.

class *Finite-Meet-cpo* = *cpo* +
assumes *binary-meet-exists*: $\exists\ l.\ l \sqsubseteq x \wedge l \sqsubseteq y \wedge (\forall\ z.\ z \sqsubseteq x \longrightarrow z \sqsubseteq y \longrightarrow z \sqsubseteq l)$
begin

lemma *binary-meet-exists'*: $\exists\, l.\ \{x,\ y\} >>| \; l$
⟨*proof*⟩

lemma *finite-meet-exists*:
assumes $S \neq \{\}$
and *finite S*
shows $\exists\, x.\ S >>| \; x$
⟨*proof*⟩
end

definition *meet* :: $'a::cpo \Rightarrow 'a \Rightarrow 'a$ (**infix** ‹⊓› *80*) **where**
$x \sqcap y = (if\ \exists\ z.\ \{x,\ y\} >>| \; z\ then\ glb\ \{x,\ y\}\ else\ x)$

lemma *meet-def'*: $(x::'a::Finite\text{-}Meet\text{-}cpo) \sqcap y = glb\ \{x,\ y\}$
⟨*proof*⟩

lemma *meet-comm*: $(x::'a::Finite\text{-}Meet\text{-}cpo) \sqcap y = y \sqcap x$ ⟨*proof*⟩

lemma *meet-bot1*[*simp*]:
fixes y :: $'a$ :: {*Finite-Meet-cpo*,*pcpo*}
shows $(\bot \sqcap y) = \bot$ ⟨*proof*⟩
lemma *meet-bot2*[*simp*]:
fixes x :: $'a$ :: {*Finite-Meet-cpo*,*pcpo*}
shows $(x \sqcap \bot) = \bot$ ⟨*proof*⟩

lemma *meet-below1*[*intro*]:
fixes $x\ y$ :: $'a$ :: *Finite-Meet-cpo*
assumes $x \sqsubseteq z$
shows $(x \sqcap y) \sqsubseteq z$ ⟨*proof*⟩
lemma *meet-below2*[*intro*]:
fixes $x\ y$ :: $'a$ :: *Finite-Meet-cpo*
assumes $y \sqsubseteq z$
shows $(x \sqcap y) \sqsubseteq z$ ⟨*proof*⟩

lemma *meet-above-iff*:
fixes $x\ y\ z$ :: $'a$ :: *Finite-Meet-cpo*
shows $z \sqsubseteq x \sqcap y \longleftrightarrow z \sqsubseteq x \wedge z \sqsubseteq y$
⟨*proof*⟩

lemma *below-meet*[*simp*]:
  fixes $x\ y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
  assumes $x \sqsubseteq z$
  shows $(x \sqcap z) = x$ ⟨*proof*⟩

lemma *below-meet2*[*simp*]:
  fixes $x\ y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
  assumes $z \sqsubseteq x$
  shows $(x \sqcap z) = z$ ⟨*proof*⟩

lemma *meet-aboveI*:
  fixes $x\ y\ z :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
  shows $z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq x \sqcap y$ ⟨*proof*⟩

lemma *is-meetI*:
  fixes $x\ y\ z :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
  assumes $z \sqsubseteq x$
  assumes $z \sqsubseteq y$
  assumes $\bigwedge a.\ [\![\ a \sqsubseteq x\ ;\ a \sqsubseteq y\ ]\!] \Longrightarrow a \sqsubseteq z$
  shows $x \sqcap y = z$
⟨*proof*⟩

lemma *meet-assoc*[*simp*]: $((x{::}'a{::}\mathit{Finite\text{-}Meet\text{-}cpo}) \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
  ⟨*proof*⟩

lemma *meet-self*[*simp*]: $r \sqcap r = (r{::}'a{::}\mathit{Finite\text{-}Meet\text{-}cpo})$
  ⟨*proof*⟩

lemma [*simp*]: $(r{::}'a{::}\mathit{Finite\text{-}Meet\text{-}cpo}) \sqcap (r \sqcap x) = r \sqcap x$
  ⟨*proof*⟩

lemma *meet-monofun1*:
  fixes $y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
  shows $\mathit{monofun}\ (\lambda x.\ (x \sqcap y))$
  ⟨*proof*⟩

lemma *chain-meet1*:
  fixes $y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
  assumes $\mathit{chain}\ Y$
  shows $\mathit{chain}\ (\lambda\ i.\ Y\ i \sqcap y)$
⟨*proof*⟩

class $\mathit{cont\text{-}binary\text{-}meet} = \mathit{Finite\text{-}Meet\text{-}cpo}\ +$
  assumes *meet-cont′*: $\mathit{chain}\ Y \Longrightarrow (\bigsqcup\ i.\ Y\ i) \sqcap y = (\bigsqcup\ i.\ Y\ i \sqcap y)$

lemma *meet-cont1*:
  fixes $y :: 'a :: \mathit{cont\text{-}binary\text{-}meet}$
  shows $\mathit{cont}\ (\lambda x.\ (x \sqcap y))$
  ⟨*proof*⟩

lemma *meet-cont2*:
  fixes x :: $'a$ :: *cont-binary-meet*
  shows *cont* $(\lambda y.\ (x \sqcap y))$ ⟨*proof*⟩

lemma *meet-cont*[*cont2cont*,*simp*]:*cont* $f \Longrightarrow$ *cont* $g \Longrightarrow$ *cont* $(\lambda x.\ (f\,x \sqcap (g\ x{::}'a{::}cont\text{-}binary\text{-}meet)))$
  ⟨*proof*⟩

end

2.8 Nominal-HOLCF

theory *Nominal−HOLCF*
imports
  Nominal−Utils HOLCF−Utils
begin

2.8.1 Type class of continuous permutations and variations thereof

class *cont-pt* =
  cpo +
  pt +
  assumes *perm-cont*: $\bigwedge p.$ *cont* $((permute\ p) :: 'a{::}\{cpo,pt\} \Rightarrow 'a)$

class *discr-pt* =
  discrete-cpo +
  pt

class *pcpo-pt* =
  cont-pt +
  pcpo

instance *pcpo-pt* ⊆ *cont-pt*
 ⟨*proof*⟩

instance *discr-pt* ⊆ *cont-pt*
 ⟨*proof*⟩

lemma (**in** *cont-pt*) *perm-cont-simp*[*simp*]: $\pi \cdot x \sqsubseteq \pi \cdot y \longleftrightarrow x \sqsubseteq y$
  ⟨*proof*⟩

lemma (**in** *cont-pt*) *perm-below-to-right*: $\pi \cdot x \sqsubseteq y \longleftrightarrow x \sqsubseteq -\ \pi \cdot\ y$
  ⟨*proof*⟩

lemma *perm-is-ub-simp*[*simp*]: $\pi \cdot S <| \pi \cdot (x{::}'a{::}cont\text{-}pt) \longleftrightarrow S <| x$
  ⟨*proof*⟩

lemma *perm-is-ub-eqvt*[*simp*,*eqvt*]: $S <| (x{::}'a{::}cont\text{-}pt) \Longrightarrow \pi \cdot S <| \pi \cdot x$
  ⟨*proof*⟩

lemma *perm-is-lub-simp*[*simp*]: $\pi \cdot S <<| \pi \cdot (x::'a::cont\text{-}pt) \longleftrightarrow S <<| x$
⟨*proof*⟩

lemma *perm-is-lub-eqvt*[*simp*,*eqvt*]: $S <<| (x::'a::cont\text{-}pt) ==> \pi \cdot S <<| \pi \cdot x$
⟨*proof*⟩

lemmas *perm-cont2cont*[*simp*,*cont2cont*] = *cont-compose*[*OF perm-cont*]

lemma *perm-still-cont*: $cont\ (\pi \cdot f) = cont\ (f :: ('a :: cont\text{-}pt) \Rightarrow ('b :: cont\text{-}pt))$
⟨*proof*⟩

lemma *perm-bottom*[*simp*,*eqvt*]: $\pi \cdot \bot = (\bot::'a::\{cont\text{-}pt,pcpo\})$
⟨*proof*⟩

lemma *bot-supp*[*simp*]: $supp\ (\bot :: 'a :: pcpo\text{-}pt) = \{\}$
⟨*proof*⟩

lemma *bot-fresh*[*simp*]: $a \sharp (\bot :: 'a :: pcpo\text{-}pt)$
⟨*proof*⟩

lemma *bot-fresh-star*[*simp*]: $a \sharp* (\bot :: 'a :: pcpo\text{-}pt)$
⟨*proof*⟩

lemma *below-eqvt* [*eqvt*]:
$\pi \cdot (x \sqsubseteq y) = (\pi \cdot x \sqsubseteq \pi \cdot (y::'a::cont\text{-}pt))$ ⟨*proof*⟩

lemma *lub-eqvt*[*simp*]:
$(\exists\ z.\ S <<| (z::'a::\{cont\text{-}pt\})) \Longrightarrow \pi \cdot lub\ S = lub\ (\pi \cdot S)$
⟨*proof*⟩

lemma *chain-eqvt*[*eqvt*]:
fixes $F :: nat \Rightarrow 'a::cont\text{-}pt$
shows $chain\ F \Longrightarrow chain\ (\pi \cdot F)$
⟨*proof*⟩

2.8.2 Instance for *cfun*

instantiation *cfun* :: (*cont-pt*, *cont-pt*) *pt*
begin
definition $p \cdot (f :: 'a \rightarrow 'b) = (\Lambda\ x.\ p \cdot (f \cdot (-\ p \cdot x)))$

instance
⟨*proof*⟩
end

lemma *permute-cfun-eq*: $permute\ p = (\lambda\ f.\ (Abs\text{-}cfun\ (permute\ p))\ oo\ f\ oo\ (Abs\text{-}cfun\ (permute\ (-p))))$
⟨*proof*⟩

lemma *Cfun-app-eqvt*[*eqvt*]:
$\pi \cdot (f \cdot x) = (\pi \cdot f) \cdot (\pi \cdot x)$
⟨*proof*⟩

lemma *permute-Lam*: $\mathit{cont}\ f \Longrightarrow p \cdot (\Lambda\ x.\ f\ x) = (\Lambda\ x.\ (p \cdot f)\ x)$
⟨*proof*⟩

lemma *Abs-cfun-eqvt*: $\mathit{cont}\ f \Longrightarrow (p \cdot \mathit{Abs\text{-}cfun})\ f = \mathit{Abs\text{-}cfun}\ f$
⟨*proof*⟩

lemma *cfun-eqvtI*: $(\bigwedge x.\ p \cdot (f \cdot x) = f' \cdot (p \cdot x)) \Longrightarrow p \cdot f = f'$
⟨*proof*⟩

lemma *ID-eqvt*[*eqvt*]: $\pi \cdot \mathit{ID} = \mathit{ID}$
⟨*proof*⟩

instance *cfun* :: (*cont-pt*, *cont-pt*) *cont-pt*
⟨*proof*⟩

instance *cfun* :: ({*pure*,*cont-pt*}, {*pure*,*cont-pt*}) *pure*
⟨*proof*⟩

instance *cfun* :: (*cont-pt*, *pcpo-pt*) *pcpo-pt*
⟨*proof*⟩

2.8.3 Instance for *fun*

lemma *permute-fun-eq*: $\mathit{permute}\ p = (\lambda\ f.\ (\mathit{permute}\ p) \circ f \circ (\mathit{permute}\ (-p)))$
⟨*proof*⟩

instance *fun* :: (*pt*, *cont-pt*) *cont-pt*
⟨*proof*⟩

lemma *fix-eqvt*[*eqvt*]:
$\pi \cdot \mathit{fix} = (\mathit{fix} :: ('a \to 'a) \to 'a::\{\mathit{cont\text{-}pt},\mathit{pcpo}\})$
⟨*proof*⟩

2.8.4 Instance for *u*

instantiation *u* :: (*cont-pt*) *pt*
begin
definition $p \cdot (x :: 'a\ u) = \mathit{fup}\cdot(\Lambda\ x.\ \mathit{up}\cdot(p \cdot x))\cdot x$
instance
⟨*proof*⟩
end

instance *u* :: (*cont-pt*) *cont-pt*
⟨*proof*⟩

instance *u* :: (*cont-pt*) *pcpo-pt* ⟨*proof*⟩

class *pure-cont-pt* = *pure* + *cont-pt*

instance *u* :: (*pure-cont-pt*) *pure*
⟨*proof*⟩

lemma *up-eqvt*[*eqvt*]: $\pi \cdot up = up$
⟨*proof*⟩

lemma *fup-eqvt*[*eqvt*]: $\pi \cdot fup = fup$
⟨*proof*⟩

2.8.5 Instance for *lift*

instantiation *lift* :: (*pt*) *pt*
begin
definition $p \cdot (x :: 'a\ lift) = case\text{-}lift\ \bot\ (\lambda\ x.\ Def\ (p \cdot x))\ x$
instance
⟨*proof*⟩
end

instance *lift* :: (*pt*) *cont-pt*
⟨*proof*⟩

instance *lift* :: (*pt*) *pcpo-pt* ⟨*proof*⟩

instance *lift* :: (*pure*) *pure*
⟨*proof*⟩

lemma *Def-eqvt*[*eqvt*]: $\pi \cdot (Def\ x) = Def\ (\pi \cdot x)$
⟨*proof*⟩

lemma *case-lift-eqvt*[*eqvt*]: $\pi \cdot case\text{-}lift\ d\ f\ x = case\text{-}lift\ (\pi \cdot d)\ (\pi \cdot f)\ (\pi \cdot x)$
⟨*proof*⟩

2.8.6 Instance for *prod*

instance *prod* :: (*cont-pt*, *cont-pt*) *cont-pt*
⟨*proof*⟩

end

2.9 Env

theory *Env*
imports *Main HOLCF−Join−Classes*

begin

default-sort *type*

Our type for environments is a function with a pcpo as the co-domain; this theory collects related definitions.

2.9.1 The domain of a pcpo-valued function

definition $edom :: ('key \Rightarrow 'value::pcpo) \Rightarrow 'key\ set$
where $edom\ m = \{x.\ m\ x \neq \bot\}$

lemma *bot-edom*[*simp*]: $edom\ \bot = \{\}$ ⟨*proof*⟩

lemma *bot-edom2*[*simp*]: $edom\ (\lambda\text{-}\ .\ \bot) = \{\}$ ⟨*proof*⟩

lemma *edomIff*: $(a \in edom\ m) = (m\ a \neq \bot)$ ⟨*proof*⟩
lemma *edom-iff2*: $(m\ a = \bot) \longleftrightarrow (a \notin edom\ m)$ ⟨*proof*⟩

lemma *edom-empty-iff-bot*: $edom\ m = \{\} \longleftrightarrow m = \bot$
⟨*proof*⟩

lemma *lookup-not-edom*: $x \notin edom\ m \Longrightarrow m\ x = \bot$ ⟨*proof*⟩

lemma *lookup-edom*[*simp*]: $m\ x \neq \bot \Longrightarrow x \in edom\ m$ ⟨*proof*⟩

lemma *edom-mono*: $x \sqsubseteq y \Longrightarrow edom\ x \subseteq edom\ y$
⟨*proof*⟩

lemma *edom-subset-adm*[*simp*]:
$adm\ (\lambda ae'.\ edom\ ae' \subseteq S)$
⟨*proof*⟩

2.9.2 Updates

lemma *edom-fun-upd-subset*: $edom\ (h\ (x := v)) \subseteq insert\ x\ (edom\ h)$
⟨*proof*⟩

declare *fun-upd-same*[*simp*] *fun-upd-other*[*simp*]

2.9.3 Restriction

definition $env\text{-}restr :: 'a\ set \Rightarrow ('a \Rightarrow 'b::pcpo) \Rightarrow ('a \Rightarrow 'b)$
where $env\text{-}restr\ S\ m = (\lambda\ x.\ if\ x \in S\ then\ m\ x\ else\ \bot)$

abbreviation *env-restr-rev* (**infixl** ‹*f*|‘› *110*)
where $env\text{-}restr\text{-}rev\ m\ S \equiv env\text{-}restr\ S\ m$

notation (*latex* **output**) *env-restr-rev* (‹-|_›)

lemma *env-restr-empty-iff*[*simp*]: $m\ f|^{\backprime}\ S = \bot \longleftrightarrow$ *edom* $m \cap S = \{\}$
⟨*proof*⟩
lemmas *env-restr-empty* = *iffD2*[*OF env-restr-empty-iff*, *simp*]

lemma *lookup-env-restr*[*simp*]: $x \in S \Longrightarrow (m\ f|^{\backprime}\ S)\ x = m\ x$
⟨*proof*⟩

lemma *lookup-env-restr-not-there*[*simp*]: $x \notin S \Longrightarrow$ (*env-restr* S m) $x = \bot$
⟨*proof*⟩

lemma *lookup-env-restr-eq*: $(m\ f|^{\backprime}\ S)\ x =$ (*if* $x \in S$ *then* $m\ x$ *else* $\bot$)
⟨*proof*⟩

lemma *env-restr-eqI*: $(\bigwedge x.\ x \in S \Longrightarrow m_1\ x = m_2\ x) \Longrightarrow m_1\ f|^{\backprime}\ S = m_2\ f|^{\backprime}\ S$
⟨*proof*⟩

lemma *env-restr-eqD*: $m_1\ f|^{\backprime}\ S = m_2\ f|^{\backprime}\ S \Longrightarrow x \in S \Longrightarrow m_1\ x = m_2\ x$
⟨*proof*⟩

lemma *env-restr-belowI*: $(\bigwedge x.\ x \in S \Longrightarrow m_1\ x \sqsubseteq m_2\ x) \Longrightarrow m_1\ f|^{\backprime}\ S \sqsubseteq m_2\ f|^{\backprime}\ S$
⟨*proof*⟩

lemma *env-restr-belowD*: $m_1\ f|^{\backprime}\ S \sqsubseteq m_2\ f|^{\backprime}\ S \Longrightarrow x \in S \Longrightarrow m_1\ x \sqsubseteq m_2\ x$
⟨*proof*⟩

lemma *env-restr-env-restr*[*simp*]:
$x\ f|^{\backprime}\ d2\ f|^{\backprime}\ d1 = x\ f|^{\backprime}\ (d1 \cap d2)$
⟨*proof*⟩

lemma *env-restr-env-restr-subset*:
$d1 \subseteq d2 \Longrightarrow x\ f|^{\backprime}\ d2\ f|^{\backprime}\ d1 = x\ f|^{\backprime}\ d1$
⟨*proof*⟩

lemma *env-restr-useless*: *edom* $m \subseteq S \Longrightarrow m\ f|^{\backprime}\ S = m$
⟨*proof*⟩

lemma *env-restr-UNIV*[*simp*]: $m\ f|^{\backprime}\ UNIV = m$
⟨*proof*⟩

lemma *env-restr-fun-upd*[*simp*]: $x \in S \Longrightarrow m1(x := v)\ f|^{\backprime}\ S = (m1\ f|^{\backprime}\ S)(x := v)$
⟨*proof*⟩

lemma *env-restr-fun-upd-other*[*simp*]: $x \notin S \Longrightarrow m1(x := v)\ f|^{\backprime}\ S = m1\ f|^{\backprime}\ S$
⟨*proof*⟩

lemma *env-restr-eq-subset*:
assumes $S \subseteq S'$

and $m1\ f|`\ S' = m2\ f|`\ S'$
shows $m1\ f|`\ S = m2\ f|`\ S$
$\langle proof \rangle$

lemma *env-restr-below-subset*:
assumes $S \subseteq S'$
and $m1\ f|`\ S' \sqsubseteq m2\ f|`\ S'$
shows $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
$\langle proof \rangle$

lemma *edom-env*[*simp*]:
$edom\ (m\ f|`\ S) = edom\ m \cap S$
$\langle proof \rangle$

lemma *env-restr-below-self*: $f\ f|`\ S \sqsubseteq f$
$\langle proof \rangle$

lemma *env-restr-below-trans*:
$m1\ f|`\ S1 \sqsubseteq m2\ f|`\ S1 \Longrightarrow m2\ f|`\ S2 \sqsubseteq m3\ f|`\ S2 \Longrightarrow m1\ f|`\ (S1 \cap S2) \sqsubseteq m3\ f|`\ (S1 \cap S2)$
$\langle proof \rangle$

lemma *env-restr-cont*: $cont\ (env\text{-}restr\ S)$
$\langle proof \rangle$

lemma *env-restr-mono*: $m1 \sqsubseteq m2 \Longrightarrow m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
$\langle proof \rangle$

lemma *env-restr-mono2*: $S2 \subseteq S1 \Longrightarrow m\ f|`\ S2 \sqsubseteq m\ f|`\ S1$
$\langle proof \rangle$

lemmas *cont-compose*[*OF env-restr-cont*, *cont2cont*, *simp*]

lemma *env-restr-cong*: $(\bigwedge x.\ edom\ m \subseteq S \cap S' \cup -S \cap -S') \Longrightarrow m\ f|`\ S = m\ f|`\ S'$
$\langle proof \rangle$

2.9.4 Deleting

definition *env-delete* :: $'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b::pcpo)$
where $env\text{-}delete\ x\ m = m(x := \bot)$

lemma *lookup-env-delete*[*simp*]:
$x' \neq x \Longrightarrow env\text{-}delete\ x\ m\ x' = m\ x'$
$\langle proof \rangle$

lemma *lookup-env-delete-None*[*simp*]:
$env\text{-}delete\ x\ m\ x = \bot$
$\langle proof \rangle$

lemma *edom-env-delete*[*simp*]:
$edom\ (env\text{-}delete\ x\ m) = edom\ m - \{x\}$
⟨*proof*⟩

lemma *edom-env-delete-subset*:
$edom\ (env\text{-}delete\ x\ m) \subseteq edom\ m$ ⟨*proof*⟩

lemma *env-delete-fun-upd*[*simp*]:
$env\text{-}delete\ x\ (m(x := v)) = env\text{-}delete\ x\ m$
⟨*proof*⟩

lemma *env-delete-fun-upd2*[*simp*]:
$(env\text{-}delete\ x\ m)(x := v) = m(x := v)$
⟨*proof*⟩

lemma *env-delete-fun-upd3*[*simp*]:
$x \neq y \Longrightarrow env\text{-}delete\ x\ (m(y := v)) = (env\text{-}delete\ x\ m)(y := v)$
⟨*proof*⟩

lemma *env-delete-noop*[*simp*]:
$x \notin edom\ m \Longrightarrow env\text{-}delete\ x\ m = m$
⟨*proof*⟩

lemma *fun-upd-env-delete*[*simp*]: $x \in edom\ \Gamma \Longrightarrow (env\text{-}delete\ x\ \Gamma)(x := \Gamma\ x) = \Gamma$
⟨*proof*⟩

lemma *env-restr-env-delete-other*[*simp*]: $x \notin S \Longrightarrow env\text{-}delete\ x\ m\ f|`\ S = m\ f|`\ S$
⟨*proof*⟩

lemma *env-delete-restr*: $env\text{-}delete\ x\ m = m\ f|`\ (-\{x\})$
⟨*proof*⟩

lemma *below-env-deleteI*: $f\ x = \bot \Longrightarrow f \sqsubseteq g \Longrightarrow f \sqsubseteq env\text{-}delete\ x\ g$
⟨*proof*⟩

lemma *env-delete-below-cong*[*intro*]:
assumes $x \neq v \Longrightarrow e1\ x \sqsubseteq e2\ x$
shows $env\text{-}delete\ v\ e1\ x \sqsubseteq env\text{-}delete\ v\ e2\ x$
⟨*proof*⟩

lemma *env-delete-env-restr-swap*:
$env\text{-}delete\ x\ (env\text{-}restr\ S\ e) = env\text{-}restr\ S\ (env\text{-}delete\ x\ e)$
⟨*proof*⟩

lemma *env-delete-mono*:
$m \sqsubseteq m' \Longrightarrow env\text{-}delete\ x\ m \sqsubseteq env\text{-}delete\ x\ m'$
⟨*proof*⟩

lemma *env-delete-below-arg*:

env-delete x m $\sqsubseteq$ *m*
⟨*proof*⟩

2.9.5 Merging of two functions

We'd like to have some nice syntax for *override-on*.

abbreviation *override-on-syn* (‹- ++- -› [*100*, *0*, *100*] *100*) **where** $f1 \mathrel{++_S} f2 \equiv$ *override-on f1 f2 S*

lemma *override-on-bot*[*simp*]:
$\bot \mathrel{++_S} m = m\ f|`\ S$
$m \mathrel{++_S} \bot = m\ f|`\ (-S)$
⟨*proof*⟩

lemma *edom-override-on*[*simp*]: $edom\ (m1 \mathrel{++_S} m2) = (edom\ m1\ -\ S) \cup (edom\ m2 \cap S)$
⟨*proof*⟩

lemma *lookup-override-on-eq*: $(m1 \mathrel{++_S} m2)\ x = (\mathit{if}\ x \in S\ \mathit{then}\ m2\ x\ \mathit{else}\ m1\ x)$
⟨*proof*⟩

lemma *override-on-upd-swap*:
$x \notin S \Longrightarrow \varrho(x := z) \mathrel{++_S} \varrho' = (\varrho \mathrel{++_S} \varrho')(x := z)$
⟨*proof*⟩

lemma *override-on-upd*:
$x \in S \Longrightarrow \varrho \mathrel{++_S} (\varrho'(x := z)) = (\varrho \mathrel{++_{S\ -\ \{x\}}} \varrho')(x := z)$
⟨*proof*⟩

lemma *env-restr-add*: $(m1 \mathrel{++_{S2}} m2)\ f|`\ S = m1\ f|`\ S \mathrel{++_{S2}} m2\ f|`\ S$
⟨*proof*⟩

lemma *env-delete-add*: $\mathit{env\text{-}delete}\ x\ (m1 \mathrel{++_S} m2) = \mathit{env\text{-}delete}\ x\ m1 \mathrel{++_{S\ -\ \{x\}}} \mathit{env\text{-}delete}\ x\ m2$
⟨*proof*⟩

2.9.6 Environments with binary joins

lemma *edom-join*[*simp*]: $edom\ (f \sqcup (g::('a::type \Rightarrow 'b::\{Finite\text{-}Join\text{-}cpo,pcpo\}))) = edom\ f \cup edom\ g$
⟨*proof*⟩

lemma *env-delete-join*[*simp*]: $\mathit{env\text{-}delete}\ x\ (f \sqcup (g::('a::type \Rightarrow 'b::\{Finite\text{-}Join\text{-}cpo,pcpo\}))) = \mathit{env\text{-}delete}\ x\ f \sqcup \mathit{env\text{-}delete}\ x\ g$
⟨*proof*⟩

lemma *env-restr-join*:
fixes $m1\ m2 :: 'a::type \Rightarrow 'b::\{Finite\text{-}Join\text{-}cpo,pcpo\}$
shows $(m1 \sqcup m2)\ f|`\ S = (m1\ f|`\ S) \sqcup (m2\ f|`\ S\ )$

$\langle proof \rangle$

lemma *env-restr-join2*:
fixes m :: $'a$::*type* $\Rightarrow$ $'b$::{*Finite-Join-cpo*,*pcpo*}
shows $m\ f|`\ S \sqcup m\ f|`\ S' = m\ f|`\ (S \cup S')$
$\langle proof \rangle$

lemma *join-env-restr-UNIV*:
fixes m :: $'a$::*type* $\Rightarrow$ $'b$::{*Finite-Join-cpo*,*pcpo*}
shows $S1 \cup S2 = UNIV \Longrightarrow (m\ f|`\ S1) \sqcup (m\ f|`\ S2) = m$
$\langle proof \rangle$

lemma *env-restr-split*:
fixes m :: $'a$::*type* $\Rightarrow$ $'b$::{*Finite-Join-cpo*,*pcpo*}
shows $m = m\ f|`\ S \sqcup m\ f|`\ (-\ S)$
$\langle proof \rangle$

lemma *env-restr-below-split*:
$m\ f|`\ S \sqsubseteq m' \Longrightarrow m\ f|`\ (-\ S) \sqsubseteq m' \Longrightarrow m \sqsubseteq m'$
$\langle proof \rangle$

2.9.7 Singleton environments

definition *esing* :: $'a \Rightarrow 'b$::{*pcpo*} $\rightarrow$ $('a \Rightarrow 'b)$
where $esing\ x = (\Lambda\ a.\ (\lambda\ y\ .\ (if\ x = y\ then\ a\ else\ \bot)))$

lemma *esing-bot*[*simp*]: $esing\ x \cdot \bot = \bot$
$\langle proof \rangle$

lemma *esing-simps*[*simp*]:
$(esing\ x \cdot n)\ x = n$
$x' \neq x \Longrightarrow (esing\ x \cdot n)\ x' = \bot$
$\langle proof \rangle$

lemma *esing-eq-up-iff*[*simp*]: $(esing\ x \cdot (up \cdot a))\ y = up \cdot a' \longleftrightarrow (x = y \wedge a = a')$
$\langle proof \rangle$

lemma *esing-below-iff*[*simp*]: $esing\ x \cdot a \sqsubseteq ae \ \longleftrightarrow a \sqsubseteq ae\ x$
$\langle proof \rangle$

lemma *edom-esing-subset*: $edom\ (esing\ x \cdot n) \subseteq \{x\}$
$\langle proof \rangle$

lemma *edom-esing-up*[*simp*]: $edom\ (esing\ x \cdot (up \cdot n)) = \{x\}$
$\langle proof \rangle$

lemma *env-delete-esing*[*simp*]: $env\text{-}delete\ x\ (esing\ x \cdot n) = \bot$
$\langle proof \rangle$

lemma *env-restr-esing*[*simp*]:
$x \in S \Longrightarrow esing\ x{\cdot}v\ f|`\ S = esing\ x{\cdot}v$
⟨*proof*⟩

lemma *env-restr-esing2*[*simp*]:
$x \notin S \Longrightarrow esing\ x{\cdot}v\ f|`\ S = \bot$
⟨*proof*⟩

lemma *esing-eq-iff*[*simp*]:
$esing\ x{\cdot}v = esing\ x{\cdot}v' \longleftrightarrow v = v'$
⟨*proof*⟩

end

2.10 Env-Nominal

theory *Env−Nominal*
imports *Env Nominal−Utils Nominal−HOLCF*
begin

2.10.1 Equivariance lemmas

lemma *edom-perm*:
fixes $f :: {}'a{::}pt \Rightarrow {}'b{::}\{pcpo\text{-}pt\}$
shows $edom\ (\pi \bullet f) = \pi \bullet (edom\ f)$
⟨*proof*⟩

lemmas *edom-perm-rev*[*simp*,*eqvt*] = *edom-perm*[*symmetric*]

lemma *mem-edom-perm*[*simp*]:
fixes $\varrho :: {}'a{::}at\text{-}base \Rightarrow {}'b{::}\{pcpo\text{-}pt\}$
shows $xa \in edom\ (p \bullet \varrho) \longleftrightarrow -\ p \bullet xa \in edom\ \varrho$
⟨*proof*⟩

lemma *env-restr-eqvt*[*eqvt*]:
fixes $m :: {}'a{::}pt \Rightarrow {}'b{::}\{cont\text{-}pt,pcpo\}$
shows $\pi \bullet m\ f|`\ d = (\pi \bullet m)\ f|`\ (\pi \bullet d)$
⟨*proof*⟩

lemma *env-delete-eqvt*[*eqvt*]:
fixes $m :: {}'a{::}pt \Rightarrow {}'b{::}\{cont\text{-}pt,pcpo\}$
shows $\pi \bullet env\text{-}delete\ x\ m = env\text{-}delete\ (\pi \bullet x)\ (\pi \bullet m)$
⟨*proof*⟩

lemma *esing-eqvt*[*eqvt*]: $\pi \bullet (esing\ x) = esing\ (\pi \bullet x)$
⟨*proof*⟩

2.10.2 Permutation and restriction

lemma *env-restr-perm*:
  fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
  assumes *supp* p $\sharp*$ S **and** [*simp*]: *finite* S
  shows $(p \cdot \varrho)\ f|`\ S = \varrho\ f|`\ S$
⟨*proof*⟩

lemma *env-restr-perm'*:
  fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
  assumes *supp* p $\sharp*$ S **and** [*simp*]: *finite* S
  shows $p \cdot (\varrho\ f|`\ S) = \varrho\ f|`\ S$
  ⟨*proof*⟩

lemma *env-restr-flip*:
  fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
  assumes $x \notin S$ **and** $x' \notin S$
  shows $((x' \leftrightarrow x) \cdot \varrho)\ f|`\ S = \varrho\ f|`\ S$
  ⟨*proof*⟩

lemma *env-restr-flip'*:
  fixes ϱ :: $'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo-pt*,*pure*}
  assumes $x \notin S$ **and** $x' \notin S$
  shows $(x' \leftrightarrow x) \cdot (\varrho\ f|`\ S) = \varrho\ f|`\ S$
  ⟨*proof*⟩

2.10.3 Pure codomains

lemma *edom-fv-pure*:
  fixes f :: ($'a$::*at-base* $\Rightarrow$ $'b$::{*pcpo*,*pure*})
  assumes *finite* (*edom* f)
  shows *fv* f $\subseteq$ *edom* f
⟨*proof*⟩

end

2.11 Env-HOLCF

theory *Env−HOLCF*
  imports *Env HOLCF−Utils*
begin

2.11.1 Continuity and pcpo-valued functions

lemma *override-on-belowI*:
  assumes $\bigwedge a.\ a \in S \Longrightarrow y\ a \sqsubseteq z\ a$
  and $\bigwedge a.\ a \notin S \Longrightarrow x\ a \sqsubseteq z\ a$
  shows $x\ {+}{+}_S\ y \sqsubseteq z$

$\langle proof \rangle$

lemma *override-on-cont1*: $cont\ (\lambda\ x.\ x\ {+}{+}_S\ m)$
$\langle proof \rangle$

lemma *override-on-cont2*: $cont\ (\lambda\ x.\ m\ {+}{+}_S\ x)$
$\langle proof \rangle$

lemma *override-on-cont2cont*[*simp*, *cont2cont*]:
assumes $cont\ f$
assumes $cont\ g$
shows $cont\ (\lambda\ x.\ f\ x\ {+}{+}_S\ g\ x)$
$\langle proof \rangle$

lemma *override-on-mono*:
assumes $x1 \sqsubseteq (x2 :: 'a::type \Rightarrow 'b::cpo)$
assumes $y1 \sqsubseteq y2$
shows $x1\ {+}{+}_S\ y1 \sqsubseteq x2\ {+}{+}_S\ y2$
$\langle proof \rangle$

lemma *fun-upd-below-env-deleteI*:
assumes $env\text{-}delete\ x\ \varrho \sqsubseteq env\text{-}delete\ x\ \varrho'$
assumes $y \sqsubseteq \varrho'\ x$
shows $\varrho(x := y) \sqsubseteq \varrho'$
$\langle proof \rangle$

lemma *fun-upd-belowI2*:
assumes $\bigwedge\ z\ .\ z \neq x \Longrightarrow \varrho\ z \sqsubseteq \varrho'\ z$
assumes $\varrho\ x \sqsubseteq y$
shows $\varrho \sqsubseteq \varrho'(x := y)$
$\langle proof \rangle$

lemma *env-restr-belowI*:
assumes $\bigwedge\ x.\ x \in S \Longrightarrow (m1\ f|`\ S)\ x \sqsubseteq (m2\ f|`\ S)\ x$
shows $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
$\langle proof \rangle$

lemma *env-restr-belowI2*:
assumes $\bigwedge\ x.\ x \in S \Longrightarrow m1\ x \sqsubseteq m2\ x$
shows $m1\ f|`\ S \sqsubseteq m2$
$\langle proof \rangle$

lemma *env-restr-below-itself*:
shows $m\ f|`\ S \sqsubseteq m$
$\langle proof \rangle$

lemma *env-restr-cont*: $cont\ (env\text{-}restr\ S)$
$\langle proof \rangle$

lemma *env-restr-belowD*:
  assumes $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
  assumes $x \in S$
  shows $m1\ x \sqsubseteq m2\ x$
  $\langle proof \rangle$

lemma *env-restr-eqD*:
  assumes $m1\ f|`\ S = m2\ f|`\ S$
  assumes $x \in S$
  shows $m1\ x = m2\ \ x$
  $\langle proof \rangle$

lemma *env-restr-below-subset*:
  assumes $S \subseteq S'$
  and $m1\ f|`\ S' \sqsubseteq m2\ f|`\ S'$
  shows $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
$\langle proof \rangle$

lemma *override-on-below-restrI*:
  assumes $x\ f|`\ (-S) \sqsubseteq z\ f|`\ (-S)$
  and $y\ f|`\ S \sqsubseteq z\ f|`\ S$
  shows $x ++_S y \sqsubseteq z$
$\langle proof \rangle$

lemma *fmap-below-add-restrI*:
  assumes $x\ f|`\ (-S) \sqsubseteq y\ f|`\ (-S)$
  and $x\ f|`\ S \sqsubseteq z\ f|`\ S$
  shows $x \sqsubseteq y ++_S z$
$\langle proof \rangle$

lemmas *env-restr-cont2cont*[*simp*,*cont2cont*] = *cont-compose*[*OF env-restr-cont*]

lemma *env-delete-cont*: *cont* (*env-delete* x)
  $\langle proof \rangle$
lemmas *env-delete-cont2cont*[*simp*,*cont2cont*] = *cont-compose*[*OF env-delete-cont*]

end

2.12 EvalHeap

theory *EvalHeap*
  imports *AList−Utils Env Nominal2.Nominal2 HOLCF−Utils*
begin

2.12.1 Conversion from heaps to environments

fun
  evalHeap :: $('var \times 'exp)$ *list* $\Rightarrow$ $('exp \Rightarrow 'value::\{pure,pcpo\}) \Rightarrow 'var \Rightarrow 'value$
where
  $evalHeap\ []\ \text{-} = \bot$
$|$ $evalHeap\ ((x,e)\#h)\ eval = (evalHeap\ h\ eval)\ (x := eval\ e)$

lemma *cont2cont-evalHeap*[*simp*, *cont2cont*]:
  $(\bigwedge e\ .\ e \in snd\ `\ set\ h \Longrightarrow cont\ (\lambda\varrho.\ eval\ \varrho\ e)) \Longrightarrow cont\ (\lambda\ \varrho.\ evalHeap\ h\ (eval\ \varrho))$
  ⟨*proof*⟩

lemma *evalHeap-eqvt*[*eqvt*]:
  $\pi \cdot evalHeap\ h\ eval = evalHeap\ (\pi \cdot h)\ (\pi \cdot eval)$
  ⟨*proof*⟩

lemma *edom-evalHeap-subset*:$edom\ (evalHeap\ h\ eval) \subseteq domA\ h$
  ⟨*proof*⟩

lemma *evalHeap-cong*[*fundef-cong*]:
  $[\![\ heap1 = heap2\ ;\ (\bigwedge e.\ e \in snd\ `\ set\ heap2 \Longrightarrow eval1\ e = eval2\ e)\ ]\!]$
    $\Longrightarrow\ evalHeap\ heap1\ eval1 = evalHeap\ heap2\ eval2$
 ⟨*proof*⟩

lemma *lookupEvalHeap*:
  assumes $v \in domA\ h$
  shows $(evalHeap\ h\ f)\ v = f\ (the\ (map\text{-}of\ h\ v))$
  ⟨*proof*⟩

lemma *lookupEvalHeap'*:
  assumes $map\text{-}of\ \Gamma\ v = Some\ e$
  shows $(evalHeap\ \Gamma\ f)\ v = f\ e$
  ⟨*proof*⟩

lemma *lookupEvalHeap-other*[*simp*]:
  assumes $v \notin domA\ \Gamma$
  shows $(evalHeap\ \Gamma\ f)\ v = \bot$
  ⟨*proof*⟩

lemma *env-restr-evalHeap-noop*:
  $domA\ h \subseteq S \Longrightarrow env\text{-}restr\ S\ (evalHeap\ h\ eval) = evalHeap\ h\ eval$
  ⟨*proof*⟩

lemma *env-restr-evalHeap-same*[*simp*]:
  $env\text{-}restr\ (domA\ h)\ (evalHeap\ h\ eval) = evalHeap\ h\ eval$
  ⟨*proof*⟩

lemma *evalHeap-cong'*:
  $[\![\ (\bigwedge x.\ x \in domA\ heap \Longrightarrow eval1\ (the\ (map\text{-}of\ heap\ x)) = eval2\ (the\ (map\text{-}of\ heap\ x)))\ ]\!]$
    $\Longrightarrow\ evalHeap\ heap\ eval1 = evalHeap\ heap\ eval2$

$\langle proof \rangle$

lemma *lookupEvalHeapNotAppend*[*simp*]:
assumes $x \notin domA\ \Gamma$
shows $(evalHeap\ (\Gamma@h)\ f)\ x = evalHeap\ h\ f\ x$
$\langle proof \rangle$

lemma *evalHeap-delete*[*simp*]: $evalHeap\ (delete\ x\ \Gamma)\ eval = env\text{-}delete\ x\ (evalHeap\ \Gamma\ eval)$
$\langle proof \rangle$

lemma *evalHeap-mono*:
$x \notin domA\ \Gamma \Longrightarrow$
$evalHeap\ \Gamma\ eval \sqsubseteq evalHeap\ ((x,\ e)\ \#\ \Gamma)\ eval$
$\langle proof \rangle$

2.12.2 Reordering lemmas

lemma *evalHeap-reorder*:
assumes $map\text{-}of\ \Gamma = map\text{-}of\ \Delta$
shows $evalHeap\ \Gamma\ h = evalHeap\ \Delta\ h$
$\langle proof \rangle$

lemma *evalHeap-reorder-head*:
assumes $x \neq y$
shows $evalHeap\ ((x,e1)\#(y,e2)\#\Gamma)\ eval = evalHeap\ ((y,e2)\#(x,e1)\#\Gamma)\ eval$
$\langle proof \rangle$

lemma *evalHeap-reorder-head-append*:
assumes $x \notin domA\ \Gamma$
shows $evalHeap\ ((x,e)\#\Gamma@\Delta)\ eval = evalHeap\ (\Gamma\ @\ ((x,e)\#\Delta))\ eval$
$\langle proof \rangle$

lemma *evalHeap-subst-exp*:
assumes $eval\ e = eval\ e'$
shows $evalHeap\ ((x,e)\#\Gamma)\ eval = evalHeap\ ((x,e')\#\Gamma)\ eval$
$\langle proof \rangle$

end

3 Launchbury's natural semantics

3.1 Vars

theory *Vars*
imports *Nominal2.Nominal2*
begin

The type of variables is abstract and provided by the Nominal package. All we know is that it is countable.

atom-decl *var*

end

3.2 Terms

theory *Terms*
  imports *Nominal−Utils Vars ALList−Utils−Nominal*
begin

3.2.1 Expressions

This is the main data type of the development; our minimal lambda calculus with recursive let-bindings. It is created using the nominal_datatype command, which creates alpha-equivalence classes.

The package does not support nested recursion, so the bindings of the let cannot simply be of type (*var*, *exp*) *list*. Instead, the definition of lists have to be inlined here, as the custom type *assn*. Later we create conversion functions between these two types, define a properly typed *let* and redo the various lemmas in terms of that, so that afterwards, the type *assn* is no longer referenced.

nominal-datatype *exp* =
  Var var
| *App exp var*
| *LetA as*::*assn body*::*exp* **binds** *bn as* **in** *body as*
| *Lam x*::*var body*::*exp* **binds** *x* **in** *body* (‹*Lam* [-]. -› [*100*, *100*] *100*)
| *Bool bool*
| *IfThenElse exp exp exp* (‹((-)/ ? (-)/ : (-))› [*0*, *0*, *10*] *10*)
and *assn* =
  ANil | *ACons var exp assn*
binder
  bn :: *assn* ⇒ *atom list*
where *bn ANil* = [] | *bn* (*ACons x t as*) = (*atom x*) # (*bn as*)

notation (*latex* **output**) *Terms.Var* (‹-›)
notation (*latex* **output**) *Terms.App* (‹- -›)

notation (*latex* **output**) *Terms.Lam* (‹λ-. -› [*100*, *100*] *100*)

type-synonym *heap* = (*var* × *exp*) *list*

lemma *exp-assn-size-eqvt*[*eqvt*]: *p* • (*size* :: *exp* ⇒ *nat*) = *size*
⟨*proof*⟩

3.2.2 Rewriting in terms of heaps

We now work towards using *heap* instead of *assn*. All this could be skipped if Nominal supported nested recursion.

Conversion from *assn* to *heap*.

nominal-function *asToHeap* :: *assn* ⇒ *heap*
where *ANilToHeap*: *asToHeap ANil* = []
| *AConsToHeap*: *asToHeap* (*ACons v e as*) = (*v*, *e*) # *asToHeap as*
⟨*proof*⟩
nominal-termination(*eqvt*) ⟨*proof*⟩

lemma *asToHeap-eqvt*: *eqvt asToHeap*
⟨*proof*⟩

The other direction.

fun *heapToAssn* :: *heap* ⇒ *assn*
where *heapToAssn* [] = *ANil*
| *heapToAssn* ((*v*,*e*)#Γ) = *ACons v e* (*heapToAssn* Γ)

declare *heapToAssn.simps*[*simp del*]

lemma *heapToAssn-eqvt*[*simp*,*eqvt*]: *p* • *heapToAssn* Γ = *heapToAssn* (*p* • Γ)
⟨*proof*⟩

lemma *bn-heapToAssn*: *bn* (*heapToAssn* Γ) = *map* (λ*x*. *atom* (*fst x*)) Γ
⟨*proof*⟩

lemma *set-bn-to-atom-domA*:
set (*bn as*) = *atom* ‘ *domA* (*asToHeap as*)
⟨*proof*⟩

They are inverse to each other.

lemma *heapToAssn-asToHeap*[*simp*]:
heapToAssn (*asToHeap as*) = *as*
⟨*proof*⟩

lemma *asToHeap-heapToAssn*[*simp*]:
asToHeap (*heapToAssn as*) = *as*
⟨*proof*⟩

lemma *heapToAssn-inject*[*simp*]:
heapToAssn x = *heapToAssn* $y \longleftrightarrow x = y$
⟨*proof*⟩

They are transparent to various notions from the Nominal package.

lemma *supp-heapToAssn*: *supp* (*heapToAssn* Γ) = *supp* Γ
⟨*proof*⟩

lemma *supp-asToHeap*: *supp* (*asToHeap as*) = *supp as*
⟨*proof*⟩

lemma *fv-asToHeap*: *fv* (*asToHeap* Γ) = *fv* Γ
⟨*proof*⟩

lemma *fv-heapToAssn*: *fv* (*heapToAssn* Γ) = *fv* Γ
⟨*proof*⟩

lemma [*simp*]: *size* (*heapToAssn* Γ) = *size-list* (λ (*v*,*e*) . *size e*) Γ
⟨*proof*⟩

lemma *Lam-eq-same-var*[*simp*]: *Lam* [*y*]. e = *Lam* [*y*]. $e' \longleftrightarrow e = e'$
⟨*proof*⟩

Now we define the Let constructor in the form that we actually want.

hide-const *HOL.Let*
definition *Let* :: *heap* ⇒ *exp* ⇒ *exp*
where *Let* Γ *e* = *LetA* (*heapToAssn* Γ) *e*

notation (*latex* **output**) *Let* (‹*let* - *in* -›)

abbreviation
LetBe :: *var*⇒*exp*⇒*exp*⇒*exp* (‹*let* - *be* - *in* - › [*100*,*100*,*100*] *100*)
where
let x be t1 in t2 ≡ *Let* [(*x*,*t1*)] *t2*

We rewrite all (relevant) lemmas about *LetA* in terms of *Let*.

lemma *size-Let*[*simp*]: *size* (*Let* Γ *e*) = *size-list* (λ*p*. *size* (*snd p*)) Γ + *size e* + *Suc 0*
⟨*proof*⟩

lemma *Let-distinct*[*simp*]:
Var v ≠ *Let* Γ *e*
Let Γ *e* ≠ *Var v*
App e v ≠ *Let* Γ e'
Lam [*v*]. e' ≠ *Let* Γ *e*
Let Γ *e* ≠ *Lam* [*v*]. e'
Let Γ e' ≠ *App e v*
Bool b ≠ *Let* Γ *e*

$Let\ \Gamma\ e \neq Bool\ b$
$(scrut\ ?\ e1\ :\ e2) \neq Let\ \Gamma\ e$
$Let\ \Gamma\ e \neq (scrut\ ?\ e1\ :\ e2)$
⟨*proof*⟩

lemma *Let-perm-simps*[*simp*,*eqvt*]:
$p \bullet Let\ \Gamma\ e = Let\ (p \bullet \Gamma)\ (p \bullet e)$
⟨*proof*⟩

lemma *Let-supp*:
$supp\ (Let\ \Gamma\ e) = (supp\ e \cup supp\ \Gamma) - atom\ `\ (domA\ \Gamma)$
⟨*proof*⟩

lemma *Let-fresh*[*simp*]:
$a \sharp\ Let\ \Gamma\ e = (a \sharp\ e \wedge a \sharp\ \Gamma \vee a \in atom\ `\ domA\ \Gamma)$
⟨*proof*⟩

lemma *Abs-eq-cong*:
assumes $\bigwedge p.\ (p \bullet x = x') \longleftrightarrow (p \bullet y = y')$
assumes $supp\ y = supp\ x$
assumes $supp\ y' = supp\ x'$
shows $([a]lst.\ x = [a']lst.\ x') \longleftrightarrow ([a]lst.\ y = [a']lst.\ y')$
⟨*proof*⟩

lemma *Let-eq-iff*[*simp*]:
$(Let\ \Gamma\ e = Let\ \Gamma'\ e') = ([map\ (\lambda x.\ atom\ (fst\ x))\ \Gamma\ ]lst.\ (e, \Gamma) = [map\ (\lambda x.\ atom\ (fst\ x))\ \Gamma']lst.\ (e',\Gamma'))$
⟨*proof*⟩

lemma *exp-strong-exhaust*:
fixes $c :: 'a :: fs$
assumes $\bigwedge var.\ y = Var\ var \Longrightarrow P$
assumes $\bigwedge exp\ var.\ y = App\ exp\ var \Longrightarrow P$
assumes $\bigwedge \Gamma\ exp.\ atom\ `\ domA\ \Gamma \sharp{*}\ c \Longrightarrow y = Let\ \Gamma\ exp \Longrightarrow P$
assumes $\bigwedge var\ exp.\ \{atom\ var\} \sharp{*}\ c \Longrightarrow y = Lam\ [var].\ exp \Longrightarrow P$
assumes $\bigwedge b.\ (y = Bool\ b) \Longrightarrow P$
assumes $\bigwedge scrut\ e1\ e2.\ y = (scrut\ ?\ e1\ :\ e2) \Longrightarrow P$
shows P
⟨*proof*⟩

And finally the induction rules with *Let*.

lemma *exp-heap-induct*[*case-names Var App Let Lam Bool IfThenElse Nil Cons*]:
assumes $\bigwedge b\ var.\ P1\ (Var\ var)$
assumes $\bigwedge exp\ var.\ P1\ exp \Longrightarrow P1\ (App\ exp\ var)$
assumes $\bigwedge \Gamma\ exp.\ P2\ \Gamma \Longrightarrow P1\ exp \Longrightarrow P1\ (Let\ \Gamma\ exp)$
assumes $\bigwedge var\ exp.\ P1\ exp \Longrightarrow P1\ (Lam\ [var].\ exp)$
assumes $\bigwedge b.\ P1\ (Bool\ b)$
assumes $\bigwedge scrut\ e1\ e2.\ P1\ scrut \Longrightarrow P1\ e1 \Longrightarrow P1\ e2 \Longrightarrow P1\ (scrut\ ?\ e1\ :\ e2)$
assumes $P2\ []$

assumes $\bigwedge$*var exp* Γ. *P1 exp* $\Longrightarrow$ *P2* Γ $\Longrightarrow$ *P2* ((*var*, *exp*)#Γ)
shows *P1 e* **and** *P2* Γ
⟨*proof*⟩

lemma *exp-heap-strong-induct*[*case-names Var App Let Lam Bool IfThenElse Nil Cons*]:
assumes $\bigwedge$*var c. P1 c* (*Var var*)
assumes $\bigwedge$*exp var c.* ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ *P1 c* (*App exp var*)
assumes $\bigwedge\Gamma$ *exp c. atom ‘ domA* Γ $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c. P2 c* Γ) $\Longrightarrow$ ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ *P1 c* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp c.* {*atom var*} $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ *P1 c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$ *b c. P1 c* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2 c.* ($\bigwedge$ *c. P1 c scrut*) $\Longrightarrow$ ($\bigwedge$ *c. P1 c e1*) $\Longrightarrow$ ($\bigwedge$ *c. P1 c e2*) $\Longrightarrow$ *P1 c* (*scrut ? e1 : e2*)
assumes $\bigwedge$*c. P2 c* []
assumes $\bigwedge$*var exp* Γ *c.* ($\bigwedge$*c. P1 c exp*) $\Longrightarrow$ ($\bigwedge$*c. P2 c* Γ) $\Longrightarrow$ *P2 c* ((*var*,*exp*)#Γ)
fixes *c* :: *'a* :: *fs*
shows *P1 c e* **and** *P2 c* Γ
⟨*proof*⟩

3.2.3 Nice induction rules

These rules can be used instead of the original induction rules, which require a separate goal for *assn*.

lemma *exp-induct*[*case-names Var App Let Lam Bool IfThenElse*]:
assumes $\bigwedge$*var. P* (*Var var*)
assumes $\bigwedge$*exp var. P exp* $\Longrightarrow$ *P* (*App exp var*)
assumes $\bigwedge\Gamma$ *exp.* ($\bigwedge$ *x. x* $\in$ *domA* Γ $\Longrightarrow$ *P* (*the* (*map-of* Γ *x*))) $\Longrightarrow$ *P exp* $\Longrightarrow$ *P* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp. P exp* $\Longrightarrow$ *P* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$ *b. P* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2. P scrut* $\Longrightarrow$ *P e1* $\Longrightarrow$ *P e2* $\Longrightarrow$ *P* (*scrut ? e1 : e2*)
shows *P exp*
⟨*proof*⟩

lemma *exp-strong-induct-set*[*case-names Var App Let Lam Bool IfThenElse*]:
assumes $\bigwedge$*var c. P c* (*Var var*)
assumes $\bigwedge$*exp var c.* ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*App exp var*)
assumes $\bigwedge\Gamma$ *exp c.*
atom ‘ domA Γ $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c x e.* (*x*,*e*) $\in$ *set* Γ $\Longrightarrow$ *P c e*) $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp c.* {*atom var*} $\sharp$* *c* $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c. P c* (*Bool b*)
assumes $\bigwedge$*scrut e1 e2 c.* ($\bigwedge$ *c. P c scrut*) $\Longrightarrow$ ($\bigwedge$ *c. P c e1*) $\Longrightarrow$ ($\bigwedge$ *c. P c e2*) $\Longrightarrow$ *P c* (*scrut ? e1 : e2*)
shows *P* (*c*::*'a*::*fs*) *exp*
⟨*proof*⟩

lemma *exp-strong-induct*[*case-names Var App Let Lam Bool IfThenElse*]:
assumes $\bigwedge$*var c. P c* (*Var var*)
assumes $\bigwedge$*exp var c.* ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*App exp var*)
assumes $\bigwedge\Gamma$ *exp c.*
atom ‘ *domA* Γ $\sharp*$ *c* $\Longrightarrow$ ($\bigwedge$*c x. x* $\in$ *domA* Γ $\Longrightarrow$ *P c* (*the* (*map-of* Γ *x*))) $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp c.* {*atom var*} $\sharp*$ *c* $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c. P c* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2 c.* ($\bigwedge$ *c. P c scrut*) $\Longrightarrow$ ($\bigwedge$ *c. P c e1*) $\Longrightarrow$ ($\bigwedge$ *c. P c e2*) $\Longrightarrow$ *P c* (*scrut ? e1 : e2*)
shows *P* (*c*::*'a*::*fs*) *exp*
⟨*proof*⟩

3.2.4 Testing alpha equivalence

lemma *alpha-test*:
shows *Lam* [*x*]. (*Var x*) = *Lam* [*y*]. (*Var y*)
⟨*proof*⟩

lemma *alpha-test2*:
shows *let x be* (*Var x*) *in* (*Var x*) = *let y be* (*Var y*) *in* (*Var y*)
⟨*proof*⟩

lemma *alpha-test3*:
shows
Let [(*x*, *Var y*), (*y*, *Var x*)] (*Var x*)
=
Let [(*y*, *Var x*), (*x*, *Var y*)] (*Var y*) (**is** *Let ?la ?lb* = -)
⟨*proof*⟩

3.2.5 Free variables

lemma *fv-supp-exp*: *supp e* = *atom* ‘ (*fv* (*e*::*exp*) :: *var set*) **and** *fv-supp-as*: *supp as* = *atom* ‘ (*fv* (*as*::*assn*) :: *var set*)
⟨*proof*⟩

lemma *fv-supp-heap*: *supp* (Γ::*heap*) = *atom* ‘ (*fv* Γ :: *var set*)
⟨*proof*⟩

lemma *fv-Lam*[*simp*]: *fv* (*Lam* [*x*]. *e*) = *fv e* − {*x*}
⟨*proof*⟩
lemma *fv-Var*[*simp*]: *fv* (*Var x*) = {*x*}
⟨*proof*⟩
lemma *fv-App*[*simp*]: *fv* (*App e x*) = *insert x* (*fv e*)
⟨*proof*⟩
lemma *fv-Let*[*simp*]: *fv* (*Let* Γ *e*) = (*fv* Γ $\cup$ *fv e*) − *domA* Γ
⟨*proof*⟩
lemma *fv-Bool*[*simp*]: *fv* (*Bool b*) = {}
⟨*proof*⟩
lemma *fv-IfThenElse*[*simp*]: *fv* (*scrut ? e1 : e2*) = *fv scrut* $\cup$ *fv e1* $\cup$ *fv e2*

⟨*proof*⟩

lemma *fv-delete-heap*:
assumes $\mathit{map\text{-}of}\ \Gamma\ x = \mathit{Some}\ e$
shows $\mathit{fv}\ (\mathit{delete}\ x\ \Gamma,\ e) \cup \{x\} \subseteq (\mathit{fv}\ (\Gamma,\ \mathit{Var}\ x) :: \mathit{var}\ \mathit{set})$
⟨*proof*⟩

3.2.6 Lemmas helping with nominal definitions

lemma *eqvt-lam-case*:
assumes $\mathit{Lam}\ [x].\ e = \mathit{Lam}\ [x'].\ e'$
assumes $\bigwedge \pi\ .\ \mathit{supp}\ (-\pi)\ \sharp*\ (\mathit{fv}\ (\mathit{Lam}\ [x].\ e) :: \mathit{var}\ \mathit{set}) \Longrightarrow$
$\mathit{supp}\ \pi\ \sharp*\ (\mathit{Lam}\ [x].\ e) \Longrightarrow$
$F\ (\pi \cdot e)\ (\pi \cdot x)\ (\mathit{Lam}\ [x].\ e) = F\ e\ x\ (\mathit{Lam}\ [x].\ e)$
shows $F\ e\ x\ (\mathit{Lam}\ [x].\ e) = F\ e'\ x'\ (\mathit{Lam}\ [x'].\ e')$
⟨*proof*⟩

lemma *eqvt-let-case*:
assumes $\mathit{Let}\ \mathit{as}\ \mathit{body} = \mathit{Let}\ \mathit{as}'\ \mathit{body}'$
assumes $\bigwedge \pi\ .$
$\mathit{supp}\ (-\pi)\ \sharp*\ (\mathit{fv}\ (\mathit{Let}\ \mathit{as}\ \mathit{body}) :: \mathit{var}\ \mathit{set}) \Longrightarrow$
$\mathit{supp}\ \pi\ \sharp*\ \mathit{Let}\ \mathit{as}\ \mathit{body} \Longrightarrow$
$F\ (\pi \cdot \mathit{as})\ (\pi \cdot \mathit{body})\ (\mathit{Let}\ \mathit{as}\ \mathit{body}) = F\ \mathit{as}\ \mathit{body}\ (\mathit{Let}\ \mathit{as}\ \mathit{body})$
shows $F\ \mathit{as}\ \mathit{body}\ (\mathit{Let}\ \mathit{as}\ \mathit{body}) = F\ \mathit{as}'\ \mathit{body}'\ (\mathit{Let}\ \mathit{as}'\ \mathit{body}')$
⟨*proof*⟩

3.2.7 A smart constructor for lets

Certian program transformations might change the bound variables, possibly making it an empty list. This smart constructor avoids the empty let in the resulting expression. Semantically, it should not make a difference.

definition *SmartLet* :: *heap* => *exp* => *exp*
where $\mathit{SmartLet}\ \Gamma\ e = (\mathit{if}\ \Gamma = []\ \mathit{then}\ e\ \mathit{else}\ \mathit{Let}\ \Gamma\ e)$

lemma *SmartLet-eqvt*[*eqvt*]: $\pi \cdot (\mathit{SmartLet}\ \Gamma\ e) = \mathit{SmartLet}\ (\pi \cdot \Gamma)\ (\pi \cdot e)$
⟨*proof*⟩

lemma *SmartLet-supp*:
$\mathit{supp}\ (\mathit{SmartLet}\ \Gamma\ e) = (\mathit{supp}\ e \cup \mathit{supp}\ \Gamma) - \mathit{atom}\ `\ (\mathit{domA}\ \Gamma)$
⟨*proof*⟩

lemma *fv-SmartLet*[*simp*]: $\mathit{fv}\ (\mathit{SmartLet}\ \Gamma\ e) = (\mathit{fv}\ \Gamma \cup \mathit{fv}\ e) - \mathit{domA}\ \Gamma$
⟨*proof*⟩

3.2.8 A predicate for value expressions

nominal-function *isLam* :: *exp* ⇒ *bool* **where**
isLam (*Var x*) = *False* |
isLam (*Lam* [*x*]. *e*) = *True* |
isLam (*App e x*) = *False* |
isLam (*Let as e*) = *False* |
isLam (*Bool b*) = *False* |
isLam (*scrut ? e1* : *e2*) = *False*
⟨*proof*⟩
nominal-termination (*eqvt*) ⟨*proof*⟩

lemma *isLam-Lam*: *isLam* (*Lam* [*x*]. *e*) ⟨*proof*⟩

lemma *isLam-obtain-fresh*:
assumes *isLam z*
obtains *y e'*
where *z* = (*Lam* [*y*]. *e'*) **and** *atom y* ♯ (*c*::'*a*::*fs*)
⟨*proof*⟩

nominal-function *isVal* :: *exp* ⇒ *bool* **where**
isVal (*Var x*) = *False* |
isVal (*Lam* [*x*]. *e*) = *True* |
isVal (*App e x*) = *False* |
isVal (*Let as e*) = *False* |
isVal (*Bool b*) = *True* |
isVal (*scrut ? e1* : *e2*) = *False*
⟨*proof*⟩
nominal-termination (*eqvt*) ⟨*proof*⟩

lemma *isVal-Lam*: *isVal* (*Lam* [*x*]. *e*) ⟨*proof*⟩
lemma *isVal-Bool*: *isVal* (*Bool b*) ⟨*proof*⟩

3.2.9 The notion of thunks

definition *thunks* :: *heap* ⇒ *var set* **where**
thunks Γ = {*x* . *case map-of* Γ *x of Some e* ⇒ ¬ *isVal e* | *None* ⇒ *False*}

lemma *thunks-Nil*[*simp*]: *thunks* [] = {} ⟨*proof*⟩

lemma *thunks-domA*: *thunks* Γ ⊆ *domA* Γ
⟨*proof*⟩

lemma *thunks-Cons*: *thunks* ((*x*,*e*)#Γ) = (*if isVal e then thunks* Γ − {*x*} *else insert x* (*thunks* Γ))
⟨*proof*⟩

lemma *thunks-append*[*simp*]: *thunks* (Δ@Γ) = *thunks* Δ ∪ (*thunks* Γ − *domA* Δ)
⟨*proof*⟩

lemma *thunks-delete*[*simp*]: *thunks* (*delete* x Γ) = *thunks* $\Gamma - \{x\}$
⟨*proof*⟩

lemma *thunksI*[*intro*]: *map-of* Γ x = *Some* $e \Longrightarrow \neg$ *isVal* $e \Longrightarrow x \in$ *thunks* Γ
⟨*proof*⟩

lemma *thunksE*[*intro*]: $x \in$ *thunks* $\Gamma \Longrightarrow$ *map-of* Γ x = *Some* $e \Longrightarrow \neg$ *isVal* e
⟨*proof*⟩

lemma *thunks-cong*: *map-of* Γ = *map-of* $\Delta \Longrightarrow$ *thunks* Γ = *thunks* Δ
⟨*proof*⟩

lemma *thunks-eqvt*[*eqvt*]:
$\pi \cdot$ *thunks* Γ = *thunks* $(\pi \cdot \Gamma)$
⟨*proof*⟩

3.2.10 Non-recursive Let bindings

definition *nonrec* :: *heap* ⇒ *bool* **where**
nonrec $\Gamma = (\exists\ x\ e.\ \Gamma = [(x,e)] \wedge x \notin fv\ e)$

lemma *nonrecE*:
assumes *nonrec* Γ
obtains x e **where** $\Gamma = [(x,e)]$ **and** $x \notin fv\ e$
⟨*proof*⟩

lemma *nonrec-eqvt*[*eqvt*]:
nonrec $\Gamma \Longrightarrow$ *nonrec* $(\pi \cdot \Gamma)$
⟨*proof*⟩

lemma *exp-induct-rec*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:
assumes $\bigwedge var.\ P\ (Var\ var)$
assumes $\bigwedge exp\ var.\ P\ exp \Longrightarrow P\ (App\ exp\ var)$
assumes $\bigwedge\Gamma\ exp.\ \neg\ nonrec\ \Gamma \Longrightarrow (\bigwedge\ x.\ x \in domA\ \Gamma \Longrightarrow\ P\ (the\ (map\text{-}of\ \Gamma\ x))) \Longrightarrow P\ exp \Longrightarrow P\ (Let\ \Gamma\ exp)$
assumes $\bigwedge x\ e\ exp.\ x \notin fv\ e \Longrightarrow P\ e \Longrightarrow P\ exp \Longrightarrow P\ (let\ x\ be\ e\ in\ exp)$
assumes $\bigwedge var\ exp.\ \ P\ exp \Longrightarrow P\ (Lam\ [var].\ exp)$
assumes $\bigwedge b.\ P\ (Bool\ b)$
assumes $\bigwedge\ scrut\ e1\ e2.\ P\ scrut \Longrightarrow P\ e1 \Longrightarrow P\ e2 \Longrightarrow P\ (scrut\ ?\ e1\ :\ e2)$
shows $P\ exp$
⟨*proof*⟩

lemma *exp-strong-induct-rec*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:
assumes $\bigwedge var\ c.\ P\ c\ (Var\ var)$
assumes $\bigwedge exp\ var\ c.\ (\bigwedge c.\ P\ c\ exp) \Longrightarrow P\ c\ (App\ exp\ var)$
assumes $\bigwedge\Gamma\ exp\ c.$
$atom\ `\ domA\ \Gamma\ \sharp{*}\ c \Longrightarrow \neg\ nonrec\ \Gamma \Longrightarrow (\bigwedge c\ x.\ x \in domA\ \Gamma \Longrightarrow\ P\ c\ (the\ (map\text{-}of\ \Gamma\ x))) \Longrightarrow (\bigwedge c.\ P\ c\ exp) \Longrightarrow P\ c\ (Let\ \Gamma\ exp)$

assumes $\bigwedge x\ e\ exp\ c.\ \{atom\ x\} \sharp* c \Longrightarrow x \notin fv\ e \Longrightarrow (\bigwedge\ c.\ P\ c\ e) \Longrightarrow (\bigwedge\ c.\ P\ c\ exp) \Longrightarrow P\ c\ (let\ x\ be\ e\ in\ exp)$
assumes $\bigwedge var\ exp\ c.\ \{atom\ var\} \sharp* c \Longrightarrow (\bigwedge c.\ P\ c\ exp) \Longrightarrow P\ c\ (Lam\ [var].\ exp)$
assumes $\bigwedge b\ c.\ P\ c\ (Bool\ b)$
assumes $\bigwedge\ scrut\ e1\ e2\ c.\ (\bigwedge\ c.\ P\ c\ scrut) \Longrightarrow (\bigwedge\ c.\ P\ c\ e1) \Longrightarrow (\bigwedge\ c.\ P\ c\ e2) \Longrightarrow P\ c\ (scrut\ ?\ e1\ :\ e2)$
shows $P\ (c::'a::fs)\ exp$
$\langle proof \rangle$

lemma *exp-strong-induct-rec-set*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:
assumes $\bigwedge var\ c.\ P\ c\ (Var\ var)$
assumes $\bigwedge exp\ var\ c.\ (\bigwedge c.\ P\ c\ exp) \Longrightarrow P\ c\ (App\ exp\ var)$
assumes $\bigwedge \Gamma\ exp\ c.$
$atom\ `\ domA\ \Gamma \sharp* c \Longrightarrow \neg\ nonrec\ \Gamma \Longrightarrow (\bigwedge c\ x\ e.\ (x,e) \in set\ \Gamma \Longrightarrow\ P\ c\ e) \Longrightarrow (\bigwedge c.\ P\ c\ exp) \Longrightarrow P\ c\ (Let\ \Gamma\ exp)$
assumes $\bigwedge x\ e\ exp\ c.\ \{atom\ x\} \sharp* c \Longrightarrow x \notin fv\ e \Longrightarrow (\bigwedge\ c.\ P\ c\ e) \Longrightarrow (\bigwedge\ c.\ P\ c\ exp) \Longrightarrow P\ c\ (let\ x\ be\ e\ in\ exp)$
assumes $\bigwedge var\ exp\ c.\ \{atom\ var\} \sharp* c \Longrightarrow (\bigwedge c.\ P\ c\ exp) \Longrightarrow P\ c\ (Lam\ [var].\ exp)$
assumes $\bigwedge b\ c.\ P\ c\ (Bool\ b)$
assumes $\bigwedge\ scrut\ e1\ e2\ c.\ (\bigwedge\ c.\ P\ c\ scrut) \Longrightarrow (\bigwedge\ c.\ P\ c\ e1) \Longrightarrow (\bigwedge\ c.\ P\ c\ e2) \Longrightarrow P\ c\ (scrut\ ?\ e1\ :\ e2)$
shows $P\ (c::'a::fs)\ exp$
$\langle proof \rangle$

3.2.11 Renaming a lambda-bound variable

lemma *change-Lam-Variable*:
assumes $y' \neq y \Longrightarrow atom\ y' \sharp (e,\ y)$
shows $Lam\ [y].\ e =\ Lam\ [y'].\ ((y \leftrightarrow y') \cdot e)$
$\langle proof \rangle$

end

3.3 Substitution

theory *Substitution*
imports *Terms*
begin

Defining a substitution function on terms turned out to be slightly tricky.

fun
$subst\text{-}var :: var \Rightarrow var \Rightarrow var \Rightarrow var$ (‹-[-::v=-]› [*1000*,*100*,*100*] *1000*)
where $x[y ::v= z] = (if\ x = y\ then\ z\ else\ x)$

nominal-function (*default case-sum* ($\lambda x.\ Inl\ undefined$) ($\lambda x.\ Inr\ undefined$),
$invariant\ \lambda\ a\ r\ .\ (\forall\ \Gamma\ y\ z\ .\ ((a = Inr\ (\Gamma,\ y,\ z) \wedge atom\ `\ domA\ \Gamma \sharp* (y,\ z)) \longrightarrow map\ (\lambda x\ .\ atom\ (fst\ x))\ \ (Sum\text{-}Type.projr\ r) = map\ (\lambda x\ .\ atom\ (fst\ x))\ \Gamma)))$

subst :: *exp* ⇒ *var* ⇒ *var* ⇒ *exp* (‹-[-::=-]› [*1000*,*100*,*100*] *1000*)
and
subst-heap :: *heap* ⇒ *var* ⇒ *var* ⇒ *heap* (‹-[-::*h*=-]› [*1000*,*100*,*100*] *1000*)
where
(*Var x*)[*y* ::= *z*] = *Var* (*x*[*y* ::*v*= *z*])
| (*App e v*)[*y* ::= *z*] = *App* (*e*[*y* ::= *z*]) (*v*[*y* ::*v*= *z*])
| *atom* ‘ *domA* Γ ♯∗ (*y*,*z*) ⟹
(*Let* Γ *body*)[*y* ::= *z*] = *Let* (Γ[*y* ::*h*= *z*]) (*body*[*y* ::= *z*])
| *atom x* ♯ (*y*,*z*) ⟹ (*Lam* [*x*].*e*)[*y* ::= *z*] = *Lam* [*x*].(*e*[*y*::=*z*])
| (*Bool b*)[*y* ::= *z*] = *Bool b*
| (*scrut ? e1* : *e2*)[*y* ::= *z*] = (*scrut*[*y* ::= *z*] *? e1*[*y* ::= *z*] : *e2*[*y* ::= *z*])
| [][*y* ::*h*= *z*] = []
| ((*v*,*e*)# Γ)[*y* ::*h*= *z*] = (*v*, *e*[*y* ::= *z*])# (Γ[*y* ::*h*= *z*])
⟨*proof*⟩

nominal-termination (*eqvt*) ⟨*proof*⟩

lemma shows
True **and** *bn-subst*[*simp*]: *domA* (*subst-heap* Γ *y z*) = *domA* Γ
⟨*proof*⟩

lemma *subst-noop*[*simp*]:
shows *e*[*y* ::= *y*] = *e* **and** Γ[*y*::*h*=*y*]= Γ
⟨*proof*⟩

lemma *subst-is-fresh*[*simp*]:
assumes *atom y* ♯ *z*
shows
atom y ♯ *e*[*y* ::= *z*]
and
atom ‘ *domA* Γ ♯∗ *y* ⟹ *atom y* ♯ Γ[*y*::*h*=*z*]
⟨*proof*⟩

lemma
subst-pres-fresh: *atom x* ♯ *e* ∨ *x* = *y* ⟹ *atom x* ♯ *z* ⟹ *atom x* ♯ *e*[*y* ::= *z*]
and
atom x ♯ Γ ∨ *x* = *y* ⟹ *atom x* ♯ *z* ⟹ *x* ∉ *domA* Γ ⟹ *atom x* ♯ (Γ[*y* ::*h*= *z*])
⟨*proof*⟩

lemma *subst-fresh-noop*: *atom x* ♯ *e* ⟹ *e*[*x* ::= *y*] = *e*
and *subst-heap-fresh-noop*: *atom x* ♯ Γ ⟹ Γ[*x* ::*h*= *y*] = Γ
⟨*proof*⟩

lemma *supp-subst-eq*: *supp* (*e*[*y*::=*x*]) = (*supp e* − {*atom y*}) ∪ (*if atom y* ∈ *supp e then* {*atom x*} *else* {})
and *atom* ‘ *domA* Γ ♯∗ *y* ⟹ *supp* (Γ[*y*::*h*=*x*]) = (*supp* Γ − {*atom y*}) ∪ (*if atom y* ∈ *supp* Γ *then* {*atom x*} *else* {})
⟨*proof*⟩

lemma *supp-subst*: $supp\ (e[y::=x]) \subseteq (supp\ e - \{atom\ y\}) \cup \{atom\ x\}$
⟨*proof*⟩

lemma *fv-subst-eq*: $fv\ (e[y::=x]) = (fv\ e - \{y\}) \cup (if\ y \in fv\ e\ then\ \{x\}\ else\ \{\})$
and $atom\ `\ domA\ \Gamma\ \sharp*\ y \Longrightarrow fv\ (\Gamma[y::h=x]) = (fv\ \Gamma - \{y\}) \cup (if\ y \in fv\ \Gamma\ then\ \{x\}\ else\ \{\})$
⟨*proof*⟩

lemma *fv-subst-subset*: $fv\ (e[y ::= x]) \subseteq (fv\ e - \{y\}) \cup \{x\}$
⟨*proof*⟩

lemma *fv-subst-int*: $x \notin S \Longrightarrow y \notin S \Longrightarrow fv\ (e[y ::= x]) \cap S = fv\ e \cap S$
⟨*proof*⟩

lemma *fv-subst-int2*: $x \notin S \Longrightarrow y \notin S \Longrightarrow S \cap fv\ (e[y ::= x]) = S \cap fv\ e$
⟨*proof*⟩

lemma *subst-swap-same*: $atom\ x \sharp\ e \Longrightarrow (x \leftrightarrow y) \cdot e = e[y ::=x]$
and $atom\ x \sharp\ \Gamma \Longrightarrow atom\ `domA\ \Gamma\ \sharp*\ y \Longrightarrow (x \leftrightarrow y) \cdot \Gamma = \Gamma[y ::h= x]$
⟨*proof*⟩

lemma *subst-subst-back*: $atom\ x \sharp\ e \Longrightarrow e[y::=x][x::=y] = e$
and $atom\ x \sharp\ \Gamma \Longrightarrow atom\ `domA\ \Gamma\ \sharp*\ y \Longrightarrow \Gamma[y::h=x][x::h=y] = \Gamma$
⟨*proof*⟩

lemma *subst-heap-delete*[*simp*]: $(delete\ x\ \Gamma)[y ::h= z] = delete\ x\ (\Gamma[y ::h= z])$
⟨*proof*⟩

lemma *subst-nil-iff*[*simp*]: $\Gamma[x ::h= z] = [] \longleftrightarrow \Gamma = []$
⟨*proof*⟩

lemma *subst-SmartLet*[*simp*]:
$atom\ `\ domA\ \Gamma\ \sharp*\ (y,z) \Longrightarrow (SmartLet\ \Gamma\ body)[y ::= z] = SmartLet\ (\Gamma[y ::h= z])\ (body[y ::= z])$
⟨*proof*⟩

lemma *subst-let-be*[*simp*]:
$atom\ x' \sharp\ y \Longrightarrow atom\ x' \sharp\ x \Longrightarrow (let\ x'\ be\ e\ in\ exp\ )[y::=x] = (let\ x'\ be\ e[y::=x]\ in\ exp[y::=x])$
⟨*proof*⟩

lemma *isLam-subst*[*simp*]: $isLam\ e[x::=y] = isLam\ e$
⟨*proof*⟩

lemma *isVal-subst*[*simp*]: $isVal\ e[x::=y] = isVal\ e$
⟨*proof*⟩

lemma *thunks-subst*[*simp*]:
$thunks\ \Gamma[y::h=x] = thunks\ \Gamma$

$\langle proof \rangle$

lemma *map-of-subst*:
$map\text{-}of\ (\Gamma[x::h=y])\ k = map\text{-}option\ (\lambda\ e\ .\ e[x::=y])\ (map\text{-}of\ \Gamma\ k)$
$\langle proof \rangle$

lemma *mapCollect-subst*[*simp*]:
$\{e\ k\ v \mid k \mapsto v \in map\text{-}of\ \Gamma[x::h=y]\} = \{e\ k\ v[x::=y] \mid k \mapsto v \in map\text{-}of\ \Gamma\}$
$\langle proof \rangle$

lemma *subst-eq-Cons*:
$\Gamma[x::h=y] = (x',\ e)\#\Delta \longleftrightarrow (\exists\ e'\ \Gamma'.\ \Gamma = (x',e')\#\Gamma' \wedge e'[x::=y] = e \wedge \Gamma'[x::h=y] = \Delta)$
$\langle proof \rangle$

lemma *nonrec-subst*:
$atom\ `\ domA\ \Gamma\ \sharp*\ x \Longrightarrow atom\ `\ domA\ \Gamma\ \sharp*\ y \Longrightarrow nonrec\ \Gamma[x::h=y] \longleftrightarrow nonrec\ \Gamma$
$\langle proof \rangle$

end

3.4 Launchbury

theory *Launchbury*
imports *Terms Substitution*
begin

3.4.1 The natural semantics

This is the semantics as in [Lau93], with two differences:

- Explicit freshness requirements for bound variables in the application and the Let rule.
- An additional parameter that stores variables that have to be avoided, but do not occur in the judgement otherwise, follwing [Ses97].

inductive
$reds :: heap \Rightarrow exp \Rightarrow var\ list \Rightarrow heap \Rightarrow exp \Rightarrow bool$
$(\langle\text{-} : \text{-} \Downarrow_{\text{-}} \text{-} : \text{-}\rangle\ [50,50,50,50]\ 50)$
where
Lambda:
$\Gamma : (Lam\ [x].\ e) \Downarrow_L \Gamma : (Lam\ [x].\ e)$
$\mid$ *Application*: $[\![$
$atom\ y\ \sharp\ (\Gamma,e,x,L,\Delta,\Theta,z)$;
$\Gamma : e \Downarrow_L \Delta : (Lam\ [y].\ e')$;
$\Delta : e'[y ::= x] \Downarrow_L \Theta : z$
$]\!]\ \Longrightarrow$
$\Gamma : App\ e\ x \Downarrow_L \Theta : z$
$\mid$ *Variable*: $[\![$

$map\text{-}of\ \Gamma\ x = Some\ e$; $delete\ x\ \Gamma : e \Downarrow_{x\#L} \Delta : z$
$\rrbracket \Longrightarrow$
$\Gamma : Var\ x \Downarrow_L (x, z) \# \Delta : z$
| *Let*: $\llbracket$
$atom\ `\ domA\ \Delta\ \sharp{*}\ (\Gamma, L)$;
$\Delta\ @\ \Gamma : body \Downarrow_L \Theta : z$
$\rrbracket \Longrightarrow$
$\Gamma : Let\ \Delta\ body \Downarrow_L \Theta : z$
| *Bool*:
$\Gamma : Bool\ b \Downarrow_L \Gamma : Bool\ b$
| *IfThenElse*: $\llbracket$
$\Gamma : scrut \Downarrow_L \Delta : (Bool\ b)$;
$\Delta : (if\ b\ then\ e_1\ else\ e_2) \Downarrow_L \Theta : z$
$\rrbracket \Longrightarrow$
$\Gamma : (scrut\ ?\ e_1 : e_2) \Downarrow_L \Theta : z$

equivariance *reds*

nominal-inductive *reds*
avoids *Application*: *y*
⟨*proof*⟩

3.4.2 Example evaluations

lemma *eval-test*:
$[] : (Let\ [(x, Lam\ [y].\ Var\ y)]\ (Var\ x)) \Downarrow_{[]} [(x, Lam\ [y].\ Var\ y)] : (Lam\ [y].\ Var\ y)$
⟨*proof*⟩

lemma *eval-test2*:
$y \neq x \Longrightarrow n \neq y \Longrightarrow n \neq x \Longrightarrow [] : (Let\ [(x, Lam\ [y].\ Var\ y)]\ (App\ (Var\ x)\ x)) \Downarrow_{[]} [(x, Lam\ [y].\ Var\ y)] : (Lam\ [y].\ Var\ y)$
⟨*proof*⟩

3.4.3 Better introduction rules

This variant do not require freshness.

lemma *reds-ApplicationI*:
assumes $\Gamma : e \Downarrow_L \Delta : Lam\ [y].\ e'$
assumes $\Delta : e'[y::=x] \Downarrow_L \Theta : z$
shows $\Gamma : App\ e\ x \Downarrow_L \Theta : z$
⟨*proof*⟩

lemma *reds-SmartLet*: $\llbracket$
$atom\ `\ domA\ \Delta\ \sharp{*}\ (\Gamma, L)$;
$\Delta\ @\ \Gamma : body \Downarrow_L \Theta : z$
$\rrbracket \Longrightarrow$
$\Gamma : SmartLet\ \Delta\ body \Downarrow_L \Theta : z$
⟨*proof*⟩

A single rule for values

lemma *reds-isValI*:
  $isVal\ z \Longrightarrow \Gamma : z \Downarrow_L \Gamma : z$
$\langle proof \rangle$

3.4.4 Properties of the semantics

Heap entries are never removed.

lemma *reds-doesnt-forget*:
  $\Gamma : e \Downarrow_L \Delta : z \Longrightarrow domA\ \Gamma \subseteq domA\ \Delta$
$\langle proof \rangle$

Live variables are not added to the heap.

lemma *reds-avoids-live'*:
 assumes $\Gamma : e \Downarrow_L \Delta : z$
 shows $(domA\ \Delta - domA\ \Gamma) \cap set\ L = \{\}$
$\langle proof \rangle$

lemma *reds-avoids-live*:
  $[\![\Gamma : e \Downarrow_L \Delta : z;$
   $x \in set\ L;$
   $x \notin domA\ \Gamma$
  $]\!] \Longrightarrow x \notin domA\ \Delta$
$\langle proof \rangle$

Fresh variables either stay fresh or are added to the heap.

lemma *reds-fresh*: $[\![\Gamma : e \Downarrow_L \Delta : z;$
   $atom\ (x{::}var) \sharp (\Gamma, e)$
  $]\!] \Longrightarrow atom\ x \sharp (\Delta, z) \vee x \in (domA\ \Delta - set\ L)$
$\langle proof \rangle$

lemma *reds-fresh-fv*: $[\![\Gamma : e \Downarrow_L \Delta : z;$
   $x \in fv\ (\Delta, z) \wedge (x \notin domA\ \Delta \vee\ x \in set\ L)$
  $]\!] \Longrightarrow x \in fv\ (\Gamma, e)$
$\langle proof \rangle$

lemma *new-free-vars-on-heap*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  shows $fv\ (\Delta, z) - domA\ \Delta \subseteq fv\ (\Gamma, e) - domA\ \Gamma$
$\langle proof \rangle$

lemma *reds-pres-closed*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and $fv\ (\Gamma, e) \subseteq set\ L \cup domA\ \Gamma$
  shows $fv\ (\Delta, z) \subseteq set\ L \cup domA\ \Delta$
$\langle proof \rangle$

Reducing the set of variables to avoid is always possible.

lemma *reds-smaller-L*: $[\![\Gamma : e \Downarrow_L \Delta : z;$
  $set\ L' \subseteq set\ L$
$]\!] \Longrightarrow \Gamma : e \Downarrow_{L'} \Delta : z$
⟨*proof*⟩

Things are evaluated to a lambda expression, and the variable can be freely chose.

lemma *result-evaluated*:
  $\Gamma : e \Downarrow_L \Delta : z \Longrightarrow isVal\ z$
⟨*proof*⟩

lemma *result-evaluated-fresh*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  obtains $y\ e'$
  where $z = (Lam\ [y].\ e')$ **and** $atom\ y \sharp (c::'a::fs)$ | b **where** $z = Bool\ b$
⟨*proof*⟩

end

4 Denotational domain

4.1 Value

theory *Value*
  imports *HOLCF*
begin

4.1.1 The semantic domain for values and environments

domain $Value = Fn$ (**lazy** $Value \to Value$) | B (**lazy** $bool\ discr$)

fixrec $Fn\text{-}project :: Value \to Value \to Value$
 where $Fn\text{-}project\cdot(Fn\cdot f) = f$

abbreviation *Fn-project-abbr* (**infix** ‹↓*Fn*› *55*)
  where $f \downarrow Fn\ v \equiv Fn\text{-}project\cdot f\cdot v$

lemma [*simp*]:
  $\bot \downarrow Fn\ x = \bot$
  $(B\cdot b) \downarrow Fn\ x = \bot$
  ⟨*proof*⟩

fixrec $B\text{-}project :: Value \to Value \to Value \to Value$ **where**
  $B\text{-}project\cdot(B\cdot db)\cdot v_1\cdot v_2 = (if\ undiscr\ db\ then\ v_1\ else\ v_2)$

lemma [*simp*]:

$B\text{-}project\cdot(B\cdot(Discr\ b))\cdot v_1\cdot v_2 = (if\ b\ then\ v_1\ else\ v_2)$
$B\text{-}project\cdot\bot\cdot v_1\cdot v_2 = \bot$
$B\text{-}project\cdot(Fn\cdot f)\cdot v_1\cdot v_2 = \bot$
⟨*proof*⟩

A chain in the domain *Value* is either always bottom, or eventually *Fn* of another chain

lemma *Value-chainE*[*consumes 1*, *case-names bot B Fn*]:
assumes *chain Y*
obtains $Y = (\lambda\ \text{-}\ .\ \bot)$ |
$n\ b$ **where** $Y = (\lambda\ m.\ (if\ m < n\ then\ \bot\ else\ B\cdot b))$ |
$n\ Y'$ **where** $Y = (\lambda\ m.\ (if\ m < n\ then\ \bot\ else\ Fn\cdot(Y'\ (m-n))))$ $chain\ Y'$
⟨*proof*⟩

end

4.2 Value-Nominal

theory *Value−Nominal*
imports *Value Nominal−Utils Nominal−HOLCF*
begin

Values are pure, i.e. contain no variables.

instantiation *Value* :: *pure*
begin
definition $p \bullet (v{::}Value) = v$
instance
⟨*proof*⟩
end

instance *Value* :: *pcpo-pt*
⟨*proof*⟩

end

5 Denotational semantics

5.1 Iterative

theory *Iterative*
imports *Env−HOLCF*
begin

A setup for defining a fixed point of mutual recursive environments iteratively

locale *iterative* =
  fixes ϱ :: $'a$::*type* $\Rightarrow$ $'b$::*pcpo*
   and *e1* :: $('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b)$
   and *e2* :: $('a \Rightarrow 'b) \rightarrow 'b$
   and S :: $'a$ *set* **and** x :: $'a$
  assumes *ne*:$x \notin S$
begin
  abbreviation $L == (\Lambda\ \varrho'.\ (\varrho\ {++}_S\ e1 \cdot \varrho')(x := e2 \cdot \varrho'))$
  abbreviation $H == (\lambda\ \varrho'.\ \Lambda\ \varrho''.\ \varrho'\ {++}_S\ e1 \cdot \varrho'')$
  abbreviation $R == (\Lambda\ \varrho'.\ (\varrho\ {++}_S\ (fix \cdot (H\ \varrho')))(x := e2 \cdot \varrho'))$
  abbreviation $R' == (\Lambda\ \varrho'.\ (\varrho\ {++}_S\ (fix \cdot (H\ \varrho')))(x := e2 \cdot (fix \cdot (H\ \varrho'))))$

  lemma *split-x*:
    fixes y
    obtains $y = x$ **and** $y \notin S$ | $y \in S$ **and** $y \neq x$ | $y \notin S$ **and** $y \neq x$ ⟨*proof*⟩
  lemmas *below* = *fun-belowI*[*OF split-x*, **where** $y1 = \lambda x.\ x$]
  lemmas *eq* = *ext*[*OF split-x*, **where** $y1 = \lambda x.\ x$]

  lemma *lookup-fix*[*simp*]:
    fixes y **and** F :: $('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b)$
    shows $(fix \cdot F)\ y = (F \cdot (fix \cdot F))\ y$
    ⟨*proof*⟩

  lemma *R-S*: $\bigwedge y.\ y \in S \Longrightarrow (fix \cdot R)\ y = (e1 \cdot (fix \cdot (H\ (fix \cdot R))))\ y$
    ⟨*proof*⟩

  lemma *R'-S*: $\bigwedge y.\ y \in S \Longrightarrow (fix \cdot R')\ y = (e1 \cdot (fix \cdot (H\ (fix \cdot R'))))\ y$
    ⟨*proof*⟩

  lemma *HR-is-R*[*simp*]: $fix{\cdot}(H\ (fix \cdot R)) = fix \cdot R$
    ⟨*proof*⟩

  lemma *HR'-is-R'*[*simp*]: $fix \cdot (H\ (fix \cdot R')) = fix \cdot R'$
    ⟨*proof*⟩

  lemma *H-noop*:
    fixes $\varrho'\ \varrho''$
    assumes $\bigwedge y.\ y \in S \Longrightarrow y \neq x \Longrightarrow (e1 \cdot \varrho'')\ y \sqsubseteq \varrho'\ y$
    shows $H\ \varrho' \cdot \varrho'' \sqsubseteq \varrho'$
      ⟨*proof*⟩

lemma *HL-is-L*[*simp*]: $\mathit{fix} \cdot (H\ (\mathit{fix} \cdot L)) = \mathit{fix} \cdot L$
⟨*proof*⟩

lemma *iterative-override-on*:
shows $\mathit{fix} \cdot L = \mathit{fix} \cdot R$
⟨*proof*⟩

lemma *iterative-override-on'*:
shows $\mathit{fix} \cdot L = \mathit{fix} \cdot\ R'$
⟨*proof*⟩
end

end

5.2 HasESem

theory *HasESem*
imports *Nominal−HOLCF Env−HOLCF*
begin

A local to work abstract in the expression type and semantics.

locale *has-ESem* =
fixes *ESem* :: *'exp*::*pt* ⇒ (*'var*::*at-base* ⇒ *'value*) → *'value*::{*pure*,*pcpo*}
begin
abbreviation *ESem-syn* (‹⟦ - ⟧_› [*0*,*0*] *110*) **where** $[\![e]\!]_{\varrho} \equiv \mathit{ESem}\ e \cdot \varrho$
end

locale *has-ignore-fresh-ESem* = *has-ESem* +
assumes *fv-supp*: *supp e* = *atom* ‘ (*fv e* :: *'b set*)
assumes *ESem-considers-fv*: $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho\ f|`\ (fv\ e)}$

end

5.3 HeapSemantics

theory *HeapSemantics*
imports *EvalHeap AList−Utils−Nominal HasESem Iterative Env−Nominal*
begin

5.3.1 A locale for heap semantics, abstract in the expression semantics

context *has-ESem*
begin

abbreviation *EvalHeapSem-syn* (‹⟦ - ⟧_› [*0*,*0*] *110*)
where *EvalHeapSem-syn* $\Gamma\ \varrho \equiv \mathit{evalHeap}\ \Gamma\ (\lambda\ e.\ [\![e]\!]_{\varrho})$

definition
HSem :: $('var \times 'exp)$ *list* $\Rightarrow ('var \Rightarrow 'value) \rightarrow ('var \Rightarrow 'value)$
where *HSem* $\Gamma = (\Lambda\ \varrho\ .\ (\mu\ \varrho'.\ \varrho\ {+}{+}_{domA\ \Gamma}\ [\![\Gamma]\!]_{\varrho'}))$

abbreviation *HSem-syn* (‹⦃ - ⦄-› [0,60] 60)
where $⦃\Gamma⦄\varrho \equiv$ *HSem* $\Gamma \cdot \varrho$

lemma *HSem-def'*: $⦃\Gamma⦄\varrho = (\mu\ \varrho'.\ \varrho\ {+}{+}_{domA\ \Gamma}\ [\![\Gamma]\!]_{\varrho'})$
⟨*proof*⟩

5.3.2 Induction and other lemmas about *HSem*

lemma *HSem-ind*:
assumes *adm P*
assumes $P\ \bot$
assumes *step*: $\bigwedge\ \varrho'.\ P\ \varrho' \Longrightarrow\ P\ (\varrho\ {+}{+}_{domA\ \Gamma}\ [\![\Gamma]\!]_{\varrho'})$
shows $P\ (⦃\Gamma⦄\varrho)$
⟨*proof*⟩

lemma *HSem-below*:
assumes *rho*: $\bigwedge x.\ x \notin domA\ h \Longrightarrow \varrho\ x \sqsubseteq r\ x$
assumes *h*: $\bigwedge x.\ x \in domA\ h \Longrightarrow [\![the\ (map\text{-}of\ h\ x)]\!]_r \sqsubseteq r\ x$
shows $⦃h⦄\varrho \sqsubseteq r$
⟨*proof*⟩

lemma *HSem-bot-below*:
assumes *h*: $\bigwedge x.\ x \in domA\ h \Longrightarrow [\![the\ (map\text{-}of\ h\ x)]\!]_r \sqsubseteq r\ x$
shows $⦃h⦄\bot \sqsubseteq r$
⟨*proof*⟩

lemma *HSem-bot-ind*:
assumes *adm P*
assumes $P\ \bot$
assumes *step*: $\bigwedge\ \varrho'.\ P\ \varrho' \Longrightarrow P\ ([\![\ \Gamma\ ]\!]_{\varrho'})$
shows $P\ (⦃\Gamma⦄\bot)$
⟨*proof*⟩

lemma *parallel-HSem-ind*:
assumes *adm* $(\lambda\varrho'.\ P\ (fst\ \varrho')\ (snd\ \varrho'))$
assumes $P\ \bot\ \bot$
assumes *step*: $\bigwedge y\ z.\ P\ y\ z \Longrightarrow$
$P\ (\varrho_1\ {+}{+}_{domA\ \Gamma_1}\ [\![\Gamma_1]\!]y)\ (\varrho_2\ {+}{+}_{domA\ \Gamma_2}\ [\![\Gamma_2]\!]z)$
shows $P\ (⦃\Gamma_1⦄\varrho_1)\ (⦃\Gamma_2⦄\varrho_2)$
⟨*proof*⟩

lemma *HSem-eq*:
shows $⦃\Gamma⦄\varrho = \varrho\ {+}{+}_{domA\ \Gamma}\ [\![\Gamma]\!]_{⦃\Gamma⦄\varrho}$
⟨*proof*⟩

lemma *HSem-bot-eq*:
  shows $\{\!|\Gamma|\!\}\bot = [\![\Gamma]\!]_{\{\!|\Gamma|\!\}\bot}$
  ⟨*proof*⟩

lemma *lookup-HSem-other*:
  assumes $y \notin domA\ h$
  shows $(\{\!|h|\!\}\varrho)\ y = \varrho\ y$
  ⟨*proof*⟩

lemma *lookup-HSem-heap*:
  assumes $y \in domA\ h$
  shows $(\{\!|h|\!\}\varrho)\ y = [\![\ the\ (map\text{-}of\ h\ y)\ ]\!]_{\{\!|h|\!\}\varrho}$
  ⟨*proof*⟩

lemma *HSem-edom-subset*: $edom\ (\{\!|\Gamma|\!\}\varrho) \subseteq edom\ \varrho \cup domA\ \Gamma$
  ⟨*proof*⟩

lemma (**in** $-$) *env-restr-override-onI*:$-S2 \subseteq S \Longrightarrow env\text{-}restr\ S\ \varrho 1\ ++_{S2}\ \varrho 2 = \varrho 1\ ++_{S2}\ \varrho 2$
  ⟨*proof*⟩

lemma *HSem-restr*:
  $\{\!|h|\!\}(\varrho\ f|`\ (-\ domA\ h)) = \{\!|h|\!\}\varrho$
  ⟨*proof*⟩

lemma *HSem-restr-cong*:
  assumes $\varrho\ f|`\ (-\ domA\ h) = \varrho'\ f|`\ (-\ domA\ h)$
  shows $\{\!|h|\!\}\varrho = \{\!|h|\!\}\varrho'$
  ⟨*proof*⟩

lemma *HSem-restr-cong-below*:
  assumes $\varrho\ f|`\ (-\ domA\ h) \sqsubseteq \varrho'\ f|`\ (-\ domA\ h)$
  shows $\{\!|h|\!\}\varrho \sqsubseteq \{\!|h|\!\}\varrho'$
  ⟨*proof*⟩

lemma *HSem-reorder*:
  assumes $map\text{-}of\ \Gamma = map\text{-}of\ \Delta$
  shows $\{\!|\Gamma|\!\}\varrho = \{\!|\Delta|\!\}\varrho$
⟨*proof*⟩

lemma *HSem-reorder-head*:
  assumes $x \neq y$
  shows $\{\!|(x,e1)\#(y,e2)\#\Gamma|\!\}\varrho = \{\!|(y,e2)\#(x,e1)\#\Gamma|\!\}\varrho$
⟨*proof*⟩

lemma *HSem-reorder-head-append*:
  assumes $x \notin domA\ \Gamma$
  shows $\{\!|(x,e)\#\Gamma@\Delta|\!\}\varrho = \{\!|\Gamma\ @\ ((x,e)\#\Delta)|\!\}\varrho$
⟨*proof*⟩

lemma *env-restr-HSem*:
  assumes $domA\ \Gamma \cap S = \{\}$
  shows $(\{\!|\ \Gamma\ |\!\}\varrho)\ f|\text{`}\ S = \varrho\ f|\text{`}\ S$
⟨*proof*⟩

lemma *env-restr-HSem-noop*:
  assumes $domA\ \Gamma \cap edom\ \varrho = \{\}$
  shows $(\{\!|\ \Gamma\ |\!\}\varrho)\ f|\text{`}\ edom\ \varrho = \varrho$
  ⟨*proof*⟩

lemma *HSem-Nil*[*simp*]: $\{\!|[]|\!\}\varrho = \varrho$
  ⟨*proof*⟩

5.3.3 Substitution

lemma *HSem-subst-exp*:
  assumes $\bigwedge\varrho'.\ [\![\ e\ ]\!]_{\varrho'} = [\![\ e'\ ]\!]_{\varrho'}$
  shows $\{\!|(x,\ e)\ \#\ \Gamma|\!\}\varrho = \{\!|(x,\ e')\ \#\ \Gamma|\!\}\varrho$
  ⟨*proof*⟩

lemma *HSem-subst-expr-below*:
  assumes *below*: $[\![\ e1\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho} \sqsubseteq [\![\ e2\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho}$
  shows $\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho \sqsubseteq \{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho$
  ⟨*proof*⟩

lemma *HSem-subst-expr*:
  assumes *below1*: $[\![\ e1\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho} \sqsubseteq [\![\ e2\ ]\!]_{\{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho}$
  assumes *below2*: $[\![\ e2\ ]\!]_{\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho} \sqsubseteq [\![\ e1\ ]\!]_{\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho}$
  shows $\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho = \{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho$
  ⟨*proof*⟩

5.3.4 Re-calculating the semantics of the heap is idempotent

lemma *HSem-redo*:
  shows $\{\!|\Gamma|\!\}(\{\!|\Gamma\ @\ \Delta|\!\}\varrho)\ f|\text{`}\ (edom\ \varrho \cup domA\ \Delta) = \{\!|\Gamma\ @\ \Delta|\!\}\varrho$ (**is** *?LHS* = *?RHS*)
⟨*proof*⟩

5.3.5 Iterative definition of the heap semantics

lemma *iterative-HSem*:
  assumes $x \notin domA\ \Gamma$
  shows $\{\!|(x,e)\ \#\ \Gamma|\!\}\varrho = (\mu\ \varrho'.\ (\varrho\ {+}{+}_{domA\ \Gamma}\ (\{\!|\Gamma|\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'}))$
⟨*proof*⟩

lemma *iterative-HSem'*:
  assumes $x \notin domA\ \Gamma$
  shows $(\mu\ \varrho'.\ (\varrho\ {+}{+}_{domA\ \Gamma}\ (\{\!|\Gamma|\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'}))$
    $= (\mu\ \varrho'.\ (\varrho\ {+}{+}_{domA\ \Gamma}\ (\{\!|\Gamma|\!\}\varrho'))(\ x := [\![e]\!]_{\{\!|\Gamma|\!\}\varrho'}))$
⟨*proof*⟩

5.3.6 Fresh variables on the heap are irrelevant

lemma *HSem-ignores-fresh-restr'*:
  assumes $fv\ \Gamma \subseteq S$
  assumes $\bigwedge x\ \varrho.\ x \in domA\ \Gamma \Longrightarrow [\![\ the\ (map\text{-}of\ \Gamma\ x)\ ]\!]_{\varrho} = [\![\ the\ (map\text{-}of\ \Gamma\ x)\ ]\!]_{\varrho\ f|`\ (fv\ (the\ (map\text{-}of\ \Gamma\ x)))}$
  shows $(\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ S = \{\!\!\{\Gamma\}\!\!\}\varrho\ f|`\ S$
⟨*proof*⟩
end

5.3.7 Freshness

context *has-ignore-fresh-ESem* **begin**

lemma *ESem-fresh-cong*:
  assumes $\varrho\ f|`\ (fv\ e) = \varrho'\ f|`\ (fv\ e)$
  shows $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho'}$
⟨*proof*⟩

lemma *ESem-fresh-cong-subset*:
  assumes $fv\ e \subseteq S$
  assumes $\varrho\ f|`\ S = \varrho'\ f|`\ S$
  shows $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho'}$
⟨*proof*⟩

lemma *ESem-fresh-cong-below*:
  assumes $\varrho\ f|`\ (fv\ e) \sqsubseteq \varrho'\ f|`\ (fv\ e)$
  shows $[\![\ e\ ]\!]_{\varrho} \sqsubseteq [\![\ e\ ]\!]_{\varrho'}$
⟨*proof*⟩

lemma *ESem-fresh-cong-below-subset*:
  assumes $fv\ e \subseteq S$
  assumes $\varrho\ f|`\ S \sqsubseteq \varrho'\ f|`\ S$
  shows $[\![\ e\ ]\!]_{\varrho} \sqsubseteq [\![\ e\ ]\!]_{\varrho'}$
⟨*proof*⟩

lemma *ESem-ignores-fresh-restr*:
  assumes $atom\ `\ S\ \sharp*\ e$
  shows $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho\ f|`\ (-\ S)}$
⟨*proof*⟩

lemma *ESem-ignores-fresh-restr'*:
  assumes $atom\ `\ (edom\ \varrho - S)\ \sharp*\ e$
  shows $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho\ f|`\ S}$
⟨*proof*⟩

lemma *HSem-ignores-fresh-restr''*:
  assumes $fv\ \Gamma \subseteq S$
  shows $(\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ S = \{\!\!\{\Gamma\}\!\!\}\varrho\ f|`\ S$
⟨*proof*⟩

lemma *HSem-ignores-fresh-restr*:
  assumes $atom\ `\ S\ \sharp{*}\ \Gamma$
  shows $(\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ (-\ S) = \{\!\!\{\Gamma\}\!\!\}\varrho\ f|`\ (-\ S)$
$\langle proof \rangle$

lemma *HSem-fresh-cong-below*:
  assumes $\varrho\ f|`\ ((S \cup fv\ \Gamma) - domA\ \Gamma) \sqsubseteq \varrho'\ f|`\ ((S \cup fv\ \Gamma) - domA\ \Gamma)$
  shows $(\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ S \sqsubseteq (\{\!\!\{\Gamma\}\!\!\}\varrho')\ f|`\ S$
$\langle proof \rangle$

lemma *HSem-fresh-cong*:
  assumes $\varrho\ f|`\ ((S \cup fv\ \Gamma) - domA\ \Gamma) = \varrho'\ f|`\ ((S \cup fv\ \Gamma) - domA\ \Gamma)$
  shows $(\{\!\!\{\Gamma\}\!\!\}\varrho)\ f|`\ S = (\{\!\!\{\Gamma\}\!\!\}\varrho')\ f|`\ S$
$\langle proof \rangle$

5.3.8 Adding a fresh variable to a heap does not affect its semantics

lemma *HSem-add-fresh'*:
  assumes *fresh*: $atom\ x\ \sharp\ \Gamma$
  assumes $x \notin edom\ \varrho$
  assumes *step*: $\bigwedge e\ \varrho'.\ e \in snd\ `\ set\ \Gamma \Longrightarrow [\![\ e\ ]\!]_{\varrho'} = [\![\ e\ ]\!]_{env\text{-}delete\ x\ \varrho'}$
  shows $env\text{-}delete\ x\ (\{\!\!\{(x,\ e)\ \#\ \Gamma\}\!\!\}\varrho) = \{\!\!\{\Gamma\}\!\!\}\varrho$
$\langle proof \rangle$

lemma *HSem-add-fresh*:
  assumes $atom\ x\ \sharp\ \Gamma$
  assumes $x \notin edom\ \varrho$
  shows $env\text{-}delete\ x\ (\{\!\!\{(x,\ e)\ \#\ \Gamma\}\!\!\}\varrho) = \{\!\!\{\Gamma\}\!\!\}\varrho$
$\langle proof \rangle$

5.3.9 Mutual recursion with fresh variables

lemma *HSem-subset-below*:
  assumes *fresh*: $atom\ `\ domA\ \Gamma\ \sharp{*}\ \Delta$
  shows $\{\!\!\{\Delta\}\!\!\}(\varrho\ f|`\ (-\ domA\ \Gamma)) \sqsubseteq (\{\!\!\{\Delta @ \Gamma\}\!\!\}\varrho)\ \ f|`\ (-\ domA\ \Gamma)$
$\langle proof \rangle$

In the following lemma we show that the semantics of fresh variables can be be calculated together with the presently bound variables, or separately.

lemma *HSem-merge*:
  assumes *fresh*: $atom\ `\ domA\ \Gamma\ \sharp{*}\ \Delta$
  shows $\{\!\!\{\Gamma\}\!\!\}\{\!\!\{\Delta\}\!\!\}\varrho = \{\!\!\{\Gamma @ \Delta\}\!\!\}\varrho$
$\langle proof \rangle$
end

5.3.10 Parallel induction

lemma *parallel-HSem-ind-different-ESem*:

assumes $adm\ (\lambda\varrho'.\ P\ (fst\ \varrho')\ (snd\ \varrho'))$
assumes $P \perp \perp$
assumes $\bigwedge y\ z.\ P\ y\ z \Longrightarrow P\ (\varrho\ {++}_{domA\ h}\ evalHeap\ h\ (\lambda e.\ ESem1\ e \cdot y))\ (\varrho 2\ {++}_{domA\ h2}\ evalHeap\ h2\ (\lambda e.\ ESem2\ e \cdot z))$
shows $P\ (has\text{-}ESem.HSem\ ESem1\ h{\cdot}\varrho)\ (has\text{-}ESem.HSem\ ESem2\ h2{\cdot}\varrho 2)$
⟨*proof*⟩

5.3.11 Congruence rule

lemma *HSem-cong*[*fundef-cong*]:
$[\![\ (\bigwedge\ e.\ e \in snd\ `\ set\ heap2 \Longrightarrow ESem1\ e = ESem2\ e);\ heap1 = heap2\ ]\!]$
$\Longrightarrow has\text{-}ESem.HSem\ ESem1\ heap1 = has\text{-}ESem.HSem\ ESem2\ heap2$
⟨*proof*⟩

5.3.12 Equivariance of the heap semantics

lemma *HSem-eqvt*[*eqvt*]:
$\pi \bullet has\text{-}ESem.HSem\ ESem\ \Gamma = has\text{-}ESem.HSem\ (\pi \bullet ESem)\ (\pi \bullet \Gamma)$
⟨*proof*⟩

end

5.4 AbstractDenotational

theory *AbstractDenotational*
imports *HeapSemantics Terms*
begin

5.4.1 The denotational semantics for expressions

Because we need to define two semantics later on, we are abstract in the actual domain.

locale *semantic-domain* =
fixes $Fn :: ('Value \to 'Value) \to ('Value::\{pcpo\text{-}pt,pure\})$
fixes $Fn\text{-}project :: 'Value \to ('Value \to 'Value)$
fixes $B :: bool\ discr \to 'Value$
fixes $B\text{-}project :: 'Value \to 'Value \to 'Value \to 'Value$
fixes $tick :: 'Value \to 'Value$
begin

nominal-function
$ESem :: exp \Rightarrow (var \Rightarrow 'Value) \to 'Value$
where
$ESem\ (Lam\ [x].\ e) = (\Lambda\ \varrho.\ tick{\cdot}(Fn{\cdot}(\Lambda\ v.\ ESem\ e{\cdot}((\varrho\ f|`\ fv\ (Lam\ [x].\ e))(x := v)))))$
$|\ ESem\ (App\ e\ x) = (\Lambda\ \varrho.\ tick{\cdot}(Fn\text{-}project{\cdot}(ESem\ e{\cdot}\varrho){\cdot}(\varrho\ x)))$
$|\ ESem\ (Var\ x) = (\Lambda\ \varrho.\ tick{\cdot}(\varrho\ x))$
$|\ ESem\ (Let\ as\ body) = (\Lambda\ \varrho.\ tick{\cdot}(ESem\ body{\cdot}(has\text{-}ESem.HSem\ ESem\ as{\cdot}(\varrho\ f|`\ fv\ (Let\ as\ body)))))$
$|\ ESem\ (Bool\ b) = (\Lambda\ \varrho.\ tick{\cdot}(B{\cdot}(Discr\ b)))$

| *ESem* (*scrut ? e1 : e2*) = (Λ ϱ. *tick*·((*B-project*·(*ESem scrut*·ϱ))·(*ESem e1*·ϱ)·(*ESem e2*·ϱ)))
⟨*proof*⟩

nominal-termination (**in** *semantic-domain*) (*no-eqvt*) ⟨*proof*⟩

sublocale *has-ESem ESem*⟨*proof*⟩

notation *ESem-syn* (‹⟦ - ⟧-› [*60*,*60*] *60*)
notation *EvalHeapSem-syn* (‹⟦ - ⟧-› [*0*,*0*] *110*)
notation *HSem-syn* (‹⦃-⦄-› [*60*,*60*] *60*)
abbreviation *AHSem-bot* (‹⦃-⦄› [*60*] *60*) **where** ⦃Γ⦄ ≡ ⦃Γ⦄⊥

end
end

5.5 Abstract-Denotational-Props

theory *Abstract−Denotational−Props*
imports *AbstractDenotational Substitution*
begin

context *semantic-domain*
begin

5.5.1 The semantics ignores fresh variables

lemma *ESem-considers-fv′*: $⟦ e ⟧_{ϱ} = ⟦ e ⟧_{ϱ\ f|`\ (fv\ e)}$
⟨*proof*⟩

sublocale *has-ignore-fresh-ESem ESem*
⟨*proof*⟩

5.5.2 Nicer equations for ESem, without freshness requirements

lemma *ESem-Lam*[*simp*]: $⟦ Lam\ [x].\ e ⟧_{ϱ} = tick \cdot (Fn \cdot (Λ\ v.\ ⟦ e ⟧_{ϱ(x := v)}))$
⟨*proof*⟩
declare *ESem.simps*(*1*)[*simp del*]

lemma *ESem-Let*[*simp*]: $⟦Let\ as\ body⟧_{ϱ} = tick \cdot (⟦body⟧_{⦃as⦄ϱ})$
⟨*proof*⟩
declare *ESem.simps*(*4*)[*simp del*]

5.5.3 Denotation of Substitution

lemma *ESem-subst-same*: $ϱ\ x = ϱ\ y \Longrightarrow ⟦ e ⟧_{ϱ} = ⟦ e[x::= y] ⟧_{ϱ}$
and
$ϱ\ x = ϱ\ y \Longrightarrow (⟦ as ⟧_{ϱ}) = ⟦ as[x::h=y] ⟧_{ϱ}$
⟨*proof*⟩

lemma *ESem-subst*:
  shows $\llbracket\ e\ \rrbracket_{\sigma(x\ :=\ \sigma\ y)} = \llbracket\ e[x::=\ y]\ \rrbracket_{\sigma}$
$\langle proof \rangle$

end

end

5.6 Denotational

theory *Denotational*
  imports *Abstract−Denotational−Props Value−Nominal*
begin

This is the actual denotational semantics as found in [Lau93].

interpretation *semantic-domain Fn Fn-project B B-project* $(\Lambda\ x.\ x)\langle proof \rangle$

notation *ESem-syn* (‹⟦ - ⟧-› [*60*,*60*] *60*)
notation *EvalHeapSem-syn* (‹⟦ - ⟧-› [*0*,*0*] *110*)
notation *HSem-syn* (‹⦃-⦄-› [*60*,*60*] *60*)
notation *AHSem-bot* (‹⦃-⦄› [*60*] *60*)

lemma *ESem-simps-as-defined*:
  $\llbracket\ Lam\ [x].\ e\ \rrbracket_{\varrho} = \ Fn\cdot(\Lambda\ v.\ \llbracket\ e\ \rrbracket_{(\varrho\ f|`\ (fv\ (Lam\ [x].e)))(x\ :=\ v)})$
  $\llbracket\ App\ e\ x\ \rrbracket_{\varrho} \ = \ \llbracket\ e\ \rrbracket_{\varrho} \downarrow Fn\ \varrho\ x$
  $\llbracket\ Var\ x\ \rrbracket_{\varrho} \ = \ \varrho\ \ x$
  $\llbracket\ Bool\ b\ \rrbracket_{\varrho} \ = \ B\cdot(Discr\ b)$
  $\llbracket\ (scrut\ ?\ e_1\ :\ e_2)\ \rrbracket_{\varrho} = B\text{-}project\cdot(\llbracket\ scrut\ \rrbracket_{\varrho})\cdot(\llbracket\ e_1\ \rrbracket_{\varrho})\cdot(\llbracket\ e_2\ \rrbracket_{\varrho})$
  $\llbracket\ Let\ \Gamma\ body\ \rrbracket_{\varrho} = \llbracket body \rrbracket_{\{\!\!\{\Gamma\}\!\!\}(\varrho\ f|`\ fv\ (Let\ \Gamma\ body))}$
  $\langle proof \rangle$

lemma *ESem-simps*:
  $\llbracket\ Lam\ [x].\ e\ \rrbracket_{\varrho} = \ Fn\cdot(\Lambda\ v.\ \llbracket\ e\ \rrbracket_{\varrho(x\ :=\ v)})$
  $\llbracket\ App\ e\ x\ \rrbracket_{\varrho} \ = \ \llbracket\ e\ \rrbracket_{\varrho} \downarrow Fn\ \varrho\ x$
  $\llbracket\ Var\ x\ \rrbracket_{\varrho} \ = \ \varrho\ \ x$
  $\llbracket\ Bool\ b\ \rrbracket_{\varrho} \ = \ B\cdot(Discr\ b)$
  $\llbracket\ (scrut\ ?\ e_1\ :\ e_2)\ \rrbracket_{\varrho} = B\text{-}project\cdot(\llbracket\ scrut\ \rrbracket_{\varrho})\cdot(\llbracket\ e_1\ \rrbracket_{\varrho})\cdot(\llbracket\ e_2\ \rrbracket_{\varrho})$
  $\llbracket\ Let\ \Gamma\ body\ \rrbracket_{\varrho} = \llbracket body \rrbracket_{\{\!\!\{\Gamma\}\!\!\}\varrho}$
  $\langle proof \rangle\langle proof \rangle\langle proof \rangle$

end

6 Resourced denotational domain

6.1 C

theory *C*
imports *HOLCF Mono−Nat−Fun*
begin

default-sort *cpo*

The initial solution to the domain equation $C = C_\perp$, i.e. the completion of the natural numbers.

domain $C = C$ (**lazy** C)

lemma *below-C*: $x \sqsubseteq C \cdot x$
⟨*proof*⟩

definition *Cinf* (‹C^∞›) **where** $C^\infty = \mathit{fix} \cdot C$

lemma *C-Cinf*[*simp*]: $C \cdot C^\infty = C^\infty$ ⟨*proof*⟩

abbreviation *Cpow* (‹C^-›) **where** $C^n \equiv \mathit{iterate}\ n \cdot C \cdot \perp$

lemma *C-below-C*[*simp*]: $(C^i \sqsubseteq C^j) \longleftrightarrow i \leq j$
⟨*proof*⟩

lemma *below-Cinf*[*simp*]: $r \sqsubseteq C^\infty$
⟨*proof*⟩

lemma *C-eq-Cinf*[*simp*]: $C^i \neq C^\infty$
⟨*proof*⟩

lemma *Cinf-eq-C*[*simp*]: $C^\infty = C \cdot r \longleftrightarrow C^\infty = r$
⟨*proof*⟩

lemma *C-eq-C*[*simp*]: $(C^i = C^j) \longleftrightarrow i = j$
⟨*proof*⟩

lemma *case-of-C-below*: $(\mathit{case}\ r\ \mathit{of}\ C \cdot y \Rightarrow x) \sqsubseteq x$
⟨*proof*⟩

lemma *C-case-below*: $C\text{-}case \cdot f \sqsubseteq f$
⟨*proof*⟩

lemma *C-case-bot*[*simp*]: $C\text{-}case \cdot \perp = \perp$
⟨*proof*⟩

lemma *C-case-cong*:

assumes $\bigwedge r'.\ r = C{\cdot}r' \Longrightarrow f{\cdot}r' = g{\cdot}r'$
shows $C\text{-}case{\cdot}f{\cdot}r = C\text{-}case{\cdot}g{\cdot}r$
⟨*proof*⟩

lemma *C-cases*:
obtains n **where** $r = C^n \mid r = C^\infty$
⟨*proof*⟩

lemma *C-case-Cinf*[*simp*]: $C\text{-}case \cdot f \cdot C^\infty = f \cdot C^\infty$
⟨*proof*⟩

end

6.2 C-Meet

theory $C{-}Meet$
imports C $HOLCF{-}Meet$
begin

instantiation C :: *Finite-Meet-cpo* **begin**
fixrec *C-meet* :: $C \to C \to C$
where $C\text{-}meet{\cdot}(C{\cdot}a){\cdot}(C{\cdot}b) = C{\cdot}(C\text{-}meet{\cdot}a{\cdot}b)$

lemma[*simp*]: $C\text{-}meet{\cdot}\bot{\cdot}y = \bot$ $C\text{-}meet{\cdot}x{\cdot}\bot = \bot$ ⟨*proof*⟩

instance
⟨*proof*⟩
end

lemma *C-meet-is-meet*: $(z \sqsubseteq C\text{-}meet{\cdot}x{\cdot}y) = (z \sqsubseteq x \wedge z \sqsubseteq y)$
⟨*proof*⟩

instance C :: *cont-binary-meet*
⟨*proof*⟩

lemma [*simp*]: $C{\cdot}r \sqcap r = r$
⟨*proof*⟩

lemma [*simp*]: $r \sqcap C{\cdot}r = r$
⟨*proof*⟩

lemma [*simp*]: $C{\cdot}r \sqcap C{\cdot}r' = C{\cdot}(r \sqcap r')$
⟨*proof*⟩

end

6.3 C-restr

theory $C{-}restr$
imports C $C{-}Meet$ $HOLCF{-}Utils$
begin

6.3.1 The demand of a C-function

The demand is the least amount of resources required to produce a non-bottom element, if at all.

definition *demand* :: $(C \to {'a}{::}pcpo) \Rightarrow C$ **where**
$demand\ f = (if\ f{\cdot}C^{\infty} \neq \bot\ then\ C^{(LEAST\ n.\ f{\cdot}C^{n} \neq \bot)}\ else\ C^{\infty})$

Because of continuity, a non-bottom value can always be obtained with finite resources.

lemma *finite-resources-suffice*:
assumes $f{\cdot}C^{\infty} \neq \bot$
obtains n **where** $f{\cdot}C^{n} \neq \bot$
⟨*proof*⟩

Because of monotonicity, a non-bottom value can always be obtained with more resources.

lemma *more-resources-suffice*:
assumes $f{\cdot}r \neq \bot$ **and** $r \sqsubseteq r'$
shows $f{\cdot}r' \neq \bot$
⟨*proof*⟩

lemma *infinite-resources-suffice*:
shows $f{\cdot}r \neq \bot \Longrightarrow f{\cdot}C^{\infty} \neq \bot$
⟨*proof*⟩

lemma *demand-suffices*:
assumes $f{\cdot}C^{\infty} \neq \bot$
shows $f{\cdot}(demand\ f) \neq \bot$
⟨*proof*⟩

lemma *not-bot-demand*:
$f{\cdot}r \neq \bot \longleftrightarrow demand\ f \neq C^{\infty} \wedge demand\ f \sqsubseteq r$
⟨*proof*⟩

lemma *infinity-bot-demand*:
$f{\cdot}C^{\infty} = \bot \longleftrightarrow demand\ f = C^{\infty}$
⟨*proof*⟩

lemma *demand-suffices'*:
assumes $demand\ f = C^{n}$
shows $f{\cdot}(demand\ f) \neq \bot$
⟨*proof*⟩

lemma *demand-Suc-Least*:
  assumes [*simp*]: $f \cdot \bot = \bot$
  assumes *demand* $f \neq C^{\infty}$
  shows *demand* $f = C^{(Suc\ (LEAST\ n.\ f \cdot C^{Suc\ n} \neq \bot))}$
⟨*proof*⟩

lemma *demand-C-case*[*simp*]: *demand* $(C\text{-}case \cdot f) = C \cdot (demand\ f)$
⟨*proof*⟩

lemma *demand-contravariant*:
  assumes $f \sqsubseteq g$
  shows *demand* $g \sqsubseteq$ *demand* f
⟨*proof*⟩

6.3.2 Restricting functions with domain C

fixrec *C-restr* :: $C \rightarrow (C \rightarrow 'a::pcpo) \rightarrow (C \rightarrow 'a)$
  where $C\text{-}restr \cdot r \cdot f \cdot r' = (f \cdot (r \sqcap r'))$

abbreviation *C-restr-syn* :: $(C \rightarrow 'a::pcpo) \Rightarrow C \Rightarrow (C \rightarrow 'a)$ (‹-\|-› [*111*,*110*] *110*)
  where $f|_r \equiv C\text{-}restr \cdot r \cdot f$

lemma [*simp*]: $\bot|_r = \bot$ ⟨*proof*⟩
lemma [*simp*]: $f \cdot \bot = \bot \Longrightarrow f|_\bot = \bot$ ⟨*proof*⟩

lemma *C-restr-C-restr*[*simp*]: $(v|_{r'})|_r = v|_{(r' \sqcap r)}$
  ⟨*proof*⟩

lemma *C-restr-eqD*:
  assumes $f|_r = g|_r$
  assumes $r' \sqsubseteq r$
  shows $f \cdot r' = g \cdot r'$
⟨*proof*⟩

lemma *C-restr-eq-lower*:
  assumes $f|_r = g|_r$
  assumes $r' \sqsubseteq r$
  shows $f|_{r'} = g|_{r'}$
⟨*proof*⟩

lemma *C-restr-below*[*intro*, *simp*]:
  $x|_r \sqsubseteq x$
  ⟨*proof*⟩

lemma *C-restr-below-cong*:
  $(\bigwedge r'.\ r' \sqsubseteq r \Longrightarrow f \cdot r' \sqsubseteq g \cdot r') \Longrightarrow f|_r \sqsubseteq g|_r$
  ⟨*proof*⟩

lemma *C-restr-cong*:
$(\bigwedge r'. \; r' \sqsubseteq r \Longrightarrow f \cdot r' = g \cdot r') \Longrightarrow f|_r = g|_r$
⟨*proof*⟩

lemma *C-restr-C-cong*:
$(\bigwedge r'. \; r' \sqsubseteq r \Longrightarrow f \cdot (C \cdot r') = g \cdot (C \cdot r')) \Longrightarrow f \cdot \bot = g \cdot \bot \Longrightarrow f|_{C \cdot r} = g|_{C \cdot r}$
⟨*proof*⟩

lemma *C-restr-C-case*[*simp*]:
$(\mathit{C\text{-}case} \cdot f)|_{C \cdot r} = \mathit{C\text{-}case} \cdot (f|_r)$
⟨*proof*⟩

lemma *C-restr-bot-demand*:
assumes $C \cdot r \sqsubseteq \mathit{demand}\ f$
shows $f|_r = \bot$
⟨*proof*⟩

6.3.3 Restricting maps of C-ranged functions

definition *env-C-restr* $::$ $C \to ('\mathit{var}::\mathit{type} \Rightarrow (C \to 'a::\mathit{pcpo})) \to ('\mathit{var} \Rightarrow (C \to 'a))$ **where**
$\mathit{env\text{-}C\text{-}restr} = (\Lambda\ r\ f.\ \ \mathit{cfun\text{-}comp} \cdot (\mathit{C\text{-}restr} \cdot r) \cdot f)$

abbreviation *env-C-restr-syn* $::$ $('\mathit{var}::\mathit{type} \Rightarrow (C \to 'a::\mathit{pcpo})) \Rightarrow C \Rightarrow \ ('\mathit{var} \Rightarrow (C \to 'a))$ (
‹-|°-› [*111*,*110*] *110*)
where $f|^{\circ}{}_r \equiv \mathit{env\text{-}C\text{-}restr} \cdot r \cdot f$

lemma *env-C-restr-upd*[*simp*]: $(\varrho(x := v))|^{\circ}{}_r = (\varrho|^{\circ}{}_r)(x := v|_r)$
⟨*proof*⟩

lemma *env-C-restr-lookup*[*simp*]: $(\varrho|^{\circ}{}_r)\ v = \varrho\ v|_r$
⟨*proof*⟩

lemma *env-C-restr-bot*[*simp*]: $\bot|^{\circ}{}_r = \bot$
⟨*proof*⟩

lemma *env-C-restr-restr-below*[*intro*]: $\varrho|^{\circ}{}_r \sqsubseteq \varrho$
⟨*proof*⟩

lemma *env-C-restr-env-C-restr*[*simp*]: $(v|^{\circ}{}_{r'})|^{\circ}{}_r = v|^{\circ}{}_{(r' \sqcap r)}$
⟨*proof*⟩

lemma *env-C-restr-cong*:
$(\bigwedge x\ r'. \; r' \sqsubseteq r \Longrightarrow f\,x \cdot r' = g\ x \cdot r') \Longrightarrow f|^{\circ}{}_r = g|^{\circ}{}_r$
⟨*proof*⟩

end

6.4 CVale

theory *CValue*
imports *C*
begin

domain *CValue*
  = *CFn* (**lazy** $(C \to CValue) \to (C \to CValue)$)
  | *CB* (**lazy** *bool discr*)

fixrec *CFn-project* :: $CValue \to (C \to CValue) \to (C \to CValue)$
 where $CFn\text{-}project\cdot(CFn\cdot f)\cdot v = f \cdot v$

abbreviation *CFn-project-abbr* (**infix** ‹↓*CFn*› *55*)
  where $f \downarrow CFn\ v \equiv CFn\text{-}project\cdot f\cdot v$

lemma *CFn-project-strict*[*simp*]:
  $\bot \downarrow CFn\ v = \bot$
  $CB\cdot b \downarrow CFn\ v = \bot$
  ⟨*proof*⟩

lemma *CB-below*[*simp*]: $CB\cdot b \sqsubseteq v \longleftrightarrow v = CB\cdot b$
  ⟨*proof*⟩

fixrec *CB-project* :: $CValue \to CValue \to CValue \to CValue$ **where**
  $CB\text{-}project\cdot(CB\cdot db)\cdot v_1\cdot v_2 = (\mathit{if}\ undiscr\ db\ \mathit{then}\ v_1\ \mathit{else}\ v_2)$

lemma [*simp*]:
  $CB\text{-}project\cdot(CB\cdot(Discr\ b))\cdot v_1\cdot v_2 = (\mathit{if}\ b\ \mathit{then}\ v_1\ \mathit{else}\ v_2)$
  $CB\text{-}project\cdot\bot\cdot v_1\cdot v_2 = \bot$
  $CB\text{-}project\cdot(CFn\cdot f)\cdot v_1\cdot v_2 = \bot$
⟨*proof*⟩

lemma *CB-project-not-bot*:
  $CB\text{-}project\cdot scrut\cdot v_1\cdot v_2 \neq \bot \longleftrightarrow (\exists\ b.\ scrut = CB\cdot(Discr\ b) \land (\mathit{if}\ b\ \mathit{then}\ v_1\ \mathit{else}\ v_2) \neq \bot)$
  ⟨*proof*⟩

HOLCF provides us *CValue-take*::$nat \Rightarrow CValue \to CValue$; we want a similar function for $C \to CValue$.

abbreviation *C-to-CValue-take* :: $nat \Rightarrow (C \to CValue) \to (C \to CValue)$
   where $C\text{-}to\text{-}CValue\text{-}take\ n \equiv cfun\text{-}map\cdot ID\cdot(CValue\text{-}take\ n)$

lemma *C-to-CValue-chain-take*: *chain C-to-CValue-take*
  ⟨*proof*⟩

lemma *C-to-CValue-reach*: $(\bigsqcup\ n.\ C\text{-}to\text{-}CValue\text{-}take\ n\cdot x) = x$
  ⟨*proof*⟩

end

6.5 CValue-Nominal

theory *CValue−Nominal*
imports *CValue Nominal−Utils Nominal−HOLCF*
begin

instantiation *C* :: *pure*
begin
  definition $p \cdot (c{::}C) = c$
instance ⟨*proof*⟩
end
instance *C* :: *pcpo-pt*
  ⟨*proof*⟩

instantiation *CValue* :: *pure*
begin
  definition $p \cdot (v{::}CValue) = v$
instance
  ⟨*proof*⟩
end

instance *CValue* :: *pcpo-pt*
  ⟨*proof*⟩

end

6.6 ResourcedDenotational

theory *ResourcedDenotational*
imports *Abstract−Denotational−Props CValue−Nominal C−restr*
begin

type-synonym $CEnv = var \Rightarrow (C \to CValue)$

interpretation *semantic-domain*
  $\Lambda\ f\ .\ \Lambda\ r.\ CFn\cdot(\Lambda\ v.\ (f\cdot(v))|_r)$
  $\Lambda\ x\ y.\ (\Lambda\ r.\ (x\cdot r \downarrow CFn\ y|_r)\cdot r)$
  $\Lambda\ b\ r.\ CB\cdot b$
  $\Lambda\ scrut\ v1\ v2\ r.\ CB\text{-}project\cdot(scrut\cdot r)\cdot(v1\cdot r)\cdot(v2\cdot r)$
  C-case⟨*proof*⟩

notation *ESem-syn* (‹$\mathcal{N}[\![$ - $]\!]$-› [*60*,*60*] *60*)
notation *EvalHeapSem-syn* (‹$\boldsymbol{\mathcal{N}}[\![$ - $]\!]$-› [*0*,*0*] *110*)
notation *HSem-syn* (‹$\mathcal{N}\{\!|$-$|\!\}$-› [*60*,*60*] *60*)
notation *AHSem-bot* (‹$\mathcal{N}\{\!|$-$|\!\}$› [*60*] *60*)

Here we re-state the simplification rules, cleaned up by beta-reducing the locale parameters.

lemma *CESem-simps*:
$\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho}\ = (\Lambda\ (C \cdot r).\ CFn \cdot (\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)})|r))$
$\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho}\ = (\Lambda\ (C \cdot r).\ ((\mathcal{N}[\![\ e\ ]\!]_{\varrho}) \cdot r \downarrow CFn\ \varrho\ x|_{r}) \cdot r)$
$\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho}\ = (\Lambda\ (C \cdot r).\ (\varrho\ \ x) \cdot r)$
$\mathcal{N}[\![\ Bool\ b\ ]\!]_{\varrho}\ = (\Lambda\ (C \cdot r).\ CB \cdot (Discr\ b))$
$\mathcal{N}[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho}\ = (\Lambda\ (C \cdot r).\ CB\text{-}project \cdot ((\mathcal{N}[\![\ scrut\ ]\!]_{\varrho}) \cdot r) \cdot ((\mathcal{N}[\![\ e_1\ ]\!]_{\varrho}) \cdot r) \cdot ((\mathcal{N}[\![\ e_2\ ]\!]_{\varrho}) \cdot r))$
$\mathcal{N}[\![\ Let\ as\ body\ ]\!]_{\varrho} = (\Lambda\ (C \cdot r).\ (\mathcal{N}[\![body]\!]_{\mathcal{N}\{\!|as|\!\}\varrho}) \cdot r)$
⟨*proof*⟩

lemma *CESem-bot*[*simp*]:$(\mathcal{N}[\![\ e\ ]\!]_{\sigma}) \cdot \bot = \bot$
⟨*proof*⟩

lemma *CHSem-bot*[*simp*]:$((\mathcal{N}\{\!|\ \Gamma\ |\!\})\ x) \cdot \bot = \bot$
⟨*proof*⟩

Sometimes we do not care much about the resource usage and just want a simpler formula.

lemma *CESem-simps-no-tick*:
$(\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho}) \cdot r \sqsubseteq CFn \cdot (\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)})|r)$
$(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho}) \cdot r\ \sqsubseteq ((\mathcal{N}[\![\ e\ ]\!]_{\varrho}) \cdot r \downarrow CFn\ \varrho\ x|_{r}) \cdot r$
$\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho}\ \sqsubseteq \varrho\ x$
$(\mathcal{N}[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho}) \cdot r \sqsubseteq CB\text{-}project \cdot ((\mathcal{N}[\![\ scrut\ ]\!]_{\varrho}) \cdot r) \cdot ((\mathcal{N}[\![\ e_1\ ]\!]_{\varrho}) \cdot r) \cdot ((\mathcal{N}[\![\ e_2\ ]\!]_{\varrho}) \cdot r)$
$\mathcal{N}[\![\ Let\ as\ body\ ]\!]_{\varrho}\ \sqsubseteq\ \mathcal{N}[\![body]\!]_{\mathcal{N}\{\!|as|\!\}\varrho}$
⟨*proof*⟩

lemma *CELam-no-restr*: $(\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho}) \cdot r \sqsubseteq CFn \cdot (\Lambda\ v.\ (\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)}))$
⟨*proof*⟩

lemma *CEApp-no-restr*: $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho}) \cdot r\ \sqsubseteq ((\mathcal{N}[\![\ e\ ]\!]_{\varrho}) \cdot r \downarrow CFn\ \varrho\ x) \cdot r$
⟨*proof*⟩

end

7 Correctness of the natural semantics

7.1 CorrectnessOriginal

theory *CorrectnessOriginal*
imports *Denotational Launchbury*
begin

This is the main correctness theorem, Theorem 2 from [Lau93].

theorem *correctness*:
  assumes $\Gamma : e \Downarrow_L \Delta : v$
  and $fv\ (\Gamma,\ e) \subseteq set\ L \cup domA\ \Gamma$
  shows $[\![e]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![v]\!]_{\{\!|\Delta|\!\}\varrho}$
  and $(\{\!|\Gamma|\!\}\varrho)\ f|`\ domA\ \Gamma = (\{\!|\Delta|\!\}\varrho)\ f|`\ domA\ \Gamma$
  ⟨*proof*⟩

end

7.2 CorrectnessResourced

theory *CorrectnessResourced*
  imports *ResourcedDenotational Launchbury*
begin

theorem *correctness*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and $fv\ (\Gamma,\ e) \subseteq set\ L \cup domA\ \Gamma$
  shows $\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq \mathcal{N}[\![z]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho}$ **and** $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ f|`\ domA\ \Gamma \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\varrho)\ f|`\ domA\ \Gamma$
  ⟨*proof*⟩

corollary *correctness-empty-env*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and $fv\ (\Gamma,\ e) \subseteq set\ L$
  shows $\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![z]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **and** $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$
⟨*proof*⟩

end

8 Equivalence of the denotational semantics

8.1 ValueSimilarity

theory *ValueSimilarity*
imports *Value CValue Pointwise*
begin

This theory formalizes Section 3 of [SGHHOM11]. Their domain D is our type *Value*, their domain E is our type *CValue* and A corresponds to $C \to$ *CValue*.

In our case, the construction of the domains was taken care of by the HOLCF package ([Huf12]), so where [SGHHOM11] refers to elements of the domain approximations D_n resp. E_n, these are just elements of *Value* resp. *CValue* here. Therefore the *n-injection* $\phi^E_n : E_n \to E$ is the identity here.

The projections correspond to the take-functions generated by the HOLCF package:

$\psi^E_n : E \to E_n$	corresponds to	$CValue\text{-}take{::}nat \Rightarrow CValue \to CValue$
$\psi^A_n : A \to A_n$	corresponds to	$C\text{-}to\text{-}CValue\text{-}take{::}nat \Rightarrow (C \to CValue) \to C \to CValue$
$\psi^D_n : D \to D_n$	corresponds to	$Value\text{-}take{::}nat \Rightarrow Value \to Value.$

The syntactic overloading of $e(a)(c)$ to mean either $\mathsf{Ap}^{\perp}_{E_n}$ or $\mathsf{AP}^{\perp}_{E}$ turns into our non-overloaded - $\downarrow CFn$ -$::CValue \Rightarrow (C \to CValue) \Rightarrow C \to CValue$.

To have our presentation closer to [SGHHOM11], we introduce some notation:

notation *Value-take* ($\langle\psi^D{}_{\_}\rangle$)
notation *C-to-CValue-take* ($\langle\psi^A{}_{\_}\rangle$)
notation *CValue-take* ($\langle\psi^E{}_{\_}\rangle$)

8.1.1 A note about section 2.3

Section 2.3 of [SGHHOM11] contains equations (2) and (3) which do not hold in general. We demonstrate that fact here using our corresponding definition, but the counter-example carries over to the original formulation. Lemma (2) is a generalisation of (3) to the resourced semantics, so the counter-example for (3) is the simpler and more educating:

lemma *counter-example*:
  assumes *Equation (3)*: $\bigwedge n\ d\ d'.\ \psi^D{}_n \cdot (d \downarrow Fn\ d') = \psi^D{}_{Suc\ n} \cdot d \downarrow Fn\ \psi^D{}_n \cdot d'$
  shows *False*
$\langle proof \rangle$

For completeness, and to avoid making false assertions, the counter-example to equation (2):

lemma *counter-example2*:

assumes *Equation (2)*: $\bigwedge n\ e\ a\ c.\ \psi^E{}_n\cdot((e \downarrow CFn\ a)\cdot c) = (\psi^E{}_{Suc\ n}\cdot e \downarrow CFn\ \psi^A{}_n\cdot a)\cdot c$
shows *False*
$\langle proof \rangle$

A suitable substitute for the lemma can be found in 4.3.5 (1) in [AO93], which in our setting becomes the following (note the extra invocation of $\psi^D{}_n$ on the left hand side):

lemma *Abramsky 4,3,5 (1)*:
$\psi^D{}_n\cdot(d \downarrow Fn\ \psi^D{}_n\cdot d') = \psi^D{}_{Suc\ n}\cdot d \downarrow Fn\ \psi^D{}_n\cdot d'$
$\langle proof \rangle$

The problematic equations are used in the proof of the only-if direction of proposition 9 in [SGHHOM11]. It can be fixed by applying take-induction, which inserts the extra call to $\psi^D{}_n$ in the right spot.

8.1.2 Working with *Value* and *CValue*

Combined case distinguishing and induction rules.

lemma *value-CValue-cases*:
obtains
$x = \bot\ y = \bot$ |
f **where** $x = Fn\cdot f\ y = \bot$ |
g **where** $x = \bot\ y = CFn\cdot g$ |
$f\ g$ **where** $x = Fn\cdot f\ y = CFn \cdot g$ |
b_1 **where** $x = B\cdot(Discr\ b_1)\ y = \bot$ |
$b_1\ g$ **where** $x = B\cdot(Discr\ b_1)\ y = CFn\cdot g$ |
$b_1\ b_2$ **where** $x = B\cdot(Discr\ b_1)\ y = CB\cdot(Discr\ b_2)$ |
$f\ b_2$ **where** $x = Fn\cdot f\ y = CB\cdot(Discr\ b_2)$ |
b_2 **where** $x = \bot\ y = CB\cdot(Discr\ b_2)$
$\langle proof \rangle$

lemma *Value-CValue-take-induct*:
assumes *adm* (*case-prod P*)
assumes $\bigwedge n.\ P\ (\psi^D{}_n\cdot x)\ (\psi^A{}_n\cdot y)$
shows $P\ x\ y$
$\langle proof \rangle$

8.1.3 Restricted similarity is defined recursively

The base case

inductive *similar′-base* :: *Value* ⇒ *CValue* ⇒ *bool* **where**
bot-similar′-base[*simp,intro*]: *similar′-base* ⊥ ⊥

inductive-cases [*elim!*]:
similar′-base x y

The inductive case

inductive *similar′-step* :: $(Value \Rightarrow CValue \Rightarrow bool) \Rightarrow Value \Rightarrow CValue \Rightarrow bool$ **for** *s* **where**
bot-similar′-step[*intro*!]: *similar′-step* $s\ \bot\ \bot$ |
bool-similar′-step[*intro*]: *similar′-step* $s\ (B\cdot b)\ (CB\cdot b)$ |
Fun-similar′-step[*intro*]: $(\bigwedge\ x\ y\ .\ s\ x\ (y\cdot C^{\infty}) \Longrightarrow s\ (f\cdot x)\ (g\cdot y\cdot C^{\infty})) \Longrightarrow$ *similar′-step* $s\ (Fn\cdot f)\ (CFn\cdot g)$

inductive-cases [*elim*!]:
similar′-step $s\ x\ \bot$
similar′-step $s\ \bot\ y$
similar′-step $s\ (B\cdot f)\ (CB\cdot g)$
similar′-step $s\ (Fn\cdot f)\ (CFn\cdot g)$

We now create the restricted similarity relation, by primitive recursion over n.

This cannot be done using an inductive definition, as it would not be monotone.

fun *similar′* **where**
similar′ 0 = *similar′-base* |
similar′ $(Suc\ n)$ = *similar′-step* (*similar′* n)
declare *similar′.simps*[*simp del*]

abbreviation *similar′-syn* (‹- ◁▷- -› [*50*,*50*,*50*] *50*)
where *similar′-syn* $x\ n\ y \equiv$ *similar′* $n\ x\ y$

lemma *similar′-botI*[*intro*!,*simp*]: $\bot \lhd\!\rhd_n \bot$
⟨*proof*⟩

lemma *similar′-FnI*[*intro*]:
assumes $\bigwedge x\ y.\ \ x \lhd\!\rhd_n y\cdot C^{\infty} \Longrightarrow f\cdot x \lhd\!\rhd_n g\cdot y\cdot C^{\infty}$
shows $Fn\cdot f \lhd\!\rhd_{Suc\ n} CFn\cdot g$
⟨*proof*⟩

lemma *similar′-FnE*[*elim*!]:
assumes $Fn\cdot f \lhd\!\rhd_{Suc\ n} CFn\cdot g$
assumes $(\bigwedge x\ y.\ \ x \lhd\!\rhd_n y\cdot C^{\infty} \Longrightarrow f\cdot x \lhd\!\rhd_n g\cdot y\cdot C^{\infty}) \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *bot-or-not-bot′*:
$x \lhd\!\rhd_n y \Longrightarrow (x = \bot \longleftrightarrow y = \bot)$
⟨*proof*⟩

lemma *similar′-bot*[*elim-format*, *elim*!]:
$\bot \lhd\!\rhd_n x \Longrightarrow x = \bot$
$y \lhd\!\rhd_n \bot \Longrightarrow y = \bot$
⟨*proof*⟩

lemma *similar′-typed*[*simp*]:
$\neg\ B\cdot b \lhd\!\rhd_n CFn\cdot g$
$\neg\ Fn\cdot f \lhd\!\rhd_n CB\cdot b$

$\langle proof \rangle$

lemma *similar'-bool*[*simp*]:
$B \cdot b_1 ◁▷_{Suc\ n}\ CB \cdot b_2 \longleftrightarrow b_1 = b_2$
$\langle proof \rangle$

8.1.4 Moving up and down the similarity relations

These correspond to Lemma 7 in [SGHHOM11].

lemma *similar'-down*: $d ◁▷_{Suc\ n}\ e \Longrightarrow \psi^D{}_n \cdot d ◁▷_n \psi^E{}_n \cdot e$
and *similar'-up*: $d ◁▷_n e \Longrightarrow \psi^D{}_n \cdot d ◁▷_{Suc\ n}\ \psi^E{}_n \cdot e$
$\langle proof \rangle$

A generalisation of the above, doing multiple steps at once.

lemma *similar'-up-le*: $n \leq m \Longrightarrow \psi^D{}_n \cdot d ◁▷_n \psi^E{}_n \cdot e \Longrightarrow \psi^D{}_n \cdot d ◁▷_m \psi^E{}_n \cdot e$
$\langle proof \rangle$

lemma *similar'-down-le*: $n \leq m \Longrightarrow \psi^D{}_m \cdot d ◁▷_m \psi^E{}_m \cdot e \Longrightarrow \psi^D{}_n \cdot d ◁▷_n \psi^E{}_n \cdot e$
$\langle proof \rangle$

lemma *similar'-take*: $d ◁▷_n e \Longrightarrow \psi^D{}_n \cdot d ◁▷_n \psi^E{}_n \cdot e$
$\langle proof \rangle$

8.1.5 Admissibility

A technical prerequisite for induction is admissibility of the predicate, i.e. that the predicate holds for the limit of a chain, given that it holds for all elements.

lemma *similar'-base-adm*: *adm* (λ *x*. *similar'-base* (*fst x*) (*snd x*))
$\langle proof \rangle$

lemma *similar'-step-adm*:
assumes *adm* (λ *x*. *s* (*fst x*) (*snd x*))
shows *adm* (λ *x*. *similar'-step s* (*fst x*) (*snd x*))
$\langle proof \rangle$

lemma *similar'-adm*: *adm* (λx. *fst x* $◁▷_n$ *snd x*)
$\langle proof \rangle$

lemma *similar'-admI*: *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *adm* (λx. *f x* $◁▷_n$ *g x*)
$\langle proof \rangle$

8.1.6 The real similarity relation

This is the goal of the theory: A relation between *Value* and *CValue*.

definition *similar* :: *Value* $\Rightarrow$ *CValue* $\Rightarrow$ *bool* (**infix** ‹◁▷› *50*) **where**

$x \lhd\rhd y \longleftrightarrow (\forall\, n.\ \psi^D{}_n{\cdot}x \lhd\rhd_n \psi^E{}_n{\cdot}y)$

lemma *similarI*:
$(\bigwedge n.\ \psi^D{}_n{\cdot}x \lhd\rhd_n \psi^E{}_n{\cdot}y) \Longrightarrow x \lhd\rhd y$
⟨*proof*⟩

lemma *similarE*:
$x \lhd\rhd y \Longrightarrow \psi^D{}_n{\cdot}x \lhd\rhd_n \psi^E{}_n{\cdot}y$
⟨*proof*⟩

lemma *similar-bot*[*simp*]: $\bot \lhd\rhd \bot$ ⟨*proof*⟩

lemma *similar-bool*[*simp*]: $B{\cdot}b \lhd\rhd CB{\cdot}b$
⟨*proof*⟩

lemma [*elim-format*, *elim*!]: $x \lhd\rhd \bot \Longrightarrow x = \bot$
⟨*proof*⟩

lemma [*elim-format*, *elim*!]: $x \lhd\rhd CB{\cdot}b \Longrightarrow x = B{\cdot}b$
⟨*proof*⟩

lemma [*elim-format*, *elim*!]: $\bot \lhd\rhd y \Longrightarrow y = \bot$
⟨*proof*⟩

lemma [*elim-format*, *elim*!]: $B{\cdot}b \lhd\rhd y \Longrightarrow y = CB{\cdot}b$
⟨*proof*⟩

lemma *take-similar′-similar*:
assumes $x \lhd\rhd_n y$
shows $\psi^D{}_n{\cdot}x \lhd\rhd \psi^E{}_n{\cdot}y$
⟨*proof*⟩

lemma *bot-or-not-bot*:
$x \lhd\rhd y \Longrightarrow (x = \bot \longleftrightarrow y = \bot)$
⟨*proof*⟩

lemma *bool-or-not-bool*:
$x \lhd\rhd y \Longrightarrow (x = B{\cdot}b) \longleftrightarrow (y = CB{\cdot}b)$
⟨*proof*⟩

lemma *slimilar-bot-cases*[*consumes 1*, *case-names bot bool Fn*]:
assumes $x \lhd\rhd y$
obtains $x = \bot$ $y = \bot$ |
b **where** $x = B{\cdot}(Discr\ b)$ $y = CB{\cdot}(Discr\ b)$ |
$f\ g$ **where** $x = Fn{\cdot}f$ $y = CFn \cdot g$
⟨*proof*⟩

lemma *similar-adm*: $adm\ (\lambda x.\ fst\ x \lhd\rhd snd\ x)$
⟨*proof*⟩

lemma *similar-admI*: $cont\ f \Longrightarrow cont\ g \Longrightarrow adm\ (\lambda x.\ f\ x \triangleleft\triangleright g\ x)$
  ⟨*proof*⟩

Having constructed the relation we can how show that it indeed is the desired relation, relating $\bot$ with $\bot$ and functions with functions, if they take related arguments to related values. This corresponds to Proposition 9 in [SGHHOM11].

lemma *similar-nice-def*: $x \triangleleft\triangleright y \longleftrightarrow (x = \bot \wedge y = \bot \vee (\exists\ b.\ x = B\cdot(Discr\ b) \wedge y = CB\cdot(Discr\ b)) \vee (\exists\ f\ g.\ x = Fn\cdot f \wedge y = CFn\cdot g \wedge (\forall\ a\ b.\ a \triangleleft\triangleright b\cdot C^{\infty} \longrightarrow f\cdot a \triangleleft\triangleright g\cdot b\cdot C^{\infty})))$
  (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *similar-FnI*[*intro*]:
  assumes $\bigwedge x\ y.\ \ x \triangleleft\triangleright y\cdot C^{\infty} \Longrightarrow f\cdot x \triangleleft\triangleright g\cdot y\cdot C^{\infty}$
  shows $Fn\cdot f \triangleleft\triangleright CFn\cdot g$
⟨*proof*⟩

lemma *similar-FnD*[*elim!*]:
  assumes $Fn\cdot f \triangleleft\triangleright CFn\cdot g$
  assumes $x \triangleleft\triangleright y\cdot C^{\infty}$
  shows $f\cdot x \triangleleft\triangleright g\cdot y\cdot C^{\infty}$
⟨*proof*⟩

lemma *similar-FnE*[*elim!*]:
  assumes $Fn\cdot f \triangleleft\triangleright CFn\cdot g$
  assumes $(\bigwedge x\ y.\ \ x \triangleleft\triangleright y\cdot C^{\infty} \Longrightarrow f\cdot x \triangleleft\triangleright g\cdot y\cdot C^{\infty}) \Longrightarrow P$
  shows P
⟨*proof*⟩

8.1.7 The similarity relation lifted pointwise to functions.

abbreviation *fun-similar* :: $('a::type \Rightarrow Value) \Rightarrow ('a \Rightarrow (C \rightarrow CValue)) \Rightarrow bool$ (**infix** ‹◁▷*› *50*) **where**
  $fun\text{-}similar \equiv pointwise\ (\lambda x\ y.\ x \triangleleft\triangleright y\cdot C^{\infty})$

lemma *fun-similar-fmap-bottom*[*simp*]: $\bot \triangleleft\triangleright^* \bot$
  ⟨*proof*⟩

lemma *fun-similarE*[*elim*]:
  assumes $m \triangleleft\triangleright^* m'$
  assumes $(\bigwedge x.\ (m\ \ x) \triangleleft\triangleright (m'\ x)\cdot C^{\infty}) \Longrightarrow Q$
  shows Q
  ⟨*proof*⟩

end

8.2 Denotational-Related

theory *Denotational−Related*
imports *Denotational ResourcedDenotational ValueSimilarity*
begin

Given the similarity relation it is straight-forward to prove that the standard and the resourced denotational semantics produce similar results. (Theorem 10 in [SGHHOM11]).

theorem *denotational-semantics-similar*:
  assumes $\varrho \lhd\!\rhd^* \sigma$
  shows $[\![e]\!]_\varrho \lhd\!\rhd (\mathcal{N}[\![e]\!]_\sigma)\cdot C^\infty$
$\langle proof \rangle$

corollary *evalHeap-similar*:
  $\bigwedge y\ z.\ y \lhd\!\rhd^*\ z \Longrightarrow [\![\ \Gamma\ ]\!]_y \lhd\!\rhd^*\ \boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_z$
  $\langle proof \rangle$

theorem *heaps-similar*: $\{\!|\Gamma|\!\} \lhd\!\rhd^*\ \mathcal{N}\{\!|\Gamma|\!\}$
  $\langle proof \rangle$

end

9 Adequacy

9.1 ResourcedAdequacy

theory *ResourcedAdequacy*
imports *ResourcedDenotational Launchbury AList−Utils CorrectnessResourced*
begin

lemma *demand-not-0*: $demand\ (\mathcal{N}[\![e]\!]_{\varrho}) \neq \bot$
⟨*proof*⟩

The semantics of an expression, given only r resources, will only use values from the environment with less resources.

lemma *restr-can-restrict-env*: $(\mathcal{N}[\![\ e\ ]\!]_{\varrho})|_{C \cdot r} = (\mathcal{N}[\![\ e\ ]\!]_{\varrho|^{\circ}r})|_{C \cdot r}$
⟨*proof*⟩

lemma *can-restrict-env*:
$(\mathcal{N}[\![e]\!]_{\varrho})\cdot(C\cdot r) = (\mathcal{N}[\![\ e\ ]\!]_{\varrho|^{\circ}r})\cdot(C\cdot r)$
⟨*proof*⟩

When an expression e terminates, then we can remove such an expression from the heap and it still terminates. This is the crucial trick to handle black-holing in the resourced semantics.

lemma *add-BH*:
assumes $map\text{-}of\ \Gamma\ x = Some\ e$
assumes $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r' \neq \bot$
shows $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}})\cdot r' \neq \bot$
⟨*proof*⟩

The semantics is continuous, so we can apply induction here:

lemma *resourced-adequacy*:
assumes $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$
shows $\exists\ \Delta\ v.\ \Gamma : e \Downarrow_{S} \Delta : v$
⟨*proof*⟩

end

9.2 Adequacy

theory *Adequacy*
imports *ResourcedAdequacy Denotational−Related*
begin

theorem *adequacy*:
assumes $[\![e]\!]_{\{\!|\Gamma|\!\}} \neq \bot$

shows $\exists\ \Delta\ v.\ \Gamma : e \Downarrow_S \Delta : v$
$\langle proof \rangle$

end

References

[Abr90] Samson Abramsky, *The lazy lambda calculus*, Research topics in functional programming, 1990, pp. 65–116.

[AO93] Samson Abramsky and Chih-Hao Luke Ong, *Full abstraction in the lazy lambda calculus*, Information and Computation **105** (1993), no. 2, 159 – 267.

[Bre13] Joachim Breitner, *The correctness of launchbury's natural semantics for lazy evaluation*, Archive of Formal Proofs (2013), http://isa-afp.org/entries/Launchbury.shtml, Formal proof development.

[Huf12] Brian Huffman, *HOLCF '11: A definitional domain theory for verifying functional programs*, Ph.D. thesis, Portland State University, 2012.

[Lau93] John Launchbury, *A natural semantics for lazy evaluation*, POPL '93, 1993, pp. 144–154.

[Ses97] Peter Sestoft, *Deriving a lazy abstract machine*, Journal of Functional Programming **7** (1997), 231–264.

[SGHHOM11] Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero, and Yolanda Ortega-Mallén, *Relating function spaces to resourced function spaces*, SAC, 2011, pp. 1301–1308.

[SGHHOM14] ———, *The role of indirections in lazy natural semantics*, PSI, 2014.

[UK12] Christian Urban and Cezary Kaliszyk, *General bindings and alpha-equivalence in nominal isabelle*, Logical Methods in Computer Science **8** (2012), no. 2.