The Correctness of Launchbury's Natural Semantics for Lazy Evaluation

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

March 17, 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 **5**
- 1.1 Main definitions and theorems . . . 6
  - 1.1.1 The big picture . . . 6
  - 1.1.2 Expressions . . . 6
  - 1.1.3 The natural semantics . . . 7
  - 1.1.4 The denotational semantics . . . 7
  - 1.1.5 Correctness and Adequacy . . . 8
- 1.2 Differences to our previous work . . . 8
  - 1.2.1 The treatment of $\sqcup$ . . . 8
  - 1.2.2 The types of environments . . . 9
  - 1.2.3 No type *assn* . . . 10
- 1.3 Related work . . . 11
- 1.4 Theory overview . . . 11
- 1.5 Acknowledgements . . . 13

2 Auxiliary theories **14**
- 2.1 Pointwise . . . 14

2.2 AList-Utils . . . 14
2.2.1 The domain of an associative list . . . 14
2.2.2 Other lemmas about associative lists . . . 16
2.2.3 Syntax for map comprehensions . . . 17
2.3 Mono-Nat-Fun . . . 18
2.4 Nominal-Utils . . . 19
2.4.1 Lemmas helping with equivariance proofs . . . 19
2.4.2 Freshness via equivariance . . . 20
2.4.3 Additional simplification rules . . . 21
2.4.4 Additional equivariance lemmas . . . 21
2.4.5 Freshness lemmas . . . 24
2.4.6 Freshness and support for subsets of variables . . . 24
2.4.7 The set of free variables of an expression . . . 25
2.4.8 Other useful lemmas . . . 26
2.5 AList-Utils-Nominal . . . 28
2.5.1 Freshness lemmas related to associative lists . . . 28
2.5.2 Equivariance lemmas . . . 29
2.5.3 Freshness and distinctness . . . 29
2.5.4 Pure codomains . . . 30
2.6 HOLCF-Utils . . . 30
2.6.1 Composition of fun and cfun . . . 34
2.6.2 Additional transitivity rules . . . 34
2.7 HOLCF-Meet . . . 34
2.7.1 Towards meets: Lower bounds . . . 35
2.7.2 Greatest lower bounds . . . 35
2.8 Nominal-HOLCF . . . 39
2.8.1 Type class of continuous permutations and variations thereof . . . 39
2.8.2 Instance for *cfun* . . . 41
2.8.3 Instance for *fun* . . . 42
2.8.4 Instance for *u* . . . 42
2.8.5 Instance for *lift* . . . 43
2.8.6 Instance for *prod* . . . 44
2.9 Env . . . 44
2.9.1 The domain of a pcpo-valued function . . . 44
2.9.2 Updates . . . 45
2.9.3 Restriction . . . 45
2.9.4 Deleting . . . 48
2.9.5 Merging of two functions . . . 49
2.9.6 Environments with binary joins . . . 50
2.9.7 Singleton environments . . . 50
2.10 Env-Nominal . . . 51
2.10.1 Equivariance lemmas . . . 51
2.10.2 Permutation and restriction . . . 52
2.10.3 Pure codomains . . . 53

2.11 Env-HOLCF . . . 54
2.11.1 Continuity and pcpo-valued functions . . . 54
2.12 EvalHeap . . . 56
2.12.1 Conversion from heaps to environments . . . 56
2.12.2 Reordering lemmas . . . 58

3 Launchbury's natural semantics 59
3.1 Vars . . . 59
3.2 Terms . . . 59
3.2.1 Expressions . . . 59
3.2.2 Rewriting in terms of heaps . . . 60
3.2.3 Nice induction rules . . . 64
3.2.4 Testing alpha equivalence . . . 65
3.2.5 Free variables . . . 66
3.2.6 Lemmas helping with nominal definitions . . . 66
3.2.7 A smart constructor for lets . . . 68
3.2.8 A predicate for value expressions . . . 68
3.2.9 The notion of thunks . . . 69
3.2.10 Non-recursive Let bindings . . . 70
3.2.11 Renaming a lambda-bound variable . . . 72
3.3 Substitution . . . 72
3.4 Launchbury . . . 78
3.4.1 The natural semantics . . . 78
3.4.2 Example evaluations . . . 79
3.4.3 Better introduction rules . . . 80
3.4.4 Properties of the semantics . . . 81

4 Denotational domain 84
4.1 Value . . . 84
4.1.1 The semantic domain for values and environments . . . 84
4.2 Value-Nominal . . . 86

5 Denotational semantics 87
5.1 Iterative . . . 87
5.2 HasESem . . . 88
5.3 HeapSemantics . . . 89
5.3.1 A locale for heap semantics, abstract in the expression semantics . 89
5.3.2 Induction and other lemmas about *HSem* . . . 89
5.3.3 Substitution . . . 92
5.3.4 Re-calculating the semantics of the heap is idempotent . . . 93
5.3.5 Iterative definition of the heap semantics . . . 93
5.3.6 Fresh variables on the heap are irrelevant . . . 94
5.3.7 Freshness . . . 95
5.3.8 Adding a fresh variable to a heap does not affect its semantics . . 97

5.3.9 Mutual recursion with fresh variables 97
5.3.10 Parallel induction 99
5.3.11 Congruence rule 99
5.3.12 Equivariance of the heap semantics 100
5.4 AbstractDenotational 100
5.4.1 The denotational semantics for expressions 100
5.5 Abstract-Denotational-Props 102
5.5.1 The semantics ignores fresh variables 102
5.5.2 Nicer equations for ESem, without freshness requirements 103
5.5.3 Denotation of Substitution 104
5.6 Denotational 106

6 Resourced denotational domain 107
6.1 C 107
6.2 C-Meet 109
6.3 C-restr 110
6.3.1 The demand of a C-function 111
6.3.2 Restricting functions with domain C 113
6.3.3 Restricting maps of C-ranged functions 115
6.4 CValue 115
6.5 CValue-Nominal 116
6.6 ResourcedDenotational 117

7 Correctness of the natural semantics 119
7.1 CorrectnessOriginal 119
7.2 CorrectnessResourced 123

8 Equivalence of the denotational semantics 129
8.1 ValueSimilarity 129
8.1.1 A note about section 2.3 129
8.1.2 Working with *Value* and *CValue* 130
8.1.3 Restricted similarity is defined recursively 131
8.1.4 Moving up and down the similarity relations 132
8.1.5 Admissibility 133
8.1.6 The real similarity relation 135
8.1.7 The similarity relation lifted pointwise to functions. 139
8.2 Denotational-Related 139

9 Adequacy 142
9.1 ResourcedAdequacy 142
9.2 Adequacy 149

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.

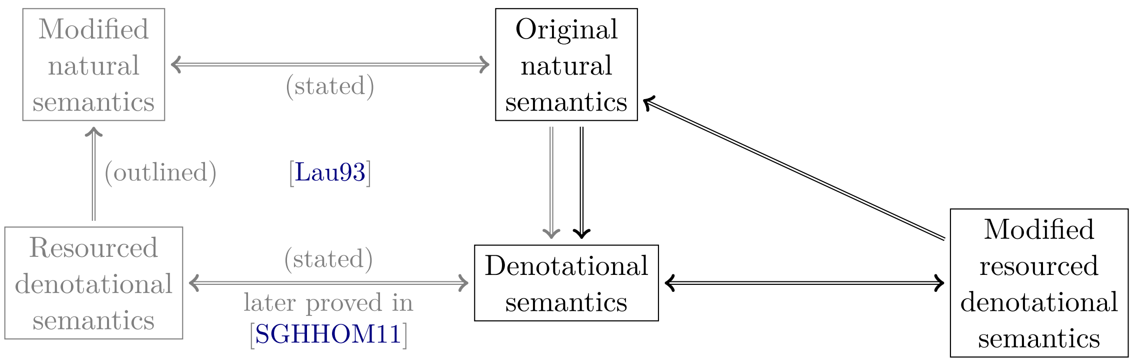
- We provide a new and more direct proof of adequacy.
- In order to do so, we formalize parts of [SGHHOM11], fixing a mistake in the proof.

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:

$$
\begin{array}{ll}
e ::= \lambda x.\ e & \text{lambda abstraction} \\
\quad\ \mid e\ x & \text{application} \\
\quad\ \mid x & \text{variable} \\
\quad\ \mid \textsf{let}\ as\ \textsf{in}\ e & \text{recursive let}
\end{array}
$$

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:

$$\frac{}{\Gamma : \lambda x.\ e \Downarrow_L \Gamma : \lambda x.\ e} \quad \textsc{Lambda}$$

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

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

$$\frac{\mathsf{dom}\ \Delta \sharp{*} (\Gamma,\ L) \qquad \Delta\ @\ \Gamma : body \Downarrow_L \Theta : z}{\Gamma : \mathsf{let}\ \Delta\ \mathsf{in}\ body \Downarrow_L \Theta : z} \quad \textsc{Let}$$

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:

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

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 $\_ \mathrel{++}_{\_} \_$ 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* ⇒ *Value*, but this causes problems when defining the value semantics, for which

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

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* ⇒ *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

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

holds unconditionally. The actual, technical definition is

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

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

This allows us to use the type $var \Rightarrow Value$ 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 $(var \times exp)$ *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 $(var \times exp)$ *list*. In [Bre13], this type and conversion functions from and to $(var \times exp)$ *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 $\_ \triangleleft\triangleright \_$ between the standard semantic domain (*Value*) and the resourced domain (*CValue*) and show that the denotation semantics yield similar results ($\varrho \triangleleft\triangleright^* \sigma \Longrightarrow [\![e]\!]_\varrho \triangleleft\triangleright (\mathcal{N}[\![e]\!]_\sigma)\cdot C^\infty$), 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.

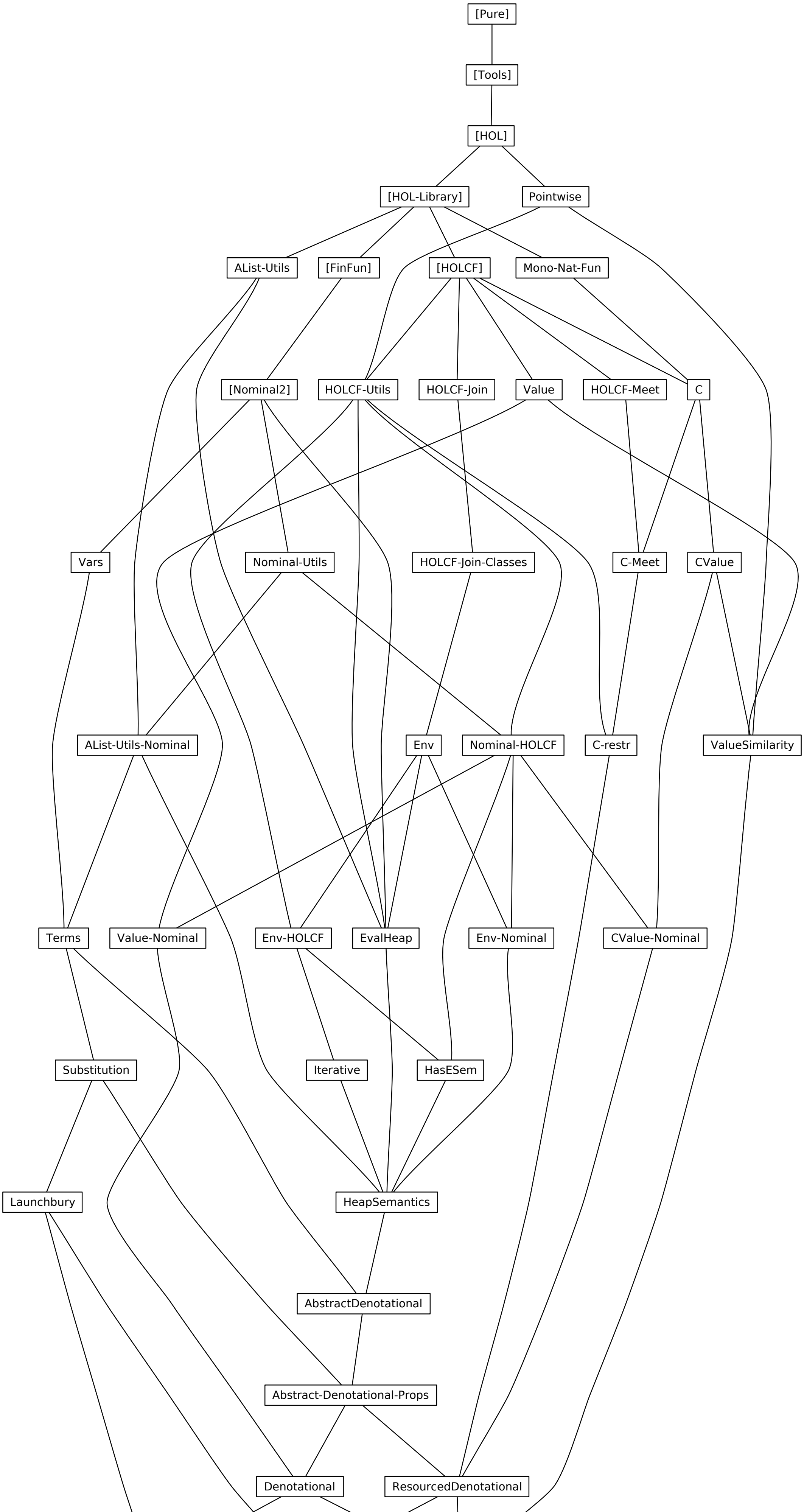
This work proves the adequacy as stated by Launchbury as well, but in contrast to his proof outline no alternative operational semantics is introduced. The problems of indirection vs. substitution and of blackholing is solved on the denotational side instead, which turned out to be much easier than proving 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* $P\ m\ m'$ **unfolding** *pointwise-def*
by *blast*

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*
  unfolding *restrict-eq* **by** (*rule filter-append*)

lemma *length-restrictA-le*: *length* (*restrictA S a*) ≤ *length a*
  by (*metis length-filter-le restrict-eq*)

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* [] = {}
  by (*auto simp add*: *domA-def*)

lemma *domA-from-set*:
$(x, e) \in set\ h \Longrightarrow x \in domA\ h$
by (*induct h*, *auto*)

lemma *finite-domA*[*simp*]:
finite (*domA* Γ)
by (*auto simp add*: *domA-def*)

lemma *domA-delete*[*simp*]:
$domA\ (delete\ x\ \Gamma) = domA\ \Gamma - \{x\}$
by (*induct* Γ) *auto*

lemma *domA-restrictA*[*simp*]:
$domA\ (restrictA\ S\ \Gamma) = domA\ \Gamma \cap S$
by (*induct* Γ) *auto*

lemma *delete-not-domA*[*simp*]:
$x \notin domA\ \Gamma \Longrightarrow\ delete\ x\ \Gamma = \Gamma$
by (*induct* Γ) *auto*

lemma *deleted-not-domA*: $x \notin domA\ (delete\ x\ \Gamma)$
by (*induct* Γ) *auto*

lemma *dom-map-of-conv-domA*:
$dom\ (map\text{-}of\ \Gamma) = domA\ \Gamma$
by (*induct* Γ) (*auto simp add*: *dom-if*)

lemma *domA-map-of-Some-the*:
$x \in domA\ \Gamma \Longrightarrow map\text{-}of\ \Gamma\ x = Some\ (the\ (map\text{-}of\ \Gamma\ x))$
by (*induct* Γ) (*auto simp add*: *dom-if*)

lemma *domA-clearjunk*[*simp*]: $domA\ (clearjunk\ \Gamma) = domA\ \Gamma$
unfolding *domA-def* **using** *dom-clearjunk*.

lemma *the-map-option-domA*[*simp*]: $x \in domA\ \Gamma \Longrightarrow the\ (map\text{-}option\ f\ (map\text{-}of\ \Gamma\ x)) = f\ (the\ (map\text{-}of\ \Gamma\ x))$
by (*induction* Γ) *auto*

lemma *map-of-domAD*: $map\text{-}of\ \Gamma\ x = Some\ e \Longrightarrow x \in domA\ \Gamma$
using *dom-map-of-conv-domA* **by** *fastforce*

lemma *restrictA-noop*: $domA\ \Gamma \subseteq S \Longrightarrow restrictA\ S\ \Gamma = \Gamma$
unfolding *restrict-eq* **by** (*induction* Γ) *auto*

lemma *restrictA-cong*:
$(\bigwedge x.\ x \in domA\ m1 \Longrightarrow x \in V \longleftrightarrow x \in V') \Longrightarrow m1 = m2 \Longrightarrow restrictA\ V\ m1 = restrictA\ V'\ m2$
unfolding *restrict-eq* **by** (*induction m1 arbitrary*: *m2*) *auto*

2.2.2 Other lemmas about associative lists

lemma *delete-set-none*: (*map-of l*)(*x* := *None*) = *map-of* (*delete x l*)
proof (*induct l*)
  case *Nil* **thus** *?case* **by** *simp*
  case (*Cons l ls*)
  from *this*[*symmetric*]
  show *?case*
  by (*cases fst l* = *x*) *auto*
qed

lemma *list-size-delete*[*simp*]: *size-list size* (*delete x l*) < *Suc* (*size-list size l*)
  by (*induct l*) *auto*

lemma *delete-append*[*simp*]: *delete x* (*l1* @ *l2*) = *delete x l1* @ *delete x l2*
  unfolding *AList.delete-eq* **by** *simp*

lemma *map-of-delete-insert*:
  assumes *map-of* Γ *x* = *Some e*
  shows *map-of* ((*x*,*e*) # *delete x* Γ) = *map-of* Γ
  using *assms* **by** (*induct* Γ) (*auto split*:*prod.split*)

lemma *map-of-delete-iff*[*simp*]: *map-of* (*delete x* Γ) *xa* = *Some e* ⟷ (*map-of* Γ *xa* = *Some e*) ∧ *xa* ≠ *x*
  by (*metis delete-conv fun-upd-same map-of-delete option.distinct*(*1*))

lemma *map-add-domA*[*simp*]:
  x ∈ *domA* Γ ⟹ (*map-of* Δ ++ *map-of* Γ) *x* = *map-of* Γ *x*
  x ∉ *domA* Γ ⟹ (*map-of* Δ ++ *map-of* Γ) *x* = *map-of* Δ *x*
    apply (*metis dom-map-of-conv-domA map-add-dom-app-simps*(*1*))
    apply (*metis dom-map-of-conv-domA map-add-dom-app-simps*(*3*))
    done

lemma *set-delete-subset*: *set* (*delete k al*) ⊆ *set al*
  by (*auto simp add*: *delete-eq*)

lemma *dom-delete-subset*: *snd* ‘ *set* (*delete k al*) ⊆ *snd* ‘ *set al*
  by (*auto simp add*: *delete-eq*)

lemma *map-ran-cong*[*fundef-cong*]:
  ⟦ ⋀ *x* . *x* ∈ *set m1* ⟹ *f1* (*fst x*) (*snd x*) = *f2* (*fst x*) (*snd x*) ; *m1* = *m2* ⟧
    ⟹ *map-ran f1 m1* = *map-ran f2 m2*
  by (*induction m1 arbitrary*: *m2*) *auto*

lemma *domA-map-ran*[*simp*]: *domA* (*map-ran f m*) = *domA m*
  unfolding *domA-def* **by** (*rule dom-map-ran*)

lemma *map-ran-delete*:
  map-ran f (*delete x* Γ) = *delete x* (*map-ran f* Γ)

by (*induction* Γ) *auto*

lemma *map-ran-restrictA*:
map-ran f (*restrictA V* Γ) = *restrictA V* (*map-ran f* Γ)
by (*induction* Γ) *auto*

lemma *map-ran-append*:
map-ran f (Γ@Δ) = *map-ran f* Γ @ *map-ran f* Δ
by (*induction* Γ) *auto*

2.2.3 Syntax for map comprehensions

definition *mapCollect* :: (′*a* ⇒ ′*b* ⇒ ′*c*) ⇒ (′*a* ⇀ ′*b*) ⇒ ′*c set*
where *mapCollect f m* = {*f k v* | *k v* . *m k* = *Some v*}

syntax
-MapCollect :: ′*c* ⇒ *pttrn* => *pttrn* ⇒ ′*a* ⇀ ′*b* => ′*c set* (‹(*1*{- |/-/↦/-/∈/-/})›)
syntax-consts
-MapCollect == *mapCollect*
translations
{*e* | *k*↦*v* ∈ *m*} == *CONST mapCollect* (λ*k v*. *e*) *m*

lemma *mapCollect-empty*[*simp*]: {*f k v* | *k* ↦ *v* ∈ *Map.empty*} = {}
unfolding *mapCollect-def* **by** *simp*

lemma *mapCollect-const*[*simp*]:
m ≠ *Map.empty* ⟹ {*e* | *k*↦*v*∈*m*} = {*e*}
unfolding *mapCollect-def* **by** *auto*

lemma *mapCollect-cong*[*fundef-cong*]:
(⋀ *k v*. *m1 k* = *Some v* ⟹ *f1 k v* = *f2 k v*) ⟹ *m1* = *m2* ⟹ *mapCollect f1 m1* = *mapCollect f2 m2*
unfolding *mapCollect-def* **by** *force*

lemma *mapCollectE*[*elim*!]:
assumes *x* ∈ {*f k v* | *k* ↦ *v* ∈ *m*}
obtains *k v* **where** *m k* = *Some v* **and** *x* = *f k v*
using *assms* **by** (*auto simp add*: *mapCollect-def*)

lemma *mapCollectI*[*intro*]:
assumes *m k* = *Some v*
shows *f k v* ∈ {*f k v* | *k* ↦ *v* ∈ *m*}
using *assms* **by** (*auto simp add*: *mapCollect-def*)

lemma *ball-mapCollect*[*simp*]:
(∀ *x* ∈ {*f k v* | *k* ↦ *v* ∈ *m*}. *P x*) ⟷ (∀ *k v*. *m k* = *Some v* ⟶ *P* (*f k v*))
by (*auto simp add*: *mapCollect-def*)

lemma *image-mapCollect*[*simp*]:
g ' $\{f\ k\ v \mid k \mapsto v \in m\} = \{\ g\ (f\ k\ v) \mid k \mapsto v \in m\}$
by (*auto simp add*: *mapCollect-def*)

lemma *mapCollect-map-upd*[*simp*]:
mapCollect f $(m(k \mapsto v))$ = *insert* $(f\ k\ v)$ (*mapCollect f* $(m(k := None)))$
unfolding *mapCollect-def* **by** *auto*

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*]:
$\{e \mid k \mapsto v \in m\ .\ False\ \} = \{\}$
unfolding *mapCollect-def mapCollectFilter-def* **by** *simp*

lemma *mapCollectFilter-const-True*[*simp*]:
$\{e \mid k \mapsto v \in m\ .\ True\ \} = \{e \mid k \mapsto v \in m\}$
unfolding *mapCollect-def mapCollectFilter-def* **by** *simp*

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 $f :: nat \Rightarrow nat$
assumes *mono f*
obtains n **where** $\bigwedge m\ .\ n \le m \Longrightarrow f\ n = f\ m \mid \bigwedge\ m\ .\ \exists\ \ n\ .\ m \le f\ n$
proof (*cases finite* (*range f*))
case *True*
from *Max-in*[*OF True*]
obtain n **where** *Max*: $f\ n = Max\ (range\ f)$ **by** *auto*

show *thesis*
proof(*rule that*(*1*))
fix *m*
assume $n \leq m$
hence $f\ n \leq f\ m$ **using** ‹*mono f*› **by** (*metis monoD*)
also
have $f\ m \leq f\ n$ **unfolding** *Max* **by** (*rule Max-ge*[*OF True rangeI*])
finally
show $f\ n = f\ m$**.**
qed
next
case *False*
thus *thesis* **by** (*fastforce intro*: *that*(*2*) *simp add*: *infinite-nat-iff-unbounded-le*)
qed

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** $?l \longleftrightarrow ?r$)
by (*metis* (*full-types*) *assms permute-minus-cancel*(*2*))

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** $?l \longleftrightarrow ?r$)
by (*metis* (*full-types*) *assms permute-minus-cancel*(*2*))

lemma *fun-eqvtI*:
assumes *f-eqvt*[*eqvt*]: $(\bigwedge p\ x.\ p \cdot (f\ x) = f\ (p \cdot x))$
shows $p \cdot f = f$ **by** *perm-simp rule*

lemma *eqvt-at-apply*:
assumes *eqvt-at f x*
shows $(p \cdot f)\ x = f\ x$
by (*metis* (*opaque-lifting*, *no-types*) *assms eqvt-at-def permute-fun-def permute-minus-cancel*(*1*))

lemma *eqvt-at-apply′*:
assumes *eqvt-at f x*
shows $p \cdot f\ x = f\ (p \cdot x)$
by (*metis* (*opaque-lifting*, *no-types*) *assms eqvt-at-def*)

lemma *eqvt-at-apply*$''$:
assumes *eqvt-at f x*
shows $(p \cdot f)\ (p \cdot x) = f\ (p \cdot x)$
by (*metis* (*opaque-lifting*, *no-types*) *assms eqvt-at-def permute-fun-def permute-minus-cancel*(*1*))

lemma *size-list-eqvt*[*eqvt*]: $p \cdot$ *size-list f x* = *size-list* $(p \cdot f)\ (p \cdot x)$
proof (*induction x*)
case (*Cons x xs*)
have $f\ x = p \cdot (f\ x)$ **by** (*simp add*: *permute-pure*)
also have ... $= (p \cdot f)\ (p \cdot x)$ **by** *simp*
with *Cons*
show *?case* **by** (*auto simp add*: *permute-pure*)
qed *simp*

2.4.2 Freshness via equivariance

lemma *eqvt-fresh-cong1*: $(\bigwedge p\ x.\ p \cdot (f\ x) = f\ (p \cdot x)) \Longrightarrow a \sharp x \Longrightarrow a \sharp f\ x$
apply (*rule fresh-fun-eqvt-app*[*of f*])
apply (*rule eqvtI*)
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*metis permute-fun-def permute-minus-cancel*(*1*))
apply *assumption*
done

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 $a \sharp f\ x\ y$
proof−
have *eqvt* (λ (x,y). $f\ x\ y$)
using *eqvt*
apply −
apply (*auto simp add*: *eqvt-def*)
apply (*rule ext*)
apply *auto*
by (*metis permute-minus-cancel*(*1*))
moreover
have $a \sharp (x,\ y)$ **using** *fresh1 fresh2* **by** *auto*
ultimately
have $a \sharp (\lambda\ (x,y).\ f\ x\ y)\ (x,\ y)$ **by** (*rule fresh-fun-eqvt-app*)
thus *?thesis* **by** *simp*
qed

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

by (*metis fresh-star-def eqvt-fresh-cong1 assms*)

lemma *eqvt-fresh-star-cong2*:
assumes *eqvt*: ($\bigwedge p\ x\ y.\ p \bullet (f\ x\ y) = f\ (p \bullet x)\ (p \bullet y)$)
and *fresh1*: $a \sharp* x$ **and** *fresh2*: $a \sharp* y$
shows $a \sharp* f\ x\ y$
by (*metis fresh-star-def eqvt-fresh-cong2 assms*)

lemma *eqvt-fresh-cong3*:
assumes *eqvt*: ($\bigwedge p\ x\ y\ z.\ p \bullet (f\ x\ y\ z) = f\ (p \bullet x)\ (p \bullet y)\ (p \bullet z)$)
and *fresh1*: $a \sharp x$ **and** *fresh2*: $a \sharp y$ **and** *fresh3*: $a \sharp z$
shows $a \sharp f\ x\ y\ z$
proof−
have *eqvt* (λ (*x*,*y*,*z*). *f x y z*)
using *eqvt*
apply −
apply (*auto simp add*: *eqvt-def*)
apply (*rule ext*)
apply *auto*
by (*metis permute-minus-cancel*(*1*))
moreover
have $a \sharp (x,\ y,\ z)$ **using** *fresh1 fresh2 fresh3* **by** *auto*
ultimately
have $a \sharp (\lambda\ (x,y,z).\ f\ x\ y\ z)\ (x,\ y,\ z)$ **by** (*rule fresh-fun-eqvt-app*)
thus *?thesis* **by** *simp*
qed

lemma *eqvt-fresh-star-cong3*:
assumes *eqvt*: ($\bigwedge p\ x\ y\ z.\ p \bullet (f\ x\ y\ z) = f\ (p \bullet x)\ (p \bullet y)\ (p \bullet z)$)
and *fresh1*: $a \sharp* x$ **and** *fresh2*: $a \sharp* y$ **and** *fresh3*: $a \sharp* z$
shows $a \sharp* f\ x\ y\ z$
by (*metis fresh-star-def eqvt-fresh-cong3 assms*)

2.4.3 Additional simplification rules

lemma *not-self-fresh*[*simp*]: *atom* $x \sharp x \longleftrightarrow$ *False*
by (*metis fresh-at-base*(*2*))

lemma *fresh-star-singleton*: $\{\ x\ \} \sharp* e \longleftrightarrow x \sharp e$
by (*simp add*: *fresh-star-def*)

2.4.4 Additional equivariance lemmas

lemma *eqvt-cases*:
fixes $f\ x\ \pi$
assumes *eqvt*: $\bigwedge x.\ \pi \bullet f\ x = f\ (\pi \bullet x)$
obtains $f\ x\ f\ (\pi \bullet x) \mid \neg\ f\ x \quad \neg\ f\ (\pi \bullet x)$
using *assms*[*symmetric*]
by (*cases f x*) *auto*

lemma *range-eqvt*: $\pi \cdot$ *range* Y = *range* $(\pi \cdot Y)$
unfolding *image-eqvt UNIV-eqvt* **..**

lemma *case-option-eqvt*[*eqvt*]:
$\pi \cdot$ *case-option d f x* = *case-option* $(\pi \cdot d)$ $(\pi \cdot f)$ $(\pi \cdot x)$
by(*cases x*)(*simp-all*)

lemma *supp-option-eqvt*:
supp (*case-option d f x*) $\subseteq$ *supp d* $\cup$ *supp f* $\cup$ *supp x*
apply (*cases x*)
apply (*auto simp add*: *supp-Some*)
apply (*metis* (*mono-tags*) *Un-iff subsetCE supp-fun-app*)
done

lemma *funpow-eqvt*[*simp*,*eqvt*]:
$\pi \cdot ((f :: 'a \Rightarrow 'a{::}pt)$ ^^ $n) = (\pi \cdot f)$ ^^ $(\pi \cdot n)$
apply (*induct n*)
apply *simp*
apply (*rule ext*)
apply *simp*
apply *perm-simp*
apply *simp*
done

lemma *delete-eqvt*[*eqvt*]:
$\pi \cdot$ *AList.delete* x Γ = *AList.delete* $(\pi \cdot x)$ $(\pi \cdot \Gamma)$
by (*induct* Γ, *auto*)

lemma *restrict-eqvt*[*eqvt*]:
$\pi \cdot$ *AList.restrict* S Γ = *AList.restrict* $(\pi \cdot S)$ $(\pi \cdot \Gamma)$
unfolding *AList.restrict-eq* **by** *perm-simp rule*

lemma *supp-restrict*:
supp (*AList.restrict* S Γ) $\subseteq$ *supp* Γ
by (*induction* Γ) (*auto simp add*: *supp-Pair supp-Cons*)

lemma *clearjunk-eqvt*[*eqvt*]:
$\pi \cdot$ *AList.clearjunk* Γ = *AList.clearjunk* $(\pi \cdot \Gamma)$
by (*induction* Γ *rule*: *clearjunk.induct*) *auto*

lemma *map-ran-eqvt*[*eqvt*]:
$\pi \cdot$ *map-ran f* Γ = *map-ran* $(\pi \cdot f)$ $(\pi \cdot \Gamma)$
by (*induct* Γ, *auto*)

lemma *dom-perm*:
dom $(\pi \cdot f) = \pi \cdot$ (*dom f*)
unfolding *dom-def* **by** (*perm-simp*) (*simp*)

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

lemma *ran-perm*[*simp*]:
$\pi \cdot (ran\ f) = ran\ (\pi \cdot f)$
unfolding *ran-def* **by** (*perm-simp*) (*simp*)

lemma *map-add-eqvt*[*eqvt*]:
$\pi \cdot (m1\ {+}{+}\ m2) = (\pi \cdot m1)\ {+}{+}\ (\pi \cdot m2)$
unfolding *map-add-def*
by (*perm-simp*, *rule*)

lemma *map-of-eqvt*[*eqvt*]:
$\pi \cdot map\text{-}of\ l = map\text{-}of\ (\pi \cdot l)$
apply (*induct l*)
apply (*simp add*: *permute-fun-def*)
apply *simp*
apply *perm-simp*
apply *auto*
done

lemma *concat-eqvt*[*eqvt*]: $\pi \cdot concat\ l = concat\ (\pi \cdot l)$
by (*induction l*)(*auto simp add*: *append-eqvt*)

lemma *tranclp-eqvt*[*eqvt*]: $\pi \cdot tranclp\ P\ v_1\ v_2 = tranclp\ (\pi \cdot P)\ (\pi \cdot v_1)\ (\pi \cdot v_2)$
unfolding *tranclp-def* **by** *perm-simp rule*

lemma *rtranclp-eqvt*[*eqvt*]: $\pi \cdot rtranclp\ P\ v_1\ v_2 = rtranclp\ (\pi \cdot P)\ (\pi \cdot v_1)\ (\pi \cdot v_2)$
unfolding *rtranclp-def* **by** *perm-simp rule*

lemma *Set-filter-eqvt*[*eqvt*]: $\pi \cdot Set.filter\ P\ S = Set.filter\ (\pi \cdot P)\ (\pi \cdot S)$
unfolding *Set.filter-def*
by *perm-simp rule*

lemma *Sigma-eqvt'*[*eqvt*]: $\pi \cdot Sigma = Sigma$
apply (*rule ext*)
apply (*rule ext*)
apply (*subst permute-fun-def*)
apply (*subst permute-fun-def*)
unfolding *Sigma-def*
apply *perm-simp*
apply (*simp add*: *permute-self*)
done

lemma *override-on-eqvt*[*eqvt*]:
$\pi \cdot (override\text{-}on\ m1\ m2\ S) = override\text{-}on\ (\pi \cdot m1)\ (\pi \cdot m2)\ (\pi \cdot S)$
by (*auto simp add*: *override-on-def*)

lemma *card-eqvt*[*eqvt*]:
$\pi \cdot (card\ S) = card\ (\pi \cdot S)$
by (*cases finite S*, *induct rule*: *finite-induct*) (*auto simp add*: *card-insert-if mem-permute-iff*

permute-pure)

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

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

2.4.5 Freshness lemmas

lemma *fresh-list-elem*:
  assumes $a \sharp \Gamma$
  and $e \in set\ \Gamma$
  shows $a \sharp e$
using *assms*
by(*induct* Γ)(*auto simp add*: *fresh-Cons*)

lemma *set-not-fresh*:
  $x \in set\ L \Longrightarrow \neg(atom\ x \sharp L)$
  by (*metis fresh-list-elem not-self-fresh*)

lemma *pure-fresh-star*[*simp*]: $a \sharp{*}\ (x :: 'a :: pure)$
  by (*simp add*: *fresh-star-def pure-fresh*)

lemma *supp-set-mem*: $x \in set\ L \Longrightarrow supp\ x \subseteq supp\ L$
  by (*induct L*) (*auto simp add*: *supp-Cons*)

lemma *set-supp-mono*: $set\ L \subseteq set\ L2 \Longrightarrow supp\ L \subseteq supp\ L2$
  by (*induct L*)(*auto simp add*: *supp-Cons supp-Nil dest*:*supp-set-mem*)

lemma *fresh-star-at-base*:
  fixes $x :: 'a :: at\text{-}base$
  shows $S \sharp{*}\ x \longleftrightarrow atom\ x \notin S$
  by (*metis fresh-at-base*(*2*) *fresh-star-def*)

2.4.6 Freshness and support for subsets of variables

lemma *supp-mono*: $finite\ (B::'a::fs\ set) \Longrightarrow A \subseteq B \Longrightarrow supp\ A \subseteq supp\ B$
  by (*metis infinite-super subset-Un-eq supp-of-finite-union*)

lemma *fresh-subset*:
  $finite\ B \Longrightarrow x \sharp (B :: 'a::at\text{-}base\ set) \Longrightarrow A \subseteq B \Longrightarrow x \sharp A$
  by (*auto dest*:*supp-mono simp add*: *fresh-def*)

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

lemma *fresh-star-set-subset*:
$x \sharp* (B :: 'a::at\text{-}base\ list) \Longrightarrow set\ A \subseteq set\ B \Longrightarrow x \sharp* A$
by (*metis fresh-star-set fresh-star-subset*[*OF finite-set*])

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*]: $\pi \cdot (fv\ e) = fv\ (\pi \cdot e)$
unfolding *fv-def* **by** *simp*

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

lemma *fv-set-at-base*[*simp*]: $fv\ (l :: ('a :: at\text{-}base)\ list) = set\ l$
by (*induction l*) *auto*

lemma *flip-not-fv*: $a \notin fv\ x \Longrightarrow b \notin fv\ x \Longrightarrow (a \leftrightarrow b) \cdot x = x$
by (*metis flip-def fresh-def fv-def mem-Collect-eq swap-fresh-fresh*)

lemma *fv-not-fresh*: $atom\ x \sharp e \longleftrightarrow x \notin fv\ e$
unfolding *fv-def fresh-def* **by** *blast*

lemma *fresh-fv*: *finite* $(fv\ e :: 'a\ set) \Longrightarrow atom\ (x :: ('a::at\text{-}base)) \sharp (fv\ e :: 'a\ set) \longleftrightarrow atom\ x$
$\sharp\ e$
unfolding *fv-def fresh-def*
by (*auto simp add*: *supp-finite-set-at-base*)

lemma *finite-fv*[*simp*]: *finite* $(fv\ (e::'a::fs) :: ('b::at\text{-}base)\ set)$
proof−
have *finite* (*supp e*) **by** (*metis finite-supp*)
hence *finite* $(atom\ -`\ supp\ e :: 'b\ set)$
apply (*rule finite-vimageI*)
apply (*rule inj-onI*)

apply (*simp*)
done
moreover
have (*atom* −‘ *supp e* :: ′*b set*) = *fv e* **unfolding** *fv-def* **by** *auto*
ultimately
show *?thesis* **by** *simp*
qed

definition *fv-list* :: ′*a*::*fs* ⇒ ′*b*::*at-base list*
where *fv-list e* = (*SOME l. set l* = *fv e*)

lemma *set-fv-list*[*simp*]: *set* (*fv-list e*) = (*fv e* :: (′*b*::*at-base*) *set*)
proof−
have *finite* (*fv e* :: ′*b set*) **by** (*rule finite-fv*)
from *finite-list*[*OF finite-fv*]
obtain *l* **where** *set l* = (*fv e* :: ′*b set*)**..**
thus *?thesis*
unfolding *fv-list-def* **by** (*rule someI*)
qed

lemma *fresh-fv-list*[*simp*]:
a ♯ (*fv-list e* :: ′*b*::*at-base list*) ⟷ *a* ♯ (*fv e* :: ′*b*::*at-base set*)
proof−
have *a* ♯ (*fv-list e* :: ′*b*::*at-base list*) ⟷ *a* ♯ *set* (*fv-list e* :: ′*b*::*at-base list*)
by (*rule fresh-set*[*symmetric*])
also have ... ⟷ *a* ♯ (*fv e* :: ′*b*::*at-base set*) **by** *simp*
finally show *?thesis***.**
qed

2.4.8 Other useful lemmas

lemma *pure-permute-id*: *permute p* = (λ *x*. (*x*::′*a*::*pure*))
by *rule* (*simp add*: *permute-pure*)

lemma *supp-set-elem-finite*:
assumes *finite S*
and (*m*::′*a*::*fs*) ∈ *S*
and *y* ∈ *supp m*
shows *y* ∈ *supp S*
using *assms supp-of-finite-sets*
by *auto*

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

lemma *mem-permute-set*:
shows *x* ∈ *p* · *S* ⟷ (− *p* · *x*) ∈ *S*
by (*metis mem-permute-iff permute-minus-cancel*(*2*))

lemma *flip-set-both-not-in*:

assumes $x \notin S$ **and** $x' \notin S$
shows $((x' \leftrightarrow x) \cdot S) = S$
unfolding *permute-set-def*
by (*auto*) (*metis assms flip-at-base-simps*(*3*))+

lemma *inj-atom*: *inj atom* **by** (*metis atom-eq-iff injI*)

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

lemma *eqvt-uncurry*: *eqvt f* $\Longrightarrow$ *eqvt* (*case-prod f*)
unfolding *eqvt-def*
by *perm-simp simp*

lemma *supp-fun-app-eqvt2*:
assumes *a*: *eqvt f*
shows *supp* (*f x y*) $\subseteq$ *supp x* $\cup$ *supp y*
proof−
from *supp-fun-app-eqvt*[*OF eqvt-uncurry* [*OF a*]]
have *supp* (*case-prod f* (*x*,*y*)) $\subseteq$ *supp* (*x*,*y*).
thus *?thesis* **by** (*simp add*: *supp-Pair*)
qed

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−
from *supp-fun-app-eqvt2*[*OF eqvt-uncurry* [*OF a*]]
have *supp* (*case-prod f* (*x*,*y*) *z*) $\subseteq$ *supp* (*x*,*y*) $\cup$ *supp z*.
thus *?thesis* **by** (*simp add*: *supp-Pair*)
qed

lemma *permute-0*[*simp*]: *permute 0* = (λ *x*. *x*)
by *auto*
lemma *permute-comp*[*simp*]: *permute x* $\circ$ *permute y* = *permute* (*x* + *y*) **by** *auto*

lemma *map-permute*: *map* (*permute p*) = *permute p*
apply *rule*
apply (*induct-tac x*)
apply *auto*
done

lemma *fresh-star-restrictA*[*intro*]: *a* $\sharp*$ $\Gamma \Longrightarrow$ *a* $\sharp*$ *AList.restrict V* Γ
by (*induction* Γ) (*auto simp add*: *fresh-star-Cons*)

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

apply *rule*
apply (*frule Abs-lst-fcb2*[**where** $f = \lambda$ *x y* - . (*x*,*y*) **and** *as* = [] **and** *bs* = *xs* **and** *c* = ()])
apply (*auto simp add*: *fresh-star-def*)
done

lemma *Abs-lst-Nil-eq2*[*simp*]: [*xs*]*lst*. (*x*::'*a*::*fs*) = [[]]*lst*. *x'* $\longleftrightarrow$ ((*xs*,*x*) = ([], *x'*))
by (*subst eq-commute*) *auto*

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*:
$x \in domA\ \Gamma \Longrightarrow \neg(atom\ x\ \sharp\ \Gamma)$
by (*induct* Γ, *auto simp add*: *fresh-Cons fresh-Pair*)

lemma *fresh-delete*:
assumes $a\ \sharp\ \Gamma$
shows $a\ \sharp$ *delete v* Γ
using *assms*
by(*induct* Γ)(*auto simp add*: *fresh-Cons*)

lemma *fresh-star-delete*:
assumes $S\ \sharp*\ \Gamma$
shows $S\ \sharp*$ *delete v* Γ
using *assms fresh-delete* **unfolding** *fresh-star-def* **by** *fastforce*

lemma *fv-delete-subset*:
fv (*delete v* Γ) $\subseteq$ *fv* Γ
using *fresh-delete* **unfolding** *fresh-def fv-def* **by** *auto*

lemma *fresh-heap-expr*:
assumes $a\ \sharp\ \Gamma$
and $(x,e) \in set\ \Gamma$
shows $a\ \sharp\ e$
using *assms*
by (*metis fresh-list-elem fresh-Pair*)

lemma *fresh-heap-expr'*:
assumes $a\ \sharp\ \Gamma$

and *e* ∈ *snd* ' *set* Γ
shows *a* ♯ *e*
using *assms*
by (*induct* Γ, *auto simp add*: *fresh-Cons fresh-Pair*)

lemma *fresh-star-heap-expr'*:
assumes *S* ♯* Γ
and *e* ∈ *snd* ' *set* Γ
shows *S* ♯* *e*
using *assms*
by (*metis fresh-star-def fresh-heap-expr'*)

lemma *fresh-map-of*:
assumes *x* ∈ *domA* Γ
assumes *a* ♯ Γ
shows *a* ♯ *the* (*map-of* Γ *x*)
using *assms*
by (*induct* Γ)(*auto simp add*: *fresh-Cons fresh-Pair*)

lemma *fresh-star-map-of*:
assumes *x* ∈ *domA* Γ
assumes *a* ♯* Γ
shows *a* ♯* *the* (*map-of* Γ *x*)
using *assms* **by** (*simp add*: *fresh-star-def fresh-map-of*)

lemma *domA-fv-subset*: *domA* Γ ⊆ *fv* Γ
by (*induction* Γ) *auto*

lemma *map-of-fv-subset*: *x* ∈ *domA* Γ ⟹ *fv* (*the* (*map-of* Γ *x*)) ⊆ *fv* Γ
by (*induction* Γ) *auto*

lemma *map-of-Some-fv-subset*: *map-of* Γ *x* = *Some e* ⟹ *fv e* ⊆ *fv* Γ
by (*metis domA-from-set map-of-fv-subset map-of-SomeD option.sel*)

2.5.2 Equivariance lemmas

lemma *domA*[*eqvt*]:
π • *domA* Γ = *domA* (π • Γ)
by (*simp add*: *domA-def*)

lemma *mapCollect*[*eqvt*]:
π • *mapCollect f m* = *mapCollect* (π • *f*) (π • *m*)
unfolding *mapCollect-def*
by *perm-simp rule*

2.5.3 Freshness and distinctness

lemma *fresh-distinct*:
assumes *atom* ' *S* ♯* Γ
shows *S* ∩ *domA* Γ = {}

proof−
  { **fix** *x*
    assume $x \in S$
    moreover
    assume $x \in domA\ \Gamma$
    hence $atom\ x \in supp\ \Gamma$
      by (*induct* Γ)(*auto simp add*: *supp-Cons domA-def supp-Pair supp-at-base*)
    ultimately
    have *False*
      using *assms*
      by (*simp add*: *fresh-star-def fresh-def*)
  }
  thus $S \cap domA\ \Gamma = \{\}$ **by** *auto*
qed

lemma *fresh-distinct-list*:
 assumes $atom\ `\ S\ \sharp*\ l$
 shows $S \cap set\ l = \{\}$
 using *assms*
 by (*metis disjoint-iff-not-equal fresh-list-elem fresh-star-def image-eqI not-self-fresh*)

lemma *fresh-distinct-fv*:
 assumes $atom\ `\ S\ \sharp*\ l$
 shows $S \cap fv\ l = \{\}$
 using *assms*
 by (*metis disjoint-iff-not-equal fresh-star-def fv-not-fresh image-eqI*)

2.5.4 Pure codomains

lemma *domA-fv-pure*:
  fixes $\Gamma :: ('a::at\text{-}base \times 'b::pure)\ list$
  shows $fv\ \Gamma = domA\ \Gamma$
  apply (*induct* Γ)
  apply *simp*
  apply (*case-tac a*)
  apply (*simp*)
  done

lemma *domA-fresh-pure*:
  fixes $\Gamma :: ('a::at\text{-}base \times 'b::pure)\ list$
  shows $x \in domA\ \Gamma \longleftrightarrow \neg(atom\ x \sharp\ \Gamma)$
  unfolding *domA-fv-pure*[*symmetric*]
  by (*auto simp add*: *fv-def fresh-def*)

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*))
by (*intro cont-apply*[*OF assms*(*4*) *assms*(*2*)]
cont2cont-fun[*OF cont-compose*[*OF - assms*(*3*)]]
cont2cont-lambda[*OF assms*(*1*)]])

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 (*rule admI*, *goal-cases*)
case *prems*: (*1 Y*)
show *?case*
apply (*rule pointwiseI*)
apply (*rule admD*[*OF adm-subst*[**where** *t* = λ*p* . (*fst p x*, *snd p x*) **for** *x*, *OF - assms*, *simplified*] ‹*chain Y*›])
using *prems*(*2*) **unfolding** *pointwise-def* **apply** *auto*
done
qed

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*
apply *simp*
apply (*subst beta-cfun*)
apply (*rule cont2cont-LAM*′)
apply (*rule assms*)
apply (*rule beta-cfun*)
apply (*rule cont2cont-fun*)
using *assms*
unfolding *prod-cont-iff*
apply *auto*
done

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

```
  apply (rule fun-belowI)
  apply (case-tac xa = x)
  apply simp
  apply (auto elim:fun-belowD)
  done

lemma fun-upd-cont[simp,cont2cont]:
  assumes cont f and cont h
  shows cont (λ x. (f x)(v := h x) :: 'a ⇒ 'b::pcpo)
  by (rule cont2cont-lambda)(auto simp add: assms)

lemma fun-upd-belowI:
  assumes ⋀ z . z ≠ x ⟹ ϱ z ⊑ ϱ' z
  assumes y ⊑ ϱ' x
  shows  ϱ(x := y) ⊑ ϱ'
  apply (rule fun-belowI)
  using assms
  apply (case-tac xa = x)
  apply auto
  done


lemma cont-if-else-above:
  assumes cont f
  assumes cont g
  assumes ⋀ x. f x ⊑ g x
  assumes ⋀ x y. x ⊑ y ⟹ P y ⟹ P x
  assumes adm P
  shows cont (λx. if P x then f x else g x) (is cont ?I)
proof(intro contI2 monofunI)
  fix x y :: 'a
  assume x ⊑ y
  with assms(4)[OF this]
  show ?I x ⊑ ?I y
    apply (auto)
    apply (rule cont2monofunE[OF assms(1)], assumption)
    apply (rule below-trans[OF cont2monofunE[OF assms(1)] assms(3)], assumption)
    apply (rule cont2monofunE[OF assms(2)], assumption)
    done
next
  fix Y :: nat ⇒ 'a
  assume chain Y
  assume chain (λi . ?I (Y i))

  have ch-f: f (⨆ i. Y i) ⊑ (⨆ i. f (Y i)) by (metis ‹chain Y› assms(1) below-refl cont2con-
tlubE)

  show ?I (⨆ i. Y i) ⊑ (⨆ i. ?I (Y i))
  proof(cases ∀ i. P (Y i))
```

case *True* **hence** $P\ (\bigsqcup\ i.\ Y\ i)$ **by** (*metis* ‹*chain Y*› *adm-def assms*(*5*))
with *True ch-f* **show** *?thesis* **by** *auto*
next
case *False*
then obtain *j* **where** $\neg\ P\ (Y\ j)$ **by** *auto*
hence $*$: $\forall\ i \geq j.\ \neg\ P\ (Y\ i)\ \neg\ P\ (\bigsqcup\ i.\ Y\ i)$
apply (*auto*)
apply (*metis assms*(*4*) *chain-mono*[*OF* ‹*chain Y*›])
apply (*metis assms*(*4*) *is-ub-thelub*[*OF* ‹*chain Y*›])
done

have $?I\ (\bigsqcup\ i.\ Y\ i) = g\ (\bigsqcup\ i.\ Y\ i)$ **using** $*$ **by** *simp*
also have $\ldots = g\ (\bigsqcup\ i.\ Y\ (i + j))$ **by** (*metis lub-range-shift*[*OF* ‹*chain Y*›])
also have $\ldots = (\bigsqcup\ i.\ (g\ (Y\ (i + j))))$ **by** (*rule cont2contlubE*[*OF assms*(*2*) *chain-shift*[*OF* ‹*chain Y*›]])
also have $\ldots = (\bigsqcup\ i.\ (?I\ (Y\ (i + j))))$ **using** $*$ **by** *auto*
also have $\ldots = (\bigsqcup\ i.\ (?I\ (Y\ i)))$ **by** (*metis lub-range-shift*[*OF* ‹*chain* $(\lambda i\ .\ ?I\ (Y\ i))$›])
finally show *?thesis* **by** *simp*
qed
qed

fun *up2option* :: $'a::cpo_{\bot} \Rightarrow\ 'a\ option$
where *up2option Ibottom* = *None*
| *up2option* (*Iup a*) = *Some a*

lemma *up2option-simps*[*simp*]:
up2option $\bot$ = *None*
up2option (*up·x*) = *Some x*
unfolding *up-def* **by** (*simp-all add*: *cont-Iup inst-up-pcpo*)

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

lemma *option2up-up2option*[*simp*]:
option2up (*up2option x*) = *x*
by (*cases x*) *auto*
lemma *up2option-option2up*[*simp*]:
up2option (*option2up x*) = *x*
by (*cases x*) *auto*

lemma *adm-subst2*: *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *adm* $(\lambda x.\ f\ (fst\ x) = g\ (snd\ x))$
apply (*rule admI*)
apply (*simp add*:
cont2contlubE[**where** *f* = *f*] *cont2contlubE*[**where** *f* = *g*]
cont2contlubE[**where** *f* = *snd*] *cont2contlubE*[**where** *f* = *fst*]
)
done

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* (λ *x*. (*f x*) ∘ (*g x*))
  unfolding *comp-def*
  by (*rule cont2cont-lambda*)
    (*intro cont2cont* ‹*cont g*› ‹*cont f*› *cont-compose2*[*OF cont2cont-fun*[*OF assms*(*1*)] *assms*(*2*)] *cont2cont-fun*)

definition *cfun-comp* :: (*'a*::*pcpo* → *'b*::*pcpo*) → (*'c*::*type* ⇒ *'a*) → (*'c*::*type* ⇒ *'b*)
  where *cfun-comp* = (Λ *f* ϱ. (λ *x*. *f*·*x*) ∘ ϱ)

lemma [*simp*]: *cfun-comp*·*f*·(ϱ(*x* := *v*)) = (*cfun-comp*·*f*·ϱ)(*x* := *f*·*v*)
  unfolding *cfun-comp-def* **by** *auto*

lemma *cfun-comp-app*[*simp*]: (*cfun-comp*·*f*·ϱ) *x* = *f*·(ϱ *x*)
  unfolding *cfun-comp-def* **by** *auto*

lemma *fix-eq-fix*:
  f·(*fix*·*g*) ⊑ *fix*·*g* ⟹ *g*·(*fix*·*f*) ⊑ *fix*·*f* ⟹ *fix*·*f* = *fix*·*g*
  by (*metis fix-least-below below-antisym*)

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 ⊑ *f x* ⟹ *x* ⊑ *y* ⟹ *cont f* ⟹ *a* ⊑ *f y*
by (*metis below-trans cont2monofunE*)

lemma *not-bot-below-trans*[*trans*]:
  a ≠ ⊥ ⟹ *a* ⊑ *b* ⟹ *b* ≠ ⊥
  by (*metis below-bottom-iff*)

lemma *not-bot-below-trans-cong*[*trans*]:
  f a ≠ ⊥ ⟹ *a* ⊑ *b* ⟹ *cont f* ⟹ *f b* ≠ ⊥
  by (*metis below-bottom-iff cont2monofunE*)

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* ⟷ (∀ *y*∈*S*. *x* ⊑ *y*)

lemma *is-lbI*: (!!*x*. *x* ∈ *S* ==> *l* ⊑ *x*) ==> *S* >| *l*
  by (*simp add*: *is-lb-def*)

lemma *is-lbD*: [|*S* >| *l*; *x* ∈ *S*|] ==> *l* ⊑ *x*
  by (*simp add*: *is-lb-def*)

lemma *is-lb-empty* [*simp*]: {} >| *l*
  unfolding *is-lb-def* **by** *fast*

lemma *is-lb-insert* [*simp*]: (*insert x A*) >| *y* = (*y* ⊑ *x* ∧ *A* >| *y*)
  unfolding *is-lb-def* **by** *fast*

lemma *is-lb-downward*: [|*S* >| *l*; *y* ⊑ *l*|] ==> *S* >| *y*
  unfolding *is-lb-def* **by** (*fast intro*: *below-trans*)

2.7.2 Greatest lower bounds

definition *is-glb* :: *'a set* ⇒ *'a* ⇒ *bool* (**infix** ‹>>|› *55*) **where**
  S >>| *x* ⟷ *S* >| *x* ∧ (∀ *u*. *S* >| *u* −−> *u* ⊑ *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*
  unfolding *is-glb-def* **by** *fast*

lemma *is-glbD2*: [|*S* >>| *x*; *S* >| *u*|] ==> *u* ⊑ *x*
  unfolding *is-glb-def* **by** *fast*

lemma (**in** *po*) *is-glbI*: [|*S* >| *x*; !!*u*. *S* >| *u* ==> *u* ⊑ *x*|] ==> *S* >>| *x*
  unfolding *is-glb-def* **by** *fast*

lemma *is-glb-above-iff*: *S* >>| *x* ==> *u* ⊑ *x* ⟷ *S* >| *u*
  unfolding *is-glb-def is-lb-def* **by** (*metis below-trans*)

glbs are unique

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

unfolding *is-glb-def is-lb-def* **by** (*blast intro*: *below-antisym*)

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

lemma *is-glb-glb*: *M* >>| *x* ==> *M* >>| *glb M*
unfolding *glb-def* **by** (*rule theI* [*OF - is-glb-unique*])

lemma *glb-eqI*: *M* >>| *l* ==> *glb M* = *l*
by (*rule is-glb-unique* [*OF is-glb-glb*])

lemma *is-glb-singleton*: {*x*} >>| *x*
by (*simp add*: *is-glb-def*)

lemma *glb-singleton* [*simp*]: *glb* {*x*} = *x*
by (*rule is-glb-singleton* [*THEN glb-eqI*])

lemma *is-glb-bin*: *x* ⊑ *y* ==> {*x*, *y*} >>| *x*
by (*simp add*: *is-glb-def*)

lemma *glb-bin*: *x* ⊑ *y* ==> *glb* {*x*, *y*} = *x*
by (*rule is-glb-bin* [*THEN glb-eqI*])

lemma *is-glb-maximal*: [|*S* >| *x*; *x* ∈ *S*|] ==> *S* >>| *x*
by (*erule is-glbI*, *erule* (*1*) *is-lbD*)

lemma *glb-maximal*: [|*S* >| *x*; *x* ∈ *S*|] ==> *glb S* = *x*
by (*rule is-glb-maximal* [*THEN glb-eqI*])

lemma *glb-above*: *S* >>| *z* ⟹ *x* ⊑ *glb S* ⟷ *S* >| *x*
by (*metis glb-eqI is-glb-above-iff*)
end

lemma (**in** *cpo*) *Meet-insert*: *S* >>| *l* ⟹ {*x*, *l*} >>| *l2* ⟹ *insert x S* >>| *l2*
apply (*rule is-glbI*)
apply (*metis is-glb-above-iff is-glb-def is-lb-insert*)
by (*metis is-glb-above-iff is-glb-def is-glb-singleton is-lb-insert*)

Binary, hence finite meets.

class *Finite-Meet-cpo* = *cpo* +
assumes *binary-meet-exists*: ∃ *l*. *l* ⊑ *x* ∧ *l* ⊑ *y* ∧ (∀ *z*. *z* ⊑ *x* ⟶ *z* ⊑ *y* ⟶ *z* ⊑ *l*)
begin

lemma *binary-meet-exists'*: ∃ *l*. {*x*, *y*} >>| *l*
using *binary-meet-exists*[*of x y*]
unfolding *is-glb-def is-lb-def*
by *auto*

lemma *finite-meet-exists*:
assumes *S* ≠ {}

and *finite S*
shows $\exists x.\ S >>|\ x$
using ‹$S \neq \{\}$›
apply (*induct rule*: *finite-induct*[*OF* ‹*finite S*›])
apply (*erule notE*, *rule refl*)[*1*]
apply (*case-tac F* = {})
apply (*metis is-glb-singleton*)
apply (*metis Meet-insert binary-meet-exists′*)
done
end

definition *meet* :: ′*a*::*cpo* ⇒ ′*a* ⇒ ′*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-Meet-cpo*) ⊓ *y* = *glb* {*x*, *y*}
unfolding *meet-def* **by** (*metis binary-meet-exists′*)

lemma *meet-comm*: (*x*::′*a*::*Finite-Meet-cpo*) ⊓ *y* = *y* ⊓ *x* **unfolding** *meet-def′* **by** (*metis insert-commute*)

lemma *meet-bot1*[*simp*]:
fixes *y* :: ′*a* :: {*Finite-Meet-cpo*,*pcpo*}
shows (⊥ ⊓ *y*) = ⊥ **unfolding** *meet-def′* **by** (*metis minimal po-class.glb-bin*)
lemma *meet-bot2*[*simp*]:
fixes *x* :: ′*a* :: {*Finite-Meet-cpo*,*pcpo*}
shows (*x* ⊓ ⊥) = ⊥ **by** (*metis meet-bot1 meet-comm*)

lemma *meet-below1*[*intro*]:
fixes *x y* :: ′*a* :: *Finite-Meet-cpo*
assumes *x* ⊑ *z*
shows (*x* ⊓ *y*) ⊑ *z* **unfolding** *meet-def′* **by** (*metis assms binary-meet-exists′ below-trans glb-eqI is-glbD1 is-lb-insert*)
lemma *meet-below2*[*intro*]:
fixes *x y* :: ′*a* :: *Finite-Meet-cpo*
assumes *y* ⊑ *z*
shows (*x* ⊓ *y*) ⊑ *z* **unfolding** *meet-def′* **by** (*metis assms binary-meet-exists′ below-trans glb-eqI is-glbD1 is-lb-insert*)

lemma *meet-above-iff*:
fixes *x y z* :: ′*a* :: *Finite-Meet-cpo*
shows *z* ⊑ *x* ⊓ *y* ⟷ *z* ⊑ *x* ∧ *z* ⊑ *y*
proof−
obtain *g* **where** {*x*,*y*} >>| *g* **by** (*metis binary-meet-exists′*)
thus *?thesis*
unfolding *meet-def′* **by** (*simp add*: *glb-above*)
qed

lemma *below-meet*[*simp*]:
fixes *x y* :: ′*a* :: *Finite-Meet-cpo*

assumes $x \sqsubseteq z$
shows $(x \sqcap z) = x$ **by** (*metis assms glb-bin meet-def'*)

lemma *below-meet2*[*simp*]:
fixes $x\ y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
assumes $z \sqsubseteq x$
shows $(x \sqcap z) = z$ **by** (*metis assms below-meet meet-comm*)

lemma *meet-aboveI*:
fixes $x\ y\ z :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
shows $z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq x \sqcap y$ **by** (*simp add*: *meet-above-iff*)

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 $x \sqcap y = z$
by (*metis assms below-antisym meet-above-iff below-refl*)

lemma *meet-assoc*[*simp*]: $((x::'a::\mathit{Finite\text{-}Meet\text{-}cpo}) \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
apply (*rule is-meetI*)
apply (*metis below-refl meet-above-iff*)
apply (*metis below-refl meet-below2*)
apply (*metis meet-above-iff*)
done

lemma *meet-self*[*simp*]: $r \sqcap r = (r::'a::\mathit{Finite\text{-}Meet\text{-}cpo})$
by (*metis below-refl is-meetI*)

lemma [*simp*]: $(r::'a::\mathit{Finite\text{-}Meet\text{-}cpo}) \sqcap (r \sqcap x) = r \sqcap x$
by (*metis below-refl is-meetI meet-below1*)

lemma *meet-monofun1*:
fixes $y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
shows *monofun* $(\lambda x.\ (x \sqcap y))$
by (*rule monofunI*)(*auto simp add*: *meet-above-iff*)

lemma *chain-meet1*:
fixes $y :: 'a :: \mathit{Finite\text{-}Meet\text{-}cpo}$
assumes *chain* Y
shows *chain* $(\lambda\ i.\ Y\ i \sqcap y)$
by (*rule chainI*) (*auto simp add*: *meet-above-iff intro*: *chainI chainE*[*OF assms*])

class *cont-binary-meet* = *Finite-Meet-cpo* +
assumes *meet-cont'*: *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 *cont* ($\lambda x.$ ($x \sqcap y$))
by (*rule contI2*[*OF meet-monofun1*]) (*simp add*: *meet-cont′*)

lemma *meet-cont2*:
fixes x :: ′*a* :: *cont-binary-meet*
shows *cont* ($\lambda y.$ ($x \sqcap y$)) **by** (*subst meet-comm*, *rule meet-cont1*)

lemma *meet-cont*[*cont2cont*,*simp*]:*cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *cont* ($\lambda x.$ ($f\,x \sqcap$ ($g\,x$::′*a*::*cont-binary-meet*)))
apply (*rule cont2cont-case-prod*[**where** $g = \lambda\ x.\ (f\,x,\ g\,x)$ **and** $f = \lambda\ p\ x\ y\ .\ x \sqcap y$, *simplified*])
apply (*rule meet-cont1*)
apply (*rule meet-cont2*)
apply (*metis cont2cont-Pair*)
done

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*} ⇒ ′*a*)

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

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

instance *pcpo-pt* ⊆ *cont-pt*
by *standard* (*auto intro*: *perm-cont*)

instance *discr-pt* ⊆ *cont-pt*
by *standard auto*

lemma (**in** *cont-pt*) *perm-cont-simp*[*simp*]: $\pi \bullet x \sqsubseteq \pi \bullet y \longleftrightarrow x \sqsubseteq y$
apply *rule*
apply (*drule cont2monofunE*[*OF perm-cont*, *of* - - $-\pi$], *simp*)[*1*]
apply (*erule cont2monofunE*[*OF perm-cont*, *of* - - π])
done

lemma (**in** *cont-pt*) *perm-below-to-right*: $\pi \bullet x \sqsubseteq y \longleftrightarrow x \sqsubseteq - \pi \bullet\ y$
  by (*metis perm-cont-simp pt-class.permute-minus-cancel*(*2*))

lemma *perm-is-ub-simp*[*simp*]: $\pi \bullet S <| \pi \bullet (x{::}'a{::}cont\text{-}pt) \longleftrightarrow S <| x$
  by (*auto simp add*: *is-ub-def permute-set-def*)

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

lemma *perm-is-lub-simp*[*simp*]: $\pi \bullet S <<| \pi \bullet (x{::}'a{::}cont\text{-}pt) \longleftrightarrow S <<| x$
  apply (*rule perm-rel-lemma*)
  by (*metis is-lubI is-lubD1 is-lubD2 perm-cont-simp perm-is-ub-simp*)

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

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

lemma *perm-still-cont*: $cont\ (\pi \bullet f) = cont\ (f :: ('a :: cont\text{-}pt) \Rightarrow ('b :: cont\text{-}pt))$
proof
  have *imp*: $\bigwedge (f :: 'a \Rightarrow 'b)\ \pi.\ cont\ f \Longrightarrow cont\ (\pi \bullet f)$
    unfolding *permute-fun-def*
    by (*metis cont-compose perm-cont*)
  show $cont\ f \Longrightarrow cont\ (\pi \bullet f)$ **using** *imp*[*of f* π]**.**
  show $cont\ (\pi \bullet f) \Longrightarrow cont\ (f)$ **using** *imp*[*of* $\pi \bullet f\ -\pi$] **by** *simp*
qed

lemma *perm-bottom*[*simp*,*eqvt*]: $\pi \bullet \bot = (\bot{::}'a{::}\{cont\text{-}pt,pcpo\})$
  proof−
  have $\bot \sqsubseteq -\pi \bullet (\bot{::}'a{::}\{cont\text{-}pt,pcpo\})$ **by** *simp*
  hence $\pi \bullet \bot \sqsubseteq \pi \bullet (-\pi \bullet (\bot{::}'a{::}\{cont\text{-}pt,pcpo\}))$ **by**(*rule cont2monofunE*[*OF perm-cont*])
  hence $\pi \bullet \bot \sqsubseteq (\bot{::}'a{::}\{cont\text{-}pt,pcpo\})$ **by** *simp*
  thus $\pi \bullet \bot = (\bot{::}'a{::}\{cont\text{-}pt,pcpo\})$ **by** *simp*
qed

lemma *bot-supp*[*simp*]: $supp\ (\bot :: 'a :: pcpo\text{-}pt) = \{\}$
  by (*rule supp-fun-eqvt*) (*simp add*: *eqvt-def*)

lemma *bot-fresh*[*simp*]: $a \sharp (\bot :: 'a :: pcpo\text{-}pt)$
  by (*simp add*: *fresh-def*)

lemma *bot-fresh-star*[*simp*]: $a \sharp{*} (\bot :: 'a :: pcpo\text{-}pt)$
  by (*simp add*: *fresh-star-def*)

lemma *below-eqvt* [*eqvt*]:
    $\pi \bullet (x \sqsubseteq y) = (\pi \bullet x \sqsubseteq \pi \bullet (y{::}'a{::}cont\text{-}pt))$ **by** (*auto simp add*: *permute-pure*)

lemma *lub-eqvt*[*simp*]:

$(\exists\ z.\ S <<| (z{::}'a{::}\{cont\text{-}pt\})) \Longrightarrow \pi \bullet lub\ S = lub\ (\pi \bullet S)$
  by (*metis lub-eqI perm-is-lub-simp*)

lemma *chain-eqvt*[*eqvt*]:
  fixes $F :: nat \Rightarrow 'a{::}cont\text{-}pt$
  shows $chain\ F \Longrightarrow chain\ (\pi \bullet F)$
  apply (*rule chainI*)
  apply (*drule-tac* $i = i$ **in** *chainE*)
  apply (*subst* (*asm*) *perm-cont-simp*[*symmetric*, **where** $\pi = \pi$])
  by (*metis permute-fun-app-eq permute-pure*)

2.8.2 Instance for *cfun*

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

  instance
  apply *standard*
  apply (*simp add*: *permute-cfun-def eta-cfun*)
  apply (*simp add*: *permute-cfun-def cfun-eqI minus-add*)
  done
end

lemma *permute-cfun-eq*: $permute\ p = (\lambda\ f.\ (Abs\text{-}cfun\ (permute\ p))\ oo\ f\ oo\ (Abs\text{-}cfun\ (permute\ (-p))))$
  by (*rule*, *rule cfun-eqI*, *auto simp add*: *permute-cfun-def*)

lemma *Cfun-app-eqvt*[*eqvt*]:
  $\pi \bullet (f \cdot x) = (\pi \bullet f) \cdot (\pi \bullet x)$
  unfolding *permute-cfun-def*
  by *auto*

lemma *permute-Lam*: $cont\ f \Longrightarrow p \bullet (\Lambda\ x.\ f\ x) = (\Lambda\ x.\ (p \bullet f)\ x)$
  apply (*rule cfun-eqI*)
  unfolding *permute-cfun-def*
  by (*metis Abs-cfun-inverse2 eqvt-lambda unpermute-def*)

lemma *Abs-cfun-eqvt*: $cont\ f \Longrightarrow (p \bullet Abs\text{-}cfun)\ f = Abs\text{-}cfun\ f$
  apply (*subst permute-fun-def*)
  by (*metis permute-Lam perm-still-cont permute-minus-cancel*(*1*))

lemma *cfun-eqvtI*: $(\bigwedge x.\ p \bullet (f \cdot x) = f' \cdot (p \bullet x)) \Longrightarrow p \bullet f = f'$
  by (*metis Cfun-app-eqvt cfun-eqI permute-minus-cancel*(*1*))

lemma *ID-eqvt*[*eqvt*]: $\pi \bullet ID = ID$
  unfolding *ID-def*
  apply *perm-simp*
  apply (*simp add*: *Abs-cfun-eqvt*)

done

instance *cfun* :: (*cont-pt*, *cont-pt*) *cont-pt*
by *standard* (*subst permute-cfun-eq*, *auto*)

instance *cfun* :: ({*pure*,*cont-pt*}, {*pure*,*cont-pt*}) *pure*
by *standard* (*auto simp add*: *permute-cfun-def permute-pure Cfun.cfun.Rep-cfun-inverse*)

instance *cfun* :: (*cont-pt*, *pcpo-pt*) *pcpo-pt*
by *standard*

2.8.3 Instance for *fun*

lemma *permute-fun-eq*: *permute* $p = (\lambda\ f.\ (permute\ p) \circ f \circ (permute\ (-p)))$
by (*rule*, *rule*, *metis comp-apply eqvt-lambda unpermute-def*)

instance *fun* :: (*pt*, *cont-pt*) *cont-pt*
apply *standard*
apply (*rule cont2cont-lambda*)
apply (*subst permute-fun-def*)
apply (*rule perm-cont2cont*)
apply (*rule cont-fun*)
done

lemma *fix-eqvt*[*eqvt*]:
$\pi \bullet fix = (fix :: ('a \to 'a) \to 'a{::}\{cont\text{-}pt,pcpo\})$
apply (*rule cfun-eqI*)
apply (*subst permute-cfun-def*)
apply *simp*
apply (*rule parallel-fix-ind*[*OF adm-subst2*])
apply (*auto simp add*: *permute-self*)
done

2.8.4 Instance for *u*

instantiation *u* :: (*cont-pt*) *pt*
begin
definition $p \bullet (x :: 'a\ u) = fup{\cdot}(\Lambda\ x.\ up{\cdot}(p \bullet x)){\cdot}x$
instance
apply *standard*
apply (*case-tac x*) **apply** (*auto simp add*: *permute-u-def*)
apply (*case-tac x*) **apply** (*auto simp add*: *permute-u-def*)
done
end

instance *u* :: (*cont-pt*) *cont-pt*
proof
fix *p*

have *permute* $p = (\lambda\ x.\ fup{\cdot}(\Lambda\ x.\ up{\cdot}(p \bullet x)){\cdot}(x{::}\ 'a\ u))$

by (*rule ext*, *rule permute-u-def*)
moreover have *cont* (λ *x*. *fup*·(Λ *x*. *up*·(*p* • *x*))·(*x*:: ′*a u*)) **by** *simp*
ultimately show *cont* (*permute p* :: ′*a u* ⇒ ′*a u*) **by** *simp*
qed

instance *u* :: (*cont-pt*) *pcpo-pt* **..**

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

instance *u* :: (*pure-cont-pt*) *pure*
apply *standard*
apply (*case-tac x*)
apply (*auto simp add*: *permute-u-def permute-pure*)
done

lemma *up-eqvt*[*eqvt*]: π • *up* = *up*
apply (*rule cfun-eqI*)
apply (*subst permute-cfun-def*, *simp*)
apply (*simp add*: *permute-u-def*)
done

lemma *fup-eqvt*[*eqvt*]: π • *fup* = *fup*
apply (*rule cfun-eqI*)
apply (*rule cfun-eqI*)
apply (*subst permute-cfun-def*, *simp*)
apply (*subst permute-cfun-def*, *simp*)
apply (*case-tac xa*)
apply *simp*
apply (*simp add*: *permute-self*)
done

2.8.5 Instance for *lift*

instantiation *lift* :: (*pt*) *pt*
begin
definition *p* • (*x* :: ′*a lift*) = *case-lift* ⊥ (λ *x*. *Def* (*p* • *x*)) *x*
instance
apply *standard*
apply (*case-tac x*) **apply** (*auto simp add*: *permute-lift-def*)
apply (*case-tac x*) **apply** (*auto simp add*: *permute-lift-def*)
done
end

instance *lift* :: (*pt*) *cont-pt*
proof
fix *p*

have *permute p* = (λ *x*. *case-lift* ⊥ (λ *x*. *Def* (*p* • *x*)) (*x*::′*a lift*))

by (*rule ext*, *rule permute-lift-def*)
moreover have *cont* (λ *x*. *case-lift* $\bot$ (λ *x*. *Def* (*p* $\cdot$ *x*)) (*x*::*'a lift*)) **by** *simp*
ultimately show *cont* (*permute p* :: *'a lift* $\Rightarrow$ *'a lift*) **by** *simp*
qed

instance *lift* :: (*pt*) *pcpo-pt* **..**

instance *lift* :: (*pure*) *pure*
apply *standard*
apply (*case-tac x*)
apply (*auto simp add*: *permute-lift-def permute-pure*)
done

lemma *Def-eqvt*[*eqvt*]: $\pi \cdot$ (*Def x*) = *Def* ($\pi \cdot x$)
by (*simp add*: *permute-lift-def*)

lemma *case-lift-eqvt*[*eqvt*]: $\pi \cdot$ *case-lift d f x* = *case-lift* ($\pi \cdot d$) ($\pi \cdot f$) ($\pi \cdot x$)
by (*cases x*) (*auto simp add*: *permute-self*)

2.8.6 Instance for *prod*

instance *prod* :: (*cont-pt*, *cont-pt*) *cont-pt*
proof
fix *p*

have *permute p* = (λ (*x* :: (*'a*, *'b*) *prod*). (*p* $\cdot$ *fst x*, *p* $\cdot$ *snd x*)) **by** *auto*
moreover have *cont* ... **by** (*intro cont2cont*)
ultimately show *cont* (*permute p* :: (*'a*,*'b*) *prod* $\Rightarrow$ (*'a*,*'b*) *prod*) **by** *simp*
qed

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* $\bot$ = {} **by** (*simp add*: *edom-def*)

lemma *bot-edom2*[*simp*]: *edom* (λ- . $\bot$) = {} **by** (*simp add*: *edom-def*)

lemma *edomIff*: ($a \in$ *edom* m) = ($m\ a \neq \bot$) **by** (*simp add*: *edom-def*)
lemma *edom-iff2*: ($m\ a = \bot$) $\longleftrightarrow$ ($a \notin$ *edom* m) **by** (*simp add*: *edom-def*)

lemma *edom-empty-iff-bot*: *edom* m = {} $\longleftrightarrow$ $m = \bot$
  by (*metis below-bottom-iff bot-edom edomIff empty-iff fun-belowI*)

lemma *lookup-not-edom*: $x \notin$ *edom* $m \Longrightarrow m\ x = \bot$ **by** (*auto iff*:*edomIff*)

lemma *lookup-edom*[*simp*]: $m\ x \neq \bot \Longrightarrow x \in$ *edom* m **by** (*auto iff*:*edomIff*)

lemma *edom-mono*: $x \sqsubseteq y \Longrightarrow$ *edom* $x \subseteq$ *edom* y
  unfolding *edom-def*
  by *auto* (*metis below-bottom-iff fun-belowD*)

lemma *edom-subset-adm*[*simp*]:
  adm ($\lambda ae'$. *edom* $ae' \subseteq S$)
  apply (*rule admI*)
  apply *rule*
  apply (*subst* (*asm*) *edom-def*) **back**
  apply *simp*
  apply (*subst* (*asm*) *lub-fun*) **apply** *assumption*
  apply (*subst* (*asm*) *lub-eq-bottom-iff*)
  apply (*erule ch2ch-fun*)
  unfolding *not-all*
  apply (*erule exE*)
  apply (*rule subsetD*)
  apply (*rule allE*) **apply** *assumption* **apply** *assumption*
  unfolding *edom-def*
  apply *simp*
  done

2.9.2 Updates

lemma *edom-fun-upd-subset*: *edom* (h ($x := v$)) $\subseteq$ *insert* x (*edom* h)
  by (*auto simp add*: *edom-def*)

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

2.9.3 Restriction

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

abbreviation *env-restr-rev* (**infixl** ‹*f*|‘› *110*)

where *env-restr-rev m S* ≡ *env-restr S m*

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

lemma *env-restr-empty-iff*[*simp*]: *m f*|‘ *S* = ⊥ ⟷ *edom m* ∩ *S* = {}
  apply (*auto simp add*: *edom-def env-restr-def lambda-strict*[*symmetric*] *split*:*if-splits*)
  apply *metis*
  apply (*fastforce simp add*: *edom-def env-restr-def lambda-strict*[*symmetric*] *split*:*if-splits*)
  done
lemmas *env-restr-empty* = *iffD2*[*OF env-restr-empty-iff*, *simp*]

lemma *lookup-env-restr*[*simp*]: *x* ∈ *S* ⟹ (*m f*|‘ *S*) *x* = *m x*
  by (*fastforce simp add*: *env-restr-def*)

lemma *lookup-env-restr-not-there*[*simp*]: *x* ∉ *S* ⟹ (*env-restr S m*) *x* = ⊥
  by (*fastforce simp add*: *env-restr-def*)

lemma *lookup-env-restr-eq*: (*m f*|‘ *S*) *x* = (*if x* ∈ *S then m x else* ⊥)
  by *simp*

lemma *env-restr-eqI*: (⋀*x*. *x* ∈ *S* ⟹ m_1 *x* = m_2 *x*) ⟹ m_1 *f*|‘ *S* = m_2 *f*|‘ *S*
  by (*auto simp add*: *lookup-env-restr-eq*)

lemma *env-restr-eqD*: m_1 *f*|‘ *S* = m_2 *f*|‘ *S* ⟹ *x* ∈ *S* ⟹ m_1 *x* = m_2 *x*
  by (*auto dest*!: *fun-cong*[**where** *x* = *x*])

lemma *env-restr-belowI*: (⋀*x*. *x* ∈ *S* ⟹ m_1 *x* ⊑ m_2 *x*) ⟹ m_1 *f*|‘ *S* ⊑ m_2 *f*|‘ *S*
  by (*auto intro*: *fun-belowI simp add*: *lookup-env-restr-eq*)

lemma *env-restr-belowD*: m_1 *f*|‘ *S* ⊑ m_2 *f*|‘ *S* ⟹ *x* ∈ *S* ⟹ m_1 *x* ⊑ m_2 *x*
  by (*auto dest*!: *fun-belowD*[**where** *x* = *x*])

lemma *env-restr-env-restr*[*simp*]:
 x f|‘ *d2 f*|‘ *d1* = *x f*|‘ (*d1* ∩ *d2*)
  by (*fastforce simp add*: *env-restr-def*)

lemma *env-restr-env-restr-subset*:
 d1 ⊆ *d2* ⟹ *x f*|‘ *d2 f*|‘ *d1* = *x f*|‘ *d1*
 by (*metis Int-absorb2 env-restr-env-restr*)

lemma *env-restr-useless*: *edom m* ⊆ *S* ⟹ *m f*|‘ *S* = *m*
  by (*rule ext*) (*auto simp add*: *lookup-env-restr-eq dest*!: *subsetD*)

lemma *env-restr-UNIV*[*simp*]: *m f*|‘ *UNIV* = *m*
  by (*rule env-restr-useless*) *simp*

lemma *env-restr-fun-upd*[*simp*]: *x* ∈ *S* ⟹ *m1*(*x* := *v*) *f*|‘ *S* = (*m1 f*|‘ *S*)(*x* := *v*)
  apply (*rule ext*)
  apply (*case-tac xa* = *x*)

apply (*auto simp add*: *lookup-env-restr-eq*)
done

lemma *env-restr-fun-upd-other*[*simp*]: $x \notin S \Longrightarrow m1(x := v)\ f|`\ S = m1\ f|`\ S$
apply (*rule ext*)
apply (*case-tac xa* = *x*)
apply (*auto simp add*: *lookup-env-restr-eq*)
done

lemma *env-restr-eq-subset*:
assumes $S \subseteq S'$
and $m1\ f|`\ S' = m2\ f|`\ S'$
shows $m1\ f|`\ S = m2\ f|`\ S$
using *assms*
by (*metis env-restr-env-restr le-iff-inf*)

lemma *env-restr-below-subset*:
assumes $S \subseteq S'$
and $m1\ f|`\ S' \sqsubseteq m2\ f|`\ S'$
shows $m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
using *assms*
by (*auto intro*!: *env-restr-belowI dest*!: *env-restr-belowD*)

lemma *edom-env*[*simp*]:
$edom\ (m\ f|`\ S) = edom\ m \cap S$
unfolding *edom-def env-restr-def* **by** *auto*

lemma *env-restr-below-self*: $f\ f|`\ S \sqsubseteq f$
by (*rule fun-belowI*) (*auto simp add*: *env-restr-def*)

lemma *env-restr-below-trans*:
$m1\ f|`\ S1 \sqsubseteq m2\ f|`\ S1 \Longrightarrow m2\ f|`\ S2 \sqsubseteq m3\ f|`\ S2 \Longrightarrow m1\ f|`\ (S1 \cap S2) \sqsubseteq m3\ f|`\ (S1 \cap S2)$
by (*auto intro*!: *env-restr-belowI dest*!: *env-restr-belowD elim*: *below-trans*)

lemma *env-restr-cont*: *cont* (*env-restr S*)
apply (*rule cont2cont-lambda*)
unfolding *env-restr-def*
apply (*intro cont2cont cont-fun*)
done

lemma *env-restr-mono*: $m1 \sqsubseteq m2 \Longrightarrow m1\ f|`\ S \sqsubseteq m2\ f|`\ S$
by (*metis env-restr-belowI fun-belowD*)

lemma *env-restr-mono2*: $S2 \subseteq S1 \Longrightarrow m\ f|`\ S2 \sqsubseteq m\ f|`\ S1$
by (*metis env-restr-below-self env-restr-env-restr-subset*)

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

lemma *env-restr-cong*: $(\bigwedge x.\ edom\ m \subseteq S \cap S' \cup - S \cap - S') \Longrightarrow m\ f|`\ S = m\ f|`\ S'$
by (*rule ext*)(*auto simp add*: *lookup-env-restr-eq edom-def*)

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*]:
$x' \neq x \Longrightarrow env\text{-}delete\ x\ m\ x' = m\ x'$
by (*simp add*: *env-delete-def*)

lemma *lookup-env-delete-None*[*simp*]:
$env\text{-}delete\ x\ m\ x = \bot$
by (*simp add*: *env-delete-def*)

lemma *edom-env-delete*[*simp*]:
$edom\ (env\text{-}delete\ x\ m) = edom\ m - \{x\}$
by (*auto simp add*: *env-delete-def edom-def*)

lemma *edom-env-delete-subset*:
$edom\ (env\text{-}delete\ x\ m) \subseteq edom\ m$ **by** *auto*

lemma *env-delete-fun-upd*[*simp*]:
$env\text{-}delete\ x\ (m(x := v)) = env\text{-}delete\ x\ m$
by (*auto simp add*: *env-delete-def*)

lemma *env-delete-fun-upd2*[*simp*]:
$(env\text{-}delete\ x\ m)(x := v) = m(x := v)$
by (*auto simp add*: *env-delete-def*)

lemma *env-delete-fun-upd3*[*simp*]:
$x \neq y \Longrightarrow env\text{-}delete\ x\ (m(y := v)) = (env\text{-}delete\ x\ m)(y := v)$
by (*auto simp add*: *env-delete-def*)

lemma *env-delete-noop*[*simp*]:
$x \notin edom\ m \Longrightarrow env\text{-}delete\ x\ m = m$
by (*auto simp add*: *env-delete-def edom-def*)

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

lemma *env-restr-env-delete-other*[*simp*]: $x \notin S \Longrightarrow env\text{-}delete\ x\ m\ f|`\ S = m\ f|`\ S$
apply (*rule ext*)
apply (*auto simp add*: *lookup-env-restr-eq*)
by (*metis lookup-env-delete*)

lemma *env-delete-restr*: $env\text{-}delete\ x\ m = m\ f|`\ (-\{x\})$
by (*auto simp add*: *lookup-env-restr-eq*)

lemma *below-env-deleteI*: $f\ x = \bot \Longrightarrow f \sqsubseteq g \Longrightarrow f \sqsubseteq$ *env-delete x g*
by (*metis env-delete-def env-delete-restr env-restr-mono fun-upd-triv*)

lemma *env-delete-below-cong*[*intro*]:
assumes $x \neq v \Longrightarrow e1\ x \sqsubseteq e2\ x$
shows *env-delete v e1 x* $\sqsubseteq$ *env-delete v e2 x*
using *assms* **unfolding** *env-delete-def* **by** *auto*

lemma *env-delete-env-restr-swap*:
env-delete x (*env-restr S e*) = *env-restr S* (*env-delete x e*)
by (*metis* (*erased*, *opaque-lifting*) *env-delete-def env-restr-fun-upd env-restr-fun-upd-other fun-upd-triv lookup-env-restr-eq*)

lemma *env-delete-mono*:
$m \sqsubseteq m' \Longrightarrow$ *env-delete x m* $\sqsubseteq$ *env-delete x* m'
unfolding *env-delete-restr*
by (*rule env-restr-mono*)

lemma *env-delete-below-arg*:
env-delete x m $\sqsubseteq m$
unfolding *env-delete-restr*
by (*rule env-restr-below-self*)

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** $f1 \mathbin{++_S} f2 \equiv$ *override-on f1 f2 S*

lemma *override-on-bot*[*simp*]:
$\bot \mathbin{++_S} m = m\ f|`\ S$
$m \mathbin{++_S} \bot = m\ f|`\ (-S)$
by (*auto simp add*: *override-on-def env-restr-def*)

lemma *edom-override-on*[*simp*]: *edom* $(m1 \mathbin{++_S} m2) = (\mathit{edom}\ m1 - S) \cup (\mathit{edom}\ m2 \cap S)$
by (*auto simp add*: *override-on-def edom-def*)

lemma *lookup-override-on-eq*: $(m1 \mathbin{++_S} m2)\ x = (\mathit{if}\ x \in S\ \mathit{then}\ m2\ x\ \mathit{else}\ m1\ x)$
by (*cases* $x \notin S$) *simp-all*

lemma *override-on-upd-swap*:
$x \notin S \Longrightarrow \varrho(x := z) \mathbin{++_S} \varrho' = (\varrho \mathbin{++_S} \varrho')(x := z)$
by (*auto simp add*: *override-on-def edom-def*)

lemma *override-on-upd*:
$x \in S \Longrightarrow \varrho \mathbin{++_S} (\varrho'(x := z)) = (\varrho \mathbin{++_{S - \{x\}}} \varrho')(x := z)$
by (*auto simp add*: *override-on-def edom-def*)

lemma *env-restr-add*: $(m1\ {+}{+}_{S2}\ m2)\ f|{`}\ S = m1\ f|{`}\ S\ {+}{+}_{S2}\ m2\ f|{`}\ S$
  by (*auto simp add*: *override-on-def edom-def env-restr-def*)

lemma *env-delete-add*: *env-delete x* $(m1\ {+}{+}_S\ m2)$ = *env-delete x m1* ${+}{+}_{S - \{x\}}$ *env-delete x m2*
  by (*auto simp add*: *override-on-def edom-def env-restr-def env-delete-def*)

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*
  unfolding *edom-def* **by** *auto*

lemma *env-delete-join*[*simp*]: *env-delete x* $(f \sqcup (g{::}('a{::}type \Rightarrow 'b{::}\{Finite\text{-}Join\text{-}cpo,pcpo\})))$ = *env-delete x f* $\sqcup$ *env-delete x g*
  by (*metis env-delete-def fun-upd-meet-simp*)

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

lemma *env-restr-join2*:
  fixes *m* :: $'a{::}type \Rightarrow 'b{::}\{Finite\text{-}Join\text{-}cpo,pcpo\}$
  shows $m\ f|{`}\ S \sqcup m\ f|{`}\ S' = m\ f|{`}\ (S \cup S')$
  by (*auto simp add*: *env-restr-def*)

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

lemma *env-restr-split*:
  fixes *m* :: $'a{::}type \Rightarrow 'b{::}\{Finite\text{-}Join\text{-}cpo,pcpo\}$
  shows $m = m\ f|{`}\ S \sqcup m\ f|{`}\ (-\ S)$
by (*simp add*: *env-restr-join2 Compl-partition*)

lemma *env-restr-below-split*:
  $m\ f|{`}\ S \sqsubseteq m' \Longrightarrow m\ f|{`}\ (-\ S) \sqsubseteq m' \Longrightarrow m \sqsubseteq m'$
  by (*metis ComplI fun-below-iff lookup-env-restr*)

2.9.7 Singleton environments

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

lemma *esing-bot*[*simp*]: *esing* $x \cdot \bot = \bot$
  by (*rule ext*)(*simp add*: *esing-def*)

lemma *esing-simps*[*simp*]:
  $(esing\ x \cdot n)\ x = n$
  $x' \neq x \Longrightarrow (esing\ x \cdot n)\ x' = \bot$
  by (*simp-all add*: *esing-def*)

lemma *esing-eq-up-iff*[*simp*]: $(esing\ x{\cdot}(up{\cdot}a))\ y = up{\cdot}a' \longleftrightarrow (x = y \wedge a = a')$
  by (*auto simp add*: *fun-below-iff esing-def*)

lemma *esing-below-iff*[*simp*]: $esing\ x \cdot a \sqsubseteq ae \ \longleftrightarrow a \sqsubseteq ae\ x$
  by (*auto simp add*: *fun-below-iff esing-def*)

lemma *edom-esing-subset*: $edom\ (esing\ x{\cdot}n) \subseteq \{x\}$
  unfolding *edom-def esing-def* **by** *auto*

lemma *edom-esing-up*[*simp*]: $edom\ (esing\ x \cdot (up \cdot n)) = \{x\}$
  unfolding *edom-def esing-def* **by** *auto*

lemma *env-delete-esing*[*simp*]: $env\text{-}delete\ x\ (esing\ x \cdot n) = \bot$
  unfolding *env-delete-def esing-def*
  by *auto*

lemma *env-restr-esing*[*simp*]:
  $x{\in}\ S \Longrightarrow esing\ x{\cdot}v\ f|`\ S = esing\ x{\cdot}v$
  by (*auto intro*: *env-restr-useless dest*: *subsetD*[*OF edom-esing-subset*])

lemma *env-restr-esing2*[*simp*]:
  $x \notin S \Longrightarrow esing\ x{\cdot}v\ f|`\ S = \bot$
  by (*auto dest*: *subsetD*[*OF edom-esing-subset*])

lemma *esing-eq-iff*[*simp*]:
  $esing\ x{\cdot}v = esing\ x{\cdot}v' \longleftrightarrow v = v'$
  by (*metis esing-simps*(*1*))

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 $edom\ (\pi \bullet f) = \pi \bullet (edom\ f)$
  by (*simp add*: *edom-def*)

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

lemma *mem-edom-perm*[*simp*]:
  fixes ϱ :: *'a*::*at-base* ⇒ *'b*::{*pcpo-pt*}
  shows $xa \in edom\ (p \cdot \varrho) \longleftrightarrow - p \cdot xa \in edom\ \varrho$
  by (*metis* (*mono-tags*) *edom-perm-rev mem-Collect-eq permute-set-eq*)

lemma *env-restr-eqvt*[*eqvt*]:
  fixes *m* :: *'a*::*pt* ⇒ *'b*::{*cont-pt*,*pcpo*}
  shows $\pi \cdot m\ f|`\ d = (\pi \cdot m)\ f|`\ (\pi \cdot d)$
  by (*auto simp add*: *env-restr-def*)

lemma *env-delete-eqvt*[*eqvt*]:
  fixes *m* :: *'a*::*pt* ⇒ *'b*::{*cont-pt*,*pcpo*}
  shows $\pi \cdot$ *env-delete x m* = *env-delete* $(\pi \cdot x)\ (\pi \cdot m)$
  by (*auto simp add*: *env-delete-def*)

lemma *esing-eqvt*[*eqvt*]: $\pi \cdot (esing\ x) = esing\ (\pi \cdot x)$
  unfolding *esing-def*
  apply *perm-simp*
  apply (*simp add*: *Abs-cfun-eqvt*)
  done

2.10.2 Permutation and restriction

lemma *env-restr-perm*:
  fixes ϱ :: *'a*::*at-base* ⇒ *'b*::{*pcpo-pt*,*pure*}
  assumes *supp p* ♯* *S* **and** [*simp*]: *finite S*
  shows $(p \cdot \varrho)\ f|`\ S = \varrho\ f|`\ S$
using *assms*
apply −
apply (*rule ext*)
apply (*case-tac* $x \in S$)
apply (*simp*)
apply (*subst permute-fun-def*)
apply (*simp add*: *permute-pure*)
apply (*subst perm-supp-eq*)
apply (*auto simp add*:*perm-supp-eq supp-minus-perm fresh-star-def fresh-def supp-set-elem-finite*)
done

lemma *env-restr-perm'*:
  fixes ϱ :: *'a*::*at-base* ⇒ *'b*::{*pcpo-pt*,*pure*}
  assumes *supp p* ♯* *S* **and** [*simp*]: *finite S*
  shows $p \cdot (\varrho\ f|`\ S) = \varrho\ f|`\ S$
  by (*simp add*: *perm-supp-eq*[*OF assms*(*1*)] *env-restr-perm*[*OF assms*])

lemma *env-restr-flip*:
  fixes ϱ :: *'a*::*at-base* ⇒ *'b*::{*pcpo-pt*,*pure*}
  assumes $x \notin S$ **and** $x' \notin S$

shows $((x' \leftrightarrow x) \cdot \varrho)\ f|`\ S = \varrho\ f|`\ S$
using *assms*
apply −
apply *rule*
apply (*auto simp add*: *permute-flip-at env-restr-def split*:*if-splits*)
by (*metis eqvt-lambda flip-at-base-simps*(*3*) *minus-flip permute-pure unpermute-def*)

lemma *env-restr-flip'*:
fixes ϱ :: *'a*::*at-base* ⇒ *'b*::{*pcpo-pt*,*pure*}
assumes $x \notin S$ **and** $x' \notin S$
shows $(x' \leftrightarrow x) \cdot (\varrho\ f|`\ S) = \varrho\ f|`\ S$
by (*simp add*: *flip-set-both-not-in*[*OF assms*] *env-restr-flip*[*OF assms*])

2.10.3 Pure codomains

lemma *edom-fv-pure*:
fixes *f* :: (*'a*::*at-base* ⇒ *'b*::{*pcpo*,*pure*})
assumes *finite* (*edom f*)
shows *fv f* ⊆ *edom f*
using *assms*
proof (*induction edom f arbitrary*: *f*)
case *empty*
hence $f = \bot$ **unfolding** *edom-def* **by** *auto*
thus *?case* **by** (*auto simp add*: *fv-def fresh-def supp-def*)
next
case (*insert x S*)
have *f* = (*env-delete x f*)(*x* := *f x*) **by** *auto*
hence *fv f* ⊆ *fv* (*env-delete x f*) ∪ *fv x* ∪ *fv* (*f x*)
using *eqvt-fresh-cong3*[**where** *f* = *fun-upd* **and** *x* = *env-delete x f* **and** *y* = *x* **and** *z* = *f x*, *OF fun-upd-eqvt*]
apply (*auto simp add*: *fv-def fresh-def*)
by (*metis fresh-def pure-fresh*)
also

from ‹*insert x S* = *edom f*› **and** ‹$x \notin S$›
have *S* = *edom* (*env-delete x f*) **by** *auto*
hence *fv* (*env-delete x f*) ⊆ *edom* (*env-delete x f*) **by** (*rule insert*)
also
have *fv* (*f x*) = {} **by** (*rule fv-pure*)
also
from ‹*insert x S* = *edom f*› **have** *x* ∈ *edom f* **by** *auto*
hence *edom* (*env-delete x f*) ∪ *fv x* ∪ {} ⊆ *edom f* **by** *auto*
finally
show *?case* **by** *this* (*intro Un-mono subset-refl*)
qed

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 $x +\!+_S\ y \sqsubseteq z$
  using *assms*
  apply −
  apply (*rule fun-belowI*)
  apply (*case-tac xa* ∈ *S*)
  apply *auto*
  done

lemma *override-on-cont1*: *cont* (λ *x*. $x +\!+_S\ m$)
  by (*rule cont2cont-lambda*) (*auto simp add*: *override-on-def*)

lemma *override-on-cont2*: *cont* (λ *x*. $m +\!+_S\ x$)
  by (*rule cont2cont-lambda*) (*auto simp add*: *override-on-def*)

lemma *override-on-cont2cont*[*simp*, *cont2cont*]:
  assumes *cont f*
  assumes *cont g*
  shows *cont* (λ *x*. $f\ x +\!+_S\ g\ x$)
by (*rule cont-apply*[*OF assms*(*1*) *override-on-cont1 cont-compose*[*OF override-on-cont2 assms*(*2*)]])

lemma *override-on-mono*:
  assumes $x1 \sqsubseteq (x2 :: {}'a{::}type \Rightarrow {}'b{::}cpo)$
  assumes $y1 \sqsubseteq y2$
  shows $x1 +\!+_S\ y1 \sqsubseteq x2 +\!+_S\ y2$
by (*rule below-trans*[*OF cont2monofunE*[*OF override-on-cont1 assms*(*1*)] *cont2monofunE*[*OF override-on-cont2 assms*(*2*)]])

lemma *fun-upd-below-env-deleteI*:
  assumes *env-delete x* $\varrho \sqsubseteq$ *env-delete x* ϱ'
  assumes $y \sqsubseteq \varrho'\ x$
  shows $\varrho(x := y) \sqsubseteq \varrho'$
  using *assms*
  apply (*auto intro*!: *fun-upd-belowI* *simp add*: *env-delete-def*)
  by (*metis fun-belowD fun-upd-other*)

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

apply (*rule fun-belowI*)
using *assms* **by** *auto*

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*
apply (*rule fun-belowI*)
by (*metis assms below-bottom-iff lookup-env-restr-not-there*)

lemma *env-restr-belowI2*:
assumes $\bigwedge$ *x*. $x \in S \Longrightarrow$ *m1 x* $\sqsubseteq$ *m2 x*
shows *m1 f*|' *S* $\sqsubseteq$ *m2*
by (*rule fun-belowI*)
(*simp add*: *assms env-restr-def*)

lemma *env-restr-below-itself*:
shows *m f*|' *S* $\sqsubseteq$ *m*
apply (*rule fun-belowI*)
apply (*case-tac* $x \in S$)
apply *auto*
done

lemma *env-restr-cont*: *cont* (*env-restr S*)
apply (*rule cont2cont-lambda*)
apply (*case-tac* $y \in S$)
apply *auto*
done

lemma *env-restr-belowD*:
assumes *m1 f*|' *S* $\sqsubseteq$ *m2 f*|' *S*
assumes $x \in S$
shows *m1 x* $\sqsubseteq$ *m2 x*
using *fun-belowD*[*OF assms*(*1*), **where** $x = x$] *assms*(*2*) **by** *simp*

lemma *env-restr-eqD*:
assumes *m1 f*|' *S* = *m2 f*|' *S*
assumes $x \in S$
shows *m1 x* = *m2 x*
by (*metis assms*(*1*) *assms*(*2*) *lookup-env-restr*)

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*
using *assms*
by (*auto intro*!: *env-restr-belowI dest*: *env-restr-belowD*)

lemma *override-on-below-restrI*:
assumes $x\, f|`\, (-S) \sqsubseteq z\, f|`\, (-S)$
and $y\, f|`\, S \sqsubseteq z\, f|`\, S$
shows $x\ {+}{+}_S\ y \sqsubseteq z$
using *assms*
by (*auto intro*: *override-on-belowI dest*:*env-restr-belowD*)

lemma *fmap-below-add-restrI*:
assumes $x\, f|`\, (-S) \sqsubseteq y\, f|`\, (-S)$
and $x\, f|`\, S \sqsubseteq z\, f|`\, S$
shows $x \sqsubseteq y\ {+}{+}_S\ z$
using *assms*
by (*auto intro*!: *fun-belowI dest*:*env-restr-belowD simp add*: *lookup-override-on-eq*)

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

lemma *env-delete-cont*: *cont* (*env-delete x*)
apply (*rule cont2cont-lambda*)
apply (*case-tac y* = *x*)
apply *auto*
done
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 [] - = $\bot$
| *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* ϱ *e*)) $\Longrightarrow$ *cont* (λ ϱ. *evalHeap h* (*eval* ϱ))
by(*induct h*, *auto*)

lemma *evalHeap-eqvt*[*eqvt*]:
$\pi \cdot$ *evalHeap h eval* = *evalHeap* ($\pi \cdot h$) ($\pi \cdot$ *eval*)
by (*induct h*) (*auto simp add*:*fun-upd-eqvt simp del*: *fun-upd-apply*)

lemma *edom-evalHeap-subset*:*edom* (*evalHeap h eval*) ⊆ *domA h*
by (*induct h eval rule*:*evalHeap.induct*) (*auto dest*:*subsetD*[*OF edom-fun-upd-subset*] *simp del*: *fun-upd-apply*)

lemma *evalHeap-cong*[*fundef-cong*]:
⟦ *heap1* = *heap2* ; (⋀ *e*. *e* ∈ *snd* ‘ *set heap2* ⟹ *eval1 e* = *eval2 e*) ⟧
⟹ *evalHeap heap1 eval1* = *evalHeap heap2 eval2*
by (*induct heap2 eval2 arbitrary*:*heap1 rule*:*evalHeap.induct*, *auto*)

lemma *lookupEvalHeap*:
assumes *v* ∈ *domA h*
shows (*evalHeap h f*) *v* = *f* (*the* (*map-of h v*))
using *assms*
by (*induct h f rule*: *evalHeap.induct*) *auto*

lemma *lookupEvalHeap′*:
assumes *map-of* Γ *v* = *Some e*
shows (*evalHeap* Γ *f*) *v* = *f e*
using *assms*
by (*induct* Γ *f rule*: *evalHeap.induct*) *auto*

lemma *lookupEvalHeap-other*[*simp*]:
assumes *v* ∉ *domA* Γ
shows (*evalHeap* Γ *f*) *v* = ⊥
using *assms*
by (*induct* Γ *f rule*: *evalHeap.induct*) *auto*

lemma *env-restr-evalHeap-noop*:
domA h ⊆ *S* ⟹ *env-restr S* (*evalHeap h eval*) = *evalHeap h eval*
apply (*rule ext*)
apply (*case-tac x* ∈ *S*)
apply (*auto simp add*: *lookupEvalHeap intro*: *lookupEvalHeap-other*)
done

lemma *env-restr-evalHeap-same*[*simp*]:
env-restr (*domA h*) (*evalHeap h eval*) = *evalHeap h eval*
by (*simp add*: *env-restr-evalHeap-noop*)

lemma *evalHeap-cong′*:
⟦ (⋀ *x*. *x* ∈ *domA heap* ⟹ *eval1* (*the* (*map-of heap x*)) = *eval2* (*the* (*map-of heap x*))) ⟧
⟹ *evalHeap heap eval1* = *evalHeap heap eval2*
apply (*rule ext*)
apply (*case-tac x* ∈ *domA heap*)
apply (*auto simp add*: *lookupEvalHeap*)
done

lemma *lookupEvalHeapNotAppend*[*simp*]:
assumes *x* ∉ *domA* Γ
shows (*evalHeap* (Γ@*h*) *f*) *x* = *evalHeap h f x*

using *assms* **by** (*induct* Γ, *auto*)

lemma *evalHeap-delete*[*simp*]: *evalHeap* (*delete x* Γ) *eval* = *env-delete x* (*evalHeap* Γ *eval*)
by (*induct* Γ) *auto*

lemma *evalHeap-mono*:
$x \notin domA\ \Gamma \Longrightarrow$
evalHeap Γ *eval* ⊑ *evalHeap* ((*x*, *e*) # Γ) *eval*
apply *simp*
apply (*rule fun-belowI*)
apply (*case-tac xa* ∈ *domA* Γ)
apply (*case-tac xa* = *x*)
apply *auto*
done

2.12.2 Reordering lemmas

lemma *evalHeap-reorder*:
assumes *map-of* Γ = *map-of* Δ
shows *evalHeap* Γ *h* = *evalHeap* Δ *h*
proof (*rule ext*)
from *assms*
have ∗: *domA* Γ = *domA* Δ **by** (*metis dom-map-of-conv-domA*)

fix *x*
show *evalHeap* Γ *h x* = *evalHeap* Δ *h x*
using *assms*(*1*) ∗
apply (*cases x* ∈ *domA* Γ)
apply (*auto simp add*: *lookupEvalHeap*)
done
qed

lemma *evalHeap-reorder-head*:
assumes *x* ≠ *y*
shows *evalHeap* ((*x*,*e1*)#(*y*,*e2*)#Γ) *eval* = *evalHeap* ((*y*,*e2*)#(*x*,*e1*)#Γ) *eval*
by (*rule evalHeap-reorder*) (*simp add*: *fun-upd-twist*[*OF assms*])

lemma *evalHeap-reorder-head-append*:
assumes *x* ∉ *domA* Γ
shows *evalHeap* ((*x*,*e*)#Γ@Δ) *eval* = *evalHeap* (Γ @ ((*x*,*e*)#Δ)) *eval*
by (*rule evalHeap-reorder*) (*simp*, *metis assms dom-map-of-conv-domA map-add-upd-left*)

lemma *evalHeap-subst-exp*:
assumes *eval e* = *eval e'*
shows *evalHeap* ((*x*,*e*)#Γ) *eval* = *evalHeap* ((*x*,*e'*)#Γ) *eval*
by (*simp add*: *assms*)

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*
 by (*metis exp-assn.size-eqvt*(*1*) *fun-eqvtI permute-pure*)

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*
unfolding *eqvt-def asToHeap-graph-aux-def*
apply *rule*
apply *simp*
apply *rule*
apply(*case-tac x rule*: *exp-assn.exhaust*(*2*))
apply *auto*
done
nominal-termination(*eqvt*) **by** *lexicographic-order*

lemma *asToHeap-eqvt*: *eqvt asToHeap*
 unfolding *eqvt-def*
 by (*auto simp add*: *permute-fun-def asToHeap.eqvt*)

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)
 by (*induct* Γ *rule*: *heapToAssn.induct*)
 (*auto simp add*: *heapToAssn.simps*)

lemma *bn-heapToAssn*: *bn* (*heapToAssn* Γ) = *map* (λ*x*. *atom* (*fst x*)) Γ
 by (*induct rule*: *heapToAssn.induct*)
 (*auto simp add*: *heapToAssn.simps exp-assn.bn-defs*)

lemma *set-bn-to-atom-domA*:
 set (*bn as*) = *atom* ‘ *domA* (*asToHeap as*)
 by (*induct as rule*: *asToHeap.induct*)
 (*auto simp add*: *exp-assn.bn-defs*)

They are inverse to each other.

lemma *heapToAssn-asToHeap*[*simp*]:
heapToAssn (*asToHeap as*) = *as*
by (*induct rule*: *exp-assn.inducts*(*2*)[*of* λ - . *True*])
(*auto simp add*: *heapToAssn.simps*)

lemma *asToHeap-heapToAssn*[*simp*]:
asToHeap (*heapToAssn as*) = *as*
by (*induct rule*: *heapToAssn.induct*)
(*auto simp add*: *heapToAssn.simps*)

lemma *heapToAssn-inject*[*simp*]:
heapToAssn x = *heapToAssn y* ⟷ *x* = *y*
by (*metis asToHeap-heapToAssn*)

They are transparent to various notions from the Nominal package.

lemma *supp-heapToAssn*: *supp* (*heapToAssn* Γ) = *supp* Γ
by (*induct rule*: *heapToAssn.induct*)
(*simp-all add*: *heapToAssn.simps exp-assn.supp supp-Nil supp-Cons supp-Pair sup-assoc*)

lemma *supp-asToHeap*: *supp* (*asToHeap as*) = *supp as*
by (*induct as rule*: *asToHeap.induct*)
(*simp-all add*: *exp-assn.supp supp-Nil supp-Cons supp-Pair sup-assoc*)

lemma *fv-asToHeap*: *fv* (*asToHeap* Γ) = *fv* Γ
unfolding *fv-def* **by** (*auto simp add*: *supp-asToHeap*)

lemma *fv-heapToAssn*: *fv* (*heapToAssn* Γ) = *fv* Γ
unfolding *fv-def* **by** (*auto simp add*: *supp-heapToAssn*)

lemma [*simp*]: *size* (*heapToAssn* Γ) = *size-list* (λ (*v*,*e*) . *size e*) Γ
by (*induct rule*: *heapToAssn.induct*)
(*simp-all add*: *heapToAssn.simps*)

lemma *Lam-eq-same-var*[*simp*]: *Lam* [*y*]. *e* = *Lam* [*y*]. *e'* ⟷ *e* = *e'*
by *auto* (*metis fresh-PairD*(*2*) *obtain-fresh*)

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*
unfolding *Let-def* **by** (*auto simp add*: *split-beta′*)

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 Γ *e* ≠ *Bool b*
(*scrut ? e1* : *e2*) ≠ *Let* Γ *e*
Let Γ *e* ≠ (*scrut ? e1* : *e2*)
unfolding *Let-def* **by** *simp-all*

lemma *Let-perm-simps*[*simp*,*eqvt*]:
p • *Let* Γ *e* = *Let* (*p* • Γ) (*p* • *e*)
unfolding *Let-def* **by** *simp*

lemma *Let-supp*:
supp (*Let* Γ *e*) = (*supp e* ∪ *supp* Γ) − *atom* ‘ (*domA* Γ)
unfolding *Let-def* **by** (*simp add*: *exp-assn.supp supp-heapToAssn bn-heapToAssn domA-def image-image*)

lemma *Let-fresh*[*simp*]:
a ♯ *Let* Γ *e* = (*a* ♯ *e* ∧ *a* ♯ Γ ∨ *a* ∈ *atom* ‘ *domA* Γ)
unfolding *fresh-def* **by** (*auto simp add*: *Let-supp*)

lemma *Abs-eq-cong*:
assumes ⋀ *p*. (*p* • *x* = *x′*) ⟷ (*p* • *y* = *y′*)
assumes *supp y* = *supp x*
assumes *supp y′* = *supp x′*
shows ([*a*]*lst*. *x* = [*a′*]*lst*. *x′*) ⟷ ([*a*]*lst*. *y* = [*a′*]*lst*. *y′*)
by (*simp add*: *Abs-eq-iff alpha-lst assms*)

lemma *Let-eq-iff*[*simp*]:
(*Let* Γ *e* = *Let* Γ′ *e′*) = ([*map* (λ*x*. *atom* (*fst x*)) Γ]*lst*. (*e*, Γ) = [*map* (λ*x*. *atom* (*fst x*)) Γ′]*lst*. (*e′*,Γ′))
unfolding *Let-def*
apply (*simp add*: *bn-heapToAssn*)
apply (*rule Abs-eq-cong*)
apply (*simp-all add*: *supp-Pair supp-heapToAssn*)
done

lemma *exp-strong-exhaust*:
fixes *c* :: *′a* :: *fs*
assumes ⋀*var*. *y* = *Var var* ⟹ *P*

assumes $\bigwedge$*exp var*. y = *App exp var* $\Longrightarrow$ P
assumes $\bigwedge\Gamma$ *exp*. *atom* ‘ *domA* Γ $\sharp*$ c $\Longrightarrow$ y = *Let* Γ *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
apply (*rule exp-assn.strong-exhaust*(*1*)[**where** y = y **and** c = c])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*) *set-bn-to-atom-domA Let-def heapToAssn-asToHeap*)
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
done

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* Γ $\Longrightarrow$ *P1 exp* $\Longrightarrow$ *P1* (*Let* Γ *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−
have *P1 e* **and** *P2* (*asToHeap* (*heapToAssn* Γ))
apply (*induct rule*: *exp-assn.inducts*[*of P1* λ *assn*. *P2* (*asToHeap assn*)])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*) *Let-def heapToAssn-asToHeap*)
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply (*metis assms*(*7*) *asToHeap.simps*(*1*))
apply (*metis assms*(*8*) *asToHeap.simps*(*2*))
done
thus *P1 e* **and** *P2* Γ **unfolding** *asToHeap-heapToAssn*.
qed

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*) ⟹ ($\bigwedge$*c. P2 c* Γ) ⟹ *P2 c* ((*var*,*exp*)#Γ)
fixes *c* :: *'a* :: *fs*
shows *P1 c e* **and** *P2 c* Γ
proof−
have *P1 c e* **and** *P2 c* (*asToHeap* (*heapToAssn* Γ))
apply (*induct rule*: *exp-assn.strong-induct*[*of P1* λ *c assn. P2 c* (*asToHeap assn*)])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*) *set-bn-to-atom-domA Let-def heapToAssn-asToHeap*)
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply (*metis assms*(*7*) *asToHeap.simps*(*1*))
apply (*metis assms*(*8*) *asToHeap.simps*(*2*))
done
thus *P1 c e* **and** *P2 c* Γ **unfolding** *asToHeap-heapToAssn*.
qed

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* ⟹ *P* (*App exp var*)
assumes $\bigwedge$Γ *exp.* ($\bigwedge$ *x. x* ∈ *domA* Γ ⟹ *P* (*the* (*map-of* Γ *x*))) ⟹ *P exp* ⟹ *P* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp. P exp* ⟹ *P* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$ *b. P* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2. P scrut* ⟹ *P e1* ⟹ *P e2* ⟹ *P* (*scrut ? e1 : e2*)
shows *P exp*
apply (*rule exp-heap-induct*[*of P* λ Γ. (∀ *x* ∈ *domA* Γ. *P* (*the* (*map-of* Γ *x*)))])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*))
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply *auto*
done

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*) ⟹ *P c* (*App exp var*)
assumes $\bigwedge$Γ *exp c.*
atom ‘ domA Γ ♯* *c* ⟹ ($\bigwedge$*c x e.* (*x*,*e*) ∈ *set* Γ ⟹ *P c e*) ⟹ ($\bigwedge$*c. P c exp*) ⟹ *P c* (*Let*

Γ *exp*)
assumes $\bigwedge$*var exp c*. {*atom var*} ♯∗ *c* ⟹ ($\bigwedge$*c*. *P c exp*) ⟹ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c*. *P c* (*Bool b*)
assumes $\bigwedge$*scrut e1 e2 c*. ($\bigwedge$ *c*. *P c scrut*) ⟹ ($\bigwedge$ *c*. *P c e1*) ⟹ ($\bigwedge$ *c*. *P c e2*) ⟹ *P c* (*scrut ? e1* : *e2*)
shows *P* (*c*::′*a*::*fs*) *exp*
apply (*rule exp-heap-strong-induct*(*1*)[*of P* λ *c* Γ. (∀ (*x*,*e*) ∈ *set* Γ. *P c e*)])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*) *split-conv*)
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply *auto*
done

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*) ⟹ *P c* (*App exp var*)
assumes $\bigwedge$Γ *exp c*.
atom ‘ *domA* Γ ♯∗ *c* ⟹ ($\bigwedge$*c x*. *x* ∈ *domA* Γ ⟹ *P c* (*the* (*map-of* Γ *x*))) ⟹ ($\bigwedge$*c*. *P c exp*) ⟹ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*var exp c*. {*atom var*} ♯∗ *c* ⟹ ($\bigwedge$*c*. *P c exp*) ⟹ *P c* (*Lam* [*var*]. *exp*)
assumes $\bigwedge$*b c*. *P c* (*Bool b*)
assumes $\bigwedge$ *scrut e1 e2 c*. ($\bigwedge$ *c*. *P c scrut*) ⟹ ($\bigwedge$ *c*. *P c e1*) ⟹ ($\bigwedge$ *c*. *P c e2*) ⟹ *P c* (*scrut ? e1* : *e2*)
shows *P* (*c*::′*a*::*fs*) *exp*
apply (*rule exp-heap-strong-induct*(*1*)[*of P* λ *c* Γ. (∀ *x* ∈ *domA* Γ. *P c* (*the* (*map-of* Γ *x*)))])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*metis assms*(*3*))
apply (*metis assms*(*4*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply *auto*
done

3.2.4 Testing alpha equivalence

lemma *alpha-test*:
shows *Lam* [*x*]. (*Var x*) = *Lam* [*y*]. (*Var y*)
by (*simp add*: *Abs1-eq-iff fresh-at-base pure-fresh*)

lemma *alpha-test2*:
shows *let x be* (*Var x*) *in* (*Var x*) = *let y be* (*Var y*) *in* (*Var y*)
by (*simp add*: *fresh-Cons fresh-Nil Abs1-eq-iff fresh-Pair add*: *fresh-at-base pure-fresh*)

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* = -)
by (*simp add*: *bn-heapToAssn Abs1-eq-iff fresh-Pair fresh-at-base*
Abs-swap2[*of atom x* (*?lb*, [(*x*, *Var y*), (*y*, *Var x*)]) [*atom x*, *atom y*] *atom y*])

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*)
by (*induction e* **and** *as rule*:*exp-assn.inducts*)
(*auto simp add*: *fv-def exp-assn.supp supp-at-base pure-supp*)

lemma *fv-supp-heap*: *supp* (Γ::*heap*) = *atom* ‘ (*fv* Γ :: *var set*)
by (*metis fv-def fv-supp-as supp-heapToAssn*)

lemma *fv-Lam*[*simp*]: *fv* (*Lam* [*x*]. *e*) = *fv e* − {*x*}
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp*)
lemma *fv-Var*[*simp*]: *fv* (*Var x*) = {*x*}
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp supp-at-base pure-supp*)
lemma *fv-App*[*simp*]: *fv* (*App e x*) = *insert x* (*fv e*)
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp supp-at-base*)
lemma *fv-Let*[*simp*]: *fv* (*Let* Γ *e*) = (*fv* Γ ∪ *fv e*) − *domA* Γ
unfolding *fv-def* **by** (*auto simp add*: *Let-supp exp-assn.supp supp-at-base set-bn-to-atom-domA*)
lemma *fv-Bool*[*simp*]: *fv* (*Bool b*) = {}
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp pure-supp*)
lemma *fv-IfThenElse*[*simp*]: *fv* (*scrut ? e1 : e2*) = *fv scrut* ∪ *fv e1* ∪ *fv e2*
unfolding *fv-def* **by** (*auto simp add*: *exp-assn.supp*)

lemma *fv-delete-heap*:
assumes *map-of* Γ *x* = *Some e*
shows *fv* (*delete x* Γ, *e*) ∪ {*x*} ⊆ (*fv* (Γ, *Var x*) :: *var set*)
proof−
have *fv* (*delete x* Γ) ⊆ *fv* Γ **by** (*metis fv-delete-subset*)
moreover
have (*x*,*e*) ∈ *set* Γ **by** (*metis assms map-of-SomeD*)
hence *fv e* ⊆ *fv* Γ **by** (*metis assms domA-from-set map-of-fv-subset option.sel*)
moreover
have *x* ∈ *fv* (*Var x*) **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

3.2.6 Lemmas helping with nominal definitions

lemma *eqvt-lam-case*:
assumes *Lam* [*x*]. *e* = *Lam* [*x'*]. *e'*
assumes ⋀ π . *supp* (−π) ♯∗ (*fv* (*Lam* [*x*]. *e*) :: *var set*) ⟹
supp π ♯∗ (*Lam* [*x*]. *e*) ⟹

$F\ (\pi \cdot e)\ (\pi \cdot x)\ (Lam\ [x].\ e) = F\ e\ x\ (Lam\ [x].\ e)$
shows $F\ e\ x\ (Lam\ [x].\ e) = F\ e'\ x'\ (Lam\ [x'].\ e')$
proof –

from *assms(1)*
have $[[atom\ x]]lst.\ (e, x) = [[atom\ x']]lst.\ (e', x')$ **by** *auto*
then obtain p
where $(supp\ (e, x) - \{atom\ x\})\ \sharp*\ p$
and $[simp]$: $p \cdot x = x'$
and $[simp]$: $p \cdot e = e'$
unfolding *Abs-eq-iff(3) alpha-lst.simps* **by** *auto*

from ‹- ♯* p›
have $*$: $supp\ (-p)\ \sharp*\ (fv\ (Lam\ [x].\ e) :: var\ set)$
by (*auto simp add*: *fresh-star-def fresh-def supp-finite-set-at-base supp-Pair fv-supp-exp fv-supp-heap supp-minus-perm*)

from ‹- ♯* p›
have $**$: $supp\ p\ \sharp*\ Lam\ [x].\ e$
by (*auto simp add*: *fresh-star-def fresh-def supp-Pair fv-supp-exp*)

have $F\ e\ x\ (Lam\ [x].\ e) = F\ (p \cdot e)\ (p \cdot x)\ (Lam\ [x].\ e)$ **by** (*rule assms(2)*[*OF* $*$ $**$, *symmetric*])
also have $\ldots = F\ e'\ x'\ (Lam\ [x'].\ e')$ **by** (*simp add*: *assms(1)*)
finally show *?thesis*.
qed

lemma *eqvt-let-case*:
assumes $Let\ as\ body = Let\ as'\ body'$
assumes $\bigwedge \pi$.
$supp\ (-\pi)\ \sharp*\ (fv\ (Let\ as\ body) :: var\ set) \Longrightarrow$
$supp\ \pi\ \sharp*\ Let\ as\ body \Longrightarrow$
$F\ (\pi \cdot as)\ (\pi \cdot body)\ (Let\ as\ body) = F\ as\ body\ (Let\ as\ body)$
shows $F\ as\ body\ (Let\ as\ body) = F\ as'\ body'\ (Let\ as'\ body')$
proof –
from *assms(1)*
have $[map\ (\lambda\ p.\ atom\ (fst\ p))\ as]lst.\ (body, as) = [map\ (\lambda\ p.\ atom\ (fst\ p))\ as']lst.\ (body', as')$ **by** *auto*
then obtain p
where $(supp\ (body, as) - atom\ `\ domA\ as)\ \sharp*\ p$
and $[simp]$: $p \cdot body = body'$
and $[simp]$: $p \cdot as = as'$
unfolding *Abs-eq-iff(3) alpha-lst.simps* **by** (*auto simp add*: *domA-def image-image*)

from ‹- ♯* p›
have $*$: $supp\ (-p)\ \sharp*\ (fv\ (Terms.Let\ as\ body) :: var\ set)$
by (*auto simp add*: *fresh-star-def fresh-def supp-finite-set-at-base supp-Pair fv-supp-exp*

fv-supp-heap supp-minus-perm)

from ‹- ♯* p›
have **: *supp p ♯* Terms.Let as body*
by (*auto simp add*: *fresh-star-def fresh-def supp-Pair fv-supp-exp fv-supp-heap*)

have *F as body* (*Let as body*) = *F* (*p* · *as*) (*p* · *body*) (*Let as body*) **by** (*rule assms*(*2*)[*OF* * **, *symmetric*])
also have ... = *F as′ body′* (*Let as′ body′*) **by** (*simp add*: *assms*(*1*))
finally show *?thesis*.
qed

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 *SmartLet* Γ *e* = (*if* Γ = [] *then e else Let* Γ *e*)

lemma *SmartLet-eqvt*[*eqvt*]: π · (*SmartLet* Γ *e*) = *SmartLet* (π · Γ) (π · *e*)
unfolding *SmartLet-def* **by** *perm-simp rule*

lemma *SmartLet-supp*:
supp (*SmartLet* Γ *e*) = (*supp e* ∪ *supp* Γ) − *atom* ‘ (*domA* Γ)
unfolding *SmartLet-def* **using** *Let-supp* **by** (*auto simp add*: *supp-Nil*)

lemma *fv-SmartLet*[*simp*]: *fv* (*SmartLet* Γ *e*) = (*fv* Γ ∪ *fv e*) − *domA* Γ
unfolding *SmartLet-def* **by** *auto*

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*
unfolding *isLam-graph-aux-def eqvt-def*
apply *simp*
apply *simp*
apply (*metis exp-strong-exhaust*)
apply *auto*
done
nominal-termination (*eqvt*) **by** *lexicographic-order*

lemma *isLam-Lam*: *isLam* (*Lam* [*x*]. *e*) **by** *simp*

lemma *isLam-obtain-fresh*:
  assumes *isLam z*
  obtains *y e′*
  where *z* = (*Lam* [*y*]. *e′*) **and** *atom y* ♯ (*c*::′*a*::*fs*)
using *assms* **by** (*nominal-induct z avoiding*: *c rule*:*exp-strong-induct*) *auto*

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*
  unfolding *isVal-graph-aux-def eqvt-def*
  apply *simp*
  apply *simp*
  apply (*metis exp-strong-exhaust*)
  apply *auto*
  done
nominal-termination (*eqvt*) **by** *lexicographic-order*

lemma *isVal-Lam*: *isVal* (*Lam* [*x*]. *e*) **by** *simp*
lemma *isVal-Bool*: *isVal* (*Bool b*) **by** *simp*

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* [] = {} **by** (*auto simp add*: *thunks-def*)

lemma *thunks-domA*: *thunks* Γ ⊆ *domA* Γ
  by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunks-Cons*: *thunks* ((*x*,*e*)#Γ) = (*if isVal e then thunks* Γ − {*x*} *else insert x* (*thunks* Γ))
  by (*auto simp add*: *thunks-def*)

lemma *thunks-append*[*simp*]: *thunks* (Δ@Γ) = *thunks* Δ ∪ (*thunks* Γ − *domA* Δ)
  by (*induction* Δ) (*auto simp add*: *thunks-def*)

lemma *thunks-delete*[*simp*]: *thunks* (*delete x* Γ) = *thunks* Γ − {*x*}
  by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunksI*[*intro*]: *map-of* Γ *x* = *Some e* ⟹ ¬ *isVal e* ⟹ *x* ∈ *thunks* Γ
  by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunksE*[*intro*]: *x* ∈ *thunks* Γ ⟹ *map-of* Γ *x* = *Some e* ⟹ ¬ *isVal e*

by (*induction* Γ) (*auto simp add*: *thunks-def*)

lemma *thunks-cong*: *map-of* Γ = *map-of* Δ ⟹ *thunks* Γ = *thunks* Δ
by (*simp add*: *thunks-def*)

lemma *thunks-eqvt*[*eqvt*]:
$\pi \cdot$ *thunks* Γ = *thunks* ($\pi \cdot$ Γ)
unfolding *thunks-def*
by *perm-simp rule*

3.2.10 Non-recursive Let bindings

definition *nonrec* :: *heap* ⇒ *bool* **where**
nonrec Γ = (∃ *x e*. Γ = [(*x*,*e*)] ∧ *x* ∉ *fv e*)

lemma *nonrecE*:
assumes *nonrec* Γ
obtains *x e* **where** Γ = [(*x*,*e*)] **and** *x* ∉ *fv e*
using *assms*
unfolding *nonrec-def*
by *blast*

lemma *nonrec-eqvt*[*eqvt*]:
nonrec Γ ⟹ *nonrec* ($\pi \cdot$ Γ)
apply (*erule nonrecE*)
apply (*auto simp add*: *nonrec-def fv-def fresh-def*)
apply (*metis fresh-at-base-permute-iff fresh-def*)
done

lemma *exp-induct-rec*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:
assumes ⋀*var*. *P* (*Var var*)
assumes ⋀*exp var*. *P exp* ⟹ *P* (*App exp var*)
assumes ⋀Γ *exp*. ¬ *nonrec* Γ ⟹ (⋀ *x*. *x* ∈ *domA* Γ ⟹ *P* (*the* (*map-of* Γ *x*))) ⟹ *P exp* ⟹ *P* (*Let* Γ *exp*)
assumes ⋀*x e exp*. *x* ∉ *fv e* ⟹ *P e* ⟹ *P exp* ⟹ *P* (*let x be e in exp*)
assumes ⋀*var exp*. *P exp* ⟹ *P* (*Lam* [*var*]. *exp*)
assumes ⋀*b*. *P* (*Bool b*)
assumes ⋀ *scrut e1 e2*. *P scrut* ⟹ *P e1* ⟹ *P e2* ⟹ *P* (*scrut ? e1 : e2*)
shows *P exp*
apply (*rule exp-induct*[*of P*])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*case-tac nonrec* Γ)
apply (*erule nonrecE*)
apply *simp*
apply (*metis assms*(*4*))
apply (*metis assms*(*3*))
apply (*metis assms*(*5*))

apply (*metis assms*(*6*))
apply (*metis assms*(*7*))
done

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* Γ ♯∗ *c* $\Longrightarrow$ ¬ *nonrec* Γ $\Longrightarrow$ ($\bigwedge$*c x. x* ∈ *domA* Γ $\Longrightarrow$ *P c* (*the* (*map-of* Γ *x*))) $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*x e exp c.* {*atom x*} ♯∗ *c* $\Longrightarrow$ *x* ∉ *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*} ♯∗ *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*
apply (*rule exp-strong-induct*[*of P*])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*case-tac nonrec* Γ)
apply (*erule nonrecE*)
apply *simp*
apply (*metis assms*(*4*))
apply (*metis assms*(*3*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply (*metis assms*(*7*))
done

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* Γ ♯∗ *c* $\Longrightarrow$ ¬ *nonrec* Γ $\Longrightarrow$ ($\bigwedge$*c x e.* (*x*,*e*) ∈ *set* Γ $\Longrightarrow$ *P c e*) $\Longrightarrow$ ($\bigwedge$*c. P c exp*) $\Longrightarrow$ *P c* (*Let* Γ *exp*)
assumes $\bigwedge$*x e exp c.* {*atom x*} ♯∗ *c* $\Longrightarrow$ *x* ∉ *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*} ♯∗ *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*
apply (*rule exp-strong-induct-set*(*1*)[*of P*])
apply (*metis assms*(*1*))
apply (*metis assms*(*2*))
apply (*case-tac nonrec* Γ)
apply (*erule nonrecE*)
apply *simp*

apply (*metis assms*(*4*))
apply (*metis assms*(*3*))
apply (*metis assms*(*5*))
apply (*metis assms*(*6*))
apply (*metis assms*(*7*))
done

3.2.11 Renaming a lambda-bound variable

lemma *change-Lam-Variable*:
assumes $y' \neq y \Longrightarrow atom\ y' \sharp (e,\ y)$
shows $Lam\ [y].\ e = Lam\ [y'].\ ((y \leftrightarrow y') \cdot e)$
proof(*cases* $y' = y$)
case *True* **thus** *?thesis* **by** *simp*
next
case *False*
from *assms*[*OF this*]
have $(y \leftrightarrow y') \cdot (Lam\ [y].\ e) = Lam\ [y].\ e$
by −(*rule flip-fresh-fresh*, (*simp add*: *fresh-Pair*)+)
moreover
have $(y \leftrightarrow y') \cdot (Lam\ [y].\ e) = Lam\ [y'].\ ((y \leftrightarrow y') \cdot e)$
by *simp*
ultimately
show $Lam\ [y].\ e = Lam\ [y'].\ ((y \leftrightarrow y') \cdot e)$ **by** (*simp add*: *fresh-Pair*)
qed

end

3.3 Substitution

theory *Substitution*
imports *Terms*
begin

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

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

nominal-function (*default case-sum* (λ*x*. *Inl undefined*) (λ*x*. *Inr undefined*),
invariant λ *a r* . (∀ Γ *y z* . ((*a* = *Inr* (Γ, *y*, *z*) ∧ *atom* ‘ *domA* Γ ♯∗ (*y*, *z*)) ⟶ *map* (λ*x* . *atom* (*fst x*)) (*Sum-Type.projr r*) = *map* (λ*x* . *atom* (*fst x*)) Γ)))
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*])

| $(App\ e\ v)[y ::= z] = App\ (e[y ::= z])\ (v[y ::v= z])$
| $atom$ ‘ $domA\ \Gamma\ \sharp{*}\ (y,z) \Longrightarrow$
$(Let\ \Gamma\ body)[y ::= z] = Let\ (\Gamma[y ::h= z])\ (body[y ::= z])$
| $atom\ x\ \sharp\ (y,z) \Longrightarrow (Lam\ [x].e)[y ::= z] = Lam\ [x].(e[y::=z])$
| $(Bool\ b)[y ::= z] = Bool\ b$
| $(scrut\ ?\ e1\ :\ e2)[y ::= z] = (scrut[y ::= z]\ ?\ e1[y ::= z]\ :\ e2[y ::= z])$
| $[][y ::h= z] = []$
| $((v,e)\#\ \Gamma)[y ::h= z] = (v,\ e[y ::= z])\#\ (\Gamma[y ::h= z])$
proof *goal-cases*

have *eqvt-at-subst*: $\bigwedge\ e\ y\ z$. *eqvt-at subst-subst-heap-sumC* (*Inl* (*e*, *y*, *z*)) $\Longrightarrow$ *eqvt-at* (λ(*a*, *b*, *c*). *subst a b c*) (*e*, *y*, *z*)
apply(*simp add*: *eqvt-at-def subst-def*)
apply(*rule*)
apply(*subst Projl-permute*)
apply(*thin-tac* -)+
apply (*simp add*: *subst-subst-heap-sumC-def*)
apply (*simp add*: *THE-default-def*)
apply (*case-tac Ex1* (*subst-subst-heap-graph* (*Inl* (*e*, *y*, *z*))))
apply(*simp*)
apply(*auto*)[*1*]
apply (*erule-tac x=x* **in** *allE*)
apply *simp*
apply(*cases rule*: *subst-subst-heap-graph.cases*)
apply(*assumption*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*

apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply(*rule-tac x=Sum-Type.projl x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *blast*
apply(*simp* (*no-asm*) *only*: *sum.sel*)
apply (*metis Inr-not-Inl*)
apply (*metis Inr-not-Inl*)
apply(*simp*)
apply(*perm-simp*)
apply(*simp*)
done

have *eqvt-at-subst-heap*: $\bigwedge$ Γ *y z* . *eqvt-at subst-subst-heap-sumC* (*Inr* (Γ, *y*, *z*)) $\Longrightarrow$ *eqvt-at* (λ(*a*, *b*, *c*). *subst-heap a b c*) (Γ, *y*, *z*)
apply(*simp add*: *eqvt-at-def subst-heap-def*)
apply(*rule*)
apply(*subst Projr-permute*)
apply(*thin-tac -*)+
apply (*simp add*: *subst-subst-heap-sumC-def*)
apply (*simp add*: *THE-default-def*)
apply (*case-tac Ex1* (*subst-subst-heap-graph* (*Inr* (Γ, *y*, *z*))))
apply(*simp*)
apply(*auto*)[*1*]
apply (*erule-tac x=x* **in** *allE*)
apply *simp*
apply(*cases rule*: *subst-subst-heap-graph.cases*)
apply(*assumption*)
apply (*metis* (*mono-tags*) *Inr-not-Inl*)+
apply(*rule-tac x=Sum-Type.projr x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *auto*[*1*]
apply(*simp* (*no-asm*) *only*: *sum.sel*)

apply(*rule-tac x=Sum-Type.projr x* **in** *exI*)
apply(*clarify*)
apply (*rule the1-equality*)
apply *auto*[*1*]
apply(*simp* (*no-asm*) *only*: *sum.sel*)

apply(*simp*)
apply(*perm-simp*)
apply(*simp*)
done

{

case *1* **thus** *?case*

unfolding *eqvt-def subst-subst-heap-graph-aux-def*
by *simp*

next case *2* **thus** *?case*
by (*induct rule*: *subst-subst-heap-graph.induct*)(*auto simp add*: *exp-assn.bn-defs fresh-star-insert*)

next case *prems*: (*3 P x*) **show** *?case*
proof(*cases x*)
case (*Inl a*) **thus** *P*
proof(*cases a*)
case (*fields a1 a2 a3*)
thus *P* **using** *Inl prems*
apply (*rule-tac y =a1* **and** *c =*(*a2*, *a3*) **in** *exp-strong-exhaust*)
apply (*auto simp add*: *fresh-star-def*)
done
qed
next
case (*Inr a*) **thus** *P*
proof (*cases a*)
case (*fields a1 a2 a3*)
thus *P* **using** *Inr prems*
by (*metis heapToAssn.cases*)
qed
qed

next case (*19 e y2 z2* Γ *e2 y z as2*) **thus** *?case*
apply −
apply (*drule eqvt-at-subst*)+
apply (*drule eqvt-at-subst-heap*)+
apply (*simp only*: *meta-eq-to-obj-eq*[*OF subst-def*, *symmetric*, *unfolded fun-eq-iff*]
meta-eq-to-obj-eq[*OF subst-heap-def*, *symmetric*, *unfolded fun-eq-iff*])

apply (*auto simp add*: *Abs-fresh-iff*)
apply (*drule-tac*
c = (*y*, *z*) **and**
as = (*map* (λ*x*. *atom* (*fst x*)) *e*) **and**
bs = (*map* (λ*x*. *atom* (*fst x*)) *e2*) **and**
f = λ *a b c* . [*a*]*lst*. (*subst* (*fst b*) *y z*, *subst-heap* (*snd b*) *y z*) **in** *Abs-lst-fcb2*)
apply (*simp add*: *perm-supp-eq fresh-Pair fresh-star-def Abs-fresh-iff*)
apply (*metis domA-def image-image image-set*)
apply (*metis domA-def image-image image-set*)
apply (*simp add*: *eqvt-at-def*, *simp add*: *fresh-star-Pair perm-supp-eq*)
apply (*simp add*: *eqvt-at-def*, *simp add*: *fresh-star-Pair perm-supp-eq*)
apply (*simp add*: *eqvt-at-def*)
done

next case (*25 x2 y2 z2 e2 x y z e*) **thus** *?case*

apply −
apply (*drule eqvt-at-subst*)+
apply (*simp only*: *Abs-fresh-iff meta-eq-to-obj-eq*[*OF subst-def*, *symmetric*, *unfolded fun-eq-iff*])

apply (*simp add*: *eqvt-at-def*)
apply *rule*
apply (*erule-tac x* = (*x2* ↔ *c*) **in** *allE*)
apply (*erule-tac x* = (*x* ↔ *c*) **in** *allE*)
apply *auto*
done
}
qed(*auto*)

nominal-termination (*eqvt*) **by** *lexicographic-order*

lemma shows
True **and** *bn-subst*[*simp*]: *domA* (*subst-heap* Γ *y z*) = *domA* Γ
by(*induct rule*:*subst-subst-heap.induct*)
(*auto simp add*: *exp-assn.bn-defs fresh-star-insert*)

lemma *subst-noop*[*simp*]:
shows *e*[*y* ::= *y*] = *e* **and** Γ[*y*::*h*=*y*]= Γ
by(*induct e y y* **and** Γ *y y rule*:*subst-subst-heap.induct*)
(*auto simp add*:*fresh-star-Pair exp-assn.bn-defs*)

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*]
using *assms*
by(*induct e y z* **and** Γ *y z rule*:*subst-subst-heap.induct*)
(*auto simp add*:*fresh-at-base fresh-star-Pair fresh-star-insert fresh-Nil fresh-Cons pure-fresh*)

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*])
by(*induct e y z* **and** Γ *y z rule*:*subst-subst-heap.induct*)
(*auto simp add*:*fresh-star-Pair exp-assn.bn-defs fresh-Cons fresh-Nil pure-fresh*)

lemma *subst-fresh-noop*: *atom x* ♯ *e* ⟹ *e*[*x* ::= *y*] = *e*
and *subst-heap-fresh-noop*: *atom x* ♯ Γ ⟹ Γ[*x* ::*h*= *y*] = Γ
by (*nominal-induct e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-def fresh-Pair fresh-at-base fresh-Cons simp del*: *exp-assn.eq-iff*)

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 $\sharp*$ $y \Longrightarrow$ *supp* $(\Gamma[y{::}h{=}x]) = ($*supp* Γ $-$ $\{$*atom* $y\}) \cup ($*if* *atom* $y \in$ *supp* Γ *then* $\{$*atom* $x\}$ *else* $\{\})$
by (*nominal-induct* *e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-def fresh-Pair supp-Nil supp-Cons supp-Pair fresh-Cons exp-assn.supp Let-supp supp-at-base pure-supp simp del*: *exp-assn.eq-iff*)

lemma *supp-subst*: *supp* $(e[y{::=}x]) \subseteq ($*supp* e $-$ $\{$*atom* $y\}) \cup \{$*atom* $x\}$
using *supp-subst-eq* **by** *auto*

lemma *fv-subst-eq*: *fv* $(e[y{::=}x]) = ($*fv* $e - \{y\}) \cup ($*if* $y \in$ *fv* e *then* $\{x\}$ *else* $\{\})$
and *atom* ' *domA* Γ $\sharp*$ $y \Longrightarrow$ *fv* $(\Gamma[y{::}h{=}x]) = ($*fv* $\Gamma - \{y\}) \cup ($*if* $y \in$ *fv* Γ *then* $\{x\}$ *else* $\{\})$
by (*nominal-induct* *e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-def fresh-Pair supp-Nil supp-Cons supp-Pair fresh-Cons exp-assn.supp Let-supp supp-at-base simp del*: *exp-assn.eq-iff*)

lemma *fv-subst-subset*: *fv* $(e[y ::= x]) \subseteq ($*fv* $e - \{y\}) \cup \{x\}$
using *fv-subst-eq* **by** *auto*

lemma *fv-subst-int*: $x \notin S \Longrightarrow y \notin S \Longrightarrow$ *fv* $(e[y ::= x]) \cap S =$ *fv* $e \cap S$
by (*auto simp add*: *fv-subst-eq*)

lemma *fv-subst-int2*: $x \notin S \Longrightarrow y \notin S \Longrightarrow S \cap$ *fv* $(e[y ::= x]) = S \cap$ *fv* e
by (*auto simp add*: *fv-subst-eq*)

lemma *subst-swap-same*: *atom* $x \sharp e \Longrightarrow (x \leftrightarrow y) \bullet e = e[y ::=x]$
and *atom* $x \sharp \Gamma \Longrightarrow$ *atom* '*domA* Γ $\sharp*$ $y \Longrightarrow (x \leftrightarrow y) \bullet \Gamma = \Gamma[y ::h= x]$
by (*nominal-induct* *e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-Pair fresh-star-at-base fresh-Cons pure-fresh permute-pure simp del*: *exp-assn.eq-iff*)

lemma *subst-subst-back*: *atom* $x \sharp e \Longrightarrow e[y{::=}x][x{::=}y] = e$
and *atom* $x \sharp \Gamma \Longrightarrow$ *atom* '*domA* Γ $\sharp*$ $y \Longrightarrow \Gamma[y{::}h{=}x][x{::}h{=}y] = \Gamma$
by(*nominal-induct* *e* **and** Γ *avoiding*: *x y rule*:*exp-heap-strong-induct*)
(*auto simp add*: *fresh-star-Pair fresh-star-at-base fresh-star-Cons fresh-Cons exp-assn.bn-defs simp del*: *exp-assn.eq-iff*)

lemma *subst-heap-delete*[*simp*]: (*delete* x $\Gamma)[y ::h= z] =$ *delete* x $(\Gamma[y ::h= z])$
by (*induction* Γ) *auto*

lemma *subst-nil-iff* [*simp*]: $\Gamma[x ::h= z] = [] \longleftrightarrow \Gamma = []$
by (*cases* Γ) *auto*

lemma *subst-SmartLet*[*simp*]:
atom ' *domA* Γ $\sharp*$ $(y,z) \Longrightarrow$ (*SmartLet* Γ *body*)$[y ::= z] =$ *SmartLet* $(\Gamma[y ::h= z])$ (*body*$[y ::= z]$)
unfolding *SmartLet-def* **by** *auto*

lemma *subst-let-be*[*simp*]:
atom x' ♯ y ⟹ *atom* x' ♯ x ⟹ (*let* x' *be* e *in* *exp*)[y::=x] = (*let* x' *be* e[y::=x] *in* *exp*[y::=x])
by (*simp add*: *fresh-star-def fresh-Pair*)

lemma *isLam-subst*[*simp*]: *isLam* e[x::=y] = *isLam* e
by (*nominal-induct* e *avoiding*: x y *rule*: *exp-strong-induct*)
(*auto simp add*: *fresh-star-Pair*)

lemma *isVal-subst*[*simp*]: *isVal* e[x::=y] = *isVal* e
by (*nominal-induct* e *avoiding*: x y *rule*: *exp-strong-induct*)
(*auto simp add*: *fresh-star-Pair*)

lemma *thunks-subst*[*simp*]:
thunks Γ[y::h=x] = *thunks* Γ
by (*induction* Γ) (*auto simp add*: *thunks-Cons*)

lemma *map-of-subst*:
map-of (Γ[x::h=y]) k = *map-option* (λ e . e[x::=y]) (*map-of* Γ k)
by (*induction* Γ) *auto*

lemma *mapCollect-subst*[*simp*]:
{e k v | k↦v∈*map-of* Γ[x::h=y]} = {e k v[x::=y] | k↦v∈*map-of* Γ}
by (*auto simp add*: *map-of-subst*)

lemma *subst-eq-Cons*:
Γ[x::h=y] = (x', e)#Δ ⟷ (∃ e' Γ′. Γ = (x',e')#Γ′ ∧ e'[x::=y] = e ∧ Γ′[x::h=y] = Δ)
by (*cases* Γ) *auto*

lemma *nonrec-subst*:
atom ‘ *domA* Γ ♯* x ⟹ *atom* ‘ *domA* Γ ♯* y ⟹ *nonrec* Γ[x::h=y] ⟷ *nonrec* Γ
by (*auto simp add*: *nonrec-def fresh-star-def subst-eq-Cons fv-subst-eq*)

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* ⇒ *exp* ⇒ *var list* ⇒ *heap* ⇒ *exp* ⇒ *bool*
(‹- : - $\Downarrow_{-}$ - : -› [*50*,*50*,*50*,*50*] *50*)
where
Lambda:
Γ : (*Lam* [x]. e) $\Downarrow_L$ Γ : (*Lam* [x]. e)
| *Application*: ⟦
atom y ♯ (Γ,e,x,L,Δ,Θ,z) ;
Γ : e $\Downarrow_L$ Δ : (*Lam* [y]. e');
Δ : $e'[y ::= x]$ $\Downarrow_L$ Θ : z
⟧ ⟹
Γ : *App* e x $\Downarrow_L$ Θ : z
| *Variable*: ⟦
map-of Γ x = *Some* e; *delete* x Γ : e $\Downarrow_{x\#L}$ Δ : z
⟧ ⟹
Γ : *Var* x $\Downarrow_L$ (x, z) # Δ : z
| *Let*: ⟦
atom ‘ *domA* Δ ♯* (Γ, L);
Δ @ Γ : *body* $\Downarrow_L$ Θ : z
⟧ ⟹
Γ : *Let* Δ *body* $\Downarrow_L$ Θ : z
| *Bool*:
Γ : *Bool* b $\Downarrow_L$ Γ : *Bool* b
| *IfThenElse*: ⟦
Γ : *scrut* $\Downarrow_L$ Δ : (*Bool* b);
Δ : (*if* b *then* e_1 *else* e_2) $\Downarrow_L$ Θ : z
⟧ ⟹
Γ : (*scrut* ? e_1 : e_2) $\Downarrow_L$ Θ : z

equivariance *reds*

nominal-inductive *reds*
avoids *Application*: y
by (*auto simp add*: *fresh-star-def fresh-Pair*)

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)
apply(*auto intro*!: *Lambda Application Variable Let*
simp add: *fresh-Pair fresh-Cons fresh-Nil fresh-star-def*)
done

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)
by (*auto intro*!: *Lambda Application Variable Let simp add*: *fresh-Pair fresh-at-base fresh-Cons fresh-Nil fresh-star-def pure-fresh*)

3.4.3 Better introduction rules

This variant do not require freshness.

lemma *reds-ApplicationI*:
  assumes Γ : $e \Downarrow_L \Delta$: *Lam* $[y]$. e'
  assumes Δ : $e'[y::=x] \Downarrow_L \Theta$: z
  shows Γ : *App* e $x \Downarrow_L \Theta$: z
proof−
  obtain y' :: *var* **where** *atom* $y' \sharp (\Gamma, e, x, L, \Delta, \Theta, z, e')$ **by** (*rule obtain-fresh*)

  have *a*: *Lam* $[y']$. $((y' \leftrightarrow y) \cdot e')$ = *Lam* $[y]$. e'
    using ‹*atom* $y' \sharp$ -›
    by (*auto simp add*: *Abs1-eq-iff fresh-Pair fresh-at-base*)

  have *b*: $((y' \leftrightarrow y) \cdot e')[y'::=x] = e'[y::=x]$
  proof(*cases* $x = y$)
    case *True*
    have *atom* $y' \sharp e'$ **using** ‹*atom* $y' \sharp$ -› **by** *simp*
    thus *?thesis*
      by (*simp add*: *True subst-swap-same subst-subst-back*)
  next
    case *False*
    hence *atom* $y \sharp x$ **by** *simp*

    have [*simp*]: $(y' \leftrightarrow y) \cdot x = x$ **using** ‹*atom* $y \sharp$ -› ‹*atom* $y' \sharp$ -›
        by (*simp add*: *flip-fresh-fresh fresh-Pair fresh-at-base*)

    have $((y' \leftrightarrow y) \cdot e')[y'::=x] = (y' \leftrightarrow y) \cdot (e'[y::=x])$ **by** *simp*
    also have $\ldots = e'[y::=x]$
      using ‹*atom* $y \sharp$ -› ‹*atom* $y' \sharp$ -›
      by (*simp add*: *flip-fresh-fresh fresh-Pair fresh-at-base subst-pres-fresh*)
    finally
    show *?thesis*.
  qed
  have *atom* $y' \sharp (\Gamma, e, x, L, \Delta, \Theta, z)$ **using** ‹*atom* $y' \sharp$ -› **by** (*simp add*: *fresh-Pair*)
  from *this assms*[*folded a b*]
  show *?thesis* **..**
qed

lemma *reds-SmartLet*: ⟦
    atom ‘ *domA* $\Delta \sharp{*} (\Gamma, L)$;
    Δ @ Γ : *body* $\Downarrow_L \Theta$: z
  ⟧ $\Longrightarrow$
    Γ : *SmartLet* Δ *body* $\Downarrow_L \Theta$: z
unfolding *SmartLet-def*
by (*auto intro*: *reds.Let*)

A single rule for values

lemma *reds-isValI*:
  isVal z $\Longrightarrow$ Γ : *z* $\Downarrow_L$ Γ : *z*
by (*cases z rule*:*isVal.cases*) (*auto intro*: *reds.intros*)

3.4.4 Properties of the semantics

Heap entries are never removed.

lemma *reds-doesnt-forget*:
  Γ : *e* $\Downarrow_L$ Δ : *z* $\Longrightarrow$ *domA* $\Gamma \subseteq$ *domA* Δ
by(*induct rule*: *reds.induct*) *auto*

Live variables are not added to the heap.

lemma *reds-avoids-live'*:
 assumes Γ : *e* $\Downarrow_L$ Δ : *z*
 shows (*domA* Δ − *domA* Γ) $\cap$ *set L* = {}
using *assms*
by(*induct rule*:*reds.induct*)
  (*auto dest*: *map-of-domAD fresh-distinct-list simp add*: *fresh-star-Pair*)

lemma *reds-avoids-live*:
  ⟦ Γ : *e* $\Downarrow_L$ Δ : *z*;
   x $\in$ *set L*;
   x $\notin$ *domA* Γ
  ⟧ $\Longrightarrow$ *x* $\notin$ *domA* Δ
using *reds-avoids-live'* **by** *blast*

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

lemma *reds-fresh*: ⟦ Γ : *e* $\Downarrow_L$ Δ : *z*;
   atom (*x*::*var*) ♯ (Γ, *e*)
  ⟧ $\Longrightarrow$ *atom x* ♯ (Δ, *z*) $\vee$ *x* $\in$ (*domA* Δ − *set L*)
proof(*induct rule*: *reds.induct*)
case (*Lambda* Γ *x e*) **thus** *?case* **by** *auto*
next
case (*Application y* Γ *e x' L* Δ Θ *z e'*)
 hence *atom x* ♯ (Δ, *Lam* [*y*]. *e'*) $\vee$ *x* $\in$ *domA* Δ − *set* (*x'* # *L*) **by** (*auto simp add*: *fresh-Pair*)

  thus *?case*
  proof
   assume *atom x* ♯ (Δ, *Lam* [*y*]. *e'*)
   hence *atom x* ♯ *e'*[*y* ::= *x'*]
    using *Application.prems*
    by (*auto intro*: *subst-pres-fresh simp add*: *fresh-Pair*)
   thus *?thesis* **using** *Application.hyps*(*5*) ‹*atom x* ♯ (Δ, *Lam* [*y*]. *e'*)› **by** *auto*
  next
   assume *x* $\in$ *domA* Δ − *set* (*x'* # *L*)
   thus *?thesis* **using** *reds-doesnt-forget*[*OF Application.hyps*(*4*)] **by** *auto*
  qed

next

case(*Variable* Γ v e L Δ z)
  have *atom* x ♯ Γ **and** *atom* x ♯ v **using** *Variable.prems*(*1*) **by** (*auto simp add*: *fresh-Pair*)
  from *fresh-delete*[*OF this*(*1*)]
  have *atom* x ♯ *delete* v Γ**.**
  moreover
  have $v \in$ *domA* Γ **using** *Variable.hyps*(*1*) **by** (*metis domA-from-set map-of-SomeD*)
  from *fresh-map-of*[*OF this* ‹*atom* x ♯ Γ›]
  have *atom* x ♯ *the* (*map-of* Γ v)**.**
  hence *atom* x ♯ e **using** ‹*map-of* Γ v = *Some* e› **by** *simp*
  ultimately
  have *atom* x ♯ (Δ, z) $\vee$ $x \in$ *domA* Δ − *set* (v # L) **using** *Variable.hyps*(*3*) **by** (*auto simp add*: *fresh-Pair*)
  thus *?case* **using** ‹*atom* x ♯ v› **by** (*auto simp add*: *fresh-Pair fresh-Cons fresh-at-base*)
next

case (*Bool* Γ b L)
  thus *?case* **by** *auto*
next

case (*IfThenElse* Γ *scrut* L Δ b e_1 e_2 Θ z)
  from ‹*atom* x ♯ (Γ, *scrut* ? e_1 : e_2)›
  have *atom* x ♯ (Γ, *scrut*) **and** *atom* x ♯ (e_1, e_2) **by** (*auto simp add*: *fresh-Pair*)
  from *IfThenElse.hyps*(*2*)[*OF this*(*1*)]
  show *?case*
  proof
    assume *atom* x ♯ (Δ, *Bool* b) **with** ‹*atom* x ♯ (e_1, e_2)›
    have *atom* x ♯ (Δ, *if* b *then* e_1 *else* e_2) **by** *auto*
    from *IfThenElse.hyps*(*4*)[*OF this*]
    show *?thesis***.**
  next
    assume $x \in$ *domA* Δ − *set* L
    with *reds-doesnt-forget*[*OF* ‹Δ : (*if* b *then* e_1 *else* e_2) $\Downarrow_L$ Θ : z›]
    show *?thesis* **by** *auto*
  qed
next

case (*Let* Δ Γ L *body* Θ z)
  show *?case*
  proof (*cases* $x \in$ *domA* Δ)
    case *False*
      hence *atom* x ♯ Δ **using** *Let.prems* **by**(*auto simp add*: *fresh-Pair*)
      show *?thesis*
        apply(*rule Let.hyps*(*3*))
        using *Let.prems* ‹*atom* x ♯ Δ› *False*
        by (*auto simp add*: *fresh-Pair fresh-append*)
  next
    case *True*

    hence $x \notin$ *set L*
      using *Let*(*1*)
      by (*metis fresh-PairD*(*2*) *fresh-star-def image-eqI set-not-fresh*)
    with *True*
    show *?thesis*
    using *reds-doesnt-forget*[*OF Let.hyps*(*2*)] **by** *auto*
  qed
qed

lemma *reds-fresh-fv*: $[\![\ \Gamma : e \Downarrow_L \Delta : z;$
   $x \in fv\ (\Delta,\ z) \wedge (x \notin domA\ \Delta \vee\ x \in set\ L)$
  $]\!] \Longrightarrow x \in fv\ (\Gamma,\ e)$
using *reds-fresh*
unfolding *fv-def fresh-def*
by *blast*

lemma *new-free-vars-on-heap*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  shows $fv\ (\Delta,\ z) - domA\ \Delta \subseteq fv\ (\Gamma,\ e) - domA\ \Gamma$
using *reds-fresh-fv*[*OF assms*(*1*)] *reds-doesnt-forget*[*OF assms*(*1*)] **by** *auto*

lemma *reds-pres-closed*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and $fv\ (\Gamma,\ e) \subseteq set\ L \cup domA\ \Gamma$
  shows $fv\ (\Delta,\ z) \subseteq set\ L \cup domA\ \Delta$
using *new-free-vars-on-heap*[*OF assms*(*1*)] *assms*(*2*) **by** *auto*

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(*nominal-induct avoiding* : L' *rule*: *reds.strong-induct*)
case (*Lambda* Γ *x e L* L')
  show *?case*
    by (*rule reds.Lambda*)
next
case (*Application y* Γ *e xa L* Δ Θ *z* e' L')
 from *Application.hyps*(*10*)[*OF Application.prems*] *Application.hyps*(*12*)[*OF Application.prems*]
  show *?case*
    by (*rule reds-ApplicationI*)
next
case (*Variable* Γ *xa e L* Δ *z* L')
  have $set\ (xa\ \#\ L') \subseteq set\ (xa\ \#\ L)$
    using *Variable.prems* **by** *auto*
  thus *?case*
    by (*rule reds.Variable*[*OF Variable*(*1*) *Variable.hyps*(*3*)])
next
case (*Bool b*)
  show *?case*..

next
case (*IfThenElse* Γ *scrut* L Δ b e_1 e_2 Θ z L')
  thus *?case* **by** (*metis reds.IfThenElse*)
next
case (*Let* Δ Γ L *body* Θ z L')
  have *atom* ‘ *domA* Δ $\sharp*$ (Γ, L')
    using *Let*(*1*−*3*) *Let.prems*
    by (*auto simp add*: *fresh-star-Pair fresh-star-set-subset*)
  thus *?case*
    by (*rule reds.Let*[*OF* - *Let.hyps*(*4*)[*OF Let.prems*]])
qed

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*
by (*induct* Γ e L Δ z *rule*:*reds.induct*) (*auto dest*: *reds-doesnt-forget*)

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−
  from *assms*
  have *isVal z* **by** (*rule result-evaluated*)
  hence ($\exists$ y e'. z = *Lam* [y]. e' $\wedge$ *atom* y $\sharp$ c) $\vee$ ($\exists$ b. z = *Bool* b)
    by (*nominal-induct* z *avoiding*: c *rule*:*exp-strong-induct*) *auto*
  thus *thesis* **using** *that* **by** *blast*
qed

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* → *Value*) | *B* (**lazy** *bool discr*)

fixrec *Fn-project* :: *Value* → *Value* → *Value*
 where *Fn-project*·(*Fn*·*f*) = *f*

abbreviation *Fn-project-abbr* (**infix** ‹↓*Fn*› *55*)
  where f ↓*Fn* $v \equiv$ *Fn-project*·f·v

lemma [*simp*]:
  $\bot$ ↓*Fn* $x = \bot$
  (B·b) ↓*Fn* $x = \bot$
  by (*fixrec-simp*)+

fixrec *B-project* :: *Value* → *Value* → *Value* → *Value* **where**
  B-project·(B·*db*)·v_1·v_2 = (*if undiscr db then* v_1 *else* v_2)

lemma [*simp*]:
  B-project·(B·(*Discr b*))·v_1·v_2 = (*if b then* v_1 *else* v_2)
  B-project·$\bot$·v_1·v_2 = $\bot$
  B-project·(*Fn*·f)·v_1·v_2 = $\bot$
by *fixrec-simp*+

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 = (λ m. (*if* $m < n$ *then* $\bot$ *else* B·b)) |
      n Y' **where** Y = (λ m. (*if* $m < n$ *then* $\bot$ *else* *Fn*·(Y' ($m-n$)))) *chain* Y'
proof(*cases* $Y = (\lambda\ \text{-}\ .\ \bot)$)
  case *True*
  thus *?thesis* **by** (*rule that*(*1*))
next
  case *False*
  hence $\exists\ i.\ Y\ i \neq \bot$ **by** *auto*
  hence $\exists\ n.\ Y\ n \neq \bot \wedge (\forall m.\ Y\ m \neq \bot \longrightarrow m \geq n)$
    by (*rule exE*)(*rule ex-has-least-nat*)
  then obtain n **where** $Y\ n \neq \bot$ **and** $\forall m.\ m < n \longrightarrow Y\ m = \bot$ **by** *fastforce*
  hence ($\exists$ f. Y n = *Fn*·f) $\vee$ ($\exists$ b. Y n = B·b) **by** (*metis Value.exhaust*)
  thus *?thesis*
  proof
    assume ($\exists$ f. Y n = *Fn*·f)
    then obtain f **where** Y n = *Fn* · f **by** *blast*
    {
      fix i
      from ‹*chain Y*› **have** $Y\ n \sqsubseteq Y\ (i+n)$ **by** (*metis chain-mono le-add2*)
      with ‹$Y\ n$ = -›
      have $\exists$ g. (Y (i+n) = *Fn* · g)
        by (*metis Value.dist-les*(*1*) *Value.exhaust below-bottom-iff*)
    }
    then obtain Y' **where** Y': $\bigwedge$ i. Y ($i + n$) = *Fn* · (Y' i) **by** *metis*

    have Y = (λm. *if* $m < n$ *then* $\bot$ *else* *Fn*·(Y' ($m - n$)))
      using ‹$\forall m$. -› Y' **by** (*metis add-diff-inverse add.commute*)
    moreover

have *chain Y′* **using** ‹*chain Y*›
by (*auto intro*!:*chainI elim*: *chainE simp add*: *Value.inverts*[*symmetric*] *Y′*[*symmetric*] *simp del*: *Value.inverts*)
ultimately
show *?thesis* **by** (*rule that*(*3*))
next
assume (∃ *b*. *Y n* = *B*·*b*)
then obtain *b* **where** *Y n* = *B*·*b* **by** *blast*
{
fix *i*
from ‹*chain Y*› **have** *Y n* ⊑ *Y* (*i*+*n*) **by** (*metis chain-mono le-add2*)
with ‹*Y n* = -›
have *Y* (*i*+*n*) = *B*·*b*
by (*metis Value.dist-les*(*2*) *Value.exhaust Value.inverts*(*2*) *below-bottom-iff discrete-cpo*)
}
hence *Y′*: ⋀ *i*. *Y* (*i* + *n*) = *B*·*b* **by** *metis*

have *Y* = (λ*m*. *if m* < *n then* ⊥ *else B*·*b*)
using ‹∀ *m*. -› *Y′* **by** (*metis add-diff-inverse add.commute*)
thus *?thesis* **by** (*rule that*(*2*))
qed
qed

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* • (*v*::*Value*) = *v*
instance
apply *standard*
apply (*auto simp add*: *permute-Value-def*)
done
end

instance *Value* :: *pcpo-pt*
by *standard* (*simp add*: *pure-permute-id*)

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** $y \neq x$ **using** *ne* **by** *blast*
  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 $(fix \cdot F)\ y = (F \cdot (fix \cdot F))\ y$
    by (*subst fix-eq*, *rule*)

  lemma *R-S*: $\bigwedge\ y.\ y \in S \Longrightarrow (fix \cdot R)\ y = (e1 \cdot (fix \cdot (H\ (fix \cdot R))))\ y$
    by (*case-tac y rule*: *split-x*) *simp-all*

  lemma *R'-S*: $\bigwedge\ y.\ y \in S \Longrightarrow (fix \cdot R')\ y = (e1 \cdot (fix \cdot (H\ (fix \cdot R'))))\ y$
    by (*case-tac y rule*: *split-x*) *simp-all*

  lemma *HR-is-R*[*simp*]: $fix{\cdot}(H\ (fix \cdot R)) = fix \cdot R$
    by (*rule eq*) *simp-all*

  lemma *HR'-is-R'*[*simp*]: $fix \cdot (H\ (fix \cdot R')) = fix \cdot R'$
    by (*rule eq*) *simp-all*

  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 $H\ \varrho' \cdot \varrho'' \sqsubseteq \varrho'$
      using *assms*

by −(*rule below*, *simp-all*)

lemma *HL-is-L*[*simp*]: *fix* · (*H* (*fix* · *L*)) = *fix* · *L*
proof (*rule below-antisym*)
show *fix* · (*H* (*fix* · *L*)) ⊑ *fix* · *L*
by (*rule fix-least-below*[*OF H-noop*]) *simp*
hence ∗: *e2* · (*fix* · (*H* (*fix* · *L*))) ⊑ *e2* · (*fix* · *L*) **by** (*rule monofun-cfun-arg*)

show *fix* · *L* ⊑ *fix* · (*H* (*fix* · *L*))
by (*rule fix-least-below*[*OF below*]) (*simp-all add*: *ne* ∗)
qed

lemma *iterative-override-on*:
shows *fix* · *L* = *fix* · *R*
proof(*rule below-antisym*)
show *fix* · *R* ⊑ *fix* · *L*
by (*rule fix-least-below*[*OF below*]) *simp-all*

show *fix* · *L* ⊑ *fix* · *R*
apply (*rule fix-least-below*[*OF below*])
apply *simp*
apply (*simp del*: *lookup-fix add*: *R-S*)
apply *simp*
done
qed

lemma *iterative-override-on′*:
shows *fix* · *L* = *fix* · *R′*
proof(*rule below-antisym*)
show *fix* · *R′* ⊑ *fix* · *L*
by (*rule fix-least-below*[*OF below*]) *simp-all*

show *fix* · *L* ⊑ *fix* · *R′*
apply (*rule fix-least-below*[*OF below*])
apply *simp*
apply (*simp del*: *lookup-fix add*: *R′-S*)
apply *simp*
done
qed
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** $⟦e⟧_ϱ$ ≡ *ESem e* · *ϱ*
end

locale *has-ignore-fresh-ESem* = *has-ESem* +
  assumes *fv-supp*: *supp e* = *atom* ‘ (*fv e* :: *'b set*)
  assumes *ESem-considers-fv*: $⟦ e ⟧_ϱ$ = $⟦ e ⟧_{ϱ\ f|` (fv\ e)}$

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* Γ *ϱ* ≡ *evalHeap* Γ (λ *e*. $⟦e⟧_ϱ$)

definition
  HSem :: (*'var* × *'exp*) *list* ⇒ (*'var* ⇒ *'value*) → (*'var* ⇒ *'value*)
  where *HSem* Γ = (Λ *ϱ* . (μ *ϱ'*. $ϱ ++_{domA\ Γ} ⟦Γ⟧_{ϱ'}$))

abbreviation *HSem-syn* (‹⦃ - ⦄-› [*0*,*60*] *60*)
  where ⦃Γ⦄*ϱ* ≡ *HSem* Γ · *ϱ*

lemma *HSem-def'*: ⦃Γ⦄*ϱ* = (μ *ϱ'*. $ϱ ++_{domA\ Γ} ⟦Γ⟧_{ϱ'}$)
  unfolding *HSem-def* **by** *simp*

5.3.2 Induction and other lemmas about *HSem*

lemma *HSem-ind*:
  assumes *adm P*
  assumes *P* ⊥
  assumes *step*: ⋀ *ϱ'*. *P ϱ'* ⟹ *P* ($ϱ ++_{domA\ Γ} ⟦Γ⟧_{ϱ'}$)
  shows *P* (⦃Γ⦄*ϱ*)
  unfolding *HSem-def'*
  apply (*rule fix-ind*[*OF assms*(*1*), *OF assms*(*2*)])
  using *step* **by** *simp*

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 (*rule HSem-ind*, *goal-cases*)
  case *1* **show** *?case* **by** (*auto*)
next
  case *2* **show** *?case* **by** (*rule minimal*)
next
  case ($3\ \varrho'$)
    show *?case*
    by (*rule override-on-belowI*)
      (*auto simp add*: *lookupEvalHeap below-trans*[*OF monofun-cfun-arg*[*OF* ‹$\varrho' \sqsubseteq r$›] *h*] *rho*)
qed

lemma *HSem-bot-below*:
  assumes *h*: $\bigwedge x.\ x \in domA\ h \Longrightarrow [\![the\ (map\text{-}of\ h\ x)]\!]_r \sqsubseteq r\ x$
  shows $\{\!|h|\!\}\bot \sqsubseteq r$
  using *assms*
by (*metis HSem-below fun-belowD minimal*)

lemma *HSem-bot-ind*:
  assumes *adm P*
  assumes $P\ \bot$
  assumes *step*: $\bigwedge\ \varrho'.\ P\ \varrho' \Longrightarrow P\ ([\![\ \Gamma\ ]\!]_{\varrho'})$
  shows $P\ (\{\!|\Gamma|\!\}\bot)$
  apply (*rule HSem-ind*[*OF assms*(*1*,*2*)])
  apply (*drule assms*(*3*))
  apply *simp*
  done

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 $P\ (\{\!|\Gamma_1|\!\}\varrho_1)\ (\{\!|\Gamma_2|\!\}\varrho_2)$
  unfolding *HSem-def'*
  apply (*rule parallel-fix-ind*[*OF assms*(*1*), *OF assms*(*2*)])
  using *step* **by** *simp*

lemma *HSem-eq*:
  shows $\{\!|\Gamma|\!\}\varrho = \varrho\ {+}{+}_{domA\ \Gamma}\ [\![\Gamma]\!]_{\{\!|\Gamma|\!\}\varrho}$
  unfolding *HSem-def'*
  by (*subst fix-eq*) *simp*

lemma *HSem-bot-eq*:
  shows $\{\!|\Gamma|\!\}\bot = [\![\Gamma]\!]_{\{\!|\Gamma|\!\}\bot}$
  by (*subst HSem-eq*) *simp*

lemma *lookup-HSem-other*:
  assumes $y \notin domA\ h$
  shows $(\{\!|h|\!\}\varrho)\ y = \varrho\ y$
  apply (*subst HSem-eq*)
  using *assms* **by** *simp*

lemma *lookup-HSem-heap*:
  assumes $y \in domA\ h$
  shows $(\{\!|h|\!\}\varrho)\ y = [\![\ the\ (map\text{-}of\ h\ y)\ ]\!]_{\{\!|h|\!\}\varrho}$
  apply (*subst HSem-eq*)
  using *assms* **by** (*simp add*: *lookupEvalHeap*)

lemma *HSem-edom-subset*: $edom\ (\{\!|\Gamma|\!\}\varrho) \subseteq edom\ \varrho \cup domA\ \Gamma$
  apply *rule*
  unfolding *edomIff*
  apply (*case-tac* $x \in domA\ \Gamma$)
  apply (*auto simp add*: *lookup-HSem-other*)
  done

lemma (**in** −) *env-restr-override-onI*: $-S2 \subseteq S \Longrightarrow env\text{-}restr\ S\ \varrho 1 \ {++}_{S2}\ \varrho 2 = \varrho 1\ {++}_{S2}\ \varrho 2$
  by (*rule ext*) (*auto simp add*: *lookup-override-on-eq*)

lemma *HSem-restr*:
  $\{\!|h|\!\}(\varrho\ f|\text{`}\ (-\ domA\ h)) = \{\!|h|\!\}\varrho$
  apply (*rule parallel-HSem-ind*)
  apply *simp*
  apply *auto*[*1*]
  apply (*subst env-restr-override-onI*)
  apply *simp-all*
  done

lemma *HSem-restr-cong*:
  assumes $\varrho\ f|\text{`}\ (-\ domA\ h) = \varrho'\ f|\text{`}\ (-\ domA\ h)$
  shows $\{\!|h|\!\}\varrho = \{\!|h|\!\}\varrho'$
  apply (*subst* (*1 2*) *HSem-restr*[*symmetric*])
  by (*simp add*: *assms*)

lemma *HSem-restr-cong-below*:
  assumes $\varrho\ f|\text{`}\ (-\ domA\ h) \sqsubseteq \varrho'\ f|\text{`}\ (-\ domA\ h)$
  shows $\{\!|h|\!\}\varrho \sqsubseteq \{\!|h|\!\}\varrho'$
  by (*subst* (*1 2*) *HSem-restr*[*symmetric*]) (*rule monofun-cfun-arg*[*OF assms*])

lemma *HSem-reorder*:
  assumes $map\text{-}of\ \Gamma = map\text{-}of\ \Delta$
  shows $\{\!|\Gamma|\!\}\varrho = \{\!|\Delta|\!\}\varrho$
by (*simp add*: *HSem-def′ evalHeap-reorder*[*OF assms*] *assms dom-map-of-conv-domA*[*symmetric*])

lemma *HSem-reorder-head*:

assumes $x \neq y$
shows $\{\!\!|(x,e1)\#(y,e2)\#\Gamma|\!\!\}\varrho = \{\!\!|(y,e2)\#(x,e1)\#\Gamma|\!\!\}\varrho$
proof−
have *set* $((x,e1)\#(y,e2)\#\Gamma) = set\ ((y,e2)\#(x,e1)\#\Gamma)$
by *auto*
thus *?thesis*
unfolding *HSem-def evalHeap-reorder-head*[*OF assms*]
by (*simp add*: *domA-def*)
qed

lemma *HSem-reorder-head-append*:
assumes $x \notin domA\ \Gamma$
shows $\{\!\!|(x,e)\#\Gamma@\Delta|\!\!\}\varrho = \{\!\!|\Gamma\ @\ ((x,e)\#\Delta)|\!\!\}\varrho$
proof−
have *set* $((x,e)\#\Gamma@\Delta) = set\ (\Gamma\ @\ ((x,e)\#\Delta))$ **by** *auto*
thus *?thesis*
unfolding *HSem-def evalHeap-reorder-head-append*[*OF assms*]
by *simp*
qed

lemma *env-restr-HSem*:
assumes $domA\ \Gamma \cap S = \{\}$
shows $(\{\!\!|\ \Gamma\ |\!\!\}\varrho)\ f|`\ S = \varrho\ f|`\ S$
proof (*rule env-restr-eqI*)
fix x
assume $x \in S$
hence $x \notin domA\ \Gamma$ **using** *assms* **by** *auto*
thus $(\{\!\!|\ \Gamma\ |\!\!\}\varrho)\ x = \varrho\ x$
by (*rule lookup-HSem-other*)
qed

lemma *env-restr-HSem-noop*:
assumes $domA\ \Gamma \cap edom\ \varrho = \{\}$
shows $(\{\!\!|\ \Gamma\ |\!\!\}\varrho)\ f|`\ edom\ \varrho = \varrho$
by (*simp add*: *env-restr-HSem*[*OF assms*] *env-restr-useless*)

lemma *HSem-Nil*[*simp*]: $\{\!\!|[]|\!\!\}\varrho = \varrho$
by (*subst HSem-eq*, *simp*)

5.3.3 Substitution

lemma *HSem-subst-exp*:
assumes $\bigwedge\varrho'.\ [\![\ e\ ]\!]_{\varrho'} = [\![\ e'\ ]\!]_{\varrho'}$
shows $\{\!\!|(x,\ e)\ \#\ \Gamma|\!\!\}\varrho = \{\!\!|(x,\ e')\ \#\ \Gamma|\!\!\}\varrho$
by (*rule parallel-HSem-ind*) (*auto simp add*: *assms evalHeap-subst-exp*)

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

by (*rule HSem-below*) (*auto simp add*: *lookup-HSem-heap below lookup-HSem-other*)

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 $\{\!|(x,\ e1)\ \#\ \Gamma|\!\}\varrho = \{\!|(x,\ e2)\ \#\ \Gamma|\!\}\varrho$
by (*metis assms HSem-subst-expr-below below-antisym*)

5.3.4 Re-calculating the semantics of the heap is idempotent

lemma *HSem-redo*:
shows $\{\!|\Gamma|\!\}(\{\!|\Gamma\ @\ \Delta|\!\}\varrho)\ f|`\ (edom\ \varrho \cup domA\ \Delta) = \{\!|\Gamma\ @\ \Delta|\!\}\varrho$ (**is** *?LHS = ?RHS*)
proof (*rule below-antisym*)
show *?LHS* ⊑ *?RHS*
by (*rule HSem-below*)
(*auto simp add*: *lookup-HSem-heap fun-belowD*[*OF env-restr-below-itself*])

show *?RHS* ⊑ *?LHS*
proof(*rule HSem-below*, *goal-cases*)
case (*1 x*)
thus *?case*
by (*cases* $x \notin edom\ \varrho$) (*auto simp add*: *lookup-HSem-other dest*:*lookup-not-edom*)
next
case *prems*: (*2 x*)
thus *?case*
proof(*cases* $x \in domA\ \Gamma$)
case *True*
thus *?thesis* **by** (*auto simp add*: *lookup-HSem-heap*)
next
case *False*
hence *delta*: $x \in domA\ \Delta$ **using** *prems* **by** *auto*
with *False* ‹*?LHS* ⊑ *?RHS*›
show *?thesis* **by** (*auto simp add*: *lookup-HSem-other lookup-HSem-heap monofun-cfun-arg*)
qed
qed
qed

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−
from *assms*
interpret *iterative*
where $e1 = \Lambda\ \varrho'.\ [\![\Gamma]\!]_{\varrho'}$
and $e2 = \Lambda\ \varrho'.\ [\![e]\!]_{\varrho'}$
and $S = domA\ \Gamma$
and $x = x$ **by** *unfold-locales*

have $\{\!|(x,e)\ \#\ \Gamma|\!\}\varrho = \mathit{fix} \cdot L$
by (*simp add*: *HSem-def′ override-on-upd ne*)
also have ... $= \mathit{fix} \cdot R$
by (*rule iterative-override-on*)
also have ... $= (\mu\ \varrho'.\ (\varrho\ {+}{+}_{domA\ \Gamma}\ (\{\!|\Gamma|\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'}))$
by (*simp add*: *HSem-def′*)
finally show *?thesis*.
qed

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−
from *assms*
interpret *iterative*
where $e1 = \Lambda\ \varrho'.\ [\![\Gamma]\!]_{\varrho'}$
and $e2 = \Lambda\ \varrho'.\ [\![e]\!]_{\varrho'}$
and $S = domA\ \Gamma$
and $x = x$ **by** *unfold-locales*

have $(\mu\ \varrho'.\ (\varrho\ {+}{+}_{domA\ \Gamma}\ (\{\!|\Gamma|\!\}\varrho'))(\ x := [\![e]\!]_{\varrho'})) = \mathit{fix} \cdot R$
by (*simp add*: *HSem-def′*)
also have ... $= \mathit{fix} \cdot L$
by (*rule iterative-override-on*[*symmetric*])
also have ... $= \mathit{fix} \cdot R'$
by (*rule iterative-override-on′*)
also have ... $= (\mu\ \varrho'.\ (\varrho\ {+}{+}_{domA\ \Gamma}\ (\{\!|\Gamma|\!\}\varrho'))(\ x := [\![e]\!]_{\{\!|\Gamma|\!\}\varrho'}))$
by (*simp add*: *HSem-def′*)
finally
show *?thesis*.
qed

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(*induction rule*: *parallel-HSem-ind*[*case-names adm base step*])
case *adm* **thus** *?case* **by** *simp*
next
case *base*
show *?case* **by** *simp*
next
case (*step y z*)
have $[\![\ \Gamma\ ]\!]_{y} = [\![\ \Gamma\ ]\!]_{z}$

proof(*rule evalHeap-cong'*)
fix *x*
assume $x \in \mathit{domA}\ \Gamma$
hence *fv* (*the* (*map-of* Γ *x*)) ⊆ *fv* Γ **by** (*rule map-of-fv-subset*)
with *assms*(*1*)
have *fv* (*the* (*map-of* Γ *x*)) ∩ *S* = *fv* (*the* (*map-of* Γ *x*)) **by** *auto*
with *step*
have *y f*|' *fv* (*the* (*map-of* Γ *x*)) = *z f*|' *fv* (*the* (*map-of* Γ *x*)) **by** *auto*
with ‹*x* ∈ *domA* Γ›
show $[\![\ \mathit{the}\ (\mathit{map\text{-}of}\ \Gamma\ x)\ ]\!]_y = [\![\ \mathit{the}\ (\mathit{map\text{-}of}\ \Gamma\ x)\ ]\!]_z$
by (*subst* (*1 2*) *assms*(*2*)[*OF* ‹*x* ∈ *domA* Γ›]) *simp*
qed
moreover
have *domA* Γ ⊆ *S* **using** *domA-fv-subset assms*(*1*) **by** *auto*
ultimately
show *?case* **by** (*simp add*: *env-restr-add env-restr-evalHeap-noop*)
qed
end

5.3.7 Freshness

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

lemma *ESem-fresh-cong*:
assumes ϱ *f*|' (*fv e*) = ϱ' *f*|' (*fv e*)
shows $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho'}$
by (*metis assms ESem-considers-fv*)

lemma *ESem-fresh-cong-subset*:
assumes *fv e* ⊆ *S*
assumes ϱ *f*|' *S* = ϱ' *f*|' *S*
shows $[\![\ e\ ]\!]_{\varrho} = [\![\ e\ ]\!]_{\varrho'}$
by (*rule ESem-fresh-cong*[*OF env-restr-eq-subset*[*OF assms*]])

lemma *ESem-fresh-cong-below*:
assumes ϱ *f*|' (*fv e*) ⊑ ϱ' *f*|' (*fv e*)
shows $[\![\ e\ ]\!]_{\varrho} \sqsubseteq [\![\ e\ ]\!]_{\varrho'}$
by (*metis assms ESem-considers-fv monofun-cfun-arg*)

lemma *ESem-fresh-cong-below-subset*:
assumes *fv e* ⊆ *S*
assumes ϱ *f*|' *S* ⊑ ϱ' *f*|' *S*
shows $[\![\ e\ ]\!]_{\varrho} \sqsubseteq [\![\ e\ ]\!]_{\varrho'}$
by (*rule ESem-fresh-cong-below*[*OF env-restr-below-subset*[*OF assms*]])

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

have *fv e* $\cap - S$ = *fv e* **using** *assms* **by** (*auto simp add*: *fresh-def fresh-star-def fv-supp*)
thus *?thesis* **by** (*subst* (*1 2*) *ESem-considers-fv*) *simp*
qed

lemma *ESem-ignores-fresh-restr'*:
assumes *atom* ‘ (*edom* ϱ − *S*) ♯* *e*
shows $⟦ e ⟧_{\varrho} = ⟦ e ⟧_{\varrho\ f|‘\ S}$
proof−
have $⟦ e ⟧_{\varrho} = ⟦ e ⟧_{\varrho\ f|‘\ (-\ (edom\ \varrho\ -\ S))}$
by (*rule ESem-ignores-fresh-restr*[*OF assms*])
also have ϱ *f*|‘ (− (*edom* ϱ − *S*)) = ϱ *f*|‘ *S*
by (*rule ext*) (*auto simp add*: *lookup-env-restr-eq dest*: *lookup-not-edom*)
finally show *?thesis***.**
qed

lemma *HSem-ignores-fresh-restr''*:
assumes *fv* Γ ⊆ *S*
shows (⦃Γ⦄ϱ) *f*|‘ *S* = ⦃Γ⦄ϱ *f*|‘ *S*
by (*rule HSem-ignores-fresh-restr'*[*OF assms*(*1*) *ESem-considers-fv*])

lemma *HSem-ignores-fresh-restr*:
assumes *atom* ‘ *S* ♯* Γ
shows (⦃Γ⦄ϱ) *f*|‘ (− *S*) = ⦃Γ⦄ϱ *f*|‘ (− *S*)
proof−
from *assms* **have** *fv* Γ ⊆ − *S* **by** (*auto simp add*: *fv-def fresh-star-def fresh-def*)
thus *?thesis* **by** (*rule HSem-ignores-fresh-restr''*)
qed

lemma *HSem-fresh-cong-below*:
assumes ϱ *f*|‘ ((*S* ∪ *fv* Γ) − *domA* Γ) ⊑ ϱ' *f*|‘ ((*S* ∪ *fv* Γ) − *domA* Γ)
shows (⦃Γ⦄ϱ) *f*|‘ *S* ⊑ (⦃Γ⦄ϱ') *f*|‘ *S*
proof−
from *assms*
have ⦃Γ⦄(ϱ *f*|‘ (*S* ∪ *fv* Γ)) ⊑ ⦃Γ⦄(ϱ' *f*|‘ (*S* ∪ *fv* Γ))
by (*auto intro*: *HSem-restr-cong-below simp add*: *Diff-eq inf-commute*)
hence (⦃Γ⦄ϱ) *f*|‘ (*S* ∪ *fv* Γ) ⊑ (⦃Γ⦄ϱ') *f*|‘ (*S* ∪ *fv* Γ)
by (*subst* (*1 2*) *HSem-ignores-fresh-restr''*) *simp-all*
thus *?thesis*
by (*rule env-restr-below-subset*[*OF Un-upper1*])
qed

lemma *HSem-fresh-cong*:
assumes ϱ *f*|‘ ((*S* ∪ *fv* Γ) − *domA* Γ) = ϱ' *f*|‘ ((*S* ∪ *fv* Γ) − *domA* Γ)
shows (⦃Γ⦄ϱ) *f*|‘ *S* = (⦃Γ⦄ϱ') *f*|‘ *S*
apply (*rule below-antisym*)
apply (*rule HSem-fresh-cong-below*[*OF eq-imp-below*[*OF assms*]])
apply (*rule HSem-fresh-cong-below*[*OF eq-imp-below*[*OF assms*[*symmetric*]]])
done

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

lemma *HSem-add-fresh'*:
  assumes *fresh*: *atom* x ♯ Γ
  assumes $x \notin$ *edom* ϱ
  assumes *step*: $\bigwedge e\ \varrho'.\ e \in$ *snd* ‘ *set* $\Gamma \Longrightarrow [\![\ e\ ]\!]_{\varrho'} = [\![\ e\ ]\!]_{env\text{-}delete\ x\ \varrho'}$
  shows *env-delete* x ($\{\!\!|(x,\ e)$ # $\Gamma|\!\!\}\varrho) = \{\!\!|\Gamma|\!\!\}\varrho$
proof (*rule parallel-HSem-ind*, *goal-cases*)
  case *1* **show** *?case* **by** *simp*
next
  case *2* **show** *?case* **by** *auto*
next
  case *prems*: (*3 y z*)
  have *env-delete* $x\ \varrho = \varrho$ **using** ‹$x \notin$ *edom* ϱ› **by** (*rule env-delete-noop*)
  moreover
  from *fresh* **have** $x \notin$ *domA* Γ **by** (*metis domA-not-fresh*)
  hence *env-delete* x ($[\![\ (x,\ e)$ # $\Gamma\ ]\!]_y) = [\![\ \Gamma\ ]\!]_y$
    by (*auto intro*: *env-delete-noop dest*: *subsetD*[*OF edom-evalHeap-subset*])
  moreover
  have ... $= [\![\ \Gamma\ ]\!]_z$
    apply (*rule evalHeap-cong*[*OF refl*])
    apply (*subst* (*1*) *step*, *assumption*)
    using *prems*(*1*) **apply** *auto*
    done
  ultimately
  show *?case* **using** ‹$x \notin$ *domA* Γ›
    by (*simp add*: *env-delete-add*)
qed

lemma *HSem-add-fresh*:
  assumes *atom* x ♯ Γ
  assumes $x \notin$ *edom* ϱ
  shows *env-delete* x ($\{\!\!|(x,\ e)$ # $\Gamma|\!\!\}\varrho) = \{\!\!|\Gamma|\!\!\}\varrho$
proof(*rule HSem-add-fresh'*[*OF assms*], *goal-cases*)
  case (*1 e* ϱ')
  assume $e \in$ *snd* ‘ *set* Γ
  hence *atom* x ♯ e **by** (*metis assms*(*1*) *fresh-heap-expr'*)
  hence $x \notin$ *fv* e **by** (*simp add*: *fv-def fresh-def*)
  thus *?case*
    by (*rule ESem-fresh-cong*[*OF env-restr-env-delete-other*[*symmetric*]])
qed

5.3.9 Mutual recursion with fresh variables

lemma *HSem-subset-below*:
  assumes *fresh*: *atom* ‘ *domA* Γ ♯∗ Δ
  shows $\{\!\!|\Delta|\!\!\}(\varrho\ f|$‘ $(-$ *domA* $\Gamma)) \sqsubseteq (\{\!\!|\Delta @ \Gamma|\!\!\}\varrho)\ \ f|$‘ $(-$ *domA* $\Gamma)$
proof(*rule HSem-below*)
  fix x
  assume [*simp*]: $x \in$ *domA* Δ

with *assms* **have** $*$: *atom* ‘ *domA* Γ $\sharp*$ *the* (*map-of* Δ *x*) **by** (*metis fresh-star-map-of*)
hence [*simp*]: $x \notin$ *domA* Γ **using** *fresh* ‹$x \in$ *domA* Δ› **by** (*metis fresh-star-def domA-not-fresh image-eqI*)
show $[\![$ *the* (*map-of* Δ *x*) $]\!]_{(\{\!\!\{\Delta\ @\ \Gamma\}\!\!\}\varrho)\ f|`\ (-\ domA\ \Gamma)} \sqsubseteq ((\{\!\!\{\Delta\ @\ \Gamma\}\!\!\}\varrho)\ f|`\ (-\ domA\ \Gamma))\ x$
by (*simp add*: *lookup-HSem-heap ESem-ignores-fresh-restr*[*OF* $*$, *symmetric*])
qed (*simp add*: *lookup-HSem-other lookup-env-restr-eq*)

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* Γ $\sharp*$ Δ
shows $\{\!\!\{\Gamma\}\!\!\}\{\!\!\{\Delta\}\!\!\}\varrho = \{\!\!\{\Gamma@\Delta\}\!\!\}\varrho$
proof(*rule below-antisym*)
have *map-of-eq*: *map-of* (Δ @ Γ) = *map-of* (Γ @ Δ)
proof
fix *x*
show *map-of* (Δ @ Γ) *x* = *map-of* (Γ @ Δ) *x*
proof (*cases* $x \in$ *domA* Γ)
case *True*
hence $x \notin$ *domA* Δ **by** (*metis fresh-distinct fresh IntI equals0D*)
thus *map-of* (Δ @ Γ) *x* = *map-of* (Γ @ Δ) *x*
by (*simp add*: *map-add-dom-app-simps dom-map-of-conv-domA*)
next
case *False*
thus *map-of* (Δ @ Γ) *x* = *map-of* (Γ @ Δ) *x*
by (*simp add*: *map-add-dom-app-simps dom-map-of-conv-domA*)
qed
qed

show $\{\!\!\{\Gamma\}\!\!\}\{\!\!\{\Delta\}\!\!\}\varrho \sqsubseteq \{\!\!\{\Gamma@\Delta\}\!\!\}\varrho$
proof(*rule HSem-below*)
fix *x*
assume [*simp*]:$x \notin$ *domA* Γ

have $(\{\!\!\{\Delta\}\!\!\}\varrho)\ x = ((\{\!\!\{\Delta\}\!\!\}\varrho)\ f|`\ (-\ domA\ \Gamma))\ x$ **by** *simp*
also have ... $= (\{\!\!\{\Delta\}\!\!\}(\varrho\ f|`\ (-\ domA\ \Gamma)))\ x$
by (*rule arg-cong*[*OF HSem-ignores-fresh-restr*[*OF fresh*]])
also have ... $\sqsubseteq ((\{\!\!\{\Delta@\Gamma\}\!\!\}\varrho)\ \ f|`\ (-\ domA\ \Gamma))\ x$
by (*rule fun-belowD*[*OF HSem-subset-below*[*OF fresh*]])
also have ... $= (\{\!\!\{\Delta@\Gamma\}\!\!\}\varrho)\ x$ **by** *simp*
also have ... $= (\{\!\!\{\Gamma\ @\ \Delta\}\!\!\}\varrho)\ x$ **by** (*rule arg-cong*[*OF HSem-reorder*[*OF map-of-eq*]])
finally
show $(\{\!\!\{\Delta\}\!\!\}\varrho)\ x \sqsubseteq (\{\!\!\{\Gamma\ @\ \Delta\}\!\!\}\varrho)\ x$.
qed (*auto simp add*: *lookup-HSem-heap lookup-env-restr-eq*)

have $*$: $\bigwedge x.\ x \in domA\ \Delta \Longrightarrow x \notin domA\ \Gamma$
using *fresh* **by** (*auto simp add*: *fresh-Pair fresh-star-def domA-not-fresh*)

have *foo*: *edom* ϱ $\cup$ *domA* Δ $\cup$ *domA* Γ $-$ (*edom* ϱ $\cup$ *domA* Δ $\cup$ *domA* Γ) $\cap$ $-$ *domA* Γ =

domA Γ **by** *auto*
have *foo2*:(*edom* ϱ ∪ *domA* Δ − (*edom* ϱ ∪ *domA* Δ) ∩ − *domA* Γ) ⊆ *domA* Γ **by** *auto*

{ fix *x*
assume *x* ∈ *domA* Δ
hence ∗: *atom* ‘ *domA* Γ ♯∗ *the* (*map-of* Δ *x*)
by (*rule fresh-star-map-of*[*OF* - *fresh*])

have ⟦ *the* (*map-of* Δ *x*) ⟧$_{⦃Γ⦄⦃Δ⦄ϱ}$ = ⟦ *the* (*map-of* Δ *x*) ⟧$_{(⦃Γ⦄⦃Δ⦄ϱ)\ f|`\ (-\ domA\ Γ)}$
by (*rule ESem-ignores-fresh-restr*[*OF* ∗])
also have (⦃Γ⦄⦃Δ⦄ϱ) *f*|‘ (− *domA* Γ) = (⦃Δ⦄ϱ) *f*|‘ (− *domA* Γ)
by (*simp add*: *env-restr-HSem*)
also have ⟦ *the* (*map-of* Δ *x*) ⟧$_{...}$ = ⟦ *the* (*map-of* Δ *x*) ⟧$_{⦃Δ⦄ϱ}$
by (*rule ESem-ignores-fresh-restr*[*symmetric*, *OF* ∗])
finally
have ⟦ *the* (*map-of* Δ *x*) ⟧$_{⦃Γ⦄⦃Δ⦄ϱ}$ = ⟦ *the* (*map-of* Δ *x*) ⟧$_{⦃Δ⦄ϱ}$.
}
thus ⦃Γ@Δ⦄ϱ ⊑ ⦃Γ⦄⦃Δ⦄ϱ
by −(*rule HSem-below*, *auto simp add*: *lookup-HSem-other lookup-HSem-heap* ∗)
qed
end

5.3.10 Parallel induction

lemma *parallel-HSem-ind-different-ESem*:
assumes *adm* (λϱ′. *P* (*fst* ϱ′) (*snd* ϱ′))
assumes *P* ⊥ ⊥
assumes ⋀*y z*. *P y z* ⟹ *P* (ϱ ++$_{domA\ h}$ *evalHeap h* (λ*e*. *ESem1 e* · *y*)) (ϱ*2* ++$_{domA\ h2}$ *evalHeap h2* (λ*e*. *ESem2 e* · *z*))
shows *P* (*has-ESem.HSem ESem1 h*·ϱ) (*has-ESem.HSem ESem2 h2*·ϱ*2*)
proof−
interpret *HSem1*: *has-ESem ESem1*.
interpret *HSem2*: *has-ESem ESem2*.

show *?thesis*
unfolding *HSem1.HSem-def′ HSem2.HSem-def′*
apply (*rule parallel-fix-ind*[*OF assms*(*1*)])
apply (*rule assms*(*2*))
apply *simp*
apply (*erule assms*(*3*))
done
qed

5.3.11 Congruence rule

lemma *HSem-cong*[*fundef-cong*]:
⟦ (⋀ *e*. *e* ∈ *snd* ‘ *set heap2* ⟹ *ESem1 e* = *ESem2 e*); *heap1* = *heap2* ⟧
⟹ *has-ESem.HSem ESem1 heap1* = *has-ESem.HSem ESem2 heap2*
unfolding *has-ESem.HSem-def*

by (*auto cong*:*evalHeap-cong*)

5.3.12 Equivariance of the heap semantics

lemma *HSem-eqvt*[*eqvt*]:
$\pi \cdot$ *has-ESem.HSem ESem* Γ = *has-ESem.HSem* ($\pi \cdot$ *ESem*) ($\pi \cdot \Gamma$)
proof−
show *?thesis*
unfolding *has-ESem.HSem-def*
apply (*subst permute-Lam*, *simp*)
apply (*subst eqvt-lambda*)
apply (*simp add*: *Cfun-app-eqvt permute-Lam*)
done
qed

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* → *'Value*) → (*'Value*::{*pcpo-pt*,*pure*})
fixes *Fn-project* :: *'Value* → (*'Value* → *'Value*)
fixes *B* :: *bool discr* → *'Value*
fixes *B-project* :: *'Value* → *'Value* → *'Value* → *'Value*
fixes *tick* :: *'Value* → *'Value*
begin

nominal-function
ESem :: *exp* ⇒ (*var* ⇒ *'Value*) → *'Value*
where
ESem (*Lam* [*x*]. *e*) = (Λ ϱ. *tick*·(*Fn*·(Λ *v*. *ESem e*·((ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*))(*x* := *v*)))))
| *ESem* (*App e x*) = (Λ ϱ. *tick*·(*Fn-project*·(*ESem e*·ϱ)·(ϱ *x*)))
| *ESem* (*Var x*) = (Λ ϱ. *tick*·(ϱ *x*))
| *ESem* (*Let as body*) = (Λ ϱ. *tick*·(*ESem body*·(*has-ESem.HSem ESem as*·(ϱ *f*|‘ *fv* (*Let as body*)))))
| *ESem* (*Bool b*) = (Λ ϱ. *tick*·(*B*·(*Discr b*)))
| *ESem* (*scrut ? e1* : *e2*) = (Λ ϱ. *tick*·((*B-project*·(*ESem scrut*·ϱ))·(*ESem e1*·ϱ)·(*ESem e2*·ϱ)))
proof *goal-cases*

The following proofs discharge technical obligations generated by the Nominal package.

case *1* **thus** *?case*

unfolding *eqvt-def ESem-graph-aux-def*
apply *rule*
apply (*perm-simp*)
apply (*simp add*: *Abs-cfun-eqvt*)
apply (*simp add*: *unpermute-def permute-pure*)
done
next
case (*3 P x*)
thus *?case* **by** (*metis* (*poly-guards-query*) *exp-strong-exhaust*)
next

case *prems*: (*4 x e x′ e′*)
from *prems*(*5*)
show *?case*
proof (*rule eqvt-lam-case*)
fix π :: *perm*
assume ∗: *supp* ($-\pi$) ♯∗ (*fv* (*Lam* [*x*]. *e*) :: *var set*)
{ fix ϱ *v*
have *ESem-sumC* (π • *e*)·((ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*))((π • *x*) := *v*)) = − π • *ESem-sumC* (π • *e*)·((ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*))((π • *x*) := *v*))
by (*simp add*: *permute-pure*)
also have ... = *ESem-sumC e*·((− π • (ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*)))(*x* := *v*)) **by** (*simp add*: *pemute-minus-self eqvt-at-apply*[*OF prems*(*1*)])
also have − π • (ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*)) = (ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*)) **by** (*rule env-restr-perm′*[*OF* ∗]) *auto*
finally have *ESem-sumC* (π • *e*)·((ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*))((π • *x*) := *v*)) = *ESem-sumC e*·((ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*))(*x* := *v*)).
}
thus (Λ ϱ. *tick*·(*Fn*·(Λ *v*. *ESem-sumC* (π • *e*)·((ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*))(π • *x* := *v*))))) = (Λ ϱ. *tick*·(*Fn*·(Λ *v*. *ESem-sumC e*·((ϱ *f*|‘ *fv* (*Lam* [*x*]. *e*))(*x* := *v*))))) **by** *simp*
qed
next

case *prems*: (*19 as body as′ body′*)
from *prems*(*9*)
show *?case*
proof (*rule eqvt-let-case*)
fix π :: *perm*
assume ∗: *supp* ($-\pi$) ♯∗ (*fv* (*Terms.Let as body*) :: *var set*)

{ fix ϱ
have *ESem-sumC* (π • *body*)·(*has-ESem.HSem ESem-sumC* (π • *as*)·(ϱ *f*|‘ *fv* (*Terms.Let as body*)))
= − π • *ESem-sumC* (π • *body*)·(*has-ESem.HSem ESem-sumC* (π • *as*)·(ϱ *f*|‘ *fv* (*Terms.Let as body*)))
by (*rule permute-pure*[*symmetric*])
also have ... = (− π • *ESem-sumC*) *body*·(*has-ESem.HSem* (− π • *ESem-sumC*) *as*·(− π • ϱ *f*|‘ *fv* (*Terms.Let as body*)))
by (*simp add*: *pemute-minus-self*)

also have ($-\ \pi \cdot$ *ESem-sumC*) *body* = *ESem-sumC body*
by (*rule eqvt-at-apply*[*OF* ‹*eqvt-at ESem-sumC body*›])
also have *has-ESem.HSem* ($-\ \pi \cdot$ *ESem-sumC*) *as* = *has-ESem.HSem ESem-sumC as*
by (*rule HSem-cong*[*OF eqvt-at-apply*[*OF prems*(*2*)] *refl*])
also have $-\ \pi \cdot \varrho\ f|`\ fv$ (*Let as body*) = $\varrho\ f|`\ fv$ (*Let as body*)
by (*rule env-restr-perm'*[*OF* ∗], *simp*)
finally have *ESem-sumC* ($\pi \cdot$ *body*)·(*has-ESem.HSem ESem-sumC* ($\pi \cdot$ *as*)·($\varrho\ f|`\ fv$ (*Let as body*))) = *ESem-sumC body*·(*has-ESem.HSem ESem-sumC as*·($\varrho\ f|`\ fv$ (*Let as body*))).
}
thus ($\Lambda\ \varrho$. *tick*·(*ESem-sumC* ($\pi \cdot$ *body*)·(*has-ESem.HSem ESem-sumC* ($\pi \cdot$ *as*)·($\varrho\ f|`\ fv$ (*Let as body*))))) =
($\Lambda\ \varrho$. *tick*·(*ESem-sumC body*·(*has-ESem.HSem ESem-sumC as*·($\varrho\ f|`\ fv$ (*Let as body*)))))
by (*simp only*:)
qed
qed *auto*

nominal-termination (**in** *semantic-domain*) (*no-eqvt*) **by** *lexicographic-order*

sublocale *has-ESem ESem*.

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'*: $⟦\ e\ ⟧_{\varrho} = ⟦\ e\ ⟧_{\varrho\ f|`\ (fv\ e)}$
proof (*induct e arbitrary*: ϱ *rule*:*exp-induct*)
case *Var*
show *?case* **by** *simp*
next
have [*simp*]: $\bigwedge$ *S x*. *S* ∩ *insert x S* = *S* **by** *auto*
case *App*
show *?case*
by (*simp*, *subst* (*1 2*) *App*, *simp*)
next

case (*Lam x e*)
show *?case* **by** *simp*
next
case (*IfThenElse scrut* e_1 e_2)
have [*simp*]: (*fv scrut* ∩ (*fv scrut* ∪ *fv* e_1 ∪ *fv* e_2)) = *fv scrut* **by** *auto*
have [*simp*]: (*fv* e_1 ∩ (*fv scrut* ∪ *fv* e_1 ∪ *fv* e_2)) = *fv* e_1 **by** *auto*
have [*simp*]: (*fv* e_2 ∩ (*fv scrut* ∪ *fv* e_1 ∪ *fv* e_2)) = *fv* e_2 **by** *auto*
show *?case*
apply *simp*
apply (*subst* (*1 2*) *IfThenElse*(*1*))
apply (*subst* (*1 2*) *IfThenElse*(*2*))
apply (*subst* (*1 2*) *IfThenElse*(*3*))
apply (*simp*)
done
next
case (*Let as e*)

have $⟦e⟧_{\{\!|as|\!\}\varrho} = ⟦e⟧_{(\{\!|as|\!\}\varrho)\ f|`\ (fv\ as \cup fv\ e)}$
apply (*subst* (*1 2*) *Let*(*2*))
apply *simp*
done
also
have *fv as* ⊆ *fv as* ∪ *fv e* **by** (*rule inf-sup-ord*(*3*))
hence ({|*as*|}ϱ) *f*|‘ (*fv as* ∪ *fv e*) = {|*as*|}(ϱ *f*|‘ (*fv as* ∪ *fv e*))
by (*rule HSem-ignores-fresh-restr'*[*OF* - *Let*(*1*)])
also
have {|*as*|}(ϱ *f*|‘ (*fv as* ∪ *fv e*)) = {|*as*|}ϱ *f*|‘ (*fv as* ∪ *fv e* − *domA as*)
by (*rule HSem-restr-cong*) (*auto simp add*: *lookup-env-restr-eq*)
finally
show *?case* **by** *simp*
qed *auto*

sublocale *has-ignore-fresh-ESem ESem*
by *standard* (*rule fv-supp-exp*, *rule ESem-considers-fv'*)

5.5.2 Nicer equations for ESem, without freshness requirements

lemma *ESem-Lam*[*simp*]: $⟦\ Lam\ [x].\ e\ ⟧_\varrho = tick \cdot (Fn \cdot (\Lambda\ v.\ ⟦\ e\ ⟧_{\varrho(x := v)}))$
proof−
have ∗: $\bigwedge v.\ ((\varrho\ f|`\ (fv\ e - \{x\}))(x := v))\ f|`\ fv\ e = (\varrho(x := v))\ f|`\ fv\ e$
by (*rule ext*) (*auto simp add*: *lookup-env-restr-eq*)

have $⟦\ Lam\ [x].\ e\ ⟧_\varrho = ⟦\ Lam\ [x].\ e\ ⟧_{env\text{-}delete\ x\ \varrho}$
by (*rule ESem-fresh-cong*) *simp*
also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ ⟦\ e\ ⟧_{(\varrho\ f|`\ (fv\ e - \{x\}))(x := v)}))$
by *simp*
also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ ⟦\ e\ ⟧_{((\varrho\ f|`\ (fv\ e - \{x\}))(x := v))\ f|`\ fv\ e}))$
by (*subst ESem-considers-fv*, *rule*)
also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ ⟦\ e\ ⟧_{\varrho(x := v)\ f|`\ fv\ e}))$

unfolding $*$**..**
also have $\ldots = tick \cdot (Fn \cdot (\Lambda\ v.\ [\![\ e\ ]\!]_{\varrho(x\ :=\ v)}))$
unfolding *ESem-considers-fv*[*symmetric*]**..**
finally show *?thesis***.**
qed
declare *ESem.simps*(*1*)[*simp del*]

lemma *ESem-Let*[*simp*]: $[\![Let\ as\ body]\!]_\varrho = tick \cdot ([\![body]\!]_{\{\!|as|\!\}\varrho})$
proof−
have $[\![\ Let\ as\ body\ ]\!]_\varrho = tick \cdot ([\![body]\!]_{\{\!|as|\!\}(\varrho\ f|`\ fv\ (Let\ as\ body))})$
by *simp*
also have $\{\!|as|\!\}(\varrho\ f|`\ fv(Let\ as\ body)) = \{\!|as|\!\}(\varrho\ f|`\ (fv\ as \cup fv\ body))$
by (*rule HSem-restr-cong*) (*auto simp add*: *lookup-env-restr-eq*)
also have $\ldots = (\{\!|as|\!\}\varrho)\ f|`\ (fv\ as \cup fv\ body)$
by (*rule HSem-ignores-fresh-restr'*[*symmetric*, *OF* - *ESem-considers-fv*]) *simp*
also have $[\![body]\!]_{\ldots} = [\![body]\!]_{\{\!|as|\!\}\varrho}$
by (*rule ESem-fresh-cong*) (*auto simp add*: *lookup-env-restr-eq*)
finally show *?thesis***.**
qed
declare *ESem.simps*(*4*)[*simp del*]

5.5.3 Denotation of Substitution

lemma *ESem-subst-same*: $\varrho\ x = \varrho\ y \Longrightarrow\ [\![\ e\ ]\!]_\varrho = [\![\ e[x::=\ y]\ ]\!]_\varrho$
and
$\varrho\ x = \varrho\ y\ \Longrightarrow\ ([\![\ as\ ]\!]_\varrho) = [\![\ as[x::h=y]\ ]\!]_\varrho$
proof (*nominal-induct e* **and** *as avoiding*: *x y arbitrary*: ϱ **and** ϱ *rule*:*exp-heap-strong-induct*)
case *Var* **thus** *?case* **by** *auto*
next
case *App*
from *App*(*1*)[*OF App*(*2*)] *App*(*2*)
show *?case* **by** *auto*
next
case (*Let as exp x y* ϱ)
from ‹*atom* ‘ *domA as* ♯∗ *x*› ‹*atom* ‘ *domA as* ♯∗ *y*›
have $x \notin domA\ as$ $y \notin domA\ as$
by (*metis fresh-star-at-base imageI*)+
hence [*simp*]:*domA* (*as*[*x*::*h*=*y*]) = *domA as*
by (*metis bn-subst*)

from ‹$\varrho\ x = \varrho\ y$›
have $(\{\!|as|\!\}\varrho)\ x = (\{\!|as|\!\}\varrho)\ y$
using ‹$x \notin domA\ as$› ‹$y \notin domA\ as$›
by (*simp add*: *lookup-HSem-other*)
hence $[\![exp]\!]_{\{\!|as|\!\}\varrho} = [\![exp[x::=y]]\!]_{\{\!|as|\!\}\varrho}$
by (*rule Let*)
moreover
from ‹$\varrho\ x = \varrho\ y$›
have $\{\!|as|\!\}\varrho = \{\!|as[x::h=y]|\!\}\varrho$ **and** $(\{\!|as|\!\}\varrho)\ x = (\{\!|as[x::h=y]|\!\}\varrho)\ y$

apply (*induction rule*: *parallel-HSem-ind*)
apply (*intro adm-lemmas cont2cont cont2cont-fun*)
apply *simp*
apply *simp*
apply *simp*
apply (*erule arg-cong*[*OF Let*(*3*)])
using ‹$x \notin$ *domA as*› ‹$y \notin$ *domA as*›
apply *simp*
done
ultimately
show *?case* **using** *Let*(*1*,*2*,*3*) **by** (*simp add*: *fresh-star-Pair*)
next
case (*Lam var exp x y* ϱ)
from ‹$\varrho\ x = \varrho\ y$›
have $\bigwedge v.\ (\varrho(var := v))\ x = (\varrho(var := v))\ y$
using *Lam*(*1*,*2*) **by** (*simp add*: *fresh-at-base*)
hence $\bigwedge\ v.\ [\![exp]\!]_{\varrho(var\ :=\ v)} = [\![exp[x::=y]]\!]_{\varrho(var\ :=\ v)}$
by (*rule Lam*)
thus *?case* **using** *Lam*(*1*,*2*) **by** *simp*
next
case *IfThenElse*
from *IfThenElse*(*1*)[*OF IfThenElse*(*4*)] *IfThenElse*(*2*)[*OF IfThenElse*(*4*)] *IfThenElse*(*3*)[*OF IfThenElse*(*4*)]
show *?case*
by *simp*
next
case *Nil* **thus** *?case* **by** *auto*
next
case *Cons*
from *Cons*(*1*,*2*)[*OF Cons*(*3*)] *Cons*(*3*)
show *?case* **by** *auto*
qed *auto*

lemma *ESem-subst*:
shows $[\![\ e\ ]\!]_{\sigma(x\ :=\ \sigma\ y)} = [\![\ e[x::=\ y]\ ]\!]_{\sigma}$
proof(*cases* $x = y$)
case *False*
hence [*simp*]: $x \notin fv\ e[x::=y]$ **by** (*auto simp add*: *fv-def supp-subst supp-at-base dest*: *subsetD*[*OF supp-subst*])

have $[\![\ e\ ]\!]_{\sigma(x\ :=\ \sigma\ y)} = [\![\ e[x::=\ y]\ ]\!]_{\sigma(x\ :=\ \sigma\ y)}$
by (*rule ESem-subst-same*) *simp*
also have $\ldots = [\![\ e[x::=\ y]\ ]\!]_{\sigma}$
by (*rule ESem-fresh-cong*) *simp*
finally
show *?thesis*.
next
case *True*
thus *?thesis* **by** *simp*

qed

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* $(\Lambda\ x.\ x)$**.**

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*:
$[\![\ Lam\ [x].\ e\ ]\!]_{\varrho} = \ Fn\cdot(\Lambda\ v.\ [\![\ e\ ]\!]_{(\varrho\ f|`\ (fv\ (Lam\ [x].e)))(x := v)})$
$[\![\ App\ e\ x\ ]\!]_{\varrho} \ = \ [\![\ e\ ]\!]_{\varrho} \downarrow Fn\ \varrho\ x$
$[\![\ Var\ x\ ]\!]_{\varrho} \ = \ \varrho\ \ x$
$[\![\ Bool\ b\ ]\!]_{\varrho} \ = \ B\cdot(Discr\ b)$
$[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho} = B\text{-}project\cdot([\![\ scrut\ ]\!]_{\varrho})\cdot([\![\ e_1\ ]\!]_{\varrho})\cdot([\![\ e_2\ ]\!]_{\varrho})$
$[\![\ Let\ \Gamma\ body\ ]\!]_{\varrho} = [\![body]\!]_{\{\!|\Gamma|\!\}(\varrho\ f|`\ fv\ (Let\ \Gamma\ body))}$
by (*simp-all del*: *ESem-Lam ESem-Let add*: *ESem.simps*(*1*,*4*))

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

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*: $x \sqsubseteq C \cdot x$
  by (*induct x*) *auto*

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

lemma *C-Cinf*[*simp*]: $C \cdot C^\infty = C^\infty$ **unfolding** *Cinf-def* **by** (*rule fix-eq*[*symmetric*])

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

lemma *C-below-C*[*simp*]: $(C^i \sqsubseteq C^j) \longleftrightarrow i \leq j$
  apply (*induction i arbitrary*: *j*)
  apply *simp*
  apply (*case-tac j*, *auto*)
  done

lemma *below-Cinf*[*simp*]: $r \sqsubseteq C^\infty$
  apply (*induct r*)
  apply *simp-all*[*2*]
  apply (*metis* (*full-types*) *C-Cinf monofun-cfun-arg*)
  done

lemma *C-eq-Cinf*[*simp*]: $C^i \neq C^\infty$
  by (*metis C-below-C Suc-n-not-le-n below-Cinf*)

lemma *Cinf-eq-C*[*simp*]: $C^\infty = C \cdot r \longleftrightarrow C^\infty = r$
  by (*metis C.injects C-Cinf*)

lemma *C-eq-C*[*simp*]: $(C^i = C^j) \longleftrightarrow i = j$
  by (*metis C-below-C le-antisym le-refl*)

lemma *case-of-C-below*: $(\mathit{case}\ r\ \mathit{of}\ C \cdot y \Rightarrow x) \sqsubseteq x$
  by (*cases r*) *auto*

lemma *C-case-below*: $C\text{-}\mathit{case} \cdot f \sqsubseteq f$

by (*metis cfun-belowI C.case-rews(2) below-C monofun-cfun-arg*)

lemma *C-case-bot*[*simp*]: *C-case* $\cdot \perp = \perp$
apply (*subst eq-bottom-iff*)
apply (*rule C-case-below*)
done

lemma *C-case-cong*:
assumes $\bigwedge r'.\ r = C{\cdot}r' \Longrightarrow f{\cdot}r' = g{\cdot}r'$
shows $C\text{-}case{\cdot}f{\cdot}r = C\text{-}case{\cdot}g{\cdot}r$
using *assms* **by** (*cases r*) *auto*

lemma *C-cases*:
obtains *n* **where** $r = C^{n} \mid r = C^{\infty}$
proof−
{ fix *m*
have $\exists\ n.\ C\text{-}take\ m \cdot r = C^{n}$
proof (*rule C.finite-induct*)
have $\perp = C^{0}$ **by** *simp*
thus $\exists n.\ \perp = C^{n}$**..**
next
fix *r*
show $\exists n.\ r = C^{n} \Longrightarrow \exists n.\ C{\cdot}r = C^{n}$
by (*auto simp del*: *iterate-Suc simp add*: *iterate-Suc*[*symmetric*])
qed
}
then obtain *f* **where** *take*: $\bigwedge\ m.\ C\text{-}take\ m \cdot r = C^{f\ m}$ **by** *metis*
have *chain* $(\lambda\ m.\ C^{f\ m})$ **using** *ch2ch-Rep-cfunL*[*OF C.chain-take*, **where** *x*=*r*, *unfolded take*]**.**
hence *mono f* **by** (*auto simp add*: *mono-iff-le-Suc chain-def elim*!:*chainE*)
have *r*: $r = (\bigsqcup\ m.\ C^{f\ m})$ **by** (*metis* (*lifting*) *take C.reach lub-eq*)
from ‹*mono f*›
show *thesis*
proof(*rule nat-mono-characterization*)
fix *n*
assume *n*: $\bigwedge\ m.\ n \leq m ==> f\ n = f\ m$
have *max-in-chain n* $(\lambda\ m.\ C^{f\ m})$
apply (*rule max-in-chainI*)
apply *simp*
apply (*erule n*)
done
hence $(\bigsqcup\ m.\ C^{f\ m}) = C^{f\ n}$ **unfolding** *maxinch-is-thelub*[*OF* ‹*chain* -›]**.**
thus *?thesis* **using** *that* **unfolding** *r* **by** *blast*
next
assume $\bigwedge m.\ \exists n.\ m \leq f\ n$
hence $\bigwedge\ n.\ C^{n} \sqsubseteq r$ **unfolding** *r* **by** (*fastforce intro*: *below-lub*[*OF* ‹*chain* -›])
hence $(\bigsqcup\ n.\ C^{n}) \sqsubseteq r$
by (*rule lub-below*[*OF chain-iterate*])

hence $C^\infty \sqsubseteq r$ **unfolding** *Cinf-def fix-def2*.
hence $C^\infty = r$ **using** *below-Cinf* **by** (*metis below-antisym*)
thus *thesis* **using** *that* **by** *blast*
qed
qed

lemma *C-case-Cinf*[*simp*]: *C-case* $\cdot f \cdot C^\infty = f \cdot C^\infty$
unfolding *Cinf-def*
by (*subst fix-eq*) *simp*

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-meet*$\cdot$(*C*$\cdot$*a*)$\cdot$(*C*$\cdot$*b*) = *C*$\cdot$(*C-meet*$\cdot$*a*$\cdot$*b*)

lemma[*simp*]: *C-meet*$\cdot\bot\cdot y = \bot$ *C-meet*$\cdot x \cdot \bot = \bot$ **by** (*fixrec-simp*, *cases x*, *fixrec-simp*+)

instance
apply *standard*
proof(*intro exI conjI strip*)
fix *x y*
{
fix *t*
have ($t \sqsubseteq$ *C-meet*$\cdot x \cdot y$) = ($t \sqsubseteq x \land t \sqsubseteq y$)
proof (*induct t rule*:*C.take-induct*)
fix *n*
show (*C-take n*$\cdot t \sqsubseteq$ *C-meet*$\cdot x\cdot y$) = (*C-take n*$\cdot t \sqsubseteq x \land$ *C-take n*$\cdot t \sqsubseteq y$)
proof (*induct n arbitrary*: *t x y rule*:*nat-induct*)
case *0* **thus** *?case* **by** *auto*
next
case (*Suc n t x y*)
with *C.nchotomy*[*of t*] *C.nchotomy*[*of x*] *C.nchotomy*[*of y*]
show *?case* **by** *fastforce*
qed
qed *auto*
} **note** ∗ = *this*
show *C-meet*$\cdot x\cdot y \sqsubseteq x$ **using** ∗ **by** *auto*
show *C-meet*$\cdot x\cdot y \sqsubseteq y$ **using** ∗ **by** *auto*
fix *z*
assume $z \sqsubseteq x$ **and** $z \sqsubseteq y$
thus $z \sqsubseteq$ *C-meet*$\cdot x\cdot y$ **using** ∗ **by** *auto*

qed
end

lemma *C-meet-is-meet*: $(z \sqsubseteq C\text{-}meet \cdot x \cdot y) = (z \sqsubseteq x \wedge z \sqsubseteq y)$
proof (*induct z rule*:*C.take-induct*)
  fix *n*
  show $(C\text{-}take\ n \cdot z \sqsubseteq C\text{-}meet \cdot x \cdot y) = (C\text{-}take\ n \cdot z \sqsubseteq x \wedge C\text{-}take\ n \cdot z \sqsubseteq y)$
  proof (*induct n arbitrary*: *z x y rule*:*nat-induct*)
    case *0* **thus** *?case* **by** *auto*
    next
    case (*Suc n z x y*) **thus** *?case*
      apply −
      apply (*cases z*, *simp*)
      apply (*cases x*, *simp*)
      apply (*cases y*, *simp*)
      apply (*fastforce simp add*: *cfun-below-iff*)
      done
  qed
qed *auto*

instance *C* :: *cont-binary-meet*
proof (*standard*, *goal-cases*)
  have [*simp*]: $\bigwedge x\ y.\ x \sqcap y = C\text{-}meet \cdot x \cdot y$
    using *C-meet-is-meet*
    by (*blast intro*: *is-meetI*)
  case *1* **thus** *?case*
    by (*simp add*: *ch2ch-Rep-cfunR contlub-cfun-arg contlub-cfun-fun*)
qed

lemma [*simp*]: $C \cdot r \sqcap r = r$
  by (*auto intro*: *is-meetI simp add*: *below-C*)

lemma [*simp*]: $r \sqcap C \cdot r = r$
  by (*auto intro*: *is-meetI simp add*: *below-C*)

lemma [*simp*]: $C \cdot r \sqcap C \cdot r' = C \cdot (r \sqcap r')$
  apply (*rule is-meetI*)
  apply (*metis below-refl meet-below1 monofun-cfun-arg*)
  apply (*metis below-refl meet-below2 monofun-cfun-arg*)
  apply (*case-tac a*)
  apply *auto*
  by (*metis meet-above-iff*)

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−
{
assume $\forall\, n.\ f{\cdot}(C^n) = \bot$
hence $f{\cdot}C^\infty \sqsubseteq \bot$
by (*auto intro*: *lub-below*[*OF ch2ch-Rep-cfunR*[*OF chain-iterate*]]
simp add: *Cinf-def fix-def2 contlub-cfun-arg*[*OF chain-iterate*])
with *assms* **have** *False* **by** *simp*
}
thus *?thesis* **using** *that* **by** *blast*
qed

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 $f{\cdot}r' \neq \bot$
using *assms*(*1*) *monofun-cfun-arg*[*OF assms*(*2*), **where** $f = f$]
by *auto*

lemma *infinite-resources-suffice*:
shows $f{\cdot}r \neq \bot \Longrightarrow f{\cdot}C^\infty \neq \bot$
by (*erule more-resources-suffice*[*OF - below-Cinf*])

lemma *demand-suffices*:
assumes $f{\cdot}C^\infty \neq \bot$
shows $f{\cdot}(demand\ f) \neq \bot$
apply (*simp add*: *assms demand-def*)
apply (*rule finite-resources-suffice*[*OF assms*])
apply (*rule LeastI*)
apply *assumption*
done

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

proof(*intro iffI*)
  assume $f{\cdot}r \neq \bot$
  thus *demand f* $\neq C^{\infty} \wedge$ *demand f* $\sqsubseteq r$
    apply (*cases r rule*: *C-cases*)
    apply (*auto intro*: *Least-le simp add*: *demand-def dest*: *infinite-resources-suffice*)
    done
next
  assume $*$: *demand f* $\neq C^{\infty}$ $\wedge$ *demand f* $\sqsubseteq r$
  then have $f{\cdot}C^{\infty} \neq \bot$ **by** (*auto intro*: *Least-le simp add*: *demand-def dest*: *infinite-resources-suffice*)
  hence $f{\cdot}$(*demand f*) $\neq \bot$ **by** (*rule demand-suffices*)
  moreover from $*$ **have** *demand f* $\sqsubseteq r$**..**
  ultimately
  show $f{\cdot}r \neq \bot$ **by** (*rule more-resources-suffice*)
qed

lemma *infinity-bot-demand*:
  $f{\cdot}C^{\infty} = \bot \longleftrightarrow$ *demand f* $= C^{\infty}$
  by (*metis below-Cinf not-bot-demand*)

lemma *demand-suffices'*:
  assumes *demand f* $= C^{n}$
  shows $f{\cdot}$(*demand f*) $\neq \bot$
  by (*metis C-eq-Cinf assms demand-suffices infinity-bot-demand*)

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−
  from *assms*
  have *demand f* $= C^{(LEAST\ n.\ f\cdot C^{n} \neq \bot)}$ **by** (*auto simp add*: *demand-def*)
  also
  then obtain n **where** $f{\cdot}C^{n} \neq \bot$ **by** (*metis demand-suffices'*)
  hence $(LEAST\ n.\ f\cdot C^{n} \neq \bot) = Suc\ (LEAST\ n.\ f\cdot C^{Suc\ n} \neq \bot)$
    apply (*rule Least-Suc*) **by** *simp*
  finally show *?thesis***.**
qed

lemma *demand-C-case*[*simp*]: *demand* (*C-case*$\cdot f$) $= C \cdot$ (*demand f*)
proof(*cases demand* (*C-case*$\cdot f$) $= C^{\infty}$)
  case *True*
  then have *C-case*$\cdot f\cdot C^{\infty} = \bot$
    by (*metis infinity-bot-demand*)
  with *True*
  show *?thesis* **apply** *auto* **by** (*metis infinity-bot-demand*)
next
  case *False*
  hence *demand* (*C-case*$\cdot f$) $= C^{Suc\ (LEAST\ n.\ (C\text{-}case\cdot f)\cdot C^{Suc\ n} \neq \bot)}$
    by (*rule demand-Suc-Least*[*OF C.case-rews*(*1*)])

also have ... $= C \cdot C^{LEAST\ n.\ f \cdot C^n \neq \bot}$ **by** *simp*
also have ... $= C \cdot (demand\ \ f)$
using *False* **unfolding** *demand-def* **by** *auto*
finally show *?thesis*.
qed

lemma *demand-contravariant*:
assumes $f \sqsubseteq g$
shows $demand\ g \sqsubseteq demand\ f$
proof(*cases demand f rule*:*C-cases*)
fix n
assume $demand\ f = C^n$
hence $f \cdot (demand\ f) \neq \bot$ **by** (*metis demand-suffices′*)
also note *monofun-cfun-fun*[*OF assms*]
finally have $g \cdot (demand\ f) \neq \bot$ **by** *this* (*intro cont2cont*)
thus $demand\ g \sqsubseteq demand\ f$ **unfolding** *not-bot-demand* **by** *auto*
qed *auto*

6.3.2 Restricting functions with domain C

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

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

lemma [*simp*]: $\bot|_r = \bot$ **by** *fixrec-simp*
lemma [*simp*]: $f \cdot \bot = \bot \Longrightarrow f|_\bot = \bot$ **by** *fixrec-simp*

lemma *C-restr-C-restr*[*simp*]: $(v|_{r'})|_r = v|_{(r' \sqcap r)}$
by (*rule cfun-eqI*) *simp*

lemma *C-restr-eqD*:
assumes $f|_r = g|_r$
assumes $r' \sqsubseteq r$
shows $f \cdot r' = g \cdot r'$
by (*metis C-restr.simps assms below-refl is-meetI*)

lemma *C-restr-eq-lower*:
assumes $f|_r = g|_r$
assumes $r' \sqsubseteq r$
shows $f|_{r'} = g|_{r'}$
by (*metis C-restr-C-restr assms below-refl is-meetI*)

lemma *C-restr-below*[*intro*, *simp*]:
$x|_r \sqsubseteq x$
apply (*rule cfun-belowI*)
apply *simp*
by (*metis below-refl meet-below2 monofun-cfun-arg*)

lemma *C-restr-below-cong*:
$(\bigwedge r'.\ r' \sqsubseteq r \Longrightarrow f \cdot r' \sqsubseteq g \cdot r') \Longrightarrow f|_r \sqsubseteq g|_r$
apply (*rule cfun-belowI*)
apply *simp*
by (*metis below-refl meet-below1*)

lemma *C-restr-cong*:
$(\bigwedge r'.\ r' \sqsubseteq r \Longrightarrow f \cdot r' = g \cdot r') \Longrightarrow f|_r = g|_r$
apply (*intro below-antisym C-restr-below-cong*)
by (*metis below-refl*)+

lemma *C-restr-C-cong*:
$(\bigwedge r'.\ r' \sqsubseteq r \Longrightarrow f \cdot (C \cdot r') = g \cdot (C \cdot r')) \Longrightarrow f \cdot \bot = g \cdot \bot \Longrightarrow f|_{C \cdot r} = g|_{C \cdot r}$
apply (*rule C-restr-cong*)
by (*case-tac* r', *auto*)

lemma *C-restr-C-case*[*simp*]:
$(C\text{-}case \cdot f)|_{C \cdot r} = C\text{-}case \cdot (f|_r)$
apply (*rule cfun-eqI*)
apply *simp*
apply (*case-tac x*)
apply *simp*
apply *simp*
done

lemma *C-restr-bot-demand*:
assumes $C \cdot r \sqsubseteq$ *demand f*
shows $f|_r = \bot$
proof(*rule cfun-eqI*)
fix r'
have $f \cdot (r \sqcap r') = \bot$
proof(*rule classical*)
have $r \sqsubseteq C \cdot r$ **by** (*rule below-C*)
also
note *assms*
also
assume $*$: $f \cdot (r \sqcap r') \neq \bot$
hence *demand f* $\sqsubseteq (r \sqcap r')$ **unfolding** *not-bot-demand* **by** *auto*
hence *demand f* $\sqsubseteq r$ **by** (*metis below-refl meet-below1 below-trans*)
finally (*below-antisym*) **have** $r =$ *demand f* **by** *this* (*intro cont2cont*)
with *assms*
have *demand f* $= C^\infty$ **by** (*cases demand f rule*: *C-cases*) (*auto simp add*: *iterate-Suc*[*symmetric*] *simp del*: *iterate-Suc*)
thus $f \cdot (r \sqcap r') = \bot$ **by** (*metis not-bot-demand*)
qed
thus $(f|_r) \cdot r' = \bot \cdot r'$ **by** *simp*
qed

6.3.3 Restricting maps of C-ranged functions

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

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

lemma *env-C-restr-upd*[*simp*]: $(\varrho(x := v))|^{\circ}{}_{r} = (\varrho|^{\circ}{}_{r})(x := v|_{r})$
  unfolding *env-C-restr-def* **by** *simp*

lemma *env-C-restr-lookup*[*simp*]: $(\varrho|^{\circ}{}_{r})\ v = \varrho\ v|_{r}$
  unfolding *env-C-restr-def* **by** *simp*

lemma *env-C-restr-bot*[*simp*]: $\bot|^{\circ}{}_{r} = \bot$
  unfolding *env-C-restr-def* **by** *auto*

lemma *env-C-restr-restr-below*[*intro*]: $\varrho|^{\circ}{}_{r} \sqsubseteq \varrho$
  by (*auto intro*: *fun-belowI*)

lemma *env-C-restr-env-C-restr*[*simp*]: $(v|^{\circ}{}_{r'})|^{\circ}{}_{r} = v|^{\circ}{}_{(r' \sqcap r)}$
  unfolding *env-C-restr-def* **by** *auto*

lemma *env-C-restr-cong*:
  $(\bigwedge x\ r'.\ r' \sqsubseteq r \Longrightarrow f\ x \cdot r' = g\ x \cdot r') \Longrightarrow f|^{\circ}{}_{r} = g|^{\circ}{}_{r}$
  unfolding *env-C-restr-def*
  by (*rule ext*) (*auto intro*: *C-restr-cong*)

end

6.4 CValue

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*]:

$\bot \downarrow CFn\ v = \bot$
$CB \cdot b \downarrow CFn\ v = \bot$
by (*fixrec-simp*)+

lemma *CB-below*[*simp*]: $CB \cdot b \sqsubseteq v \longleftrightarrow v = CB \cdot b$
by (*cases v*) *auto*

fixrec *CB-project* :: *CValue* → *CValue* → *CValue* → *CValue* **where**
CB-project·(*CB*·*db*)·v_1·v_2 = (*if undiscr db then* v_1 *else* v_2)

lemma [*simp*]:
CB-project·(*CB*·(*Discr b*))·v_1·v_2 = (*if b then* v_1 *else* v_2)
CB-project·⊥·v_1·v_2 = ⊥
CB-project·(*CFn*·*f*)·v_1·v_2 = ⊥
by *fixrec-simp*+

lemma *CB-project-not-bot*:
CB-project·*scrut*·v_1·v_2 ≠ ⊥ ⟷ (∃ *b*. *scrut* = *CB*·(*Discr b*) ∧ (*if b then* v_1 *else* v_2) ≠ ⊥)
apply (*cases scrut*)
apply *simp*
apply *simp*
by (*metis* (*poly-guards-query*) *CB-project.simps CValue.injects*(*2*) *discr.exhaust undiscr-Discr*)

HOLCF provides us *CValue-take*::*nat* ⇒ *CValue* → *CValue*; we want a similar function for *C* → *CValue*.

abbreviation *C-to-CValue-take* :: *nat* ⇒ (*C* → *CValue*) → (*C* → *CValue*)
where *C-to-CValue-take n* ≡ *cfun-map*·*ID*·(*CValue-take n*)

lemma *C-to-CValue-chain-take*: *chain C-to-CValue-take*
by (*auto intro*: *chainI cfun-belowI chainE*[*OF CValue.chain-take*] *monofun-cfun-fun*)

lemma *C-to-CValue-reach*: (⨆ *n*. *C-to-CValue-take n*·*x*) = *x*
by (*auto intro*: *cfun-eqI simp add*: *contlub-cfun-fun*[*OF ch2ch-Rep-cfunL*[*OF C-to-CValue-chain-take*]] *CValue.reach*)

end

6.5 CValue-Nominal

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

instantiation *C* :: *pure*
begin
definition *p* • (*c*::*C*) = *c*
instance by *standard* (*auto simp add*: *permute-C-def*)

end
instance *C* :: *pcpo-pt*
  by *standard* (*simp add*: *pure-permute-id*)

instantiation *CValue* :: *pure*
begin
  definition $p \cdot (v{::}CValue) = v$
instance
  apply *standard*
  apply (*auto simp add*: *permute-CValue-def*)
  done
end

instance *CValue* :: *pcpo-pt*
  by *standard* (*simp add*: *pure-permute-id*)

end

6.6 ResourcedDenotational

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

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

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.

notation *ESem-syn* (‹$\mathcal{N}[\![\ \text{-}\ ]\!]_{\text{-}}$› [*60*,*60*] *60*)
notation *EvalHeapSem-syn* (‹$\boldsymbol{\mathcal{N}}[\![\ \text{-}\ ]\!]_{\text{-}}$› [*0*,*0*] *110*)
notation *HSem-syn* (‹$\mathcal{N}\{\!|\text{-}|\!\}\text{-}$› [*60*,*60*] *60*)
notation *AHSem-bot* (‹$\mathcal{N}\{\!|\text{-}|\!\}$› [*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* $⟧_\varrho = (\Lambda\ (C \cdot r).\ (\mathcal{N}⟦body⟧_{\mathcal{N}\{\!|as|\!\}\varrho}) \cdot r)$
by (*auto simp add*: *eta-cfun*)

lemma *CESem-bot*[*simp*]:$(\mathcal{N}⟦\ e\ ⟧_\sigma)\cdot\bot = \bot$
by (*nominal-induct e arbitrary*: σ *rule*: *exp-strong-induct*) *auto*

lemma *CHSem-bot*[*simp*]:$((\mathcal{N}\{\!|\ \Gamma\ |\!\})\ x)\cdot\bot = \bot$
by (*cases* $x \in$ *domA* Γ) (*auto simp add*: *lookup-HSem-heap lookup-HSem-other*)

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

lemma *CESem-simps-no-tick*:
$(\mathcal{N}⟦\ Lam\ [x].\ e\ ⟧_\varrho)\cdot r \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}⟦\ e\ ⟧_{\varrho(x\ :=\ v)})|_r)$
$(\mathcal{N}⟦\ App\ e\ x\ ⟧_\varrho)\cdot r \quad \sqsubseteq ((\mathcal{N}⟦\ e\ ⟧_\varrho)\cdot r \downarrow CFn\ \varrho\ x|_r)\cdot r$
$\mathcal{N}⟦\ Var\ x\ ⟧_\varrho \qquad \sqsubseteq \varrho\ x$
$(\mathcal{N}⟦\ (scrut\ ?\ e_1 : e_2)\ ⟧_\varrho)\cdot r \sqsubseteq CB\text{-}project\cdot((\mathcal{N}⟦\ scrut\ ⟧_\varrho)\cdot r)\cdot((\mathcal{N}⟦\ e_1\ ⟧_\varrho)\cdot r)\cdot((\mathcal{N}⟦\ e_2\ ⟧_\varrho)\cdot r)$
$\mathcal{N}⟦\ Let\ as\ body\ ⟧_\varrho \quad \sqsubseteq\ \mathcal{N}⟦body⟧_{\mathcal{N}\{\!|as|\!\}\varrho}$
apply −
apply (*rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
apply (*rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
apply (*rule cfun-belowI*, *rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
apply (*rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
apply (*rule cfun-belowI*, *rule below-trans*[*OF monofun-cfun-arg*[*OF below-C*]], *simp*)
done

lemma *CELam-no-restr*: $(\mathcal{N}⟦\ Lam\ [x].\ e\ ⟧_\varrho)\cdot r \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}⟦\ e\ ⟧_{\varrho(x\ :=\ v)}))$
proof−
have $(\mathcal{N}⟦\ Lam\ [x].\ e\ ⟧_\varrho)\cdot r \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}⟦\ e\ ⟧_{\varrho(x\ :=\ v)})|_r)$ **by** (*rule CESem-simps-no-tick*)
also have $\ldots \sqsubseteq CFn\cdot(\Lambda\ v.\ (\mathcal{N}⟦\ e\ ⟧_{\varrho(x\ :=\ v)}))$
by (*intro cont2cont monofun-LAM below-trans*[*OF C-restr-below*] *monofun-cfun-arg below-refl fun-upd-mono*)
finally show *?thesis* **by** *this* (*intro cont2cont*)
qed

lemma *CEApp-no-restr*: $(\mathcal{N}⟦\ App\ e\ x\ ⟧_\varrho)\cdot r \quad \sqsubseteq ((\mathcal{N}⟦\ e\ ⟧_\varrho)\cdot r \downarrow CFn\ \varrho\ x)\cdot r$
proof−
have $(\mathcal{N}⟦\ App\ e\ x\ ⟧_\varrho)\cdot r \ \sqsubseteq ((\mathcal{N}⟦\ e\ ⟧_\varrho)\cdot r \downarrow CFn\ \varrho\ x|_r)\cdot r$ **by** (*rule CESem-simps-no-tick*)
also have $\varrho\ x|_r \sqsubseteq \varrho\ x$ **by** (*rule C-restr-below*)
finally show *?thesis* **by** *this* (*intro cont2cont*)
qed

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 $(\{\!|\Gamma|\!\}\varrho)\ f|\text{`}\ domA\ \Gamma = (\{\!|\Delta|\!\}\varrho)\ f|\text{`}\ domA\ \Gamma$
  using *assms*
proof(*nominal-induct arbitrary*: ϱ *rule*:*reds.strong-induct*)
case *Lambda*
  case *1* **show** *?case*..
  case *2* **show** *?case*..
next
case (*Application* $y\ \Gamma\ e\ x\ L\ \Delta\ \Theta\ v\ e'$)
  have *Gamma-subset*: $domA\ \Gamma \subseteq domA\ \Delta$
    by (*rule reds-doesnt-forget*[*OF Application.hyps*(*8*)])

  case *1*
  hence *prem1*: $fv\ (\Gamma,\ e) \subseteq set\ L \cup domA\ \Gamma$ **and** $x \in set\ L \cup domA\ \Gamma$ **by** *auto*
  moreover
  note *reds-pres-closed*[*OF Application.hyps*(*8*) *prem1*]
  moreover
  note *reds-doesnt-forget*[*OF Application.hyps*(*8*)]
  moreover
  have $fv\ (e'[y{::=}x]) \subseteq fv\ (Lam\ [y].\ e') \cup \{x\}$
    by (*auto simp add*: *fv-subst-eq*)
  ultimately
  have *prem2*: $fv\ (\Delta,\ e'[y{::=}x]) \subseteq set\ L \cup domA\ \Delta$ **by** *auto*

  have $*$: $(\{\!|\Gamma|\!\}\varrho)\ x = (\{\!|\Delta|\!\}\varrho)\ x$
  proof(*cases* $x \in domA\ \Gamma$)
    case *True*
    from *Application.hyps*(*10*)[*OF prem1*, **where** $\varrho = \varrho$]
    have $((\{\!|\Gamma|\!\}\varrho)\ f|\text{`}\ domA\ \Gamma)\ x = ((\{\!|\Delta|\!\}\varrho)\ f|\text{`}\ domA\ \Gamma)\ x$ **by** *simp*
    with *True* **show** *?thesis* **by** *simp*
  next
    case *False*
    from *False* ‹$x \in set\ L \cup domA\ \Gamma$› *reds-avoids-live*[*OF Application.hyps*(*8*)]
    show *?thesis* **by** (*auto simp add*: *lookup-HSem-other*)
  qed **have** $[\![\ App\ e\ x\ ]\!]_{\{\!|\Gamma|\!\}\varrho} = ([\![\ e\ ]\!]_{\{\!|\Gamma|\!\}\varrho})\ \downarrow Fn\ (\{\!|\Gamma|\!\}\varrho)\ x$
    by *simp*

also have ... $= (\llbracket\ Lam\ [y].\ e'\ \rrbracket_{\{\!|\Delta|\!\}\varrho})\ \downarrow Fn\ (\{\!|\Gamma|\!\}\varrho)\ x$
using *Application.hyps(9)[OF prem1]* **by** *simp*
also have ... $= (\llbracket\ Lam\ [y].\ e'\ \rrbracket_{\{\!|\Delta|\!\}\varrho})\ \downarrow Fn\ (\{\!|\Delta|\!\}\varrho)\ x$
unfolding ∗..
also have ... $= (Fn\cdot(\Lambda\ z.\ \llbracket\ e'\ \rrbracket_{(\{\!|\Delta|\!\}\varrho)(y\ :=\ z)}))\ \downarrow Fn\ (\{\!|\Delta|\!\}\varrho)\ x$
by *simp*
also have ... $= \llbracket\ e'\ \rrbracket_{(\{\!|\Delta|\!\}\varrho)(y\ :=\ (\{\!|\Delta|\!\}\varrho)\ x)}$
by *simp*
also have ... $= \llbracket\ e'[y\ ::=\ x]\ \rrbracket_{\{\!|\Delta|\!\}\varrho}$
unfolding *ESem-subst*..
also have ... $= \llbracket\ v\ \rrbracket_{\{\!|\Theta|\!\}\varrho}$
by (*rule Application.hyps(12)[OF prem2]*)
finally
show $\llbracket\ App\ e\ x\ \rrbracket_{\{\!|\Gamma|\!\}\varrho} = \llbracket\ v\ \rrbracket_{\{\!|\Theta|\!\}\varrho}$. **show** $(\{\!|\Gamma|\!\}\varrho)\ f|`\ domA\ \Gamma = (\{\!|\Theta|\!\}\varrho)\ f|`\ domA\ \Gamma$
using *Application.hyps(10)[OF prem1]*
env-restr-eq-subset[OF Gamma-subset Application.hyps(13)[OF prem2]]
by (*rule trans*)
next
case (*Variable* $\Gamma\ x\ e\ L\ \Delta\ v$)
hence [*simp*]:$x \in domA\ \Gamma$ **by** (*metis domA-from-set map-of-SomeD*)

let $?\Gamma = delete\ x\ \Gamma$

case *2*
have $x \notin domA\ \Delta$
by (*rule reds-avoids-live[OF Variable.hyps(2)], simp-all*)

have *subset*: $domA\ ?\Gamma \subseteq domA\ \Delta$
by (*rule reds-doesnt-forget[OF Variable.hyps(2)]*)

let $?new = domA\ \Delta - domA\ \Gamma$
have $fv\ (?\Gamma,\ e) \cup \{x\} \subseteq fv\ (\Gamma,\ Var\ x)$
by (*rule fv-delete-heap[OF ‹map-of Γ x = Some e›]*)
hence *prem*: $fv\ (?\Gamma,\ e) \subseteq set\ (x\ \#\ L) \cup domA\ ?\Gamma$ **using** *2* **by** *auto*
hence *fv-subset*: $fv\ (?\Gamma,\ e) - domA\ ?\Gamma \subseteq -\ ?new$
using *reds-avoids-live'[OF Variable.hyps(2)]* **by** *auto*

have $domA\ \Gamma \subseteq (-?new)$ **by** *auto*

have $\{\!|\Gamma|\!\}\varrho = \{\!|(x,e)\ \#\ ?\Gamma|\!\}\varrho$
by (*rule HSem-reorder[OF map-of-delete-insert[symmetric, OF Variable(1)]]*)
also have ... $= (\mu\ \varrho'.\ (\varrho\ {++}_{(domA\ ?\Gamma)}\ (\{\!|?\Gamma|\!\}\varrho'))(\ x := \llbracket\ e\ \rrbracket_{\varrho'}))$
by (*rule iterative-HSem, simp*)
also have ... $= (\mu\ \varrho'.\ (\varrho\ {++}_{(domA\ ?\Gamma)}\ (\{\!|?\Gamma|\!\}\varrho'))(\ x := \llbracket\ e\ \rrbracket_{\{\!|?\Gamma|\!\}\varrho'}))$
by (*rule iterative-HSem', simp*)
finally
have $(\{\!|\Gamma|\!\}\varrho) f|`\ (-\ ?new) = (...)\ f|`\ (-\ ?new)$ **by** *simp*
also have ... $= (\mu\ \varrho'.\ (\varrho\ {++}_{domA\ \Delta}\ (\{\!|\Delta|\!\}\varrho'))(\ x := \llbracket\ v\ \rrbracket_{\{\!|\Delta|\!\}\varrho'}))\ f|`\ (-\ ?new)$

proof (*induction rule*: *parallel-fix-ind*[**where** $P = \lambda\ x\ y.\ x\ f|`\ (-\ ?new) = y\ f|`\ (-\ ?new)$])
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **..**
next
case ($3\ \sigma\ \sigma'$)
hence $[\![\ e\ ]\!]_{\{\!|?\Gamma|\!\}\sigma} = [\![\ e\ ]\!]_{\{\!|?\Gamma|\!\}\sigma'}$
and $(\{\!|?\Gamma|\!\}\sigma)\ f|`\ domA\ ?\Gamma = (\{\!|?\Gamma|\!\}\sigma')\ f|`\ domA\ ?\Gamma$
using *fv-subset* **by** (*auto intro*: *ESem-fresh-cong HSem-fresh-cong env-restr-eq-subset*[*OF* - *3*])
from *trans*[*OF this*(*1*) *Variable*(*3*)[*OF prem*]] *trans*[*OF this*(*2*) *Variable*(*4*)[*OF prem*]]
have $[\![\ e\ ]\!]_{\{\!|?\Gamma|\!\}\sigma} = [\![\ v\ ]\!]_{\{\!|\Delta|\!\}\sigma'}$
and $(\{\!|?\Gamma|\!\}\sigma)\ f|`\ domA\ ?\Gamma = (\{\!|\Delta|\!\}\sigma')\ f|`\ domA\ ?\Gamma$.
thus *?case*
using *subset*
by (*fastforce simp add*: *lookup-override-on-eq lookup-env-restr-eq dest*: *env-restr-eqD*)
qed
also have ... $= (\mu\ \varrho'.\ (\varrho ++_{domA\ \Delta}\ (\{\!|\Delta|\!\}\varrho'))(\ x := [\![\ v\ ]\!]_{\varrho'}))\ f|`\ (-?new)$
by (*rule arg-cong*[*OF iterative-HSem'*[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
also have ... $= (\{\!|(x,v)\ \#\ \Delta|\!\}\varrho)\ \ f|`\ (-?new)$
by (*rule arg-cong*[*OF iterative-HSem*[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
finally
show *le*: *?case* **by** (*rule env-restr-eq-subset*[*OF* ‹$domA\ \Gamma \subseteq (-?new)$›])

have $[\![\ Var\ x\ ]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![\ Var\ x\ ]\!]_{\{\!|(x,\ v)\ \#\ \Delta|\!\}\varrho}$
using *env-restr-eqD*[*OF le*, **where** $x = x$]
by *simp*
also have ... $= [\![\ v\ ]\!]_{\{\!|(x,\ v)\ \#\ \Delta|\!\}\varrho}$
by (*auto simp add*: *lookup-HSem-heap*)
finally
show $[\![\ Var\ x\ ]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![\ v\ ]\!]_{\{\!|(x,\ v)\ \#\ \Delta|\!\}\varrho}$.
next
case (*Bool b*)
case *1*
show *?case* **by** *simp*
case *2*
show *?case* **by** *simp*
next
case (*IfThenElse* Γ *scrut* $L\ \Delta\ b\ e_1\ e_2\ \Theta\ v$)
have *Gamma-subset*: $domA\ \Gamma \subseteq domA\ \Delta$
by (*rule reds-doesnt-forget*[*OF IfThenElse.hyps*(*1*)])

let $?e = if\ b\ then\ e_1\ else\ e_2$

case *1*

hence *prem1*: $fv\ (\Gamma,\ scrut) \subseteq set\ L \cup domA\ \Gamma$
and *prem2*: $fv\ (\Delta,\ ?e) \subseteq set\ L \cup domA\ \Delta$

```
    and fv ?e ⊆ domA Γ ∪ set L
    using new-free-vars-on-heap[OF IfThenElse.hyps(1)] Gamma-subset by auto

  have ⟦ (scrut ? e1 : e2) ⟧⦃Γ⦄ϱ = B-project·(⟦ scrut ⟧⦃Γ⦄ϱ)·(⟦ e1 ⟧⦃Γ⦄ϱ)·(⟦ e2 ⟧⦃Γ⦄ϱ) by simp
  also have ... = B-project·(⟦ Bool b ⟧⦃Δ⦄ϱ)·(⟦ e1 ⟧⦃Γ⦄ϱ)·(⟦ e2 ⟧⦃Γ⦄ϱ)
    unfolding IfThenElse.hyps(2)[OF prem1]..
  also have ... = ⟦ ?e ⟧⦃Γ⦄ϱ by simp
  also have ... = ⟦ ?e ⟧⦃Δ⦄ϱ
    proof(rule ESem-fresh-cong-subset[OF ‹fv ?e ⊆ domA Γ ∪ set L› env-restr-eqI])
      fix x
      assume x ∈ domA Γ ∪ set L
      thus (⦃Γ⦄ϱ) x = (⦃Δ⦄ϱ) x
      proof(cases x ∈ domA Γ)
        assume x ∈ domA Γ
        from IfThenElse.hyps(3)[OF prem1]
        have ((⦃Γ⦄ϱ) f|‘ domA Γ) x = ((⦃Δ⦄ϱ) f|‘ domA Γ) x by simp
        with ‹x ∈ domA Γ› show ?thesis by simp
      next
        assume x ∉ domA Γ
        from this ‹x ∈ domA Γ ∪ set L› reds-avoids-live[OF IfThenElse.hyps(1)]
        show ?thesis
          by (simp add: lookup-HSem-other)
      qed
    qed
  also have ... = ⟦ v ⟧⦃Θ⦄ϱ
    unfolding IfThenElse.hyps(5)[OF prem2]..
  finally
  show ?case.
  thm env-restr-eq-subset
  show (⦃Γ⦄ϱ) f|‘ domA Γ = (⦃Θ⦄ϱ) f|‘ domA Γ
    using IfThenElse.hyps(3)[OF prem1]
        env-restr-eq-subset[OF Gamma-subset IfThenElse.hyps(6)[OF prem2]]
    by (rule trans)
next
case (Let as Γ L body Δ v)
  case 1
  { fix a
    assume a: a ∈ domA  as
    have atom a ♯ Γ
      by (rule Let(1)[unfolded fresh-star-def, rule-format, OF imageI[OF a]])
    hence a ∉ domA Γ
      by (metis domA-not-fresh)
  }
  note * = this


  have fv (as @ Γ, body) − domA (as @ Γ) ⊆  fv (Γ, Let as body) − domA Γ
    by auto
```

with *1* **have** *prem*: *fv* (*as* @ Γ, *body*) ⊆ *set L* ∪ *domA* (*as* @ Γ) **by** *auto*

have *f1*: *atom* ' *domA as* ♯* Γ
  using *Let*(*1*) **by** (*simp add*: *set-bn-to-atom-domA*)

have $[\![\, Let\ as\ body\,]\!]_{\{\!|\Gamma|\!\}\varrho} = [\![\, body\,]\!]_{\{\!|as|\!\}\{\!|\Gamma|\!\}\varrho}$
  by (*simp*)
also have ... $= [\![\, body\,]\!]_{\{\!|as\ @\ \Gamma|\!\}\varrho}$
  by (*rule arg-cong*[*OF HSem-merge*[*OF f1*]])
also have ... $= [\![\, v\,]\!]_{\{\!|\Delta|\!\}\varrho}$
  by (*rule Let.hyps*(*4*)[*OF prem*])
finally
show *?case*.

have $(\{\!|\Gamma|\!\}\varrho)\ f|`\ (domA\ \Gamma) = (\{\!|as|\!\}(\{\!|\Gamma|\!\}\varrho))\ f|`\ (domA\ \Gamma)$
  apply (*rule ext*)
  apply (*case-tac x* ∈ *domA as*)
  apply (*auto simp add*: *lookup-HSem-other lookup-env-restr-eq* ∗)
  done
also have ... $= (\{\!|as\ @\ \Gamma|\!\}\varrho)\ f|`\ (domA\ \Gamma)$
  by (*rule arg-cong*[*OF HSem-merge*[*OF f1*]])
also have ... $= (\{\!|\Delta|\!\}\varrho)\ f|`\ (domA\ \Gamma)$
  by (*rule env-restr-eq-subset*[*OF - Let.hyps*(*5*)[*OF prem*]]) *simp*
finally
show $(\{\!|\Gamma|\!\}\varrho)\ f|`\ domA\ \Gamma = (\{\!|\Delta|\!\}\varrho)\ f|`\ domA\ \Gamma$.
qed

end

7.2 CorrectnessResourced

theory *CorrectnessResourced*
  imports *ResourcedDenotational Launchbury*
begin

theorem *correctness*:
  assumes $\Gamma : e \Downarrow_L \Delta : z$
  and *fv* (Γ, *e*) ⊆ *set L* ∪ *domA* Γ
  shows $\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq \mathcal{N}[\![z]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho}$ **and** $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ f|`\ domA\ \Gamma \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\varrho)\ f|`\ domA\ \Gamma$
  using *assms*
proof(*nominal-induct arbitrary*: ϱ *rule*:*reds.strong-induct*)
case *Lambda*
  case *1* **show** *?case*..
  case *2* **show** *?case*..
next
case (*Application y* Γ *e x L* Δ Θ *z e'*)
  have *Gamma-subset*: *domA* Γ ⊆ *domA* Δ
    by (*rule reds-doesnt-forget*[*OF Application.hyps*(*8*)])

case *1*
hence *prem1*: $fv\ (\Gamma,\ e) \subseteq set\ L \cup domA\ \Gamma$ **and** $x \in set\ L \cup domA\ \Gamma$ **by** *auto*
moreover
note *reds-pres-closed*[*OF Application.hyps*(*8*) *prem1*]
moreover
note *reds-doesnt-forget*[*OF Application.hyps*(*8*)]
moreover
have $fv\ (e'[y{::=}x]) \subseteq fv\ (Lam\ [y].\ e') \cup \{x\}$
  by (*auto simp add*: *fv-subst-eq*)
ultimately
have *prem2*: $fv\ (\Delta,\ e'[y{::=}x]) \subseteq set\ L \cup domA\ \Delta$ **by** *auto*

have ∗: $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ x \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\varrho)\ x$
proof(*cases* $x \in domA\ \Gamma$)
  case *True*
  thus *?thesis*
    using *fun-belowD*[*OF Application.hyps*(*10*)[*OF prem1*], **where** $\varrho 1 = \varrho$ **and** $x = x$]
    by *simp*
next
  case *False*
  from *False* ‹$x \in set\ L \cup domA\ \Gamma$› *reds-avoids-live*[*OF Application.hyps*(*8*)]
  show *?thesis* **by** (*auto simp add*: *lookup-HSem-other*)
qed

{
fix *r*
have $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r \sqsubseteq ((\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r \downarrow CFn\ (\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ x)\cdot r$
  by (*rule CEApp-no-restr*)
also have $((\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})) \sqsubseteq ((\mathcal{N}[\![\ Lam\ [y].\ e'\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho}))$
  using *Application.hyps*(*9*)[*OF prem1*]**.**
also note ‹$((\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ x) \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\varrho)\ x$›
also have $(\mathcal{N}[\![\ Lam\ [y].\ e'\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho})\cdot r \sqsubseteq (CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e'\ ]\!]_{(\mathcal{N}\{\!|\Delta|\!\}\varrho)(y\ :=\ v)})))$
  by (*rule CELam-no-restr*)
also have $CFn\cdot(\Lambda\ v.\ (\mathcal{N}[\![\ e'\ ]\!]_{(\mathcal{N}\{\!|\Delta|\!\}\varrho)(y\ :=\ v)}))\downarrow CFn\ ((\mathcal{N}\{\!|\Delta|\!\}\varrho)\ x) = (\mathcal{N}[\![\ e'\ ]\!]_{(\mathcal{N}\{\!|\Delta|\!\}\varrho)(y\ :=\ (\mathcal{N}\{\!|\Delta|\!\}\varrho)\ x)})$
  by *simp*
also have $\ldots = (\mathcal{N}[\![\ e'[y\ {::=}\ x]\ ]\!]_{(\mathcal{N}\{\!|\Delta|\!\}\varrho)})$
  unfolding *ESem-subst***..**
also have $\ldots \sqsubseteq \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Theta|\!\}\varrho}$
  using *Application.hyps*(*12*)[*OF prem2*]**.**
finally
have $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r \sqsubseteq (\mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Theta|\!\}\varrho})\cdot r$ **by** *this* (*intro cont2cont*)+
}
thus *?case* **by** (*rule cfun-belowI*)

show $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ f|`\ (domA\ \Gamma) \sqsubseteq (\mathcal{N}\{\!|\Theta|\!\}\varrho)\ \ f|`\ (domA\ \Gamma)$
  using *Application.hyps*(*10*)[*OF prem1*]

env-restr-below-subset[*OF Gamma-subset Application.hyps*(*13*)[*OF prem2*]]
by (*rule below-trans*)
next
case (*Variable* Γ *x e L* Δ *z*)
hence [*simp*]:$x \in$ *domA* Γ
by (*metis domA-from-set map-of-SomeD*)

case *2*

have $x \notin$ *domA* Δ
by (*rule reds-avoids-live*[*OF Variable.hyps*(*2*)], *simp-all*)

have *subset*: *domA* (*delete x* Γ) $\subseteq$ *domA* Δ
by (*rule reds-doesnt-forget*[*OF Variable.hyps*(*2*)])

let *?new* = *domA* Δ − *domA* Γ
have *fv* (*delete x* Γ, *e*) $\cup$ $\{x\}$ $\subseteq$ *fv* (Γ, *Var x*)
by (*rule fv-delete-heap*[*OF* ‹*map-of* Γ *x* = *Some e*›])
hence *prem*: *fv* (*delete x* Γ, *e*) $\subseteq$ *set* (*x* # *L*) $\cup$ *domA* (*delete x* Γ) **using** *2* **by** *auto*
hence *fv-subset*: *fv* (*delete x* Γ, *e*) − *domA* (*delete x* Γ) $\subseteq$ − *?new*
using *reds-avoids-live'*[*OF Variable.hyps*(*2*)] **by** *auto*

have *domA* $\Gamma \subseteq$ (−*?new*) **by** *auto*

have $\mathcal{N}\{\!|\Gamma|\!\}\varrho = \mathcal{N}\{\!|(x,e)$ # *delete x* $\Gamma|\!\}\varrho$
by (*rule HSem-reorder*[*OF map-of-delete-insert*[*symmetric*, *OF Variable*(*1*)]])
also have ... $= (\mu\ \varrho'.\ (\varrho ++_{(domA\ (delete\ x\ \Gamma))} (\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\varrho'))(\ x := \mathcal{N}[\![\ e\ ]\!]_{\varrho'}))$
by (*rule iterative-HSem*, *simp*)
also have ... $= (\mu\ \varrho'.\ (\varrho ++_{(domA\ (delete\ x\ \Gamma))} (\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\varrho'))(\ x := \mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\varrho'}))$
by (*rule iterative-HSem'*, *simp*)
finally
have $(\mathcal{N}\{\!|\Gamma|\!\}\varrho) f|^{\backprime} (-\ ?new) \sqsubseteq (...)\ f|^{\backprime} (-\ ?new)$ **by** (*rule ssubst*) (*rule below-refl*)
also have ... $\sqsubseteq (\mu\ \varrho'.\ (\varrho ++_{domA\ \Delta} (\mathcal{N}\{\!|\Delta|\!\}\varrho'))(\ x := \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho'}))\ f|^{\backprime} (-\ ?new)$
proof (*induction rule*: *parallel-fix-ind*[**where** $P = \lambda\ x\ y.\ x\ f|^{\backprime} (-\ ?new) \sqsubseteq y\ f|^{\backprime} (-\ ?new)$])
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **..**
next
case (*3* σ σ')
hence $\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma} \sqsubseteq \mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma'}$
and $(\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma)\ f|^{\backprime}\ domA\ (delete\ x\ \Gamma) \sqsubseteq (\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma')\ f|^{\backprime}\ domA\ (delete\ x\ \Gamma)$
using *fv-subset* **by** (*auto intro*: *ESem-fresh-cong-below HSem-fresh-cong-below env-restr-below-subset*[*OF* - *3*])
from *below-trans*[*OF this*(*1*) *Variable*(*3*)[*OF prem*]] *below-trans*[*OF this*(*2*) *Variable*(*4*)[*OF prem*]]

have $\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma} \sqsubseteq \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\sigma'}$
and $(\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}\sigma)\ f|`\ domA\ (delete\ x\ \Gamma) \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\sigma')\ f|`\ domA\ (delete\ x\ \Gamma)$.
thus *?case*
using *subset*
by (*auto intro*!: *fun-belowI simp add*: *lookup-override-on-eq lookup-env-restr-eq elim*: *env-restr-belowD*)
qed
also have ... $= (\mu\ \varrho'.\ (\varrho ++_{domA\ \Delta} (\mathcal{N}\{\!|\Delta|\!\}\varrho'))(\ x := \mathcal{N}[\![\ z\ ]\!]_{\varrho'}))\ f|`\ (-?new)$
by (*rule arg-cong*[*OF iterative-HSem*$'$[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
also have ... $= (\mathcal{N}\{\!|(x,z)\ \#\ \Delta|\!\}\varrho)\ \ f|`\ (-?new)$
by (*rule arg-cong*[*OF iterative-HSem*[*symmetric*], *OF* ‹$x \notin domA\ \Delta$›])
finally
show *le*: *?case* **by** (*rule env-restr-below-subset*[*OF* ‹$domA\ \Gamma \subseteq (-?new)$›]) (*intro cont2cont*)+

have $\mathcal{N}[\![\ Var\ x\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq (\mathcal{N}\{\!|\Gamma|\!\}\varrho)\ x$ **by** (*rule CESem-simps-no-tick*)
also have ... $\sqsubseteq (\mathcal{N}\{\!|(x,\ z)\ \#\ \Delta|\!\}\varrho)\ x$
using *fun-belowD*[*OF le*, **where** $x = x$] **by** *simp*
also have ... $= \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|(x,\ z)\ \#\ \Delta|\!\}\varrho}$
by (*simp add*: *lookup-HSem-heap*)
finally
show $\mathcal{N}[\![\ Var\ x\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|(x,\ z)\ \#\ \Delta|\!\}\varrho}$ **by** *this* (*intro cont2cont*)+
next
case (*Bool b*)
case *1*
show *?case* **by** *simp*
case *2*
show *?case* **by** *simp*
next
case (*IfThenElse* Γ *scrut L* Δ *b* e_1 e_2 Θ *z*)
have *Gamma-subset*: $domA\ \Gamma \subseteq domA\ \Delta$
by (*rule reds-doesnt-forget*[*OF IfThenElse.hyps*(*1*)])

let *?e* = *if b then* e_1 *else* e_2

case *1*

hence *prem1*: $fv\ (\Gamma,\ scrut) \subseteq set\ L \cup domA\ \Gamma$
and *prem2*: $fv\ (\Delta,\ ?e) \subseteq set\ L \cup domA\ \Delta$
and $fv\ ?e \subseteq domA\ \Gamma \cup set\ L$
using *new-free-vars-on-heap*[*OF IfThenElse.hyps*(*1*)] *Gamma-subset* **by** *auto*

{
fix *r*
have $(\mathcal{N}[\![\ (scrut\ ?\ e_1 : e_2)\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r \sqsubseteq CB\text{-}project\cdot((\mathcal{N}[\![\ scrut\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_2\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)$
by (*rule CESem-simps-no-tick*)
also have ... $\sqsubseteq CB\text{-}project\cdot((\mathcal{N}[\![\ Bool\ b\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)\cdot((\mathcal{N}[\![\ e_2\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r)$
by (*intro monofun-cfun-fun monofun-cfun-arg IfThenElse.hyps*(*2*)[*OF prem1*])

also have ... = $(\mathcal{N}[\![~?e~]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r$ **by** (*cases r*) *simp-all*
also have ... $\sqsubseteq$ $(\mathcal{N}[\![~?e~]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho})\cdot r$
proof(*rule monofun-cfun-fun*[*OF ESem-fresh-cong-below-subset*[*OF* ‹*fv ?e* $\subseteq$ *domA* Γ ∪ *set L*› *Env.env-restr-belowI*]])
fix *x*
assume $x \in$ *domA* Γ ∪ *set L*
thus $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)~x \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\}\varrho)~x$
proof(*cases* $x \in$ *domA* Γ)
assume $x \in$ *domA* Γ
from *IfThenElse.hyps*(*3*)[*OF prem1*]
have $((\mathcal{N}\{\!|\Gamma|\!\}\varrho)~f|^{`}~domA~\Gamma)~x \sqsubseteq ((\mathcal{N}\{\!|\Delta|\!\}\varrho)~f|^{`}~domA~\Gamma)~x$ **by** (*rule fun-belowD*)
with ‹$x \in$ *domA* Γ› **show** *?thesis* **by** *simp*
next
assume $x \notin$ *domA* Γ
from *this* ‹$x \in$ *domA* Γ ∪ *set L*› *reds-avoids-live*[*OF IfThenElse.hyps*(*1*)]
show *?thesis*
by (*simp add*: *lookup-HSem-other*)
qed
qed
also have ... $\sqsubseteq$ $(\mathcal{N}[\![~z~]\!]_{\mathcal{N}\{\!|\Theta|\!\}\varrho})\cdot r$
by (*intro monofun-cfun-fun monofun-cfun-arg IfThenElse.hyps*(*5*)[*OF prem2*])
finally
have $(\mathcal{N}[\![~(scrut~?~e_1~:~e_2)~]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho})\cdot r \sqsubseteq (\mathcal{N}[\![~z~]\!]_{\mathcal{N}\{\!|\Theta|\!\}\varrho})\cdot r$ **by** *this* (*intro cont2cont*)+
}
thus *?case* **by** (*rule cfun-belowI*)

show $(\mathcal{N}\{\!|\Gamma|\!\}\varrho)~f|^{`}~(domA~\Gamma) \sqsubseteq (\mathcal{N}\{\!|\Theta|\!\}\varrho)~f|^{`}~(domA~\Gamma)$
using *IfThenElse.hyps*(*3*)[*OF prem1*]
env-restr-below-subset[*OF Gamma-subset IfThenElse.hyps*(*6*)[*OF prem2*]]
by (*rule below-trans*)
next
case (*Let as* Γ *L body* Δ *z*)
case *1*
have ∗: *domA as* ∩ *domA* Γ = {} **by** (*metis Let.hyps*(*1*) *fresh-distinct*)

have *fv* (*as* @ Γ, *body*) − *domA* (*as* @ Γ) ⊆ *fv* (Γ, *Let as body*) − *domA* Γ
by *auto*
with *1* **have** *prem*: *fv* (*as* @ Γ, *body*) ⊆ *set L* ∪ *domA* (*as* @ Γ) **by** *auto*

have *f1*: *atom* ‘ *domA as* ♯∗ Γ
using *Let*(*1*) **by** (*simp add*: *set-bn-to-atom-domA*)

have $\mathcal{N}[\![~Let~as~body~]\!]_{\mathcal{N}\{\!|\Gamma|\!\}\varrho} \sqsubseteq \mathcal{N}[\![~body~]\!]_{\mathcal{N}\{\!|as|\!\}\mathcal{N}\{\!|\Gamma|\!\}\varrho}$
by (*rule CESem-simps-no-tick*)
also have ... = $\mathcal{N}[\![~body~]\!]_{\mathcal{N}\{\!|as~@~\Gamma|\!\}\varrho}$
by (*rule arg-cong*[*OF HSem-merge*[*OF f1*]])
also have ... $\sqsubseteq$ $\mathcal{N}[\![~z~]\!]_{\mathcal{N}\{\!|\Delta|\!\}\varrho}$
by (*rule Let.hyps*(*4*)[*OF prem*])

finally
show *?case* **by** *this* (*intro cont2cont*)+

have ($\mathcal{N}\{\!|\Gamma|\!\}\varrho$) *f*|' (*domA* Γ) = ($\mathcal{N}\{\!|as|\!\}$($\mathcal{N}\{\!|\Gamma|\!\}\varrho$)) *f*|' (*domA* Γ)
unfolding *env-restr-HSem*[*OF* ∗]**..**
also have $\mathcal{N}\{\!|as|\!\}$($\mathcal{N}\{\!|\Gamma|\!\}\varrho$) = ($\mathcal{N}\{\!|as\ @\ \Gamma|\!\}\varrho$)
by (*rule HSem-merge*[*OF f1*])
also have ... *f*|' *domA* Γ ⊑ ($\mathcal{N}\{\!|\Delta|\!\}\varrho$) *f*|' *domA* Γ
by (*rule env-restr-below-subset*[*OF* - *Let.hyps*(*5*)[*OF prem*]]) *simp*
finally
show ($\mathcal{N}\{\!|\Gamma|\!\}\varrho$) *f*|' *domA* Γ ⊑ ($\mathcal{N}\{\!|\Delta|\!\}\varrho$) *f*|' *domA* Γ**.**
qed

corollary *correctness-empty-env*:
assumes Γ : *e* $\Downarrow_L$ Δ : *z*
and *fv* (Γ, *e*) ⊆ *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−
from *assms*(*2*) **have** *fv* (Γ, *e*) ⊆ *set L* ∪ *domA* Γ **by** *auto*
note *corr* = *correctness*[*OF assms*(*1*) *this*, **where** ϱ = ⊥]

show $\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![\ z\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **using** *corr*(*1*)**.**

have $\mathcal{N}\{\!|\Gamma|\!\}$ = ($\mathcal{N}\{\!|\Gamma|\!\}$) *f*|' *domA* Γ
using *env-restr-useless*[*OF HSem-edom-subset*, **where** *ϱ1* = ⊥] **by** *simp*
also have ... ⊑ ($\mathcal{N}\{\!|\Delta|\!\}$) *f*|' *domA* Γ **using** *corr*(*2*)**.**
also have ... ⊑ $\mathcal{N}\{\!|\Delta|\!\}$ **by** (*rule env-restr-below-itself*)
finally show $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$ **by** *this* (*intro cont2cont*)+
qed

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:

$$\begin{aligned}
\psi^E_n : E \to E_n &\quad \text{corresponds to} \quad \mathit{CValue\text{-}take}::\mathit{nat} \Rightarrow \mathit{CValue} \to \mathit{CValue} \\
\psi^A_n : A \to A_n &\quad \text{corresponds to} \quad \mathit{C\text{-}to\text{-}CValue\text{-}take}::\mathit{nat} \Rightarrow (C \to \mathit{CValue}) \to C \to \mathit{CValue} \\
\psi^D_n : D \to D_n &\quad \text{corresponds to} \quad \mathit{Value\text{-}take}::\mathit{nat} \Rightarrow \mathit{Value} \to \mathit{Value}.
\end{aligned}$$

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 $\text{-} \downarrow CFn \text{ -}:: CValue \Rightarrow (C \to CValue) \Rightarrow C \to CValue$.

To have our presentation closer to [SGHHOM11], we introduce some notation:

notation *Value-take* (‹ψ^D_›)
notation *C-to-CValue-take* (‹ψ^A_›)
notation *CValue-take* (‹ψ^E_›)

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*
proof−
  define $n :: nat$ **where** $n = 1$
  define d **where** $d = Fn\cdot(\Lambda\ e.\ (e \downarrow Fn\ \bot))$
  define d' **where** $d' = Fn\cdot(\Lambda\ \text{-}.\ Fn\cdot(\Lambda\ \text{-}.\ \bot))$
  have $Fn\cdot(\Lambda\ \text{-}.\ \bot) = \psi^D{}_n\cdot(d \downarrow Fn\ d')$

by (*simp add*: *d-def d′-def n-def cfun-map-def*)
also
have $\ldots = \psi^D{}_{Suc\ n} \cdot d \downarrow Fn\ \psi^D{}_n \cdot d'$
using *Equation* (*3*)**.**
also have $\ldots = \bot$
by (*simp add*: *d-def d′-def n-def*)
finally show *False* **by** *simp*
qed

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*
proof−
define $n :: nat$ **where** $n = 1$
define e **where** $e = CFn \cdot (\Lambda\ e\ r.\ (e \cdot r \downarrow CFn\ \bot) \cdot r)$
define $a :: C \rightarrow CValue$ **where** $a = (\Lambda\ \text{-}\ .\ CFn \cdot (\Lambda\ \text{-}\ \text{-}.\ CFn \cdot (\Lambda\ \text{-}\ \text{-}.\ \bot)))$
fix $c :: C$
have $CFn \cdot (\Lambda\ \text{-}\ \text{-}.\ \bot) = \psi^E{}_n \cdot ((e \downarrow CFn\ a) \cdot c)$
by (*simp add*: *e-def a-def n-def cfun-map-def*)
also
have $\ldots = (\psi^E{}_{Suc\ n} \cdot e \downarrow CFn\ \psi^A{}_n \cdot a) \cdot c$
using *Equation* (*2*)**.**
also have $\ldots = \bot$
by (*simp add*: *e-def a-def n-def*)
finally show *False* **by** *simp*
qed

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*):
$\psi^D{}_n \cdot (d \downarrow Fn\ \psi^D{}_n \cdot d') = \psi^D{}_{Suc\ n} \cdot d \downarrow Fn\ \psi^D{}_n \cdot d'$
by (*cases d*) (*auto simp add*: *Value.take-take*)

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** $x = \bot$ $y = CB \cdot (Discr\ b_2)$
by (*metis CValue.exhaust Discr-undiscr Value.exhaust*)

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*
proof−
have *case-prod P* ($\bigsqcup$ *n*. ($\psi^D{}_n \cdot x$, $\psi^A{}_n \cdot y$))
by (*rule admD*[*OF* ‹*adm* (*case-prod P*)› *ch2ch-Pair*[*OF ch2ch-Rep-cfunL*[*OF Value.chain-take*] *ch2ch-Rep-cfunL*[*OF C-to-CValue-chain-take*]]])
(*simp add*: *assms*(*2*))
hence *case-prod P* (*x*,*y*)
by (*simp add*: *lub-Pair*[*OF ch2ch-Rep-cfunL*[*OF Value.chain-take*] *ch2ch-Rep-cfunL*[*OF C-to-CValue-chain-take*]]
Value.reach C-to-CValue-reach)
thus *?thesis* **by** *simp*
qed

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* ⇒ *CValue* ⇒ *bool*) ⇒ *Value* ⇒ *CValue* ⇒ *bool* **for** *s* **where**
bot-similar′-step[*intro*!]: *similar′-step s* ⊥ ⊥ |
bool-similar′-step[*intro*]: *similar′-step s* ($B \cdot b$) ($CB \cdot b$) |
Fun-similar′-step[*intro*]: ($\bigwedge$ *x y* . *s x* ($y \cdot C^\infty$) ⟹ *s* ($f \cdot x$) ($g \cdot y \cdot C^\infty$)) ⟹ *similar′-step s* ($Fn \cdot f$) ($CFn \cdot g$)

inductive-cases [*elim*!]:
similar′-step s x ⊥
similar′-step s ⊥ *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 $similar'\text{-}syn\ x\ n\ y \equiv similar'\ n\ x\ y$

lemma *similar′-botI[intro!,simp]*: $\bot \lhd\rhd_n \bot$
by (*cases n*) (*auto simp add*: *similar′.simps*)

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 $Fn{\cdot}f \lhd\rhd_{Suc\ n}\ CFn{\cdot}g$
using *assms* **by** (*auto simp add*: *similar′.simps*)

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
using *assms* **by** (*auto simp add*: *similar′.simps*)

lemma *bot-or-not-bot′*:
$x \lhd\rhd_n y \Longrightarrow (x = \bot \longleftrightarrow y = \bot)$
by (*cases n*) (*auto simp add*: *similar′.simps elim*: *similar′-base.cases similar′-step.cases*)

lemma *similar′-bot[elim-format, elim!]*:
$\bot \lhd\rhd_n x \Longrightarrow x = \bot$
$y \lhd\rhd_n \bot \Longrightarrow y = \bot$
by (*metis bot-or-not-bot′*)+

lemma *similar′-typed[simp]*:
$\neg\ B{\cdot}b \lhd\rhd_n CFn{\cdot}g$
$\neg\ Fn{\cdot}f \lhd\rhd_n CB{\cdot}b$
by (*cases n*, *auto simp add*: *similar′.simps elim*: *similar′-base.cases similar′-step.cases*)+

lemma *similar′-bool[simp]*:
$B{\cdot}b_1 \lhd\rhd_{Suc\ n}\ CB{\cdot}b_2 \longleftrightarrow b_1 = b_2$
by (*auto simp add*: *similar′.simps elim*: *similar′-base.cases similar′-step.cases*)

8.1.4 Moving up and down the similarity relations

These correspond to Lemma 7 in [SGHHOM11].

lemma *similar′-down*: $d \lhd\rhd_{Suc\ n}\ e \Longrightarrow {\psi^D}_n{\cdot}d \lhd\rhd_n {\psi^E}_n{\cdot}e$
and *similar′-up*: $d \lhd\rhd_n e \Longrightarrow {\psi^D}_n{\cdot}d \lhd\rhd_{Suc\ n}\ {\psi^E}_n{\cdot}e$
proof (*induction n arbitrary*: *d e*)
case (*Suc n*) **case** *1* **with** *Suc*

show *?case*
by (*cases d e rule*:*value-CValue-cases*) *auto*
next
case (*Suc n*) **case** *2* **with** *Suc*
show *?case*
by (*cases d e rule*:*value-CValue-cases*) *auto*
qed *auto*

A generalisation of the above, doing multiple steps at once.

lemma *similar′-up-le*: $n \leq m \Longrightarrow \psi^D{}_n{\cdot}d \lhd\!\rhd_n \psi^E{}_n{\cdot}e \Longrightarrow \psi^D{}_n{\cdot}d \lhd\!\rhd_m \psi^E{}_n{\cdot}e$
by (*induction rule*: *dec-induct*)
(*auto dest*: *similar′-up simp add*: *Value.take-take CValue.take-take min-absorb2*)

lemma *similar′-down-le*: $n \leq m \Longrightarrow \psi^D{}_m{\cdot}d \lhd\!\rhd_m \psi^E{}_m{\cdot}e \Longrightarrow \psi^D{}_n{\cdot}d \lhd\!\rhd_n \psi^E{}_n{\cdot}e$
by (*induction rule*: *inc-induct*)
(*auto dest*: *similar′-down simp add*: *Value.take-take CValue.take-take min-absorb1*)

lemma *similar′-take*: $d \lhd\!\rhd_n e \Longrightarrow \psi^D{}_n{\cdot}d \lhd\!\rhd_n \psi^E{}_n{\cdot}e$
apply (*drule similar′-up*)
apply (*drule similar′-down*)
apply (*simp add*: *Value.take-take CValue.take-take*)
done

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*))
proof (*rule admI*, *goal-cases*)
case (*1 Y*)
then have $Y = (\lambda\ \text{-}\ .\ \bot)$ **by** (*metis prod.exhaust fst-eqD inst-prod-pcpo similar′-base.simps snd-eqD*)
thus *?case* **by** *auto*
qed

lemma *similar′-step-adm*:
assumes *adm* (λ *x*. *s* (*fst x*) (*snd x*))
shows *adm* (λ *x*. *similar′-step s* (*fst x*) (*snd x*))
proof (*rule admI*, *goal-cases*)
case *prems*: (*1 Y*)
from ‹*chain Y*›
have *chain* (λ *i*. *fst* (*Y i*)) **by** (*rule ch2ch-fst*)
thus *?case*
proof(*cases rule*: *Value-chainE*)
case *bot*
hence ∗: $\bigwedge$ *i*. *fst* (*Y i*) = ⊥ **by** *metis*
with *prems*(*2*)[*unfolded split-beta*]
have $\bigwedge$*i*. *snd* (*Y i*) = ⊥ **by** *auto*

hence $Y = (\lambda\ i.\ (\bot, \bot))$ **using** $*$ **by** (*metis surjective-pairing*)
thus *?thesis* **by** *auto*
next
case $(B\ n\ b)$
hence $\forall\, i.\ fst\ (Y\ (i + n)) = B{\cdot}b$ **by** (*metis add.commute not-add-less1*)
with *prems*(*2*)
have $\forall\, i.\ Y\ (i + n) = (B{\cdot}b,\ CB{\cdot}b)$
apply *auto*
apply (*erule-tac* $x = i + n$ **in** *allE*)
apply (*erule-tac* $x = i$ **in** *allE*)
apply (*erule similar′-step.cases*)
apply *auto*
by (*metis fst-conv old.prod.exhaust snd-conv*)
hence *similar′-step s* $(fst\ (\bigsqcup\ i.\ Y\ (i + n)))\ (snd\ (\bigsqcup\ i.\ Y\ (i + n)))$ **by** *auto*
thus *?thesis*
by (*simp add*: *lub-range-shift*[*OF* ‹*chain Y*›])
next
fix n
fix Y'
assume *chain* Y' **and** $(\lambda i.\ fst\ (Y\ i)) = (\lambda\ m.\ (if\ m < n\ then\ \bot\ else\ Fn{\cdot}(Y'\ (m{-}n))))$
hence Y': $\bigwedge\ i.\ fst\ (Y\ (i{+}n)) = Fn{\cdot}(Y'\ i)$ **by** (*metis add-diff-cancel-right′ not-add-less2*)
with *prems*(*2*)[*unfolded split-beta*]
have $\bigwedge i.\ \exists\ g'.\ snd\ (Y\ (i{+}n)) = CFn{\cdot}g'$
by −(*erule-tac* $x = i + n$ **in** *allE*, *auto elim*!: *similar′-step.cases*)
then obtain Y'' **where** Y'': $\bigwedge\ i.\ snd\ (Y\ (i{+}n)) = CFn{\cdot}(Y''\ i)$ **by** *metis*
from *prems*(*1*) **have** $\bigwedge i.\ Y\ i \sqsubseteq Y\ (Suc\ i)$
by (*simp add*: *po-class.chain-def*)
then have $*$: $\bigwedge i.\ Y\ (i + n) \sqsubseteq Y\ (Suc\ i + n)$
by *simp*
have *chain* Y''
apply (*rule chainI*)
apply (*rule iffD1*[*OF CValue.inverts*(*1*)])
apply (*subst* (*1 2*) Y''[*symmetric*])
apply (*rule snd-monofun*)
apply (*rule* $*$)
done

have *similar′-step s* $(Fn{\cdot}(\bigsqcup\ i.\ (Y'\ i)))\ (CFn\ \cdot\ (\bigsqcup\ i.\ Y''\ i))$
proof (*rule Fun-similar′-step*)
fix $x\ y$
from *prems*(*2*) $Y'\ Y''$
have $\bigwedge i.$ *similar′-step s* $(Fn{\cdot}(Y'\ i))\ (CFn{\cdot}(Y''\ i))$ **by** *metis*
moreover
assume $s\ x\ (y{\cdot}C^\infty)$
ultimately
have $\bigwedge i.\ s\ (Y'\ i{\cdot}x)\ (Y''\ i{\cdot}y{\cdot}C^\infty)$ **by** *auto*
hence *case-prod s* $(\bigsqcup\ i.\ ((Y'\ i){\cdot}x,\ (Y''\ i){\cdot}y{\cdot}C^\infty))$
apply −
apply (*rule admD*[*OF adm-case-prod*[**where** $P = \lambda$- . s, *OF assms*]])

apply (*simp add*: ‹*chain Y′*› ‹*chain Y′′*›)
apply *simp*
done
thus *s* ((⨆ *i*. *Y′ i*)·*x*) ((⨆ *i*. *Y′′ i*)·*y*·C^∞)
by (*simp add*: *lub-Pair ch2ch-Rep-cfunL contlub-cfun-fun* ‹*chain Y′*› ‹*chain Y′′*›)
qed
hence *similar′-step s* (*fst* (⨆ *i*. *Y* (*i*+*n*))) (*snd* (⨆ *i*. *Y* (*i*+*n*)))
by (*simp add*: *Y′ Y′′*
cont2contlubE[*OF cont-fst chain-shift*[*OF prems*(*1*)]] *cont2contlubE*[*OF cont-snd chain-shift*[*OF prems*(*1*)]]
contlub-cfun-arg[*OF* ‹*chain Y′′*›] *contlub-cfun-arg*[*OF* ‹*chain Y′*›])
thus *similar′-step s* (*fst* (⨆ *i*. *Y i*)) (*snd* (⨆ *i*. *Y i*))
by (*simp add*: *lub-range-shift*[*OF* ‹*chain Y*›])
qed
qed

lemma *similar′-adm*: *adm* (λ*x*. *fst x* ◁▷$_n$ *snd x*)
by (*induct n*) (*auto simp add*: *similar′.simps intro*: *similar′-base-adm similar′-step-adm*)

lemma *similar′-admI*: *cont f* ⟹ *cont g* ⟹ *adm* (λ*x*. *f x* ◁▷$_n$ *g x*)
by (*rule adm-subst*[*OF - similar′-adm*, **where** *t* = λ*x*. (*f x*, *g x*), *simplified*]) *auto*

8.1.6 The real similarity relation

This is the goal of the theory: A relation between *Value* and *CValue*.

definition *similar* :: *Value* ⇒ *CValue* ⇒ *bool* (**infix** ‹◁▷› *50*) **where**
x ◁▷ *y* ⟷ (∀ *n*. $\psi^D{}_n$·*x* ◁▷$_n$ $\psi^E{}_n$·*y*)

lemma *similarI*:
(⋀ *n*. $\psi^D{}_n$·*x* ◁▷$_n$ $\psi^E{}_n$·*y*) ⟹ *x* ◁▷ *y*
unfolding *similar-def* **by** *blast*

lemma *similarE*:
x ◁▷ *y* ⟹ $\psi^D{}_n$·*x* ◁▷$_n$ $\psi^E{}_n$·*y*
unfolding *similar-def* **by** *blast*

lemma *similar-bot*[*simp*]: ⊥ ◁▷ ⊥ **by** (*auto intro*: *similarI*)

lemma *similar-bool*[*simp*]: *B*·*b* ◁▷ *CB*·*b*
by (*rule similarI*, *case-tac n*, *auto*)

lemma [*elim-format*, *elim*!]: *x* ◁▷ ⊥ ⟹ *x* = ⊥
unfolding *similar-def*
apply (*cases x*)
apply *auto*
apply (*erule-tac x* = *Suc 0* **in** *allE*, *auto*)+
done

lemma [*elim-format*, *elim*!]: $x \lhd\rhd CB{\cdot}b \Longrightarrow x = B{\cdot}b$
unfolding *similar-def*
apply (*cases x*)
apply *auto*
apply (*erule-tac x* = *Suc 0* **in** *allE*, *auto*)+
done

lemma [*elim-format*, *elim*!]: $\bot \lhd\rhd y \Longrightarrow y = \bot$
unfolding *similar-def*
apply (*cases y*)
apply *auto*
apply (*erule-tac x* = *Suc 0* **in** *allE*, *auto*)+
done

lemma [*elim-format*, *elim*!]: $B{\cdot}b \lhd\rhd y \Longrightarrow y = CB{\cdot}b$
unfolding *similar-def*
apply (*cases y*)
apply *auto*
apply (*erule-tac x* = *Suc 0* **in** *allE*, *auto*)+
done

lemma *take-similar′-similar*:
assumes $x \lhd\rhd_n y$
shows $\psi^D{}_n{\cdot}x \lhd\rhd \psi^E{}_n{\cdot}y$
proof(*rule similarI*)
fix *m*
from *assms*
have $\psi^D{}_n{\cdot}x \lhd\rhd_n \psi^E{}_n{\cdot}y$ **by** (*rule similar′-take*)
moreover
have $n \leq m \lor m \leq n$ **by** *auto*
ultimately
show $\psi^D{}_m{\cdot}(\psi^D{}_n{\cdot}x) \lhd\rhd_m \psi^E{}_m{\cdot}(\psi^E{}_n{\cdot}y)$
by (*auto elim*: *similar′-up-le similar′-down-le dest*: *similar′-take*
simp add: *min-absorb2 min-absorb1 Value.take-take CValue.take-take*)
qed

lemma *bot-or-not-bot*:
$x \lhd\rhd y \Longrightarrow (x = \bot \longleftrightarrow y = \bot)$
by (*cases x y rule:value-CValue-cases*) *auto*

lemma *bool-or-not-bool*:
$x \lhd\rhd y \Longrightarrow (x = B{\cdot}b) \longleftrightarrow (y = CB{\cdot}b)$
by (*cases x y rule:value-CValue-cases*) *auto*

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** $x = Fn \cdot f\ y = CFn \cdot g$
using *assms*
by (*metis CValue.exhaust Value.exhaust bool-or-not-bool bot-or-not-bot discr.exhaust*)

lemma *similar-adm*: *adm* ($\lambda x.$ *fst* x ◁▷ *snd* x)
unfolding *similar-def*
by (*intro adm-lemmas similar′-admI cont2cont*)

lemma *similar-admI*: *cont f* $\Longrightarrow$ *cont g* $\Longrightarrow$ *adm* ($\lambda x.\ f\ x$ ◁▷ $g\ x$)
by (*rule adm-subst*[*OF - similar-adm*, **where** $t = \lambda x.\ (f\ x,\ g\ x)$, *simplified*]) *auto*

Having constructed the relation we can how show that it indeed is the desired relation, relating ⊥ with ⊥ and functions with functions, if they take related arguments to related values. This corresponds to Proposition 9 in [SGHHOM11].

lemma *similar-nice-def*: x ◁▷ $y \longleftrightarrow (x = \bot \wedge y = \bot \vee (\exists\ b.\ x = B\cdot(Discr\ b) \wedge y = CB\cdot(Discr\ b)) \vee (\exists\ f\ g.\ x = Fn\cdot f \wedge y = CFn\cdot g \wedge (\forall\ a\ b.\ a$ ◁▷ $b\cdot C^\infty \longrightarrow f\cdot a$ ◁▷ $g\cdot b\cdot C^\infty)))$
(**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
thus *?R*
proof (*cases x y rule*:*slimilar-bot-cases*)
case *bot* **thus** *?thesis* **by** *simp*
next
case *bool* **thus** *?thesis* **by** *simp*
next
case (*Fn f g*)
note ‹*?L*›[*unfolded Fn*]
have $\forall\ a\ b.\ a$ ◁▷ $b\cdot C^\infty \longrightarrow f\cdot a$ ◁▷ $g\cdot b\cdot C^\infty$
proof(*intro impI allI*)
fix *a b*
assume a ◁▷ $b\cdot C^\infty$

show $f\cdot a$ ◁▷ $g\cdot b\cdot C^\infty$
proof(*rule similarI*)
fix *n*
have *adm* ($\lambda(b,\ a).\ \psi^D{}_n\cdot(f\cdot b)$ ◁▷$_n$ $\psi^E{}_n\cdot(g\cdot a\cdot C^\infty)$)
by (*intro adm-case-prod similar′-admI cont2cont*)
thus $\psi^D{}_n\cdot(f\cdot a)$ ◁▷$_n$ $\psi^E{}_n\cdot(g\cdot b\cdot C^\infty)$
proof (*induct a b rule*: *Value-CValue-take-induct*[*consumes 1*])

This take induction is required to avoid the wrong equation shown above.

fix *m*

from ‹a ◁▷ $b\cdot C^\infty$›
have $\psi^D{}_m\cdot a$ ◁▷$_m$ $\psi^E{}_m\cdot(b\cdot C^\infty)$ **by** (*rule similarE*)
hence $\psi^D{}_m\cdot a$ ◁▷$_{max\ m\ n}$ $\psi^E{}_m\cdot(b\cdot C^\infty)$ **by** (*rule similar′-up-le*[*rotated*]) *auto*
moreover
from ‹$Fn\cdot f$ ◁▷ $CFn\cdot g$›

have $\psi^D{}_{Suc\ (max\ m\ n)}\cdot(Fn\cdot f)\ \lhd\rhd_{Suc\ (max\ m\ n)}\ \psi^E{}_{Suc\ (max\ m\ n)}\cdot(CFn\cdot g)$ **by** (*rule similarE*)
ultimately
have $\psi^D{}_{max\ m\ n}\cdot(f\cdot(\psi^D{}_{max\ m\ n}\cdot(\psi^D{}_m\cdot a)))\ \lhd\rhd_{max\ m\ n}\ \psi^E{}_{max\ m\ n}\cdot(g\cdot(\psi^A{}_{max\ m\ n}\cdot(\psi^A{}_m\cdot b))\cdot C^\infty)$
by *auto*
hence $\psi^D{}_{max\ m\ n}\cdot(f\cdot(\psi^D{}_m\cdot a))\ \lhd\rhd_{max\ m\ n}\ \psi^E{}_{max\ m\ n}\cdot(g\cdot(\psi^A{}_m\cdot b)\cdot C^\infty)$
by (*simp add*: *Value.take-take cfun-map-map CValue.take-take ID-def eta-cfun min-absorb2 min-absorb1*)
thus $\psi^D{}_n\cdot(f\cdot(\psi^D{}_m\cdot a))\ \lhd\rhd_n\ \psi^E{}_n\cdot(g\cdot(\psi^A{}_m\cdot b)\cdot C^\infty)$
by (*rule similar′-down-le*[*rotated*]) *auto*
qed
qed
qed
thus *?thesis* **unfolding** *Fn* **by** *simp*
qed
next
assume *?R*
thus *?L*
proof(*elim conjE disjE exE ssubst*)
show $\bot\ \lhd\rhd\ \bot$ **by** *simp*
next
fix *b*
show $B\cdot(Discr\ b)\ \lhd\rhd\ CB\cdot(Discr\ b)$ **by** *simp*
next
fix *f g*
assume *imp*: $\forall\ a\ b.\ a\ \lhd\rhd\ b\cdot C^\infty \longrightarrow f\cdot a\ \lhd\rhd\ g\cdot b\cdot C^\infty$
show $Fn\cdot f\ \lhd\rhd\ CFn\cdot g$
proof (*rule similarI*)
fix *n*
show $\psi^D{}_n\cdot(Fn\cdot f)\ \lhd\rhd_n\ \psi^E{}_n\cdot(CFn\cdot g)$
proof(*cases n*)
case *0* **thus** *?thesis* **by** *simp*
next
case (*Suc n*)
{ fix *x y*
assume $x\ \lhd\rhd_n\ y\cdot C^\infty$
hence $\psi^D{}_n\cdot x\ \lhd\rhd\ \psi^E{}_n\cdot(y\cdot C^\infty)$ **by** (*rule take-similar′-similar*)
hence $f\cdot(\psi^D{}_n\cdot x)\ \lhd\rhd\ g\cdot(\psi^A{}_n\cdot y)\cdot C^\infty$ **using** *imp* **by** *auto*
hence $\psi^D{}_n\cdot(f\cdot(\psi^D{}_n\cdot x))\ \lhd\rhd_n\ \psi^E{}_n\cdot(g\cdot(\psi^A{}_n\cdot y)\cdot C^\infty)$
by (*rule similarE*)
}
with *Suc*
show *?thesis* **by** *auto*
qed
qed
qed
qed

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 $Fn{\cdot}f \triangleleft\triangleright CFn{\cdot}g$
by (*metis assms similar-nice-def*)

lemma *similar-FnD*[*elim*!]:
assumes $Fn{\cdot}f \triangleleft\triangleright CFn{\cdot}g$
assumes $x \triangleleft\triangleright y{\cdot}C^{\infty}$
shows $f{\cdot}x \triangleleft\triangleright g{\cdot}y{\cdot}C^{\infty}$
using *assms*
by (*subst* (*asm*) *similar-nice-def*) *auto*

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
by (*metis assms similar-FnD*)

8.1.7 The similarity relation lifted pointwise to functions.

abbreviation *fun-similar* :: $('a{::}type \Rightarrow Value) \Rightarrow ('a \Rightarrow (C \to CValue)) \Rightarrow bool$ (**infix** ‹$\triangleleft\triangleright^*$› *50*) **where**
fun-similar ≡ *pointwise* $(\lambda x\ y.\ x \triangleleft\triangleright y{\cdot}C^{\infty})$

lemma *fun-similar-fmap-bottom*[*simp*]: $\bot \triangleleft\triangleright^* \bot$
by *auto*

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
using *assms* **unfolding** *pointwise-def* **by** *blast*

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 $\varrho \triangleleft\triangleright^* \sigma$
shows $[\![e]\!]_{\varrho} \triangleleft\triangleright (\mathcal{N}[\![e]\!]_{\sigma}){\cdot}C^{\infty}$
using *assms*
proof(*induct e arbitrary*: $\varrho\ \sigma$ *rule*:*exp-induct*)
case (*Var v*)

from *Var* **have** $\varrho\ v \triangleleft\triangleright (\sigma\ v)\cdot C^{\infty}$ **by** *cases auto*
thus *?case* **by** *simp*
next
case (*Lam v e*)
{ fix *x y*
assume $x \triangleleft\triangleright y\cdot C^{\infty}$
with $\langle \varrho \triangleleft\triangleright^{*} \sigma \rangle$
have $\varrho(v := x) \triangleleft\triangleright^{*} \sigma(v := y)$
by (*auto 1 4*)
hence $[\![e]\!]_{\varrho(v := x)} \triangleleft\triangleright (\mathcal{N}[\![e]\!]_{\sigma(v := y)})\cdot C^{\infty}$
by (*rule Lam.hyps*)
}
thus *?case* **by** *auto*
next
case (*App e v* $\varrho\ \sigma$)
hence *App′*: $[\![e]\!]_{\varrho} \triangleleft\triangleright (\mathcal{N}[\![e]\!]_{\sigma})\cdot C^{\infty}$ **by** *auto*
thus *?case*
proof (*cases rule*: *slimilar-bot-cases*)
case (*Fn f g*)
from $\langle \varrho \triangleleft\triangleright^{*} \sigma \rangle$
have $\varrho\ v \triangleleft\triangleright (\sigma\ v)\cdot C^{\infty}$ **by** *auto*
thus *?thesis* **using** *Fn App′* **by** *auto*
qed *auto*
next
case (*Bool b*)
thus $[\![Bool\ b]\!]_{\varrho} \triangleleft\triangleright (\mathcal{N}[\![Bool\ b]\!]_{\sigma})\cdot C^{\infty}$ **by** *auto*
next
case (*IfThenElse scrut* $e_1\ e_2$)
hence *IfThenElse′*:
$[\![\ scrut\ ]\!]_{\varrho} \triangleleft\triangleright (\mathcal{N}[\![\ scrut\ ]\!]_{\sigma})\cdot C^{\infty}$
$[\![\ e_1\ ]\!]_{\varrho} \triangleleft\triangleright (\mathcal{N}[\![\ e_1\ ]\!]_{\sigma})\cdot C^{\infty}$
$[\![\ e_2\ ]\!]_{\varrho} \triangleleft\triangleright (\mathcal{N}[\![\ e_2\ ]\!]_{\sigma})\cdot C^{\infty}$ **by** *auto*
from *IfThenElse′*(*1*)
show *?case*
proof (*cases rule*: *slimilar-bot-cases*)
case (*bool b*)
thus *?thesis* **using** *IfThenElse′* **by** *auto*
qed *auto*
next
case (*Let as e* $\varrho\ \sigma$)
have $\{\!|as|\!\}\varrho \triangleleft\triangleright^{*} \mathcal{N}\{\!|as|\!\}\sigma$
proof (*rule parallel-HSem-ind-different-ESem*[*OF pointwise-adm*[*OF similar-admI*] *fun-similar-fmap-bottom*])
fix $\varrho' :: var \Rightarrow Value$ **and** $\sigma' :: var \Rightarrow C \to CValue$
assume $\varrho' \triangleleft\triangleright^{*} \sigma'$
show $\varrho \mathbin{++_{domA\ as}} [\![\ as\ ]\!]_{\varrho'} \triangleleft\triangleright^{*} \sigma \mathbin{++_{domA\ as}} evalHeap\ as\ (\lambda e.\ \mathcal{N}[\![\ e\ ]\!]_{\sigma'})$
proof(*rule pointwiseI*, *goal-cases*)
case (*1 x*)
show *?case* **using** $\langle \varrho \triangleleft\triangleright^{*} \sigma \rangle$
by (*auto simp add*: *lookup-override-on-eq lookupEvalHeap elim*: *Let*(*1*)[*OF -* $\langle \varrho' \triangleleft\triangleright^{*} \sigma' \rangle$]

)
  qed
 qed *auto*
 hence $[\![e]\!]_{\{\!|as|\!\}\varrho} \triangleleft\triangleright (\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|as|\!\}\sigma})\cdot C^{\infty}$ **by** (*rule Let*(*2*))
 thus *?case* **by** *simp*
qed

corollary *evalHeap-similar*:
 $\bigwedge y\ z.\ y \triangleleft\triangleright^{*} z \Longrightarrow [\![\ \Gamma\ ]\!]_{y} \triangleleft\triangleright^{*} \boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{z}$
 by (*rule pointwiseI*)
  (*case-tac* $x \in$ *domA* Γ, *auto simp add*: *lookupEvalHeap denotational-semantics-similar*)

theorem *heaps-similar*: $\{\!|\Gamma|\!\} \triangleleft\triangleright^{*} \mathcal{N}\{\!|\Gamma|\!\}$
 by (*rule parallel-HSem-ind-different-ESem*[*OF pointwise-adm*[*OF similar-admI*]])
  (*auto simp add*: *evalHeap-similar*)

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
  assume *demand* $(\mathcal{N}[\![e]\!]_{\varrho}) = \bot$
  with *demand-suffices′*[**where** $n = 0$, *simplified*, *OF this*]
  have $(\mathcal{N}[\![e]\!]_{\varrho})\cdot\bot \neq \bot$ **by** *simp*
  thus *False* **by** *simp*
qed

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(*induction e arbitrary*: ϱ *r rule*: *exp-induct*)
  case (*Var x*)
  show *?case*
  proof (*rule C-restr-C-cong*)
    fix r'
    assume $r' \sqsubseteq r$
    have $(\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho})\cdot(C\cdot r') = \varrho\ x\cdot r'$ **by** *simp*
    also have $\ldots = ((\varrho\ x)|_{r})\cdot r'$ **using** ‹$r' \sqsubseteq r$› **by** *simp*
    also have $\ldots = (\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho|^{\circ}{}_{r}})\cdot(C\cdot r')$ **by** *simp*
    finally show $(\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho})\cdot(C\cdot r') = (\mathcal{N}[\![\ Var\ x\ ]\!]_{\varrho|^{\circ}{}_{r}})\cdot(C\cdot r')$.
  qed *simp*
next
  case (*Lam x e*)
  show *?case*
  proof(*rule C-restr-C-cong*)
    fix r'
    assume $r' \sqsubseteq r$
    hence $r' \sqsubseteq C\cdot r$ **by** (*metis below-C below-trans*)
    {
      fix v
      have $\varrho(x := v)|^{\circ}{}_{r} = (\varrho|^{\circ}{}_{r})(x := v)|^{\circ}{}_{r}$
        by *simp*
      hence $(\mathcal{N}[\![\ e\ ]\!]_{\varrho(x\ :=\ v)})|_{r'} = (\mathcal{N}[\![\ e\ ]\!]_{(\varrho|^{\circ}{}_{r})(x\ :=\ v)})|_{r'}$
        by (*subst* (*1 2*) *C-restr-eq-lower*[*OF Lam* ‹$r' \sqsubseteq C\cdot r$›]) *simp*
    }
    thus $(\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho})\cdot(C\cdot r') = (\mathcal{N}[\![\ Lam\ [x].\ e\ ]\!]_{\varrho|^{\circ}{}_{r}})\cdot(C\cdot r')$
      by *simp*
  qed *simp*
next

case $(App\ e\ x)$
show *?case*
proof (*rule C-restr-C-cong*)
fix r'
assume $r' \sqsubseteq r$
hence $r' \sqsubseteq C \cdot r$ **by** (*metis below-C below-trans*)
hence $(\mathcal{N}[\![\ e\ ]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![\ e\ ]\!]_{\varrho|^{\circ}r}) \cdot r'$
by (*rule C-restr-eqD*[*OF App*])
thus $(\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho}) \cdot (C \cdot r') = (\mathcal{N}[\![\ App\ e\ x\ ]\!]_{\varrho|^{\circ}r}) \cdot (C \cdot r')$
using $\langle r' \sqsubseteq r \rangle$ **by** *simp*
qed *simp*
next
case $(Bool\ b)$
show *?case* **by** *simp*
next
case $(IfThenElse\ scrut\ e_1\ e_2)$
show *?case*
proof (*rule C-restr-C-cong*)
fix r'
assume $r' \sqsubseteq r$
hence $r' \sqsubseteq C \cdot r$ **by** (*metis below-C below-trans*)

have $(\mathcal{N}[\![\ scrut\ ]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![\ scrut\ ]\!]_{\varrho|^{\circ}r}) \cdot r'$
using $\langle r' \sqsubseteq C \cdot r \rangle$ **by** (*rule C-restr-eqD*[*OF IfThenElse*(*1*)])
moreover
have $(\mathcal{N}[\![\ e_1\ ]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![\ e_1\ ]\!]_{\varrho|^{\circ}r}) \cdot r'$
using $\langle r' \sqsubseteq C \cdot r \rangle$ **by** (*rule C-restr-eqD*[*OF IfThenElse*(*2*)])
moreover
have $(\mathcal{N}[\![\ e_2\ ]\!]_{\varrho}) \cdot r' = (\mathcal{N}[\![\ e_2\ ]\!]_{\varrho|^{\circ}r}) \cdot r'$
using $\langle r' \sqsubseteq C \cdot r \rangle$ **by** (*rule C-restr-eqD*[*OF IfThenElse*(*3*)])
ultimately
show $(\mathcal{N}[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho}) \cdot (C \cdot r') = (\mathcal{N}[\![\ (scrut\ ?\ e_1\ :\ e_2)\ ]\!]_{\varrho|^{\circ}r}) \cdot (C \cdot r')$
using $\langle r' \sqsubseteq r \rangle$ **by** *simp*
qed *simp*
next
case $(Let\ \Gamma\ e)$

The lemma, lifted to heaps

have *restr-can-restrict-env-heap* : $\bigwedge r.\ (\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho)|^{\circ}r = (\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho|^{\circ}r)|^{\circ}r$
proof(*rule has-ESem.parallel-HSem-ind*)
fix $\varrho_1\ \varrho_2 :: CEnv$ **and** $r :: C$
assume $\varrho_1|^{\circ}r = \varrho_2|^{\circ}r$

show $(\varrho \mathbin{++}_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho_1})|^{\circ}r = (\varrho|^{\circ}r \mathbin{++}_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho_2})|^{\circ}r$
proof(*rule env-C-restr-cong*)
fix x **and** r'
assume $r' \sqsubseteq r$
hence $r' \sqsubseteq C \cdot r$ **by** (*metis below-C below-trans*)

show $(\varrho \mathbin{++}_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho_1})\ x\cdot r' = (\varrho|^\circ{}_r \mathbin{++}_{domA\ \Gamma} \boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho_2})\ x\cdot r'$
proof(*cases* $x \in$ *domA* Γ)
case *True*
have $(\mathcal{N}[\![\ \textit{the}\ (\textit{map-of}\ \Gamma\ x)\ ]\!]_{\varrho_1})\cdot r' = (\mathcal{N}[\![\ \textit{the}\ (\textit{map-of}\ \Gamma\ x)\ ]\!]_{\varrho_1|^\circ{}_r})\cdot r'$
by (*rule C-restr-eqD*[*OF Let*(*1*)[*OF True*] ‹$r' \sqsubseteq C\cdot r$›])
also have ... $= (\mathcal{N}[\![\ \textit{the}\ (\textit{map-of}\ \Gamma\ x)\ ]\!]_{\varrho_2|^\circ{}_r})\cdot r'$
unfolding ‹$\varrho_1|^\circ{}_r = \varrho_2|^\circ{}_r$›..
also have ... $= (\mathcal{N}[\![\ \textit{the}\ (\textit{map-of}\ \Gamma\ x)\ ]\!]_{\varrho_2})\cdot r'$
by (*rule C-restr-eqD*[*OF Let*(*1*)[*OF True*] ‹$r' \sqsubseteq C\cdot r$›, *symmetric*])
finally
show *?thesis* **using** *True* **by** (*simp add*: *lookupEvalHeap*)
next
case *False*
with ‹$r' \sqsubseteq r$›
show *?thesis* **by** *simp*
qed
qed
qed *simp-all*

show *?case*
proof (*rule C-restr-C-cong*)
fix r'
assume $r' \sqsubseteq r$
hence $r' \sqsubseteq C\cdot r$ **by** (*metis below-C below-trans*)

have $(\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho)|^\circ{}_r = (\mathcal{N}\{\!\!\{\Gamma\}\!\!\}(\varrho|^\circ{}_r))|^\circ{}_r$
by (*rule restr-can-restrict-env-heap*)
hence $(\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho})\cdot r' = (\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!\!\{\Gamma\}\!\!\}\varrho|^\circ{}_r})\cdot r'$
by (*subst* (*1 2*) *C-restr-eqD*[*OF Let*(*2*) ‹$r' \sqsubseteq C\cdot r$›]) *simp*

thus $(\mathcal{N}[\![\ \textit{Let}\ \Gamma\ e\ ]\!]_{\varrho})\cdot(C\cdot r') = (\mathcal{N}[\![\ \textit{Let}\ \Gamma\ e\ ]\!]_{\varrho|^\circ{}_r})\cdot(C\cdot r')$
using ‹$r' \sqsubseteq r$› **by** *simp*
qed *simp*
qed

lemma *can-restrict-env*:
$(\mathcal{N}[\![e]\!]_{\varrho})\cdot(C\cdot r) = (\mathcal{N}[\![\ e\ ]\!]_{\varrho|^\circ{}_r})\cdot(C\cdot r)$
by (*rule C-restr-eqD*[*OF restr-can-restrict-env below-refl*])

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-of* Γ $x =$ *Some* e
assumes $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\!\{\Gamma\}\!\!\}})\cdot r' \neq \bot$
shows $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\!\{\textit{delete}\ x\ \Gamma\}\!\!\}})\cdot r' \neq \bot$

proof−
obtain r **where** r: $C \cdot r = demand\ (\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})$
using *demand-not-0* **by** (*cases demand* $(\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})$) *auto*

from *assms*(*2*)
have $C \cdot r \sqsubseteq r'$ **unfolding** *r not-bot-demand* **by** *simp*

from *assms*(*1*)
have [*simp*]: *the* (*map-of* Γ x) = e **by** (*metis option.sel*)

from *assms*(*1*)
have [*simp*]: $x \in domA\ \Gamma$ **by** (*metis domIff dom-map-of-conv-domA not-Some-eq*)

define *ub* **where** $ub = \mathcal{N}\{\!|\Gamma|\!\}$ — An upper bound for the induction

have *heaps*: $(\mathcal{N}\{\!|\Gamma|\!\})|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|delete\ x\ \Gamma|\!\}$ **and** $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq ub$
proof (*induction rule*: *HSem-bot-ind*)
fix ϱ
assume $\varrho|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|delete\ x\ \Gamma|\!\}$
assume $\varrho \sqsubseteq ub$

show $(\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho})|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|delete\ x\ \Gamma|\!\}$
proof (*rule fun-belowI*)
fix y
show $((\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho})|^{\circ}{}_{r})\ y \sqsubseteq (\mathcal{N}\{\!|delete\ x\ \Gamma|\!\})\ y$
proof (*cases* $y = x$)
case *True*
have $((\boldsymbol{\mathcal{N}}[\![\ \Gamma\ ]\!]_{\varrho})|^{\circ}{}_{r})\ x = (\mathcal{N}[\![\ e\ ]\!]_{\varrho})|_{r}$
by (*simp add*: *lookupEvalHeap*)
also have $\ldots \sqsubseteq (\mathcal{N}[\![\ e\ ]\!]_{ub})|_{r}$
using ‹$\varrho \sqsubseteq ub$› **by** (*intro monofun-cfun-arg*)
also have $\ldots = (\mathcal{N}[\![\ e\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})|_{r}$
unfolding *ub-def*..
also have $\ldots = \bot$
using *r* **by** (*rule C-restr-bot-demand*[*OF eq-imp-below*])
also have $\ldots = (\mathcal{N}\{\!|delete\ x\ \Gamma|\!\})\ x$
by (*simp add*: *lookup-HSem-other*)
finally
show *?thesis* **unfolding** *True*.
next
case *False*
show *?thesis*
proof (*cases* $y \in domA\ \Gamma$)
case *True*
have $(\mathcal{N}[\![\ the\ (map\text{-}of\ \Gamma\ y)\ ]\!]_{\varrho})|_{r} = (\mathcal{N}[\![\ the\ (map\text{-}of\ \Gamma\ y)\ ]\!]_{\varrho|^{\circ}{}_{r}})|_{r}$
by (*rule C-restr-eq-lower*[*OF restr-can-restrict-env below-C*])
also have $\ldots \sqsubseteq \mathcal{N}[\![\ the\ (map\text{-}of\ \Gamma\ y)\ ]\!]_{\varrho|^{\circ}{}_{r}}$
by (*rule C-restr-below*)
also note ‹$\varrho|^{\circ}{}_{r} \sqsubseteq \mathcal{N}\{\!|delete\ x\ \Gamma|\!\}$›

```
          finally
          show ?thesis
            using ‹y ∈ domA Γ› ‹y ≠ x›
            by (simp add: lookupEvalHeap lookup-HSem-heap)
        next
          case False
          thus ?thesis by simp
        qed
      qed
    qed

    from ‹ϱ ⊑ ub›
    have (𝓝⟦ Γ ⟧ϱ) ⊑ (𝓝⟦ Γ ⟧ub)
      by (rule cont2monofunE[rotated]) simp
    also have … = ub
      unfolding ub-def HSem-bot-eq[symmetric]..
    finally
    show (𝓝⟦ Γ ⟧ϱ) ⊑ ub.
  qed simp-all

  from assms(2)
  have (𝒩⟦e⟧𝒩⦃Γ⦄)·(C·r) ≠ ⊥
    unfolding r
    by (rule demand-suffices[OF infinite-resources-suffice])
  also
  have (𝒩⟦e⟧𝒩⦃Γ⦄)·(C·r) = (𝒩⟦e⟧(𝒩⦃Γ⦄)|°r)·(C·r)
    by (rule can-restrict-env)
  also
  have … ⊑ (𝒩⟦e⟧𝒩⦃delete x Γ⦄)·(C·r)
    by (intro monofun-cfun-arg monofun-cfun-fun heaps )
  also
  have … ⊑ (𝒩⟦e⟧𝒩⦃delete x Γ⦄)·r'
    using ‹C·r ⊑ r'› by (rule monofun-cfun-arg)
  finally
  show ?thesis by this (intro cont2cont)+
qed
```

The semantics is continuous, so we can apply induction here:

```
lemma resourced-adequacy:
  assumes (𝒩⟦e⟧𝒩⦃Γ⦄)·r ≠ ⊥
  shows ∃ Δ v. Γ : e ⇓S Δ : v
using assms
proof(induction r arbitrary: Γ e S rule: C.induct[case-names adm bot step])
  case adm show ?case by simp
next
  case bot
  hence False by auto
  thus ?case..
```

next
  case (*step r*)
  show *?case*
  proof(*cases e rule*:*exp-strong-exhaust*(*1*)[**where** $c = (\Gamma,S)$, *case-names Var App Let Lam Bool IfThenElse*])
  case (*Var x*)
    let *?e* = *the* (*map-of* Γ *x*)
    from *step.prems*[*unfolded Var*]
    have $x \in$ *domA* Γ
      by (*auto intro*: *ccontr simp add*: *lookup-HSem-other*)
    hence *map-of* Γ *x* = *Some ?e* **by** (*rule domA-map-of-Some-the*)
    moreover
    from *step.prems*[*unfolded Var*] ‹*map-of* Γ *x* = *Some ?e*› ‹$x \in$ *domA* Γ›
    have $(\mathcal{N}[\![?e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$ **by** (*auto simp add*: *lookup-HSem-heap simp del*: *app-strict*)
    hence $(\mathcal{N}[\![?e]\!]_{\mathcal{N}\{\!|delete\ x\ \Gamma|\!\}})\cdot r \neq \bot$ **by** (*rule add-BH*[*OF* ‹*map-of* Γ *x* = *Some ?e*›])
    from *step.IH*[*OF this*]
    obtain Δ *v* **where** *delete x* Γ : *?e* $\Downarrow_{x\ \#\ S}$ Δ : *v* **by** *blast*
    ultimately
    have Γ : (*Var x*) $\Downarrow_S$ (*x*,*v*) # Δ : *v* **by** (*rule Variable*)
    thus *?thesis* **using** *Var* **by** *auto*
  next
  case (*App e′ x*)
    have *finite* (*set S* ∪ *fv* (Γ, *e′*)) **by** *simp*
    from *finite-list*[*OF this*]
    obtain *S′* **where** *S′*: *set S′* = *set S* ∪ *fv* (Γ, *e′*)**..**

    from *step.prems*[*unfolded App*]
    have *prem*: $((\mathcal{N}[\![\ e'\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \downarrow CFn\ (\mathcal{N}\{\!|\Gamma|\!\})\ x|_r)\cdot r \neq \bot$ **by** (*auto simp del*: *app-strict*)
    hence $(\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$ **by** *auto*
    from *step.IH*[*OF this*]
    obtain Δ *v* **where** *lhs′*: Γ : *e′* $\Downarrow_{S'}$ Δ : *v* **by** *blast*

    have *fv* (Γ, *e′*) ⊆ *set S′* **using** *S′* **by** *auto*
    from *correctness-empty-env*[*OF lhs′ this*]
    have *correct1*: $\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![v]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **and** $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$ **by** *auto*

    from *prem*
    have $((\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r \downarrow CFn\ (\mathcal{N}\{\!|\Gamma|\!\})\ x|_r)\cdot r \neq \bot$
      by (*rule not-bot-below-trans*)(*intro correct1 monofun-cfun-fun monofun-cfun-arg*)
    with *result-evaluated*[*OF lhs′*]
    have *isLam v* **by** (*cases r*, *auto*, *cases v rule*: *isVal.cases*, *auto*)
    then obtain *y e′′* **where** *n′*: *v* = (*Lam* [*y*]. *e′′*) **by** (*rule isLam-obtain-fresh*)
    with *lhs′*
    have *lhs*: Γ : *e′* $\Downarrow_{S'}$ Δ : *Lam* [*y*]. *e′′* **by** *simp*

    have $((\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r \downarrow CFn\ (\mathcal{N}\{\!|\Gamma|\!\})\ x|_r)\cdot r \neq \bot$ **by** *fact*
    also have $(\mathcal{N}\{\!|\Gamma|\!\})\ x|_r \sqsubseteq (\mathcal{N}\{\!|\Gamma|\!\})\ x$ **by** (*rule C-restr-below*)
    also note ‹*v* = -›

also note ‹$(\mathcal{N}\{\!|\Gamma|\!\}) \sqsubseteq (\mathcal{N}\{\!|\Delta|\!\})$›
also have $(\mathcal{N}[\![\ \mathit{Lam}\ [y].\ e''\ ]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r \sqsubseteq \mathit{CFn}\cdot(\Lambda\ v.\ \mathcal{N}[\![e']\!]_{(\mathcal{N}\{\!|\Delta|\!\})(y\ :=\ v)})$
by (*rule CELam-no-restr*)
also have $(\ldots \downarrow\mathit{CFn}\ (\mathcal{N}\{\!|\Delta|\!\})\ x)\cdot r = (\mathcal{N}[\![e']\!]_{(\mathcal{N}\{\!|\Delta|\!\})(y\ :=\ ((\mathcal{N}\{\!|\Delta|\!\})\ x))})\cdot r$ **by** *simp*
also have $\ldots = (\mathcal{N}[\![e''[y{::=}x]]\!]_{\mathcal{N}\{\!|\Delta|\!\}})\cdot r$
unfolding *ESem-subst*..
finally
have $\ldots \neq \bot$ **by** *this* (*intro cont2cont cont-fun*)+
then
obtain $\Theta\ v'$ **where** *rhs*: $\Delta : e''[y{::=}x] \Downarrow_{S'} \Theta : v'$ **using** *step.IH* **by** *blast*

have $\Gamma : \mathit{App}\ e'\ x \Downarrow_{S'} \Theta : v'$
by (*rule reds-ApplicationI*[*OF lhs rhs*])
hence $\Gamma : \mathit{App}\ e'\ x \Downarrow_{S} \Theta : v'$
apply (*rule reds-smaller-L*) **using** S' **by** *auto*
thus *?thesis* **using** *App* **by** *auto*
next
case (*Lam v e'*)
have $\Gamma : \mathit{Lam}\ [v].\ e' \Downarrow_{S} \Gamma : \mathit{Lam}\ [v].\ e'$ ..
thus *?thesis* **using** *Lam* **by** *blast*
next
case (*Bool b*)
have $\Gamma : \mathit{Bool}\ b \Downarrow_{S} \Gamma : \mathit{Bool}\ b$ **by** *rule*
thus *?thesis* **using** *Bool* **by** *blast*
next
case (*IfThenElse scrut* e_1 e_2)

from *step.prems*[*unfolded IfThenElse*]
have *prem*: $\mathit{CB\text{-}project}\cdot((\mathcal{N}[\![\ \mathit{scrut}\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r)\cdot((\mathcal{N}[\![\ e_1\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r)\cdot((\mathcal{N}[\![\ e_2\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r) \neq \bot$ **by** (*auto simp del*: *app-strict*)
then obtain b **where**
is-CB: $(\mathcal{N}[\![\ \mathit{scrut}\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r = \mathit{CB}\cdot(\mathit{Discr}\ b)$
and *not-bot2*: $((\mathcal{N}[\![\ (\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2)\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r) \neq \bot$
unfolding *CB-project-not-bot* **by** (*auto split*: *if-splits*)

have *finite* (*set* $S \cup \mathit{fv}\ (\Gamma,\ \mathit{scrut})$) **by** *simp*
from *finite-list*[*OF this*]
obtain S' **where** S': *set* S' = *set* $S \cup \mathit{fv}\ (\Gamma,\ \mathit{scrut})$..

from *is-CB* **have** $(\mathcal{N}[\![\ \mathit{scrut}\ ]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot r \neq \bot$ **by** *simp*
from *step.IH*[*OF this*]
obtain $\Delta\ v$ **where** *lhs'*: $\Gamma : \mathit{scrut} \Downarrow_{S'} \Delta : v$ **by** *blast*
then have *isVal v* **by** (*rule result-evaluated*)

have $\mathit{fv}\ (\Gamma,\ \mathit{scrut}) \subseteq \mathit{set}\ S'$ **using** S' **by** *simp*
from *correctness-empty-env*[*OF lhs' this*]
have *correct1*: $\mathcal{N}[\![\mathit{scrut}]\!]_{\mathcal{N}\{\!|\Gamma|\!\}} \sqsubseteq \mathcal{N}[\![v]\!]_{\mathcal{N}\{\!|\Delta|\!\}}$ **and** *correct2*: $\mathcal{N}\{\!|\Gamma|\!\} \sqsubseteq \mathcal{N}\{\!|\Delta|\!\}$ **by** *auto*

from *correct1*
have $(\mathcal{N}[\![\ scrut\ ]\!]_{\mathcal{N}\{\!\![\Gamma]\!\!\}})\cdot r \sqsubseteq (\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!\![\Delta]\!\!\}})\cdot r$ **by** (*rule monofun-cfun-fun*)
with *is-CB*
have $(\mathcal{N}[\![\ v\ ]\!]_{\mathcal{N}\{\!\![\Delta]\!\!\}})\cdot r = CB\cdot(Discr\ b)$ **by** *simp*
with ‹*isVal v*›
have $v = Bool\ b$ **by** (*cases v rule*: *isVal.cases*) (*case-tac r*, *auto*)+

from *not-bot2* ‹$\mathcal{N}\{\!\![\Gamma]\!\!\} \sqsubseteq \mathcal{N}\{\!\![\Delta]\!\!\}$›
have $(\mathcal{N}[\![\ (if\ b\ then\ e_1\ else\ e_2)\ ]\!]_{\mathcal{N}\{\!\![\Delta]\!\!\}})\cdot r \neq \bot$
by (*rule not-bot-below-trans*[*OF - monofun-cfun-fun*[*OF monofun-cfun-arg*]])
from *step.IH*[*OF this*]
obtain $\Theta\ v'$ **where** *rhs*: $\Delta : (if\ b\ then\ e_1\ else\ e_2) \Downarrow_{S'} \Theta : v'$ **by** *blast*

from *lhs'*[*unfolded* ‹$v = -$›] *rhs*
have $\Gamma : (scrut\ ?\ e_1 : e_2) \Downarrow_{S'} \Theta : v'$ **by** *rule*
hence $\Gamma : (scrut\ ?\ e_1 : e_2) \Downarrow_{S} \Theta : v'$
apply (*rule reds-smaller-L*) **using** S' **by** *auto*
thus *?thesis* **unfolding** *IfThenElse* **by** *blast*
next
case $(Let\ \Delta\ e')$
from *step.prems*[*unfolded Let*(*2*)]
have *prem*: $(\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!\![\Delta]\!\!\}\mathcal{N}\{\!\![\Gamma]\!\!\}})\cdot r \neq \bot$
by (*simp del*: *app-strict*)
also
have *atom ‘ domA* $\Delta\ \sharp*\ \Gamma$ **using** *Let*(*1*) **by** (*simp add*: *fresh-star-Pair*)
hence $\mathcal{N}\{\!\![\Delta]\!\!\}\mathcal{N}\{\!\![\Gamma]\!\!\} = \mathcal{N}\{\!\![\Delta\ @\ \Gamma]\!\!\}$ **by** (*rule HSem-merge*)
finally
have $(\mathcal{N}[\![e']\!]_{\mathcal{N}\{\!\![\Delta\ @\ \Gamma]\!\!\}})\cdot r \neq \bot$.
then
obtain $\Theta\ v$ **where** $\Delta\ @\ \Gamma : e' \Downarrow_S \Theta : v$ **using** *step.IH* **by** *blast*
hence $\Gamma : Let\ \Delta\ e' \Downarrow_S \Theta : v$
by (*rule reds.Let*[*OF Let*(*1*)])
thus *?thesis* **using** *Let* **by** *auto*
qed
qed

end

9.2 Adequacy

theory *Adequacy*
imports *ResourcedAdequacy Denotational−Related*
begin

theorem *adequacy*:
assumes $[\![e]\!]_{\{\!\![\Gamma]\!\!\}} \neq \bot$
shows $\exists\ \Delta\ v.\ \Gamma : e \Downarrow_S \Delta : v$

proof −
  have $\{\!|\Gamma|\!\} \triangleleft\triangleright^* \mathcal{N}\{\!|\Gamma|\!\}$ **by** (*rule heaps-similar*)
  hence $[\![e]\!]_{\{\!|\Gamma|\!\}} \triangleleft\triangleright (\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot C^{\infty}$ **by** (*rule denotational-semantics-similar*)
  from *bot-or-not-bot*[*OF this*] *assms*
  have $(\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!|\Gamma|\!\}})\cdot C^{\infty} \neq \bot$ **by** *metis*
  thus *?thesis* **by** (*rule resourced-adequacy*)
qed

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.