

The Correctness of Launchbury’s Natural Semantics for Lazy Evaluation

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

May 26, 2024

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 <i>assn</i>	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	18
2.4.1	Lemmas helping with equivariance proofs	18
2.4.2	Freshness via equivariance	19
2.4.3	Additional simplification rules	20
2.4.4	Additional equivariance lemmas	20
2.4.5	Freshness lemmas	22
2.4.6	Freshness and support for subsets of variables	22
2.4.7	The set of free variables of an expression	23
2.4.8	Other useful lemmas	24
2.5	AList-Utils-Nominal	25
2.5.1	Freshness lemmas related to associative lists	25
2.5.2	Equivariance lemmas	26
2.5.3	Freshness and distinctness	27
2.5.4	Pure codomains	27
2.6	HOLCF-Utils	27
2.6.1	Composition of fun and cfun	29
2.6.2	Additional transitivity rules	29
2.7	HOLCF-Meet	30
2.7.1	Towards meets: Lower bounds	30
2.7.2	Greatest lower bounds	30
2.8	Nominal-HOLCF	34
2.8.1	Type class of continuous permutations and variations thereof	34
2.8.2	Instance for <i>cfun</i>	35
2.8.3	Instance for <i>fun</i>	36
2.8.4	Instance for <i>u</i>	36
2.8.5	Instance for <i>lift</i>	37
2.8.6	Instance for <i>prod</i>	37
2.9	Env	37
2.9.1	The domain of a pcpo-valued function	38
2.9.2	Updates	38
2.9.3	Restriction	38
2.9.4	Deleting	40
2.9.5	Merging of two functions	42
2.9.6	Environments with binary joins	42
2.9.7	Singleton environments	43
2.10	Env-Nominal	44
2.10.1	Equivariance lemmas	44
2.10.2	Permutation and restriction	44
2.10.3	Pure codomains	45

2.11	Env-HOLCF	45
2.11.1	Continuity and pcpo-valued functions	45
2.12	EvalHeap	47
2.12.1	Conversion from heaps to environments	47
2.12.2	Reordering lemmas	49
3	Launchbury's natural semantics	50
3.1	Vars	50
3.2	Terms	50
3.2.1	Expressions	50
3.2.2	Rewriting in terms of heaps	51
3.2.3	Nice induction rules	54
3.2.4	Testing alpha equivalence	55
3.2.5	Free variables	55
3.2.6	Lemmas helping with nominal definitions	56
3.2.7	A smart constructor for lets	56
3.2.8	A predicate for value expressions	57
3.2.9	The notion of thunks	57
3.2.10	Non-recursive Let bindings	58
3.2.11	Renaming a lambda-bound variable	59
3.3	Substitution	59
3.4	Launchbury	62
3.4.1	The natural semantics	62
3.4.2	Example evaluations	63
3.4.3	Better introduction rules	63
3.4.4	Properties of the semantics	64
4	Denotational domain	65
4.1	Value	65
4.1.1	The semantic domain for values and environments	65
4.2	Value-Nominal	66
5	Denotational semantics	67
5.1	Iterative	67
5.2	HasESem	68
5.3	HeapSemantics	68
5.3.1	A locale for heap semantics, abstract in the expression semantics	68
5.3.2	Induction and other lemmas about <i>HSem</i>	69
5.3.3	Substitution	71
5.3.4	Re-calculating the semantics of the heap is idempotent	71
5.3.5	Iterative definition of the heap semantics	71
5.3.6	Fresh variables on the heap are irrelevant	72
5.3.7	Freshness	72
5.3.8	Adding a fresh variable to a heap does not affect its semantics	73

5.3.9	Mutual recursion with fresh variables	73
5.3.10	Parallel induction	73
5.3.11	Congruence rule	74
5.3.12	Equivariance of the heap semantics	74
5.4	AbstractDenotational	74
5.4.1	The denotational semantics for expressions	74
5.5	Abstract-Denotational-Props	75
5.5.1	The semantics ignores fresh variables	75
5.5.2	Nicer equations for ESem, without freshness requirements	75
5.5.3	Denotation of Substitution	75
5.6	Denotational	76
6	Resourced denotational domain	77
6.1	C	77
6.2	C-Meet	78
6.3	C-restr	79
6.3.1	The demand of a C-function	79
6.3.2	Restricting functions with domain C	80
6.3.3	Restricting maps of C-ranged functions	81
6.4	CValue	82
6.5	CValue-Nominal	83
6.6	ResourcedDenotational	83
7	Correctness of the natural semantics	85
7.1	CorrectnessOriginal	85
7.2	CorrectnessResourced	85
8	Equivalence of the denotational semantics	86
8.1	ValueSimilarity	86
8.1.1	A note about section 2.3	86
8.1.2	Working with Value and CValue	87
8.1.3	Restricted similarity is defined recursively	87
8.1.4	Moving up and down the similarity relations	89
8.1.5	Admissibility	89
8.1.6	The real similarity relation	89
8.1.7	The similarity relation lifted pointwise to functions.	91
8.2	Denotational-Related	92
9	Adequacy	93
9.1	ResourcedAdequacy	93
9.2	Adequacy	93

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 Launchbury’s 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.

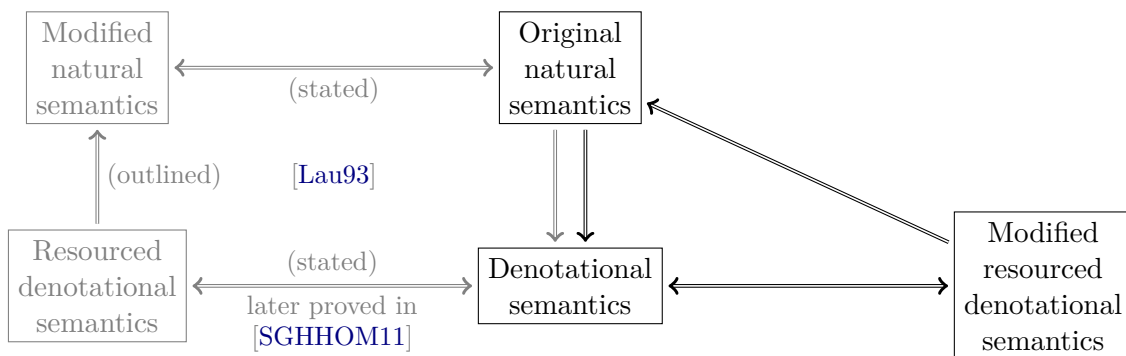
- We provide a new and more direct proof of adequacy.
- In order to do so, we formalize parts of [SGHHOM11], fixing a mistake in the proof.

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 $set::'a\ list \Rightarrow 'a\ set$) 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:

$e ::= \lambda x. e$	lambda abstraction
$ e x$	application
$ x$	variable
$ let\ as\ in\ e$	recursive let

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:

$$\begin{array}{c}
\frac{}{\Gamma : \lambda x. e \Downarrow_L \Gamma : \lambda x. e} \quad \text{LAMBDA} \\
\\
\frac{y \# (\Gamma, e, x, L, \Delta, \Theta, z) \quad \Gamma : e \Downarrow_L \Delta : \lambda y. e' \quad \Delta : e'[y := x] \Downarrow_L \Theta : z}{\Gamma : e x \Downarrow_L \Theta : z} \quad \text{APPLICATION} \\
\\
\frac{(x, e) \in \Gamma \quad \Gamma \setminus x : e \Downarrow_x . L \Delta : z}{\Gamma : x \Downarrow_L (x, z) \cdot \Delta : z} \quad \text{VARIABLE} \\
\\
\frac{dom \Delta \#* (\Gamma, L) \quad \Delta @ \Gamma : body \Downarrow_L \Theta : z}{\Gamma : let \Delta in body \Downarrow_L \Theta : z} \quad \text{LET}
\end{array}$$

1.1.4 The denotational semantics

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

$$D = [D \rightarrow D]_{\perp}$$

as introduced in [Abr90]. The type *Value*, together with the bottom value $\perp :: Value$, the injection $Fn :: (Value \rightarrow Value) \rightarrow Value$ and the projection $_ \Downarrow Fn _ :: Value \rightarrow Value \rightarrow Value$, 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 $\rho :: var \Rightarrow Value$ is written $\llbracket e \rrbracket_{\rho} :: Value$ and defined by the following equations:

$$\begin{aligned}
\llbracket \lambda x. e \rrbracket_{\rho} &= Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\rho(x := v)}) \\
\llbracket e x \rrbracket_{\rho} &= \llbracket e \rrbracket_{\rho} \Downarrow Fn \rho x \\
\llbracket x \rrbracket_{\rho} &= \rho x \\
\llbracket let \Gamma in body \rrbracket_{\rho} &= \llbracket body \rrbracket_{\llbracket \Gamma \rrbracket_{\rho}}
\end{aligned}$$

The expression $\llbracket \Gamma \rrbracket_{\varrho}$ maps the evaluation function over a heap, returning an environment:

$$\begin{aligned} (\llbracket \Gamma \rrbracket_{\varrho}) v &= \llbracket e \rrbracket_{\varrho} && \text{if } (v, e) \in \Gamma \\ (\llbracket \Gamma \rrbracket_{\varrho}) v &= \perp && \text{if } v \notin \text{dom } \Gamma \end{aligned}$$

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

$$\{\Gamma\}_{\varrho} = \varrho \text{ ++}_{\text{dom } \Gamma} \llbracket \Gamma \rrbracket_{\{\Gamma\}_{\varrho}}$$

where

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

The semantics of the heap in the empty environment \perp 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, $fv(\Gamma, e) \subseteq L \cup \text{dom } \Gamma$ holds, then

$$\llbracket e \rrbracket_{\{\Gamma\}_{\varrho}} = \llbracket v \rrbracket_{\{\Delta\}_{\varrho}}.$$

The statement of adequacy reads:

$$\text{If } \llbracket e \rrbracket_{\{\Gamma\}} \neq \perp \text{ then } \exists \Delta v. \Gamma : e \Downarrow_S \Delta : v.$$

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 solely work with right sided updates. We also avoid the ambiguous syntax \sqcup and write $_ ++ _ _$ instead (the index indicates on what set the function on the right overrides the function on the left). The alternative operational semantics is not included in this work.

1.2.2 The types of environments

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

$$\llbracket \lambda x. e \rrbracket_{\varrho} = Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho(x := v)})$$

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 ($x \# \varrho$), 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 $x \# \varrho$). More precisely: The type of the argument must be a member of the *fs* typeclass provided by the Nominal package. The type $var \Rightarrow Value$ cannot be made a member of this class, as there obviously are elements that have infinite support. The fix here was to introduce a new type constructor, *fmap*, for partial functions with finite domain. This is fine: Only functions with finite domain matter in our formalisation.

The introduction of *fmap* had further consequences. The main type class of the HOLCF package, which we use to define domains and continuous functions on them, is the class *cpo*, of chain-complete partial orders. With the usual ordering on partial functions, $(var, Value)$ *fmap* cannot be a member of this class. The fix here is to use a different ordering and only let elements be comparable that have the same domain. In our formalisation, the domain is always 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

$$\llbracket \lambda x. e \rrbracket_{\varrho} = Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho(x := v)})$$

holds unconditionally. The actual, technical definition is

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

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 $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho|_{fv} e}$ can be proved. With that, the desired equation for $\llbracket \lambda x. e \rrbracket_{\varrho}$ follows. The same trick is applied to the equation for $\llbracket \text{let } \Gamma \text{ in body} \rrbracket_{\varrho}$.

This allows us to use the type $var \Rightarrow Value$ for the semantic environments 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 $LetA::assn \Rightarrow exp \Rightarrow exp$. Afterwards we define conversions functions and the desired constructor $Let::(var \times exp) list \Rightarrow exp \Rightarrow exp$, 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 upper 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 $_ \triangleleft _$ between the standard semantic domain (*Value*) and the resourced domain (*CValue*) and show that the denotation semantics yield similar results ($\varrho \triangleleft^* \sigma \implies \llbracket e \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket e \rrbracket_\sigma) \cdot C^\infty$), 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.

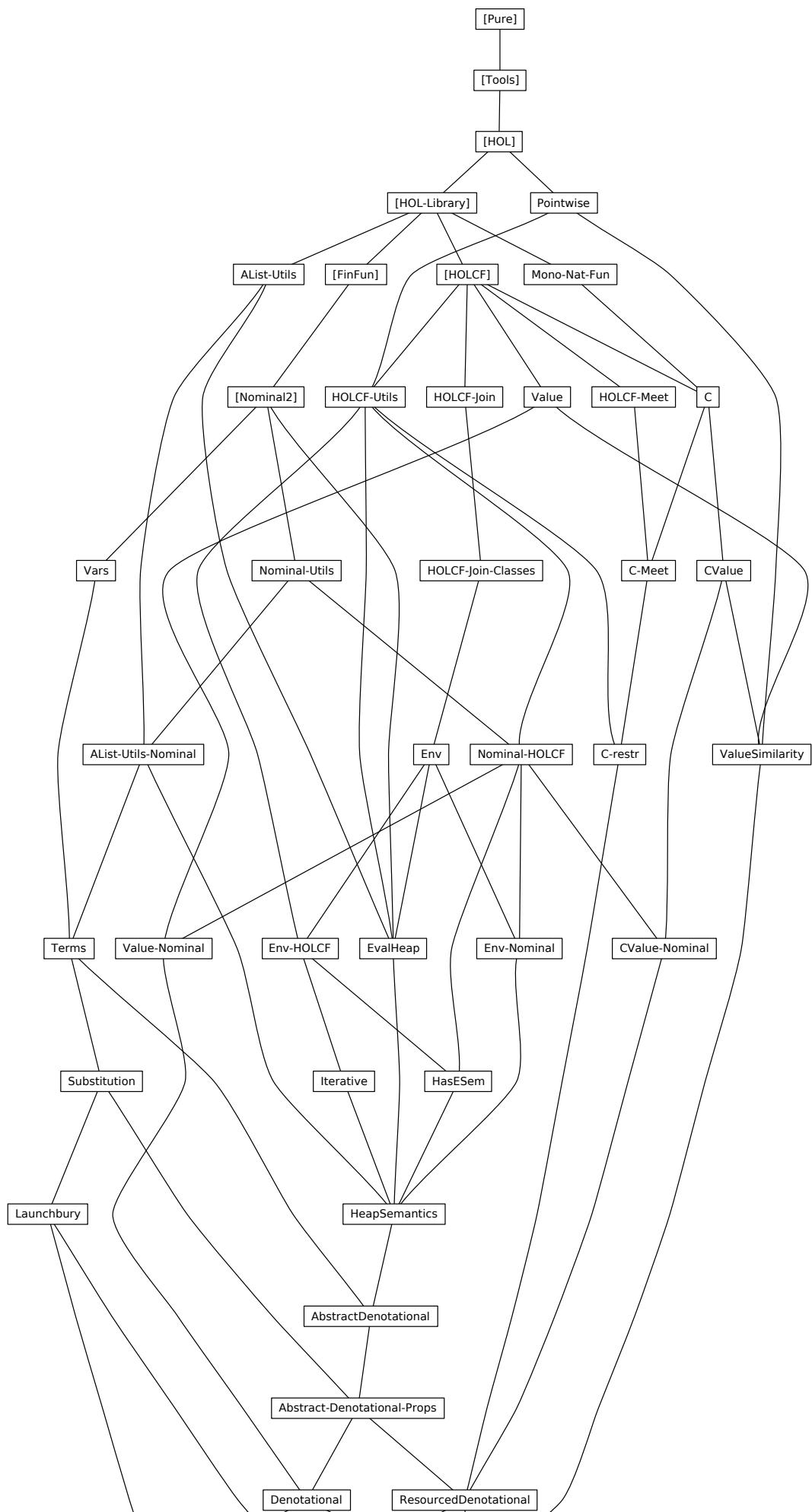
This work proves the adequacy as stated by Launchbury as well, but in contrast to his proof outline no alternative operational semantics is introduced. The problems of indirection vs. substitution and of blackholing is solved on the denotational side instead, which turned out to be much easier than proving 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 $\mathbb{N} \rightarrow \mathbb{N}$ (*Mono-Nat-Fun*).
- General utility functions extending Nominal (*Nominal-Utils*).



- 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]*: $(\bigwedge\ x.\ P\ (m\ x)\ (m'\ x)) \implies\ \textit{pointwise}\ P\ m\ m'$ *<proof>*

end

2.2 AList-Utills

theory *AList-Utills*

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* $\equiv\ \textit{AList.delete}$

abbreviation *update* **where** *update* $\equiv\ \textit{AList.update}$

abbreviation *restrictA* **where** *restrictA* $\equiv\ \textit{AList.restrict}$

abbreviation *clearjunk* **where** *clearjunk* $\equiv\ \textit{AList.clearjunk}$

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

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

lemma *restrictA-append*: $\textit{restrictA}\ S\ (a@b) = \textit{restrictA}\ S\ a\ @\ \textit{restrictA}\ S\ b$
<proof>

lemma *length-restrictA-le*: $\textit{length}\ (\textit{restrictA}\ S\ a) \leq \textit{length}\ a$
<proof>

2.2.1 The domain of an associative list

definition *domA*

where *domA* $h = \textit{fst}\ 'set\ h$

lemma *domA-append[simp]*: $\textit{domA}\ (a\ @\ b) = \textit{domA}\ a\ \cup\ \textit{domA}\ b$

and [*simp*]: $\textit{domA}\ ((v,e)\ \#\ h) = \textit{insert}\ v\ (\textit{domA}\ h)$

and [*simp*]: $\textit{domA}\ (p\ \#\ h) = \textit{insert}\ (\textit{fst}\ p)\ (\textit{domA}\ h)$

and [*simp*]: $\textit{domA}\ [] = \{\}$

<proof>

lemma *domA-from-set*:

$(x, e) \in \text{set } h \implies x \in \text{dom}A \ h$
 $\langle \text{proof} \rangle$

lemma *finite-domA[simp]*:
 $\text{finite } (\text{dom}A \ \Gamma)$
 $\langle \text{proof} \rangle$

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

lemma *domA-restrictA[simp]*:
 $\text{dom}A \ (\text{restrict}A \ S \ \Gamma) = \text{dom}A \ \Gamma \cap S$
 $\langle \text{proof} \rangle$

lemma *delete-not-domA[simp]*:
 $x \notin \text{dom}A \ \Gamma \implies \text{delete } x \ \Gamma = \Gamma$
 $\langle \text{proof} \rangle$

lemma *deleted-not-domA*: $x \notin \text{dom}A \ (\text{delete } x \ \Gamma)$
 $\langle \text{proof} \rangle$

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

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

lemma *domA-clearjunk[simp]*: $\text{dom}A \ (\text{clearjunk } \Gamma) = \text{dom}A \ \Gamma$
 $\langle \text{proof} \rangle$

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

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

lemma *restrictA-noop*: $\text{dom}A \ \Gamma \subseteq S \implies \text{restrict}A \ S \ \Gamma = \Gamma$
 $\langle \text{proof} \rangle$

lemma *restrictA-cong*:
 $(\bigwedge x. x \in \text{dom}A \ m1 \implies x \in V \longleftrightarrow x \in V') \implies m1 = m2 \implies \text{restrict}A \ V \ m1 = \text{restrict}A$
 $V' \ m2$
 $\langle \text{proof} \rangle$

2.2.2 Other lemmas about associative lists

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

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

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

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

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

2.2.3 Syntax for map comprehensions

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

syntax

$-MapCollect :: 'c \Rightarrow ptttrn \Rightarrow ptttrn \Rightarrow 'a \rightarrow 'b \Rightarrow 'c \text{ set} \quad ((1\{- \mid - / \mapsto - / \in - / \})$

translations

$\{e \mid k \mapsto v \in m\} == CONST mapCollect (\lambda k v . e) m$

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

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

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

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

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

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

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

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

definition $mapCollectFilter :: ('a \Rightarrow 'b \Rightarrow (bool \times 'c)) \Rightarrow ('a \rightarrow 'b) \Rightarrow 'c \text{ set}$

where $\text{mapCollectFilter } f \ m = \{ \text{snd } (f \ k \ v) \mid k \ v . \ m \ k = \text{Some } v \wedge \text{fst } (f \ k \ v) \}$

syntax

$\text{-MapCollectFilter} :: 'c \Rightarrow \text{pttrn} \Rightarrow \text{pttrn} \Rightarrow ('a \rightarrow 'b) \Rightarrow \text{bool} \Rightarrow 'c \ \text{set} \quad ((1 \{ - \mid / - / \mapsto / - / \in / - / . / - \}))$

translations

$\{ e \mid k \mapsto v \in m . P \} == \text{CONST } \text{mapCollectFilter } (\lambda k \ v . (P, e)) \ m$

lemma $\text{mapCollectFilter-const-False[simp]}$:

$\{ e \mid k \mapsto v \in m . \text{False} \} = \{ \}$
 $\langle \text{proof} \rangle$

lemma $\text{mapCollectFilter-const-True[simp]}$:

$\{ e \mid k \mapsto v \in m . \text{True} \} = \{ e \mid k \mapsto v \in m \}$
 $\langle \text{proof} \rangle$

end

2.3 Mono-Nat-Fun

theory Mono-Nat-Fun

imports $\text{HOL-Library.Infinite-Set}$

begin

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

lemma $\text{nat-mono-characterization}$:

fixes $f :: \text{nat} \Rightarrow \text{nat}$

assumes $\text{mono } f$

obtains $n \ \text{where } \bigwedge m . n \leq m \implies f \ n = f \ m \mid \bigwedge m . \exists n . m \leq f \ n$

$\langle \text{proof} \rangle$

end

2.4 Nominal-Utills

theory Nominal-Utills

imports $\text{Nominal2.Nominal2 } \text{HOL-Library.AList}$

begin

2.4.1 Lemmas helping with equivariance proofs

lemma perm-rel-lemma :

assumes $\bigwedge \pi \ x \ y . r \ (\pi \cdot x) \ (\pi \cdot y) \implies r \ x \ y$

shows $r \ (\pi \cdot x) \ (\pi \cdot y) \longleftrightarrow r \ x \ y \ (\text{is } ?l \longleftrightarrow ?r)$

$\langle \text{proof} \rangle$

lemma *perm-rel-lemma2*:

assumes $\bigwedge \pi x y. r x y \implies r (\pi \cdot x) (\pi \cdot y)$
shows $r x y \iff r (\pi \cdot x) (\pi \cdot y)$ (**is** ?l \iff ?r)
<proof>

lemma *fun-evtI*:

assumes *f-evt*[*evt*]: $(\bigwedge p x. p \cdot (f x) = f (p \cdot x))$
shows $p \cdot f = f$ *<proof>*

lemma *evt-at-apply*:

assumes *evt-at* *f* *x*
shows $(p \cdot f) x = f x$
<proof>

lemma *evt-at-apply'*:

assumes *evt-at* *f* *x*
shows $p \cdot f x = f (p \cdot x)$
<proof>

lemma *evt-at-apply''*:

assumes *evt-at* *f* *x*
shows $(p \cdot f) (p \cdot x) = f (p \cdot x)$
<proof>

lemma *size-list-evt*[*evt*]: $p \cdot \text{size-list } f x = \text{size-list } (p \cdot f) (p \cdot x)$

<proof>

2.4.2 Freshness via equivariance

lemma *evt-fresh-cong1*: $(\bigwedge p x. p \cdot (f x) = f (p \cdot x)) \implies a \# x \implies a \# f x$

<proof>

lemma *evt-fresh-cong2*:

assumes *evt*: $(\bigwedge p x y. p \cdot (f x y) = f (p \cdot x) (p \cdot y))$
and *fresh1*: $a \# x$ **and** *fresh2*: $a \# y$
shows $a \# f x y$

<proof>

lemma *evt-fresh-star-cong1*:

assumes *evt*: $(\bigwedge p x. p \cdot (f x) = f (p \cdot x))$
and *fresh1*: $a \#* x$
shows $a \#* f x$

<proof>

lemma *evt-fresh-star-cong2*:

assumes *evt*: $(\bigwedge p x y. p \cdot (f x y) = f (p \cdot x) (p \cdot y))$
and *fresh1*: $a \#* x$ **and** *fresh2*: $a \#* y$

shows $a \#* f x y$
<proof>

lemma *eqvt-fresh-cong3*:

assumes *eqvt*: $(\bigwedge p x y z. p \cdot (f x y z) = f (p \cdot x) (p \cdot y) (p \cdot z))$
and *fresh1*: $a \# x$ **and** *fresh2*: $a \# y$ **and** *fresh3*: $a \# z$
shows $a \# f x y z$
<proof>

lemma *eqvt-fresh-star-cong3*:

assumes *eqvt*: $(\bigwedge p x y z. p \cdot (f x y z) = f (p \cdot x) (p \cdot y) (p \cdot z))$
and *fresh1*: $a \#* x$ **and** *fresh2*: $a \#* y$ **and** *fresh3*: $a \#* z$
shows $a \#* f x y z$
<proof>

2.4.3 Additional simplification rules

lemma *not-self-fresh[simp]*: $\text{atom } x \# x \longleftrightarrow \text{False}$
<proof>

lemma *fresh-star-singleton*: $\{ x \} \#* e \longleftrightarrow x \# e$
<proof>

2.4.4 Additional equivariance lemmas

lemma *eqvt-cases*:

fixes $f x \pi$
assumes *eqvt*: $\bigwedge x. \pi \cdot f x = f (\pi \cdot x)$
obtains $f x f (\pi \cdot x) \mid \neg f x \quad \neg f (\pi \cdot x)$
<proof>

lemma *range-eqvt*: $\pi \cdot \text{range } Y = \text{range } (\pi \cdot Y)$
<proof>

lemma *case-option-eqvt[eqvt]*:

$\pi \cdot \text{case-option } d f x = \text{case-option } (\pi \cdot d) (\pi \cdot f) (\pi \cdot x)$
<proof>

lemma *supp-option-eqvt*:

$\text{supp } (\text{case-option } d f x) \subseteq \text{supp } d \cup \text{supp } f \cup \text{supp } x$
<proof>

lemma *funpow-eqvt[simp,eqvt]*:

$\pi \cdot ((f :: 'a \Rightarrow 'a::pt) \overset{\sim}{\sim} n) = (\pi \cdot f) \overset{\sim}{\sim} (\pi \cdot n)$
<proof>

lemma *delete-eqvt[eqvt]*:

$\pi \cdot \text{AList.delete } x \Gamma = \text{AList.delete } (\pi \cdot x) (\pi \cdot \Gamma)$
<proof>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *override-on-eqv*[*eqvt*]:

$\pi \cdot (\text{override-on } m1 \ m2 \ S) = \text{override-on } (\pi \cdot m1) \ (\pi \cdot m2) \ (\pi \cdot S)$
<proof>

lemma *card-eqv*[*eqvt*]:

$\pi \cdot (\text{card } S) = \text{card } (\pi \cdot S)$
<proof>

lemma *Projl-permute*:

assumes $a: \exists y. f = \text{Inl } y$

shows $(p \cdot (\text{Sum-Type.proj1 } f)) = \text{Sum-Type.proj1 } (p \cdot f)$

<proof>

lemma *Projr-permute*:

assumes $a: \exists y. f = \text{Inr } y$

shows $(p \cdot (\text{Sum-Type.proj2 } f)) = \text{Sum-Type.proj2 } (p \cdot f)$

<proof>

2.4.5 Freshness lemmas

lemma *fresh-list-elem*:

assumes $a \# \Gamma$

and $e \in \text{set } \Gamma$

shows $a \# e$

<proof>

lemma *set-not-fresh*:

$x \in \text{set } L \implies \neg(\text{atom } x \# L)$

<proof>

lemma *pure-fresh-star*[*simp*]: $a \#* (x :: 'a :: \text{pure})$

<proof>

lemma *supp-set-mem*: $x \in \text{set } L \implies \text{supp } x \subseteq \text{supp } L$

<proof>

lemma *set-supp-mono*: $\text{set } L \subseteq \text{set } L2 \implies \text{supp } L \subseteq \text{supp } L2$

<proof>

lemma *fresh-star-at-base*:

fixes $x :: 'a :: \text{at-base}$

shows $S \#* x \longleftrightarrow \text{atom } x \notin S$

<proof>

2.4.6 Freshness and support for subsets of variables

lemma *supp-mono*: $\text{finite } (B :: 'a :: \text{fs set}) \implies A \subseteq B \implies \text{supp } A \subseteq \text{supp } B$

<proof>

lemma *fresh-subset*:

$finite\ B \implies x \# (B :: 'a::at-base\ set) \implies A \subseteq B \implies x \# A$
<proof>

lemma *fresh-star-subset*:

$finite\ B \implies x \#* (B :: 'a::at-base\ set) \implies A \subseteq B \implies x \#* A$
<proof>

lemma *fresh-star-set-subset*:

$x \#* (B :: 'a::at-base\ list) \implies set\ A \subseteq set\ B \implies x \#* A$
<proof>

2.4.7 The set of free variables of an expression

definition $fv :: 'a::pt \Rightarrow 'b::at-base\ set$

where $fv\ e = \{v.\ atom\ v \in\ supp\ e\}$

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

<proof>

lemma *fv-Nil[simp]*: $fv\ [] = \{\}$

<proof>

lemma *fv-Cons[simp]*: $fv\ (x \# xs) = fv\ x \cup fv\ xs$

<proof>

lemma *fv-Pair[simp]*: $fv\ (x, y) = fv\ x \cup fv\ y$

<proof>

lemma *fv-append[simp]*: $fv\ (x @ y) = fv\ x \cup fv\ y$

<proof>

lemma *fv-at-base[simp]*: $fv\ a = \{a::'a::at-base\}$

<proof>

lemma *fv-pure[simp]*: $fv\ (a::'a::pure) = \{\}$

<proof>

lemma *fv-set-at-base[simp]*: $fv\ (l :: ('a :: at-base)\ list) = set\ l$

<proof>

lemma *flip-not-fv*: $a \notin fv\ x \implies b \notin fv\ x \implies (a \leftrightarrow b) \cdot x = x$

<proof>

lemma *fv-not-fresh*: $atom\ x \# e \longleftrightarrow x \notin fv\ e$

<proof>

lemma *fresh-fv*: $finite\ (fv\ e :: 'a\ set) \implies atom\ (x :: ('a::at-base)) \# (fv\ e :: 'a\ set) \longleftrightarrow atom\ x$

$\# e$

<proof>

lemma *finite-fv[simp]*: $finite\ (fv\ (e::'a::fs) :: ('b::at-base)\ set)$

<proof>

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

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

lemma $fresh\text{-fv}\text{-list}[simp]:$
 $a \# (fv\text{-list } e :: 'b::at\text{-base list}) \longleftrightarrow a \# (fv\ e :: 'b::at\text{-base set})$
 $\langle proof \rangle$

2.4.8 Other useful lemmas

lemma $pure\text{-permute}\text{-id}: permute\ p = (\lambda\ x.\ (x::'a::pure))$
 $\langle proof \rangle$

lemma $supp\text{-set}\text{-elem}\text{-finite}:$
assumes $finite\ S$
and $(m::'a::fs) \in S$
and $y \in supp\ m$
shows $y \in supp\ S$
 $\langle proof \rangle$

lemmas $fresh\text{-star}\text{-Cons} = fresh\text{-star}\text{-list}(2)$

lemma $mem\text{-permute}\text{-set}:$
shows $x \in p \cdot S \longleftrightarrow (-\ p \cdot x) \in S$
 $\langle proof \rangle$

lemma $flip\text{-set}\text{-both}\text{-not}\text{-in}:$
assumes $x \notin S$ **and** $x' \notin S$
shows $((x' \leftrightarrow x) \cdot S) = S$
 $\langle proof \rangle$

lemma $inj\text{-atom}: inj\ atom\ \langle proof \rangle$

lemmas $image\text{-Int}[OF\ inj\text{-atom}, simp]$

lemma $eqvt\text{-uncurry}: eqvt\ f \implies eqvt\ (case\text{-prod } f)$
 $\langle proof \rangle$

lemma $supp\text{-fun}\text{-app}\text{-eqvt}2:$
assumes $a: eqvt\ f$
shows $supp\ (f\ x\ y) \subseteq supp\ x \cup supp\ y$
 $\langle proof \rangle$

lemma $supp\text{-fun}\text{-app}\text{-eqvt}3:$
assumes $a: eqvt\ f$
shows $supp\ (f\ x\ y\ z) \subseteq supp\ x \cup supp\ y \cup supp\ z$

<proof>

lemma *permute-0[simp]*: *permute 0 = (λ x. x)*

<proof>

lemma *permute-comp[simp]*: *permute x ∘ permute y = permute (x + y)* *<proof>*

lemma *map-permute*: *map (permute p) = permute p*

<proof>

lemma *fresh-star-restrictA[intro]*: *a #* Γ ⇒ a #* AList.restrict V Γ*

<proof>

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

<proof>

lemma *Abs-lst-Nil-eq2[simp]*: *[xs]lst. (x::'a::fs) = [[[]]lst. x' ↔ ((xs,x) = ([], x'))*

<proof>

end

2.5 AList-Utills-Nominal

theory *AList-Utills-Nominal*

imports *AList-Utills Nominal-Utills*

begin

2.5.1 Freshness lemmas related to associative lists

lemma *domA-not-fresh*:

x ∈ domA Γ ⇒ ¬(atom x # Γ)

<proof>

lemma *fresh-delete*:

assumes *a # Γ*

shows *a # delete v Γ*

<proof>

lemma *fresh-star-delete*:

assumes *S #* Γ*

shows *S #* delete v Γ*

<proof>

lemma *fv-delete-subset*:
 $fv (delete\ v\ \Gamma) \subseteq fv\ \Gamma$
 $\langle proof \rangle$

lemma *fresh-heap-expr*:
assumes $a \# \Gamma$
and $(x, e) \in set\ \Gamma$
shows $a \# e$
 $\langle proof \rangle$

lemma *fresh-heap-expr'*:
assumes $a \# \Gamma$
and $e \in snd\ 'set\ \Gamma$
shows $a \# e$
 $\langle proof \rangle$

lemma *fresh-star-heap-expr'*:
assumes $S \#* \Gamma$
and $e \in snd\ 'set\ \Gamma$
shows $S \#* e$
 $\langle proof \rangle$

lemma *fresh-map-of*:
assumes $x \in domA\ \Gamma$
assumes $a \# \Gamma$
shows $a \# the\ (map-of\ \Gamma\ x)$
 $\langle proof \rangle$

lemma *fresh-star-map-of*:
assumes $x \in domA\ \Gamma$
assumes $a \#* \Gamma$
shows $a \#* the\ (map-of\ \Gamma\ x)$
 $\langle proof \rangle$

lemma *domA-fv-subset*: $domA\ \Gamma \subseteq fv\ \Gamma$
 $\langle proof \rangle$

lemma *map-of-fv-subset*: $x \in domA\ \Gamma \implies fv\ (the\ (map-of\ \Gamma\ x)) \subseteq fv\ \Gamma$
 $\langle proof \rangle$

lemma *map-of-Some-fv-subset*: $map-of\ \Gamma\ x = Some\ e \implies fv\ e \subseteq fv\ \Gamma$
 $\langle proof \rangle$

2.5.2 Equivariance lemmas

lemma *domA[eqvt]*:
 $\pi \cdot domA\ \Gamma = domA\ (\pi \cdot \Gamma)$
 $\langle proof \rangle$

lemma *mapCollect[eqvt]*:
 $\pi \cdot \text{mapCollect } f \ m = \text{mapCollect } (\pi \cdot f) (\pi \cdot m)$
 ⟨*proof*⟩

2.5.3 Freshness and distinctness

lemma *fresh-distinct*:
assumes *atom* ' $S \ \#\ * \ \Gamma$
shows $S \cap \text{dom} A \ \Gamma = \{\}$
 ⟨*proof*⟩

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

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

2.5.4 Pure codomains

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

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

end

2.6 HOLCF-Utills

theory *HOLCF-Utills*
imports *HOLCF Pointwise*
begin

default-sort *type*

lemmas *cont-fun[simp]*
lemmas *cont2cont-fun[simp]*

lemma *cont-compose2*:
assumes $\bigwedge y. \text{cont } (\lambda x. c \ x \ y)$
assumes $\bigwedge x. \text{cont } (\lambda y. c \ x \ y)$

assumes *cont f*
assumes *cont g*
shows *cont* ($\lambda x. c (f x) (g x)$)
<proof>

lemma *pointwise-adm*:
fixes $P :: 'a::pcpo \Rightarrow 'b::pcpo \Rightarrow bool$
assumes *adm* ($\lambda x. P (fst x) (snd x)$)
shows *adm* ($\lambda m. pointwise P (fst m) (snd m)$)
<proof>

lemma *cfun-beta-Pair*:
assumes *cont* ($\lambda p. f (fst p) (snd p)$)
shows *csplit*.($\Lambda a b . f a b$).(x, y) = $f x y$
<proof>

lemma *fun-upd-mono*:
 $\rho1 \sqsubseteq \rho2 \implies v1 \sqsubseteq v2 \implies \rho1(x := v1) \sqsubseteq \rho2(x := v2)$
<proof>

lemma *fun-upd-cont[simp,cont2cont]*:
assumes *cont f* **and** *cont h*
shows *cont* ($\lambda x. (f x)(v := h x) :: 'a \Rightarrow 'b::pcpo$)
<proof>

lemma *fun-upd-belowI*:
assumes $\bigwedge z. z \neq x \implies \rho z \sqsubseteq \rho' z$
assumes $y \sqsubseteq \rho' x$
shows $\rho(x := y) \sqsubseteq \rho'$
<proof>

lemma *cont-if-else-above*:
assumes *cont f*
assumes *cont g*
assumes $\bigwedge x. f x \sqsubseteq g x$
assumes $\bigwedge x y. x \sqsubseteq y \implies P y \implies P x$
assumes *adm P*
shows *cont* ($\lambda x. if P x then f x else g x$) (**is cont ?I**)
<proof>

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

lemma *up2option-simps[simp]*:
up2option \perp = *None*
up2option (*up*. x) = *Some x*

$\langle \text{proof} \rangle$

fun *option2up* :: 'a option \Rightarrow 'a::cpo \perp
 where *option2up* None = \perp
 | *option2up* (Some a) = *up*·a

lemma *option2up-up2option[simp]*:
 option2up (*up2option* x) = x
 $\langle \text{proof} \rangle$

lemma *up2option-option2up[simp]*:
 up2option (*option2up* x) = x
 $\langle \text{proof} \rangle$

lemma *adm-subst2*: *cont* f \Longrightarrow *cont* g \Longrightarrow *adm* ($\lambda x. f$ (fst x) = g (snd x))
 $\langle \text{proof} \rangle$

2.6.1 Composition of fun and cfun

lemma *cont2cont-comp [simp, cont2cont]*:
 assumes *cont* f
 assumes $\bigwedge x. \text{cont } (f x)$
 assumes *cont* g
 shows *cont* ($\lambda x. (f x) \circ (g x)$)
 $\langle \text{proof} \rangle$

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

lemma [*simp*]: *cfun-comp*·f·($\varrho(x := v)$) = (*cfun-comp*·f· ϱ)($x := f \cdot v$)
 $\langle \text{proof} \rangle$

lemma *cfun-comp-app[simp]*: (*cfun-comp*·f· ϱ) x = f·(ϱx)
 $\langle \text{proof} \rangle$

lemma *fix-eq-fix*:
 f·(fix·g) \sqsubseteq fix·g \Longrightarrow g·(fix·f) \sqsubseteq fix·f \Longrightarrow fix·f = fix·g
 $\langle \text{proof} \rangle$

2.6.2 Additional transitivity rules

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

lemma *below-trans-cong[trans]*:
 a \sqsubseteq f x \Longrightarrow x \sqsubseteq y \Longrightarrow *cont* f \Longrightarrow a \sqsubseteq f y
 $\langle \text{proof} \rangle$

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

<proof>

lemma *not-bot-below-trans-cong*[*trans*]:
 $f a \neq \perp \implies a \sqsubseteq b \implies \text{cont } f \implies f b \neq \perp$
<proof>

end

2.7 HOLCF-Meet

theory *HOLCF-Meet*
imports *HOLCF*
begin

This theory defines the \sqcap 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 \Rightarrow 'a \Rightarrow bool (**infix** $>|$ 55) **where**
 $S >| x \longleftrightarrow (\forall y \in S. x \sqsubseteq y)$

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

lemma *is-lbD*: $[|S >| l; x \in S|] \implies l \sqsubseteq x$
<proof>

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

lemma *is-lb-insert* [*simp*]: $(\text{insert } x A) >| y = (y \sqsubseteq x \wedge A >| y)$
<proof>

lemma *is-lb-downward*: $[|S >| l; y \sqsubseteq l|] \implies S >| y$
<proof>

2.7.2 Greatest lower bounds

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

definition *glb* :: 'a set \Rightarrow 'a (\sqcap -[60]60) **where**
 $\text{glb } S = (\text{THE } x. S >>| x)$

Access to the definition as inference rule

lemma *is-glbD1*: $S \gg| x \implies S >| x$
<proof>

lemma *is-glbD2*: $[|S \gg| x; S >| u|] \implies u \sqsubseteq x$
<proof>

lemma (in *po*) *is-glbI*: $[|S >| x; \forall u. S >| u \implies u \sqsubseteq x|] \implies S \gg| x$
<proof>

lemma *is-glb-above-iff*: $S \gg| x \implies u \sqsubseteq x \longleftrightarrow S >| u$
<proof>

glbs are unique

lemma *is-glb-unique*: $[|S \gg| x; S \gg| y|] \implies x = y$
<proof>

technical lemmas about *glb* and ($\gg|$)

lemma *is-glb-glb*: $M \gg| x \implies M \gg| \text{glb } M$
<proof>

lemma *glb-eqI*: $M \gg| l \implies \text{glb } M = l$
<proof>

lemma *is-glb-singleton*: $\{x\} \gg| x$
<proof>

lemma *glb-singleton [simp]*: $\text{glb } \{x\} = x$
<proof>

lemma *is-glb-bin*: $x \sqsubseteq y \implies \{x, y\} \gg| x$
<proof>

lemma *glb-bin*: $x \sqsubseteq y \implies \text{glb } \{x, y\} = x$
<proof>

lemma *is-glb-maximal*: $[|S >| x; x \in S|] \implies S \gg| x$
<proof>

lemma *glb-maximal*: $[|S >| x; x \in S|] \implies \text{glb } S = x$
<proof>

lemma *glb-above*: $S \gg| z \implies x \sqsubseteq \text{glb } S \longleftrightarrow S >| x$
<proof>

end

lemma (in *cpo*) *Meet-insert*: $S \gg| l \implies \{x, l\} \gg| l2 \implies \text{insert } x S \gg| l2$
<proof>

Binary, hence finite meets.

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

```
  lemma binary-meet-exists':  $\exists l. \{x, y\} \gg | l$ 
    <proof>
```

```
  lemma finite-meet-exists:
    assumes  $S \neq \{\}$ 
    and finite S
    shows  $\exists x. S \gg | x$ 
  <proof>
```

```
end
```

```
definition meet :: 'a::cpo  $\Rightarrow$  'a  $\Rightarrow$  'a (infix  $\sqcap$  80) where
   $x \sqcap y = (\text{if } \exists z. \{x, y\} \gg | z \text{ then } \text{glb } \{x, y\} \text{ else } x)$ 
```

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

```
lemma meet-comm:  $(x::'a::Finite-Meet-cpo) \sqcap y = y \sqcap x$  <proof>
```

```
lemma meet-bot1[simp]:
  fixes  $y :: 'a :: \{Finite-Meet-cpo, pcpo\}$ 
  shows  $(\perp \sqcap y) = \perp$  <proof>
```

```
lemma meet-bot2[simp]:
  fixes  $x :: 'a :: \{Finite-Meet-cpo, pcpo\}$ 
  shows  $(x \sqcap \perp) = \perp$  <proof>
```

```
lemma meet-below1[intro]:
  fixes  $x y :: 'a :: Finite-Meet-cpo$ 
  assumes  $x \sqsubseteq z$ 
  shows  $(x \sqcap y) \sqsubseteq z$  <proof>
```

```
lemma meet-below2[intro]:
  fixes  $x y :: 'a :: Finite-Meet-cpo$ 
  assumes  $y \sqsubseteq z$ 
  shows  $(x \sqcap y) \sqsubseteq z$  <proof>
```

```
lemma meet-above-iff:
  fixes  $x y z :: 'a :: Finite-Meet-cpo$ 
  shows  $z \sqsubseteq x \sqcap y \longleftrightarrow z \sqsubseteq x \wedge z \sqsubseteq y$ 
  <proof>
```

```
lemma below-meet[simp]:
  fixes  $x y :: 'a :: Finite-Meet-cpo$ 
  assumes  $x \sqsubseteq z$ 
  shows  $(x \sqcap z) = x$  <proof>
```


lemma *below-meet2*[simp]:

fixes $x\ y :: 'a :: \text{Finite-Meet-cpo}$

assumes $z \sqsubseteq x$

shows $(x \sqcap z) = z$ *<proof>*

lemma *meet-aboveI*:

fixes $x\ y\ z :: 'a :: \text{Finite-Meet-cpo}$

shows $z \sqsubseteq x \implies z \sqsubseteq y \implies z \sqsubseteq x \sqcap y$ *<proof>*

lemma *is-meetI*:

fixes $x\ y\ z :: 'a :: \text{Finite-Meet-cpo}$

assumes $z \sqsubseteq x$

assumes $z \sqsubseteq y$

assumes $\bigwedge a. [a \sqsubseteq x ; a \sqsubseteq y] \implies a \sqsubseteq z$

shows $x \sqcap y = z$

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *meet-monofun1*:

fixes $y :: 'a :: \text{Finite-Meet-cpo}$

shows *monofun* $(\lambda x. (x \sqcap y))$

<proof>

lemma *chain-meet1*:

fixes $y :: 'a :: \text{Finite-Meet-cpo}$

assumes *chain* Y

shows *chain* $(\lambda i. Y\ i \sqcap y)$

<proof>

class *cont-binary-meet* = *Finite-Meet-cpo* +

assumes *meet-cont'*: *chain* $Y \implies (\bigsqcap i. Y\ i) \sqcap y = (\bigsqcap i. Y\ i \sqcap y)$

lemma *meet-cont1*:

fixes $y :: 'a :: \text{cont-binary-meet}$

shows *cont* $(\lambda x. (x \sqcap y))$

<proof>

lemma *meet-cont2*:

fixes $x :: 'a :: \text{cont-binary-meet}$

shows *cont* $(\lambda y. (x \sqcap y))$ *<proof>*

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

end

2.8 Nominal-HOLCF

theory *Nominal-HOLCF*

imports

Nominal-Utils HOLCF-Utils

begin

2.8.1 Type class of continuous permutations and variations thereof

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

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

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

instance *pcpo-pt* \subseteq *cont-pt*
 ⟨*proof*⟩

instance *discr-pt* \subseteq *cont-pt*
 ⟨*proof*⟩

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

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

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

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

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

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

<proof>

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

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

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

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

lemma *bot-fresh*[*simp*]: $a \# (\perp :: 'a :: \text{pcpo-pt})$
<proof>

lemma *bot-fresh-star*[*simp*]: $a \#^* (\perp :: 'a :: \text{pcpo-pt})$
<proof>

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

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

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

2.8.2 Instance for *cfun*

instantiation *cfun* :: $(\text{cont-pt}, \text{cont-pt}) \text{ pt}$

begin

definition $p \cdot (f :: 'a \rightarrow 'b) = (\Lambda x. p \cdot (f \cdot (- p \cdot x)))$

instance

<proof>

end

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

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

lemma *permute-Lam*: $\text{cont } f \implies p \cdot (\Lambda x. f x) = (\Lambda x. (p \cdot f) x)$
<proof>

lemma *Abs-cfun-eqv*: $\text{cont } f \implies (p \cdot \text{Abs-cfun}) f = \text{Abs-cfun } f$
<proof>

lemma *cfun-eqvI*: $(\Lambda x. p \cdot (f \cdot x) = f' \cdot (p \cdot x)) \implies p \cdot f = f'$
<proof>

lemma *ID-eqv*[*eqv*]: $\pi \cdot \text{ID} = \text{ID}$
<proof>

instance *cfun* :: (*cont-pt*, *cont-pt*) *cont-pt*
<proof>

instance *cfun* :: ({*pure,cont-pt*}, {*pure,cont-pt*}) *pure*
<proof>

instance *cfun* :: (*cont-pt*, *pcpo-pt*) *pcpo-pt*
<proof>

2.8.3 Instance for *fun*

lemma *permute-fun-eq*: $\text{permute } p = (\lambda f. (\text{permute } p) \circ f \circ (\text{permute } (-p)))$
<proof>

instance *fun* :: (*pt*, *cont-pt*) *cont-pt*
<proof>

lemma *fix-eqv*[*eqv*]:
 $\pi \cdot \text{fix} = (\text{fix} :: ('a \rightarrow 'a) \rightarrow 'a :: \{\text{cont-pt}, \text{pcpo}\})$
<proof>

2.8.4 Instance for *u*

instantiation *u* :: (*cont-pt*) *pt*
begin

definition $p \cdot (x :: 'a \ u) = \text{fup} \cdot (\Lambda x. \text{up} \cdot (p \cdot x)) \cdot x$

instance
<proof>

end

instance *u* :: (*cont-pt*) *cont-pt*
<proof>

instance *u* :: (*cont-pt*) *pcpo-pt* *<proof>*

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

instance *u* :: (*pure-cont-pt*) *pure*

<proof>

lemma *up-eqvt*[*eqvt*]: $\pi \cdot up = up$
<proof>

lemma *fup-eqvt*[*eqvt*]: $\pi \cdot fup = fup$
<proof>

2.8.5 Instance for *lift*

instantiation *lift* :: (*pt*) *pt*

begin

definition $p \cdot (x :: 'a \text{ lift}) = \text{case-lift } \perp (\lambda x. \text{Def } (p \cdot x)) x$

instance

<proof>

end

instance *lift* :: (*pt*) *cont-pt*
<proof>

instance *lift* :: (*pt*) *pcpo-pt* *<proof>*

instance *lift* :: (*pure*) *pure*
<proof>

lemma *Def-eqvt*[*eqvt*]: $\pi \cdot (\text{Def } x) = \text{Def } (\pi \cdot x)$
<proof>

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

2.8.6 Instance for *prod*

instance *prod* :: (*cont-pt*, *cont-pt*) *cont-pt*
<proof>

end

2.9 Env

theory *Env*

imports *Main HOLCF-Join-Classes*

begin

default-sort *type*

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

2.9.1 The domain of a pcpo-valued function

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

lemma $bot-edom[simp]: edom\ \perp = \{\}$ $\langle proof \rangle$

lemma $bot-edom2[simp]: edom\ (\lambda\ .\ \perp) = \{\}$ $\langle proof \rangle$

lemma $edomIff: (a \in edom\ m) = (m\ a \neq \perp)$ $\langle proof \rangle$

lemma $edom-iff2: (m\ a = \perp) \longleftrightarrow (a \notin edom\ m)$ $\langle proof \rangle$

lemma $edom-empty-iff-bot: edom\ m = \{\} \longleftrightarrow m = \perp$
 $\langle proof \rangle$

lemma $lookup-not-edom: x \notin edom\ m \Longrightarrow m\ x = \perp$ $\langle proof \rangle$

lemma $lookup-edom[simp]: m\ x \neq \perp \Longrightarrow x \in edom\ m$ $\langle proof \rangle$

lemma $edom-mono: x \sqsubseteq y \Longrightarrow edom\ x \subseteq edom\ y$
 $\langle proof \rangle$

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

2.9.2 Updates

lemma $edom-fun-upd-subset: edom\ (h\ (x := v)) \subseteq insert\ x\ (edom\ h)$
 $\langle proof \rangle$

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 $env-restr\ S\ m = (\lambda\ x. if\ x \in S\ then\ m\ x\ else\ \perp)$

abbreviation $env-restr-rev\ (infixl\ f\ |'\ 110)$
where $env-restr-rev\ m\ S \equiv env-restr\ S\ m$

notation (*latex output*) $env-restr-rev\ (-|.)$

lemma $env-restr-empty-iff[simp]: m\ f\ |'\ S = \perp \longleftrightarrow edom\ m \cap S = \{\}$
 $\langle proof \rangle$

lemmas *env-restr-empty* = *iffD2*[*OF env-restr-empty-iff*, *simp*]

lemma *lookup-env-restr*[*simp*]: $x \in S \implies (m f|' S) x = m x$
<proof>

lemma *lookup-env-restr-not-there*[*simp*]: $x \notin S \implies (env-restr S m) x = \perp$
<proof>

lemma *lookup-env-restr-eq*: $(m f|' S) x = (if x \in S then m x else \perp)$
<proof>

lemma *env-restr-eqI*: $(\bigwedge x. x \in S \implies m_1 x = m_2 x) \implies m_1 f|' S = m_2 f|' S$
<proof>

lemma *env-restr-eqD*: $m_1 f|' S = m_2 f|' S \implies x \in S \implies m_1 x = m_2 x$
<proof>

lemma *env-restr-belowI*: $(\bigwedge x. x \in S \implies m_1 x \sqsubseteq m_2 x) \implies m_1 f|' S \sqsubseteq m_2 f|' S$
<proof>

lemma *env-restr-belowD*: $m_1 f|' S \sqsubseteq m_2 f|' S \implies x \in S \implies m_1 x \sqsubseteq m_2 x$
<proof>

lemma *env-restr-env-restr*[*simp*]:
 $x f|' d2 f|' d1 = x f|' (d1 \cap d2)$
<proof>

lemma *env-restr-env-restr-subset*:
 $d1 \subseteq d2 \implies x f|' d2 f|' d1 = x f|' d1$
<proof>

lemma *env-restr-useless*: $edom m \subseteq S \implies m f|' S = m$
<proof>

lemma *env-restr-UNIV*[*simp*]: $m f|' UNIV = m$
<proof>

lemma *env-restr-fun-upd*[*simp*]: $x \in S \implies m1(x := v) f|' S = (m1 f|' S)(x := v)$
<proof>

lemma *env-restr-fun-upd-other*[*simp*]: $x \notin S \implies m1(x := v) f|' S = m1 f|' S$
<proof>

lemma *env-restr-eq-subset*:
assumes $S \subseteq S'$
and $m1 f|' S' = m2 f|' S'$
shows $m1 f|' S = m2 f|' S$
<proof>

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

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

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

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

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

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

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

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

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

2.9.4 Deleting

definition *env-delete* :: $'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b::pcpo)$
where $env-delete x m = m(x := \perp)$

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

lemma *lookup-env-delete-None[simp]*:
 $env-delete x m x = \perp$
 $\langle proof \rangle$

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

lemma *edom-env-delete-subset*:

$$\text{edom } (\text{env-delete } x \ m) \subseteq \text{edom } m \ \langle \text{proof} \rangle$$

lemma *env-delete-fun-upd[simp]*:

$$\text{env-delete } x \ (m(x := v)) = \text{env-delete } x \ m \\ \langle \text{proof} \rangle$$

lemma *env-delete-fun-upd2[simp]*:

$$(\text{env-delete } x \ m)(x := v) = m(x := v) \\ \langle \text{proof} \rangle$$

lemma *env-delete-fun-upd3[simp]*:

$$x \neq y \implies \text{env-delete } x \ (m(y := v)) = (\text{env-delete } x \ m)(y := v) \\ \langle \text{proof} \rangle$$

lemma *env-delete-noop[simp]*:

$$x \notin \text{edom } m \implies \text{env-delete } x \ m = m \\ \langle \text{proof} \rangle$$

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

$\langle \text{proof} \rangle$

lemma *env-restr-env-delete-other[simp]*: $x \notin S \implies \text{env-delete } x \ m \ f|' S = m \ f|' S$

$\langle \text{proof} \rangle$

lemma *env-delete-restr*: $\text{env-delete } x \ m = m \ f|' (-\{x\})$

$\langle \text{proof} \rangle$

lemma *below-env-deleteI*: $f \ x = \perp \implies f \sqsubseteq g \implies f \sqsubseteq \text{env-delete } x \ g$

$\langle \text{proof} \rangle$

lemma *env-delete-below-cong[intro]*:

assumes $x \neq v \implies e1 \ x \sqsubseteq e2 \ x$

shows $\text{env-delete } v \ e1 \ x \sqsubseteq \text{env-delete } v \ e2 \ x$

$\langle \text{proof} \rangle$

lemma *env-delete-env-restr-swap*:

$$\text{env-delete } x \ (\text{env-restr } S \ e) = \text{env-restr } S \ (\text{env-delete } x \ e)$$

$\langle \text{proof} \rangle$

lemma *env-delete-mono*:

$$m \sqsubseteq m' \implies \text{env-delete } x \ m \sqsubseteq \text{env-delete } x \ m'$$

$\langle \text{proof} \rangle$

lemma *env-delete-below-arg*:

$$\text{env-delete } x \ m \sqsubseteq m$$

$\langle \text{proof} \rangle$

2.9.5 Merging of two functions

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

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

lemma *override-on-bot*[simp]:

$$\begin{aligned} \perp \text{++}_S m &= m \text{ } f \text{ } S \\ m \text{++}_S \perp &= m \text{ } f \text{ } (-S) \\ \langle \text{proof} \rangle \end{aligned}$$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *override-on-upd-swap*:

$$\begin{aligned} x \notin S \implies \varrho(x := z) \text{++}_S \varrho' &= (\varrho \text{++}_S \varrho')(x := z) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *override-on-upd*:

$$\begin{aligned} x \in S \implies \varrho \text{++}_S (\varrho'(x := z)) &= (\varrho \text{++}_S - \{x\} \ \varrho')(x := z) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *env-restr-add*: $(m1 \text{++}_{S2} m2) \ f \text{ } S = m1 \ f \text{ } S \text{++}_{S2} m2 \ f \text{ } S$

$\langle \text{proof} \rangle$

lemma *env-delete-add*: $\text{env-delete } x \ (m1 \text{++}_S m2) = \text{env-delete } x \ m1 \text{++}_S - \{x\} \ \text{env-delete } x \ m2$

$\langle \text{proof} \rangle$

2.9.6 Environments with binary joins

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *env-restr-join*:

$$\begin{aligned} \text{fixes } m1 \ m2 &:: 'a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\} \\ \text{shows } (m1 \sqcup m2) \ f \text{ } S &= (m1 \ f \text{ } S) \sqcup (m2 \ f \text{ } S) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *env-restr-join2*:

$$\text{fixes } m &:: 'a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\}$$

shows $m f|^{\prime} S \sqcup m f|^{\prime} S' = m f|^{\prime} (S \cup S')$
 ⟨proof⟩

lemma *join-env-restr-UNIV*:

fixes $m :: 'a::type \Rightarrow 'b::\{Finite-Join-cpo,pcpo\}$
shows $S1 \cup S2 = UNIV \Longrightarrow (m f|^{\prime} S1) \sqcup (m f|^{\prime} S2) = m$
 ⟨proof⟩

lemma *env-restr-split*:

fixes $m :: 'a::type \Rightarrow 'b::\{Finite-Join-cpo,pcpo\}$
shows $m = m f|^{\prime} S \sqcup m f|^{\prime} (- S)$
 ⟨proof⟩

lemma *env-restr-below-split*:

$m f|^{\prime} S \sqsubseteq m' \Longrightarrow m f|^{\prime} (- S) \sqsubseteq m' \Longrightarrow m \sqsubseteq m'$
 ⟨proof⟩

2.9.7 Singleton environments

definition *esing* :: $'a \Rightarrow 'b::\{pcpo\} \rightarrow ('a \Rightarrow 'b)$

where $esing\ x = (\Lambda a. (\lambda y. (if\ x = y\ then\ a\ else\ \perp)))$

lemma *esing-bot[simp]*: $esing\ x \cdot \perp = \perp$
 ⟨proof⟩

lemma *esing-simps[simp]*:

$(esing\ x \cdot n)\ x = n$
 $x' \neq x \Longrightarrow (esing\ x \cdot n)\ x' = \perp$
 ⟨proof⟩

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

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

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

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

lemma *env-delete-esing[simp]*: $env-delete\ x\ (esing\ x \cdot n) = \perp$
 ⟨proof⟩

lemma *env-restr-esing[simp]*:

$x \in S \Longrightarrow esing\ x \cdot v f|^{\prime} S = esing\ x \cdot v$
 ⟨proof⟩

lemma *env-restr-esing2*[*simp*]:
 $x \notin S \implies \text{esing } x.v \text{ f} |' S = \perp$
 ⟨*proof*⟩

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

end

2.10 Env-Nominal

theory *Env-Nominal*
imports *Env Nominal-Utills Nominal-HOLCF*
begin

2.10.1 Equivariance lemmas

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

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

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

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

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

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

2.10.2 Permutation and restriction

lemma *env-restr-perm*:
fixes $\varrho :: 'a::at-base \Rightarrow 'b::\{pcpo-pt,pure\}$
assumes $\text{supp } p \#* S$ and [*simp*]: *finite S*

shows $(p \cdot \varrho) f|' S = \varrho f|' S$
 $\langle proof \rangle$

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

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

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

2.10.3 Pure codomains

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

end

2.11 Env-HOLCF

theory *Env-HOLCF*
imports *Env HOLCF-Utills*
begin

2.11.1 Continuity and pcpo-valued functions

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

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

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

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

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

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

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

lemma *env-restr-belowI*:
assumes $\bigwedge x. x \in S \implies (m1 f|' S) x \sqsubseteq (m2 f|' S) x$
shows $m1 f|' S \sqsubseteq m2 f|' S$
<proof>

lemma *env-restr-belowI2*:
assumes $\bigwedge x. x \in S \implies m1 x \sqsubseteq m2 x$
shows $m1 f|' S \sqsubseteq m2$
<proof>

lemma *env-restr-below-itself*:
shows $m f|' S \sqsubseteq m$
<proof>

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

lemma *env-restr-belowD*:
assumes $m1 f|' S \sqsubseteq m2 f|' S$
assumes $x \in S$

shows $m1\ x \sqsubseteq m2\ x$
<proof>

lemma *env-restr-eqD*:
assumes $m1\ f|' S = m2\ f|' S$
assumes $x \in S$
shows $m1\ x = m2\ x$
<proof>

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

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

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

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

lemma *env-delete-cont*: *cont* (*env-delete* x)
<proof>

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

end

2.12 EvalHeap

theory *EvalHeap*
imports *AList-Utills Env Nominal2.Nominal2 HOLCF-Utills*
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

$evalHeap \ [] \ - = \perp$
 $| evalHeap ((x,e)\#h) eval = (evalHeap h eval) (x := eval e)$

lemma *cont2cont-evalHeap[simp, cont2cont]*:
 $(\bigwedge e . e \in snd \text{ ' set } h \implies cont (\lambda \varrho . eval \varrho e)) \implies cont (\lambda \varrho . evalHeap h (eval \varrho))$
 $\langle proof \rangle$

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

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

lemma *evalHeap-cong[fundef-cong]*:
 $\llbracket heap1 = heap2 ; (\bigwedge e . e \in snd \text{ ' set } heap2 \implies eval1 e = eval2 e) \rrbracket$
 $\implies evalHeap heap1 eval1 = evalHeap heap2 eval2$
 $\langle proof \rangle$

lemma *lookupEvalHeap*:
assumes $v \in domA h$
shows $(evalHeap h f) v = f (the (map-of h v))$
 $\langle proof \rangle$

lemma *lookupEvalHeap'*:
assumes $map-of \Gamma v = Some e$
shows $(evalHeap \Gamma f) v = f e$
 $\langle proof \rangle$

lemma *lookupEvalHeap-other[simp]*:
assumes $v \notin domA \Gamma$
shows $(evalHeap \Gamma f) v = \perp$
 $\langle proof \rangle$

lemma *env-restr-evalHeap-noop*:
 $domA h \subseteq S \implies env-restr S (evalHeap h eval) = evalHeap h eval$
 $\langle proof \rangle$

lemma *env-restr-evalHeap-same[simp]*:
 $env-restr (domA h) (evalHeap h eval) = evalHeap h eval$
 $\langle proof \rangle$

lemma *evalHeap-cong'*:
 $\llbracket (\bigwedge x . x \in domA heap \implies eval1 (the (map-of heap x)) = eval2 (the (map-of heap x))) \rrbracket$
 $\implies evalHeap heap eval1 = evalHeap heap eval2$
 $\langle proof \rangle$

lemma *lookupEvalHeapNotAppend[simp]*:
assumes $x \notin domA \Gamma$

shows $(evalHeap (\Gamma @ h) f) x = evalHeap h f x$
 $\langle proof \rangle$

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

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

2.12.2 Reordering lemmas

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

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

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

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

end

3 Launchbury's natural semantics

3.1 Vars

```
theory Vars
imports Nominal2.Nominal2
begin
```

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

```
atom-decl var

end
```

3.2 Terms

```
theory Terms
imports Nominal-Utils Vars AList-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  $\Rightarrow$  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* ($\lambda\cdot$. - [100, 100] 100)

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

lemma *exp-assn-size-eqvt*[*eqvt*]: $p \cdot (\text{size} :: \text{exp} \Rightarrow \text{nat}) = \text{size}$
⟨*proof*⟩

3.2.2 Rewriting in terms of heaps

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

Conversion from *assn* to *heap*.

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

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

The other direction.

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

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

lemma *heapToAssn-eqvt*[*simp,eqvt*]: $p \cdot \text{heapToAssn } \Gamma = \text{heapToAssn } (p \cdot \Gamma)$
⟨*proof*⟩

lemma *bn-heapToAssn*: $\text{bn } (\text{heapToAssn } \Gamma) = \text{map } (\lambda x. \text{atom } (\text{fst } x)) \Gamma$
⟨*proof*⟩

lemma *set-bn-to-atom-domA*:
 $\text{set } (\text{bn } as) = \text{atom } \text{' domA } (asToHeap as)$
⟨*proof*⟩

They are inverse to each other.

lemma *heapToAssn-asToHeap*[*simp*]:
 $\text{heapToAssn } (asToHeap as) = as$
⟨*proof*⟩

lemma *asToHeap-heapToAssn*[*simp*]:
 $asToHeap (\text{heapToAssn } as) = as$
⟨*proof*⟩

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

They are transparent to various notions from the Nominal package.

lemma *supp-heapToAssn*: $supp\ (heapToAssn\ \Gamma) = supp\ \Gamma$
 ⟨proof⟩

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

lemma *fv-asToHeap*: $fv\ (asToHeap\ \Gamma) = fv\ \Gamma$
 ⟨proof⟩

lemma *fv-heapToAssn*: $fv\ (heapToAssn\ \Gamma) = fv\ \Gamma$
 ⟨proof⟩

lemma [simp]: $size\ (heapToAssn\ \Gamma) = size-list\ (\lambda\ (v,e) . size\ e)\ \Gamma$
 ⟨proof⟩

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

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

hide-const *HOL.Let*

definition *Let* :: $heap \Rightarrow exp \Rightarrow exp$
where $Let\ \Gamma\ e = LetA\ (heapToAssn\ \Gamma)\ e$

notation (*latex output*) $Let\ (let\ -\ in\ -)$

abbreviation

$LetBe :: var \Rightarrow exp \Rightarrow exp \Rightarrow exp\ (let\ -\ be\ -\ in\ -\ [100,100,100]\ 100)$

where

$let\ x\ be\ t1\ in\ t2 \equiv Let\ [(x,t1)]\ t2$

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

lemma *size-Let*[simp]: $size\ (Let\ \Gamma\ e) = size-list\ (\lambda p. size\ (snd\ p))\ \Gamma + size\ e + Suc\ 0$
 ⟨proof⟩

lemma *Let-distinct*[simp]:

$Var\ v \neq Let\ \Gamma\ e$
 $Let\ \Gamma\ e \neq Var\ v$
 $App\ e\ v \neq Let\ \Gamma\ e'$
 $Lam\ [v].\ e' \neq Let\ \Gamma\ e$
 $Let\ \Gamma\ e \neq Lam\ [v].\ e'$
 $Let\ \Gamma\ e' \neq App\ e\ v$
 $Bool\ b \neq Let\ \Gamma\ e$

$Let \Gamma e \neq Bool b$
 $(scrut ? e1 : e2) \neq Let \Gamma e$
 $Let \Gamma e \neq (scrut ? e1 : e2)$
 $\langle proof \rangle$

lemma *Let-perm-simps*[simp,eqvt]:
 $p \cdot Let \Gamma e = Let (p \cdot \Gamma) (p \cdot e)$
 $\langle proof \rangle$

lemma *Let-supp*:
 $supp (Let \Gamma e) = (supp e \cup supp \Gamma) - atom \text{ ' } (domA \Gamma)$
 $\langle proof \rangle$

lemma *Let-fresh*[simp]:
 $a \# Let \Gamma e = (a \# e \wedge a \# \Gamma \vee a \in atom \text{ ' } domA \Gamma)$
 $\langle proof \rangle$

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

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

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

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 $\bigwedge exp var. P1 exp \implies P1 (App exp var)$
assumes $\bigwedge \Gamma exp. P2 \Gamma \implies P1 exp \implies P1 (Let \Gamma exp)$
assumes $\bigwedge var exp. P1 exp \implies P1 (Lam [var]. exp)$
assumes $\bigwedge b. P1 (Bool b)$
assumes $\bigwedge scrut e1 e2. P1 scrut \implies P1 e1 \implies P1 e2 \implies P1 (scrut ? e1 : e2)$
assumes $P2 \square$

assumes $\bigwedge var\ exp\ \Gamma. P1\ exp \implies P2\ \Gamma \implies P2\ ((var, exp)\#\Gamma)$
shows $P1\ e$ **and** $P2\ \Gamma$
 $\langle proof \rangle$

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

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 $\bigwedge exp\ var. P\ exp \implies P\ (App\ exp\ var)$
assumes $\bigwedge \Gamma\ exp. (\bigwedge x. x \in domA\ \Gamma \implies P\ (the\ (map-of\ \Gamma\ x))) \implies P\ exp \implies P\ (Let\ \Gamma\ exp)$
assumes $\bigwedge var\ exp. P\ exp \implies P\ (Lam\ [var].\ exp)$
assumes $\bigwedge b. P\ (Bool\ b)$
assumes $\bigwedge scrut\ e1\ e2. P\ scrut \implies P\ e1 \implies P\ e2 \implies P\ (scrut\ ?\ e1 : e2)$
shows $P\ exp$
 $\langle proof \rangle$

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

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

3.2.4 Testing alpha equivalence

lemma *alpha-test*:
shows $Lam\ [x].\ (Var\ x) = Lam\ [y].\ (Var\ y)$
 $\langle proof \rangle$

lemma *alpha-test2*:
shows $let\ x\ be\ (Var\ x)\ in\ (Var\ x) = let\ y\ be\ (Var\ y)\ in\ (Var\ y)$
 $\langle proof \rangle$

lemma *alpha-test3*:
shows
 $Let\ [(x,\ Var\ y),\ (y,\ Var\ x)]\ (Var\ x)$
 $=$
 $Let\ [(y,\ Var\ x),\ (x,\ Var\ y)]\ (Var\ y)$ (**is** $Let\ ?la\ ?lb = -$)
 $\langle proof \rangle$

3.2.5 Free variables

lemma *fv-supp-exp*: $supp\ e = atom\ ' (fv\ (e::exp) :: var\ set)$ **and** *fv-supp-as*: $supp\ as = atom\ ' (fv\ (as::assn) :: var\ set)$
 $\langle proof \rangle$

lemma *fv-supp-heap*: $supp\ (\Gamma::heap) = atom\ ' (fv\ \Gamma :: var\ set)$
 $\langle proof \rangle$

lemma *fv-Lam[simp]*: $fv\ (Lam\ [x].\ e) = fv\ e - \{x\}$
 $\langle proof \rangle$

lemma *fv-Var[simp]*: $fv\ (Var\ x) = \{x\}$
 $\langle proof \rangle$

lemma *fv-App[simp]*: $fv\ (App\ e\ x) = insert\ x\ (fv\ e)$
 $\langle proof \rangle$

lemma *fv-Let[simp]*: $fv\ (Let\ \Gamma\ e) = (fv\ \Gamma \cup fv\ e) - domA\ \Gamma$
 $\langle proof \rangle$

lemma *fv-Bool[simp]*: $fv\ (Bool\ b) = \{\}$
 $\langle proof \rangle$

lemma *fv-IfThenElse[simp]*: $fv\ (scrut\ ?\ e1\ : e2) = fv\ scrut \cup fv\ e1 \cup fv\ e2$

<proof>

lemma *fv-delete-heap*:

assumes *map-of* $\Gamma x = \text{Some } e$

shows $fv (\text{delete } x \ \Gamma, e) \cup \{x\} \subseteq (fv (\Gamma, Var \ x) :: var \ set)$

<proof>

3.2.6 Lemmas helping with nominal definitions

lemma *eqvt-lam-case*:

assumes $Lam \ [x]. e = Lam \ [x']. e'$

assumes $\bigwedge \pi . \text{supp } (-\pi) \ \#\ast (fv (Lam \ [x]. e) :: var \ set) \implies$
 $\text{supp } \pi \ \#\ast (Lam \ [x]. e) \implies$

$F (\pi \cdot e) (\pi \cdot x) (Lam \ [x]. e) = F \ e \ x (Lam \ [x]. e)$

shows $F \ e \ x (Lam \ [x]. e) = F \ e' \ x' (Lam \ [x']. e')$

<proof>

lemma *eqvt-let-case*:

assumes $Let \ as \ body = Let \ as' \ body'$

assumes $\bigwedge \pi .$

$\text{supp } (-\pi) \ \#\ast (fv (Let \ as \ body) :: var \ set) \implies$

$\text{supp } \pi \ \#\ast Let \ as \ body \implies$

$F (\pi \cdot as) (\pi \cdot body) (Let \ as \ body) = F \ as \ body (Let \ as \ body)$

shows $F \ as \ body (Let \ as \ body) = F \ as' \ body' (Let \ as' \ body')$

<proof>

3.2.7 A smart constructor for lets

Certain 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* :: $heap \Rightarrow exp \Rightarrow exp$

where $SmartLet \ \Gamma \ e = (if \ \Gamma = [] \ \text{then } e \ \text{else } Let \ \Gamma \ e)$

lemma *SmartLet-eqvt[eqvt]*: $\pi \cdot (SmartLet \ \Gamma \ e) = SmartLet (\pi \cdot \Gamma) (\pi \cdot e)$

<proof>

lemma *SmartLet-supp*:

$\text{supp } (SmartLet \ \Gamma \ e) = (\text{supp } e \cup \text{supp } \Gamma) - \text{atom } ' (domA \ \Gamma)$

<proof>

lemma *fv-SmartLet[simp]*: $fv (SmartLet \ \Gamma \ e) = (fv \ \Gamma \cup fv \ e) - domA \ \Gamma$

<proof>

3.2.8 A predicate for value expressions

nominal-function $isLam :: exp \Rightarrow bool$ **where**

$isLam (Var x) = False$ |
 $isLam (Lam [x]. e) = True$ |
 $isLam (App e x) = False$ |
 $isLam (Let as e) = False$ |
 $isLam (Bool b) = False$ |
 $isLam (scrut ? e1 : e2) = False$
 $\langle proof \rangle$

nominal-termination ($eqvt$) $\langle proof \rangle$

lemma $isLam-Lam$: $isLam (Lam [x]. e) \langle proof \rangle$

lemma $isLam-obtain-fresh$:

assumes $isLam z$
obtains $y e'$
where $z = (Lam [y]. e')$ **and** $atom y \# (c::'a::fs)$
 $\langle proof \rangle$

nominal-function $isVal :: exp \Rightarrow bool$ **where**

$isVal (Var x) = False$ |
 $isVal (Lam [x]. e) = True$ |
 $isVal (App e x) = False$ |
 $isVal (Let as e) = False$ |
 $isVal (Bool b) = True$ |
 $isVal (scrut ? e1 : e2) = False$
 $\langle proof \rangle$

nominal-termination ($eqvt$) $\langle proof \rangle$

lemma $isVal-Lam$: $isVal (Lam [x]. e) \langle proof \rangle$

lemma $isVal-Bool$: $isVal (Bool b) \langle proof \rangle$

3.2.9 The notion of thunks

definition $thunks :: heap \Rightarrow var\ set$ **where**

$thunks \Gamma = \{x . case\ map-of\ \Gamma\ x\ of\ Some\ e \Rightarrow \neg isVal\ e \mid None \Rightarrow False\}$

lemma $thunks-Nil[simp]$: $thunks [] = \{\}$ $\langle proof \rangle$

lemma $thunks-domA$: $thunks \Gamma \subseteq domA \Gamma$

$\langle proof \rangle$

lemma $thunks-Cons$: $thunks ((x,e)\#\Gamma) = (if\ isVal\ e\ then\ thunks\ \Gamma - \{x\}\ else\ insert\ x\ (thunks\ \Gamma))$

$\langle proof \rangle$

lemma $thunks-append[simp]$: $thunks (\Delta @ \Gamma) = thunks \Delta \cup (thunks \Gamma - domA \Delta)$

$\langle proof \rangle$

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

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

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

lemma *thunks-cong*: $map-of\ \Gamma = map-of\ \Delta \implies thunks\ \Gamma = thunks\ \Delta$
 ⟨proof⟩

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

3.2.10 Non-recursive Let bindings

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

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

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

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

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

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

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 $\bigwedge exp\ var c. (\bigwedge c. P\ c\ \exp) \implies P\ c\ (App\ \exp\ var)$
assumes $\bigwedge \Gamma\ exp\ c.$
 $atom\ 'domA\ \Gamma\ \#* c \implies \neg\ nonrec\ \Gamma \implies (\bigwedge c\ x\ e. (x,e) \in set\ \Gamma \implies P\ c\ e) \implies (\bigwedge c. P\ c\ \exp) \implies P\ c\ (Let\ \Gamma\ \exp)$
assumes $\bigwedge x e \exp c. \{atom\ x\} \#* c \implies x \notin fv\ e \implies (\bigwedge c. P\ c\ e) \implies (\bigwedge c. P\ c\ \exp) \implies P\ c$
(let x be e in exp)
assumes $\bigwedge var \exp c. \{atom\ var\} \#* c \implies (\bigwedge c. P\ c\ \exp) \implies P\ c\ (Lam\ [var].\ \exp)$
assumes $\bigwedge b c. P\ c\ (Bool\ b)$
assumes $\bigwedge scrut\ e1\ e2\ c. (\bigwedge c. P\ c\ scrut) \implies (\bigwedge c. P\ c\ e1) \implies (\bigwedge c. P\ c\ e2) \implies P\ c$
(scrut ? e1 : e2)
shows $P\ (c::'a::fs)\ \exp$
<proof>

3.2.11 Renaming a lambda-bound variable

lemma *change-Lam-Variable*:

assumes $y' \neq y \implies atom\ y' \# (e,\ y)$
shows $Lam\ [y].\ e = Lam\ [y'].\ ((y \leftrightarrow y') \cdot e)$
<proof>

end

3.3 Substitution

theory *Substitution*

imports *Terms*

begin

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

fun

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

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

$subst :: exp \Rightarrow var \Rightarrow var \Rightarrow exp \ (-[::=] \ [1000,100,100] \ 1000)$
and
 $subst\text{-}heap :: heap \Rightarrow var \Rightarrow var \Rightarrow heap \ (-[::h=] \ [1000,100,100] \ 1000)$
where
 $(Var \ x)[y ::= z] = Var \ (x[y ::= v = z])$
 $| (App \ e \ v)[y ::= z] = App \ (e[y ::= z]) \ (v[y ::= v = z])$
 $| atom \ ' \ domA \ \Gamma \ \#^* \ (y,z) \Longrightarrow$
 $\quad (Let \ \Gamma \ body)[y ::= z] = Let \ (\Gamma[y ::= h = z]) \ (body[y ::= z])$
 $| atom \ x \ \# \ (y,z) \Longrightarrow (Lam \ [x].e)[y ::= z] = Lam \ [x].(e[y ::= z])$
 $| (Bool \ b)[y ::= z] = Bool \ b$
 $| (scrut \ ? \ e1 \ : \ e2)[y ::= z] = (scrut[y ::= z] \ ? \ e1[y ::= z] \ : \ e2[y ::= z])$
 $| [][y ::= h = z] = []$
 $| ((v,e)\# \ \Gamma)[y ::= h = z] = (v, e[y ::= z])\# \ (\Gamma[y ::= h = z])$
 $\langle proof \rangle$

nominal-termination (*eqvt*) $\langle proof \rangle$

lemma shows

True and $bn\text{-}subst[simp]: domA \ (subst\text{-}heap \ \Gamma \ y \ z) = domA \ \Gamma$
 $\langle proof \rangle$

lemma $subst\text{-}noop[simp]:$

shows $e[y ::= y] = e$ **and** $\Gamma[y::h=y] = \Gamma$
 $\langle proof \rangle$

lemma $subst\text{-}is\text{-}fresh[simp]:$

assumes $atom \ y \ \# \ z$

shows

$atom \ y \ \# \ e[y ::= z]$

and

$atom \ ' \ domA \ \Gamma \ \#^* \ y \Longrightarrow atom \ y \ \# \ \Gamma[y::h=z]$

$\langle proof \rangle$

lemma

$subst\text{-}pres\text{-}fresh: atom \ x \ \# \ e \vee x = y \Longrightarrow atom \ x \ \# \ z \Longrightarrow atom \ x \ \# \ e[y ::= z]$

and

$atom \ x \ \# \ \Gamma \vee x = y \Longrightarrow atom \ x \ \# \ z \Longrightarrow x \notin domA \ \Gamma \Longrightarrow atom \ x \ \# \ (\Gamma[y ::= h = z])$

$\langle proof \rangle$

lemma $subst\text{-}fresh\text{-}noop: atom \ x \ \# \ e \Longrightarrow e[x ::= y] = e$

and $subst\text{-}heap\text{-}fresh\text{-}noop: atom \ x \ \# \ \Gamma \Longrightarrow \Gamma[x ::= h = y] = \Gamma$

$\langle proof \rangle$

lemma $supp\text{-}subst\text{-}eq: supp \ (e[y ::= x]) = (supp \ e - \{atom \ y\}) \cup (if \ atom \ y \in supp \ e \ then \ \{atom \ x\} \ else \ \{\})$

and $atom \ ' \ domA \ \Gamma \ \#^* \ y \Longrightarrow supp \ (\Gamma[y::h=x]) = (supp \ \Gamma - \{atom \ y\}) \cup (if \ atom \ y \in supp \ \Gamma \ then \ \{atom \ x\} \ else \ \{\})$

$\langle proof \rangle$

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

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

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

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

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

lemma *subst-swap-same*: $\text{atom } x \# e \implies (x \leftrightarrow y) \cdot e = e[y ::= x]$
and $\text{atom } x \# \Gamma \implies \text{atom } \text{'domA } \Gamma \#* y \implies (x \leftrightarrow y) \cdot \Gamma = \Gamma[y ::h= x]$
 ⟨proof⟩

lemma *subst-subst-back*: $\text{atom } x \# e \implies e[y::=x][x::=y] = e$
and $\text{atom } x \# \Gamma \implies \text{atom } \text{'domA } \Gamma \#* y \implies \Gamma[y::h=x][x::h=y] = \Gamma$
 ⟨proof⟩

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

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

lemma *subst-SmartLet[simp]*:
 $\text{atom } \text{'domA } \Gamma \#* (y, z) \implies (\text{SmartLet } \Gamma \text{ body})[y ::= z] = \text{SmartLet } (\Gamma[y ::h= z]) (\text{body}[y ::= z])$
 ⟨proof⟩

lemma *subst-let-be[simp]*:
 $\text{atom } x' \# y \implies \text{atom } x' \# x \implies (\text{let } x' \text{ be } e \text{ in } \text{exp})[y ::= x] = (\text{let } x' \text{ be } e[y ::= x] \text{ in } \text{exp}[y ::= x])$
 ⟨proof⟩

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

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

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

<proof>

lemma *map-of-subst*:

map-of ($\Gamma[x::h=y]$) *k* = *map-option* ($\lambda e . e[x::=y]$) (*map-of* Γ *k*)

<proof>

lemma *mapCollect-subst[simp]*:

$\{e\ k\ v \mid k \mapsto v \in \text{map-of } \Gamma[x::h=y]\} = \{e\ k\ v[x::=y] \mid k \mapsto v \in \text{map-of } \Gamma\}$

<proof>

lemma *subst-eq-Cons*:

$\Gamma[x::h=y] = (x', e) \# \Delta \longleftrightarrow (\exists e' \Gamma'. \Gamma = (x', e') \# \Gamma' \wedge e'[x::=y] = e \wedge \Gamma'[x::h=y] = \Delta)$

<proof>

lemma *nonrec-subst*:

atom ' *domA* Γ $\#^* x \implies \text{atom}$ ' *domA* Γ $\#^* y \implies \text{nonrec } \Gamma[x::h=y] \longleftrightarrow \text{nonrec } \Gamma$

<proof>

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, following [Ses97].

inductive

reds :: *heap* \Rightarrow *exp* \Rightarrow *var list* \Rightarrow *heap* \Rightarrow *exp* \Rightarrow *bool*

($- : - \Downarrow - : - [50, 50, 50, 50] 50$)

where

Lambda:

$\Gamma : (\text{Lam } [x]. e) \Downarrow_L \Gamma : (\text{Lam } [x]. e)$

| *Application*: \llbracket

atom *y* $\# (\Gamma, e, x, L, \Delta, \Theta, z)$;

$\Gamma : e \Downarrow_L \Delta : (\text{Lam } [y]. e')$;

$\Delta : e'[y ::= x] \Downarrow_L \Theta : z$

$\rrbracket \implies$

$\Gamma : \text{App } e\ x \Downarrow_L \Theta : z$

| *Variable*: \llbracket

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

equivariance *reds*

nominal-inductive *reds*

avoids *Application*: y
 $\langle proof \rangle$

3.4.2 Example evaluations

lemma *eval-test*:

$\Downarrow : (\text{Let } [(x, Lam [y]. Var y)] (Var x)) \Downarrow_{\Downarrow} [(x, Lam [y]. Var y)] : (Lam [y]. Var y)$
 $\langle proof \rangle$

lemma *eval-test2*:

$y \neq x \Rightarrow n \neq y \Rightarrow n \neq x \Rightarrow \Downarrow : (\text{Let } [(x, Lam [y]. Var y)] (App (Var x) x)) \Downarrow_{\Downarrow} [(x, Lam [y]. Var y)] : (Lam [y]. Var y)$
 $\langle proof \rangle$

3.4.3 Better introduction rules

This variant do not require freshness.

lemma *reds-ApplicationI*:

assumes $\Gamma : e \Downarrow_L \Delta : Lam [y]. e'$
assumes $\Delta : e'[y::=x] \Downarrow_L \Theta : z$
shows $\Gamma : App \ e \ x \Downarrow_L \Theta : z$
 $\langle proof \rangle$

lemma *reds-SmartLet*: \Downarrow

$atom \text{ ' } domA \ \Delta \ \#* (\Gamma, L);$
 $\Delta @ \Gamma : body \Downarrow_L \Theta : z$
 $\Downarrow \Rightarrow$
 $\Gamma : \text{SmartLet } \Delta \ body \Downarrow_L \Theta : z$
 $\langle proof \rangle$

A single rule for values

lemma *reds-isValI*:

$isVal\ z \implies \Gamma : z \Downarrow_L \Gamma : z$

<proof>

3.4.4 Properties of the semantics

Heap entries are never removed.

lemma *reds-doesnt-forget*:

$\Gamma : e \Downarrow_L \Delta : z \implies domA\ \Gamma \subseteq domA\ \Delta$

<proof>

Live variables are not added to the heap.

lemma *reds-avoids-live'*:

assumes $\Gamma : e \Downarrow_L \Delta : z$

shows $(domA\ \Delta - domA\ \Gamma) \cap set\ L = \{\}$

<proof>

lemma *reds-avoids-live*:

$\llbracket \Gamma : e \Downarrow_L \Delta : z;$

$x \in set\ L;$

$x \notin domA\ \Gamma$

$\rrbracket \implies x \notin domA\ \Delta$

<proof>

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

lemma *reds-fresh*: $\llbracket \Gamma : e \Downarrow_L \Delta : z;$

$atom\ (x::var)\ \#\ (\Gamma, e)$

$\rrbracket \implies atom\ x\ \#\ (\Delta, z) \vee x \in (domA\ \Delta - set\ L)$

<proof>

lemma *reds-fresh-fv*: $\llbracket \Gamma : e \Downarrow_L \Delta : z;$

$x \in fv\ (\Delta, z) \wedge (x \notin domA\ \Delta \vee x \in set\ L)$

$\rrbracket \implies x \in fv\ (\Gamma, e)$

<proof>

lemma *new-free-vars-on-heap*:

assumes $\Gamma : e \Downarrow_L \Delta : z$

shows $fv\ (\Delta, z) - domA\ \Delta \subseteq fv\ (\Gamma, e) - domA\ \Gamma$

<proof>

lemma *reds-pres-closed*:

assumes $\Gamma : e \Downarrow_L \Delta : z$

and $fv\ (\Gamma, e) \subseteq set\ L \cup domA\ \Gamma$

shows $fv\ (\Delta, z) \subseteq set\ L \cup domA\ \Delta$

<proof>

Reducing the set of variables to avoid is always possible.

lemma *reds-smaller-L*: $\llbracket \Gamma : e \Downarrow_L \Delta : z ;$
 $set\ L' \subseteq set\ L$
 $\rrbracket \implies \Gamma : e \Downarrow_{L'} \Delta : z$
 $\langle proof \rangle$

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

lemma *result-evaluated*:
 $\Gamma : e \Downarrow_L \Delta : z \implies isVal\ z$
 $\langle proof \rangle$

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

end

4 Denotational domain

4.1 Value

theory *Value*
imports *HOLCF*
begin

4.1.1 The semantic domain for values and environments

domain *Value* = $Fn\ (lazy\ Value \rightarrow Value) \mid B\ (lazy\ bool\ discr)$

fixrec *Fn-project* :: $Value \rightarrow Value \rightarrow Value$
where $Fn-project.(Fn.f) = f$

abbreviation *Fn-project-abbr* (**infix** $\Downarrow_{Fn}\ 55$)
where $f \Downarrow_{Fn}\ v \equiv Fn-project.f.v$

lemma [*simp*]:
 $\perp \Downarrow_{Fn}\ x = \perp$
 $(B.b) \Downarrow_{Fn}\ x = \perp$
 $\langle proof \rangle$

fixrec *B-project* :: $Value \rightarrow Value \rightarrow Value \rightarrow Value$ **where**
 $B-project.(B.db).v_1.v_2 = (if\ undiscr\ db\ then\ v_1\ else\ v_2)$

lemma [*simp*]:

$B\text{-project} \cdot (B \cdot (\text{Discr } b)) \cdot v_1 \cdot v_2 = (\text{if } b \text{ then } v_1 \text{ else } v_2)$
 $B\text{-project} \cdot \perp \cdot v_1 \cdot v_2 = \perp$
 $B\text{-project} \cdot (Fn \cdot f) \cdot v_1 \cdot v_2 = \perp$
 <proof>

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

lemma *Value-chainE*[consumes 1, case-names bot B Fn]:

assumes *chain* *Y*
obtains $Y = (\lambda \cdot \perp) \mid$
 $n \ b \ \mathbf{where} \ Y = (\lambda \ m. (\text{if } m < n \text{ then } \perp \text{ else } B \cdot b)) \mid$
 $n \ Y' \ \mathbf{where} \ Y = (\lambda \ m. (\text{if } m < n \text{ then } \perp \text{ else } Fn \cdot (Y' (m-n)))) \ \mathbf{chain} \ Y'$
 <proof>

end

4.2 Value-Nominal

theory *Value-Nominal*

imports *Value Nominal-Utils Nominal-HOLCF*

begin

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

instantiation *Value* :: *pure*

begin

definition $p \cdot (v :: \text{Value}) = v$

instance

<proof>

end

instance *Value* :: *pcpo-pt*

<proof>

end

5 Denotational semantics

5.1 Iterative

```
theory Iterative
imports Env-HOLCF
begin
```

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

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

  lemma split-x:
    fixes  $y$ 
    obtains  $y = x$  and  $y \notin S \mid y \in S$  and  $y \neq x \mid y \notin S$  and  $y \neq x$   $\langle proof \rangle$ 
  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  $(fix \cdot F) y = (F \cdot (fix \cdot F)) y$ 
     $\langle proof \rangle$ 

  lemma R-S:  $\bigwedge y. y \in S \implies (fix \cdot R) y = (e1 \cdot (fix \cdot (H (fix \cdot R)))) y$ 
     $\langle proof \rangle$ 

  lemma R'-S:  $\bigwedge y. y \in S \implies (fix \cdot R') y = (e1 \cdot (fix \cdot (H (fix \cdot R')))) y$ 
     $\langle proof \rangle$ 

  lemma HR-is-R[simp]:  $fix \cdot (H (fix \cdot R)) = fix \cdot R$ 
     $\langle proof \rangle$ 

  lemma HR'-is-R'[simp]:  $fix \cdot (H (fix \cdot R')) = fix \cdot R'$ 
     $\langle proof \rangle$ 

  lemma H-noop:
    fixes  $\rho' \rho''$ 
    assumes  $\bigwedge y. y \in S \implies y \neq x \implies (e1 \cdot \rho'') y \sqsubseteq \rho' y$ 
    shows  $H \rho' \cdot \rho'' \sqsubseteq \rho'$ 
     $\langle proof \rangle$ 
```

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

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

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

end

end

5.2 HasESem

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

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

locale *has-ESem* =
 fixes *ESem* :: $'exp::pt \Rightarrow ('var::at-base \Rightarrow 'value) \rightarrow 'value::\{pure,pcpo\}$
begin
 abbreviation *ESem-syn* ($\llbracket - \rrbracket$ - $[0,0]$ 110) **where** $\llbracket e \rrbracket_{\varrho} \equiv ESem \ e \cdot \varrho$
end

locale *has-ignore-fresh-ESem* = *has-ESem* +
 assumes *fv-supp*: $supp \ e = atom \ ' (fv \ e :: 'b \ set)$
 assumes *ESem-considers-fv*: $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} f | ' (fv \ e)$

end

5.3 HeapSemantics

theory *HeapSemantics*
imports *EvalHeap AList-Utills-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* ($\llbracket - \rrbracket$ - $[0,0]$ 110)
where *EvalHeapSem-syn* $\Gamma \ \varrho \equiv evalHeap \ \Gamma \ (\lambda \ e. \llbracket e \rrbracket_{\varrho})$

definition

$HSem :: ('var \times 'exp) list \Rightarrow ('var \Rightarrow 'value) \rightarrow ('var \Rightarrow 'value)$
where $HSem \Gamma = (\Lambda \varrho \cdot (\mu \varrho'. \varrho ++_{domA} \Gamma \llbracket \Gamma \rrbracket_{\varrho'}))$

abbreviation $HSem-syn (\{\!|\cdot|\!\} - [0,60] 60)$

where $\{\!|\Gamma|\!\} \varrho \equiv HSem \Gamma \cdot \varrho$

lemma $HSem-def'$: $\{\!|\Gamma|\!\} \varrho = (\mu \varrho'. \varrho ++_{domA} \Gamma \llbracket \Gamma \rrbracket_{\varrho'})$

$\langle proof \rangle$

5.3.2 Induction and other lemmas about $HSem$ **lemma** $HSem-ind$:

assumes $adm P$

assumes $P \perp$

assumes $step: \bigwedge \varrho'. P \varrho' \Longrightarrow P (\varrho ++_{domA} \Gamma \llbracket \Gamma \rrbracket_{\varrho'})$

shows $P (\{\!|\Gamma|\!\} \varrho)$

$\langle proof \rangle$

lemma $HSem-below$:

assumes $rho: \bigwedge x. x \notin domA h \Longrightarrow \varrho x \sqsubseteq r x$

assumes $h: \bigwedge x. x \in domA h \Longrightarrow \llbracket the (map-of h x) \rrbracket_r \sqsubseteq r x$

shows $\{\!|h|\!\} \varrho \sqsubseteq r$

$\langle proof \rangle$

lemma $HSem-bot-below$:

assumes $h: \bigwedge x. x \in domA h \Longrightarrow \llbracket the (map-of h x) \rrbracket_r \sqsubseteq r x$

shows $\{\!|h|\!\} \perp \sqsubseteq r$

$\langle proof \rangle$

lemma $HSem-bot-ind$:

assumes $adm P$

assumes $P \perp$

assumes $step: \bigwedge \varrho'. P \varrho' \Longrightarrow P (\llbracket \Gamma \rrbracket_{\varrho'})$

shows $P (\{\!|\Gamma|\!\} \perp)$

$\langle proof \rangle$

lemma $parallel-HSem-ind$:

assumes $adm (\lambda \varrho'. P (fst \varrho') (snd \varrho'))$

assumes $P \perp \perp$

assumes $step: \bigwedge y z. P y z \Longrightarrow$

$P (\varrho_1 ++_{domA} \Gamma_1 \llbracket \Gamma_1 \rrbracket y) (\varrho_2 ++_{domA} \Gamma_2 \llbracket \Gamma_2 \rrbracket z)$

shows $P (\{\!|\Gamma_1|\!\} \varrho_1) (\{\!|\Gamma_2|\!\} \varrho_2)$

$\langle proof \rangle$

lemma $HSem-eq$:

shows $\{\!|\Gamma|\!\} \varrho = \varrho ++_{domA} \Gamma \llbracket \Gamma \rrbracket_{\{\!|\Gamma|\!\} \varrho}$

$\langle proof \rangle$

lemma *HSem-bot-eq*:

shows $\{\Gamma\}^\perp = \llbracket \Gamma \rrbracket \{\Gamma\}^\perp$

<proof>

lemma *lookup-HSem-other*:

assumes $y \notin \text{dom} A \ h$

shows $(\{h\}^\varrho) y = \varrho y$

<proof>

lemma *lookup-HSem-heap*:

assumes $y \in \text{dom} A \ h$

shows $(\{h\}^\varrho) y = \llbracket \text{the } (\text{map-of } h \ y) \rrbracket \{h\}^\varrho$

<proof>

lemma *HSem-edom-subset*: $\text{edom } (\{\Gamma\}^\varrho) \subseteq \text{edom } \varrho \cup \text{dom} A \ \Gamma$

<proof>

lemma (*in* $-$) *env-restr-override-onI*: $-S2 \subseteq S \implies \text{env-restr } S \ \varrho1 \ ++_{S2} \ \varrho2 = \varrho1 \ ++_{S2} \ \varrho2$

<proof>

lemma *HSem-restr*:

$\{h\}^\varrho (f |' (- \text{dom} A \ h)) = \{h\}^\varrho$

<proof>

lemma *HSem-restr-cong*:

assumes $\varrho f |' (- \text{dom} A \ h) = \varrho' f |' (- \text{dom} A \ h)$

shows $\{h\}^\varrho = \{h\}^{\varrho'}$

<proof>

lemma *HSem-restr-cong-below*:

assumes $\varrho f |' (- \text{dom} A \ h) \sqsubseteq \varrho' f |' (- \text{dom} A \ h)$

shows $\{h\}^\varrho \sqsubseteq \{h\}^{\varrho'}$

<proof>

lemma *HSem-reorder*:

assumes $\text{map-of } \Gamma = \text{map-of } \Delta$

shows $\{\Gamma\}^\varrho = \{\Delta\}^\varrho$

<proof>

lemma *HSem-reorder-head*:

assumes $x \neq y$

shows $\{(x, e1) \# (y, e2) \# \Gamma\}^\varrho = \{(y, e2) \# (x, e1) \# \Gamma\}^\varrho$

<proof>

lemma *HSem-reorder-head-append*:

assumes $x \notin \text{dom} A \ \Gamma$

shows $\{(x, e) \# \Gamma @ \Delta\}^\varrho = \{\Gamma @ ((x, e) \# \Delta)\}^\varrho$

<proof>

lemma *env-restr-HSem*:
assumes $\text{dom}A \ \Gamma \cap S = \{\}$
shows $(\{\Gamma\} \rho) f |' S = \rho f |' S$
 $\langle \text{proof} \rangle$

lemma *env-restr-HSem-noop*:
assumes $\text{dom}A \ \Gamma \cap \text{edom} \ \rho = \{\}$
shows $(\{\Gamma\} \rho) f |' \text{edom} \ \rho = \rho$
 $\langle \text{proof} \rangle$

lemma *HSem-Nil[simp]*: $\{\{\}\} \rho = \rho$
 $\langle \text{proof} \rangle$

5.3.3 Substitution

lemma *HSem-subst-exp*:
assumes $\bigwedge \rho'. \llbracket e \rrbracket_{\rho'} = \llbracket e' \rrbracket_{\rho'}$
shows $\{\!(x, e) \# \Gamma\} \rho = \{\!(x, e') \# \Gamma\} \rho$
 $\langle \text{proof} \rangle$

lemma *HSem-subst-expr-below*:
assumes *below*: $\llbracket e1 \rrbracket \{\!(x, e2) \# \Gamma\} \rho \sqsubseteq \llbracket e2 \rrbracket \{\!(x, e2) \# \Gamma\} \rho$
shows $\{\!(x, e1) \# \Gamma\} \rho \sqsubseteq \{\!(x, e2) \# \Gamma\} \rho$
 $\langle \text{proof} \rangle$

lemma *HSem-subst-expr*:
assumes *below1*: $\llbracket e1 \rrbracket \{\!(x, e2) \# \Gamma\} \rho \sqsubseteq \llbracket e2 \rrbracket \{\!(x, e2) \# \Gamma\} \rho$
assumes *below2*: $\llbracket e2 \rrbracket \{\!(x, e1) \# \Gamma\} \rho \sqsubseteq \llbracket e1 \rrbracket \{\!(x, e1) \# \Gamma\} \rho$
shows $\{\!(x, e1) \# \Gamma\} \rho = \{\!(x, e2) \# \Gamma\} \rho$
 $\langle \text{proof} \rangle$

5.3.4 Re-calculating the semantics of the heap is idempotent

lemma *HSem-redo*:
shows $\{\Gamma\}(\{\Gamma \ @ \ \Delta\} \rho) f |' (\text{edom} \ \rho \cup \text{dom}A \ \Delta) = \{\Gamma \ @ \ \Delta\} \rho$ (is ?LHS = ?RHS)
 $\langle \text{proof} \rangle$

5.3.5 Iterative definition of the heap semantics

lemma *iterative-HSem*:
assumes $x \notin \text{dom}A \ \Gamma$
shows $\{\!(x, e) \# \Gamma\} \rho = (\mu \ \rho'. (\rho \ ++_{\text{dom}A \ \Gamma} (\{\Gamma\} \rho')))(x := \llbracket e \rrbracket_{\rho'})$
 $\langle \text{proof} \rangle$

lemma *iterative-HSem'*:
assumes $x \notin \text{dom}A \ \Gamma$
shows $(\mu \ \rho'. (\rho \ ++_{\text{dom}A \ \Gamma} (\{\Gamma\} \rho')))(x := \llbracket e \rrbracket_{\rho'})$
 $= (\mu \ \rho'. (\rho \ ++_{\text{dom}A \ \Gamma} (\{\Gamma\} \rho')))(x := \llbracket e \rrbracket_{\{\Gamma\} \rho'})$
 $\langle \text{proof} \rangle$

5.3.6 Fresh variables on the heap are irrelevant

lemma *HSem-ignores-fresh-restr'*:

assumes $fv \Gamma \subseteq S$

assumes $\bigwedge x \varrho. x \in dom A \Gamma \implies \llbracket the (map-of \Gamma x) \rrbracket_{\varrho} = \llbracket the (map-of \Gamma x) \rrbracket_{\varrho} f|' (fv (the (map-of \Gamma x)))$

shows $(\{\Gamma\}_{\varrho}) f|' S = \{\Gamma\}_{\varrho} f|' S$

<proof>

end

5.3.7 Freshness

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

lemma *ESem-fresh-cong*:

assumes $\varrho f|' (fv e) = \varrho' f|' (fv e)$

shows $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho'}$

<proof>

lemma *ESem-fresh-cong-subset*:

assumes $fv e \subseteq S$

assumes $\varrho f|' S = \varrho' f|' S$

shows $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho'}$

<proof>

lemma *ESem-fresh-cong-below*:

assumes $\varrho f|' (fv e) \sqsubseteq \varrho' f|' (fv e)$

shows $\llbracket e \rrbracket_{\varrho} \sqsubseteq \llbracket e \rrbracket_{\varrho'}$

<proof>

lemma *ESem-fresh-cong-below-subset*:

assumes $fv e \subseteq S$

assumes $\varrho f|' S \sqsubseteq \varrho' f|' S$

shows $\llbracket e \rrbracket_{\varrho} \sqsubseteq \llbracket e \rrbracket_{\varrho'}$

<proof>

lemma *ESem-ignores-fresh-restr*:

assumes $atom' S \#* e$

shows $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} f|' (- S)$

<proof>

lemma *ESem-ignores-fresh-restr'*:

assumes $atom' (edom \varrho - S) \#* e$

shows $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} f|' S$

<proof>

lemma *HSem-ignores-fresh-restr''*:

assumes $fv \Gamma \subseteq S$

shows $(\{\Gamma\}_{\varrho}) f|' S = \{\Gamma\}_{\varrho} f|' S$

<proof>

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

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

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

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

lemma *HSem-add-fresh':*
assumes $fresh: atom \ x \ \# \ \Gamma$
assumes $x \notin edom \ \varrho$
assumes $step: \bigwedge e \ \varrho'. e \in snd \ ' set \ \Gamma \implies \llbracket e \rrbracket_{\varrho'} = \llbracket e \rrbracket_{env-delete \ x \ \varrho'}$
shows $env-delete \ x \ (\{(x, e) \ \# \ \Gamma\}\varrho) = \{\Gamma\}\varrho$
 $\langle proof \rangle$

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

5.3.9 Mutual recursion with fresh variables

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

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

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

5.3.10 Parallel induction

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

```

assumes adm ( $\lambda \rho'. P$  ( $\text{fst } \rho'$ ) ( $\text{snd } \rho'$ ))
assumes  $P \perp \perp$ 
assumes  $\bigwedge y z. P y z \implies P (\rho \text{ ++ domA } h \text{ evalHeap } h (\lambda e. \text{ESem1 } e \cdot y)) (\rho' \text{ ++ domA } h \text{ evalHeap } h' (\lambda e. \text{ESem2 } e \cdot z))$ 
shows  $P (\text{has-ESem.HSem ESem1 } h \cdot \rho) (\text{has-ESem.HSem ESem2 } h' \cdot \rho')$ 
<proof>

```

5.3.11 Congruence rule

```

lemma HSem-cong[fundef-cong]:
   $\llbracket (\bigwedge e. e \in \text{snd } ' \text{ set heap2} \implies \text{ESem1 } e = \text{ESem2 } e); \text{heap1} = \text{heap2} \rrbracket$ 
   $\implies \text{has-ESem.HSem ESem1 heap1} = \text{has-ESem.HSem ESem2 heap2}$ 
<proof>

```

5.3.12 Equivariance of the heap semantics

```

lemma HSem-eqvt[eqvt]:
   $\pi \cdot \text{has-ESem.HSem ESem } \Gamma = \text{has-ESem.HSem } (\pi \cdot \text{ESem}) (\pi \cdot \Gamma)$ 
<proof>

```

end

5.4 AbstractDenotational

```

theory AbstractDenotational
imports HeapSemantics Terms
begin

```

5.4.1 The denotational semantics for expressions

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

```

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

```

nominal-function

```

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

```

| $ESem (scrut ? e1 : e2) = (\Lambda \varrho. tick \cdot ((B\text{-project} \cdot (ESem\ scrut \cdot \varrho)) \cdot (ESem\ e1 \cdot \varrho)) \cdot (ESem\ e2 \cdot \varrho))$
 <proof>

nominal-termination (in *semantic-domain*) (*no-eqvt*) <proof>

sublocale *has-ESem ESem* <proof>

notation *ESem-syn* ($\llbracket - \rrbracket_- [60,60] 60$)

notation *EvalHeapSem-syn* ($\llbracket - \rrbracket_- [0,0] 110$)

notation *HSem-syn* ($\{\!\!-\!\!\} [60,60] 60$)

abbreviation *AHSem-bot* ($\{\!\!-\!\!\} [60] 60$) **where** $\{\!\!-\!\!\} \equiv \{\!\!-\!\!\} \perp$

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'*: $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho f|' (fv\ e)}$
 <proof>

sublocale *has-ignore-fresh-ESem ESem*
 <proof>

5.5.2 Nicer equations for ESem, without freshness requirements

lemma *ESem-Lam[simp]*: $\llbracket Lam\ [x].\ e \rrbracket_{\varrho} = tick \cdot (Fn \cdot (\Lambda\ v.\ \llbracket e \rrbracket_{\varrho(x := v)}))$
 <proof>

declare *ESem.simps(1)[simp del]*

lemma *ESem-Let[simp]*: $\llbracket Let\ as\ body \rrbracket_{\varrho} = tick \cdot (\llbracket body \rrbracket_{\{\!\!-\!\!\} as \varrho})$
 <proof>

declare *ESem.simps(4)[simp del]*

5.5.3 Denotation of Substitution

lemma *ESem-subst-same*: $\varrho\ x = \varrho\ y \implies \llbracket e \rrbracket_{\varrho} = \llbracket e[x ::= y] \rrbracket_{\varrho}$
and

$\varrho\ x = \varrho\ y \implies (\llbracket as \rrbracket_{\varrho}) = \llbracket as[x::h=y] \rrbracket_{\varrho}$
 <proof>

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

end

end

5.6 Denotational

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

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

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

notation *ESem-syn* $(\llbracket - \rrbracket_{-} [60,60] 60)$
notation *EvalHeapSem-syn* $(\llbracket - \rrbracket_{-} [0,0] 110)$
notation *HSem-syn* $(\{\!\! \{-\}\!\! \}_{-} [60,60] 60)$
notation *AHSem-bot* $(\{\!\! \{-\}\!\! \}_{-} [60] 60)$

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

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

end

6 Resourced denotational domain

6.1 C

```
theory C
imports HOLCF Mono-Nat-Fun
begin
```

```
default-sort cpo
```

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

```
domain C = C (lazy C)
```

```
lemma below-C:  $x \sqsubseteq C \cdot x$ 
  <proof>
```

```
definition Cinf (C∞) where C∞ = fix C
```

```
lemma C-Cinf[simp]:  $C \cdot C^{\infty} = C^{\infty}$  <proof>
```

```
abbreviation Cpow (C-) where Cn ≡ iterate n C · ⊥
```

```
lemma C-below-C[simp]:  $(C^i \sqsubseteq C^j) \longleftrightarrow i \leq j$ 
  <proof>
```

```
lemma below-Cinf[simp]:  $r \sqsubseteq C^{\infty}$ 
  <proof>
```

```
lemma C-eq-Cinf[simp]:  $C^i \neq C^{\infty}$ 
  <proof>
```

```
lemma Cinf-eq-C[simp]:  $C^{\infty} = C \cdot r \longleftrightarrow C^{\infty} = r$ 
  <proof>
```

```
lemma C-eq-C[simp]:  $(C^i = C^j) \longleftrightarrow i = j$ 
  <proof>
```

```
lemma case-of-C-below:  $(\text{case } r \text{ of } C \cdot y \Rightarrow x) \sqsubseteq x$ 
  <proof>
```

```
lemma C-case-below:  $C \cdot \text{case} \cdot f \sqsubseteq f$ 
  <proof>
```

```
lemma C-case-bot[simp]:  $C \cdot \text{case} \cdot \perp = \perp$ 
  <proof>
```

```
lemma C-case-cong:
```

assumes $\bigwedge r'. r = C.r' \implies f.r' = g.r'$
shows $C\text{-case}.f.r = C\text{-case}.g.r$
 $\langle\text{proof}\rangle$

lemma $C\text{-cases}$:
obtains n **where** $r = C^n \mid r = C^\infty$
 $\langle\text{proof}\rangle$

lemma $C\text{-case}\text{-Cinf}[simp]$: $C\text{-case} . f . C^\infty = f . C^\infty$
 $\langle\text{proof}\rangle$

end

6.2 C-Meet

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

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

lemma $[simp]$: $C\text{-meet}.\perp.y = \perp \ C\text{-meet}.x.\perp = \perp \ \langle\text{proof}\rangle$

instance
 $\langle\text{proof}\rangle$
end

lemma $C\text{-meet-is-meet}$: $(z \sqsubseteq C\text{-meet}.x.y) = (z \sqsubseteq x \wedge z \sqsubseteq y)$
 $\langle\text{proof}\rangle$

instance $C :: \text{cont-binary-meet}$
 $\langle\text{proof}\rangle$

lemma $[simp]$: $C.r \sqcap r = r$
 $\langle\text{proof}\rangle$

lemma $[simp]$: $r \sqcap C.r = r$
 $\langle\text{proof}\rangle$

lemma $[simp]$: $C.r \sqcap C.r' = C.(r \sqcap r')$
 $\langle\text{proof}\rangle$

end

6.3 C-restr

```
theory C-restr
imports C C-Meet HOLCF-Utills
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* :: $(C \rightarrow 'a::pcpo) \Rightarrow C$ **where**
 $demand\ f = (if\ f \cdot C^\infty \neq \perp\ then\ C^{(LEAST\ n.\ f \cdot C^n \neq \perp)}\ else\ C^\infty)$

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

lemma *finite-resources-suffice*:
assumes $f \cdot C^\infty \neq \perp$
obtains n **where** $f \cdot C^n \neq \perp$
<proof>

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

lemma *more-resources-suffice*:
assumes $f \cdot r \neq \perp$ **and** $r \sqsubseteq r'$
shows $f \cdot r' \neq \perp$
<proof>

lemma *infinite-resources-suffice*:
shows $f \cdot r \neq \perp \implies f \cdot C^\infty \neq \perp$
<proof>

lemma *demand-suffices*:
assumes $f \cdot C^\infty \neq \perp$
shows $f \cdot (demand\ f) \neq \perp$
<proof>

lemma *not-bot-demand*:
 $f \cdot r \neq \perp \iff demand\ f \neq C^\infty \wedge demand\ f \sqsubseteq r$
<proof>

lemma *infinity-bot-demand*:
 $f \cdot C^\infty = \perp \iff demand\ f = C^\infty$
<proof>

lemma *demand-suffices'*:
assumes $demand\ f = C^n$
shows $f \cdot (demand\ f) \neq \perp$
<proof>

lemma *demand-Suc-Least*:

assumes *[simp]*: $f \cdot \perp = \perp$

assumes *demand* $f \neq C^\infty$

shows $\text{demand } f = C^{(\text{Suc } (\text{LEAST } n. f \cdot C^{\text{Suc } n} \neq \perp))}$

<proof>

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

<proof>

lemma *demand-contravariant*:

assumes $f \sqsubseteq g$

shows $\text{demand } g \sqsubseteq \text{demand } f$

<proof>

6.3.2 Restricting functions with domain C

fixrec *C-restr* :: $C \rightarrow (C \rightarrow 'a::\text{pcpo}) \rightarrow (C \rightarrow 'a)$

where $C\text{-restr}\cdot r \cdot f \cdot r' = (f \cdot (r \sqcap r'))$

abbreviation *C-restr-syn* :: $(C \rightarrow 'a::\text{pcpo}) \Rightarrow C \Rightarrow (C \rightarrow 'a)$ (*-|_* *[111,110]* 110)

where $f|_r \equiv C\text{-restr}\cdot r \cdot f$

lemma *[simp]*: $\perp|_r = \perp$ *<proof>*

lemma *[simp]*: $f \cdot \perp = \perp \implies f|_\perp = \perp$ *<proof>*

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

<proof>

lemma *C-restr-eqD*:

assumes $f|_r = g|_r$

assumes $r' \sqsubseteq r$

shows $f \cdot r' = g \cdot r'$

<proof>

lemma *C-restr-eq-lower*:

assumes $f|_r = g|_r$

assumes $r' \sqsubseteq r$

shows $f|_{r'} = g|_{r'}$

<proof>

lemma *C-restr-below**[intro, simp]*:

$x|_r \sqsubseteq x$

<proof>

lemma *C-restr-below-cong*:

$(\bigwedge r'. r' \sqsubseteq r \implies f \cdot r' \sqsubseteq g \cdot r') \implies f|_r \sqsubseteq g|_r$

<proof>

lemma *C-restr-cong*:

$$(\bigwedge r'. r' \sqsubseteq r \implies f \cdot r' = g \cdot r') \implies f|_r = g|_r$$

<proof>

lemma *C-restr-C-cong*:

$$(\bigwedge r'. r' \sqsubseteq r \implies f \cdot (C \cdot r') = g \cdot (C \cdot r')) \implies f \cdot \perp = g \cdot \perp \implies f|_{C \cdot r} = g|_{C \cdot r}$$

<proof>

lemma *C-restr-C-case[simp]*:

$$(C\text{-case} \cdot f)|_{C \cdot r} = C\text{-case} \cdot (f|_r)$$

<proof>

lemma *C-restr-bot-demand*:

assumes $C \cdot r \sqsubseteq \text{demand } f$

shows $f|_r = \perp$

<proof>

6.3.3 Restricting maps of C-ranged functions

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

abbreviation *env-C-restr-syn* :: $('var::type \Rightarrow (C \rightarrow 'a::pcpo)) \Rightarrow C \Rightarrow ('var \Rightarrow (C \rightarrow 'a))$ (
 $\text{-}|^\circ \text{-}$ [111,110] 110)

where $f|^\circ_r \equiv \text{env-C-restr} \cdot r \cdot f$

lemma *env-C-restr-upd[simp]*: $(\varrho(x := v))|^\circ_r = (\varrho|^\circ_r)(x := v|_r)$

<proof>

lemma *env-C-restr-lookup[simp]*: $(\varrho|^\circ_r) v = \varrho v|_r$

<proof>

lemma *env-C-restr-bot[simp]*: $\perp|^\circ_r = \perp$

<proof>

lemma *env-C-restr-restr-below[intro]*: $\varrho|^\circ_r \sqsubseteq \varrho$

<proof>

lemma *env-C-restr-env-C-restr[simp]*: $(v|^\circ_{r'})|^\circ_r = v|^\circ_{(r' \sqcap r)}$

<proof>

lemma *env-C-restr-cong*:

$$(\bigwedge x r'. r' \sqsubseteq r \implies f x \cdot r' = g x \cdot r') \implies f|^\circ_r = g|^\circ_r$$

<proof>

end

6.4 CValue

theory *CValue*

imports *C*

begin

domain *CValue*

= *CFn* (**lazy** (*C* → *CValue*) → (*C* → *CValue*))
 | *CB* (**lazy** *bool* *discr*)

fixrec *CFn-project* :: *CValue* → (*C* → *CValue*) → (*C* → *CValue*)

where *CFn-project*.(*CFn.f*).*v* = *f* · *v*

abbreviation *CFn-project-abbr* (**infix** ↓*CFn* 55)

where *f* ↓*CFn* *v* ≡ *CFn-project.f.v*

lemma *CFn-project-strict[simp]*:

⊥ ↓*CFn* *v* = ⊥
CB.b ↓*CFn* *v* = ⊥
 ⟨*proof*⟩

lemma *CB-below[simp]*: *CB.b* ⊆ *v* ↔ *v* = *CB.b*

⟨*proof*⟩

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 = ⊥

⟨*proof*⟩

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) ≠ ⊥)
 ⟨*proof*⟩

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*

⟨*proof*⟩

lemma *C-to-CValue-reach*: (⊔ *n*. *C-to-CValue-take* *n*.)*x* = *x*

⟨*proof*⟩

end

6.5 CValue-Nominal

```
theory CValue-Nominal
imports CValue Nominal-Utills Nominal-HOLCF
begin
```

```
instantiation C :: pure
begin
  definition p · (c::C) = c
  instance ⟨proof⟩
end
instance C :: pcpo-pt
  ⟨proof⟩
```

```
instantiation CValue :: pure
begin
  definition p · (v::CValue) = v
  instance
    ⟨proof⟩
end
```

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

end

6.6 ResourcedDenotational

```
theory ResourcedDenotational
imports Abstract-Denotational-Props CValue-Nominal C-restr
begin
```

```
type-synonym CEnv = var ⇒ (C → CValue)
```

```
interpretation semantic-domain
  Λ f . Λ r. CFn.(Λ v. (f·(v))|r)
  Λ x y. (Λ r. (x·r ↓ CFn y|r)·r)
  Λ b r. CB·b
  Λ scrut v1 v2 r. CB-project·(scrut·r)·(v1·r)·(v2·r)
  C-case⟨proof⟩
```

```
notation ESem-syn (N[[ - ]]- [60,60] 60)
notation EvalHeapSem-syn (N[[ - ]]- [0,0] 110)
notation HSem-syn (N{|-|}- [60,60] 60)
notation AHSem-bot (N{|-|}- [60] 60)
```

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

lemma *CESem-simps*:

$$\begin{aligned}
\mathcal{N} \llbracket \text{Lam } [x]. e \rrbracket_{\varrho} &= (\Lambda (C \cdot r). \text{CFn} \cdot (\Lambda v. (\mathcal{N} \llbracket e \rrbracket_{\varrho}(x := v)) | r)) \\
\mathcal{N} \llbracket \text{App } e x \rrbracket_{\varrho} &= (\Lambda (C \cdot r). ((\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r \downarrow \text{CFn } \varrho x | r) \cdot r) \\
\mathcal{N} \llbracket \text{Var } x \rrbracket_{\varrho} &= (\Lambda (C \cdot r). (\varrho x) \cdot r) \\
\mathcal{N} \llbracket \text{Bool } b \rrbracket_{\varrho} &= (\Lambda (C \cdot r). \text{CB} \cdot (\text{Discr } b)) \\
\mathcal{N} \llbracket (\text{scrut } ? e_1 : e_2) \rrbracket_{\varrho} &= (\Lambda (C \cdot r). \text{CB-project} \cdot ((\mathcal{N} \llbracket \text{scrut} \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_1 \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_2 \rrbracket_{\varrho}) \cdot r)) \\
\mathcal{N} \llbracket \text{Let as body} \rrbracket_{\varrho} &= (\Lambda (C \cdot r). (\mathcal{N} \llbracket \text{body} \rrbracket_{\mathcal{N} \{as\} \varrho}) \cdot r) \\
\langle \text{proof} \rangle
\end{aligned}$$

lemma *CESem-bot[simp]*: $(\mathcal{N} \llbracket e \rrbracket_{\sigma}) \cdot \perp = \perp$

$\langle \text{proof} \rangle$

lemma *CHSem-bot[simp]*: $(\mathcal{N} \{ \Gamma \} x) \cdot \perp = \perp$

$\langle \text{proof} \rangle$

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

lemma *CESem-simps-no-tick*:

$$\begin{aligned}
(\mathcal{N} \llbracket \text{Lam } [x]. e \rrbracket_{\varrho}) \cdot r &\sqsubseteq \text{CFn} \cdot (\Lambda v. (\mathcal{N} \llbracket e \rrbracket_{\varrho}(x := v)) | r) \\
(\mathcal{N} \llbracket \text{App } e x \rrbracket_{\varrho}) \cdot r &\sqsubseteq ((\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r \downarrow \text{CFn } \varrho x | r) \cdot r \\
\mathcal{N} \llbracket \text{Var } x \rrbracket_{\varrho} &\sqsubseteq \varrho x \\
(\mathcal{N} \llbracket (\text{scrut } ? e_1 : e_2) \rrbracket_{\varrho}) \cdot r &\sqsubseteq \text{CB-project} \cdot ((\mathcal{N} \llbracket \text{scrut} \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_1 \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_2 \rrbracket_{\varrho}) \cdot r) \\
\mathcal{N} \llbracket \text{Let as body} \rrbracket_{\varrho} &\sqsubseteq \mathcal{N} \llbracket \text{body} \rrbracket_{\mathcal{N} \{as\} \varrho} \\
\langle \text{proof} \rangle
\end{aligned}$$

lemma *CELam-no-restr*: $(\mathcal{N} \llbracket \text{Lam } [x]. e \rrbracket_{\varrho}) \cdot r \sqsubseteq \text{CFn} \cdot (\Lambda v. (\mathcal{N} \llbracket e \rrbracket_{\varrho}(x := v)))$

$\langle \text{proof} \rangle$

lemma *CEApp-no-restr*: $(\mathcal{N} \llbracket \text{App } e x \rrbracket_{\varrho}) \cdot r \sqsubseteq ((\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r \downarrow \text{CFn } \varrho x) \cdot r$

$\langle \text{proof} \rangle$

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  $fv(\Gamma, e) \subseteq set\ L \cup dom\ A\ \Gamma$ 
  shows  $\llbracket e \rrbracket_{\{\Gamma\}\varrho} = \llbracket v \rrbracket_{\{\Delta\}\varrho}$ 
  and  $(\{\Gamma\}\varrho) f|' dom\ A\ \Gamma = (\{\Delta\}\varrho) f|' dom\ A\ \Gamma$ 
  <proof>

end
```

7.2 CorrectnessResourced

```
theory CorrectnessResourced
imports ResourcedDenotational Launchbury
begin
```

```
theorem correctness:
  assumes  $\Gamma : e \Downarrow_L \Delta : z$ 
  and  $fv(\Gamma, e) \subseteq set\ L \cup dom\ A\ \Gamma$ 
  shows  $\mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\Gamma\}\varrho} \sqsubseteq \mathcal{N}\llbracket z \rrbracket_{\mathcal{N}\{\Delta\}\varrho}$  and  $(\mathcal{N}\{\Gamma\}\varrho) f|' dom\ A\ \Gamma \sqsubseteq (\mathcal{N}\{\Delta\}\varrho) f|' dom\ A\ \Gamma$ 
  <proof>
```

```
corollary correctness-empty-env:
```

```
  assumes  $\Gamma : e \Downarrow_L \Delta : z$ 
  and  $fv(\Gamma, e) \subseteq set\ L$ 
  shows  $\mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\Gamma\}} \sqsubseteq \mathcal{N}\llbracket z \rrbracket_{\mathcal{N}\{\Delta\}}$  and  $\mathcal{N}\{\Gamma\} \sqsubseteq \mathcal{N}\{\Delta\}$ 
  <proof>
```

```
end
```

8 Equivalence of the denotational semantics

8.1 ValueSimilarity

```
theory ValueSimilarity
imports Value CValue Pointwise
begin
```

This theory formalizes Section 3 of [SGHHOM11]. Their domain D is our type $Value$, their domain E is our type $CValue$ and A corresponds to $C \rightarrow CValue$.

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 $\phi_n^E: E_n \rightarrow E$ is the identity here.

The projections correspond to the take-functions generated by the HOLCF package:

$$\begin{aligned} \psi_n^E: E \rightarrow E_n & \text{ corresponds to } & CValue\text{-take}::nat \Rightarrow CValue \rightarrow CValue \\ \psi_n^A: A \rightarrow A_n & \text{ corresponds to } & C\text{-to-}CValue\text{-take}::nat \Rightarrow (C \rightarrow CValue) \rightarrow C \rightarrow CValue \\ \psi_n^D: D \rightarrow D_n & \text{ corresponds to } & Value\text{-take}::nat \Rightarrow Value \rightarrow Value. \end{aligned}$$

The syntactic overloading of $e(a)(c)$ to mean either $\text{Ap}_{E_n}^\perp$ or AP_E^\perp turns into our non-overloaded $\downarrow CFn \text{ } :: CValue \Rightarrow (C \rightarrow CValue) \Rightarrow C \rightarrow CValue$.

To have our presentation closer to [SGHHOM11], we introduce some notation:

```
notation Value-take ( $\psi^D$ .)
notation C-to-CValue-take ( $\psi^A$ .)
notation CValue-take ( $\psi^E$ .)
```

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): $\bigwedge n d d'. \psi_n^D.(d \downarrow Fn d') = \psi_{Suc\ n}^D.d \downarrow Fn \psi_n^D.d'$

shows False

<proof>

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*⟩

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 ψ^D_n on the left hand side):

lemma *Abramsky 4,3,5 (1)*:
 $\psi^D_n \cdot (d \downarrow Fn \ \psi^D_n \cdot d') = \psi^D_{Suc \ n} \cdot d \downarrow Fn \ \psi^D_n \cdot d'$
 ⟨*proof*⟩

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 ψ^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
 $x = \perp \ y = \perp \mid$
 $f \ \mathbf{where} \ x = Fn \cdot f \ y = \perp \mid$
 $g \ \mathbf{where} \ x = \perp \ y = CFn \cdot g \mid$
 $f \ g \ \mathbf{where} \ x = Fn \cdot f \ y = CFn \cdot g \mid$
 $b_1 \ \mathbf{where} \ x = B \cdot (Discr \ b_1) \ y = \perp \mid$
 $b_1 \ g \ \mathbf{where} \ x = B \cdot (Discr \ b_1) \ y = CFn \cdot g \mid$
 $b_1 \ b_2 \ \mathbf{where} \ x = B \cdot (Discr \ b_1) \ y = CB \cdot (Discr \ b_2) \mid$
 $f \ b_2 \ \mathbf{where} \ x = Fn \cdot f \ y = CB \cdot (Discr \ b_2) \mid$
 $b_2 \ \mathbf{where} \ x = \perp \ y = CB \cdot (Discr \ b_2)$
 ⟨*proof*⟩

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*⟩

8.1.3 Restricted similarity is defined recursively

The base case

inductive *similar'-base* :: *Value* \Rightarrow *CValue* \Rightarrow *bool* **where**
bot-similar'-base[*simp,intro*]: *similar'-base* $\perp \ \perp$

inductive-cases [*elim!*]:
similar'-base $x \ y$

The inductive case

inductive *similar'-step* :: (Value \Rightarrow CValue \Rightarrow bool) \Rightarrow Value \Rightarrow CValue \Rightarrow bool **for** *s* **where**
bot-similar'-step[intro!]: *similar'-step* *s* \perp \perp |
bool-similar'-step[intro]: *similar'-step* *s* (B·b) (CB·b) |
Fun-similar'-step[intro]: ($\bigwedge x y . s x (y \cdot C^\infty) \Longrightarrow s (f \cdot x) (g \cdot y \cdot C^\infty)$) \Longrightarrow *similar'-step* *s* (Fn·f)
(CFn·g)

inductive-cases [elim!]:
similar'-step *s* $x \perp$
similar'-step *s* $\perp y$
similar'-step *s* (B·f) (CB·g)
similar'-step *s* (Fn·f) (CFn·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* (- \triangleleft - [50,50,50] 50)
where *similar'-syn* *x n y* \equiv *similar'* *n x y*

lemma *similar'-botI*[intro!,simp]: $\perp \triangleleft_n \perp$
⟨proof⟩

lemma *similar'-FnI*[intro]:
assumes $\bigwedge x y . x \triangleleft_n y \cdot C^\infty \Longrightarrow f \cdot x \triangleleft_n g \cdot y \cdot C^\infty$
shows $Fn \cdot f \triangleleft_{Suc\ n} CFn \cdot g$
⟨proof⟩

lemma *similar'-FnE*[elim!]:
assumes $Fn \cdot f \triangleleft_{Suc\ n} CFn \cdot g$
assumes ($\bigwedge x y . x \triangleleft_n y \cdot C^\infty \Longrightarrow f \cdot x \triangleleft_n g \cdot y \cdot C^\infty$) $\Longrightarrow P$
shows *P*
⟨proof⟩

lemma *bot-or-not-bot'*:
 $x \triangleleft_n y \Longrightarrow (x = \perp \longleftrightarrow y = \perp)$
⟨proof⟩

lemma *similar'-bot*[elim-format, elim!]:
 $\perp \triangleleft_n x \Longrightarrow x = \perp$
 $y \triangleleft_n \perp \Longrightarrow y = \perp$
⟨proof⟩

lemma *similar'-typed*[simp]:
 $\neg B \cdot b \triangleleft_n CFn \cdot g$
 $\neg Fn \cdot f \triangleleft_n CB \cdot b$

<proof>

lemma *similar'-bool[simp]*:

$B \cdot b_1 \triangleleft_{Suc\ n} CB \cdot b_2 \iff b_1 = b_2$

<proof>

8.1.4 Moving up and down the similarity relations

These correspond to Lemma 7 in [SGHHOM11].

lemma *similar'-down*: $d \triangleleft_{Suc\ n} e \implies \psi^D_n \cdot d \triangleleft_n \psi^E_n \cdot e$

and *similar'-up*: $d \triangleleft_n e \implies \psi^D_n \cdot d \triangleleft_{Suc\ n} \psi^E_n \cdot e$

<proof>

A generalisation of the above, doing multiple steps at once.

lemma *similar'-up-le*: $n \leq m \implies \psi^D_n \cdot d \triangleleft_n \psi^E_n \cdot e \implies \psi^D_n \cdot d \triangleleft_m \psi^E_n \cdot e$

<proof>

lemma *similar'-down-le*: $n \leq m \implies \psi^D_m \cdot d \triangleleft_m \psi^E_m \cdot e \implies \psi^D_n \cdot d \triangleleft_n \psi^E_n \cdot e$

<proof>

lemma *similar'-take*: $d \triangleleft_n e \implies \psi^D_n \cdot d \triangleleft_n \psi^E_n \cdot e$

<proof>

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*: $adm\ (\lambda x. similar'\text{-base}\ (fst\ x)\ (snd\ x))$

<proof>

lemma *similar'-step-adm*:

assumes $adm\ (\lambda x. s\ (fst\ x)\ (snd\ x))$

shows $adm\ (\lambda x. similar'\text{-step}\ s\ (fst\ x)\ (snd\ x))$

<proof>

lemma *similar'-adm*: $adm\ (\lambda x. fst\ x \triangleleft_n snd\ x)$

<proof>

lemma *similar'-admI*: $cont\ f \implies cont\ g \implies adm\ (\lambda x. f\ x \triangleleft_n g\ x)$

<proof>

8.1.6 The real similarity relation

This is the goal of the theory: A relation between *Value* and *CValue*.

definition *similar* :: $Value \Rightarrow CValue \Rightarrow bool$ (**infix** \triangleleft 50) **where**

$$x \triangleleft y \iff (\forall n. \psi^D_n \cdot x \triangleleft_n \psi^E_n \cdot y)$$

lemma *similarI*:

$$(\bigwedge n. \psi^D_n \cdot x \triangleleft_n \psi^E_n \cdot y) \implies x \triangleleft y$$

<proof>

lemma *similarE*:

$$x \triangleleft y \implies \psi^D_n \cdot x \triangleleft_n \psi^E_n \cdot y$$

<proof>

lemma *similar-bot[simp]*: $\perp \triangleleft \perp$ *<proof>*

lemma *similar-bool[simp]*: $B \cdot b \triangleleft CB \cdot b$
<proof>

lemma [*elim-format, elim!*]: $x \triangleleft \perp \implies x = \perp$
<proof>

lemma [*elim-format, elim!*]: $x \triangleleft CB \cdot b \implies x = B \cdot b$
<proof>

lemma [*elim-format, elim!*]: $\perp \triangleleft y \implies y = \perp$
<proof>

lemma [*elim-format, elim!*]: $B \cdot b \triangleleft y \implies y = CB \cdot b$
<proof>

lemma *take-similar'-similar*:

assumes $x \triangleleft_n y$

shows $\psi^D_n \cdot x \triangleleft \psi^E_n \cdot y$

<proof>

lemma *bot-or-not-bot*:

$$x \triangleleft y \implies (x = \perp \iff y = \perp)$$

<proof>

lemma *bool-or-not-bool*:

$$x \triangleleft y \implies (x = B \cdot b \iff y = CB \cdot b)$$

<proof>

lemma *similar-bot-cases[consumes 1, case-names bot bool Fn]*:

assumes $x \triangleleft y$

obtains $x = \perp \ y = \perp \mid$

b **where** $x = B \cdot (Discr\ b) \ y = CB \cdot (Discr\ b) \mid$

$f\ g$ **where** $x = Fn \cdot f \ y = CFn \cdot g$

<proof>

lemma *similar-adm*: $adm\ (\lambda x. fst\ x \triangleleft snd\ x)$

<proof>

lemma *similar-admI*: $cont\ f \implies cont\ g \implies adm\ (\lambda x. f\ x \triangleleft g\ x)$
 ⟨proof⟩

Having constructed the relation we can now show that it indeed is the desired relation, relating \perp with \perp and functions with functions, if they take related arguments to related values. This corresponds to Proposition 9 in [SGHHOM11].

lemma *similar-nice-def*: $x \triangleleft y \iff (x = \perp \wedge y = \perp \vee (\exists b. x = B.(Discr\ b) \wedge y = CB.(Discr\ b)) \vee (\exists f\ g. x = Fn.f \wedge y = CFn.g \wedge (\forall a\ b. a \triangleleft b.C^\infty \implies f.a \triangleleft g.b.C^\infty)))$
 (is ?L \iff ?R)
 ⟨proof⟩

lemma *similar-FnI[intro]*:
 assumes $\bigwedge x\ y. x \triangleleft y.C^\infty \implies f.x \triangleleft g.y.C^\infty$
 shows $Fn.f \triangleleft CFn.g$
 ⟨proof⟩

lemma *similar-FnD[elim!]*:
 assumes $Fn.f \triangleleft CFn.g$
 assumes $x \triangleleft y.C^\infty$
 shows $f.x \triangleleft g.y.C^\infty$
 ⟨proof⟩

lemma *similar-FnE[elim]*:
 assumes $Fn.f \triangleleft CFn.g$
 assumes $(\bigwedge x\ y. x \triangleleft y.C^\infty \implies f.x \triangleleft g.y.C^\infty) \implies P$
 shows P
 ⟨proof⟩

8.1.7 The similarity relation lifted pointwise to functions.

abbreviation *fun-similar* :: $('a::type \Rightarrow Value) \Rightarrow ('a \Rightarrow (C \rightarrow CValue)) \Rightarrow bool$ (infix \triangleleft^* 50) **where**
 $fun-similar \equiv pointwise\ (\lambda x\ y. x \triangleleft y.C^\infty)$

lemma *fun-similar-fmap-bottom[simp]*: $\perp \triangleleft^* \perp$
 ⟨proof⟩

lemma *fun-similarE[elim]*:
 assumes $m \triangleleft^* m'$
 assumes $(\bigwedge x. (m\ x) \triangleleft (m'\ x).C^\infty) \implies Q$
 shows Q
 ⟨proof⟩

end

8.2 Denotational-Related

theory *Denotational-Related*

imports *Denotational ResourcedDenotational ValueSimilarity*

begin

Given the similarity relation it is straight-forward to prove that the standard and the re-sourced denotational semantics produce similar results. (Theorem 10 in [SGHHOM11]).

theorem *denotational-semantics-similar*:

assumes $\rho \triangleleft^* \sigma$

shows $\llbracket e \rrbracket_\rho \triangleleft (\mathcal{N} \llbracket e \rrbracket_\sigma) \cdot C^\infty$

<proof>

corollary *evalHeap-similar*:

$\bigwedge y z. y \triangleleft^* z \implies \llbracket \Gamma \rrbracket_y \triangleleft^* \mathcal{N} \llbracket \Gamma \rrbracket_z$

<proof>

theorem *heaps-similar*: $\{\Gamma\} \triangleleft^* \mathcal{N} \{\Gamma\}$

<proof>

end

9 Adequacy

9.1 ResourcedAdequacy

theory *ResourcedAdequacy*

imports *ResourcedDenotational Launchbury AList–Utils CorrectnessResourced*
begin

lemma *demand-not-0*: $\text{demand } (\mathcal{N} \llbracket e \rrbracket_{\rho}) \neq \perp$
<proof>

The semantics of an expression, given only r resources, will only use values from the environment with less resources.

lemma *restr-can-restrict-env*: $(\mathcal{N} \llbracket e \rrbracket_{\rho})|_{C \cdot r} = (\mathcal{N} \llbracket e \rrbracket_{\rho|^\circ r})|_{C \cdot r}$
<proof>

lemma *can-restrict-env*:
 $(\mathcal{N} \llbracket e \rrbracket_{\rho}) \cdot (C \cdot r) = (\mathcal{N} \llbracket e \rrbracket_{\rho|^\circ r}) \cdot (C \cdot r)$
<proof>

When an expression e terminates, then we can remove such an expression from the heap and it still terminates. This is the crucial trick to handle black-holing in the resourced semantics.

lemma *add-BH*:
assumes *map-of* $\Gamma x = \text{Some } e$
assumes $(\mathcal{N} \llbracket e \rrbracket_{\mathcal{N} \setminus \{ \Gamma \}}) \cdot r' \neq \perp$
shows $(\mathcal{N} \llbracket e \rrbracket_{\mathcal{N} \setminus \{ \text{delete } x \Gamma \}}) \cdot r' \neq \perp$
<proof>

The semantics is continuous, so we can apply induction here:

lemma *resourced-adequacy*:
assumes $(\mathcal{N} \llbracket e \rrbracket_{\mathcal{N} \setminus \{ \Gamma \}}) \cdot r \neq \perp$
shows $\exists \Delta v. \Gamma : e \Downarrow_{\mathcal{S}} \Delta : v$
<proof>

end

9.2 Adequacy

theory *Adequacy*

imports *ResourcedAdequacy Denotational–Related*
begin

theorem *adequacy*:
assumes $\llbracket e \rrbracket_{\Gamma} \neq \perp$

shows $\exists \Delta v. \Gamma : e \Downarrow_S \Delta : v$
<proof>

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.