LNgen: Tool Support for Locally Nameless Representations MS-CIS-10-24

Brian Aydemir
University of Pennsylvania
baydemir@cis.upenn.edu

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

June 2010

Abstract

Given the complexity of the metatheoretic reasoning about current programming languages and their type systems, techniques for mechanical formalization and checking of such metatheory have received much recent attention. In previous work, we advocated a combination of locally nameless representation and cofinite quantification as a lightweight style for carrying out such formalizations in the Coq proof assistant. As part of the presentation of that methodology, we described a number of operations associated with variable binding and listed a number of properties, called "infrastructure lemmas", about those operations that needed to be shown. The proofs of these infrastructure lemmas are straightforward but tedious.

In this work, we present LNgen, a prototype tool for automatically generating statements and proofs of infrastructure lemmas from Ott language specifications. Furthermore, the tool also generates a recursion scheme for defining functions over syntax, which was not available in our previous work. LNgen works in concert with Ott to effectively alleviate much of the tedium of working with locally nameless syntax. For the case of untyped lambda terms, we show that the combined output from the two tools is sound and complete, with LNgen automatically proving many of the key lemmas. We prove the soundness of our representation with respect to a fully concrete representation, and we argue that the representation is complete—that we generate the right set of lemmas—with respect to Gordon and Melham's "Five Axioms of Alpha-Conversion."

1 Introduction

Mechanical formalizations of programming languages have received much recent attention. One question that is foremost in any mechanization is the treatment of binding. Many tools exist to aid in this practice—Abella [1], Hybrid [2], Lambda Tamer [3], Nominal Isabelle [4], Twelf [5]—as well as many representation techniques—de Bruijn indices [6], higher-order abstract syntax (HOAS) [7], locally named [8], locally nameless [9], weak HOAS [10], etc.

As a programming language designer, how should we compare these methodologies? What tools should we use? The PoplMark challenge [11] laid out a number of criteria, which we have come to interpret with respect to existing technologies, for evaluating potential answers:

1. *Transparency.* Reasoning should be similar to that done with pencil and paper. For example, de Bruijn indices are not transparent. Metatheory involving them often includes many lemmas about shifting—lemmas that have no correspondence to pencil and paper proofs.

2. *Logical expressivity.* There should be minimal restriction on the logic that we use for formal developments. For example, the models of Nominal Logic [12] require that all definable relations be equivariant. To allow similar reasoning in higher-order logic, where this is not the case, Nominal Isabelle must require equivariance proofs (many of which can be provided automatically).

3. *Traction.* The strategy should draw on the strengths of the proof assistant. For example, in previous work [13], we explored nominal reasoning in Coq by defining an interface which specified the constructors of a nominal datatype, as well as an induction principal and recursion scheme for that datatype. We chose not to pursue that line of work because the interface, while usable, prevented users from taking advantage of Coq's built-in features. Utilizing distinctness and injectivity of datatype constructors, reasoning by induction, and defining functions by recursion all required the explicit use of special theorems and combinators. Furthermore, functions defined by the recursion combinator would not reduce by Coq's definitional equality—we had to use explicit rewriting.

From these criteria, we draw the following conclusions: We want a representation that involves reasoning about variable names, not indices, because that is the most transparent. We want to use this representation in a general purpose logic, such as higher-order logic or the Calculus of Inductive Constructions (CIC), but we want to automate as much of the tedious machinery as possible. And we want our representation of syntax to use what proof assistants are good at: specifying inductive datatypes and generating their associated induction principles and recursion schemes.

In previous work [14], we proposed a completely manual scheme for reasoning about binding structure based on locally nameless representations and defining inference rules with cofinite quantification. We described a number of operations associated with variable binding (free variable calculation, index substitution, free variable substitution, and free variable closing) and listed a number of properties, called "infrastructure lemmas", about those operations that needed to be shown. This strategy is lightweight in that the definitions of the operations are simple structural recursions, so proofs of their properties are straightforward. We have successfully used this strategy in our own developments and know of its use by others—for example, by Jia et al. [15], Pratikakis et al. [16], Rossberg et al. [17], and many more.

However, our previous work did not fully explain its own success. Why were the "infrastructure lemmas" the right set of lemmas to show? Would future formalizations require still more lemmas? Furthermore, if the proofs of the infrastructure lemmas are so straightforward and mechanical, should it not be possible to automatically generate those lemma statements and their proofs?

In this paper, we describe a prototype tool, LNgen, that we have developed for exactly this last purpose. LNgen uses the same input language as Ott [18], a tool for translating language specifications written in an intuitive syntax into output for LaTeX and proof assistants. While Ott generates locally nameless definitions—datatypes for syntax and relations, functions to calculate free variables and substitutions—from the specification, LNgen provides recursion schemes for defining functions over syntax and a large collection of infrastructure lemmas. LNgen automates much of the tedium associated with the locally nameless style, even in our streamlined style, by allowing users to focus on the more interesting aspects of their developments instead of on infrastructure lemmas. In Sec. 2, we describe in additional detail the input to and output from LNgen, highlighting the important properties that are automatically proved.

Following the overview of LNgen, we discuss soundness (Sec. 3) and completeness (Sec. 4) in the particular case of the untyped lambda calculus. For soundness, we prove that the locally nameless definition generated by Ott is adequate with respect to fully concrete terms identified up to alpha equivalence. The lemmas proved by LNgen provide many of the key lemmas required in this proof. For completeness, we prove that even though we use a locally nameless representation, the lemmas generated by LNgen are enough to shield users from the de Bruijn indices used to represent bound variables. Specifically, we give a model of Gordon and Melham's "Five Axioms of Alpha-Conversion" [19]. Although we think that the output of Ott and LNgen is more convenient to work with than the five axioms, we can implement these five axioms in an extremely straightforward manner, by using the lemmas proved by LNgen and without reasoning about de Bruijn indices or by induction on syntax.

We and others have experience with using LNgen in significant developments. Section 5 gives an overview of the case studies. Our experiences suggest that this tool has the advantages of code generators without the drawbacks of generating executable code. In particular, the output of LNgen is straightforward for programmers to effectively understand (definitions and lemma statements must be comprehended, but proofs do not) and robust to change (lemma statements do not change significantly as the language is modified).

We conclude the paper with related work (Sec. 6), and our conclusions and future work (Sec. 7).

```
metavar expvar, x, y, z ::=
  {{ repr-locally-nameless }}

grammar

exp, e, f, g :: '' ::=
  | x       :: :: var
  | e1 e2   :: :: app
  | \ x . e :: :: abs
    (+ bind x in e +)

substitutions
  single e x :: subst

freevars
  e x :: fv
```

```
Definition expvar := var.

Inductive exp : Set :=
  | var_b : nat -> exp
  | var_f : expvar -> exp
  | app : exp -> exp -> exp.
  | abs : exp -> exp
```

Figure 1: Input file (left) and output Coq datatype (right) for lambda terms

2 The LNgen Tool

LNgen is a prototype tool for generating locally nameless definitions and infrastructure for the Coq proof assistant. While LNgen is still under active development, the current version is available and has been used for significant developments.[1] LNgen relies on Ott [18] to generate the core locally nameless definitions for a language. It then generates additional definitions and lemmas that are often needed in developments—the main benefit that it provides to users over using Ott alone.

The input language for LNgen is a proper subset of the Ott specification language. Figure 1 shows an example input file for untyped lambda terms. The syntax is intended to mimic what one might write informally. Ott is specifically designed for specifying programming languages in a manner that is both convenient for people and machines, e.g., proof assistants. Thus, Ott is a natural starting point for the input language to LNgen. We can take advantage of the work that has gone into the design of Ott, not require users to learn a new specification language, and allow our tool to work in parallel with Ott, relying on Ott for the generation of some of the Coq definitions as well as LaTeX output.

Below, we use the example to give a brief overview of the subset of Ott that LNgen supports; a detailed description of the Ott language can be found elsewhere [20]. The first part of an input file for LNgen consists of a list of `metavar` declarations. Each declaration defines a new type for object language variables—LNgen and Ott define binding and substitution for these variables. In Fig. 1, the text `repr-locally-nameless` indicates that binding should be represented using a locally nameless encoding. (Ott can also output definitions using a concrete representation of binding.) The second part, the `grammar`, consists of a list of context-free grammar definitions for nonterminals. Each declaration defines a new, inductively defined type for object-language abstract syntax trees. Binding specifications may be attached to each constructor. For example in the `abs` constructor, the metavariable `x` is a binding occurrence in the nonterminal `e`. The third part follows the `substitutions` keyword and indicates that functions for substituting for free variables should be generated. The final part follows the `freevars` keyword and indicates that functions for calculating free variables should be generated. Anything else in the file is ignored by LNgen but may be processed by Ott, e.g., specifications of inductively defined relations.

2.1 Generated Definitions

Figure 1 also shows the output representation that Ott produces for the untyped lambda terms. Metavariables are implemented by the type `var`, which is provided by our metatheory library.[2] The constructor names

[1]LNgen is available from `http://www.cis.upenn.edu/~baydemir/`.
[2]The metatheory library is included with LNgen and also available from `http://www.plclub.org/metalib/`.

for the syntactic forms are determined by the input file, except the constructors for free and bound variables, where `_f` and `_b` are appended to the specified name. The variable in the `abs` constructor disappears because the binding specification indicates that this is a binding constructor.

Figure 2 lists the basic operations and predicates generated from the input in Fig. 1. For accessibility and brevity, we use mathematical notation instead of listing the Coq output directly. Ott generated the definitions of fv and subst; LNgen generated everything else. In general, the output follows our previously described style for working with locally nameless representations [14]. The operations include calculating the free variables of an expression (fv), substituting for an index (open_i), replacing a free variable with an index (close_i), and substituting for a free variable (subst). Note that close allows us to construct a concrete expression without explicitly referring to indices. Using $\mathsf{lam}\,x$ as an abbreviation for $\mathsf{abs} \circ \mathsf{close}\,x$, we can transparently write $\lambda x.\lambda y.\lambda z.z(xy)$ as

$$\mathsf{lam}\,x\,(\mathsf{lam}\,y\,(\mathsf{lam}\,z\,(\mathsf{app}\,(\mathsf{var\_f}\,z)\,(\mathsf{app}\,(\mathsf{var\_f}\,x)\,(\mathsf{var\_f}\,y))))) \,.$$

Note also that the versions of open_i and close_i presented here are derived from those of Pollack [9] and are slightly more general than that of our previous work—they may initially be called with an index other than zero. Previously, we promoted the absolute simplest definitions to make working by hand easy. Here, we have tool support, so it makes little difference if these definitions are more complicated. If anything, they are actually *easier* for LNgen to work with because they require tracking fewer invariants.

The final definitions in Fig. 2 give the constructors for the inductively defined lc and $\mathsf{lc\_set}$ predicates, which hold for *locally closed* lambda terms—those with no unresolved de Bruijn indices. Only expressions that satisfy these predicates correspond to lambda calculus terms. The only difference between the two predicates is that the former is in Prop and the latter is in Set; their definitions are otherwise identical. Because of Coq's distinction between Prop and Set, their uses are not. An object of type $\mathsf{lc}\,e$ is treated as a proof and may be analyzed only to produce another proof; an object of type $\mathsf{lc\_set}\,e$ may be analyzed freely. The inductive definition of lc provides an induction principle for reasoning about expressions, while the inductive definition of $\mathsf{lc\_set}$ provides a recursion scheme for defining functions over expressions. The induction principle and recursion scheme are both shown in Fig. 3.

Our treatment of local closure departs from our previous work in that we previously did not provide $\mathsf{lc\_set}$ and the recursion scheme that comes with it. We can use the scheme, for example, to define a function to perform parallel β-reduction on lambda terms:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{beta} = \mathsf{lc_set_rec}\,(\lambda\,_\,.\,\mathsf{exp})\,\mathit{fvar}\,\mathit{fapp}\,\mathit{fabs}\ \text{where}} \\
\quad \mathit{fvar}\,x & = & \mathsf{var_f}\,x \\
\quad \mathit{fapp}\,_\,_\,_\,(\mathsf{abs}\,e_1')\,_\,e_2' & = & \mathsf{open}\,e_2'\,e_1' \\
\quad \mathit{fapp}\,_\,_\,_\,e_1'\,_\,e_2' & = & \mathsf{app}\,e_1'\,e_2' \\
\quad \mathit{fabs}\,e_1\,_\,f' & = & \mathsf{abs}\,(\mathsf{close}\,x\,(f'\,x))\ \text{for some}\ x \notin \mathsf{fv}\,e_1
\end{array}$$

(In Coq, one would use this recursion scheme via `Fixpoint`, writing the function more naturally using explicit pattern matching on the local closure proof, and explicit recursive calls.) In the variable case, *beta* simply returns that variable. In the application case, the result of reducing the first component is examined: if it is an abstraction ($\mathsf{abs}\,e_1'$), *beta* substitutes the reduced second component e_2' for the first index in the body of the abstraction. Otherwise, reduction continues into both components of the application. In the abstraction case, *beta* reduces the body of the abstraction by picking a fresh variable to give to f'. This argument to *fabs* is a function that, when given a name for the variable bound at this location, computes the result of *beta* for the body of the abstraction using that name. After this recursive call, the branch removes that fresh variable from the result with close and creates a new abstraction.

In another departure from our previous work, neither lc nor $\mathsf{lc\_set}$ uses cofinite quantification. Instead, both use "universal" quantification in the abs case, by requiring that the premise hold for all names. This choice results in the strongest possible induction principle and recursion scheme. For lc, LNgen generates as a lemma an "existential" form of the $\mathsf{lc\_abs}$ constructor (lemma *lc-abs-exists* in Fig. 4) that requires showing the premise for only one name. This lemma provides the easiest to use introduction principle for proving $\mathsf{lc}\,(\mathsf{abs}\,e)$. This style of using a "universal" and an "existential" rule is based on the style of McKinna and Pollack [8]. While cofinite quantification is a good compromise between these two extremes when doing everything by hand, with tool support, it makes sense to provide these stronger principles. Using universal quantification also allows us to prove the uniqueness of lc proofs (lemma *lc-unique* in Fig. 4).

$$
\begin{array}{lcll}
\mathsf{fv} & : & \mathsf{exp} \rightarrow \mathsf{expvarset} & \\
\mathsf{fv}\,(\mathsf{var\_f}\ x) & = & \{\, x \,\} & \\
\mathsf{fv}\,(\mathsf{var\_b}\ i) & = & \emptyset & \\
\mathsf{fv}\,(\mathsf{abs}\ e_1) & = & \mathsf{fv}\ e_1 & \\
\mathsf{fv}\,(\mathsf{app}\ e_1\ e_2) & = & \mathsf{fv}\ e_1 \cup \mathsf{fv}\ e_2 & \\
\\
\mathsf{open}_i & : & \mathsf{nat} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp} & \\
\mathsf{open}_i\ e\,(\mathsf{var\_b}\ i_1) & = & (\mathsf{var\_b}\ i_1) & \text{when } i_1 < i \\
\mathsf{open}_i\ e\,(\mathsf{var\_b}\ i_1) & = & e & \text{when } i_1 = i \\
\mathsf{open}_i\ e\,(\mathsf{var\_b}\ i_1) & = & (\mathsf{var\_b}\,(i_1 - 1)) & \text{when } i_1 > i \\
\mathsf{open}_i\ e\,(\mathsf{var\_f}\ x) & = & \mathsf{var\_f}\ x & \\
\mathsf{open}_i\ e\,(\mathsf{abs}\ e_1) & = & \mathsf{abs}\,(\mathsf{open}_{(i+1)}\ e\ e_1) & \\
\mathsf{open}_i\ e\,(\mathsf{app}\ e_1\ e_2) & = & \mathsf{app}\,(\mathsf{open}_i\ e\ e_1)\,(\mathsf{open}_i\ e\ e_2) & \\
\\
\mathsf{open}\ e_1\ e_2 & = & \mathsf{open}_0\ e_1\ e_2 & \\
\\
\mathsf{close}_i & : & \mathsf{nat} \rightarrow \mathsf{expvar} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp} & \\
\mathsf{close}_i\ x\,(\mathsf{var\_b}\ i_1) & = & \mathsf{var\_b}\ i_1 & \text{when } i_1 < i \\
\mathsf{close}_i\ x\,(\mathsf{var\_b}\ i_1) & = & \mathsf{var\_b}\,(1 + i_1) & \text{when } i_1 \geq i \\
\mathsf{close}_i\ x\,(\mathsf{var\_f}\ y) & = & \mathsf{var\_b}\ i & \text{when } x = y \\
\mathsf{close}_i\ x\,(\mathsf{var\_f}\ y) & = & \mathsf{var\_f}\ y & \text{when } x \neq y \\
\mathsf{close}_i\ x\,(\mathsf{abs}\ e_1) & = & \mathsf{abs}\,(\mathsf{close}_{(1+i)}\ x\ e_1) & \\
\mathsf{close}_i\ x\,(\mathsf{app}\ e_1\ e_2) & = & \mathsf{app}\,(\mathsf{close}_i\ x\ e_1)\,(\mathsf{close}_i\ x\ e_2) & \\
\\
\mathsf{close}\ x\ e & = & \mathsf{close}_0\ x\ e & \\
\\
\mathsf{subst} & : & \mathsf{exp} \rightarrow \mathsf{expvar} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp} & \\
\mathsf{subst}\ e\ x\,(\mathsf{var\_b}\ i_1) & = & \mathsf{var\_b}\ i_1 & \\
\mathsf{subst}\ e\ x\,(\mathsf{var\_f}\ y) & = & e & \text{when } x = y \\
\mathsf{subst}\ e\ x\,(\mathsf{var\_f}\ y) & = & \mathsf{var\_f}\ y & \text{when } x \neq y \\
\mathsf{subst}\ e\ x\,(\mathsf{abs}\ e_1) & = & \mathsf{abs}\,(\mathsf{subst}\ e\ x\ e_1) & \\
\mathsf{subst}\ e\ x\,(\mathsf{app}\ e_1\ e_2) & = & \mathsf{app}\,(\mathsf{subst}\ e\ x\ e_1)\,(\mathsf{subst}\ e\ x\ e_2) & \\
\\
\mathsf{lc} & : & \mathsf{exp} \rightarrow \mathsf{Prop} & \\
\mathsf{lc\_var} & : & \forall\, x\,,\ \mathsf{lc}\,(\mathsf{var\_f}\ x) & \\
\mathsf{lc\_app} & : & \forall\, e_1\ e_2\,,\ \mathsf{lc}\ e_1 \rightarrow \mathsf{lc}\ e_2 \rightarrow \mathsf{lc}\,(\mathsf{app}\ e_1\ e_2) & \\
\mathsf{lc\_abs} & : & \forall\, e_1\,,\ (\forall\, x\,,\ \mathsf{lc}\,(\mathsf{open}\,(\mathsf{var\_f}\ x)\ e_1)) \rightarrow \mathsf{lc}\,(\mathsf{abs}\ e_1) & \\
\\
\mathsf{lc\_set} & : & \mathsf{exp} \rightarrow \mathsf{Set} & \\
\mathsf{lc\_set\_var} & : & \forall\, x\,,\ \mathsf{lc\_set}\,(\mathsf{var\_f}\ x) & \\
\mathsf{lc\_set\_app} & : & \forall\, e_1\ e_2\,,\ \mathsf{lc\_set}\ e_1 \rightarrow \mathsf{lc\_set}\ e_2 \rightarrow \mathsf{lc\_set}\,(\mathsf{app}\ e_1\ e_2) & \\
\mathsf{lc\_set\_abs} & : & \forall\, e_1\,,\ (\forall\, x\,,\ \mathsf{lc\_set}\,(\mathsf{open}\,(\mathsf{var\_f}\ x)\ e_1)) \rightarrow \mathsf{lc\_set}\,(\mathsf{abs}\ e_1) &
\end{array}
$$

Convention: The first two arguments to lc_app and lc_set_app are implicit, as are the first arguments to lc_abs and lc_set_abs.

Figure 2: Definitions generated by Ott and LNgen

Induction principle ($\mathsf{lc\_ind}$)

$$\begin{array}{l}
\forall\,(P : \mathsf{exp} \to \mathsf{Prop}),\\
(\forall\,x\,,\ P\,(\mathsf{var_f}\,x)) \to\\
(\forall\,e_1\,e_2\,,\ \mathsf{lc}\,e_1 \to P\,e_1 \to \mathsf{lc}\,e_2 \to P\,e_2 \to P\,(\mathsf{app}\,e_1\,e_2)) \to\\
(\forall\,e_1,\\
\quad (\forall\,x\,,\ \mathsf{lc}\,(\mathsf{open}\,(\mathsf{var_f}\,x)\,e_1)) \to (\forall\,x\,,\ P\,(\mathsf{open}\,(\mathsf{var_f}\,x)\,e_1)) \to P\,(\mathsf{abs}\,e_1)) \to\\
\forall\,e\,,\ \mathsf{lc}\,e \to P\,e
\end{array}$$

Recursion scheme ($\mathsf{lc\_set\_rec}$)

$\mathsf{lc\_set\_rec}$ has the same type as $\mathsf{lc\_ind}$, except with Set instead of Prop, and $\mathsf{lc\_set}$ instead of lc. It behaves as follows: If $f = \mathsf{lc\_set\_rec}\,P\,\mathit{fvar}\,\mathit{fapp}\,\mathit{fabs}$, then

$$\begin{array}{llcl}
f & (\mathsf{var_f}\,x) & (\mathsf{lc_var}\,x) & = & \mathit{fvar}\,x\\
f & (\mathsf{app}\,e_1\,e_2) & (\mathsf{lc_app}\,\mathit{lcp}_1\,\mathit{lcp}_2) & = & \mathit{fapp}\,e_1\,e_2\,\mathit{lcp}_1\,(f\,e_1\,\mathit{lcp}_1)\,\mathit{lcp}_2\,(f\,e_2\,\mathit{lcp}_2)\\
f & (\mathsf{abs}\,e_1) & (\mathsf{lc_abs}\,\mathit{lcp}) & = & \mathit{fabs}\,e_1\,\mathit{lcp}\,(\lambda x.\,f\,(\mathsf{open}\,(\mathsf{var_f}\,x)\,e_1)\,(\mathit{lcp}\,x))\,.
\end{array}$$

Figure 3: Induction principal and recursion scheme

2.2 Generated Lemmas

The main benefit to using LNgen is that it automatically generates a collection of lemmas (with their proofs) about expressions that are useful in metatheoretic reasoning. We highlight the most important of these in Fig. 4. The collection shown includes all of the lemmas that we discussed in our previous work [14]. For convenience, LNgen also generates several variants of the lemmas shown and others besides. Our goal in picking the set of lemmas to generate was not to determine some minimal "complete" set for working with metatheory but to generate a set that, from our experience, we know to be useful in formalizations.

Many of the lemmas in Fig. 4 describe the interaction between the various operations. For example, the first group of lemmas (1–6) describe what happens when fv is applied to expressions built from open, close and subst.

The next eight lemmas (7–14) are primarily about subst. Lemma *subst-spec* decomposes substitution into open composed with close, which was Gordon's definition of substitution [21]. We prefer our version because it commutes directly with constructors. (A definition in terms of open and close would need to use open_i and close_i once it went under a binder.) Lemma *subst-abs* lets us reason about how substitution interacts with abstractions, while making sure that we call subst only on locally closed terms. (The definition of subst just pushes through an abstraction, calling itself recursively on the body, which may have an unresolved index.)

The remaining lemmas (15–23) describe properties of open, close, and lc. Lemma *lc-abs-exists* asserts the existence of an operation that constructs a local closure proof for an abstraction from a proof about a single variable. (Recall that the definition of local closure required that the body be closed for any name for the free variable; this one requires only a single name.) Lemma *lc-subst* asserts the existence of an operation that shows that local closure proofs are preserved by substitution. Lemma *lc-unique* shows that all local closure proofs about the same expression are equivalent.[3] Finally, lemmas *lc-of-lc-set* and *lc-set-of-lc* show the equivalence between lc and $\mathsf{lc\_set}$.

2.3 Generated Proofs

LNgen is able to automatically generate the proofs of each of the lemmas in Fig. 4 because, in general, they are "boring" infrastructure lemmas whose proofs are straightforward inductions. At any given point in a proof, there is little choice about what step to take next. Thus, most of the proof scripts start by applying an induction tactic and then use a "power tactic" to apply a default set of simplifications to the resulting

[3] The proof of this lemma requires extensional equality on functions, which may safely be asserted in Coq as an axiom.

1. *fv-open-upper:*
   $\mathsf{fv}\,(\mathsf{open}\; e_1\; e_2) \;\subseteq\; \mathsf{fv}\; e_1 \;\cup\; \mathsf{fv}\; e_2$

2. *fv-open-lower:*
   $\mathsf{fv}\; e_2 \;\subseteq\; \mathsf{fv}\,(\mathsf{open}\; e_1\; e_2)$

3. *fv-close:*
   $\mathsf{fv}\,(\mathsf{close}\, x\; e) \;=\; \mathsf{fv}\; e \;\setminus\; \{\, x \,\}$

4. *fv-subst-upper:*
   $\mathsf{fv}\,(\mathsf{subst}\; e_1\; x\; e_2) \;\subseteq\; \mathsf{fv}\; e_1 \;\cup\; (\mathsf{fv}\; e_2 \;\setminus\; \{\, x \,\})$

5. *fv-subst-lower:*
   $(\mathsf{fv}\; e_2 \;\setminus\; \{\, x \,\}) \;\subseteq\; \mathsf{fv}\,(\mathsf{subst}\; e_1\; x\; e_2)$

6. *fv-subst-fresh:*
   $\mathsf{fv}\,(\mathsf{subst}\; e_1\; x\; e_2) \;=\; \mathsf{fv}\; e_2$
   when $x \notin \mathsf{fv}\; e_2$

7. *subst-fresh-eq:*
   $\mathsf{subst}\; e_1\; x\; e_2 \;=\; e_2$
   when $x \notin \mathsf{fv}\; e_2$.

8. *subst-subst:*
   $\mathsf{subst}\; e_1\; x\,(\mathsf{subst}\; e_2\; y\; e) =$
   $\mathsf{subst}\,(\mathsf{subst}\; e_1\; x\; e_2)\; y\,(\mathsf{subst}\; e_1\; x\; e)$
   when $y \notin \mathsf{fv}\; e_1$ and $y \neq x$

9. *subst-spec:*
   $\mathsf{subst}\; e_1\; x\; e_2 \;=\; \mathsf{open}\; e_1\,(\mathsf{close}\, x\; e_2)$

10. *subst-open:*
    $\mathsf{subst}\; e_1\; x\,(\mathsf{open}\; e_2\; e_3) =$
    $\mathsf{open}\,(\mathsf{subst}\; e_1\; x\; e_2)\,(\mathsf{subst}\; e_1\; x\; e_3)$
    when $\mathsf{lc}\; e_1$

11. *subst-open-var:*
    $\mathsf{subst}\; e_1\; x\,(\mathsf{open}\,(\mathsf{var\_f}\; y)\; e_2) \;=\; \mathsf{open}\,(\mathsf{var\_f}\; y)\,(\mathsf{subst}\; e_1\; x\; e_2)$
    when $x \neq y$ and $\mathsf{lc}\; e_1$

12. *subst-abs:*
    $\mathsf{subst}\; e_1\; x\,(\mathsf{abs}\; e_2) \;=\; \mathsf{abs}\,(\mathsf{close}\, z\,(\mathsf{subst}\; e_1\; x\,(\mathsf{open}\,(\mathsf{var\_f}\; z)\; e_2)))$
    when $z \notin \mathsf{fv}\; e_1 \cup \mathsf{fv}\; e_2 \cup \{\, x \,\}$ and $\mathsf{lc}\; e_1$

13. *subst-close:*
    $\mathsf{subst}\; e_1\; x\,(\mathsf{close}\, y\; e_2) \;=\; \mathsf{close}\, y\,(\mathsf{subst}\; e_1\; x\; e_2)$
    when $x \neq y$ and $y \notin \mathsf{fv}\; e_1$ and $\mathsf{lc}\; e_1$

14. *subst-intro:*
    $\mathsf{open}\; e_1\; e_2 \;=\; \mathsf{subst}\; e_1\; x\,(\mathsf{open}\,(\mathsf{var\_f}\; x)\; e_2)$ when $x \notin \mathsf{fv}\; e_2$

15. *open-close:*
    $\mathsf{open}\,(\mathsf{var\_f}\; x)\,(\mathsf{close}\, x\; e) \;=\; e$

16. *close-open:*
    $\mathsf{close}\, x\,(\mathsf{open}\,(\mathsf{var\_f}\; x)\; e) \;=\; e$
    when $x \notin \mathsf{fv}\; e$

17. *open-inj:*
    $\mathsf{open}\,(\mathsf{var\_f}\; x)\; e_1 = \mathsf{open}\,(\mathsf{var\_f}\; x)\; e_2$
    implies $e_1 \;=\; e_2$
    when $x \notin \mathsf{fv}\; e_1 \;\cup\; \mathsf{fv}\; e_2$

18. *close-inj:*
    $\mathsf{close}\, x\; e_1 \;=\; \mathsf{close}\, x\; e_2$
    implies $e_1 \;=\; e_2$

19. *lc-abs-exists:*
    $\mathsf{lc\_abs\_exists}\, x\; lcp \;:\; \mathsf{lc}\,(\mathsf{abs}\; e)$
    when $lcp \;:\; \mathsf{lc}\,(\mathsf{open}\,(\mathsf{var\_f}\; x)\; e)$

20. *lc-subst:*
    $\mathsf{lc\_subst}\; lcp_1\; x\; lcp_2 : \mathsf{lc}\,(\mathsf{subst}\; e_1\; x\; e_2)$
    when $lcp_1 : \mathsf{lc}\; e_1$ and $lcp_2 : \mathsf{lc}\; e_2$

21. *lc-unique:*
    If $(lcp_1 : \mathsf{lc}\; e)$ and $(lcp_2 : \mathsf{lc}\; e)$,
    then $lcp_1 = lcp_2$

22. *lc-of-lc-set:*
    $\mathsf{lc\_set}\; e$ implies $\mathsf{lc}\; e$

23. *lc-set-of-lc:*
    $\mathsf{lc}\; e$ implies $\mathsf{lc\_set}\; e$

Figure 4: Some of the lemmas generated by LNgen

subgoals. In cases where this is not sufficient, LNgen generates more complex scripts based on our knowledge of how such proofs normally proceed. There is no worry about the soundness of our reasoning: the scripts generated by LNgen must be run by Coq to generate proof terms that are then checked.

We favor generating proof scripts over proof terms because it keeps the implementation of LNgen simple. Proof terms are specific to individual lemmas and vary from language to language. By contrast, our tactics—which are useful in their own right—apply to multiple lemmas and do not need to vary from language to language. Unfortunately, because Coq's tactic language is incompletely specified, it is impossible for us to guarantee that our scripts will always succeed. These scripts have never failed on any of our case studies. However, if some proof should fail, the effect is localized. The user may have to do that proof by hand (if they would like to use that lemma) but other generated definitions, lemmas, and proofs will still be available.

2.4 Input Restrictions

LNgen supports only a subset of the Ott language. List forms (for specifying constructors of variable arity) and subgrammars (for indicating that, for example, values are a subset of expressions) are both unsupported. The only binding specifications accepted by LNgen are those where a single metavariable binds in a single nonterminal. This excludes Ott's auxiliary functions for computing the set of binders in an object, e.g., those introduced by nested record patterns. We see no reason why some future version of LNgen could not be extended with these forms.

3 Soundness

Since everything generated by Ott and LNgen must be run through Coq, there is no need to worry that one is building a development on top of an inconsistent foundation—Coq will complain if a definition is ill-formed or if a proof is incomplete. However, this is not the same as saying that their outputs faithfully reflect the language that the user specified. Binding specifications in Ott use names (i.e., metavariables) to indicate binding occurrences of variables, as is common in informal practice. Intuitively, terms in the specification use a fully concrete encoding of binding: all variables are named, and terms are identified up to alpha equivalence. On the other hand, we use Ott and LNgen to generate output that uses a locally nameless representation for binding, where bound variables are represented as de Bruijn indices and where syntactic equality corresponds to alpha equivalence.

In this section, we prove that the user need not worry about this difference in representations: the locally nameless representations generated by Ott and LNgen are adequate representations of the fully concrete ones. Informally, this means that there is a bijection between the terms of the two representations and that substitution is compositional with respect to this bijection [22]. Terms representable in one representation are representable in the other, and substitution means the same thing for both representations. Below, we make these notions precise and carry out the proofs for the specific case of untyped lambda terms (Fig. 1). By considering adequacy for a particular (and small) language, we keep the proofs below relatively simple, while still demonstrating the utility of the lemmas generated by LNgen. A language-independent account of adequacy would require a precise semantics for Ott specifications and a precise specification of how Ott and LNgen generate their output. We leave developing these for future work. We also leave as future work formalization in Coq of the proofs below. Ott does not generate a definition of capture avoiding substitution or of alpha equivalence. Furthermore, *mechanized* reasoning about these notions is difficult and extremely tedious—precisely the reasons why we prefer to represent binding in some other way! Without tool support, we must work out ourselves properties of capture avoiding substitution and alpha equivalence that are ordinarily taken for granted when writing out proofs by hand.

Fully concrete lambda terms are defined in Fig. 5, along with free variables, capture-avoiding substitution, and alpha equivalence. Note that capture-avoiding substitution is defined by induction on the height of terms simultaneously with a proof that substituting a variable preserves the height of terms. (In the second case for lambda abstractions, the recursive call is not on an immediate subterm.) By assuming that picking a variable fresh for a finite set is deterministic, we obviate the need to show that the definition of substitution actually defines a function—this is trivially the case. We find it convenient to work with a definition of

$\boxed{\text{Expressions}}$

$$M, N \quad ::= \quad x \mid M_1\, M_2 \mid \lambda\, x \,.\, M_1$$

$\boxed{\text{Free variables}}$

$$\begin{array}{lcl} \mathsf{fv}\,(x) & \stackrel{\text{def}}{=} & \{\, x \,\} \\ \mathsf{fv}\,(M_1\, M_2) & \stackrel{\text{def}}{=} & (\mathsf{fv}\, M_1) \cup (\mathsf{fv}\, M_2) \\ \mathsf{fv}\,(\lambda\, x \,.\, M_1) & \stackrel{\text{def}}{=} & (\mathsf{fv}\, M_1) \setminus \{\, x \,\} \end{array}$$

$\boxed{\text{Capture avoiding substitution}}$

$$\begin{array}{lcl} [\, N \,/\, x \,]\,(x) & \stackrel{\text{def}}{=} & N \\ [\, N \,/\, x \,]\,(y) & \stackrel{\text{def}}{=} & y \text{ when } y \neq x \\ [\, N \,/\, x \,]\,(M_1\, M_2) & \stackrel{\text{def}}{=} & ([\, N \,/\, x \,]\, M_1)\,([\, N \,/\, x \,]\, M_2) \\ [\, N \,/\, x \,]\,(\lambda\, x \,.\, M_1) & \stackrel{\text{def}}{=} & \lambda\, x \,.\, M_1 \\ [\, N \,/\, x \,]\,(\lambda\, y \,.\, M_1) & \stackrel{\text{def}}{=} & (\lambda\, z \,.\, [\, N \,/\, x \,]\,[\, z \,/\, y \,]\, M_1) \\ & & \text{for some } z \notin \mathsf{fv}\, N \cup \mathsf{fv}\, M_1 \text{ and when } y \neq x \end{array}$$

$\boxed{\text{Alpha equivalence}}$

The binary relation $=_\alpha$ on expressions is the least congruence closed under

$$\lambda\, x \,.\, M_1 =_\alpha \lambda\, y \,.\, [\, y \,/\, x \,]\, M_1 \qquad \text{when } y \notin \mathsf{fv}\, M_1$$

Figure 5: Fully concrete lambda terms

capture-avoiding substitution that is total, so the abstraction case always renames the bound variable to avoid capture.

To show the adequacy of our locally nameless representation, we prove that there is an alpha-equivalence respecting bijection between concrete terms and locally nameless terms that are locally closed. We give this bijection by defining functions between the two sets and then proving that they are inverses of each other. We define the function $\lceil - \rceil$ from concrete terms to locally nameless ones as follows:

$$\begin{array}{lcl} \lceil x \rceil & \stackrel{\text{def}}{=} & \mathsf{var\_f}\; x \\ \lceil M_1\, M_2 \rceil & \stackrel{\text{def}}{=} & \mathsf{app}\, \lceil M_1 \rceil\, \lceil M_2 \rceil \\ \lceil \lambda\, x \,.\, M_1 \rceil & \stackrel{\text{def}}{=} & \mathsf{abs}\,(\mathsf{close}\; x\, \lceil M_1 \rceil)\,. \end{array}$$

The fact that this function yields only locally closed terms follows by structural induction on its argument, using lemmas *lc-abs-exists* and *open-close* in the case for abstractions. We define the function $\lfloor - \rfloor$ from locally nameless terms that are locally closed to concrete terms using the recursion principle in Fig. 3. Note that this definition is also a function, again because we assume that picking a fresh variable not in a particular set is deterministic.

$$\begin{array}{lcl} \lfloor \mathsf{var\_f}\; x \rfloor & \stackrel{\text{def}}{=} & x \\ \lfloor \mathsf{app}\; e_1\; e_2 \rfloor & \stackrel{\text{def}}{=} & \lfloor e_1 \rfloor\, \lfloor e_2 \rfloor \\ \lfloor \mathsf{abs}\; e_1 \rfloor & \stackrel{\text{def}}{=} & \lambda\, x \,.\, \lfloor \mathsf{open}\,(\mathsf{var\_f}\; x)\; e_1 \rfloor \quad \text{for some } x \notin \mathsf{fv}\; e_1 \end{array}$$

In the remainder of this section, we sketch out the proof of adequacy; additional details can be found in the appendix. The proofs below are straightforward given the lemmas generated by LNgen. We need only to be careful about ordering properly the lemmas and theorems.

We first need to show that both $\lceil - \rceil$ and $\lfloor - \rfloor$ preserve free variables. These proofs also serve as basic sanity checks: it would be odd for corresponding terms in the two representations to have different sets of free variables.

Lemma 1 $\mathsf{fv}\,(M) \;=\; \mathsf{fv}\,(\lceil M \rceil)$ *for any* M*.*

Proof

By induction on the structure of M. In the case for abstractions, we need lemma *fv-close*. □ □

Lemma 2 $\mathsf{fv}\,(\lfloor e \rfloor) \;=\; \mathsf{fv}\,(e)$ *for any locally closed* e*.*

Proof

By induction on the proof that e is locally closed. In the case for abstractions, we need lemmas *fv-close* and *close-open*. □ □

Next, we prove simultaneously that $\lceil - \rceil$ commutes with substitution and that it preserves alpha equivalence. For $\lfloor - \rfloor$, we prove that it commutes with substitution; it trivially preserves alpha equivalence.

Theorem 3 *For all* M*,*

1. *Substitution commutes with* $\lceil - \rceil$*. That is, for any* N *and* x*,*

$$\lceil [\, N \,/\, x \,]\, M \rceil \;=\; \mathsf{subst}\, \lceil N \rceil\, x\, \lceil M \rceil \,.$$

2. $\lceil - \rceil$ *respects alpha-equivalence. That is, for any* N *such that* $N =_\alpha M$*,*

$$\lceil M \rceil \;=\; \lceil N \rceil \,.$$

Proof

We prove these two results simultaneously by induction on the height of M, observing that substituting a variable does not change the height of a term. We need lemmas *fv-close*, *subst-fresh-eq*, *subst-spec*, *subst-close*, and *close-open*. □
□

Theorem 4 $\lfloor - \rfloor$ *commutes with substitution. That is,*

$$\lfloor \mathsf{subst}\, g\, x\, e \rfloor \;=_\alpha\; [\, \lfloor g \rfloor \,/\, x \,]\, \lfloor e \rfloor$$

for all locally closed e *and* g*, and for all* x*.*

Proof

By induction on the proof that e is locally closed. In the case for abstractions, we need lemmas *subst-fresh-eq*, *subst-spec*, *subst-abs*, *open-close*, and *close-open*. □ □

Finally, we prove that $\lceil - \rceil$ and $\lfloor - \rfloor$ are inverses of each other. It follows that each function defines a bijection.

Theorem 5 $\lfloor \lceil M \rceil \rfloor \;=_\alpha\; M$ *for any* M*.*

Proof

By induction on the structure of M. In the case for abstractions, we need theorem 4, and lemmas *fv-close* and *subst-spec*. □ □

Theorem 6 $\lceil \lfloor e \rfloor \rceil \;=\; e$ *for any locally closed* e*.*

Proof

By induction on the proof that e is locally closed. In the case for abstractions, we need lemma *close-open*.
□ □

Taken together, theorems 3–6 suffice to prove that the locally nameless representation generated by Ott and LNgen is adequate with respect to the fully concrete interpretation of the original Ott specification.

4 Completeness

Does LNgen generate enough definitions and properties to get work done? Of course, this is an impossible question to answer because the tool cannot possibly generate proofs of every property that one could need or want. However, we can limit the scope of the question by showing that LNgen *trivially* models some specification of binding. By choosing a specification that makes no mention of de Bruijn indices, this result implies that the user need only work with locally-closed terms and never reason explicitly about de Bruijn indices.

We make our claim by showing that the output of Ott and LNgen for untyped lambda terms (Fig. 1) is not very far from Gordon and Melham's "Five Axioms of Alpha-Conversion" [19]. In fact, we can derive these axioms with only currying, uncurrying, and applications of lemmas generated by LNgen. This work is a bit tedious, but none of it includes reasoning about de Bruijn indices, doing induction on raw expressions, or doing induction on local closure derivations. Thus, it substantiates our claim that the output of our tool provides users with enough machinery to reason about binding. The LNgen distribution includes a straightforward, mechanical formalization in Coq of the results of this section.

Gordon and Melham's five axioms are defined in terms of a type Term, three constructors for that type,

$$\begin{array}{lcl} \mathsf{Var} & : & \mathsf{expvar} \to \mathsf{Term} \\ \mathsf{App} & : & \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term} \\ \mathsf{Lam} & : & \mathsf{expvar} \to \mathsf{Term} \to \mathsf{Term}\,, \end{array}$$

and three operations for that type,

$$\begin{array}{lcl} \mathsf{Fv} & : & \mathsf{Term} \to \mathsf{expvarset} \\ \mathsf{Subst} & : & \mathsf{Term} \to (\mathsf{Term} \times \mathsf{expvar}) \to \mathsf{Term} \\ \mathsf{Abs} & : & (\mathsf{expvar} \to \mathsf{Term}) \to \mathsf{Term}\,. \end{array}$$

Our implementation starts by defining Term as a dependent pair of a raw expression and a proof that it is locally closed.

Definition 7 (Term)

$$\mathsf{Term} \stackrel{\text{def}}{=} \Sigma\, e : \mathsf{exp}.\, \mathsf{lc}\; e\,.$$

The definitions of the three constructors simply construct and propagate local closure proofs. In the definition of Lam, we explicitly use the "existential" version of $\mathsf{lc\_abs}$ (i.e., $\mathsf{lc\_abs\_exists}$) and implicitly use lemma *open-close* to show that the local closure proof applies to first component of the tuple.

Definition 8 (Gordon-Melham Constructors)

$$\begin{array}{lcl} \mathsf{Var}\, x & \stackrel{\text{def}}{=} & (\mathsf{var\_f}\; x, \mathsf{lc\_var}\; x) \\ \mathsf{App}\,(e_1, lcp_1)\,(e_2, lcp_2) & \stackrel{\text{def}}{=} & (\mathsf{app}\; e_1\; e_2, \mathsf{lc\_app}\; lcp_1\; lcp_2) \\ \mathsf{Lam}\; x\;(e_1, lcp_1) & \stackrel{\text{def}}{=} & (\mathsf{abs}\,(\mathsf{close}\; x\; e_1), \mathsf{lc\_abs\_exists}\; x\; lcp_1) \end{array}$$

The definitions for free variables (Fv) and substitution (Subst) simply push the operations on raw terms through the dependent pair. For substitution, we rely on the fact that substitution preserves local closure.

Definition 9 (Fv **and** Subst)

$$\begin{array}{lcl} \mathsf{Fv}\,(e_1, lcp_1) & \stackrel{\text{def}}{=} & \mathsf{fv}\; e_1 \\ \mathsf{Subst}\,(e_1, lcp_1)\,((e_2, lcp_2),\; x) & \stackrel{\text{def}}{=} & (\mathsf{subst}\; e_2\; x\; e_1, \mathsf{lc\_subst}\; lcp_2\; x\; lcp_1) \end{array}$$

The final operation, Abs, reifies a function from variable names to terms into a lambda term. We defer its definition until later, when we discuss the last of the five axioms.

With the model above, we can derive Gordon and Melham's five axioms. The proofs of their five axioms involve little more than projecting out components of dependent pairs and applying lemmas generated by

LNgen to construct local closure derivations. In fact, the only interesting aspect of these proofs is that they are so uninteresting. Below, we only mention the lemmas that the proofs depend on; additional details can be found in the appendix.

The first three axioms are basic facts about free variables, capture-avoiding substitution, and alpha conversion.

Theorem 10 (Axiom 1: Free variables)

1. $\mathsf{Fv}\,(\mathsf{Var}\,x) = \{\, x \,\}$
2. $\mathsf{Fv}\,(\mathsf{App}\,t_1\,t_2) = \mathsf{Fv}\,t_1 \,\cup\, \mathsf{Fv}\,t_2$
3. $\mathsf{Fv}\,(\mathsf{Lam}\,x\,t_1) = \mathsf{Fv}\,t_1 \,\setminus\, \{\, x \,\}$

Proof

By unfolding definitions. Part 3 requires lemma *fv-close*. □ □

Theorem 11 (Axiom 2: Substitution)

1. $\mathsf{Subst}\,(\mathsf{Var}\,x)\,(u,\,x) \;=\; u$
2. $x \,\neq\, y$ *implies* $\mathsf{Subst}\,(\mathsf{Var}\,y)\,(u,\,x) = \mathsf{Var}\,y$
3. $\mathsf{Subst}\,(\mathsf{App}\,t_1\,t_2)\,(u,\,x) \;=\; \mathsf{App}\,(\mathsf{Subst}\,t_1\,(u,\,x))\,(\mathsf{Subst}\,t_2\,(u,\,x))$
4. $\mathsf{Subst}\,(\mathsf{Lam}\,x\,t)\,(u,\,x) \;=\; \mathsf{Lam}\,x\,t$
5. $x \,\neq\, y$ *and* $y \,\notin\, (\mathsf{Fv}\,u)$ *imply* $\mathsf{Subst}\,(\mathsf{Lam}\,y\,t)\,(u,\,x) \;=\; \mathsf{Lam}\,y\,(\mathsf{Subst}\,t\,(u,\,x))$

Proof

By unfolding definitions. All parts require lemma *lc-unique*. Part 4 also requires lemmas *fv-close* and *subst-fresh-eq*. Part 5 also requires lemma *subst-close*. □
□

Theorem 12 (Axiom 3: Alpha conversion)

$y \,\notin\, \mathsf{Fv}\,(\mathsf{Lam}\,x\,t)$ *implies*
$\mathsf{Lam}\,x\,t = \mathsf{Lam}\,y\,(\mathsf{Subst}\,t\,(\mathsf{Var}\,y,\,x))$

Proof

By unfolding definitions. The proof requires lemmas *fv-close*, *subst-spec*, *close-open*, and *lc-unique*. □ □

To support the definition of functions over lambda-calculus expressions, Gordon and Melham's work states an iteration axiom and uses it to derive a recursion scheme through pairing. However, because Coq produces recursion schemes already, we define the recursion scheme directly. The iterative version follows as a simple corollary.

Theorem 13 (Axiom 4: Recursion scheme) *For all result types R and all*

$$\begin{array}{l}(\mathit{fvar} : \mathsf{expvar} \,\to\, R)\\ (\mathit{fapp} : R \,\to\, R \,\to\, \mathsf{Term} \,\to\, \mathsf{Term} \,\to\, R)\\ (\mathit{fabs} : (\mathsf{expvar} \,\to\, R) \,\to\, (\mathsf{expvar} \,\to\, \mathsf{Term}) \,\to\, R)\,,\end{array}$$

there exists a unique f of type $\mathsf{Term} \,\to\, R$ *such that*

1. $f\,(\mathsf{Var}\,x) = \mathit{fvar}\,x$
2. $f\,(\mathsf{App}\,t_1\,t_2) = \mathit{fapp}\,(f\,t_1)\,(f\,t_2)\,t_1\,t_2$

3. $f\,(\mathsf{Lam}\,x\,t) = \mathit{fabs}\,(\lambda y.\,f(\mathsf{Subst}\,t\,(\mathsf{Var}\,y,\ x)))\,(\lambda y.\,\mathsf{Subst}\,t\,(\mathsf{Var}\,y,\ x))$.

Proof

By unfolding definitions. All parts require lemma *lc-unique*. Part 3 also requires lemma *subst-spec*. □ □

The final axiom concerns Abs, an operation for turning functions from expvars to Terms into lambda abstractions. This operation allows the Gordon-Melham recursion combinator to create a new term in the lambda case. The trickiest part of the definition of Abs is picking a variable name to use for the binder that is fresh for the body of the abstraction. We do this in two stages: We first access the body with an arbitrary variable x_0 (which may already appear in the body), and then we use the resulting term to pick a variable certain to be fresh for body. We use lc_abs_exists and lemma *open-close* similarly to how we did in the definition of Lam.

Definition 14 (Abs)

$$\begin{array}{rl} \mathsf{Abs}\,f \;=& (\mathsf{abs}\,(\mathsf{close}\,y\,e_2), \mathsf{lc\_abs\_exists}\,y\,\mathit{lcp}_2) \\ \mathit{where} & (e_1, \_) = f\,x_0 \\ & y \notin (\mathsf{fv}\,e_1) \\ & (e_2, \mathit{lcp}_2) = f\,y \end{array}$$

With Abs defined, we can now state and derive the final axiom.

Theorem 15 (Axiom 5: Abstraction)

$$\mathsf{Abs}\,(\lambda y.\,\mathsf{Subst}\,t\,(\mathsf{Var}\,y,\ x)) \;=\; \mathsf{Lam}\,x\,t$$

Proof

By unfolding definitions. The proof requires lemmas *fv-close*, *fv-subst-lower*, *subst-spec*, *close-open*, and *lc-unique*. □
□

The abstraction operation is the only definition that is not trivial in that it first must calculate a fresh variable for the term. The advantage of axiom 5 is that it lets one have a lambda expression without naming its binder. However, in some sense, Abs is not necessary for our style of reasoning. Certainly, all of this effort is not required to define functions with lc_set_rec, e.g., *beta* in Sec. 2.1.

5 Case Studies

We have used LNgen to streamline proofs of type safety for the simply-typed lambda calculus and for System F with subtyping, i.e., parts 1A and 2A of the PoplMark challenge. In both cases, the only proofs that needed to be mechanized by hand were lemmas about the relations of their respective systems. (Because LNgen works only with syntax, it cannot be expected to generate these proofs.) Every necessary lemma concerning only the calculation of free variables, substitution, and local closure was automatically proved by LNgen.

Others have used LNgen for far more substantial developments than the two above. Greenberg et al. [23] used LNgen to help formalize a proof of confluence for parallel reduction in dependent λ^h, a language with manifest contracts. Greenberg reports[4] that, "All in all, LNgen was great—it covered most of the stupid facts I needed." The tool failed to generate only one set of lemmas, which concerned how substitution maintains invariants about the free variables of terms. Jia et al. [24] used LNgen when they proved type soundness for a dependently-typed language with strong eliminators and an abstract definition of program equivalence. The authors report[5] that without the 9000 lines of lemmas and proofs that LNgen generated for their language, they would have been unable to complete their formalization in a timely fashion. Because the tool provided every infrastructure lemma they needed, they were able to focus their efforts on the novel

[4] By personal communication.
[5] Again, by personal communication.

aspects of their language's design and complete their formalization in about nine days—an impressive feat given the complexity of their design and the fact that they were tweaking the design in the process. Taken together, these two non-trivial developments provide a compelling story about the effectiveness of LNgen in eliminating the tedium associated with locally nameless encodings.

6 Related Work

Much work has been done in the area of representing binding. For example, we have already discussed the "Five Axioms of Alpha-Conversion." In previous work [14], we also gave an extensive survey of first-order representation techniques. Thus, we focus this section on work that is specifically related to the issues described in this paper.

Logical frameworks—such as Abella [1], Hybrid [2], and Twelf [5]—are specifically designed to represent and reason about logics and programming languages. Their specialized meta-logics encourage the use of higher-order abstract syntax (HOAS), which represents binding in an object language using binding in the framework's meta-logic. Thus, when reasoning about an object language, one gets facts about alpha equivalence, substitution, and free variables "for free." Unfortunately, the generality of Coq's logic precludes traditional HOAS encodings, and first-order representations (e.g., locally nameless) require that one explicitly deal with free variable calculation and substitution. LNgen steps in here to recover the benefits to working in a traditional logical framework by automatically proving properties about syntax that one expects to have "for free."

The Lambda Tamer project [3] also automatically proves a variety of facts about programming languages encoded in Coq. Compared to LNgen, Lambda Tamer favors the use of dependent types when representing syntax, ensuring that only well-typed syntax, according to the type system of the *object* language, can be represented. It uses generic programming techniques to ensure that generated proofs are correct by construction. As mentioned previously (Sec. 2.3), we prefer to generate proof scripts because of the approach's simplicity—writing generic proofs directly is a non-trivial exercise and would have slowed the development of LNgen.

Parametric higher-order abstract syntax (PHOAS) [25] is a representation technique that allows one to use HOAS-like approach to represent binding, thus obtaining "for free" facts about syntax that LNgen has to prove about locally nameless encodings. The key idea is to represent the body of an abstraction not as a function from expression to expressions, as with HOAS, but as a function from *variables* to expressions, an approach reminiscent of weak HOAS [10]. Ill-formed terms are ruled out by by universally quantifying over the type of variables and appealing to parametricity to ensure that the type for variables is treated abstractly. Without a general proof of parametricity for Coq, one must assert that parametricity holds for particular terms as needed or as an axiom.

7 Conclusions and Future Work

Since LNgen is currently only a prototype, there are a number of promising avenues for future development and research. We developed LNgen independently from Ott in order to make it easier to experiment with its output: which definitions to generate, which lemmas to generate, how to generate proofs, etc. But, it might be beneficial to add such support to Ott directly. The ideas we have presented here are not particular to Coq, and we expect that they can be generalized to the full spectrum of Ott's binding forms. We also believe that it is possible to automatically generate theorems about some judgements: equivariance (invariance under swappings of variables), weakening, and substitution, for example. Support for defining functions directly in Ott specifications and having them translated into locally nameless definitions, using schemes such as lc_set_rec, would also be useful. In particular, one would like to know that something similar to the "freshness condition for binders" from Nominal Isabelle holds whenever a function is defined. In the case of binding constructors, this would allow one to conclude that the behavior of the function does not depend on the particular choice of name for the bound variable (recall the definition of *beta* in Sec. 2.1). On a more theoretical note, we envision giving a general account of how to transform a fully concrete representation into a locally nameless one, thus making it possible to give a general account of soundness for LNgen.

In the end, what we provide *now* is a usable prototype tool for taking our locally nameless style—already a lightweight representation technique—and making it even lighter weight. We have shown that Ott and LNgen are sound and complete in the specific case of untyped lambda terms. Compared to our previous work, we now provide a recursion scheme for defining functions, and it comes "for free" from our definitions. On a day to day basis, the benefit of our work is simple: no more boring infrastructure proofs.

Acknowledgements

This work was funded by DARPA, *CSSG Phase II: Machine Checked Metatheory for Security-Oriented Languages.*

References

[1] Gacek, A.: The Abella interactive theorem prover (system description). In Armando, A., Baumgartner, P., Dowek, G., eds.: Automated Reasoning: Fourth International Joint Conference, IJCAR 2008. Volume 5195 of Lecture Notes in Artificial Intelligence. Springer (2008) 154–161

[2] Momigliano, A., Martin, A.J., Felty, A.P.: Two-level Hybrid: A system for reasoning using higher-order abstract syntax. In Abel, A., Urban, C., eds.: Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008). Volume 228 of Electronic Notes in Theoretical Computer Science. Elsevier (2009) 85–93

[3] Chlipala, A.: Generic programming and proving for programming language metatheory. Technical Report UCB/EECS-2007-147, University of California, Berkeley (2007)

[4] Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning **40**(4) (2008) 327–356

[5] Pfenning, F., Schürmann, C.: System description: Twelf — A meta-logical framework for deductive systems. In Ganzinger, H., ed.: Automated Deduction, CADE 16: 16th International Conference on Automated Deduction. Volume 1632 of Lecture Notes in Artificial Intelligence. Springer (1999) 202–206

[6] de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae **34**(5) (1972) 381–392

[7] Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. ACM (1988) 199–208

[8] McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. Journal of Automated Reasoning **23**(3–4) (1999) 373–409

[9] Pollack, R.: Closure under alpha-conversion. In Barendregt, H., Nipkow, T., eds.: Types for Proofs and Programs: International Workshop, TYPES 1993. Volume 806 of Lecture Notes in Computer Science. Springer (1994) 313–332

[10] Despeyroux, J., Felty, A., Hirschowitz, A.: Higher-order abstract syntax in Coq. In: Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95. Volume 902 of Lecture Notes in Computer Science. Springer (1995) 124–138. Also available as INRIA Research report 2556

[11] Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The PoplMark challenge. In Hurd, J., Melham, T., eds.: Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005. Volume 3603 of Lecture Notes in Computer Science. Springer (2005) 50–65

[12] Pitts, A.M.: Nominal logic, a first order theory of names and binding. Information and Computation **186** (2003) 165–193

[13] Aydemir, B., Bohannon, A., Weirich, S.: Nominal reasoning techniques in Coq (extended abstract). In Momigliano, A., Pientka, B., eds.: Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006). Volume 174 of Electronic Notes in Theoretical Computer Science. Elsevier (2007) 69–77

[14] Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM (2008) 3–15

[15] Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.: Aura: A programming language for authorization and audit. In: ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ACM (2008) 27–38

[16] Pratikakis, P., Foster, J.S., Hicks, M., Neamtiu, I.: Formalizing soundness of contextual effects. In Ait Mohamed, O., Muñoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008. Volume 5170 of Lecture Notes in Computer Science. Springer (2008) 262–277

[17] Rossberg, A., Russo, C., Dreyer, D.: F-ing modules. Submitted for publication (October 2010)

[18] Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. In: ICFP '07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming. ACM (2007) 1–12

[19] Gordon, A.D., Melham, T.: Five axioms of alpha-conversion. In von Wright, J., Grundy, J., Harrison, J., eds.: Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96. Volume 1125 of Lecture Notes in Computer Science. Springer (1996) 173–190

[20] Sewell, P., Zappa Nardelli, F.: Ott. Available from `http://www.cl.cam.ac.uk/~pes20/ott/` (2009)

[21] Gordon, A.D.: A mechanisation of name-carrying syntax up to alpha-conversion. In Joyce, J.J., Seger, C.J.H., eds.: Higher-order Logic Theorem Proving And Its Applications, Proceedings, 1993. Volume 780 of Lecture Notes in Computer Science. Springer (1994) 414–426

[22] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1) (1993) 143–184

[23] Greenberg, M., Pierce, B., Weirich, S.: Contracts made manifest. In: POPL '10: Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Madrid, Spain, ACM (January 2010). To appear.

[24] Jia, L., Zhao, J., Sjöberg, V., Weirich, S.: Dependent types and program equivalence. In: POPL '10: Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Madrid, Spain, ACM (January 2010). To appear.

[25] Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ACM (2008) 143–156

A Proofs

A.1 Proof of Theorem 3

We prove these two results simultaneously by induction on the height of M, observing that substituting a variable does not change the height of a term. We need lemmas *fv-close*, *subst-fresh-eq*, *subst-spec*, *subst-close*, and *close-open*. For example, the abstraction case of the second part is shown below, where $M = (\lambda\, x\, .\, M_1)$ and $N = \lambda\, y\, .\, [\, y \,/\, x\,]\, M_1$, with $y \notin \mathsf{fv}\, M_1$.

$$
\begin{aligned}
& \lceil \lambda\, x\, .\, M_1 \rceil \\
& \qquad \text{by definition of } \lceil - \rceil \\
& = \mathsf{abs}\,(\mathsf{close}\, x\, \lceil M_1 \rceil) \\
& \qquad \text{by lemma } \textit{close-open} \\
& = \mathsf{abs}\,(\mathsf{close}\, y\, (\mathsf{open}\, (\mathsf{var\_f}\, y)\, (\mathsf{close}\, x\, \lceil M_1 \rceil))) \\
& \qquad \text{by lemma } \textit{subst-spec} \\
& = \mathsf{abs}\,(\mathsf{close}\, y\, (\mathsf{subst}\, (\mathsf{var\_f}\, y)\, x\, \lceil M_1 \rceil)) \\
& \qquad \text{by IH(1)} \\
& = \mathsf{abs}\,(\mathsf{close}\, y\, \lceil [\, y \,/\, x\,]\, M_1 \rceil) \\
& \qquad \text{by definition of } \lceil - \rceil \\
& = \lceil \lambda\, y\, .\, [\, y \,/\, x\,]\, M_1 \rceil
\end{aligned}
$$

A.2 Proof of Theorem 4

By induction on the proof that e is locally closed. The only interesting case is when $e = \mathsf{abs}\, e_1$, for some e_1. Let $y \notin \mathsf{fv}\, e_1$ and $w \notin \mathsf{fv}\, g \cup \{\, x\,\} \cup \mathsf{fv}\, e_1 \cup \{\, y\,\}$. We consider two cases for x and y. First, suppose that $x \neq y$.

$$
\begin{aligned}
& [\, \lfloor g \rfloor \,/\, x\,]\, \lfloor \mathsf{abs}\, e_1 \rfloor \\
& \qquad \text{for some } y \notin \mathsf{fv}\, e_1 \\
& = [\, \lfloor g \rfloor \,/\, x\,]\, (\lambda\, y\, .\, \lfloor \mathsf{open}\, (\mathsf{var\_f}\, y)\, e_1 \rfloor) \\
& \qquad \text{by alpha conversion} \\
& =_\alpha [\, \lfloor g \rfloor \,/\, x\,]\, (\lambda\, w\, .\, [\, w \,/\, y\,]\, \lfloor \mathsf{open}\, (\mathsf{var\_f}\, y)\, e_1 \rfloor) \\
& \qquad \text{by property of substitution} \\
& =_\alpha \lambda\, w\, .\, [\, \lfloor g \rfloor \,/\, x\,]\, [\, w \,/\, y\,]\, (\lfloor \mathsf{open}\, (\mathsf{var\_f}\, y)\, e_1 \rfloor) \\
& \qquad \text{by IH} \\
& =_\alpha \lambda\, w\, .\, [\, \lfloor g \rfloor \,/\, x\,]\, (\lfloor \mathsf{subst}\, (\mathsf{var\_f}\, w)\, y\, (\mathsf{open}\, (\mathsf{var\_f}\, y)\, e_1) \rfloor) \\
& \qquad \text{by lemmas } \textit{subst-spec} \text{ and } \textit{close-open} \\
& = \lambda\, w\, .\, [\, \lfloor g \rfloor \,/\, x\,]\, (\lfloor \mathsf{open}\, (\mathsf{var\_f}\, w)\, e_1 \rfloor) \\
& \qquad \text{by IH} \\
& =_\alpha \lambda\, w\, .\, \lfloor \mathsf{subst}\, g\, x\, (\mathsf{open}\, (\mathsf{var\_f}\, w)\, e_1) \rfloor \\
& \qquad \text{by lemma } \textit{open-close} \\
& = \lambda\, w\, .\, \lfloor \mathsf{open}\, (\mathsf{var\_f}\, w)\, (\mathsf{close}\, w\, (\mathsf{subst}\, g\, x\, (\mathsf{open}\, (\mathsf{var\_f}\, w)\, e_1))) \rfloor \\
& \qquad \text{by the definition of } \lfloor - \rfloor \\
& = \lfloor \mathsf{abs}\, (\mathsf{close}\, w\, (\mathsf{subst}\, g\, x\, (\mathsf{open}\, (\mathsf{var\_f}\, w)\, e_1))) \rfloor \\
& \qquad \text{by lemma } \textit{subst-abs} \\
& = \lfloor \mathsf{subst}\, g\, x\, (\mathsf{abs}\, e_1) \rfloor
\end{aligned}
$$

Second, suppose that $x = y$. Since $y \notin \mathsf{fv}\, e_1$, it is also the case that $x \notin \mathsf{fv}\, e_1$.

$$
\begin{aligned}
& [\,\lfloor g \rfloor \,/\, x\,]\,\lfloor \mathsf{abs}\, e_1 \rfloor \\
& \qquad \text{for some } y \notin \mathsf{fv}\, e_1 \\
= {} & [\,\lfloor g \rfloor \,/\, x\,]\,(\lambda\, y \,.\, \lfloor \mathsf{open}\,(\mathsf{var\_f}\; y)\; e_1 \rfloor) \\
& \qquad \text{by definition of substitution} \\
= {} & \lambda\, y \,.\, \lfloor \mathsf{open}\,(\mathsf{var\_f}\; y)\; e_1 \rfloor \\
& \qquad \text{by definition of } \lfloor - \rfloor \\
= {} & \lfloor \mathsf{abs}\, e_1 \rfloor \\
& \qquad \text{by lemma } \textit{subst-fresh-eq} \\
= {} & \lfloor \mathsf{subst}\; g\; x\; (\mathsf{abs}\; e_1) \rfloor
\end{aligned}
$$

A.3 Proof of Theorem 5

By induction on the structure of M. The only interesting case is when $M = \lambda\, x \,.\, M_1$, for some M_1.

$$
\begin{aligned}
& \lfloor \lceil \lambda\, x \,.\, M_1 \rceil \rfloor \\
& \qquad \text{by definition of } \lceil - \rceil \\
= {} & \lfloor \mathsf{abs}\,(\mathsf{close}\; x\; \lceil M_1 \rceil) \rfloor \\
& \qquad \text{for some } y \notin \mathsf{fv}\,(\mathsf{close}\; x\; \lceil M_1 \rceil) \\
= {} & \lambda\, y \,.\, \lfloor \mathsf{open}\,(\mathsf{var\_f}\; y)\,(\mathsf{close}\; x\; \lceil M_1 \rceil) \rfloor \\
& \qquad \text{by lemma } \textit{subst-spec} \\
= {} & \lambda\, y \,.\, \lfloor \mathsf{subst}\,(\mathsf{var\_f}\; y)\; x\; \lceil M_1 \rceil \rfloor \\
& \qquad \text{by theorem 4} \\
= {} & \lambda\, y \,.\, [\, y \,/\, x \,]\, \lfloor \lceil M_1 \rceil \rfloor \\
& \qquad \text{by alpha conversion and lemma } \textit{fv-close} \\
=_\alpha {} & \lambda\, x \,.\, \lfloor \lceil M_1 \rceil \rfloor \\
& \qquad \text{by IH} \\
=_\alpha {} & \lambda\, x \,.\, M_1
\end{aligned}
$$

Note that in the alpha conversion step, we assumed that $x \neq y$. When $x = y$, the result follows trivially.

A.4 Proof of Theorem 6

By induction on the proof that e is locally closed. The only interesting case is when $e = \mathsf{abs}\, e_1$, for some e_1.

$$
\begin{aligned}
& \lceil \lfloor \mathsf{abs}\, e_1 \rfloor \rceil \\
& \qquad \text{for some } y \notin \mathsf{fv}\, e_1 \\
= {} & \lceil \lambda\, y \,.\, \lfloor \mathsf{open}\,(\mathsf{var\_f}\; y)\; e_1 \rfloor \rceil \\
& \qquad \text{by definition of } \lceil - \rceil \\
= {} & \mathsf{abs}\,(\mathsf{close}\; y\; \lceil \lfloor \mathsf{open}\,(\mathsf{var\_f}\; y)\; e_1 \rfloor \rceil) \\
& \qquad \text{by IH} \\
= {} & \mathsf{abs}\,(\mathsf{close}\; y\; (\mathsf{open}\,(\mathsf{var\_f}\; y)\; e_1)) \\
& \qquad \text{by lemma } \textit{close-open} \\
= {} & \mathsf{abs}\; e_1
\end{aligned}
$$

A.5 Proof of Theorem 11

We first observe that for any e, any two derivations of $\mathsf{lc}\, e$ are equal by *lc-unique*. Therefore, to show that each equality holds, it suffices to show that the first components of each side of the equality are equal. In the proofs below, we use _ as a place holder for the second components.

After unfolding definitions, parts 1, 2, and 3 are trivial.

For part 4, we have:

$$
\begin{aligned}
& \mathsf{Subst}\,(\mathsf{Lam}\,x\,t)\,(u,\ x) \\
& \qquad \text{decomposing } t \text{ and } u \text{ as } (e_1, \_) \text{ and } (e_2, \_) \\
&= \mathsf{Subst}\,(\mathsf{Lam}\,x\,(e_1, \_))\,((e_2, \_),\ x) \\
& \qquad \text{by definition} \\
&= (\mathsf{subst}\,e_2\,x\,(\mathsf{abs}\,(\mathsf{close}\,x\,e_1)), \_) \\
& \qquad \text{by lemmas } \textit{fv-close} \text{ and } \textit{subst-fresh-eq} \\
&= (\mathsf{abs}\,(\mathsf{close}\,x\,e_1), \_) \\
& \qquad \text{by lemma } \textit{lc-unique} \\
&= \mathsf{Lam}\,x\,t\,.
\end{aligned}
$$

For part 5, we have:

$$
\begin{aligned}
& \mathsf{Subst}\,(\mathsf{Lam}\,y\,t)\,(u,\ x) \\
& \qquad \text{decomposing } t \text{ and } u \text{ as } (e_1, \_) \text{ and } (e_2, \_) \\
&= \mathsf{Subst}\,(\mathsf{Lam}\,y\,(e_1, \_))\,((e_2, \_),\ x) \\
& \qquad \text{by definition} \\
&= (\mathsf{subst}\,e_2\,x\,(\mathsf{abs}\,(\mathsf{close}\,y\,e_1)), \_) \\
& \qquad \text{by definition of subst} \\
&= (\mathsf{abs}\,(\mathsf{subst}\,e_2\,x\,(\mathsf{close}\,y\,e_1)), \_) \\
& \qquad \text{by lemma } \textit{subst-close} \\
&= (\mathsf{abs}\,(\mathsf{close}\,y\,(\mathsf{subst}\,e_2\,x\,e_1)), \_) \\
& \qquad \text{by lemma } \textit{lc-unique} \\
&= \mathsf{Lam}\,y\,(\mathsf{Subst}\,t\,(u,\ x))\,.
\end{aligned}
$$

A.6 Proof of Theorem 12

We first decompose t as (e, lcp). By unfolding definitions and making use of lemma *lc-unique*, as we did in the proof of theorem 11, we must show that

$$\mathsf{abs}\,(\mathsf{close}\,x\,e) \ = \ \mathsf{abs}\,(\mathsf{close}\,y\,(\mathsf{subst}\,(\mathsf{var\_f}\,y)\,x\,e))$$

under the assumption that $y \notin (\mathsf{fv}\,e) \setminus \{\,x\,\}$, i.e., that $y \notin \mathsf{fv}\,(\mathsf{close}\,x\,e)$ (recall lemma *close-fv*). Starting with the right-hand side of the conclusion, we have the following chain of equalities:

$$
\begin{aligned}
& \mathsf{abs}\,(\mathsf{close}\,y\,(\mathsf{subst}\,(\mathsf{var\_f}\,y)\,x\,e)) \\
& \qquad \text{by lemma } \textit{subst-spec} \\
&= \mathsf{abs}\,(\mathsf{close}\,y\,(\mathsf{open}\,(\mathsf{var\_f}\,y)\,(\mathsf{close}\,x\,e))) \\
& \qquad \text{by lemma } \textit{close-open} \\
&= \mathsf{abs}\,(\mathsf{close}\,x\,e)\,.
\end{aligned}
$$

A.7 Proof of Theorem 13

The function f is derived from the recursion scheme given to use by $\mathsf{lc\_set}$—recall Fig. 3. We define f by instantiating P with $(\lambda\_.\,R)$ and by rearranging the arguments of the Gordon-Melham cases:

$$
\begin{aligned}
& f\,(e, lcp) = \mathsf{lc\_set\_rec}\,\textit{fvar}\,\textit{fapp}'\,\textit{fabs}'\,e\,lcp \\
& \quad \text{where} \quad \textit{fapp}' \ = \ \lambda e_1, e_2, lcp_1, r_1, lcp_2, r_2.\,\textit{fapp}\,r_1\,r_2\,(e_1, lcp_1)\,(e_2, lcp_2) \\
& \qquad\qquad\quad \textit{fabs}' \ = \ \lambda e_1, lcp_1, r_1.\,\textit{fabs}\,r_1\,(\lambda x.\,(\mathsf{open}\,(\mathsf{var\_f}\,x)\,e_1, lcp_1\,x))
\end{aligned}
$$

The uniqueness of this operator is by definition. Furthermore, suppose f is an operator defined as above. Showing the equalities in the Var and App cases is straightforward. For the Lam case, suppose the body of the Term is $t = (e_1, lcp_1)$, and let f' be $\mathsf{lc\_set\_rec}\,\textit{fvar}\,\textit{fapp}'\,\textit{fabs}'$. Using *lc-unique* to ignore local closure proofs, much as we did in the proof of theorem 11, we have the following:

$$
\begin{aligned}
& f\,(\mathsf{Lam}\,x\,(e_1, \_)) \\
& \qquad \text{by definition} \\
& = f'\,(\mathsf{abs}\,(\mathsf{close}\,x\,e_1))\,\_ \\
& \qquad \text{by property of } \mathsf{lc\_set\_rec} \\
& = \mathit{fabs}'\,(\mathsf{close}\,x\,e_1)\,\_\,(\lambda y.f\,(\mathsf{open}\,(\mathsf{var\_f}\,y)\,(\mathsf{close}\,x\,e_1), \_)) \\
& \qquad \text{by definition of } \mathit{fabs}' \\
& = \mathit{fabs}\,(\lambda y.f\,(\mathsf{open}\,(\mathsf{var\_f}\,y)\,(\mathsf{close}\,x\,e_1), \_))\,(\lambda y.(\mathsf{open}\,(\mathsf{var\_f}\,y)\,(\mathsf{close}\,x\,e_1), \_)) \\
& \qquad \text{by lemma } \textit{subst-spec} \\
& = \mathit{fabs}\,(\lambda y.f\,(\mathsf{subst}\,(\mathsf{var\_f}\,y)\,x\,e_1, \_))\,(\lambda y.(\mathsf{subst}\,(\mathsf{var\_f}\,y)\,x\,e_1, \_)) \\
& \qquad \text{by definition of } \mathsf{Subst} \\
& = \mathit{fabs}(\lambda y.f\,(\mathsf{Subst}\,t\,(\mathsf{Var}\,y,\ x)))(\lambda y.(\mathsf{Subst}\,t\,(\mathsf{Var}\,y,\ x)))
\end{aligned}
$$

A.8 Proof of Theorem 15

We decompose t as $(e_1, \_)$ and make use of lemma *lc-unique* in the same way we did as in the proof of theorem 11.

$$
\begin{aligned}
& \mathsf{Abs}\,(\lambda y.\,\mathsf{Subst}\,t\,(\mathsf{Var}\,y,\ x)) \\
& \qquad \text{by definition of } \mathsf{Abs} \text{ and } \mathsf{Subst} \\
& \qquad \text{for some } y \notin \mathsf{Fv}\,(\mathsf{Subst}\,t\,(\mathsf{Var}\,x_0,\ x)) \\
& = (\mathsf{abs}\,(\mathsf{close}\,y\,(\mathsf{subst}\,(\mathsf{var\_f}\,y)\,x\,e_1)), \_) \\
& \qquad \text{by lemma } \textit{subst-spec} \\
& = (\mathsf{abs}\,(\mathsf{close}\,y\,(\mathsf{open}\,(\mathsf{var\_f}\,y)\,(\mathsf{close}\,x\,e_1))), \_) \\
& \qquad \text{by lemma } \textit{close-open}, \\
& \qquad \text{discharging the side condition by lemmas } \textit{fv-close} \text{ and } \textit{fv-subst-lower} \\
& = (\mathsf{abs}\,(\mathsf{close}\,x\,e_1), \_) \\
& \qquad \text{by definition of } \mathsf{Lam} \text{ and lemma } \textit{lc-unique} \\
& = \mathsf{Lam}\,x\,t
\end{aligned}
$$