

Mechanical Proof of the Optimality of a Partial Evaluator

Sebastian C. Skalberg

24th February 1999

Preface

The following paper is a master's thesis, and is turned in as partial fulfilment of the requirements for the danish master's degree (cand.scient) at DIKU, the Department of Computer Science at the University of Copenhagen. It reports on work done in the period of October 1998 through February 1999, with Professor Neil D. Jones as supervisor.

We present a proof of the optimality of *lambda-mix*, Gomard's partial evaluator for an un-typed applied lambda calculus. We also report on a mechanically verified version of the proof, which was done using Isabelle/HOL, the typed higher order logic instance of the generic proof system Isabelle. As far as we know, this is the first proof of the optimality of a partial evaluator. In addition, we have found only few references to other mechanical proofs involving concrete programs, eg. pieces of code. Thus, the thesis should be of interest to researchers working in both partial evaluation and automated proof systems.

The paper contains short introductions to both partial evaluation and automated proof systems. We thereby hope that readers new to either field can understand the motivation for—and follow the presentation of—the proofs given here. We assume that the reader has some knowledge in the fields of logic, lambda calculus, and semantics.

Acknowledgements

First and foremost I would like to thank Neil D. Jones for providing an excellent topic for the thesis, and the fruitful input he has provided regarding all aspects of the work.

Olivier Danvy, the external examiner, for being very patient and helpful in the last phase of the thesis.

Larry Paulson, Tobias Nipkow, and the rest of the Isabelle community for providing swift and precise answers to many questions pertaining to Isabelle. The mechanized proof would certainly not have been possible by my hand without them.

Jens Ulrik Skakkebak for giving an introduction to the PVS system, and for providing initial pointers to information on automated proof systems.

Jakob Grue Simonsen and Peter Møller Neergaard for proofreading earlier (and later!) drafts. Both have given valuable comments and suggestions that helped improve the exposition.

Peter Møller Neergaard for answering my endless questions regarding \TeX and \LaTeX , and for co-writing the semantic package, which I have used extensively. Thanks are also due to the rest of the “penthouse-gang” for providing a pleasant place to be (and *many* meals): Jarl Friis, Bo Kjellerup, Signe Schmidt Nielsen, and Jakob Grue Simonsen.

Finally, and mostly, to my girl-friend Hanne for her love and patience.

Contents

1 Introduction	2
1.1 Overview	2
2 Partial Evaluation	3
2.1 Theory	3
2.2 Practice	3
2.3 Optimality	4
3 Lambda-mix	5
3.1 Background	5
3.2 Syntax	6
3.3 Semantics	9
3.4 Evaluation	12
3.5 Optimality	13
4 Paper Proof	14
4.1 Preliminaries	14
4.2 Lambda-mix Environments	14
4.3 Evaluation Lemmata	17
4.4 Proof of Optimality	20
4.4.1 Termination	20
4.4.2 Uniqueness	21
5 Mechanical Proof	28
5.1 Choice of Proof System	28
5.2 Lambda-mix	28
5.2.1 Syntax	28
5.2.2 Semantics	33
5.2.3 Optimality	33
5.3 Proof of Optimality	33
5.3.1 Lambda-mix Environments	33
5.3.2 The Proof	36
5.4 Summary	37
6 Conclusion	38
A Program Listings	39
A.1 The Self-Interpreter	39
A.2 The Annotated Self-Interpreter	40
B A Quick Overview of Isabelle/HOL	42

C	Isabelle Proof Scripts	45
C.1	LMixEnv	46
C.2	Preliminary Definitions	50
C.3	SExpr	53
C.4	Alpha	54
C.5	L1Expr	55
C.6	L2Eval	57
C.7	LMix	60
C.8	LemRen	60
C.9	NatInf	70
C.10	Rename	70
C.11	Sint	71
C.12	Miscellaneous Proofs	73
Last Page	120

Chapter 1

Introduction

In 1987 the first international workshop on partial evaluation was held in Gl. Avernæs, Denmark. The proceedings included a list of challenging problems, one of which was that of constructing a partial evaluator that is “strong enough”:

Partial evaluation of a self-interpreter for the subject language with respect to a program should be able to yield essentially the same program as output.

(Jones [12], problem 3.8). A formal definition has since been given (and is repeated in section 2.3), and partial evaluators satisfying the above property are today called *optimal*.

While several partial evaluators are thought to be optimal, no proof of optimality has yet been given. This was therefore chosen as the subject for the present thesis: to prove that an optimal partial evaluator exists. We have succeeded in providing such a proof, for an existing partial evaluator.

The partial evaluator we have proven optimal is *lambda-mix*, first presented in 1989 by Carsten Gomard in his master’s thesis [6]. It is a partial evaluator for the untyped lambda calculus, and it was chosen because it was believed optimal, and for its simplicity. Thus, a proof of the optimality of lambda-mix seemed feasible.

To (hopefully) avoid any future dispute over the correctness of the proof, we have chosen to mechanically verify it, using one of the many automated proof systems. Though no actual errors were found during this verification, we have, in fact, unearthed several points in the proof where reasoning beyond the anticipated was required. The process of mechanically verifying the proof has therefore not only resulted in a verified proof, but also improved on the exposition of the original proof.

1.1 Overview

Chapter two is a short introduction to partial evaluation, including a formal definition of optimality of a partial evaluator.

In chapter three we define the version of lambda-mix which we use in the rest of the thesis. We also state the two main theorems of the thesis, that amount to the optimality of lambda-mix.

Chapter four contains the actual proof of the two theorems mentioned above.

Chapter five documents the work in formalizing chapter three and subsequently mechanically verifying the proof of chapter four.

Finally, we conclude in chapter six and refer to related work.

Chapter 2

Partial Evaluation

We will now provide a quick introduction to partial evaluation. Readers interested in a thorough presentation are referred to (Jones, et al. [15]).

2.1 Theory

A partial evaluator *mix* is basically an implementation of Kleene’s *s-m-n* theorem, ie. it satisfies the equation

$$\llbracket p \rrbracket(s, d) = \llbracket \llbracket mix \rrbracket(p, s) \rrbracket(d) \quad (2.1.1)$$

for all programs p and inputs s and d . The equation says that if we divide the input of program p in a *static* part s and *dynamic* part d , we can computably construct a new program $p_s = \llbracket mix \rrbracket(p, s)$. The program p_s is called the *residual* or *partially evaluated* program. When given input d , it returns the same result as the original program does given both s and d , ie. $\llbracket p_s \rrbracket = \llbracket p \rrbracket(s, \cdot)$.

In 1971 Yoshihiko Futamura, in [5], was the first to observe that such a program *mix* could, in theory, be used to compile, generate compilers from interpreters, and generate compiler-generators, programs that take an interpreter as input and return a “corresponding” compiler. The now famous Futamura projections are

$$\llbracket mix \rrbracket(int, p) = p' \quad (2.1.2)$$

$$\llbracket mix \rrbracket(mix, int) = comp \quad (2.1.3)$$

$$\llbracket mix \rrbracket(mix, mix) = cogen \quad (2.1.4)$$

Actually implementing these projections is difficult. It was not until 1984 that Neil Jones, Peter Sestoft, and Harald Søndergaard succeeded in constructing a partial evaluator able to implement the third Futamura projection (equation 2.1.4 above).

2.2 Practice

Theory is one thing, actually constructing a partial evaluator satisfying equation 2.1.1 is a whole different story. A partial evaluator has to decide, given a program and partial input, which computations specified in the program are to be taken at *partial evaluation time*, and which should postponed to *run time* due, eg. to insufficient data? Here, “partial evaluation time” is when the partial evaluator is run, while “run time” is used when the program generated by the partial evaluator, the so-called *residual* program, is run.

The partial evaluator we present is an *off-line* partial evaluator, meaning it operates in two stages: *first* it decides which computations to take, *then* it just follows the decisions made earlier.

The information can propagate from the first to the second stage in several ways. The one used by lambda-mix, and thus by us, is that of *annotating* the source program based on which input is available. Note that in the first stage, what is important is which input is available, not the actual values of these input.

The annotations are made possible by defining a two-level syntax for the language in question. Each construct in the original language is found twice in the two-level language, since occurrences in source programs can be marked as either static (do the computation now, at partial evaluation time) or dynamic (postpone the computation til run time). Such two-level languages also need a lifting construct, that creates a piece of code from a piece of data, which is needed whenever a static computation is found where a dynamic was expected.

2.3 Optimality

A natural question, when given a partial evaluator *mix*, is of course: does a *better* exist? First we will have to be more specific as to what we mean by a better partial evaluator: we will say that *mix* is at least as good as another partial evaluator *mix'* when, for all programs *p* and input *s* and *d*, we have $T_{p_s}(d) \leq T_{p'_s}(d)$, where $p_s = \llbracket mix \rrbracket(p, s)$ and $p'_s = \llbracket mix' \rrbracket(p, s)$, and where $T_p(d)$ is some measure of the time taken to execute program *p* on input *d*,

We now move on to ask the next logical question: is there an *optimal* partial evaluator, ie. one that is at least as good as all other partial evaluators? The answer to this question is a clear *no*. To see this, consider a program *p* with a single input parameter. Specializing *p* wrt. some *s* is thus simply interpreting *p* with input *s*. An optimal partial evaluator would have to decide whether *p* terminates with input *s*, a problem which we know to be undecidable. (Jones [14] contains a good introduction to computability theory.)

We can however state a definition of optimality that is satisfiable and also meaningful. By the first Futamura projection (equation 2.1.2, above), we see that compilation is possible by a partial evaluator, given an interpreter. When the interpreter in question is in fact a self-interpreter, the residual program will be in the same language as the original program, and thus a comparison of the efficiency of the two programs make sense. The definition of optimality given here is taken from (Jones, et al. [15]) and states that for all input programs, the partial evaluator *mix* removes all interpretational overhead from the self-interpreter *sint*.

Definition 2.3.1 (Jones, et al.) *A partial evaluator mix is said to be optimal if there exists a self-interpreter sint, such that for all programs p and input d, we have*

$$T_{sint_p}(d) \leq T_p(d),$$

where $sint_p = \llbracket mix \rrbracket(sint, p)$.

Of course, this definition can be “cheated”, by modifying any partial evaluator to return its second argument in case the first is equal to *sint*. Such partial evaluators are not considered in this thesis.

While optimality in the above sense may seem unobtainable, several partial evaluators have been believed to be just that, although no partial evaluator has actually been proven optimal, before now. Among the best known (presumed) optimal partial evaluators are lambda-mix (Gomard [6, 8, 9]; Gomard and Jones [10]) and the partial evaluators for the pure lambda calculus found in (Mogensen [16, 17]).

Chapter 3

Lambda-mix

In the present chapter, we present lambda-mix in the form which we study in the remainder of the thesis. Lambda-mix was originally defined by Gomard in his master's thesis [6]. We will make an effort to keep our presentation as close as possible to the one given there. We will, however, need to modify lambda-mix in certain areas, most notably we will use an operational semantics, rather than a denotational as Gomard did. We return to this and other deviances as we encounter them. Though we present no proof, we claim that the subject language and partial evaluator defined are essentially those of Gomard's lambda-mix, and will use the name lambda-mix of both systems.

After a brief introduction to the background of lambda-mix and work related to it, we define the syntax and semantics of the subject language considered in this thesis. We then define a two-level version of the subject language, together with a suitable two-level semantics. Essentially defining a partial evaluator. We conclude by stating the main theorems of the thesis: termination and uniqueness, and give an intuitive argument for their validity.

3.1 Background

While the first self-applicable partial evaluator for a first order language was constructed back in 1984 as mentioned in the previous chapter, a self-applicable partial evaluator for a higher order language was not obtained prior to 1989. This was when Gomard reported on lambda-mix, a partial evaluator for an applied lambda calculus. Two self-applicable partial evaluators for the Scheme language were developed at the same time: Similix (Bondorf [2]) and Schism (Consel [3]). We will not say more about the two latter systems.

Gomard used an untyped lambda calculus with constants, conditionals, and a fixed point operator as his subject language, and defined a partial evaluator for a two-level version of this language. When specializing programs, these were first annotated, ie. converted to two-level programs, by typing them wrt. a type inference system. This was done using a modified version of Milner's algorithm \mathcal{W} (Damas and Milner [4]), and is reported in (Gomard [7]).

A natural question to ask after obtaining a self-applicable partial evaluator for an applied lambda calculus is if such a partial evaluator could be found for the *pure* lambda calculus. (Or rather, since the pure lambda calculus is known to be Turing complete, how such a partial evaluator could be constructed.) Inspired by lambda-mix, Mogensen presented an offline self-applicable partial evaluator for the pure lambda calculus in (Mogensen [16]), using higher order abstract syntax (Pfenning and Elliot [27]) to represent lambda terms. A few years later, he succeeded in constructing an *online* self-applicable partial evaluator for the pure lambda calculus, as reported in (Mogensen [17]).

For readers interested in the definition of Gomard's lambda-mix, the original presentation can be found in (Gomard [6]). Good presentations can also be found in (Gomard [8, 9]), (Gomard

and Jones [10]), and (Jones, et al. [15]).

3.2 Syntax

Following the example of (Andersen [1]), we define our subject language using a concrete syntax based on so-called S-expressions, known from Lisp and Scheme. There are several reasons for this. First of all, we want to prove properties of a concrete self-interpreter, so we might as well be concrete on the syntax from the very beginning. As for choosing S-expressions, they are simple to define and work with, programs are easily viewed as data, and Gomard implemented lambda-mix in Scheme (for the two previous reasons). Thus, by using S-expressions, we keep close to Gomard's presentation, and are therefore able to use, eg. his self-interpreter relatively unmodified.

Having decided for S-expressions, we now give a formal definition of them.

Definition 3.2.1 (S-Expressions) *Given a set \mathbb{A} of syntactic constructs, we define the set $\mathbb{D}_{\mathbb{A}}$ of S-expressions (over \mathbb{A}) to be the smallest set satisfying the following two recursive conditions:*

$$\mathbb{A} \subset \mathbb{D}_{\mathbb{A}}$$

and

$$\forall d_1, d_2 \in \mathbb{D}_{\mathbb{A}} : (d_1 . d_2) \in \mathbb{D}_{\mathbb{A}}.$$

We call elements of $\mathbb{D}_{\mathbb{A}}$ of the form $(d_1 . d_2)$ *pairs*, while elements originally from the set \mathbb{A} are called *atoms*. (We make the restriction on the set \mathbb{A} that for no two elements a_1 and a_2 from \mathbb{A} does $(a_1 . a_2) \in \mathbb{A}$ hold, ie. no element of $\mathbb{D}_{\mathbb{A}}$ is simultaneously an atom and a pair.) We define equality between S-expressions to be syntactic equality, eg. $a = a$ and $(a . b) = (a . b)$, but $(a . b) \neq (b . a)$. Finally, whenever the set \mathbb{A} is clear from the context, we shall be inclined to drop the index from $\mathbb{D}_{\mathbb{A}}$.

Following standard practice, when the set \mathbb{A} includes the atom `nil`, a list d_1, \dots, d_n of S-expressions is coded as the S-expression $(d_1 . (d_2 . (\dots (d_n . \text{nil}) \dots))$ which is by notational convention written as

$$(d_1 \ d_2 \ \dots \ d_n).$$

As a special case, $()$ denotes the empty list, ie. the atom `nil`. Of course, the d_i 's above may themselves be representations of lists.

Before defining our subject language, we need to impose some restrictions on the set of atoms used. Like all other programming languages, our subject language requires certain syntactic constructs to be available. These are given by the set \mathbb{S}_1 below. Anticipating the two-level language, we will already require the existence of all atoms found in the set \mathbb{S}_2 . We will call the elements of \mathbb{S}_2 *symbols*.

$$\begin{aligned} \mathbb{S}_0 &= \{\text{clos}, \text{delay}, \text{nil}, \#\text{t}\} \\ \mathbb{S}_1 &= \mathbb{S}_0 \cup \{\text{lamb}, \text{@}, \text{fix}, \text{if}, \text{quote}, \text{cons}, \text{car}, \text{cdr}, \text{eq?}, \text{atom?}, \text{error}\} \\ \mathbb{S}_2 &= \mathbb{S}_1 \cup \{\underline{\text{lamb}}, \underline{\text{@}}, \underline{\text{fix}}, \underline{\text{if}}, \underline{\text{quote}}, \underline{\text{cons}}, \underline{\text{car}}, \underline{\text{cdr}}, \underline{\text{eq?}}, \underline{\text{atom?}}, \underline{\text{error}}\} \cup \{\text{lift}\} \end{aligned}$$

We will, as pointed out earlier, also need to reason about a concrete self-interpreter. We therefore require that the following atoms, that are used as variables in the self-interpreter, also occur in the set of atoms we consider:

$$\mathbb{V}_{\text{sint}} = \{\text{eval}, \text{expr}, \text{env}, \text{value}, \text{var}, \text{x}\}.$$

In the remaining part of the thesis, \mathbb{A} will denote some infinite set of atoms with $\mathbb{V}_{\text{sint}} \cup \mathbb{S}_2 \subset \mathbb{A}$. All atoms that are not symbols, ie. all elements of the set $\mathbb{A} \setminus \mathbb{S}_2$, are called *variables*, and the

```

 $\mathbb{L}_1 ::= \mathbb{V}$ 
      | (quote  $\mathbb{D}$ )
      | (lam  $\mathbb{V} \mathbb{L}_1$ )
      | (@  $\mathbb{L}_1 \mathbb{L}_1$ )
      | (fix  $\mathbb{L}_1$ )
      | (if  $\mathbb{L}_1 \mathbb{L}_1 \mathbb{L}_1$ )
      | (cons  $\mathbb{L}_1 \mathbb{L}_1$ )
      | (car  $\mathbb{L}_1$ )
      | (cdr  $\mathbb{L}_1$ )
      | (eq?  $\mathbb{L}_1 \mathbb{L}_1$ )
      | (atom?  $\mathbb{L}_1$ )
      | (error  $\mathbb{L}_1$ )

```

Figure 3.1: Concrete syntax for \mathbb{L}_1 -expressions.

set of all variables will be denoted by \mathbb{V} . Thus, $\mathbb{V}_{\text{sint}} \subset \mathbb{V}$, and since \mathbb{S}_2 is finite, it follows that \mathbb{V} is infinite.

We are now ready to define the syntax of our subject language, which we shall call \mathbb{L}_1 . The concrete syntax for \mathbb{L}_1 -expressions can be found in figure 3.1, and is very similar to the syntax given by Gomard. Gomard uses constants to extend the lambda calculus. Thus, the operations `cons`, `car`, `cdr`, etc. are all bound in the environment. However, this gives rise to one important problem: lambda-mix, as defined by Gomard, is not optimal as it stands, but requires a post-processing stage after specialization. As explained in (Mogensen [18]), the problem is an *inherited limit*: the syntax given by Gomard makes no restrictions on the number of constants used in a program, while a self-interpreter, no matter how constructed, must necessarily contain a fixed number of such constants.

Rather than adding a post-processing phase (simple as it may be), we prefer to remedy the problem by incorporating the needed constants in the actual syntax of the language.¹ Thus, instead of writing, eg.

```
(@ (@ cons x) y)
```

one writes

```
(cons x y).
```

The price of this is that our subject language has twelve constructors, while Gomard's only has seven—on the other hand, \mathbb{L}_1 -expressions are shorter and easier to read than corresponding programs in Gomard's subject language. In addition, our partial evaluator is optimal, which is of course the main benefit of this modification.

Moving on, we need the following map that collects all variables found in an S-expression. It is used both in the definition of the two-level semantics, and when defining α -equivalence of \mathbb{L}_1 -expressions, which we will also define shortly. First the definition of the map *vars*, where $\mathcal{P}(X)$ denotes the powerset of X :

¹Strictly speaking, we do not need the `cons` construct for our purposes, but it seemed odd to have a language with `car` and `cdr`, but no `cons`, so we left it in.

Definition 3.2.2 The map $\text{vars} : \mathbb{D}_{\mathbb{A}} \rightarrow \mathcal{P}(\mathbb{V})$ is given by

$$\begin{aligned} \text{vars } v &= \{v\} \cap \mathbb{V} \\ \text{vars } (d_1 . d_2) &= \text{vars } d_1 \cup \text{vars } d_2 \end{aligned}$$

It is easy to see that for all lists $(d_1 d_2 \cdots d_n)$, we have

$$\text{vars } (d_1 d_2 \cdots d_n) = \text{vars } d_1 \cup \text{vars } d_2 \cup \cdots \cup \text{vars } d_n$$

For the purpose of defining α -equivalence of \mathbb{L}_1 -expressions, we define renaming of variables:

Definition 3.2.3 (Renaming) The renaming of variable v to variable v' in the \mathbb{L}_1 -expression M , written $M[v := v']$, is defined by

$$\begin{aligned} v[v := v'] &= v' \\ x[v := v'] &= x, \quad \text{if } x \neq v \\ (\text{quote } d)[v := v'] &= (\text{quote } d) \\ (\text{lam } v M)[v := v'] &= (\text{lam } v M) \\ (\text{lam } x M)[v := v'] &= (\text{lam } x M[v := v']), \quad \text{if } x \neq v \\ (@ P Q)[v := v'] &= (@ P[v := v'] Q[v := v']) \\ (\text{fix } P)[v := v'] &= (\text{fix } P[v := v']) \\ (\text{if } P Q R)[v := v'] &= (\text{if } P[v := v'] Q[v := v'] R[v := v']) \\ (\text{cons } P Q)[v := v'] &= (\text{cons } P[v := v'] Q[v := v']) \\ (\text{car } P)[v := v'] &= (\text{car } P[v := v']) \\ (\text{cdr } P)[v := v'] &= (\text{cdr } P[v := v']) \\ (\text{eq? } P Q)[v := v'] &= (\text{eq? } P[v := v'] Q[v := v']) \\ (\text{atom? } P)[v := v'] &= (\text{atom? } P[v := v']) \\ (\text{error } P)[v := v'] &= (\text{error } P[v := v']) \end{aligned}$$

This definition is completely standard, and we therefore hasten to continue with the definition of α -equivalence of \mathbb{L}_1 -expressions:

Definition 3.2.4 (α -equivalence) We define α -equivalence, written $=_{\alpha}$, to be the least equivalence relation defined on $\mathbb{L}_1 \times \mathbb{L}_1$, and closed under the following rules for all variables v and x and \mathbb{L}_1 -expressions M, M', N, N', P , and P' :

$$\begin{aligned} M =_{\alpha} M' &\implies (\text{lam } v M) =_{\alpha} (\text{lam } v M') \\ v \notin \text{vars } M &\implies (\text{lam } x M) =_{\alpha} (\text{lam } v M[x := v]) \\ M =_{\alpha} M' \wedge N =_{\alpha} N' &\implies (@ M N) =_{\alpha} (@ M' N') \\ M =_{\alpha} M' &\implies (\text{fix } M) =_{\alpha} (\text{fix } M') \\ M =_{\alpha} M' \wedge N =_{\alpha} N' \wedge P =_{\alpha} P' &\implies (\text{if } M N P) =_{\alpha} (\text{if } M' N' P') \\ M =_{\alpha} M' \wedge N =_{\alpha} N' &\implies (\text{cons } M N) =_{\alpha} (\text{cons } M' N') \\ M =_{\alpha} M' &\implies (\text{car } M) =_{\alpha} (\text{car } M') \\ M =_{\alpha} M' &\implies (\text{cdr } M) =_{\alpha} (\text{cdr } M') \\ M =_{\alpha} M' \wedge N =_{\alpha} N' &\implies (\text{eq? } M N) =_{\alpha} (\text{eq? } M' N') \\ M =_{\alpha} M' &\implies (\text{atom? } M) =_{\alpha} (\text{atom? } M') \\ M =_{\alpha} M' &\implies (\text{error } M) =_{\alpha} (\text{error } M') \end{aligned}$$

$$\begin{aligned}
\mathbb{L}_2 ::= & \mathbb{L}_1 \\
& | \text{ (lift } \mathbb{L}_1 \text{)} \\
& | \text{ (quote } \mathbb{D} \text{)} \\
& | \text{ (lam } \forall \mathbb{L}_2 \text{)} \\
& | \text{ (@ } \mathbb{L}_2 \mathbb{L}_2 \text{)} \\
& | \text{ (fix } \mathbb{L}_2 \text{)} \\
& | \text{ (if } \mathbb{L}_2 \mathbb{L}_2 \mathbb{L}_2 \text{)} \\
& | \text{ (cons } \mathbb{L}_2 \mathbb{L}_2 \text{)} \\
& | \text{ (car } \mathbb{L}_2 \text{)} \\
& | \text{ (cdr } \mathbb{L}_2 \text{)} \\
& | \text{ (eq? } \mathbb{L}_2 \mathbb{L}_2 \text{)} \\
& | \text{ (atom? } \mathbb{L}_2 \text{)} \\
& | \text{ (error } \mathbb{L}_2 \text{)}
\end{aligned}$$

Figure 3.2: Concrete syntax for \mathbb{L}_2 -expressions.

Note that we in the above definition consider the least *equivalence* relation $=_\alpha$, thus $=_\alpha$ is reflexive, symmetric, and transitive. Also, the above definition is smaller than would normally be expected, because of the $v \notin \text{vars } M$ in the second line. In effect, this means that

$$(\text{lam } x \text{ (lam } x \text{ x)}) \neq_\alpha (\text{lam } y \text{ (lam } x \text{ x)})$$

which it is contrary to “ordinary” α -equivalence. It should, on the other hand, be clear that the relation defined above is a restriction of the ordinary α -equivalence relation. Thus, if we can prove that \mathbb{L}_1 -expressions M and M' are α -equivalent by the above definition, they are also α -equivalent in the ordinary sense. We have chosen the above formulation as it simpler than the ordinary definition (we need not define “free variables”), and it serves our purposes, as will become clear later.

Concluding this section is the definition of the syntax of the two-level language, which we denote by \mathbb{L}_2 . The concrete syntax of \mathbb{L}_2 -expressions can be found in figure 3.2, and extends \mathbb{L}_1 in ways completely analogous to the development by Gomard. Hence, we also have both a residual quote and a lift construct, though the latter renders the former superfluous.

3.3 Semantics

As already mentioned in the beginning of the chapter, we have chosen to use operational semantics, rather than denotational semantics, in our presentation here. The reasons for this are twofold: Proving optimality of our partial evaluator means reasoning about the way the self-interpreter *operates*, which is easier to do in the framework of operational semantics (hence, the name). On top of that, not only we, but also an automated proof assistant must be able to reason on the basis of the semantics. As this would be our first encounter with such an assistant, we ventured that we would have enough hassle with such an assistant using an operational semantics, much more so using a denotational semantics. For these reasons, we now restate the semantics given by Gomard in an operational setting. The overall structure of the semantics

will otherwise be the same, eg. we still use environments to bind values to variables, rather than do textual substitution. Finally, we should note that the operational semantics given here is call-by-value.

Environments will be represented in a standard way, as a list of (variable, value) pairs, eg. the environment $[x \mapsto d_x, y \mapsto d_y]$ is represented by the S-expression $((x.d_x) (y.d_y))$. The lookup function is then defined as:

Definition 3.3.1 *The partial map $lookup : \mathbb{D}_A \times \mathbb{V} \rightarrow \mathbb{D}_A$ is given by*

$$\begin{aligned} lookup(\mathbf{nil}, v) &= v \\ lookup((x.d_x) . \rho), v) &= \begin{cases} d_v & \text{if } v = x \\ lookup(\rho, v) & \text{otherwise} \end{cases} \end{aligned}$$

If the first argument to $lookup$ is not a valid environment, the result is undefined.

Note that variables are by default bound to themselves. This is necessary in order to handle non-closed \mathbb{L}_1 -expressions later on.

An important consequence of restating the original semantics as operational semantics can already be seen in the semantics for the \mathbb{L}_1 -expressions, found in figure 3.3 on the facing page, in rule 3.3.6. While fixed points are naturally represented in denotational semantics, this is not so in operational semantics. We have followed standard practice, and made a special fixed point closure, which we then handle separately in the variable case (see rules 3.3.6 and 3.3.2). As long as evaluating the body of the fixed point operator terminates, we have successfully modelled the denotational semantics. If, on the other hand, the evaluation of the body loops, this leaves us with a problem wrt. the original semantics: Consider the expression

$$(@ (\mathbf{lam} \ x \ (\mathbf{lam} \ y \ y)) (\mathbf{fix} \ M)).$$

If the evaluation of M does not terminate, the semantics given here will result in the entire expression above not terminating, though the semantics given by Gomard do (since fixed points in denotational semantics do not have termination properties, as such). The problem is, in short, that we need to evaluate M in order to find the fixed point variable, eg. v in $(\mathbf{fix} \ (\mathbf{lam} \ v \ M))$.

The obvious solution is to make the variable explicit, eg. $(\mathbf{fix} \ f \ M)$. However, this complicates the semantics of the two-level language presented, as well as the self-interpreter. Considering this as a too severe modification of the original lambda-mix, we abandon this solution.

The solution we have used is to restrict the expression M found in $(\mathbf{fix} \ M)$ to be an abstraction. This way, evaluation of M always terminates, by rule 3.3.4. As the only fixed point expression we will consider has this form (see the expression found in appendix A.1), our semantics agree with Gomard's on this *particular* program.

We now move on to the semantics of \mathbb{L}_2 -expressions. Since \mathbb{L}_2 is our two-level language, the semantics is in fact the partial evaluator for \mathbb{L}_1 . Thus, for the \mathbb{L}_1 -subset of \mathbb{L}_2 , ie. the static expressions, we use the semantics already given in figure 3.3. For the rest of the \mathbb{L}_2 -expressions, we use the semantics given in figure 3.4 on page 12. These semantics are, more or less, a direct translation of the semantics given by Gomard. We have, though, defined the residual abstraction semantics, rule 3.3.18, formally where Gomard settled for an informal description. By requiring

$$v_{\text{new}} \in \mathbb{V} \setminus (\text{vars } \rho \cup \text{vars } M)$$

we make sure that v_{new} is indeed new—the environment ρ holds information on all variables that have been seen so far, and the expression M , of course, contains all variables we might encounter, including free variables.

$\frac{\text{lookup}(\rho, v) = d \quad d \in \mathbb{A} \vee (d = (d_1 . d_2) \wedge d_1 \neq \text{delay})}{\rho \vdash v \longrightarrow d}$	(3.3.1)
$\frac{\text{lookup}(\rho, v) = (\text{delay } M \ \rho') \quad \rho' \vdash M \longrightarrow d}{\rho \vdash v \longrightarrow d}$	(3.3.2)
$\frac{}{\rho \vdash (\text{quote } d) \longrightarrow d}$	(3.3.3)
$\frac{}{\rho \vdash (\text{lam } v \ M) \longrightarrow (\text{clos } v \ M \ \rho)}$	(3.3.4)
$\frac{\rho \vdash M \longrightarrow (\text{clos } v \ M' \ \rho') \quad \rho \vdash N \longrightarrow d_N \quad ((v . d_N) . \rho') \vdash M' \longrightarrow d}{\rho \vdash (@ \ M \ N) \longrightarrow d}$	(3.3.5)
$\frac{\rho \vdash M \longrightarrow (\text{clos } v \ M' \ \rho') \quad ((v . (\text{delay } (\text{fix } M) \ \rho)) . \rho') \vdash M' \longrightarrow d}{\rho \vdash (\text{fix } M) \longrightarrow d}$	(3.3.6)
$\frac{\rho \vdash M \longrightarrow d_M \quad d_M \neq \#t \quad \rho \vdash P \longrightarrow d_P}{\rho \vdash (\text{if } M \ N \ P) \longrightarrow d_P}$	(3.3.7)
$\frac{\rho \vdash M \longrightarrow \#t \quad \rho \vdash N \longrightarrow d_N}{\rho \vdash (\text{if } M \ N \ P) \longrightarrow d_N}$	(3.3.8)
$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N}{\rho \vdash (\text{cons } M \ N) \longrightarrow (d_M . d_N)}$	(3.3.9)
$\frac{\rho \vdash M \longrightarrow (d_1 . d_2)}{\rho \vdash (\text{car } M) \longrightarrow d_1}$	(3.3.10)
$\frac{\rho \vdash M \longrightarrow (d_1 . d_2)}{\rho \vdash (\text{cdr } M) \longrightarrow d_2}$	(3.3.11)
$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N \quad d_M \neq d_N}{\rho \vdash (\text{eq? } M \ N) \longrightarrow \text{nil}}$	(3.3.12)
$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N \quad d_M = d_N}{\rho \vdash (\text{eq? } M \ N) \longrightarrow \#t}$	(3.3.13)
$\frac{\rho \vdash M \longrightarrow d_M \quad d_M \in \mathbb{A}}{\rho \vdash (\text{atom? } M) \longrightarrow \#t}$	(3.3.14)
$\frac{\rho \vdash M \longrightarrow (d_1 . d_2)}{\rho \vdash (\text{atom? } M) \longrightarrow \text{nil}}$	(3.3.15)

Figure 3.3: Semantics for \mathbb{L}_1 -expressions.

$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\text{lift } M) \longrightarrow (\text{quote } d_M)}$	(3.3.16)
$\frac{}{\rho \vdash (\text{quote } d) \longrightarrow (\text{quote } d)}$	(3.3.17)
$\frac{v_{\text{new}} \in \mathbb{V} \setminus (\text{vars } \rho \cup \text{vars } M) \quad ((v.v_{\text{new}}).\rho) \vdash M \longrightarrow d_M}{\rho \vdash (\text{lam } v M) \longrightarrow (\text{lam } v_{\text{new}} d_M)}$	(3.3.18)
$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N}{\rho \vdash (@ M N) \longrightarrow (@ d_M d_N)}$	(3.3.19)
$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\text{fix } M) \longrightarrow (\text{fix } d_M)}$	(3.3.20)
$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N \quad \rho \vdash P \longrightarrow d_P}{\rho \vdash (\text{if } M N P) \longrightarrow (\text{if } d_M d_N d_P)}$	(3.3.21)
$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N}{\rho \vdash (\text{cons } M N) \longrightarrow (\text{cons } d_M d_N)}$	(3.3.22)
$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\text{car } M) \longrightarrow (\text{car } d_M)}$	(3.3.23)
$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\text{cdr } M) \longrightarrow (\text{cdr } d_M)}$	(3.3.24)
$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N}{\rho \vdash (\text{eq? } M N) \longrightarrow (\text{eq? } d_M d_N)}$	(3.3.25)
$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\text{atom? } M) \longrightarrow (\text{atom? } d_M)}$	(3.3.26)
$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\text{error } M) \longrightarrow (\text{error } d_M)}$	(3.3.27)

Figure 3.4: (Part of the) Semantics for \mathbb{L}_2 -expressions.

3.4 Evaluation

In both semantics, the result of evaluating a \mathbb{L}_1 - or \mathbb{L}_2 -expression M is some S-expression d_M , with

$$\text{nil} \vdash M \longrightarrow d_M$$

In the case of \mathbb{L}_2 -expressions, there may be several such d_M satisfying this, because of the residual abstraction rule. In both semantics, if no such d_M exists, we say that the result of evaluating M is undefined. There can be several reasons for this, among them nontermination (eg. infinite loop) and type errors, like trying to apply a quoted constant as a function.

Note that, unlike the original lambda-mix, \mathbb{L}_1 - and \mathbb{L}_2 -expressions do not take their arguments through their free variables, but rather take them through abstractions. That is, instead of evaluating

$$[x_1 \mapsto d_1, x_2 \mapsto d_2] \vdash M \longrightarrow d,$$

we write

$$\text{nil} \vdash (@ (@ (\text{lam } x_1 (\text{lam } x_2 M)) d_1) d_2) \vdash M \longrightarrow d.$$

The reason for this is primarily aesthetics—our initial environment is completely empty, and all available information is found directly in the annotated expression.

3.5 Optimality

We can now formally state the overall goal of this thesis: optimality of lambda-mix. We recall that, to prove optimality of a partial evaluator, we need to exhibit a self-interpreter that, when specialized wrt. to any program P , returns another program that is no less efficient than P , by some measure, eg. a timed semantics. In particular, the specialization of the self-interpreter must terminate for all valid programs.

In the case of lambda-mix, we do *not* present a timed semantics of \mathbb{L}_1 -expressions, but will prove that the result of specializing the self-interpreter found in appendix A.1 on page 39 wrt. a \mathbb{L}_1 -expression always returns a \mathbb{L}_1 -expression α -equivalent to the original. We, as many before us, claim that any reasonable timing of evaluating lambda expressions cannot time α -equivalent expressions differently.

The first step of specializing the given self-interpreter is to annotate it, to get a \mathbb{L}_2 -expression. This has been done, and the result can be found in appendix A.2. Using $\mathit{shint}_{\text{ann}}$ as a shorthand for the annotated self-interpreter, we now state the two main theorems of this thesis:

Theorem 3.5.1 (Termination) *For all \mathbb{L}_1 -expressions M , a \mathbb{L}_1 -expression d_M exists, such that*

$$\text{nil} \vdash (@ (@ \mathit{shint}_{\text{ann}} (\text{quote } M)) (\text{lam } x \ x)) \longrightarrow d_M.$$

Theorem 3.5.2 (Uniqueness) *For all \mathbb{L}_1 -expressions M*

$$\text{nil} \vdash (@ (@ \mathit{shint}_{\text{ann}} (\text{quote } M)) (\text{lam } x \ x)) \longrightarrow d_M \implies M =_{\alpha} d_M.$$

Note that the Termination theorem says that *specialization* always terminates. Obviously, evaluation of the residual program may not terminate.

It is fairly intuitive that these theorems hold: As to termination, the self-interpreter is *compositional*, which implies totality, see (Jones [13]). Regarding uniqueness, by inspection of the annotated self-interpreter $\mathit{shint}_{\text{ann}}$ it is seen that all language constructs are simply copied, except for abstractions. By further inspection of rule 3.3.18 on the preceding page, however, it follows that the code handling abstractions simply reduces to variable renaming (this takes a little work with pen and paper to see, but is simple enough).

In summary, we hope to have convinced the reader that the system defined in this chapter is indeed (a version of) Gomard's lambda-mix, and therefore \mathbb{L}_2 defines a partial evaluator for the language \mathbb{L}_1 . We believe that Gomard's proof of the correctness of lambda-mix could be modified to work for our system. We have not, and neither has Gomard, proved that shint is indeed a correct self-interpreter. The pedant may very well say optimality of lambda-mix is *not* proven till we know that it is indeed a self-interpreter we specialize. After all, the specialization of the \mathbb{L}_1 -expression

$$(\text{lam } \text{expr} \ (\text{lam } \text{env} \ \text{expr}))$$

also terminates with a term α -equivalent to M when applied to $(\text{quote } M)$ and $(\text{lam } x \ x)$, but can hardly be viewed a self-interpreter.

We hope, then, that the reader will bear with us, when we claim that proof of the two theorems above is in fact proof of the optimality of a partial evaluator.

Chapter 4

Paper Proof

Having defined lambda-mix in the previous chapter, we now turn to the overall goal of the thesis: to prove lambda-mix optimal wrt. the self-interpreter *sint* given in appendix A.1. The proof given in this chapter is an ordinary paper proof, while a mechanized version of this proof is presented in chapter 5.

4.1 Preliminaries

First a note on the derivations found in this chapter: The only non-deterministic inference rule of the semantics given in figures 3.3 and 3.4, is the residual abstraction rule, rule 3.3.18. In particular, all derivations *not* using this rule will be deterministic. This fact will be used implicitly in many proofs and arguments of statements of the kind

$$\rho \vdash M \longrightarrow d \iff \rho' \vdash M' \longrightarrow d'$$

where we will write “proof by derivation” to mean “ \Rightarrow follows by derivation, and \Leftarrow by the uniqueness of said derivation” (or vice versa).

Throughout the paper, we denote by *sint_{ann}* the actual code for the annotated self-interpreter, as given in appendix A.2 on page 40. For easier reference, we further denote by *body_{ann}* the body of *sint_{ann}*, ie.

$$\textit{sint}_{\text{ann}} = (\text{fix } (\text{lam eval } (\text{lam expr } (\text{lam env } \textit{body}_{\text{ann}}))))).$$

Whenever *M* is an \mathbb{L}_2 -expression, we will use the following abbreviations for some common \mathbb{L}_2 -expressions:

$$\begin{aligned} (\textit{cadr } M) &= (\text{car } (\text{cdr } M)) \\ (\textit{caddr } M) &= (\text{car } (\text{cdr } (\text{cdr } M))) \\ (\textit{cadddr } M) &= (\text{car } (\text{cdr } (\text{cdr } (\text{cdr } M)))) \\ \textit{env}_\lambda &= (\text{lam var } (\text{if } (\text{eq? } \text{var } (\textit{cadr } \text{expr})) \text{value } (@ \text{env } \text{var}))) \end{aligned}$$

4.2 Lambda-mix Environments

As can be imagined from the semantics given in figures 3.3 and 3.4, the environments constructed during the course of an evaluation may get very unreadable indeed. In the general case, nothing can be said, a priori, of an environment found in an arbitrary derivation

$$\rho \vdash M \longrightarrow d.$$

However, in the case of evaluating the annotated self-interpreter the body will always be evaluated in an environment of a very specific form. We will call such environments *lambda-mix environments* and will introduce them, intuitively and formally, in the present section. Readers not interested in intuitive “chit-chat” are encouraged to skip this section and refer to definition 4.2.1 and lemma 4.2.2 below, when needed.

The derivations in question are the subderivations of

$$\text{nil} \vdash (@ (@ \text{ sint}_{\text{ann}} (\text{quote } M)) (\text{lam } x \ x)) \longrightarrow d_M \quad (4.2.1)$$

on the form $\rho \vdash \text{body}_{\text{ann}} \longrightarrow d$. As a first insight, we can by a direct derivation show that judgement (4.2.1) is derivable if and only if we can derive

$$((\text{env} . (\text{clos } x \ x \ \text{nil})) (\text{expr} . M) (\text{eval} . (\text{delay } \text{ sint}_{\text{ann}} \ \text{nil}))) \vdash \text{body}_{\text{ann}} \longrightarrow d_M.$$

The environment found above is our first example of a lambda-mix environment, and we shall denote it by $\langle M, [] \rangle$. This should intuitively be understood as the environment that, when used to evaluate body_{ann} , evaluates the \mathbb{L}_1 -expression M using no variable renamings (hence the “[]”).

As a second insight, we examine the derivation of

$$\langle (\text{lam } v \ M), [] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d \quad (4.2.2)$$

as this will allow us to identify the second (and last) form of lambda-mix environments. Assuming the existence of derivation 4.2.2 above, we deduce that a subderivation of the form

$$\begin{aligned} & ((\text{env} . (\text{clos } \text{var } (\text{if } (\text{eq? } \text{var } (\text{cadr } \text{expr})) \ \text{value } (@ \ \text{env } \ \text{var}))) \\ & ((\text{value} . v') . \langle (\text{lam } v \ M), [] \rangle)) (\text{expr} . M) (\text{eval} . (\text{delay } \text{ sint}_{\text{ann}} \ \text{nil}))) \\ & \vdash \text{body}_{\text{ann}} \longrightarrow d \end{aligned} \quad (4.2.3)$$

must also exist, where v' is a variable chosen by rule 3.3.18, ie.

$$v' \notin \text{vars } \langle (\text{lam } v \ M), [] \rangle \cup \text{vars } (@ (@ \ \text{eval } (\text{caddr } \ \text{expr})) \ \text{env}_\lambda) \quad (4.2.4)$$

We will denote the environment $((\text{env} . (\text{clos } \dots)) \dots (\text{delay } \text{ sint}_{\text{ann}} \ \text{nil})))$ from derivation 4.2.3 by $\langle M, [v := v' | M] \rangle$. (We will get back to the M in $[v := v' | M]$ in a moment.)

As to the intuitive understanding of lambda-mix environments, when considering derivations of the form

$$\langle M, [v_1 := v'_1 | M_1] \dots [v_n := v'_n | M_n] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d$$

one should think of this as expressing the evaluation of M with variable renamings $[v_1 := v'_1]$, $[v_2 := v'_2]$, \dots applied *in that order*, eg.

$$\langle x, [x := x' | M_x][x := x'' | M_y] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow x'$$

but not

$$\langle x, [x := x' | M_x][x := x'' | M_y] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow x''.$$

It remains to say a bit about the M_i s above. These are the bodies of the abstractions that triggered the extension of the lambda-mix environment, and as such, they contain information on variables used in the expressions, including free variables, that otherwise wouldn't be found in the environment. They are needed for technical reasons only, and are not important for the intuitive understanding of lambda-mix environments. As an example of how these expressions appear, consider the evaluation of the **K**-combinator, $(\text{lam } x \ (\text{lam } y \ x))$ ¹:

$$\begin{aligned} & \langle (\text{lam } x \ (\text{lam } y \ x)), [] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow (\text{lam } x_{\text{new}} \ (\text{lam } y_{\text{new}} \ x_{\text{new}})) \\ \iff & \langle (\text{lam } y \ x), [x := x_{\text{new}} | (\text{lam } y \ x)] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow (\text{lam } y_{\text{new}} \ x_{\text{new}}) \\ \iff & \langle x, [y := y_{\text{new}} | x][x := x_{\text{new}} | (\text{lam } y \ x)] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow x_{\text{new}} \end{aligned}$$

¹We assume y is available as a variable.

With hopefully an intuitive understanding of lambda-mix environments, we now turn to their formal definition:

Definition 4.2.1 (Lambda-mix Environments) *The set \mathbb{E} of lambda-mix environments is the least set closed under the following two rules:*

i. Whenever M is a \mathbb{L}_1 -expression, then $\langle M, [] \rangle \in \mathbb{E}$, where $\langle M, [] \rangle$ is the S-expression

$$((\text{env}.\text{clos } x \ x \ \text{nil})) (\text{expr}.M) (\text{eval}.\text{delay } \text{shint}_{\text{ann}} \ \text{nil}))$$

ii. If

(a) $\langle (\text{lam } v \ M_v), vl \rangle \in \mathbb{E}$,

(b) M is a \mathbb{L}_1 -expression with $\text{vars } M \subseteq \text{vars } M_v$, and

(c) v' is a variable with $v' \notin \text{vars } \langle (\text{lam } v \ M_v), vl \rangle$

then $\langle M, [v := v'|M_v]vl \rangle \in \mathbb{E}$, where $\langle M, [v := v'|M_v]vl \rangle$ is The S-expression

$$\begin{aligned} & ((\text{env}.\text{clos } \text{var} \\ & \quad (\text{if } (\text{eq? } \text{var } (\text{cadr } \text{expr})) \text{value } (@ \ \text{env } \text{var})) \\ & \quad ((\text{value}.v').\langle (\text{lam } v \ M_v), vl \rangle)) \\ & \quad (\text{expr}.M) \\ & \quad (\text{eval}.\text{delay } \text{shint}_{\text{ann}} \ \text{nil}))) \end{aligned}$$

We now state formally the fact we conveyed on the page before:

Lemma 4.2.2 *For all \mathbb{L}_1 -expressions M and S-expressions d we have*

$$\text{nil} \vdash (@ \ (@ \ \text{shint}_{\text{ann}} \ (\text{quote } M)) \ (\text{lam } x \ x)) \longrightarrow d_M$$

if and only if

$$\langle M, [] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M.$$

Proof. By derivation. We construct the unique derivation of the form

$$\frac{\nabla \quad \nabla' \quad \langle M, [] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M}{\text{nil} \vdash (@ \ (@ \ \text{shint}_{\text{ann}} \ (\text{quote } M)) \ (\text{lam } x \ x)) \longrightarrow d_M}$$

where ∇ and ∇' are subderivations. □

In the following pages, it will often be the case that we need to prove that a certain S-expression $\langle M, [v_1 := v'_1|M_1] \cdots [v_n := v'_n|M_n] \rangle$ is in fact a lambda-mix environment. When proving the case $n > 0$, it will be advantageous to think of the definition above as stating that

$$\text{vars } M \subseteq \text{vars } M_1 \subseteq \cdots \subseteq \text{vars } M_n$$

and for all $i > 0$:

$$\begin{aligned} v_i & \in \text{vars } M_{i-1} \\ v'_i & \notin \{v_1, \dots, v_i, v'_1, \dots, v'_{i-1}\} \end{aligned}$$

Note that in the case $i = 1$, the last two conditions reduce to $v'_1 \neq v_1$.

4.3 Evaluation Lemmata

With the notation for lambda-mix environments in place, we now state and prove the Evaluation lemmata. These lemmata deal with the inner workings of the annotated self-interpreter, eg. relating recursive calls of the body.

Lemma 4.3.1 (First Variable Evaluation) For all $\langle v, [] \rangle \in \mathbb{E}$ we have

$$\langle v, [] \rangle \vdash body_{\text{ann}} \longrightarrow d' \iff d' = v.$$

Proof. By derivation. □

Lemma 4.3.2 (Second Variable Evaluation) For all $\langle v, [v := v' | M] vl \rangle \in \mathbb{E}$ we have

$$\langle v, [v := v' | M] vl \rangle \vdash body_{\text{ann}} \longrightarrow d' \iff d' = v'.$$

Proof. By derivation. □

Lemma 4.3.3 (Third Variable Evaluation) For all $\langle x, [v := v' | M] vl \rangle \in \mathbb{E}$ with $x \neq v$ we have

$$\langle x, [v := v' | M] vl \rangle \vdash body_{\text{ann}} \longrightarrow d'$$

if and only if

$$\langle x, vl \rangle \vdash body_{\text{ann}} \longrightarrow d'.$$

Proof. First, by derivation, we reduce the problem to proving that for all $\langle x, [v := v' | M] vl \rangle \in \mathbb{E}$ with $x \neq v$, we have

$$\langle x, [v := v' | M] vl \rangle \vdash (\textcircled{\text{env}} \text{ expr}) \longrightarrow d' \iff \langle x, vl \rangle \vdash (\textcircled{\text{env}} \text{ expr}) \longrightarrow d'.$$

By a further derivation, we can deduce

$$\langle x, [v := v' | M] vl \rangle \vdash (\textcircled{\text{env}} \text{ expr}) \longrightarrow d'$$

if and only if

$$((\text{var } x) . ((\text{value } v') . \langle (\text{lam } v \ M), vl \rangle)) \vdash (\textcircled{\text{env}} \text{ var}) \longrightarrow d'.$$

But now we are finished, since

$$\begin{aligned} lookup(\langle x, vl \rangle, \text{env}) &= lookup(((\text{var } x) . ((\text{value } v') . \langle (\text{lam } v \ M), vl \rangle)), \text{env}) \\ lookup(\langle x, vl \rangle, \text{expr}) &= lookup(((\text{var } x) . ((\text{value } v') . \langle (\text{lam } v \ M), vl \rangle)), \text{var}) \end{aligned}$$

and thus the premises for the inference rule for application are identical in the two cases, consequently the conclusions must also be identical, as wanted. □

Lemma 4.3.4 (Quote Evaluation) For all $\langle (\text{quote } d), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{quote } d), vl \rangle \vdash body_{\text{ann}} \longrightarrow d' \iff d' = (\text{quote } d).$$

Proof. By derivation. □

Lemma 4.3.5 (Abstraction Evaluation) For all $\langle (\text{lam } v \ M), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{lam } v \ M), vl \rangle \vdash body_{\text{ann}} \longrightarrow d$$

if and only if there exists a variable v' and an S-expression d_M such that

- i. $d = (\text{lam } v' \ d_M)$,
- ii. $\langle M, [v := v'|M]vl \rangle \in \mathbb{E}$, and
- iii. $\langle M, [v := v'|M]vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$.

Proof. By derivation, except for case ii when proving the “if” direction: In order to establish $\langle M, [v := v'|M]vl \rangle \in \mathbb{E}$, we need to prove the third case in the definition of lambda-mix environments, ie. $v' \notin \langle (\text{lam } v \ M), vl \rangle$. From the generalization of rule 4.2.4 on page 15, ie.

$$v' \notin \text{vars } \langle (\text{lam } v \ M), vl \rangle \cup \text{vars } (\text{@ } (\text{@ eval } (\text{caddr expr}) \ \text{env}_\lambda))$$

it follows immediately. □

Lemma 4.3.6 (Application Evaluation) For all $\langle (\text{@ } M \ N), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{@ } M \ N), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exist S-expressions d_M and d_N such that

- i. $d' = (\text{@ } d_M \ d_N)$,
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$, and
- iii. $\langle N, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_N$.

Proof. By derivation. □

Lemma 4.3.7 (Fixpoint Evaluation) For all $\langle (\text{fix } M), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{fix } M), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exists an S-expression d_M such that

- i. $d' = (\text{fix } d_M)$ and
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$.

Proof. By derivation. □

Lemma 4.3.8 (Conditional Evaluation) For all $\langle (\text{if } M \ N \ P), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{if } M \ N \ P), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exist S-expressions d_M , d_N , and d_P such that

- i. $d' = (\text{if } d_M \ d_N \ d_P)$,
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$,
- iii. $\langle N, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_N$, and
- iv. $\langle P, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_P$.

Proof. By derivation. □

Lemma 4.3.9 (Cons Evaluation) For all $\langle (\text{cons } M \ N), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{cons } M \ N), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exist S-expressions d_M and d_N such that

- i. $d' = (\text{cons } d_M \ d_N)$,
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$, and
- iii. $\langle N, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_N$.

Proof. By derivation. □

Lemma 4.3.10 (Car Evaluation) For all $\langle (\text{car } M), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{car } M), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exists an S-expression d_M such that

- i. $d' = (\text{car } d_M)$ and
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$.

Proof. By derivation. □

Lemma 4.3.11 (Cdr Evaluation) For all $\langle (\text{cdr } M), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{cdr } M), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exists an S-expression d_M such that

- i. $d' = (\text{cdr } d_M)$ and
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$.

Proof. By derivation. □

Lemma 4.3.12 (Equality Test Evaluation) For all $\langle (\text{eq? } M \ N), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{eq? } M \ N), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exist S-expressions d_M and d_N such that

- i. $d' = (\text{eq? } d_M \ d_N)$,
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$, and
- iii. $\langle N, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_N$.

Proof. By derivation. □

Lemma 4.3.13 (Atom Test Evaluation) For all $\langle (\text{atom? } M), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{atom? } M), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exists an S-expression d_M such that

- i. $d' = (\text{atom? } d_M)$ and
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$.

Proof. By derivation. □

Lemma 4.3.14 (Error Evaluation) For all $\langle (\text{error } M), vl \rangle \in \mathbb{E}$ we have

$$\langle (\text{error } M), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

if and only if there exists an S-expression d_M such that

- i. $d' = (\text{error } d_M)$ and
- ii. $\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M$.

Proof. By derivation. □

4.4 Proof of Optimality

We now arrive at the actual proofs of the Termination and Uniqueness theorems. While the Termination theorem, as we shall see shortly, is fairly easy to prove, the Uniqueness theorem demands a bit more by way of technical lemmata.

4.4.1 Termination

That $\text{shint}_{\text{ann}}$ is compositional is expressed in the Evaluation lemmata, and it should therefore come as no surprise that these suffice to prove the Termination theorem. We now state and prove a closely related lemma from which the Termination theorem follows easily:

Lemma 4.4.1 For all $\langle M, vl \rangle \in \mathbb{E}$ an \mathbb{L}_1 -expression d_M exists, such that

$$\langle M, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M.$$

Proof. By induction over the structure of M :

- i. *Case* $M \equiv v$. This case is proven by induction on the length of vl . The first Variable Evaluation lemma gives us the induction start, while the induction step is proven by splitting on $v = v_n$, and using the second and third Variable Evaluation lemmata.
- ii. *Case* $M \equiv (\text{quote } d)$. Follows directly from the Quote Evaluation lemma.
- iii. *Case* $M \equiv (\text{lam } v P)$. Choose a variable v' with $v' \notin \text{vars } \langle (\text{lam } v P), vl \rangle$. (This can be done since the number of variables in $\langle (\text{lam } v P), vl \rangle$ is finite, as it is for any S-expression, while the set of variables is infinite.) By the induction hypothesis, a \mathbb{L}_1 -expression d_P exists, such that

$$\langle P, [v := v'|P]vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_P.$$

From definition 4.2.1 we have $\langle P, [v := v'|P]vl \rangle$, and now the Abstraction Evaluation lemma gives us

$$\langle (\text{lam } v P), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow (\text{lam } v' d_P)$$

as wanted.

- iv. Cases $M \equiv (\text{@ } P \ Q)$, $M \equiv (\text{fix } P)$, $M \equiv (\text{if } P \ Q \ R)$, $M \equiv (\text{cons } P \ Q)$, $M \equiv (\text{car } P)$, $M \equiv (\text{cdr } P)$, $M \equiv (\text{eq? } P \ Q)$, $M \equiv (\text{atom? } P)$, and $M \equiv (\text{error } P)$. These cases are all similar, and we will only prove the case $M \equiv (\text{@ } P \ Q)$.

By the induction hypothesis \mathbb{L}_1 -expressions d_P and d_Q exist, such that

$$\langle P, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_P$$

and

$$\langle Q, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_Q.$$

By the Application Evaluation lemma, we have

$$\langle (\text{@ } P \ Q), vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow (\text{@ } d_P \ d_Q)$$

as wanted.

This concludes the proof. \square

Proof (Termination Theorem). By the above lemma, and lemma 4.2.2 on page 16. \square

4.4.2 Uniqueness

As we will be working intensively with the definition of lambda-mix environments in this subsection, we will start with the following two small lemmata, that will prove very useful in the proofs to come.

Lemma 4.4.2 *If $\langle M, vl \rangle \in \mathbb{E}$ and $M' \in \mathbb{L}_1$ with $\text{vars } M' \subseteq \text{vars } M$ then $\langle M', vl \rangle \in \mathbb{E}$.*

Proof. If $vl = []$, $\langle M', [] \rangle \in \mathbb{E}$ is trivial. Assume $vl = [v := v'|M_v]vl'$. We now have

- i. $\langle (\text{lam } v \ M_v), vl' \rangle \in \mathbb{E}$,
- ii. $\text{vars } M \subseteq \text{vars } M_v$, and
- iii. $v' \notin \text{vars } \langle (\text{lam } v \ M_v), vl' \rangle$.

From this $\langle M', [v := v'|M_v]vl' \rangle \in \mathbb{E}$ follows since $\text{vars } M' \subseteq \text{vars } M \subseteq \text{vars } M_v$. \square

Lemma 4.4.3 *If $\langle M, vl_1[v := v'|M_v]vl_2 \rangle \in \mathbb{E}$ then $\langle M, vl_1 vl_2 \rangle \in \mathbb{E}$.*

Proof. By induction on the length of vl_1 .

- i. *Case $vl_1 = []$.* If also $vl_2 = []$ then by definition $\langle M, [] \rangle \in \mathbb{E}$. Now assume $vl_2 = [y := y'|M_y]vl'_2$. Thus, by definition 4.2.1, we now have $\langle (\text{lam } v \ M_v), [y := y'|M_y]vl'_2 \rangle \in \mathbb{E}$, and hence $\langle (\text{lam } y \ M_y), vl'_2 \rangle \in \mathbb{E}$ and $y' \notin \text{vars } \langle (\text{lam } y \ M_y), vl'_2 \rangle$. Finally, $\text{vars } M \subseteq \text{vars } M_v \subseteq \text{vars } M_y$ is obvious.
- ii. *Case $vl_1 = [y := y'|M_y]vl'_1$.* Assume $\langle M, [y := y'|M_y]vl'_1[v := v'|M_v]vl_2 \rangle$. To prove $\langle M, [y := y'|M_y]vl'_1 vl_2 \rangle$, we must prove
 - (a) $\langle (\text{lam } y \ M_y), vl'_1 vl_2 \rangle \in \mathbb{E}$,
 - (b) $\text{vars } M \subseteq \text{vars } M_y$, and
 - (c) $y' \notin \text{vars } \langle (\text{lam } y \ M_y), vl'_1 vl_2 \rangle$.

From the assumption and the definition of lambda-mix environments, we get $\text{vars } M \subseteq \text{vars } M_y$, which takes care of case iib. Further, we get $\langle (\text{lam } y \ M_y), vl'_1[v := v'|M_v]vl_2 \rangle \in \mathbb{E}$. Using the induction hypothesis, this gives us case iia. Finally, we have

$$y' \notin \langle (\text{lam } y \ M_y), vl'_1[v := v'|M_v]vl_2 \rangle$$

which together with the observation

$$\text{vars } \langle (\text{lam } y \ M_y), vl'_1 vl_2 \rangle \subseteq \text{vars } \langle (\text{lam } y \ M_y), vl'_1[v := v'|M_v]vl_2 \rangle$$

gives us case iic.

This completes the proof. \square

A pivot for the proof of the Uniqueness theorem is the validity of the intuitive understanding of lambda-mix environments, as discussed on page 15. We there suggested that the vl part of the lambda-mix environment $\langle M, vl \rangle$ should be thought of as a series of renamings. A first attempt at formalizing this would be to say that for all lambda-mix environments $\langle M, [v := v'|M_v]vl \rangle$, we have $\langle M, [v := v'|M_v]vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d$ implies $\langle M[v := v'], vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d$. Unfortunately, from the fact that $\langle M, [v := v'|M_v]vl \rangle$ is a lambda-mix environment, we cannot deduce that $\langle M[v := v'], vl \rangle$ is one too. To see this, take vl to be $[y := y'|M_y]vl'$. Now, for $\langle M[v := v'], [y := y'|M_y]vl' \rangle$ to be a lambda-mix environment, we must have $\text{vars } M[v := v'] \subseteq \text{vars } M_y$. When $v' \in \text{vars } M[v := v']$, this cannot hold since v' was specifically chosen such that $v' \notin \text{vars } \langle (\text{lam } v \ M_v), vl \rangle$, and so in particular $v' \notin \text{vars } M_y$.

To remedy the problem, we introduce the following operator:

Definition 4.4.4 (Extension) Whenever M is an \mathbb{L}_1 -expression and v is a variable, $M \uplus v$ denotes some \mathbb{L}_1 -expression M' with $\text{vars } (M \uplus v) = \text{vars } M \cup \{v\}$. By overloading, we will write $\langle M, [v_1 := v'_1|M_1] \cdots [v_n := v'_n|M_n] \uplus v \rangle$ for the lambda-mix environment

$$\langle M, [v_1 := v'_1|M_1 \uplus v] \cdots [v_n := v'_n|M_n \uplus v] \rangle.$$

The actual \mathbb{L}_1 -expressions denoted by $M \uplus v$ will not be relevant, though to be concrete one could use, eg. $(\text{let } M \ v)$.

We can now prove the following lemma:

Lemma 4.4.5 If $\langle M, [v := v'|M_v]vl \rangle \in \mathbb{E}$ then $\langle M[v := v'], vl \uplus v' \rangle \in \mathbb{E}$.

Proof. By induction on the length of vl . If $vl = []$, then $\langle M[v := v'], [] \uplus v' \rangle$ is by definition a lambda-mix environment, since $\langle M[v := v'], [] \uplus v' \rangle = \langle M[v := v'], [] \rangle$. Now assume $vl = [y := y'|M_y]vl'$, and $\langle M, [v := v'|M_v][y := y'|M_y]vl' \rangle \in \mathbb{E}$. Using lemma 4.4.3 on the initial assumption we get $\langle M, [v := v'|M_v]vl' \rangle \in \mathbb{E}$ and $\langle M, [y := y'|M_y]vl' \rangle \in \mathbb{E}$. We now do a case analysis of vl' :

i. Case $vl' = []$. We are to prove $\langle M[v := v'], [y := y'|M_y] \uplus v' \rangle \in \mathbb{E}$, ie.

- (a) $\langle (\text{lam } y \ M_y \uplus v'), [] \rangle \in \mathbb{E}$,
- (b) $\text{vars } M[v := v'] \subseteq \text{vars } M_y \uplus v'$, and
- (c) $y' \notin \langle (\text{lam } y \ M_y \uplus v'), [] \rangle \in \mathbb{E}$.

Case ia is immediate, case ib follows easily from $\text{vars } M \subseteq \text{vars } M_v \subseteq \text{vars } M_y$, and case ic uses $\langle M, [y := y'|M_y] \rangle \in \mathbb{E}$ and $v' \neq y'$, which follows from

$$v' \notin \text{vars } \langle (\text{lam } v \ M_v), [y := y'|M_y] \rangle.$$

ii. *Case* $vl' = [z := z' | M_z] vl''$. We must prove $\langle M[v := v'], [y := y' | M_y][z := z' | M_z] vl'' \uplus v' \rangle \in \mathbb{E}$, ie.

- (a) $\langle (\text{lam } y \ M_y \uplus v'), [z := z' | M_z \uplus v'](vl'' \uplus v') \rangle \in \mathbb{E}$,
- (b) $\text{vars } M[v := v'] \subseteq \text{vars } M_y \uplus v'$, and
- (c) $y' \notin \text{vars } \langle (\text{lam } y \ M_y \uplus v'), [z := z' | M_z \uplus v'](vl'' \uplus v') \rangle$.

Cases iib and iic are straightforward, using the same reasoning as above. Case iia is easy, once we use the induction hypothesis to get

$$\langle M[v := v'], [z := z' | M_z \uplus v'](vl'' \uplus v') \rangle \in \mathbb{E}.$$

This completes the proof. \square

The formalization of the renaming-intuition stated on page 15 is given in the following lemma, which will play a central role in proving the Uniqueness theorem.

Lemma 4.4.6 (Renaming) *For all* $\langle M, [v := v' | M_v] vl \rangle \in \mathbb{E}$,

$$\langle M, [v := v' | M_v] vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M \implies \langle M[v := v'], vl \uplus v' \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M.$$

Before giving the proof of the Renaming lemma, we will need an array of technical lemmata, allowing us to manipulate lambda-mix environments to suit our needs. Overall, the Renaming lemma will be proven by a simple structural induction over M . There are two cases, however, that demand special attention: variables and abstractions (as might be expected).

The next two lemmata given here take care of the variable case.

Lemma 4.4.7 *For all* $\langle y, [v := v' | M_v] vl \rangle \in \mathbb{E}$

$$\langle y, vl \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d' \implies \langle y, vl \uplus v' \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$$

Proof. First note that $\langle y, [v := v' | M_v] vl \rangle \in \mathbb{E}$ implies $\langle y, vl \rangle \in \mathbb{E}$ and $\langle y, vl \uplus v' \rangle \in \mathbb{E}$. The proof proceeds by induction on the length of vl . The case $vl = []$ is trivial, so assume $vl = [z := z' | M_z] vl'$. We split on $y = z$:

- i. *Case* $y = z$. In this case, the second Variable Evaluation lemma gives us $d' = y$, and using it again, we get $\langle y, [z := z' | M_z \uplus v'] vl' \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$, as wanted.
- ii. *Case* $y \neq z$. The third Variable Evaluation lemma gives us $\langle y, vl' \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$, and by the induction hypothesis we get $\langle y, vl' \uplus v' \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$. Finally, the third Variable Evaluation lemma gives us $\langle y, [z := z' | M_z \uplus v'](vl' \uplus v') \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$, which of course is the same as $\langle y, [z := z' | M_z] vl' \uplus v' \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d'$, as wanted.

This completes the proof. \square

Lemma 4.4.8 *For all* $\langle v, [v_n := v'_n | M_n] \cdots [v_1 := v'_1 | M_1] \rangle \in \mathbb{E}$, *if* $v \notin \{v_1, \dots, v_n\}$, *we have*

$$\langle v, [v_n := v'_n | M_n] \cdots [v_1 := v'_1 | M_1] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d \iff d = v.$$

Proof. By induction on n . The case $n = 0$ follows from the first Variable Evaluation lemma. For $n > 0$, since $v \neq v_n$, we can use the third Variable Evaluation lemma, and the induction hypothesis then completes the proof. \square

The next three lemmata, with corresponding corollaries, handle the abstraction case. Due to time-constraints, the proofs have been omitted. As the proofs proceed much like, eg. the proofs for lemmata 4.4.3 and 4.4.5, and they have been mechanically verified, we do not consider this a problem.

Lemma 4.4.9 For all $\langle M, vl_1[x := x'|M_x][y := y'|M_y]vl_2 \rangle \in \mathbb{E}$, if $x = y$ and

$$\langle M, vl_1[x := x'|M_x][y := y'|M_y]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M$$

then

$$\langle M, vl_1[y := y'|M_y][x := x'|M_y \uplus y]vl_2 \rangle \in \mathbb{E}$$

and

$$\langle M, vl_1[y := y'|M_y][x := x'|M_y \uplus y]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Proof. Omitted. □

Corollary 4.4.10 (Swapping) For all $\langle M, [x := x'|M_x][y := y'|M_y]vl_2 \rangle \in \mathbb{E}$, if $x = y$ and

$$\langle M, [x := x'|M_x][y := y'|M_y]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M$$

then

$$\langle M, [y := y'|M_y][x := x'|M_y \uplus y]vl_2 \rangle \in \mathbb{E}$$

and

$$\langle M, [y := y'|M_y][x := x'|M_y \uplus y]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Lemma 4.4.11 For all $\langle M, vl_1[x := x'|M_x][y := y'|M_y]vl_2 \rangle \in \mathbb{E}$, if $x \neq y$ and

$$\langle M, vl_1[x := x'|M_x][y := y'|M_y]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M$$

then

$$\langle M, vl_1[x := x'|M_x]vl_2 \uplus y' \rangle \in \mathbb{E}$$

and

$$\langle M, vl_1[x := x'|M_x]vl_2 \uplus y' \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Proof. Omitted. □

Corollary 4.4.12 (Deletion) For all $\langle M, [x := x'|M_x][y := y'|M_y]vl_2 \rangle \in \mathbb{E}$, if $x \neq y$ and

$$\langle M, [x := x'|M_x][y := y'|M_y]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M$$

then

$$\langle M, [x := x'|M_x]vl_2 \uplus y' \rangle \in \mathbb{E}$$

and

$$\langle M, [x := x'|M_x]vl_2 \uplus y' \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Lemma 4.4.13 For all $\langle M, vl_1[x := x'|M_x]vl_2 \rangle \in \mathbb{E}$, if

$$\langle M, vl_1[x := x'|M_x]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M$$

and $\langle M, vl_1[x := x'|M'_x]vl_2 \rangle$ is another lambda-mix environment with $vars M'_x \subseteq vars M_x$ then

$$\langle M, vl_1[x := x'|M'_x]vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Proof. Omitted. □

Corollary 4.4.14 (Weakening) For all $\langle M, [x := x' | M_x] vl_2 \rangle \in \mathbb{E}$, if

$$\langle M, [x := x' | M_x] vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M$$

then

$$\langle M, [x := x' | M] vl_2 \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

We are now ready to give the proof of the Renaming lemma:

Proof (Renaming lemma). By induction over the structure of M . In all cases, we assume

$$\langle M, [v := v' | M_v] vl \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

i. *Case* $M \equiv y$. We will prove $\langle y[v := v'], vl \rangle \vdash body_{\text{ann}} \longrightarrow d_M$, from which the wanted derivation follows by application of lemma 4.4.7 on page 23. First, split on $y = v$:

(a) *Case* $y = v$. The second Variable Evaluation lemma gives us $d_M = v'$, and since $v' \notin \text{vars } \langle (\text{lam } v \ M_v), vl \rangle$, we can use lemma 4.4.8 on page 23, which gives us

$$\langle v', vl \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Since $y[v := v'] = v'$, this is the wanted derivation.

(b) *Case* $y \neq v$. The third Variable Evaluation lemma gives us

$$\langle y, vl \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Since $y[v := v'] = y$, this is exactly what we're after.

ii. *Case* $M \equiv (\text{quote } d)$. The Quote Evaluation lemma gives us $d_M = (\text{quote } d)$. Another application of the same lemma gives us

$$\langle (\text{quote } d), vl \uplus v' \rangle \vdash body_{\text{ann}} \longrightarrow d_M.$$

Again, this is what we're after since $(\text{quote } d)[v := v'] = (\text{quote } d)$.

iii. *Case* $M \equiv (\text{lam } y \ P)$. The Abstraction Evaluation lemma gives us $d_M = (\text{lam } y' \ d_P)$ for some variable $y' \notin \text{vars } \langle M, [v := v' | M_v] vl \rangle$ and an S-expression d_P such that

$$\langle P, [y := y' | P][v := v' | M_v] vl \rangle \vdash body_{\text{ann}} \longrightarrow d_P.$$

We now split the proof on whether $v = y$.

(a) *Case* $v = y$. Using the Deletion corollary we get

$$\langle P, [y := y' | P](vl \uplus v') \rangle \in \mathbb{E}$$

and

$$\langle P, [y := y' | P](vl \uplus v') \rangle \vdash body_{\text{ann}} \longrightarrow d_P$$

and hence, by the Abstraction Evaluation lemma,

$$\langle (\text{lam } v \ P), vl \uplus v' \rangle \vdash body_{\text{ann}} \longrightarrow (\text{lam } v' \ d_P).$$

This is what we want, since $v = y$ implies $(\text{lam } v \ P)[v := v'] = (\text{lam } v \ P)$.

(b) *Case* $v \neq y$. In this case, we use the Swapping corollary to obtain

$$\langle P, [v := v' | M_v][y := y' | M_v \uplus v] vl \rangle \in \mathbb{E}$$

and

$$\langle P, [v := v' | M_v][y := y' | M_v \uplus v] vl \rangle \vdash body_{\text{ann}} \longrightarrow d_P.$$

By the induction hypothesis we now have

$$\langle P[v := v'], [y := y' | M_v \uplus v] vl \uplus v' \rangle \vdash body_{\text{ann}} \longrightarrow d_P.$$

Using the Weakening corollary and lemma 4.4.5, we get

$$\langle P[v := v'], [y := y' | P[v := v']](vl \uplus v') \rangle \in \mathbb{E}$$

and

$$\langle P[v := v'], [y := y' | P[v := v']](vl \uplus v') \rangle \vdash body_{\text{ann}} \longrightarrow d_P,$$

and the Abstraction Evaluation lemma now gives us

$$\langle (\text{lam } y \ P[v := v']), vl \uplus v' \rangle \vdash body_{\text{ann}} \longrightarrow (\text{lam } y' \ d_P).$$

By observing that $(\text{lam } y \ P)[v := v'] = (\text{lam } y \ P[v := v'])$, we are through this case.

- iv. *Cases* $M \equiv (\text{@ } P \ Q)$, $M \equiv (\text{fix } P)$, $M \equiv (\text{if } P \ Q \ R)$, $M \equiv (\text{cons } P \ Q)$, $M \equiv (\text{car } P)$, $M \equiv (\text{cdr } P)$, $M \equiv (\text{eq? } P \ Q)$, $M \equiv (\text{atom? } P)$, and $M \equiv (\text{error } P)$. These cases are all similar, and we will only prove the case $M \equiv (\text{@ } P \ Q)$.

Using the Application Evaluation lemma, we get $d_M = (\text{@ } d_P \ d_Q)$ for some \mathbb{L}_1 -expressions d_P and d_Q with

$$\langle P, [v := v' | M_v] vl \rangle \vdash body_{\text{ann}} \longrightarrow d_P$$

and

$$\langle Q, [v := v' | M_v] vl \rangle \vdash body_{\text{ann}} \longrightarrow d_Q.$$

By the induction hypothesis and lemma 4.4.2, this implies

$$\langle P[v := v'], vl \uplus v' \rangle \vdash body_{\text{ann}} \longrightarrow d_P$$

and

$$\langle Q[v := v'], vl \uplus v' \rangle \vdash body_{\text{ann}} \longrightarrow d_Q,$$

and hence, using lemma 4.4.5 and the Application Evaluation lemma once more, we get

$$\langle (\text{@ } P[v := v'] \ Q[v := v']), vl \uplus v' \rangle \vdash body_{\text{ann}} \longrightarrow (\text{@ } d_P \ d_Q).$$

Since $(\text{@ } P \ Q)[v := v'] = (\text{@ } P[v := v'] \ Q[v := v'])$, this is what we want.

This concludes the proof of the Renaming lemma. \square

Finally, we are able to prove the following lemma:

Lemma 4.4.15 *For all \mathbb{L}_1 -expressions M and d_M :*

$$\langle M, [] \rangle \vdash body_{\text{ann}} \longrightarrow d_M \implies M =_{\alpha} d_M.$$

Proof. We proceed by induction over the size of M . In all cases, we assume

$$\langle M, [] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_M.$$

- i. *Case* $M \equiv x$. By the first Variable Evaluation lemma, we have $d_M = x$, which is trivially α -equivalent to x .
- ii. *Case* $M \equiv (\text{quote } d)$. By the Quote Evaluation lemma, we have $d_M = (\text{quote } d)$, which is trivially α -equivalent to $(\text{quote } d)$.
- iii. *Case* $M \equiv (\text{lam } v \ P)$. The Abstraction Evaluation lemma gives us $d_M = (\text{lam } v' \ d_P)$ for some variable $v' \notin \text{vars } \langle P, [] \rangle$ and S-expression d_P such that

$$\langle P, [v := v'|P] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_P.$$

Using the Renaming lemma, we get

$$\langle P[v := v'], [] \uplus v' \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_P.$$

Since $\langle P[v := v'], [] \uplus v' \rangle = \langle P[v := v'], [] \rangle$, we can use the induction hypothesis to get $d_P =_\alpha P[v := v']$. This, in turn, gives us

$$(\text{lam } v \ P) =_\alpha (\text{lam } v' \ P[v := v']) =_\alpha (\text{lam } v' \ d_P),$$

as wanted.

- iv. *Cases* $M \equiv (@ \ P \ Q)$, $M \equiv (\text{fix } P)$, $M \equiv (\text{if } P \ Q \ R)$, $M \equiv (\text{cons } P \ Q)$, $M \equiv (\text{car } P)$, $M \equiv (\text{cdr } P)$, $M \equiv (\text{eq? } P \ Q)$. $M \equiv (\text{atom? } P)$, and $M \equiv (\text{error } P)$. These cases are all similar, and we will only prove the case $M \equiv (@ \ P \ Q)$.

The Application Evaluation lemma gives us $d_M = (@ \ d_P \ d_Q)$ for some \mathbb{L}_1 -expressions d_P and d_Q with

$$\langle P, [] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_P$$

and

$$\langle Q, [] \rangle \vdash \text{body}_{\text{ann}} \longrightarrow d_Q.$$

By the induction hypothesis, we have $P =_\alpha d_P$ and $Q =_\alpha d_Q$, and hence

$$(@ \ P \ Q) =_\alpha (@ \ d_P \ d_Q)$$

as wanted.

This concludes the proof. □

Proof (Uniqueness Theorem). By the above lemma, and lemma 4.2.2 on page 16. □

Chapter 5

Mechanical Proof

We now present a mechanical proof of the optimality of lambda-mix, based on the paper proof given in the preceding chapter. The full Isabelle/HOL proof scripts can be found in appendix C on page 45.

5.1 Choice of Proof System

The first step in mechanically proving something, is of course to choose a proof system. Surveying the available proof systems, we found HOL, PVS, and Isabelle to be relevant, because people with knowledge in these systems were present, or reachable, at DIKU at the time this thesis began.

Our first choice was PVS ([22, 23, 28]), but unfortunately PVS' is not very good at automatically instantiating quantifiers, as is also apparently admitted by the designers (Stringer-Calvert [29], p. 148). The problem is, basically, that given a statement of the form

$$\forall y. \forall x. P_1 \wedge \dots \wedge P_k \wedge x = K \wedge P_{k+1} \wedge \dots \wedge P_n$$

PVS is unable to automatically instantiate $\forall x$ with K , though this is an obvious, sound simplification.

To automatically evaluate programs using an operational semantics, such simplifications are paramount. We therefore abandoned PVS, and tried Isabelle ([25, 26]). Instantiations of the sort mentioned above are handled automatically by Isabelle, and we therefore stuck with Isabelle.

We have provided an ad-hoc overview of Isabelle/HOL, the Isabelle instance used in the following, in appendix B.

5.2 Lambda-mix

We now present our formalization of chapter 3 within Isabelle/HOL.

5.2.1 Syntax

Our first decision is how to represent the set of atoms. While the most æsthetic solution would be to use Isabelle's axiomatic type classes (Wenzel [31]), this is not tractable, because of the many distinct atoms we need. Using axiomatic type classes, we would need a series of axioms along the lines of

$$\text{clos} \notin \{\text{delay}, \text{nil}, \#t, \text{lam}, \dots\}$$

```

delay ∉ {clos, nil, #t, lam, ...}
nil   ∉ {clos, delay, #t, lam, ...}
      ∴

```

etc. Besides the inconvenience, the resulting system would be large and slow.

We have therefore chosen to use natural numbers as atoms

```
types atom = nat
```

and use a series of declarations of the sort

```

nil   = 0
clos  = 1
      ∴

```

etc. to ensure that the atoms found in \mathbb{S}_2 and \mathbb{V}_{sint} are all distinct. We have defined the set of variables to be the set of naturals above 34:

```

consts var :: nat set
translations "a : var" == "34 < a"

```

and have therefore chosen naturals above 34 for the atoms in \mathbb{V}_{sint} (and naturals *below* 34 for symbols). Obviously the set of variables thus defined is infinite, as required.

The S-expressions are easily defined as a datatype:

```

datatype sexpr = Atom atom
               | SPair sexpr sexpr ("[_|_]" )

```

With the SPair constructor, we add “[d_1 | d_2]” as a notational alternative to the more verbose “SPair d_1 d_2 ”. With each datatype, Isabelle overloads the map size, which returns the size of its argument, ie. depth of constructors. All applications of size will be on S-expressions in the following.

We now define the set of \mathbb{L}_1 -expressions in figure 5.1. This is done in a very straightforward way, using an inductive definition. As can be seen in the figure, we have not found the time and energy needed to define a neat notation for S-lists.

Still following the presentation of chapter 3, we now define the *vars* map:

```

consts vars :: sexpr => nat set
primrec
  vars_atom "vars (Atom a) = (if (a : var) then {a} else {})"
  vars_pair "vars [d1|d2] = (vars d1) Un (vars d2)"

```

Again, the formulation in Isabelle is very natural, and should raise no brows.

Renaming of variables is now defined in figure 5.2. As the \mathbb{L}_1 -expressions are not defined as a datatype, we cannot use a primrec function. Instead, we use a general recursive definition, at the cost of having to explicitly state the termination-properties through a measure-function.

Figure 5.3 contains the α -equivalence relation. As can be seen there, we have introduced the notation $M\text{-}\alpha\text{-}M'$ for α -equivalence.

Finally, as we do not need to reason about \mathbb{L}_2 -expressions in general, we have chosen not to formalize their definition.¹

¹We realize, now, that we ought to have proven that the annotated self-interpreter is in fact an \mathbb{L}_2 -expression. However, the formalization of the syntax of \mathbb{L}_2 -expressions given in chapter 3 is straight-forward, and the annotated self-interpreter, given in appendix A.2 on page 40, is clearly an \mathbb{L}_2 -expression by this definition.


```

const l1expr :: sexpr set

inductive l1expr intrs

l1expr_var      "a : var ==> Atom a : l1expr"

l1expr_quote    "[Quote|[d|nil]] : l1expr"

l1expr_lam      "[| a : var; M : l1expr |]
==> [Lam|[Atom a|[M|nil]]] : l1expr"

l1expr_app      "[| M : l1expr ; N : l1expr |]
==> [App|[M|[N|nil]]] : l1expr"

l1expr_fix      "[| M : l1expr |] ==> [Fix|[M|nil]] : l1expr"

l1expr_if       "[| B : l1expr ; M : l1expr ; N : l1expr |]
==> [Cond|[B|[M|[N|nil]]]] : l1expr"

l1expr_cons     "[| M : l1expr ; N : l1expr |]
==> [Cons|[M|[N|nil]]] : l1expr"

l1expr_car      "[| M : l1expr |] ==> [Car|[M|nil]] : l1expr"

l1expr_cdr      "[| M : l1expr |] ==> [Cdr|[M|nil]] : l1expr"

l1expr_iseq     "[| M : l1expr ; N : l1expr |]
==> [IsEq|[M|[N|nil]]] : l1expr"

l1expr_isatom   "[| M : l1expr |] ==> [IsAtom|[M|nil]] : l1expr"

l1expr_error    "[| M : l1expr |] ==> [Error|[M|nil]] : l1expr"

```

Figure 5.1: Syntax for \mathbb{L}_1 -expressions

```

consts rename :: "(atom*atom*sexpr) => sexpr"

recdef rename "measure (%(y,z,M). size M)"

"rename (y,z,Atom w) = (if y=w then (Atom z) else (Atom w))"

"rename (y,z,[Quote|d|nil]) = [Quote|d|nil]"

"rename (y,z,[Lam|[Atom w|[M|nil]]]) =
  (if y=w then [Lam|[Atom w|[M|nil]]]
   else [Lam|[Atom w|[rename (y,z,M)|nil]]])"

"rename (y,z,[App|[M|[N|nil]]]) =
  [App|[rename (y,z,M)|[rename (y,z,N)|nil]]]"

"rename (y,z,[Fix|[M|nil]]) = [Fix|[rename (y,z,M)|nil]]"

"rename (y,z,[Cond|[M|[N|[P|nil]]]]) =
  [Cond|[rename (y,z,M)|[rename (y,z,N)|[rename (y,z,P)|nil]]]]"

"rename (y,z,[Cons|[M|[N|nil]]]) =
  [Cons|[rename (y,z,M)|[rename (y,z,N)|nil]]]"

"rename (y,z,[Car|[M|nil]]) = [Car|[rename (y,z,M)|nil]]"

"rename (y,z,[Cdr|[M|nil]]) = [Cdr|[rename (y,z,M)|nil]]"

"rename (y,z,[IsEq|[M|[N|nil]]]) =
  [IsEq|[rename (y,z,M)|[rename (y,z,N)|nil]]]"

"rename (y,z,[IsAtom|[M|nil]]) = [IsAtom|[rename (y,z,M)|nil]]"

"rename (y,z,[Error|[M|nil]]) = [Error|[rename (y,z,M)|nil]]"

```

Figure 5.2: Renaming

```

consts  alpha  :: "(sexpr*sexpr) set"
        "@alpha" :: [sexpr,sexpr] => bool ("_ -a- _" 50)

translations "M -a- M'" == "(M,M') : alpha"

inductive alpha intrs

alpha_refl      "M : l1expr ==> M -a- M"

alpha_lam1      "[|Atom y : l1expr; M -a- M'|]
=> [Lam|Atom y|M|nil]] -a- [Lam|Atom y|M'|nil]]]"

alpha_lam2      "[|Atom z : l1expr; z ~: vars M; M' = rename (y,z,M)|]
=> [Lam|Atom y|M|nil]] -a- [Lam|Atom z|M'|nil]]]"

alpha_app       "[|M -a- M'; N -a- N'|]
=> [App|M|N|nil]] -a- [App|M'|N'|nil]]]"

alpha_fix       "[|M -a- M'|] ==> [Fix|M|nil]] -a- [Fix|M'|nil]]]"

alpha_if        "[|M -a- M'; N -a- N'; P -a- P'|]
=> [Cond|M|N|P|nil]] -a- [Cond|M'|N'|P'|nil]]]"

alpha_cons      "[|M -a- M'; N -a- N'|]
=> [Cons|M|N|nil]] -a- [Cons|M'|N'|nil]]]"

alpha_car       "[|M -a- M'|] ==> [Car|M|nil]] -a- [Car|M'|nil]]]"

alpha_cdr       "[|M -a- M'|] ==> [Cdr|M|nil]] -a- [Cdr|M'|nil]]]"

alpha_iseq      "[|M -a- M'; N -a- N'|]
=> [IsEq|M|N|nil]] -a- [IsEq|M'|N'|nil]]]"

alpha_isatom    "[|M -a- M'|]
=> [IsAtom|M|nil]] -a- [IsAtom|M'|nil]]]"

alpha_error     "[|M -a- M'|] ==> [Error|M|nil]] -a- [Error|M'|nil]]]"

alpha_sym       "M -a- M' ==> M' -a- M"

alpha_trans     "[|M -a- M'; M' -a- M''|] ==> M -a- M''"

```

Figure 5.3: α -equivalence

5.2.2 Semantics

The semantics section of chapter 3 is also easily formalized: The *lookup* map is defined by

```
consts lookup :: "(sexpr*sexpr) => sexpr"
recdef lookup "measure (%(rho,v). size rho)"
  "lookup (nil,v) = v"
  "lookup ([[y|dy]|rest],v) =
    (if (v=y) then dy else lookup (rest,v))"
```

The formalized semantics are defined in figures 5.4 and 5.5.

5.2.3 Optimality

The formalization of the annotated self-interpreter can be found on page 71. It is somewhat technically motivated, and we will have more to say about it in the next section. We now state the formalization of the two main theorems:

Goal 5.2.1 (Theorem 3.5.1, Termination)

```
M : llexpr ==> EX dM : llexpr.
  (nil |- [App|[[App|[sintann|[[Quote|[M|nil]]|nil]]]|
           [Lam|[x|[x|nil]]]|nil]]] ---> dM)
```

Goal 5.2.2 (Theorem 3.5.2, Uniqueness)

```
M : llexpr ==> (nil |- [App|[[App|[sintann|[[Quote|[M|nil]]|nil]]]|
                       [Lam|[x|[x|nil]]]|nil]]] ---> dM)
  --> (M -a- dM)
```

This concludes our formalization of chapter 3.

5.3 Proof of Optimality

We now present the mechanical proof of goals 5.2.1 and 5.2.2 of the previous section. The proof is essentially a mechanized version of the proof given in chapter 4, except for the exact definition of lambda-mix environments which is conceptually simpler in the paper proof. We will get back to this issue shortly.

First, however, we will comment on the formalization of the annotated self-interpreter of appendix A.2. As can be seen in the file `Sint.thy`, found in appendix C.11 on page 71, the actual code of the self-interpreter, bound to `sintann'`, is “hidden” by the constant function `sintann` (the same applies for the body of the self-interpreter, `bodyann'`). The reason is simply that though, by use of translations, the self-interpreter doesn’t take up much space on the screen (8 characters), internally Isabelle still operates on the entire self-interpreter. This makes Isabelle unacceptably slow when trying to automatically solve goals, and the resulting theory-dumps, ie. the saved proof-states, also grow enormously. As an example, when we introduced the constant functions `sintann` and `bodyann`, the size of our theory-dump fell from 100Mb to 30Mb, and the goals involving the self-interpreter (or the body of the self-interpreter) were solved at approx. double speed.

5.3.1 Lambda-mix Environments

As mentioned above, the definition of lambda-mix environments defined in the mechanical proof differs slightly from definition 4.2.1 on page 16. As should be clear, however, the definitions are equivalent, the definition here being an “unfolded” version of the paper version. A revision of the

```

inductive l2eval intrs

l2eval_var_nonfix "[| lookup (rho,Atom a) = d; !q. d ~= [delay|q] |]
  ==> rho |- Atom a ---> d"

l2eval_var_fix "[| lookup (rho,Atom a) = [delay|[M|[rho'|nil]]];
  rho' |- M ---> d |] ==> rho |- Atom a ---> d"

l2eval_quote "rho |- [Quote|[D|nil]] ---> D"

l2eval_lam "rho |- [Lam|[v|[M|nil]]] ---> [clos|[v|[M|[rho|nil]]]]"

l2eval_app "[| rho |- M ---> [clos|[v|[M'|[rho'|nil]]]];
  rho |- N ---> dN; [[v|dN]|rho'] |- M' ---> d |]
  ==> rho |- [App|[M|[N|nil]]] ---> d"

l2eval_fix "[| rho |- M ---> [clos|[v|[M'|[rho'|nil]]]];
  [[v|[delay|[Fix|[M|nil]]|[rho|nil]]]|rho']
  |- M' ---> d |]
  ==> rho |- [Fix|[M|nil]] ---> d"

l2eval_if_f "[| rho |- M ---> dM; dM ~= #t ; rho |- P ---> dP |]
  ==> rho |- [Cond|[M|[N|[P|nil]]]] ---> dP"

l2eval_if_t "[| rho |- M ---> #t ; rho |- N ---> dN |]
  ==> rho |- [Cond|[M|[N|[P|nil]]]] ---> dN"

l2eval_cons "[| rho |- M ---> dM; rho |- N ---> dN |]
  ==> rho |- [Cons|[M|[N|nil]]] ---> [dM|dN]"

l2eval_car "[| rho |- M ---> [d1|d2] |] ==> rho |- [Car|[M|nil]] ---> d1"

l2eval_cdr "[| rho |- M ---> [d1|d2] |] ==> rho |- [Cdr|[M|nil]] ---> d2"

l2eval_iseq_f "[| rho |- M ---> dM; rho |- N ---> dN; dM ~= dN |]
  ==> rho |- [IsEq|[M|[N|nil]]] ---> nil"

l2eval_iseq_t "[| rho |- M ---> dM; rho |- N ---> dN; dM = dN |]
  ==> rho |- [IsEq|[M|[N|nil]]] ---> #t"

l2eval_isatom_f "[| rho |- M --->dM; dM = [d1|d2] |]
  ==> rho |- [IsAtom|[M|nil]] ---> nil"

l2eval_isatom_t "[| rho |- M --->dM; dM = (Atom a) |]
  ==> rho |- [IsAtom|[M|nil]] ---> #t"

```

Figure 5.4: Formalized semantics (continued in figure 5.5)

```

l2eval_lift    "[|rho |- M ---> dM|]
               ==> rho |- [Lift|M|nil]] ---> [Quote|[dM|nil]]]"

l2eval_rquote  "rho |- [RQuote|[d|nil]] ---> [Quote|[d|nil]]]"

l2eval_rlam    "[|Atom v' : l1expr; v' ~: (vars rho) Un (vars M);
               [[Atom v|Atom v']|rho] |- M ---> dM|]
               ==> rho |- [RLam|[Atom v|[M|nil]]]
               ---> [Lam|[Atom v'|[dM|nil]]]"

l2eval_rapp    "[|rho |- M ---> dM; rho |- N ---> dN|]
               ==> rho |- [RApp|[M|[N|nil]]] ---> [App|[dM|[dN|nil]]]"

l2eval_rfix    "[|rho |- M ---> dM|]
               ==> rho |- [RFix|[M|nil]] ---> [Fix|[dM|nil]]]"

l2eval_rif     "[|rho |- M ---> dM; rho |- N ---> dN;
               rho |- P ---> dP|]
               ==> rho |- [RCond|[M|[N|[P|nil]]]]
               ---> [Cond|[dM|[dN|[dP|nil]]]]]"

l2eval_rcons   "[|rho |- M ---> dM; rho |- N ---> dN|]
               ==> rho |- [RCons|[M|[N|nil]]]
               ---> [Cons|[dM|[dN|nil]]]"

l2eval_rcar    "[|rho |- M ---> dM|]
               ==> rho |- [RCar|[M|nil]] ---> [Car|[dM|nil]]]"

l2eval_rcdr    "[|rho |- M ---> dM|]
               ==> rho |- [RCdr|[M|nil]] ---> [Cdr|[dM|nil]]]"

l2eval_riseq   "[|rho |- M ---> dM; rho |- N ---> dN|]
               ==> rho |- [RISeq|[M|[N|nil]]] ---> [IsEq|[dM|[dN|nil]]]"

l2eval_risatom "[|rho |- M ---> dM|]
               ==> rho |- [RIAtom|[M|nil]] ---> [IsAtom|[dM|nil]]]"

l2eval_rerror  "[|rho |- M ---> dM|]
               ==> rho |- [RError|[M|nil]] ---> [Error|[dM|nil]]]"

```

Figure 5.5: Formalized semantics (continued from figure 5.4)

mechanical proof to reflect the simpler definition of chapter 4 would require a major rewrite of the proof scripts, something we have not had time for.

We will first define the *syntactically* valid lambda-mix environments. After that we will, among them, select the *semantically* valid lambda-mix environments.

First, we define the syntax of the renaming lists:

```
datatype rlist = None ("[]")
              | Nemp nat nat sexpr rlist ("[_;/ _ ;/ _,/ _]")
```

The order of the parameters is slightly changed from the paper proof, so “[v' ; v ; M_v , vl]” is the formalization of the renaming list “[$v := v' | M_v$] vl ” from the paper proof.

The overall syntax of the lambda-mix environments will be the same as for the paper proof:

```
consts lwrap :: [sexpr,rlist] => sexpr ("<_,/ _>")
defs lwrap_def "lwrap M vlist == [[env|(lenv vlist)]|[[expr|M]|
                                     [[eval|[delay|[sintann|[nil|nil]]]]|nil]]]"
```

Thus, the syntactically valid lambda-mix environments are the S-expressions in the range of the map lwrap above.

We then define lenv from above:

```
primrec
  lenv_emp "lenv [] = [clos|[x|[x|[nil|nil]]]]]"
  lenv_nemp "lenv [v';v;M',vlist] =
    [clos|[var|[[Cond|[[IsEq|[var|[[Cadr|[expr|nil]]|nil]]]]|
                 [value|[[App|[env|[var|nil]]]|nil]]]]|
          [[value|Atom v']|<[Lam|[Atom v|[M'|nil]]],vlist>|nil]]]"
```

The concatenating of renaming lists, as in $\langle M, vl_1 vl_2 \rangle$, is defined by the function lconcat below. We use the infix operator # to denote concatenation.

```
consts lconcat :: [rlist,rlist] => rlist ("_ #/ _")
primrec
  lconcat_none "lconcat [] vlist = vlist"
  lconcat_nemp "lconcat [v';v;M,vlist'] vlist =
    [v';v;M,lconcat vlist' vlist]"
```

Thus, the lambda-mix environment from before is written as $\langle M, vl_1 \# vl_2 \rangle$.

We now proceed to define the semantically valid lambda-mix environments. This is done by the two inductive definitions found in figure 5.6 on the facing page.

At last we define the extension operator, which is called addvar in the mechanical proof:

```
consts addvar :: [nat,rlist] => rlist
primrec
  addvar_none "addvar y [] = []"
  addvar_nemp "addvar y [v';v;Mv,rlist] =
    [v';v;[App|[Mv|[Atom y|nil]]],addvar y rlist]"
```

5.3.2 The Proof

We would have liked to go over the mechanical proof in detail, but time prevents us. Hence, we will just say the following on the generation of new variables, and otherwise refer the reader to the actual proof scripts, found in appendix C.

In the proof of lemma 4.4.1, we argue that we can always choose a new variable-name. In the file NatInf.ML, we have adopted a scheme from Naraschewski and Nipkow [19]. We define a predicate new_var, and show that for every finite set of naturals (atoms), there is always a natural larger than 34 (variable) not in that set.

```

inductive lmixrlist intrs

lmixrlist_none "[ ] : lmixrlist"

lmixrlist_nemp "[| vlist : lmixrlist; M : l1expr; v : var; v' : var;
                v' ~: vars <[Lam|[Atom v|[M|nil]]],vlist>;
                !z' z Mz vlist'. vlist = [z';z;Mz,vlist']
                --> vars [Lam|[Atom v|[M|nil]]] <= vars Mz |]
                ==> [v';v;M,vlist] : lmixrlist"

inductive lmixenv intrs

lmixenv_init "[| M : l1expr |] ==> <M,[ ]> : lmixenv"

lmixenv_ext "[| M : l1expr ; [v';v;M',vlist] : lmixrlist;
              vars M <= vars M' |]
              ==> <M,[v';v;M',vlist]> : lmixenv"

```

Figure 5.6: The semantically valid lambda-mix environments

5.4 Summary

We have succeeded in providing a mechanical proof for the optimality of lambda-mix. While we are (largely) satisfied with the formalization of chapter 3, we regret that our proof of optimality is not very neat (and, in particular, that we have not had time to document it properly). The proof is no less valid for that, and with more time (and more experience with Isabelle), we are convinced that a mechanized proof very close to the paper proof of chapter 4 can be made.

When considering the proof, one must take into account that it was done in two months by somebody with *no* prior experience with automated proof systems (apart from a quick brush with PVS).

Chapter 6

Conclusion

We have proven that lambda-mix is optimal. We have given two related proofs: First an ordinary paper proof was given in (almost) all detail. Then that proof was used as the basis of a mechanical proof, using Isabelle/HOL. While the author had no prior experience with Isabelle, or indeed with any automated proof system at all, the complete development took around two months of full-time work.

Related Work

Others that have used automated proof systems to prove properties of partial evaluators include Welinder [30] and Hatcliff [11].

As to lambda-mix, Gomard proved the correctness of lambda-mix in [9]. His semantics, as we noted in chapter 3, were however somewhat informal, as he never defined the exact semantics of the residual abstraction. The validness of his proof has consequently been of some debate; in particular professor Eugenio Moggi, who himself has provided a correctness proof of lambda-mix using functor categories, has been very concerned with the informal nature of Gomard's semantics. While we do not agree with Moggi on the magnitude of the problem, we have made sure to be as formal as possible in our definition of lambda-mix. Using an automated proof system to validate our proof, we feel that we have done everything possible to avoid any problems of sloppiness.

Future Work

In regard to the dispute of Gomard's correctness proof, it would be interesting to adapt Gomard's proof of correctness to our lambda-mix. We believe that such adaption is possible with a modest amount of work. Also, a proof of the correctness of the self-interpreter given here would be nice, as we would then truly have proven optimality of a partial evaluator.

Appendix A

Program Listings

A.1 The Self-Interpreter

```
(fix (lam eval (lam expr (lam env

  (if (atom? expr)
      (@ env expr)

  (if (eq? (car expr) (quote quote))
      (car (cdr expr))

  (if (eq? (car expr) (quote lam))
      (lam value
        (@ (@ eval (car (cdr (cdr expr))))
          (lam var
            (if (eq? var (car (cdr expr)))
                value
                (@ env var))))))

  (if (eq? (car expr) (quote @))
      (@ (@ (@ eval (car (cdr expr))) env)
        (@ (@ eval (car (cdr (cdr expr)))) env))

  (if (eq? (car expr) (quote fix))
      (fix (@ (@ eval (car (cdr expr))) env))

  (if (eq? (car expr) (quote if))
      (if (@ (@ eval (car (cdr expr))) env)
          (@ (@ eval (car (cdr (cdr expr)))) env)
          (@ (@ eval (car (cdr (cdr (cdr expr)))))) env))

  (if (eq? (car expr) (quote cons))
      (cons (@ (@ eval (car (cdr expr))) env)
            (@ (@ eval (car (cdr (cdr expr)))) env))
```

```

(if (eq? (car expr) (quote car))
    (car (@ (@ eval (car (cdr expr))) env))

(if (eq? (car expr) (quote cdr))
    (cdr (@ (@ eval (car (cdr expr))) env))

(if (eq? (car expr) (quote eq?))
    (eq? (@ (@ eval (car (cdr expr))) env)
          (@ (@ eval (car (cdr (cdr expr)))) env))

(if (eq? (car expr) (quote atom?))
    (atom? (@ (@ eval (car (cdr expr))) env))

(if (eq? (car expr) (quote error))
    (error (@ (@ eval (car (cdr expr))) env))

(error expr)))))))))

```

A.2 The Annotated Self-Interpreter

```

(fix (lam eval (lam expr (lam env

  (if (atom? expr)
      (@ env expr)

  (if (eq? (car expr) (quote quote))
      (lift (car (cdr expr)))

  (if (eq? (car expr) (quote lam))
      (lam value
        (@ (@ eval (car (cdr (cdr expr))))
            (lam var
              (if (eq? var (car (cdr expr)))
                  value
                  (@ env var))))))

  (if (eq? (car expr) (quote @))
      (@ (@ (@ eval (car (cdr expr))) env)
          (@ (@ eval (car (cdr (cdr expr)))) env))

  (if (eq? (car expr) (quote fix))
      (fix (@ (@ eval (car (cdr expr))) env))

  (if (eq? (car expr) (quote if))
      (if (@ (@ eval (car (cdr expr))) env)

```

```
(@ (@ eval (car (cdr (cdr expr)))) env)
  (@ (@ eval (car (cdr (cdr (cdr expr)))))) env))

(if (eq? (car expr) (quote cons))
    (cons (@ (@ eval (car (cdr expr))) env)
           (@ (@ eval (car (cdr (cdr expr)))) env))

    (if (eq? (car expr) (quote car))
        (car (@ (@ eval (car (cdr expr))) env))

        (if (eq? (car expr) (quote cdr))
            (cdr (@ (@ eval (car (cdr expr))) env))

            (if (eq? (car expr) (quote eq?))
                (eq? (@ (@ eval (car (cdr expr))) env)
                       (@ (@ eval (car (cdr (cdr expr)))) env))

                (if (eq? (car expr) (quote atom?))
                    (atom? (@ (@ eval (car (cdr expr))) env))

                    (if (eq? (car expr) (quote error))
                        (error (@ (@ eval (car (cdr expr))) env))

                        (error expr))))))))))
```

Appendix B

A Quick Overview of Isabelle/HOL

We will now provide a quick overview of Isabelle, and features specific to Isabelle/HOL, its higher order logic instance. The purpose of this overview is to introduce features of Isabelle used in chapter 5, and it is by no means intended as a general introduction, which can be found in [24]. Also, an Isabelle/HOL specific tutorial can be found in [21].

Isabelle is a generic proof system, meant for defining different logics. This shows when using the system, as one has to distinguish between two levels of abstraction. At the heart of Isabelle is the meta-logic and meta-language. These are a fragment of intuitionistic higher order logic and the simply typed lambda calculus, respectively. These constitute the meta-level of Isabelle, also called *Pure Isabelle*.

The meta logic contains a universal quantifier (!!), implication (==>), and equality (==). Further, the notation

$$[| P1; P2; \dots ; Pn |] ==> P$$

abbreviates

$$P1 ==> (P2 ==> (\dots ==> (Pn ==> P) \dots))$$

New logics, called object-logics, are created by extending an existing logic, eg. the meta-logic, by *theories*.

Types

The types we shall use are found in table B.1. The type

$$['a1, 'a2, \dots, 'an] => a$$

abbreviates

$$'a1 => ('a2 => \dots ('an => a) \dots)$$

nat	a natural number
'a set	a set of elements of type 'a
'a * 'b	the product type of 'a and 'b
'a => 'b	a function with domain 'a and range 'b

Table B.1: Some types in Isabelle/HOL

Theories

A theory is a collection of definitions and declarations, defined in a single file. Theory files are split up into sections, each section containing datatype declarations, function definitions, etc. The sections we will encounter in the following proof are of the types:

`consts` Constant definitions, as in

```
consts
  x :: nat
```

which declares `x` to be a constant of type `nat`.

`defs` A (named) definition of constants, as in

```
defs
  x_isonone "x == 1"
```

which declares the constant `x` to be equal to 1. The definition can be referenced by the ML identifier `x_isonone`.

`constdefs` This combines the previous sections, as in

```
constdefs
  x :: nat "x == 1"
```

In this case, the name is fixed to `x_def`.

`datatype` Defines a datatype, and also states and proves various properties and lemmata regarding the datatype, like an induction rule, a size operator, a lemma stating the datatype injective, etc. Example (list of naturals):

```
datatype NatList
  = Empty
  | Cons nat NatList
```

`primrec` Whenever we have defined a datatype, we can define primitive recursive functions over that datatype, as in

```
primrec
  list_empty "plus Empty = 0"
  list_cons  "plus (Cons n nl) = n + (plus nl)"
```

`recdef` General recursive definitions. In the cases we use them, a decreasing measure function will be supplied, to prove totality. Apart from the measure function, the syntax is as in `primrec` sections.

`inductive` Inductive definition. The syntax is very natural, and an example can be found in figure 5.3 on page 32.

`syntax` Declares some string as syntax of a given type, rather than eg. a variable

```
syntax
  "fortytwo" :: nat ("fortytwo")
```

$P \rightarrow Q$	P implies Q
$A = B$	A is equal to B
$A \neq B$	A is not equal to B
$a : B$	a is a member of B
$a \notin B$	a is not a member of B
$A \subseteq B$	A is a subset of B
$\exists a. P$ or $\text{EX } a. P$	there exists a such that P
$\forall a. P$ or $\text{ALL } a. P$	for all a , P
$\exists a : B. P$ or $\text{EX } a : B. P$	there exists a from B such that P
$\forall a : B. P$ or $\text{ALL } a : B. P$	for all a in B , P

Table B.2: Selected notation of Isabelle/HOL

`translations` Controls parsing and printing, eg.

```
n natural == n : nat
```

In this case, writing `n natural` will be translated to `n : nat` internally, while all occurrences on the form `n : nat` will be output as `n natural`.

`types` Type synonyms, as in `naturals = nat`, allowing us to write `naturals` instead of `nat` (for whatever reason).

Notation

In the Isabelle instance we use here, Isabelle/HOL, higher order logic is both the meta-logic and object-logic. To disambiguate, `!`, `-->`, and `=` is used for object-level universal quantification, implication, and equality. This, and some more notation used in Isabelle/HOL, can be found in table B.2.

Appendix C

Isabelle Proof Scripts

Note: The proof scripts found in this appendix can also be found at

`http://www.diku.dk/students/skalberg/thesis/lambda-opt.tar.gz`

The results of chapters 3 and 4 can be found here:

Result	Where
Theorem 3.5.1	ll. 98–100 in LemAlpha.ML
Theorem 3.5.2	ll. 106–108 in LemAlpha.ML
Lemma 4.2.2	ll. 15–17 in Eval.ML
Lemma 4.3.1	ll. 75–76 in Deriv.ML
Lemma 4.3.2	ll. 85–86 in Deriv.ML
Lemma 4.3.3	ll. 101–103 in Deriv.ML
Lemma 4.3.4	ll. 142–144 in Deriv.ML
Lemma 4.3.5	ll. 153–157 in Deriv.ML
Lemma 4.3.6	ll. 219–223 in Deriv.ML
Lemma 4.3.7	ll. 324–328 in Deriv.ML
Lemma 4.3.8	ll. 449–454 in Deriv.ML
Lemma 4.3.9	ll. 254–258 in Deriv.ML
Lemma 4.3.10	ll. 349–353 in Deriv.ML
Lemma 4.3.11	ll. 374–378 in Deriv.ML
Lemma 4.3.12	ll. 289–293 in Deriv.ML
Lemma 4.3.13	ll. 399–403 in Deriv.ML
Lemma 4.3.14	ll. 424–428 in Deriv.ML
Lemma 4.4.1	l. 83 in LemAlpha.ML
Lemma 4.4.2	ll. 277–278 in EnvLem.ML
Lemma 4.4.3	ll. 61–62 in EnvLem.ML (partly)
Lemma 4.4.5	ll. 25–26 in LemRen.ML
Lemma 4.4.6	ll. 65–68 in LemRen.ML
Lemma 4.4.7	ll. 412–413 in EnvLem.ML
Lemma 4.4.8	ll. 83–84 in EnvLem.ML
Lemma 4.4.9	ll. 7–11 in LemEnv.ML
Corollary 4.4.10	ll. 7–10 in Corr.ML
Lemma 4.4.11	ll. 428–431 in EnvLem.ML
Corollary 4.4.12	ll. 18–20 in Corr.ML
Lemma 4.4.13	ll. 7–12 in LemWeak.ML
Corollary 4.4.14	—
Lemma 4.4.15	l. 89 in LemAlpha.ML

C.1 LMixEnv

```

1  (*****
2  *
3  * LMixEnv.thy
4  *
5  *****)
6
7  LMixEnv = Sint + L2Eval +
8
9  datatype rlist
10     = None ([])
11     | Nemp atom atom sexpr rlist ("[_;/ _ ;/ _;/ _]")
12
13  consts
14     lmixenv    :: sexpr set
15     lmixrlist  :: rlist set

```

```

16     varsin    :: rlist => (atom set)
17     lenv     :: rlist => sexpr
18     lwrap    :: [sexpr,rlist] => sexpr ("<_/_>")
19     lconcat  :: [rlist,rlist] => rlist ("_ #/_ _")
20
21 primrec
22     varsin_none "varsin [] = {}"
23     varsin_nemp "varsin [v';v;M,vlist] = {v',v} Un (varsin vlist)"
24
25 primrec
26     lconcat_none "lconcat [] vlist = vlist"
27     lconcat_nemp "lconcat [v';v;M,vlist'] vlist =
28         [v';v;M,lconcat vlist' vlist]"
29
30 defs
31     lwrap_def "lwrap M vlist == [[env|(lenv vlist)]|[[expr|M]|
32         [[eval|[delay|[sintann|[nil|nil]]]]|nil]]]"
33
34 primrec
35     lenv_emp "lenv [] = [clos|x|[x|[nil|nil]]]"
36     lenv_nemp "lenv [v';v;M',vlist] =
37         [clos|[var|[[Cond|[[IsEq|[var|[[Cadr|[expr|nil]]|nil]]]]|
38         [value|[[App|[env|[var|nil]]]|nil]]]]|
39         [[value|Atom v']|<[Lam|[Atom v|[M'|nil]]],vlist>|nil]]]"
40
41 inductive lmixrlist intrs
42     lmixrlist_none "[ ] : lmixrlist"
43
44     lmixrlist_nemp "[vlist : lmixrlist; M : llexpr; v : var;
45         v' : var; v' ~: vars <[Lam|[Atom v|[M|nil]]],vlist>;
46         !z' z Mz vlist'. vlist = [z';z;Mz,vlist']
47         --> vars [Lam|[Atom v|[M|nil]]] <= vars Mz|]
48         ==> [v';v;M,vlist] : lmixrlist"
49
50 inductive lmixenv intrs
51     lmixenv_init "[| M : llexpr |] ==> <M,[ ]> : lmixenv"
52     lmixenv_ext "[| M : llexpr ; [v';v;M',vlist] : lmixrlist;
53         vars M <= vars M'|]
54         ==> <M,[v';v;M',vlist]> : lmixenv"
55
56 end

1  (*****
2  *
3  * LMixEnv.ML
4  *
5  *****)
6
7  Addsimps lmixrlist.intrs;
8  AddIs lmixrlist.intrs;
9
10 Addsimps lmixenv.intrs;
11 AddIs lmixenv.intrs;
12
13 val lmrelims = map (lmixrlist.mk_cases rlist.simps)
14 [
15     "[ ] : lmixrlist",
16     "[v';v;M,vlist] : lmixrlist"
17 ];

```

```

18
19 AddSEs lmrelims;
20
21 AddSEs lmixenv.elims;
22
23 val [prem1,prem2,prem3] =
24 Goal "[|P ==> S; [|P ; S|] ==> R ; P|] ==> R";
25 br prem2 1;
26 br prem1 2;
27 br prem3 1;
28 br prem3 1;
29 qed "gen_lemma1";
30
31 val [prem1,prem2] =
32 Goal "[|P; P ==> R|] ==> R";
33 br prem2 1;
34 br prem1 1;
35 qed "gen_lemma2";
36
37 val [prem1,prem2,prem3] =
38 Goal "[|P --> S; [|P ; S|] ==> R ; P|] ==> R";
39 br prem2 1;
40 br (prem1 RS mp) 2;
41 br prem3 1;
42 br prem3 1;
43 qed "gen_lemma3";
44
45 (**
46 *
47 * LMixEnv is injective!
48 *
49 ***)
50
51 Addsimps [lwrap_def];
52
53 Goal "!vlist vlist'. (lenv vlist = lenv vlist' --> vlist = vlist')";
54 br allI 1;
55 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
56 br allI 1;
57 by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
58 force 1;
59 force 1;
60 br allI 1;
61 br impI 1;
62 by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
63 force 1;
64 by (Asm_full_simp_tac 1);
65 qed "lenv_lemma1";
66
67 Goal "(<M,vlist> = <M',vlist'>) = ((M=M') & (vlist = vlist'))";
68 auto();
69 br (lenv_lemma1 RS gen_lemma2) 1;
70 auto();
71 qed "lmixenv_inj";
72
73 Addsimps [lmixenv_inj];
74
75 (*****)

```

```

76  (*****)
77
78  Goal "!q. lenv vlist ~= [delay|q]";
79  by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
80  auto();
81  qed "lenvnotdelay";
82
83  Addsimps [lenvnotdelay];
84
85  (*****)
86  (*****)
87
88  Goal "(<Atom v,vlist> |- env ---> d) = (d = lenv vlist)";
89  auto();
90  qed "lenv_lemma2";
91
92  (*****)
93  (*****)
94
95  Goal "([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] |- \
96  \ env ---> d) = (d = lenv vlist)";
97  auto();
98  qed "lenv_lemma3";
99
100 Addsimps [lenv_lemma2,lenv_lemma3];
101
102 (*****)
103 (*****)
104
105 Addsimps [vars_atom,vars_pair];
106
107 Goal "vars (lenv (vlist # vlist')) = \
108 \ (vars (lenv vlist)) Un (vars (lenv vlist'))";
109 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
110 auto();
111 by (res_inst_tac [("rlist","vlist'")] rlist.induct 1);
112 auto();
113 qed "vars_lemma3";
114
115 Addsimps [vars_lemma3];
116
117 Goal "vars <M,vlist> <= vars <M,vlist # vlist'>";
118 auto();
119 qed "vars_lemma4";
120
121 Addsimps [vars_lemma4];
122
123 Delsimps [vars_atom,vars_pair];
124 Delsimps [lwrap_def];
125
126 Goal "[|A <= B; a ~: B|] ==> a ~: A";
127 auto();
128 qed "sub_lemma1";
129
130 Goal "(vlist # []) = vlist";
131 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
132 auto();
133 qed "lconcat_lemma2";

```

```

134
135 Goal "(vlist # [v';v;M,vlist']) ~= []";
136 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
137 auto();
138 qed "lconcat_lemma3";
139
140 Addsimps [lconcat_lemma2,lconcat_lemma3];
141
142 (**
143 Goal "(( vlist # vlist') : lmixrlist) --> \
144 \ (vlist : lmixrlist & vlist' : lmixrlist)";
145 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
146 auto();
147
148 by (eres_inst_tac [("x","z'")] allE 1);
149 by (eres_inst_tac [("x","z")] allE 1);
150 by (eres_inst_tac [("x","Mz")] allE 1);
151
152 by (res_inst_tac [("y","vlist'")] rlist.exhaust 2);
153
154 qed "lconcat_lemma1";
155 **)

```

C.2 Preliminary Definitions

```

1  (*****
2  *
3  * ROOT.ML
4  *
5  *****)
6
7  use_thy "LemRen";
8  use "LemAlpha";

1  (*****
2  *
3  * Defs.thy
4  *
5  *****)
6
7  Defs = SExpr + ExtNat +
8
9  syntax
10     "nil"           :: sexpr ("nil")
11
12     "Quote"         :: sexpr ("Quote")
13     "Lam"           :: sexpr ("Lam")
14     "App"           :: sexpr ("App")
15     "Fix"           :: sexpr ("Fix")
16     "Cond"          :: sexpr ("Cond")
17     "Cons"          :: sexpr ("Cons")
18     "Car"           :: sexpr ("Car")
19     "Cdr"           :: sexpr ("Cdr")
20     "IsEq"          :: sexpr ("IsEq")
21     "IsAtom"        :: sexpr ("IsAtom")
22     "Error"         :: sexpr ("Error")
23
24     "Lift"           :: sexpr ("Lift")

```

```

25      "RQuote"      :: sexpr ("RQuote")
26      "RLam"        :: sexpr ("RLam")
27      "RApp"         :: sexpr ("RApp")
28      "RFix"         :: sexpr ("RFix")
29      "RCond"        :: sexpr ("RCond")
30      "RCons"        :: sexpr ("RCons")
31      "RCar"         :: sexpr ("RCar")
32      "RCdr"         :: sexpr ("RCdr")
33      "RIsEq"        :: sexpr ("RIsEq")
34      "RIsAtom"      :: sexpr ("RIsAtom")
35      "RError"       :: sexpr ("RError")
36
37      "clos"         :: sexpr ("clos")
38      "delay"        :: sexpr ("delay")
39      "#t"           :: sexpr ("#t")
40
41      "eval"         :: sexpr ("eval")
42      "expr"         :: sexpr ("expr")
43      "env"          :: sexpr ("env")
44      "value"        :: sexpr ("value")
45      "var"          :: sexpr ("var")
46      "x"            :: sexpr ("x")
47
48  translations
49      "nil"           == "(Atom 0)"
50
51      "Quote"         == "(Atom 1)"
52      "Lam"           == "(Atom 2)"
53      "App"           == "(Atom 3)"
54      "Fix"           == "(Atom 4)"
55      "Cond"          == "(Atom 5)"
56      "Cons"          == "(Atom 6)"
57      "Car"           == "(Atom 7)"
58      "Cdr"           == "(Atom 8)"
59      "IsEq"          == "(Atom 9)"
60      "IsAtom"        == "(Atom 10)"
61      "Error"         == "(Atom 11)"
62
63      "Lift"          == "(Atom 12)"
64      "RQuote"        == "(Atom 13)"
65      "RLam"          == "(Atom 14)"
66      "RApp"          == "(Atom 15)"
67      "RFix"          == "(Atom 16)"
68      "RCond"         == "(Atom 17)"
69      "RCons"         == "(Atom 18)"
70      "RCar"          == "(Atom 19)"
71      "RCdr"          == "(Atom 20)"
72      "RIsEq"         == "(Atom 21)"
73      "RIsAtom"       == "(Atom 22)"
74      "RError"        == "(Atom 23)"
75
76      "delay"         == "(Atom 24)"
77      "clos"          == "(Atom 25)"
78      "#t"            == "(Atom 26)"
79
80      "eval"          == "(Atom 35)"
81      "expr"          == "(Atom 36)"
82      "env"           == "(Atom 37)"

```

```

83     "value"      == "(Atom 38)"
84     "var"        == "(Atom 39)"
85     "x"          == "(Atom 40)"
86   end

1   (*****
2   *
3   * ExtNat.thy
4   *
5   *****)
6
7   ExtNat = Nat +
8
9   syntax
10
11     "3" :: nat ( "3" )
12     "4" :: nat ( "4" )
13     "5" :: nat ( "5" )
14     "6" :: nat ( "6" )
15     "7" :: nat ( "7" )
16     "8" :: nat ( "8" )
17     "9" :: nat ( "9" )
18     "10" :: nat ( "10" )
19     "11" :: nat ( "11" )
20     "12" :: nat ( "12" )
21     "13" :: nat ( "13" )
22     "14" :: nat ( "14" )
23     "15" :: nat ( "15" )
24     "16" :: nat ( "16" )
25     "17" :: nat ( "17" )
26     "18" :: nat ( "18" )
27     "19" :: nat ( "19" )
28     "20" :: nat ( "20" )
29     "21" :: nat ( "21" )
30     "22" :: nat ( "22" )
31     "23" :: nat ( "23" )
32     "24" :: nat ( "24" )
33     "25" :: nat ( "25" )
34     "26" :: nat ( "26" )
35     "27" :: nat ( "27" )
36     "28" :: nat ( "28" )
37     "29" :: nat ( "29" )
38     "30" :: nat ( "30" )
39     "31" :: nat ( "31" )
40     "32" :: nat ( "32" )
41     "33" :: nat ( "33" )
42     "34" :: nat ( "34" )
43     "35" :: nat ( "35" )
44     "36" :: nat ( "36" )
45     "37" :: nat ( "37" )
46     "38" :: nat ( "38" )
47     "39" :: nat ( "39" )
48     "40" :: nat ( "40" )
49     "41" :: nat ( "41" )
50     "42" :: nat ( "42" )
51     "43" :: nat ( "43" )
52
53   translations
54
```

```

55     "3" == "Suc 2"
56     "4" == "Suc 3"
57     "5" == "Suc 4"
58     "6" == "Suc 5"
59     "7" == "Suc 6"
60     "8" == "Suc 7"
61     "9" == "Suc 8"
62     "10" == "Suc 9"
63     "11" == "Suc 10"
64     "12" == "Suc 11"
65     "13" == "Suc 12"
66     "14" == "Suc 13"
67     "15" == "Suc 14"
68     "16" == "Suc 15"
69     "17" == "Suc 16"
70     "18" == "Suc 17"
71     "19" == "Suc 18"
72     "20" == "Suc 19"
73     "21" == "Suc 20"
74     "22" == "Suc 21"
75     "23" == "Suc 22"
76     "24" == "Suc 23"
77     "25" == "Suc 24"
78     "26" == "Suc 25"
79     "27" == "Suc 26"
80     "28" == "Suc 27"
81     "29" == "Suc 28"
82     "30" == "Suc 29"
83     "31" == "Suc 30"
84     "32" == "Suc 31"
85     "33" == "Suc 32"
86     "34" == "Suc 33"
87     "35" == "Suc 34"
88     "36" == "Suc 35"
89     "37" == "Suc 36"
90     "38" == "Suc 37"
91     "39" == "Suc 38"
92     "40" == "Suc 39"
93     "41" == "Suc 40"
94     "42" == "Suc 41"
95     "43" == "Suc 42"
96
97 end

```

C.3 SExpr

```

1  (*****
2  *
3  * SExpr.thy
4  *
5  *****)
6
7  SExpr = Datatype + ExtNat +
8
9  types atom = nat
10
11  consts
12      var :: atom set

```



```

13
14 translations
15     "a : var" == "34 < a"
16
17 datatype sexpr
18     = Atom atom
19     | SPair sexpr sexpr ("[_|_]")
20
21 consts
22     vars :: sexpr => atom set
23
24 primrec
25     vars_atom "vars (Atom a) = (if (a : var) then {a} else {})"
26     vars_pair "vars [d1|d2] = (vars d1) Un (vars d2)"
27
28 end

1  (*****
2  *
3  * SExpr.ML
4  *
5  *****)
6
7  Delsimps [vars_atom,vars_pair];

```

C.4 Alpha

```

1  (*****
2  *
3  * Alpha.thy
4  *
5  *****)
6
7  Alpha = Rename +
8
9  consts
10     alpha :: "(sexpr*sexpr) set"
11     "@alpha" :: "[sexpr,sexpr] => bool ("_ -a- _" 50)
12
13 translations
14     "M -a- M'" == "(M,M') : alpha"
15
16 inductive alpha intrs
17
18 alpha_refl      "M : llexpr ==> M -a- M"
19
20 alpha_lam1      "[|Atom y : llexpr; M -a- M'|]
21 ==> [Lam|[Atom y|[M|nil]]] -a- [Lam|[Atom y|[M'|nil]]]"
22
23 alpha_lam2      "[|Atom z : llexpr; z ~: vars M; M' = rename (y,z,M)|]
24 ==> [Lam|[Atom y|[M|nil]]] -a- [Lam|[Atom z|[M'|nil]]]"
25
26 alpha_app       "[|M -a- M'; N -a- N'|]
27 ==> [App|[M|[N|nil]]] -a- [App|[M'|[N'|nil]]]"
28
29 alpha_fix       "[|M -a- M'|] ==> [Fix|[M|nil]] -a- [Fix|[M'|nil]]"
30
31 alpha_if        "[|M -a- M'; N -a- N'; P -a- P'|]

```

```

32         ==> [Cond|[M|[N|[P|nil]]]] -a- [Cond|[M'|[N'|[P'|nil]]]]"
33
34 alpha_cons    "[|M -a- M'; N -a- N'|]
35             ==> [Cons|[M|[N|nil]]] -a- [Cons|[M'|[N'|nil]]]"
36
37 alpha_car     "[|M -a- M'|] ==> [Car|[M|nil]] -a- [Car|[M'|nil]]"
38
39 alpha_cdr     "[|M -a- M'|] ==> [Cdr|[M|nil]] -a- [Cdr|[M'|nil]]"
40
41 alpha_iseq    "[|M -a- M'; N -a- N'|]
42             ==> [IsEq|[M|[N|nil]]] -a- [IsEq|[M'|[N'|nil]]]"
43
44 alpha_isatom  "[|M -a- M'|]
45             ==> [IsAtom|[M|nil]] -a- [IsAtom|[M'|nil]]"
46
47 alpha_error   "[|M -a- M'|] ==> [Error|[M|nil]] -a- [Error|[M'|nil]]"
48
49 alpha_sym     "M -a- M' ==> M' -a- M"
50
51 alpha_trans   "[|M -a- M'; M' -a- M''|] ==> M -a- M''"
52
53 end

1  (*****
2  *
3  * Alpha.ML
4  *
5  *****)
6  Addsimps alpha.intrs;
7  AddIs alpha.intrs;
8
9  Delsimps [alpha.alpha_sym,alpha.alpha_trans];
10 Delrules [alpha.alpha_sym,alpha.alpha_trans];
11
12 val alpelims = map (alpha.mk_cases sexpr.simps)
13 [
14     "M -a- M",
15     "[Lam|[Atom v|[M|nil]]] -a- [Lam|[Atom v'|[M'|nil]]]",
16     "[App|[M|[N|nil]]] -a- [App|[M'|[N'|nil]]]",
17     "[Fix|[M|nil]] -a- [Fix|[M'|nil]]",
18     "[Cond|[M|[N|[P|nil]]]] -a- [Cond|[M'|[N'|[P'|nil]]]]",
19     "[Cons|[M|[N|nil]]] -a- [Cons|[M'|[N'|nil]]]",
20     "[Car|[M|nil]] -a- [Car|[M'|nil]]",
21     "[Cdr|[M|nil]] -a- [Cdr|[M'|nil]]",
22     "[IsEq|[M|[N|nil]]] -a- [IsEq|[M'|[N'|nil]]]",
23     "[IsAtom|[M|nil]] -a- [IsAtom|[M'|nil]]",
24     "[Error|[M|nil]] -a- [Error|[M'|nil]]"
25 ];
26
27 AddSEs alpelims;

```

C.5 L1Expr

```

1  L1Expr = Defs +
2
3  consts
4      l1expr :: sexpr set
5

```

```

6   inductive l1expr intrs
7
8   l1expr_var      "a : var ==> Atom a : l1expr"
9
10  l1expr_quote    "[Quote|[d|nil]] : l1expr"
11
12  l1expr_lam      "[| a : var; M : l1expr |]
13                ==> [Lam|[Atom a|[M|nil]]] : l1expr"
14
15  l1expr_app      "[| M : l1expr ; N : l1expr |]
16                ==> [App|[M|[N|nil]]] : l1expr"
17
18  l1expr_fix      "[| M : l1expr |] ==> [Fix|[M|nil]] : l1expr"
19
20  l1expr_if       "[| B : l1expr ; M : l1expr ; N : l1expr |]
21                ==> [Cond|[B|[M|[N|nil]]]] : l1expr"
22
23  l1expr_cons     "[| M : l1expr ; N : l1expr |]
24                ==> [Cons|[M|[N|nil]]] : l1expr"
25
26  l1expr_car      "[| M : l1expr |] ==> [Car|[M|nil]] : l1expr"
27
28  l1expr_cdr      "[| M : l1expr |] ==> [Cdr|[M|nil]] : l1expr"
29
30  l1expr_iseq     "[| M : l1expr ; N : l1expr |]
31                ==> [IsEq|[M|[N|nil]]] : l1expr"
32
33  l1expr_isatom   "[| M : l1expr |] ==> [IsAtom|[M|nil]] : l1expr"
34
35  l1expr_error    "[| M : l1expr |] ==> [Error|[M|nil]] : l1expr"
36
37  end

1   (*****
2   *
3   * L1Expr.ML
4   *
5   *****)
6
7   Addsimps l1expr.intrs;
8   AddIs l1expr.intrs;
9
10  val l1elims = map (l1expr.mk_cases sexpr.simps)
11  [
12    "Atom a : l1expr",
13    "[Quote|[d|nil]] : l1expr",
14    "[Lam|[Atom v|[N|nil]]] : l1expr",
15    "[App|[M|[N|nil]]] : l1expr",
16    "[Fix|[M|nil]] : l1expr",
17    "[Cond|[M|[N|[P|nil]]]] : l1expr",
18    "[Cons|[M|[N|nil]]] : l1expr",
19    "[Car|[M|nil]] : l1expr",
20    "[Cdr|[M|nil]] : l1expr",
21    "[IsEq|[M|[N|nil]]] : l1expr",
22    "[IsAtom|[M|nil]] : l1expr",
23    "[Error|[M|nil]] : l1expr"
24  ];
25
26  AddSEs l1elims;

```

```

27
28 Goal "M:l1expr --> (!q. M ~= [delay|q])";
29 br impI 1;
30 by (etac l1expr.induct 1);
31 auto();
32 qed "l1notdelay";
33
34 Goal "[delay|q] ~: l1expr";
35 br notI 1;
36 by (eresolve_tac l1expr.elims 1);
37 auto();
38 qed "delaynotl1";
39
40 Addsimps [l1notdelay,delaynotl1];

```

C.6 L2Eval

```

1  (*****
2  *
3  * L2Eval.thy
4  *
5  *****)
6
7  L2Eval = L1Expr + WF_Rel +
8
9  consts
10     lookup :: "(sexpr*sexpr) => sexpr"
11     l2eval :: "(sexpr * sexpr * sexpr)set"
12     "@l2eval" :: [sexpr,sexpr,sexpr] => bool
13             ("_ |- _ / ----> _" [0,0,50] 50)
14
15  rendef lookup "measure (%(rho,v). size rho)"
16     "lookup (nil,v) = v"
17     "lookup ([|y|dy|rest],v) =
18         (if (v=y) then dy else lookup (rest,v))"
19
20  translations "en |- exp ----> res" == "(exp,en,res) : l2eval"
21
22  inductive l2eval intrs
23
24  l2eval_var_nonfix
25     "[|lookup (rho,Atom a) = d; !q. d ~= [delay|q]|]
26     ==> rho |- Atom a ----> d"
27
28  l2eval_var_fix "[|lookup (rho,Atom a) = [delay|[M|[rho'|nil]]];
29                 rho' |- M ----> d |] ==> rho |- Atom a ----> d"
30
31  l2eval_quote  "rho |- [Quote|[D|nil]] ----> D"
32
33  l2eval_lam    "rho |- [Lam|[v|[M|nil]]] ----> [clos|[v|[M|[rho|nil]]]]"
34
35  l2eval_app    "[|rho |- M ----> [clos|[v|[M'|[rho'|nil]]]];
36                 rho |- N ----> dN; [[v|dN|rho'] |- M' ----> d|]
37                 ==> rho |- [App|[M|[N|nil]]] ----> d"
38
39  l2eval_fix    "[|rho |- M ----> [clos|[v|[M'|[rho'|nil]]]];
40                 [[v|[delay|[Fix|[M|nil]]|[rho|nil]]]|rho'|
41                 |- M' ----> d |]

```

```

42      ==> rho |- [Fix|[M|nil]] ---> d"
43
44  l2eval_if_f    "[|rho |- M ---> dM; dM ~= #t ; rho |- P ---> dP|]
45      ==> rho |- [Cond|[M|[N|[P|nil]]]] ---> dP"
46
47  l2eval_if_t    "[|rho |- M ---> #t ; rho |- N ---> dN|]
48      ==> rho |- [Cond|[M|[N|[P|nil]]]] ---> dN"
49
50  l2eval_cons    "[|rho |- M ---> dM; rho |- N ---> dN|]
51      ==> rho |- [Cons|[M|[N|nil]]] ---> [dM|dN]"
52
53  l2eval_car     "[|rho |- M ---> [d1|d2] |]
54      ==> rho |- [Car|[M|nil]] ---> d1"
55
56  l2eval_cdr     "[|rho |- M ---> [d1|d2] |]
57      ==> rho |- [Cdr|[M|nil]] ---> d2"
58
59  l2eval_iseq_f  "[|rho |- M ---> dM; rho |- N ---> dN; dM ~= dN|]
60      ==> rho |- [IsEq|[M|[N|nil]]] ---> nil"
61
62  l2eval_iseq_t  "[|rho |- M ---> dM; rho |- N ---> dN; dM = dN|]
63      ==> rho |- [IsEq|[M|[N|nil]]] ---> #t"
64
65  l2eval_isatom_f "[|rho |- M --->dM; dM = [d1|d2]|]
66      ==> rho |- [IsAtom|[M|nil]] ---> nil"
67
68
69  l2eval_isatom_t "[|rho |- M --->dM; dM = (Atom a)|]
70      ==> rho |- [IsAtom|[M|nil]] ---> #t"
71
72
73  l2eval_lift    "[|rho |- M ---> dM|]
74      ==> rho |- [Lift|[M|nil]] ---> [Quote|[dM|nil]]"
75
76  l2eval_rquote  "rho |- [RQuote|[d|nil]] ---> [Quote|[d|nil]]"
77
78  l2eval_rlam    "[|Atom v' : l1expr; v' ~: (vars rho) Un (vars M);
79      [[Atom v|Atom v']|rho] |- M ---> dM|]
80      ==> rho |- [RLam|[Atom v|[M|nil]]]
81      ---> [Lam|[Atom v'|[dM|nil]]]"
82
83  l2eval_rapp    "[|rho |- M ---> dM; rho |- N ---> dN|]
84      ==> rho |- [RApp|[M|[N|nil]]] ---> [App|[dM|[dN|nil]]]"
85
86  l2eval_rfix    "[|rho |- M ---> dM|]
87      ==> rho |- [RFix|[M|nil]] ---> [Fix|[dM|nil]]"
88
89  l2eval_rif     "[|rho |- M ---> dM; rho |- N ---> dN;
90      rho |- P ---> dP|]
91      ==> rho |- [RCond|[M|[N|[P|nil]]]]
92      ---> [Cond|[dM|[dN|[dP|nil]]]]"
93
94  l2eval_rcons   "[|rho |- M ---> dM; rho |- N ---> dN|]
95      ==> rho |- [RCons|[M|[N|nil]]]
96      ---> [Cons|[dM|[dN|nil]]]"
97
98  l2eval_rcar    "[|rho |- M ---> dM|]
99      ==> rho |- [RCar|[M|nil]] ---> [Car|[dM|nil]]"

```

```

100
101 l2eval_rcdr      "[|rho |- M ---> dM|]
102                ==> rho |- [RCdr|[M|nil]] ---> [Cdr|[dM|nil]]"
103
104 l2eval_riseq     "[|rho |- M ---> dM; rho |- N ---> dN|]
105                ==> rho |- [RISeq|[M|[N|nil]]] ---> [IsEq|[dM|[dN|nil]]]"
106
107 l2eval_risatom   "[|rho |- M ---> dM|]
108                ==> rho |- [RIsAtom|[M|nil]] ---> [IsAtom|[dM|nil]]"
109
110
111 l2eval_rerror    "[|rho |- M ---> dM|]
112                ==> rho |- [RError|[M|nil]] ---> [Error|[dM|nil]]"
113
114 end

1  (*****
2  *
3  * L2Eval.ML
4  *
5  *****)
6
7  Addsimps lookup.rules;
8  Addsimps l2eval.intrs;
9  AddIs l2eval.intrs;
10
11  val l2elims = map (l2eval.mk_cases (sexpr.simps @ l1expr.defs))
12  [
13      "rho |- Atom v ---> d",
14      "rho |- [Quote|[M|nil]] ---> d",
15      "rho |- [Lam|[Atom v|[M|nil]]] ---> d",
16      "rho |- [App|[M|[N|nil]]] ---> d",
17      "rho |- [Fix|[M|nil]] ---> d",
18      "rho |- [Cond|[M|[N|[P|nil]]]] ---> d",
19      "rho |- [Cons|[M|[N|nil]]] ---> d",
20      "rho |- [Car|[M|nil]] ---> d",
21      "rho |- [Cdr|[M|nil]] ---> d",
22      "rho |- [IsEq|[M|[N|nil]]] ---> d",
23      "rho |- [IsAtom|[M|nil]] ---> d",
24      "rho |- [Error|[M|nil]] ---> d",
25      "rho |- [Lift|[M|nil]] ---> d",
26      "rho |- [RQuote|[M|nil]] ---> d",
27      "rho |- [RLam|[Atom v|[M|nil]]] ---> d",
28      "rho |- [RApp|[M|[N|nil]]] ---> d",
29      "rho |- [RFix|[M|nil]] ---> d",
30      "rho |- [RCond|[M|[N|[P|nil]]]] ---> d",
31      "rho |- [RCons|[M|[N|nil]]] ---> d",
32      "rho |- [RCar|[M|nil]] ---> d",
33      "rho |- [RCdr|[M|nil]] ---> d",
34      "rho |- [RISeq|[M|[N|nil]]] ---> d",
35      "rho |- [RIsAtom|[M|nil]] ---> d",
36      "rho |- [RError|[M|nil]] ---> d"
37  ];
38
39  AddSEs l2elims;
40
41  val deriv_forw = fn i =>
42      REPEAT (FIRST [Force_tac i, swap_res_tac l2eval.intrs i]);
43

```

```

44 val deriv_back = fn i =>
45     REPEAT (FIRST [CHANGED (Asm_full_simp_tac i),hyp_subst_tac i,
46         etac conjE i,eresolve_tac l2elims i]);

```

C.7 LMix

```

1  (*****
2  *
3  * LMix.thy
4  *
5  *****)
6
7  LMix = L2Eval + Alpha + LMixEnv + Sint + NatInf +
8  end

```

```

1  (*****
2  *
3  * LMix.ML
4  *
5  *****)
6
7  use "LWrap";
8  use "Eval";
9  use "Dispatch";
10 use "Deriv";

```

C.8 LemRen

```

1  (*****
2  *
3  * LemRen.thy
4  *
5  *****)
6
7  LemRen = LMix +
8
9  consts
10     addvar :: [atom,rlist] => rlist
11
12 primrec
13     addvar_none "addvar y [] = []"
14     addvar_nemp "addvar y [v';v;Mv,rlist] =
15         [v';v;[App|[Mv|[Atom y|nil]]],addvar y rlist]"
16
17 end

```

```

1  (*****
2  *
3  * LemRen.ML
4  *
5  *****)
6
7  use "EnvLem";
8  use "LemEnv";
9  use "Corr";
10 use "LemTerm";
11

```

```

12 Goal "M : l1expr & v' : var --> rename (v,v',M) : l1expr";
13 br impI 1;
14 by (res_inst_tac [("xa","M")] l1expr.induct 1);
15 auto();
16 qed "renl1";
17
18 Goal "M : l1expr & v' : var --> \
19 \ vars (rename (v,v',M)) <= vars M Un {v'}";
20 br impI 1;
21 by (res_inst_tac [("xa","M")] l1expr.induct 1);
22 auto();
23 qed "ren_lemma1";
24
25 Goal "vlist : lmixrlist --> (<M,[v';v;Mv,vlist]> : lmixenv --> \
26 \ (<rename (v,v',M),addvar v' vlist> : lmixenv))";
27 auto();
28 by (eresolve_tac lmixenv.elims 1);
29 force 1;
30 by (subgoal_tac "[v';v;Mv,vlist] : lmixrlist" 1);
31 force 2;
32 by (subgoal_tac "addvar v' vlist : lmixrlist" 1);
33 by (res_inst_tac [("y2","v"),("My2","Mv")]
34 ((add_lemma1 RS mp) RS mp) 2);
35 force 2;
36 force 2;
37 auto();
38 by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
39 auto();
40 by (swap_res_tac lmixenv.intrs 1);
41 br (renl1 RS mp) 1;
42 force 1;
43 by (swap_res_tac lmixenv.intrs 1);
44 br (renl1 RS mp) 1;
45 force 1;
46 force 1;
47 by (subgoal_tac "vars (rename (v,v',M)) <= vars M Un {v'}" 1);
48 br (ren_lemma1 RS mp) 2;
49 force 2;
50 auto();
51 qed "ren_lemma2";
52
53 Goal "vlist : lmixrlist --> varsin (addvar y vlist) = varsin vlist";
54 br impI 1;
55 by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
56 force 1;
57 force 1;
58 force 1;
59 qed "ren_lemma3";
60
61 Addsimps [ren_lemma3,lemma_511];
62
63 use "LemWeak";
64
65 Goal "M : l1expr --> (!vlist. <M,[v';v;Mv,vlist]> : lmixenv --> \
66 \ (vlist : lmixrlist & <rename (v,v',M),addvar v' vlist> : lmixenv) \
67 \ --> (!d. (<M,[v';v;Mv,vlist]> |- bodyann ---> d) --> \
68 \ (<rename (v,v',M),addvar v' vlist> |- bodyann ---> d)))";
69 br impI 1;

```



```

70 by (res_inst_tac [("xa","M")] llexpr.induct 1);
71 force 1;
72 force 2;
73
74 (** App case **)
75 br allI 3;
76 br impI 3;
77 br impI 3;
78 br allI 3;
79 br impI 3;
80 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
81 by (subgoal_tac "<N, [v';v;Mv,vlist]> : lmixenv" 3);
82 by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")]) (env_lemma6 RS mp) 4);
83 force 4;
84 by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")]) (env_lemma6 RS mp) 4);
85 force 4;
86 by (Asm_full_simp_tac 3);
87 by (REPEAT (etac exE 3));
88 by (res_inst_tac [("x","dM")] exI 3);
89 by (res_inst_tac [("x","dN")] exI 3);
90 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
91 by (subgoal_tac "<rename (v,v',N),addvar v' vlist> : lmixenv" 3);
92 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
93 force 4;
94 force 4;
95 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
96 force 4;
97 force 4;
98 force 3;
99
100 (** Fix case **)
101 br allI 3;
102 br impI 3;
103 br impI 3;
104 br allI 3;
105 br impI 3;
106 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
107 by (res_inst_tac [("M'1","[Fix|Ma|[nil]]")]) (env_lemma6 RS mp) 4);
108 force 4;
109 by (Asm_full_simp_tac 3);
110 by (REPEAT (etac exE 3));
111 by (res_inst_tac [("x","dM")] exI 3);
112 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
113 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
114 force 4;
115 force 4;
116 force 3;
117
118 (** Cond case **)
119 br allI 3;
120 br impI 3;
121 br impI 3;
122 br allI 3;
123 br impI 3;
124 by (subgoal_tac "<B, [v';v;Mv,vlist]> : lmixenv" 3);
125 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
126 by (subgoal_tac "<N, [v';v;Mv,vlist]> : lmixenv" 3);
127 by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")])

```

```

128     (env_lemma6 RS mp) 4);
129 force 4;
130 by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")])
131     (env_lemma6 RS mp) 4);
132 force 4;
133 by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")])
134     (env_lemma6 RS mp) 4);
135 force 4;
136 by (Asm_full_simp_tac 3);
137 by (REPEAT (etac exE 3));
138 by (res_inst_tac [("x","dM")] exI 3);
139 by (res_inst_tac [("x","dN")] exI 3);
140 by (res_inst_tac [("x","dP")] exI 3);
141 by (subgoal_tac "<rename (v,v',B),addvar v' vlist> : lmixenv" 3);
142 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
143 by (subgoal_tac "<rename (v,v',N),addvar v' vlist> : lmixenv" 3);
144 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
145 force 4;
146 force 4;
147 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
148 force 4;
149 force 4;
150 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
151 force 4;
152 force 4;
153 force 3;
154
155 (** Cons case **)
156 br allI 3;
157 br impI 3;
158 br impI 3;
159 br allI 3;
160 br impI 3;
161 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
162 by (subgoal_tac "<N, [v';v;Mv,vlist]> : lmixenv" 3);
163 by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
164 force 4;
165 by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
166 force 4;
167 by (Asm_full_simp_tac 3);
168 by (REPEAT (etac exE 3));
169 by (res_inst_tac [("x","dM")] exI 3);
170 by (res_inst_tac [("x","dN")] exI 3);
171 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
172 by (subgoal_tac "<rename (v,v',N),addvar v' vlist> : lmixenv" 3);
173 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
174 force 4;
175 force 4;
176 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
177 force 4;
178 force 4;
179 force 3;
180
181 (** Car case **)
182 br allI 3;
183 br impI 3;
184 br impI 3;
185 br allI 3;

```

```

186 br impI 3;
187 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
188 by (res_inst_tac [("M'1","[Car|[Ma|nil]]")]) (env_lemma6 RS mp) 4);
189 force 4;
190 by (Asm_full_simp_tac 3);
191 by (REPEAT (etac exE 3));
192 by (res_inst_tac [("x","dM")] exI 3);
193 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
194 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
195 force 4;
196 force 4;
197 force 3;
198
199 (** Cdr case **)
200 br allI 3;
201 br impI 3;
202 br impI 3;
203 br allI 3;
204 br impI 3;
205 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
206 by (res_inst_tac [("M'1","[Cdr|[Ma|nil]]")]) (env_lemma6 RS mp) 4);
207 force 4;
208 by (Asm_full_simp_tac 3);
209 by (REPEAT (etac exE 3));
210 by (res_inst_tac [("x","dM")] exI 3);
211 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
212 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
213 force 4;
214 force 4;
215 force 3;
216
217 (** IsEq case **)
218 br allI 3;
219 br impI 3;
220 br impI 3;
221 br allI 3;
222 br impI 3;
223 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
224 by (subgoal_tac "<N, [v';v;Mv,vlist]> : lmixenv" 3);
225 by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
226 force 4;
227 by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
228 force 4;
229 by (Asm_full_simp_tac 3);
230 by (REPEAT (etac exE 3));
231 by (res_inst_tac [("x","dM")] exI 3);
232 by (res_inst_tac [("x","dN")] exI 3);
233 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
234 by (subgoal_tac "<rename (v,v',N),addvar v' vlist> : lmixenv" 3);
235 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
236 force 4;
237 force 4;
238 by (res_inst_tac [("Mv2","Mv"]) ((ren_lemma2 RS mp) RS mp) 4);
239 force 4;
240 force 4;
241 force 3;
242
243 (** IsAtom case **)

```

```

244 br allI 3;
245 br impI 3;
246 br impI 3;
247 br allI 3;
248 br impI 3;
249 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
250 by (res_inst_tac [("M'1","[IsAtom|Ma|nil]")] (env_lemma6 RS mp) 4);
251 force 4;
252 by (Asm_full_simp_tac 3);
253 by (REPEAT (etac exE 3));
254 by (res_inst_tac [("x","dM")] exI 3);
255 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
256 by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
257 force 4;
258 force 4;
259 force 3;
260
261 (** Error case **)
262 br allI 3;
263 br impI 3;
264 br impI 3;
265 br allI 3;
266 br impI 3;
267 by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
268 by (res_inst_tac [("M'1","[Error|Ma|nil]")] (env_lemma6 RS mp) 4);
269 force 4;
270 by (Asm_full_simp_tac 3);
271 by (REPEAT (etac exE 3));
272 by (res_inst_tac [("x","dM")] exI 3);
273 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
274 by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
275 force 4;
276 force 4;
277 force 3;
278
279 (** Lam case **)
280 br allI 2;
281 br impI 2;
282 br impI 2;
283 br allI 2;
284 br impI 2;
285 by (subgoal_tac "<Ma,[v';v;Mv,vlist]> : lmixenv" 2);
286 by (res_inst_tac [("M'1","[Lam|Atom a|[Ma|nil]]")]
287   (env_lemma6 RS mp) 3);
288 force 3;
289 by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 2);
290 by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 3);
291 force 3;
292 force 3;
293
294 by (Asm_full_simp_tac 2);
295 br conjI 2;
296 by (eres_inst_tac [("x","vlist")] alle 2);
297
298 br impI 2;
299 by (Asm_full_simp_tac 2);
300 by (REPEAT (etac exE 2));
301 by (Asm_full_simp_tac 2);

```

```

302 br conjI 2;
303 by (REPEAT (etac conjE 2));
304 by (REPEAT (etac conjE 3));
305 br corr_513 3;
306 force 3;
307 force 3;
308 force 3;
309 by (swap_res_tac lmixenv.intrs 2);
310 force 2;
311 by (swap_res_tac lmixrlist.intrs 2);
312 by (res_inst_tac [("y2","a"),("My2","Mv")]
313       ((add_lemma1 RS mp) RS mp) 2);
314 force 2;
315 by (eresolve_tac lmixenv.elims 2);
316 force 2;
317 force 2;
318 force 2;
319 force 2;
320 by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,[v'a;Mv,vlist]]>")]
321       lmixenv.elim 2);
322 force 2;
323 force 2;
324 by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,[v'a;Mv,vlist]]>")]
325       lmixenv.elim 2);
326 force 2;
327 by (res_inst_tac [("y","vlist")] rlist.exhaust 3);
328 force 3;
329 force 4;
330 by (Asm_full_simp_tac 3);
331
332 by (eresolve_tac lmixenv.elims 3);
333 force 3;
334
335 by (dres_inst_tac [("s","<[Lam|[Atom a|[Ma|nil]]], \
336 \ [v'a;Mv,[atom1;atom2;sexpr,rlist]]>")] sym 3);
337
338 by (Asm_full_simp_tac 3);
339 force 3;
340 by (dres_inst_tac [("s","<Ma,[v'a;a;Ma,[v'a;Mv,vlist]]>")] sym 2);
341 by (Asm_full_simp_tac 2);
342 by (eres_inst_tac [("a","[v'a;a;Ma,[v'a;Mv,vlist]]")
343       lmixrlist.elim 2);
344 by (Asm_full_simp_tac 2);
345 by (dres_inst_tac [("s","[v'a;a;Ma,[v'a;Mv,vlist]]")
346       sym 2);
347 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
348 br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 2;
349 force 2;
350 force 2;
351
352 br impI 2;
353 by (Asm_full_simp_tac 2);
354 by (REPEAT (etac exE 2));
355 by (Asm_full_simp_tac 2);
356
357 by (subgoal_tac "<rename (v,v',Ma), \
358 \ [v'a;a;rename (v,v',Ma),addvar v' vlist]> : lmixenv" 2);
359 br conjI 2;

```

```

360 force 2;
361 by (defer_tac 2);
362 by (REPEAT (etac conjE 2));
363 by (REPEAT (etac conjE 3));
364
365 by (swap_res_tac lmixenv.intrs 2);
366 br (renl1 RS mp) 2;
367 by (eresolve_tac lmixenv.elims 2);
368 force 2;
369 force 2;
370 force 3;
371 by (swap_res_tac lmixrlist.intrs 2);
372 by (subgoal_tac "[v';v;Mv,vlist] : lmixrlist" 2);
373 by (res_inst_tac [("y2","v"),("My2","Mv")]
374       ((add_lemma1 RS mp) RS mp) 2);
375 force 2;
376 force 2;
377 by (eresolve_tac lmixenv.elims 2);
378 force 2;
379 force 2;
380 br (renl1 RS mp) 2;
381 by (eresolve_tac lmixenv.elims 2);
382 force 2;
383 force 2;
384 force 2;
385 by (res_inst_tac [("a","<Ma,[v'a;a;Ma,[v';v;Mv,vlist]]>")]
386       lmixenv.elim 2);
387 force 2;
388 force 2;
389 force 2;
390 by (res_inst_tac [("a","<Ma,[v'a;a;Ma,[v';v;Mv,vlist]]>")]
391       lmixenv.elim 2);
392 force 2;
393 force 2;
394 by (dres_inst_tac [("s","<Ma,[v'a;a;Ma,[v';v;Mv,vlist]]>")] sym 2);
395 by (Asm_full_simp_tac 2);
396 by (eres_inst_tac [("a","[v'a;a;Ma,[v';v;Mv,vlist]]")
397       lmixrlist.elim 2);
398 force 2;
399 by (dres_inst_tac [("s","[v'a;a;Ma,[v';v;Mv,vlist]]")
400       sym 2);
401 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
402 br conjI 2;
403 br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 2;
404 force 2;
405 force 2;
406 force 2;
407 br ((ren_lemma1 RS mp) RS sub_lemma1) 2;
408 force 2;
409 force 2;
410
411 by (eres_inst_tac [("x","vlist")] alle 2);
412 by (res_inst_tac [("y","vlist")] rlist.exhaust 2);
413 force 2;
414 by (Asm_full_simp_tac 2);
415 by (eresolve_tac lmixenv.elims 2);
416 force 2;
417 by (dres_inst_tac [("s","<[Lam|Atom a|[Ma|nil]]], \

```

```

418 \ [v';v;Mv,[atom1;atom2;sexpr,rlist]]>")) sym 2);
419 by (Asm_full_simp_tac 2);
420 br conjI 2;
421 force 3;
422 br impI 2;
423 br conjI 2;
424 by (eres_inst_tac
425   [("a", "<Ma, [v'a;a;Ma, [v';v;Mv, [atom1;atom2;sexpr,rlist]]>")]
426     lmixenv.elim 2);
427 force 2;
428 force 2;
429 by (subgoal_tac "vars Ma <= vars sexpr" 2);
430 force 3;
431 by (subgoal_tac "vars (rename (v,v',Ma)) <= vars Ma Un {v'}" 2);
432 br (ren_lemma1 RS mp) 3;
433 force 3;
434 by (subgoal_tac "vars Ma Un {v'} <= vars sexpr Un {v'}" 2);
435 force 3;
436 by (subgoal_tac "vars sexpr Un {v'} <= insert v' (vars sexpr)" 2);
437 force 3;
438 by (thin_tac "?xx" 2);
439 by (thin_tac "?xx" 2);
440 by (thin_tac "?xx" 2);
441 by (thin_tac "?xx" 2);
442 by (thin_tac "?xx" 2);
443 by (thin_tac "?xx" 2);
444 by (thin_tac "?xx" 2);
445 by (thin_tac "?xx" 2);
446 by (thin_tac "?xx" 2);
447 by (thin_tac "?xx" 2);
448 by (thin_tac "?xx" 2);
449 by (thin_tac "?xx" 2);
450 by (thin_tac "?xx" 2);
451 by (thin_tac "?xx" 2);
452 by (thin_tac "?xx" 2);
453 by (thin_tac "?xx" 2);
454 by (thin_tac "?xx" 2);
455 force 2;
456
457 by (subgoal_tac "<Ma, [] # \
458 \ [v';v;Mv,[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]> : lmixenv" 2);
459 br (env_lemma3 RS mp) 3;
460 force 3;
461 by (subgoal_tac "<Ma, [v';v;Mv,[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]> \
462 \ |- bodyann ---> dM" 2);
463 br corr_512 3;
464 force 3;
465 force 3;
466 force 3;
467 by (eres_inst_tac [("x", "[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]")]
468   allE 2);
469 by (Asm_full_simp_tac 2);
470
471 by (subgoal_tac "<rename(v,v',Ma), \
472 \ addvar v' [v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]> : lmixenv" 2);
473 by (res_inst_tac [("Mv2", "Mv")] ((ren_lemma2 RS mp) RS mp) 3);
474 by (eres_inst_tac
475   [("a", "<Ma, [v';v;Mv,[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]>")]

```

```

476         lmixenv.elim 3);
477 force 3;
478 force 3;
479 force 3;
480 by (Asm_full_simp_tac 2);
481 by (eres_inst_tac
482     [("a", "<Ma, [v';v;Mv, [v'a;a;[App| [Mv| [Atom v|nil]]], vlist]]>")]
483     lmixenv.elim 2);
484 force 2;
485 by (dres_inst_tac [("t", "<Maa, [v'aa;va;M', vlista]>")] sym 2);
486 by (Asm_full_simp_tac 2);
487 by (SELECT_GOAL Auto_tac 2);
488 by (eres_inst_tac [("x", "dM")] allE 2);
489 by (mp_tac 2);
490
491 by (res_inst_tac [("M2", "rename (v, v', Ma)"), ("Mv'2", "rename (v, v', Ma)"),
492     ("Mv2", "[App| [[App| [Mv| [Atom v|nil]]] | [Atom v'|nil]]]"),
493     ("vlist'2", "addvar v' vlist"), ("v'2", "v'a"), ("v2", "a"),
494     ("x", "[ ]") ((lem_weak RS mp) RS allE) 2);
495 br (ren1 RS mp) 2;
496 force 2;
497 by (Asm_full_simp_tac 2);
498 by (subgoal_tac "vars (rename (v, v', Ma)) <= vars \
499 \ [App| [[App| [Mv| [Atom v|nil]]] | [Atom v'|nil]]]" 2);
500 force 2;
501 by (Asm_full_simp_tac 2);
502 by (eres_inst_tac [("a", "<rename (v, v', Ma), \
503 \ [v'a;a;[App| [[App| [Mv| [Atom v|nil]]] | [Atom v'|nil]]], \
504 \ addvar v' vlist]>")] lmixenv.elim 2);
505 force 2;
506 by (SELECT_GOAL Auto_tac 2);
507
508 (** Var case **)
509 br allI 1;
510 br impI 1;
511 br impI 1;
512 be conjE 1;
513 br allI 1;
514 br impI 1;
515 by (case_tac "a = v" 1);
516
517 by (Asm_full_simp_tac 1);
518 by (subgoal_tac "v' ~: varsin (addvar v' vlist)" 1);
519 force 1;
520 by (Asm_full_simp_tac 1);
521 by (eresolve_tac lmixenv.elims 1);
522 force 1;
523 by (dtac sym 1);
524 by (Asm_full_simp_tac 1);
525 by (SELECT_GOAL Auto_tac 1);
526 by (subgoal_tac "v' ~: varsin vlista" 1);
527 force 1;
528 br ((vars_lemma1 RS mp) RS sub_lemma1) 1;
529 force 1;
530 force 1;
531
532 by (SELECT_GOAL Auto_tac 1);
533 by (SELECT_GOAL Auto_tac 1);

```



```

534
535 by (res_inst_tac [("y","a"),("v'","v'"),("v","v"),("M","Mv"),
536       ("vlist","vlist"),("d","d")] derivelimvar1 1);
537 force 1;
538 br add_lemma2b 1;
539 force 1;
540 force 1;
541 qed "lemma_515";

```

C.9 NatInf

```

1  (*****
2  *
3  * NatInf.thy
4  *
5  *****)
6
7  NatInf = Nat + Finite + SExpr +
8
9  consts
10     "new_var" :: "[nat,nat set] => bool"
11
12  defs
13     new_var_def "new_var n nset == (n : var) & (!m:nset. m < n)"
14
15  end

1  (*****
2  *
3  * NatInf.ML
4  *
5  *****)
6
7  Addsimps [new_var_def,max_def];
8
9  Goal "finite nset ==> (? n. new_var n nset)";
10 auto();
11 br Finites.induct 1;
12 force 1;
13 force 1;
14 be exE 1;
15 by (res_inst_tac [("x","Suc (max a n)"] exI 1);
16 auto();
17 qed "newexists";
18
19 Goal "finite nset ==> (? v. (v : var) & (v ~: nset))";
20 by (subgoal_tac "(? n. new_var n nset)" 1);
21 br newexists 2;
22 force 2;
23 be exE 1;
24 by (res_inst_tac [("x","n"] exI 1);
25 force 1;
26 qed "freshvar";

```

C.10 Rename

```

1  (*****

```

```

2  *
3  * Rename.thy
4  *
5  *****)
6
7  Rename = L1Expr + WF_Rel +
8
9  consts
10     rename :: "(atom*atom*sexpr) => sexpr"
11
12  recdef rename "measure (%(y,z,M). size M)"
13
14  "rename (y,z,Atom w) = (if y=w then (Atom z) else (Atom w))"
15
16  "rename (y,z,[Quote|d|nil]) = [Quote|d|nil]"
17
18  "rename (y,z,[Lam|[Atom w|[M|nil]]]) =
19     (if y=w then [Lam|[Atom w|[M|nil]]]
20      else [Lam|[Atom w|[rename (y,z,M)|nil]]])"
21
22  "rename (y,z,[App|[M|[N|nil]]]) =
23     [App|[rename (y,z,M)|[rename (y,z,N)|nil]]]"
24
25  "rename (y,z,[Fix|[M|nil]]) = [Fix|[rename (y,z,M)|nil]]"
26
27  "rename (y,z,[Cond|[M|[N|[P|nil]]]]) =
28     [Cond|[rename (y,z,M)|[rename (y,z,N)|[rename (y,z,P)|nil]]]]"
29
30  "rename (y,z,[Cons|[M|[N|nil]]]) =
31     [Cons|[rename (y,z,M)|[rename (y,z,N)|nil]]]"
32
33  "rename (y,z,[Car|[M|nil]]) = [Car|[rename (y,z,M)|nil]]"
34
35  "rename (y,z,[Cdr|[M|nil]]) = [Cdr|[rename (y,z,M)|nil]]"
36
37  "rename (y,z,[IsEq|[M|[N|nil]]]) =
38     [IsEq|[rename (y,z,M)|[rename (y,z,N)|nil]]]"
39
40  "rename (y,z,[IsAtom|[M|nil]]) = [IsAtom|[rename (y,z,M)|nil]]"
41
42  "rename (y,z,[Error|[M|nil]]) = [Error|[rename (y,z,M)|nil]]"
43
44  end

1  (*****)
2  *
3  * Rename.ML
4  *
5  *****)
6
7  Addsimps rename.rules;

```

C.11 Sint

```

1  (*****)
2  *
3  * Sint.thy
4  *

```

```

5      *****)
6
7      Sint = L1Expr +
8
9      consts
10         bodyann :: sexpr
11         sintann :: sexpr
12
13     syntax
14         "Cadr"      :: sexpr => sexpr ("Cadr")
15         "Caddr"     :: sexpr => sexpr ("Caddr")
16         "Caddrdr"  :: sexpr => sexpr ("Caddrdr")
17         "bodyann'"  :: sexpr ("bodyann'")
18         "sintann'"  :: sexpr ("sintann'")
19         "envlam"    :: sexpr ("envlam")
20
21     translations
22
23     "envlam" == "[Lam| [var|
24                 [[Cond| [[IsEq| [var| [[Cadr| [expr|nil]]|nil]]]|
25                 [value| [[App| [env| [var|nil]]]|nil]]]|nil]]]"
26
27     "[Cadr| [M|nil]]" == "[Car| [[Cdr| [M|nil]]|nil]]"
28
29     "[Caddr| [M|nil]]" == "[Cadr| [[Cdr| [M|nil]]|nil]]"
30
31     "[Caddrdr| [M|nil]]" == "[Caddr| [[Cdr| [M|nil]]|nil]]"
32
33     "bodyann'" ==
34     "[Cond| [[IsAtom| [expr|nil]]]|
35     [[App| [env| [expr|nil]]]|
36
37     [[Cond| [[IsEq| [[Car| [expr|nil]]| [[Quote| [Quote|nil]]|nil]]]|
38     [[Lift| [[Cadr| [expr|nil]]|nil]]]|
39
40     [[Cond| [[IsEq| [[Car| [expr|nil]]| [[Quote| [Lam|nil]]|nil]]]|
41     [[RLam| [value|
42     [[App| [[App| [eval| [[Caddr| [expr|nil]]|nil]]]| [envlam|nil]]]|nil]]]|
43
44     [[Cond| [[IsEq| [[Car| [expr|nil]]| [[Quote| [App|nil]]|nil]]]|
45     [[RApp| [[App| [[App| [eval| [[Cadr| [expr|nil]]|nil]]]| [env|nil]]]|
46     [[App| [[App| [eval| [[Caddr| [expr|nil]]|nil]]]| [env|nil]]]|nil]]]|
47
48     [[Cond| [[IsEq| [[Car| [expr|nil]]| [[Quote| [Fix|nil]]|nil]]]|
49     [[RFix| [[App| [[App| [eval| [[Cadr| [expr|nil]]|nil]]]| [env|nil]]]|nil]]]|
50
51     [[Cond| [[IsEq| [[Car| [expr|nil]]| [[Quote| [Cond|nil]]|nil]]]|
52     [[RCond| [[App| [[App| [eval| [[Cadr| [expr|nil]]|nil]]]| [env|nil]]]|
53     [[App| [[App| [eval| [[Caddr| [expr|nil]]|nil]]]| [env|nil]]]|
54     [[App| [[App| [eval| [[Caddrdr| [expr|nil]]|nil]]]| [env|nil]]]|nil]]]|
55
56     [[Cond| [[IsEq| [[Car| [expr|nil]]| [[Quote| [Cons|nil]]|nil]]]|
57     [[RCons| [[App| [[App| [eval| [[Cadr| [expr|nil]]|nil]]]| [env|nil]]]|
58     [[App| [[App| [eval| [[Caddr| [expr|nil]]|nil]]]| [env|nil]]]|nil]]]|
59
60     [[Cond| [[IsEq| [[Car| [expr|nil]]| [[Quote| [Car|nil]]|nil]]]|
61     [[RCar| [[App| [[App| [eval| [[Cadr| [expr|nil]]|nil]]]| [env|nil]]]|nil]]]|
62

```

```

63   [[Cond|[[IsEq|[[Car|expr|nil]]|[[Quote|Cdr|nil]]|nil]]|
64     [[RCdr|[[App|[[App|eval|[[Cadr|expr|nil]]|nil]]|env|nil]]|nil]]|
65
66   [[Cond|[[IsEq|[[Car|expr|nil]]|[[Quote|IsEq|nil]]|nil]]|
67     [[RIsEq|[[App|[[App|eval|[[Cadr|expr|nil]]|nil]]|env|nil]]|
68     [[App|[[App|eval|[[Caddr|expr|nil]]|nil]]|env|nil]]|nil]]|
69
70   [[Cond|[[IsEq|[[Car|expr|nil]]|[[Quote|IsAtom|nil]]|nil]]|
71     [[RIsAtom|[[App|[[App|eval|[[Cadr|expr|nil]]|nil]]|env|nil]]|nil]]|
72
73   [[Cond|[[IsEq|[[Car|expr|nil]]|[[Quote|Error|nil]]|nil]]|
74     [[RError|[[App|[[App|eval|[[Cadr|expr|nil]]|nil]]|env|nil]]|nil]]|
75
76   [[RError|expr|nil]]|nil]]|nil]]|nil]]|nil]]|nil]]|nil]]|nil]]|
77   nil]]|nil]]|nil]]|nil]]|nil]]|nil]]|nil]]|nil]]|nil]]|
78
79   "sintann'" ==
80     "[Fix|[[Lam|eval|[[Lam|expr|[[Lam|env|bodyann|nil]]|nil]]|nil]]|nil]]]"
81
82   defs
83     bodyann_def "bodyann == bodyann'"
84     sintann_def "sintann == sintann'"
85
86   end

```

C.12 Miscellaneous Proofs

```

1   (*****
2   *
3   * Corr.ML
4   *
5   *****)
6
7   Goal "[|y ~ = z; <M,[y';y;My,[z';z;Mz,vlist']> : lmixenv; \
8     \ <M,[y';y;My,[z';z;Mz,vlist']> |- bodyann ---> d |] \
9     \ ==> (<M,[z';z;Mz,[y';y;[App|Mz|[Atom z|nil]]],vlist']> \
10    \ |- bodyann ---> d)";
11   by (res_inst_tac [("x","[]"),("y2","y"),("z2","z"),("M2","M"),
12     ("y'2","y"),("My2","My"),("z'2","z"),("Mz2","Mz"),
13     ("vlist'2","vlist")] ((lemma_512 RS mp) RS allE) 1);
14   by (eresolve_tac lmixenv.elims 1);
15   auto();
16   qed "corr_512";
17
18   Goal "[|y = z;<M,[y';y;My,[z';z;Mz,vlist']> : lmixenv; \
19     \ <M,[y';y;My,[z';z;Mz,vlist']> |- bodyann ---> d |] \
20     \ ==> (<M,[y';y;My,addvar z' vlist']> |- bodyann ---> d)";
21   by (res_inst_tac [("x","[]"),("y2","y"),("z2","z"),("M2","M"),
22     ("y'2","y"),("My2","My"),("z'2","z"),("Mz2","Mz"),
23     ("vlist'2","vlist")] ((lemma_513 RS mp) RS allE) 1);
24   by (eresolve_tac lmixenv.elims 1);
25   auto();
26   qed "corr_513";

```

```

1   (*****
2   *
3   * Deriv.ML
4   *

```

```

5      *****)
6
7      Addsimps [vars_atom,vars_pair];
8
9      (*****)
10     (* Lemmata *)
11     (*****)
12
13     Goal "(<Atom v,vlist> : lmixenv) --> \
14     \ (!d. (<Atom v,vlist> |- env ---> d) = \
15     \ ([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] \
16     \ |- env ---> d))";
17     auto();
18     qed "envenv";
19
20     Goal "(<Atom v,vlist> : lmixenv) --> \
21     \ (!d. (<Atom v,vlist> |- expr ---> d) = \
22     \ ([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] \
23     \ |- var ---> d))";
24     auto();
25     by (deriv_back 2);
26     by (deriv_forw 1);
27     qed "varexpr";
28
29     Goal "(!dM. (rho |- M ---> dM) = (rho' |- M' --->dM)) \
30     \ & (!dN. (rho |- N ---> dN) = (rho' |- N' ---> dN))) --> \
31     \ (!d. (rho |- [App|[M|[N|nil]]] ---> d) = \
32     \ (rho' |- [App|[M'|[N'|nil]]] ---> d))";
33     auto();
34     by (eresolve_tac l2elims 1);
35     by (Asm_full_simp_tac 1);
36     by (swap_res_tac l2eval.intrs 1);
37     auto();
38     by (subgoal_tac "!dM.(rho'|-M' ---> dM)=(rho|-M --->dM)" 1);
39     by (subgoal_tac "!dN.(rho'|-N' ---> dN)=(rho|-N --->dN)" 1);
40     force 2;
41     force 2;
42     by (thin_tac "?Q" 1);
43     by (thin_tac "?Q" 1);
44     by (rotate_tac ~2 1);
45     by (eresolve_tac l2elims 1);
46     by (Asm_full_simp_tac 1);
47     by (swap_res_tac l2eval.intrs 1);
48     auto();
49     qed "appdet";
50
51     (*****)
52     (*****)
53
54     Goal "(<Atom v,vlist> : lmixenv) --> \
55     \ (!d. (<Atom v,vlist> |- [App|[env|[expr|nil]]] ---> d) = \
56     \ ([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] \
57     \ |- [App|[env|[var|nil]]] ---> d))";
58     br impI 1;
59     br allI 1;
60     by (res_inst_tac [("v1","v"),("vn'1","vn'"),("vn1","vn"),("M1","M"),
61     ("vlist1","vlist")] (envenv RS gen_lemma3) 1);
62     by (res_inst_tac [("v1","v"),("vn'1","vn'"),("vn1","vn"),("M1","M"),

```

```

63     ("vlist1","vlist")] (varexpr RS gen_lemma3) 1);
64 by (res_inst_tac [("rho1","<Atom v,vlist>"),("rho'1",
65 "[[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]]"),
66 ("M1","env"), ("M'1","env"), ("N1","expr"), ("N'1","var")]
67 (appdet RS gen_lemma3) 1);
68 auto();
69 qed "lemma1";
70
71 (*****
72 (* Derivation Lemmata *)
73 (*****
74
75 Goal "(<Atom w,[]> : lmixenv) --> \
76 \ (<Atom w,[]> |- bodyann ---> d') = (d' = Atom w)";
77 auto();
78 by (REPEAT ((Force_tac 2) ORELSE (CHANGED (deriv_forw 2))));
79 by (deriv_back 1);
80 qed "evaluation_varemp";
81
82 (*****
83 (*****
84
85 Goal "(<Atom v,[v';v;M,vlist]> : lmixenv) --> \
86 \ (<Atom v,[v';v;M,vlist]> |- bodyann ---> d') = (d' = (Atom v'))";
87 auto();
88 by (REPEAT ((Force_tac 2) ORELSE (CHANGED (deriv_forw 2))));
89 back();
90 back();
91 back();
92 back();
93 back();
94 back();
95 by (deriv_back 1);
96 qed "evaluation_varnonemp";
97
98 (*****
99 (*****
100
101 Goal "(<Atom v,[vn';vn;M,vlist]> : lmixenv & v ~ = vn) --> \
102 \ (<Atom v,[vn';vn;M,vlist]> |- bodyann ---> d) = \
103 \ (<Atom v,vlist> : lmixenv & (<Atom v,vlist> |- bodyann ---> d))";
104 br impI 1;
105 by (subgoal_tac "<Atom v,vlist> : lmixenv" 1);
106 auto();
107 by (eresolve_tac lmixenv.elims 3);
108 force 3;
109 by (SELECT_GOAL Auto_tac 3);
110 by (res_inst_tac [("y","vlist")] rlist.exhaust 3);
111 force 3;
112 force 3;
113 by (REPEAT (swap_res_tac l2eval.intrs 2));
114 by (REPEAT (Force_tac 2));
115 by (REPEAT (swap_res_tac l2eval.intrs 2));
116 by (REPEAT (Force_tac 2));
117 by (REPEAT (swap_res_tac l2eval.intrs 2));
118 by (REPEAT (Force_tac 2));
119 by (res_inst_tac [("v1","v"),("vlist1","vlist")]
120 (lemma1 RS gen_lemma3) 2);

```

```

121 by (rotate_tac ~1 2);
122 auto();
123 by (REPEAT (eresolve_tac l2elims 1));
124 auto();
125 by (eresolve_tac l2elims 1);
126 by (deriv_back 2);
127 by (res_inst_tac [("v1","v"),("vlist1","vlist")]
128       (lemma1 RS gen_lemma3) 1);
129 force 2;
130 by (subgoal_tac
131     "!d.([[var|Atom v] | [[value|Atom vn'] | <[Lam|[Atom vn|[M|nil]]],vlist>]] \
132 \ |- [App|[env|[var|nil]]] ---> d) = (<Atom v,vlist> \
133 \ |- [App|[env|[expr|nil]]] ---> d)" 1);
134 force 2;
135 by (rotate_tac ~1 1);
136 by (Asm_full_simp_tac 1);
137 qed "evaluation_varnonemp2";
138
139 (*****)
140 (*****)
141
142 Goal "(<[Quote|[d|nil]],vlist> : lmixenv) --> \
143 \ (<[Quote|[d|nil]],vlist> |- bodyann ---> d') = \
144 \ (d' = [Quote|[d|nil]])";
145 auto();
146 by (REPEAT ((Force_tac 2) ORELSE (CHANGED (deriv_forw 2))));
147 by (deriv_back 1);
148 qed "evaluation_quote";
149
150 (*****)
151 (*****)
152
153 Goal "(<[Lam|[Atom v|[M|nil]]],vlist> : lmixenv) --> \
154 \ ((<[Lam|[Atom v|[M|nil]]],vlist> |- bodyann ---> d') = \
155 \ (? dM v'. (<M,[v';v;M,vlist]> : lmixenv \
156 \ & (d' = [Lam|[Atom v'|[dM|nil]]]) & \
157 \ (<M,[v';v;M,vlist]> |- bodyann ---> dM))))";
158 auto();
159 by (eresolve_tac l2elims 1);
160 by (res_inst_tac [("x","dM")] exI 1);
161 by (res_inst_tac [("x","v'")] exI 1);
162 by (fold_tac l1expr.defs);
163 auto();
164 by (eresolve_tac lmixenv.elims 1);
165 force 1;
166 by (Asm_full_simp_tac 1);
167 by (REPEAT (etac conjE 1));
168 by (dtac sym 1);
169 by (Asm_full_simp_tac 1);
170 by (eresolve_tac l1elims 1);
171 auto();
172 by (deriv_back 1);
173 auto();
174 by (simp_tac (HOL_basic_ss addsimps [lwrap_def]) 1);
175 by (Simp_tac 1);
176
177 by (swap_res_tac l2eval.intrs 1);
178 by (eres_inst_tac [("a","<M,[v';v;M,vlist]>")] lmixenv.elim 1);

```

```

179 force 1;
180 force 1;
181 by (eres_inst_tac [("a","<M,[v';v;M,vlist]>")] lmixenv.elim 1);
182 force 1;
183
184 by (eresolve_tac lmrelims 1);
185 by (dtac sym 1);
186 by (asm_full_simp_tac (simpset() delsimps [vars_atom,vars_pair]) 1);
187 by (res_inst_tac [("a1","v"),("M2","[Lam|[Atom v|[M|nil]]]"),
188     ("vlist2","vlist")]
189     ((vars_lemma2 RS sub_lemma1) RS gen_lemma1) 1);
190 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
191 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
192 force 1;
193
194 by (deriv_forw 1);
195 back();
196 by (simp_tac (HOL_basic_ss addsimps [lenv_nemp RS sym]) 1);
197 by (fold_tac [lwrap_def]);
198 ba 1;
199 qed "evaluation_lam";
200
201 (*****)
202 (*****)
203
204 Goalw [lwrap_def] "<M,vlist> : lmixenv & <M',vlist> : lmixenv --> \
205 \ ([env|lookup (<M',vlist>,env)]|[ \
206 \ [expr|M|[[eval|[delay|[sintann|[nil|nil]]]]|nil]]] = <M,vlist>";
207 auto();
208 qed "envlem42";
209
210 Goalw [lwrap_def] "! q. (lookup (<M,vlist>,env) ~= [delay|q])";
211 auto();
212 qed "envlem43";
213
214 Addsimps [envlem42,envlem43];
215
216 (*****)
217 (*****)
218
219 Goal "<[App|[M|[N|nil]]],vlist> : lmixenv --> \
220 \ (<[App|[M|[N|nil]]],vlist> |- bodyann ----> d') = \
221 \ (? dM dN. (d' = [App|[dM|[dN|nil]]]) & \
222 \ <M,vlist> : lmixenv & <N,vlist> : lmixenv & \
223 \ (<M,vlist> |- bodyann ----> dM) & (<N,vlist> |- bodyann ----> dN))";
224 auto();
225 by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv" 1);
226 by (eresolve_tac l2elims 1);
227 by (res_inst_tac [("x","dM")] exI 1);
228 by (res_inst_tac [("x","dN")] exI 1);
229 auto();
230 by (eresolve_tac lmixenv.elims 3);
231 force 3;
232 by (dtac sym 3);
233 by (Asm_full_simp_tac 3);
234 by (REPEAT (etac conjE 3));
235 by (eresolve_tac l1elims 3);
236 auto();

```



```

237 by (eresolve_tac lmixenv.elims 3);
238 force 3;
239 by (dtac sym 3);
240 by (Asm_full_simp_tac 3);
241 by (REPEAT (etac conjE 3));
242 by (eresolve_tac l1elims 3);
243 auto();
244 by (thin_tac "?x |- ?y ---> dN" 1);
245 by (thin_tac "?x |- ?y ---> dM" 2);
246 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 3)
247   THEN (REPEAT1 (Force_tac 3))));
248 by (deriv_back 1);
249 qed "evaluation_app";
250
251 (*****)
252 (*****)
253
254 Goal "(<[Cons|[M|[N|nil]]],vlist> : lmixenv) --> \
255 \ ((<[Cons|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
256 \ (? dM dN. (d' = [Cons|[dM|[dN|nil]]]) & \
257 \ <M,vlist> : lmixenv & <N,vlist> : lmixenv & \
258 \ (<M,vlist> |- bodyann ---> dM) & (<N,vlist> |- bodyann ---> dN)))";
259 auto();
260 by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv" 1);
261 by (eresolve_tac l2elims 1);
262 by (res_inst_tac [("x","dM")] exI 1);
263 by (res_inst_tac [("x","dN")] exI 1);
264 auto();
265 by (eresolve_tac lmixenv.elims 3);
266 force 3;
267 by (dtac sym 3);
268 by (Asm_full_simp_tac 3);
269 by (REPEAT (etac conjE 3));
270 by (eresolve_tac l1elims 3);
271 auto();
272 by (eresolve_tac lmixenv.elims 3);
273 force 3;
274 by (dtac sym 3);
275 by (Asm_full_simp_tac 3);
276 by (REPEAT (etac conjE 3));
277 by (eresolve_tac l1elims 3);
278 auto();
279 by (thin_tac "?x |- ?y ---> dN" 1);
280 by (thin_tac "?x |- ?y ---> dM" 2);
281 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 3)
282   THEN (REPEAT1 (Force_tac 3))));
283 by (deriv_back 1);
284 qed "evaluation_cons";
285
286 (*****)
287 (*****)
288
289 Goal "(<[IsEq|[M|[N|nil]]],vlist> : lmixenv) --> \
290 \ ((<[IsEq|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
291 \ (? dM dN. (d' = [IsEq|[dM|[dN|nil]]]) & \
292 \ <M,vlist> : lmixenv & <N,vlist> : lmixenv & \
293 \ (<M,vlist> |- bodyann ---> dM) & (<N,vlist> |- bodyann ---> dN)))";
294 auto();

```

```

295 by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv" 1);
296 by (eresolve_tac l2elims 1);
297 by (res_inst_tac [("x","dM")] exI 1);
298 by (res_inst_tac [("x","dN")] exI 1);
299 auto();
300 by (eresolve_tac lmixenv.elims 3);
301 force 3;
302 by (dtac sym 3);
303 by (Asm_full_simp_tac 3);
304 by (REPEAT (etac conjE 3));
305 by (eresolve_tac l1elims 3);
306 auto();
307 by (eresolve_tac lmixenv.elims 3);
308 force 3;
309 by (dtac sym 3);
310 by (Asm_full_simp_tac 3);
311 by (REPEAT (etac conjE 3));
312 by (eresolve_tac l1elims 3);
313 auto();
314 by (thin_tac "?x |- ?y ---> dN" 1);
315 by (thin_tac "?x |- ?y ---> dM" 2);
316 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 3))
317   THEN (REPEAT1 (Force_tac 3))));
318 by (deriv_back 1);
319 qed "evaluation_iseq";
320
321 (*****)
322 (*****)
323
324 Goal "(<[Fix|[M|nil]],vlist> : lmixenv) --> \
325 \ ((<[Fix|[M|nil]],vlist> |- bodyann ---> d') = \
326 \ (? dM. (d' = [Fix|[dM|nil]]) & \
327 \ <M,vlist> : lmixenv & \
328 \ (<M,vlist> |- bodyann ---> dM)))";
329 auto();
330 by (subgoal_tac "<M,vlist> : lmixenv" 1);
331 by (eresolve_tac l2elims 1);
332 by (res_inst_tac [("x","dM")] exI 1);
333 auto();
334 by (eresolve_tac lmixenv.elims 2);
335 force 2;
336 by (dtac sym 2);
337 by (Asm_full_simp_tac 2);
338 by (REPEAT (etac conjE 2));
339 by (eresolve_tac l1elims 2);
340 auto();
341 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
342   THEN (REPEAT1 (Force_tac 2))));
343 by (deriv_back 1);
344 qed "evaluation_fix";
345
346 (*****)
347 (*****)
348
349 Goal "(<[Car|[M|nil]],vlist> : lmixenv) --> \
350 \ ((<[Car|[M|nil]],vlist> |- bodyann ---> d') = \
351 \ (? dM. (d' = [Car|[dM|nil]]) & \
352 \ <M,vlist> : lmixenv & \

```

```

353 \ (<M,vlist> |- bodyann ---> dM))";
354 auto();
355 by (subgoal_tac "<M,vlist> : lmixenv" 1);
356 by (eresolve_tac l2elims 1);
357 by (res_inst_tac [("x","dM")] exI 1);
358 auto();
359 by (eresolve_tac lmixenv.elims 2);
360 force 2;
361 by (dtac sym 2);
362 by (Asm_full_simp_tac 2);
363 by (REPEAT (etac conjE 2));
364 by (eresolve_tac l1elims 2);
365 auto();
366 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
367             THEN (REPEAT1 (Force_tac 2))));
368 by (deriv_back 1);
369 qed "evaluation_car";
370
371 (*****)
372 (*****)
373
374 Goal "(<[Cdr|[M|nil]],vlist> : lmixenv) --> \
375 \ (<[Cdr|[M|nil]],vlist> |- bodyann ---> d') = \
376 \ (? dM. (d' = [Cdr|[dM|nil]]) & \
377 \ <M,vlist> : lmixenv & \
378 \ (<M,vlist> |- bodyann ---> dM))";
379 auto();
380 by (subgoal_tac "<M,vlist> : lmixenv" 1);
381 by (eresolve_tac l2elims 1);
382 by (res_inst_tac [("x","dM")] exI 1);
383 auto();
384 by (eresolve_tac lmixenv.elims 2);
385 force 2;
386 by (dtac sym 2);
387 by (Asm_full_simp_tac 2);
388 by (REPEAT (etac conjE 2));
389 by (eresolve_tac l1elims 2);
390 auto();
391 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
392             THEN (REPEAT1 (Force_tac 2))));
393 by (deriv_back 1);
394 qed "evaluation_cdr";
395
396 (*****)
397 (*****)
398
399 Goal "(<[IsAtom|[M|nil]],vlist> : lmixenv) --> \
400 \ (<[IsAtom|[M|nil]],vlist> |- bodyann ---> d') = \
401 \ (? dM. (d' = [IsAtom|[dM|nil]]) & \
402 \ <M,vlist> : lmixenv & \
403 \ (<M,vlist> |- bodyann ---> dM))";
404 auto();
405 by (subgoal_tac "<M,vlist> : lmixenv" 1);
406 by (eresolve_tac l2elims 1);
407 by (res_inst_tac [("x","dM")] exI 1);
408 auto();
409 by (eresolve_tac lmixenv.elims 2);
410 force 2;

```

```

411 by (dtac sym 2);
412 by (Asm_full_simp_tac 2);
413 by (REPEAT (etac conjE 2));
414 by (eresolve_tac l1elims 2);
415 auto();
416 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2)
417   THEN (REPEAT1 (Force_tac 2))));
418 by (deriv_back 1);
419 qed "evaluation_isatom";
420
421 (*****)
422 (*****)
423
424 Goal "(<[Error|[M|nil]],vlist> : lmixenv) --> \
425 \ ((<[Error|[M|nil]],vlist> |- bodyann ---> d') = \
426 \ (? dM. (d' = [Error|[dM|nil]]) & \
427 \ <M,vlist> : lmixenv & \
428 \ (<M,vlist> |- bodyann ---> dM)))";
429 auto();
430 by (subgoal_tac "<M,vlist> : lmixenv" 1);
431 by (eresolve_tac l2elims 1);
432 by (res_inst_tac [("x","dM")] exI 1);
433 auto();
434 by (eresolve_tac lmixenv.elims 2);
435 force 2;
436 by (dtac sym 2);
437 by (Asm_full_simp_tac 2);
438 by (REPEAT (etac conjE 2));
439 by (eresolve_tac l1elims 2);
440 auto();
441 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2)
442   THEN (REPEAT1 (Force_tac 2))));
443 by (deriv_back 1);
444 qed "evaluation_error";
445
446 (*****)
447 (*****)
448
449 Goal "(<[Cond|[M|[N|[P|nil]]]],vlist> : lmixenv) --> \
450 \ ((<[Cond|[M|[N|[P|nil]]]],vlist> |- bodyann ---> d') = \
451 \ (? dM dN dP. (d' = [Cond|[dM|[dN|[dP|nil]]]]) & \
452 \ <M,vlist> : lmixenv & <N,vlist> : lmixenv & <P,vlist> : lmixenv & \
453 \ (<M,vlist> |- bodyann ---> dM) & (<N,vlist> |- bodyann ---> dN) & \
454 \ (<P,vlist> |- bodyann ---> dP)))";
455 auto();
456 by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv & \
457 \ <P,vlist> : lmixenv" 1);
458 by (eresolve_tac l2elims 1);
459 by (res_inst_tac [("x","dM")] exI 1);
460 by (res_inst_tac [("x","dN")] exI 1);
461 by (res_inst_tac [("x","dP")] exI 1);
462 auto();
463 by (eresolve_tac lmixenv.elims 4);
464 force 4;
465 by (dtac sym 4);
466 by (Asm_full_simp_tac 4);
467 by (REPEAT (etac conjE 4));
468 by (eresolve_tac l1elims 4);

```

```

469 auto();
470 by (eresolve_tac lmixenv.elims 4);
471 force 4;
472 by (dtac sym 4);
473 by (Asm_full_simp_tac 4);
474 by (REPEAT (etac conjE 4));
475 by (eresolve_tac l1elims 4);
476 auto();
477 by (eresolve_tac lmixenv.elims 4);
478 force 4;
479 by (dtac sym 4);
480 by (Asm_full_simp_tac 4);
481 by (REPEAT (etac conjE 4));
482 by (eresolve_tac l1elims 4);
483 auto();
484 by (thin_tac "?x |- ?y ---> dN" 1);
485 by (thin_tac "?x |- ?y ---> dP" 1);
486 by (thin_tac "?x |- ?y ---> dM" 2);
487 by (thin_tac "?x |- ?y ---> dP" 2);
488 by (thin_tac "?x |- ?y ---> dM" 3);
489 by (thin_tac "?x |- ?y ---> dN" 3);
490 by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 4))
491   THEN (REPEAT1 (Force_tac 4))));
492 by (deriv_back 1);
493 qed "evaluation_if";
494
495 val evaluation_lemmata = [evaluation_varemp,evaluation_varnonemp,evaluation_varnonemp2,
496   evaluation_quote,evaluation_lam,evaluation_app,evaluation_fix,evaluation_if,evaluation_co
497   evaluation_car,evaluation_cdr,evaluation_iseq,evaluation_isatom,evaluation_error];
498
499 Addsimps evaluation_lemmata;
500 Delsimps dispatch_lemmata;

1  (*****
2  *
3  * Determ.ML
4  *
5  *****)
6
7  AddSEs lmixenv.elims;
8
9  Goal "(<[Fix|[M|nil]],vlist> : lmixenv & \
10 \ <[Fix|[M|nil]],vlist'> : lmixenv) --> \
11 \ ((!dM. (<M,vlist> |- bodyann ---> dM) --> \
12 \ (<M,vlist'> |- bodyann ---> dM)) --> \
13 \ ((<[Fix|[M|nil]],vlist> |- bodyann ---> d) --> \
14 \ (<[Fix|[M|nil]],vlist'> |- bodyann ---> d))";
15 auto();
16 qed "determ_fix";
17
18 (*****)
19 (*****)
20
21 Goal "(<[Car|[M|nil]],vlist> : lmixenv & \
22 \ <[Car|[M|nil]],vlist'> : lmixenv) --> \
23 \ ((!dM. (<M,vlist> |- bodyann ---> dM) --> \
24 \ (<M,vlist'> |- bodyann ---> dM)) --> \
25 \ ((<[Car|[M|nil]],vlist> |- bodyann ---> d) --> \
26 \ (<[Car|[M|nil]],vlist'> |- bodyann ---> d))";

```

```

27 auto();
28 qed "determ_car";
29
30 (*****)
31 (*****)
32
33 Goal "(<[Cdr|[M|nil]],vlist> : lmixenv & \
34 \ <[Cdr|[M|nil]],vlist'> : lmixenv) --> \
35 \ ((!dM. (<M,vlist> |- bodyann ----> dM) --> \
36 \ (<M,vlist'> |- bodyann ----> dM)) --> \
37 \ ((<[Cdr|[M|nil]],vlist> |- bodyann ----> d) --> \
38 \ (<[Cdr|[M|nil]],vlist'> |- bodyann ----> d)))";
39 auto();
40 qed "determ_cdr";
41
42 (*****)
43 (*****)
44
45 Goal "(<[IsAtom|[M|nil]],vlist> : lmixenv & \
46 \ <[IsAtom|[M|nil]],vlist'> : lmixenv) --> \
47 \ ((!dM. (<M,vlist> |- bodyann ----> dM) --> \
48 \ (<M,vlist'> |- bodyann ----> dM)) --> \
49 \ ((<[IsAtom|[M|nil]],vlist> |- bodyann ----> d) --> \
50 \ (<[IsAtom|[M|nil]],vlist'> |- bodyann ----> d)))";
51 auto();
52 qed "determ_isatom";
53
54 (*****)
55 (*****)
56
57 Goal "(<[Error|[M|nil]],vlist> : lmixenv & \
58 \ <[Error|[M|nil]],vlist'> : lmixenv) --> \
59 \ ((!dM. (<M,vlist> |- bodyann ----> dM) --> \
60 \ (<M,vlist'> |- bodyann ----> dM)) --> \
61 \ ((<[Error|[M|nil]],vlist> |- bodyann ----> d) --> \
62 \ (<[Error|[M|nil]],vlist'> |- bodyann ----> d)))";
63 auto();
64 qed "determ_error";
65
66 Delrules lmixenv.elims;
67
68 (*****)
69 (*****)
70
71 Goal "(<[App|[M|[N|nil]]],vlist> : lmixenv & \
72 \ <[App|[M|[N|nil]]],vlist'> : lmixenv) --> \
73 \ (((!dM. (<M,vlist> |- bodyann ----> dM) --> \
74 \ (<M,vlist'> |- bodyann ----> dM)) & \
75 \ (!dN. (<N,vlist> |- bodyann ----> dN) --> \
76 \ (<N,vlist'> |- bodyann ----> dN))) --> \
77 \ ((<[App|[M|[N|nil]]],vlist> |- bodyann ----> d) --> \
78 \ (<[App|[M|[N|nil]]],vlist'> |- bodyann ----> d)))";
79 br impI 1;
80 be conjE 1;
81 by (subgoal_tac "M : l1expr & N : l1expr" 1);
82 by (eresolve_tac lmixenv.elims 2);
83 auto();
84 by (res_inst_tac [("M'1","[App|[M|[N|nil]]]"]) (env_lemma6 RS mp) 1);

```

```

85 by (res_inst_tac [("M'1","[App[M][N|nil]]"]) (env_lemma6 RS mp) 2);
86 auto();
87 qed "determ_app";
88
89 (*****)
90 (*****)
91
92 Goal "(<[Cons[M][N|nil]],vlist> : lmixenv & \
93 \ <[Cons[M][N|nil]],vlist'> : lmixenv) --> \
94 \ (((!dM. (<M,vlist> |- bodyann ----> dM) --> \
95 \ (<M,vlist'> |- bodyann ----> dM)) & \
96 \ (!dN. (<N,vlist> |- bodyann ----> dN) --> \
97 \ (<N,vlist'> |- bodyann ----> dN))) --> \
98 \ (((<[Cons[M][N|nil]],vlist> |- bodyann ----> d) --> \
99 \ (<[Cons[M][N|nil]],vlist'> |- bodyann ----> d))))";
100 br impI 1;
101 be conjE 1;
102 by (subgoal_tac "M : l1expr & N : l1expr" 1);
103 by (eresolve_tac lmixenv.elims 2);
104 auto();
105 by (res_inst_tac [("M'1","[Cons[M][N|nil]]"]) (env_lemma6 RS mp) 1);
106 by (res_inst_tac [("M'1","[Cons[M][N|nil]]"]) (env_lemma6 RS mp) 2);
107 auto();
108 qed "determ_cons";
109
110 (*****)
111 (*****)
112
113 Goal "(<[IsEq[M][N|nil]],vlist> : lmixenv & \
114 \ <[IsEq[M][N|nil]],vlist'> : lmixenv) --> \
115 \ (((!dM. (<M,vlist> |- bodyann ----> dM) --> \
116 \ (<M,vlist'> |- bodyann ----> dM)) & \
117 \ (!dN. (<N,vlist> |- bodyann ----> dN) --> \
118 \ (<N,vlist'> |- bodyann ----> dN))) --> \
119 \ ((([IsEq[M][N|nil]],vlist> |- bodyann ----> d) --> \
120 \ (<[IsEq[M][N|nil]],vlist'> |- bodyann ----> d))))";
121 br impI 1;
122 be conjE 1;
123 by (subgoal_tac "M : l1expr & N : l1expr" 1);
124 by (eresolve_tac lmixenv.elims 2);
125 auto();
126 by (res_inst_tac [("M'1","[IsEq[M][N|nil]]"]) (env_lemma6 RS mp) 1);
127 by (res_inst_tac [("M'1","[IsEq[M][N|nil]]"]) (env_lemma6 RS mp) 2);
128 auto();
129 qed "determ_iseq";
130
131 Goal "(<[Cond[M][N][P|nil]],vlist> : lmixenv & \
132 \ <[Cond[M][N][P|nil]],vlist'> : lmixenv) --> \
133 \ (((!dM. (<M,vlist> |- bodyann ----> dM) --> \
134 \ (<M,vlist'> |- bodyann ----> dM)) & \
135 \ (!dN. (<N,vlist> |- bodyann ----> dN) --> \
136 \ (<N,vlist'> |- bodyann ----> dN)) & \
137 \ (!dP. (<P,vlist> |- bodyann ----> dP) --> \
138 \ (<P,vlist'> |- bodyann ----> dP))) --> \
139 \ ((([Cond[M][N][P|nil]],vlist> |- bodyann ----> d) --> \
140 \ (<[Cond[M][N][P|nil]],vlist'> |- bodyann ----> d))))";
141 br impI 1;
142 be conjE 1;

```

```

143 by (subgoal_tac "M : llexpr & N : llexpr & P : llexpr" 1);
144 by (eresolve_tac lmixenv.elims 2);
145 auto();
146 by (res_inst_tac [("M'1","[Cond|[M|[N|[P|nil]]]]")])
147   (env_lemma6 RS mp) 1);
148 by (res_inst_tac [("M'1","[Cond|[M|[N|[P|nil]]]]")])
149   (env_lemma6 RS mp) 2);
150 by (res_inst_tac [("M'1","[Cond|[M|[N|[P|nil]]]]")])
151   (env_lemma6 RS mp) 3);
152 auto();
153 qed "determ_if";
154
155 Goal "(<[Quote|[d|nil]],vlist> : lmixenv & \
156 \ <[Quote|[d|nil]],vlist'> : lmixenv) --> \
157 \ ((<[Quote|[d|nil]],vlist> |- bodyann ----> d') --> \
158 \ (<[Quote|[d|nil]],vlist'> |- bodyann ----> d'))";
159 auto();
160 qed "determ_quote";
161
162 Goal "(<[Lam|[Atom v|[M|nil]]],vlist> : lmixenv & \
163 \ <[Lam|[Atom v|[M|nil]]],vlist'> : lmixenv & \
164 \ vars <[Lam|[Atom v|[M|nil]]],vlist'> \
165 \ <= vars <[Lam|[Atom v|[M|nil]]],vlist> --> \
166 \ ((!vlist''. (<M,vlist'' # vlist> : lmixenv) --> \
167 \ (!dM. (<M,vlist'' # vlist> |- bodyann ----> dM) --> \
168 \ (<M,vlist'' # vlist'> |- bodyann ----> dM))) --> \
169 \ ((<[Lam|[Atom v|[M|nil]]],vlist> |- bodyann ----> d) --> \
170 \ (<[Lam|[Atom v|[M|nil]]],vlist'> |- bodyann ----> d)))";
171 auto();
172 by (eres_inst_tac [("x","[v';v;M,[]]")] allE 1);
173 by (Asm_full_simp_tac 1);
174 by (eres_inst_tac [("x","[v';v;M,[]]")] allE 2);
175 by (Asm_full_simp_tac 2);
176
177 by (swap_res_tac lmixenv.intrs 1);
178 by (eresolve_tac lmixenv.elims 1);
179 force 1;
180 force 1;
181 force 2;
182
183 by (eres_inst_tac [("a","<M,[v';v;M,vlist]>")] lmixenv.elim 1);
184 force 1;
185 by (dtac sym 1);
186 auto();
187 by (swap_res_tac lmixrlist.intrs 1);
188 force 2;
189 force 2;
190 force 2;
191
192 AddSEs lmixenv.elims;
193 auto();
194 Delrules lmixenv.elims;
195 force 1;
196 force 1;
197 qed "determ_lam";
198
199 Goal "(<Atom v,[v';v;M,vlist]> : lmixenv & \
200 \ <Atom v,[v';v;M,vlist']> : lmixenv) --> \

```



```

201 \ ((<Atom v,[v';v;M,vlist]> |- bodyann ---> d') --> \
202 \ (<Atom v,[v';v;M,vlist']> |- bodyann ---> d'))";
203 auto();
204 qed "determ_varnonemp";
205
206 Goal "(<Atom y,[v';v;M,vlist]> : lmixenv & y ~ = v & \
207 \ (<Atom y,vlist> |- bodyann ---> d')) --> \
208 \ (<Atom y,[v';v;M,vlist]> |- bodyann ---> d')";
209 auto();
210 br (lmered RS mp) 1;
211 auto();
212 qed "determ_varnonemp2";
213
214 val [prem1,prem2] =
215 Goal "[|y ~ = v & <Atom y,[v';v;M,vlist]> : lmixenv & \
216 \ (<Atom y,[v';v;M,vlist]> |- bodyann ---> d); \
217 \ [|<Atom y,vlist> : lmixenv; <Atom y,vlist> |- bodyann ---> d|] \
218 \ ==> P|] ==> P";
219 br prem2 1;
220 by (subgoal_tac "y ~ = v & <Atom y,[v';v;M,vlist]> : lmixenv & \
221 \ (<Atom y,[v';v;M,vlist]> |- bodyann ---> d)" 1);
222 br prem1 2;
223 br (lmered RS mp) 1;
224 force 1;
225 by (subgoal_tac "y ~ = v & <Atom y,[v';v;M,vlist]> : lmixenv & \
226 \ (<Atom y,[v';v;M,vlist]> |- bodyann ---> d)" 1);
227 br prem1 2;
228 force 1;
229 qed "derivelimvar1";

1  (*****
2  *
3  * Dispatch.ML
4  *
5  *****)
6
7  Goal "(<[C|q],vlist> : lmixenv) --> \
8  \ (<[C|q],vlist> |- [IsEq [[Car|expr|nil]] [[Quote|C|nil]]|nil]] \
9  \ ---> d) = (d = #t)";
10 auto();
11 by (deriv_forw 1);
12 qed "iseqtrue";
13
14 Goal "(C ~ = C' & <[C|q],vlist> : lmixenv) --> \
15 \ (<[C|q],vlist> |- [IsEq [[Car|expr|nil]] [[Quote|C'|nil]]|nil]] \
16 \ ---> d) = (d = nil)";
17 auto();
18 by (deriv_forw 1);
19 qed "iseqfalse";
20
21 Addsimps [iseqtrue,iseqfalse];
22
23 Goal "(<Atom w,vlist> : lmixenv) --> \
24 \ (<Atom w,vlist> |- [IsAtom|expr|nil]] ---> d) = (d = #t)";
25 auto();
26 qed "isatomtrue";
27
28 Goal "(<[C|q],vlist> : lmixenv) --> \
29 \ (<[C|q],vlist> |- [IsAtom|expr|nil]] ---> d) = (d = nil)";

```

```

30 auto();
31 qed "isatomfalse";
32
33 Addsimps [isatomtrue,isatomfalse];
34
35 Goal "<Atom w,vlist> : lmixenv --> \
36 \ lookup (<Atom w,vlist>,expr) = Atom w";
37 auto();
38 qed "lookup_atom";
39 Addsimps [lookup_atom];
40
41 Delrules l2elims;
42 Delrules l2eval.intrs;
43 Delsimps l2eval.intrs;
44 Delrules lmixenv.elims;
45
46 (*****
47 (* Dispatch Lemmata *)
48 (*****
49 Goalw [bodyann_def] "<Atom w,vlist> : lmixenv) --> \
50 \ (<Atom w,vlist> |- bodyann ---> d') = \
51 \ (<Atom w,vlist> |- [App|[env|[expr|nil]]] ---> d')";
52 auto();
53 by (SELECT_GOAL (auto_tac (claset() addSIs l2eval.intrs, simpset())) 2);
54 by (deriv_back 1);
55 qed "dispatch_var";
56
57 Goalw [bodyann_def] "<[Quote|[d|nil]],vlist> : lmixenv) --> \
58 \ (<[Quote|[d|nil]],vlist> |- bodyann ---> d') = \
59 \ (<[Quote|[d|nil]],vlist> |- [Lift|[Cadr|[expr|nil]]|nil]] ---> d')";
60 auto();
61 by (deriv_forw 2);
62 back();
63 by (deriv_back 1);
64 qed "dispatch_quote";
65
66 Goalw [bodyann_def] "<[Lam|[Atom v|[M|nil]]],vlist> : lmixenv) --> \
67 \ (<[Lam|[Atom v|[M|nil]]],vlist> |- bodyann ---> d') = \
68 \ (<[Lam|[Atom v|[M|nil]]],vlist> |- [RLam| \
69 \ [value|[App|[App|[eval|[Caddr|[expr|nil]]|nil]]|[envlam|nil]]] | \
70 \ nil]]] ---> d')";
71 auto();
72 by (deriv_forw 2);
73 back();
74 by (deriv_back 1);
75 qed "dispatch_lam";
76
77 Goalw [bodyann_def] "<[App|[M|[N|nil]]],vlist> : lmixenv) --> \
78 \ (<[App|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
79 \ (<[App|[M|[N|nil]]],vlist> |- [RApp| \
80 \ [[App|[App|[eval|[Cadr|[expr|nil]]|nil]]|[env|nil]]] | \
81 \ [[App|[App|[eval|[Caddr|[expr|nil]]|nil]]|[env|nil]]|nil]] \
82 \ ---> d')";
83 auto();
84 by (deriv_forw 2);
85 back();
86 by (deriv_back 1);
87 qed "dispatch_app";

```

```

88
89 Goalw [bodyann_def] "(<[Fix|[M|nil]],vlist> : lmixenv) --> \
90 \ (<[Fix|[M|nil]],vlist> |- bodyann ----> d') = \
91 \ (<[Fix|[M|nil]],vlist> |- [RFix| \
92 \ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]|env|nil]]|nil]] \
93 \ ----> d')";
94 auto();
95 by (deriv_forw 2);
96 back();
97 by (deriv_back 1);
98 qed "dispatch_fix";
99
100 Goalw [bodyann_def] "(<[Cond|[M|[N|[P|nil]]]],vlist> : lmixenv) --> \
101 \ (<[Cond|[M|[N|[P|nil]]]],vlist> |- bodyann ----> d') = \
102 \ (<[Cond|[M|[N|[P|nil]]]],vlist> |- [RCond| \
103 \ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]|env|nil]]| \
104 \ [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]|env|nil]]| \
105 \ [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]|env|nil]]|nil]]] \
106 \ ----> d')";
107 auto();
108 by (deriv_forw 2);
109 back();
110 by (deriv_back 1);
111 qed "dispatch_if";
112
113 Goalw [bodyann_def] "(<[Cons|[M|[N|nil]]],vlist> : lmixenv) --> \
114 \ (<[Cons|[M|[N|nil]]],vlist> |- bodyann ----> d') = \
115 \ (<[Cons|[M|[N|nil]]],vlist> |- [RCons| \
116 \ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]|env|nil]]| \
117 \ [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]|env|nil]]|nil]]] \
118 \ ----> d')";
119 auto();
120 by (deriv_forw 2);
121 back();
122 by (deriv_back 1);
123 qed "dispatch_cons";
124
125 Goalw [bodyann_def] "(<[Car|[M|nil]],vlist> : lmixenv) --> \
126 \ (<[Car|[M|nil]],vlist> |- bodyann ----> d') = \
127 \ (<[Car|[M|nil]],vlist> |- [RCar| \
128 \ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]|env|nil]]|nil]] \
129 \ ----> d')";
130 auto();
131 by (deriv_forw 2);
132 back();
133 by (deriv_back 1);
134 qed "dispatch_car";
135
136 Goalw [bodyann_def] "(<[Cdr|[M|nil]],vlist> : lmixenv) --> \
137 \ (<[Cdr|[M|nil]],vlist> |- bodyann ----> d') = \
138 \ (<[Cdr|[M|nil]],vlist> |- [RCdr| \
139 \ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]|env|nil]]|nil]] \
140 \ ----> d')";
141 auto();
142 by (deriv_forw 2);
143 back();
144 by (deriv_back 1);
145 qed "dispatch_cdr";

```

```

146
147 Goalw [bodyann_def] "(<[IsEq|[M|[N|nil]]],vlist> : lmixenv) --> \
148 \ (<[IsEq|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
149 \ (<[IsEq|[M|[N|nil]]],vlist> |- [RIsEq| \
150 \ [[App|[[App|eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]| \
151 \ [[App|[[App|eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]|nil]] \
152 \ ---> d')";
153 auto();
154 by (deriv_forw 2);
155 back();
156 by (deriv_back 1);
157 qed "dispatch_iseq";
158
159 Goalw [bodyann_def] "(<[IsAtom|[M|nil]],vlist> : lmixenv) --> \
160 \ (<[IsAtom|[M|nil]],vlist> |- bodyann ---> d') = \
161 \ (<[IsAtom|[M|nil]],vlist> |- [RIsAtom| \
162 \ [[App|[[App|eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]|nil]] \
163 \ ---> d')";
164 auto();
165 by (deriv_forw 2);
166 back();
167 by (deriv_back 1);
168 qed "dispatch_isatom";
169
170 Goalw [bodyann_def] "(<[Error|[M|nil]],vlist> : lmixenv) --> \
171 \ (<[Error|[M|nil]],vlist> |- bodyann ---> d') = \
172 \ (<[Error|[M|nil]],vlist> |- [RError| \
173 \ [[App|[[App|eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]|nil]] \
174 \ ---> d')";
175 auto();
176 by (deriv_forw 2);
177 back();
178 by (deriv_back 1);
179 qed "dispatch_error";
180
181 val dispatch_lemmata = [dispatch_var,dispatch_quote,dispatch_lam,
182     dispatch_app,dispatch_fix,dispatch_if,dispatch_cons,dispatch_car,
183     dispatch_cdr,dispatch_iseq,dispatch_isatom,dispatch_error];
184
185 Addsimps dispatch_lemmata;

1  (*****
2  *
3  * EnvLem.ML
4  *
5  *****)
6
7  Goal "vlist : lmixrlist --> y' : var --> \
8  \ (vars (lenv (addvar y' vlist)) <= \
9  \ vars (lenv vlist) Un {y'})";
10 br impI 1;
11 by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
12 force 1;
13 force 1;
14 br impI 1;
15 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
16 force 1;
17 qed "lenv_lemma_add";
18

```

```

19 Goal "vlist : lmixrlist --> [y';y;My,vlist] : lmixrlist --> \
20 \ (addvar y' vlist : lmixrlist)";
21 br impI 1;
22 by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
23 force 1;
24 force 1;
25 br impI 1;
26 by (eres_inst_tac [("a","[y';y;My,[v';v;M,vlista]]") lmixrlist.elim 1);
27 force 1;
28 by (dtac sym 1);
29 by (Asm_full_simp_tac 1);
30 by (subgoal_tac "[y';y;My,vlista] : lmixrlist" 1);
31 by (swap_res_tac lmixrlist.intrs 2);
32 force 2;
33 force 2;
34 force 2;
35 force 2;
36 force 3;
37 by (asm_full_simp_tac (simpset() addsimps
38 [lwrap_def,bodyann_def,sintann_def]) 2);
39 by (mp_tac 1);
40 by (swap_res_tac lmixrlist.intrs 1);
41 force 1;
42 force 1;
43 force 1;
44 force 1;
45 by (asm_full_simp_tac (simpset() addsimps
46 [lwrap_def,bodyann_def,sintann_def]) 1);
47 by (REPEAT (etac conjE 1));
48 br conjI 1;
49 force 1;
50 br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 1;
51 force 1;
52 force 1;
53 force 1;
54 by (res_inst_tac [("y","vlista")] rlist.exhaust 1);
55 force 1;
56 by (Asm_full_simp_tac 1);
57 by (REPEAT (etac conjE 1));
58 force 1;
59 qed "add_lemma1";
60
61 Goal "<M,[v';v;Mv,vl]> : lmixenv --> <M,vl> : lmixenv";
62 br impI 1;
63 by (eresolve_tac lmixenv.elims 1);
64 by (res_inst_tac [("y","vl")] rlist.exhaust 2);
65 auto();
66 force 1;
67 qed "lmered";
68
69 Goal "((vlist # vlist') : lmixrlist) --> \
70 \ (vlist : lmixrlist & vlist' : lmixrlist)";
71 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
72 auto();
73 by (subgoal_tac
74 "atom1 ~: vars <[Lam|[Atom atom2|[sexpr|nil]]],rlist>" 1);
75 br (vars_lemma4 RS sub_lemma1) 2;
76 force 2;

```

```

77 by (subgoal_tac '!y' y My vlisty. rlist = [y';y;My,vlisty] --> \
78 \ vars [Lam|[Atom atom2|[sexpr|nil]]] <= vars My" 1);
79 force 1;
80 auto();
81 qed "lconcat_lemma1";
82
83 Goal "(<Atom v,vlist> : lmixenv & v ~: varsin vlist) --> \
84 \ ((<Atom v,vlist> |- bodyann ---> d) = (d = (Atom v)))";
85 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
86 force 1;
87 br impI 1;
88 be conjE 1;
89 by (subgoal_tac "<Atom v,rlist> : lmixenv" 1);
90 by (res_inst_tac [("v'1","atom1"),("v1","atom2"),("Mv1","sexpr")]
91 (lmered RS mp) 2);
92 force 1;
93 auto();
94 qed "lemma_511";
95
96 Goal "(<M,vlist> : lmixenv & y ~: vars <M,vlist>) --> \
97 \ (y ~: vars M Un varsin vlist)";
98 br impI 1;
99 be conjE 1;
100 by (eresolve_tac lmixenv.elims 1);
101 by (Asm_full_simp_tac 1);
102 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
103 by (dtac sym 1);
104 by (Asm_full_simp_tac 1);
105 by (subgoal_tac "y ~: varsin vlist" 1);
106 by (res_inst_tac [("vlist2","vlist"),("a","y"),("M2","M")]
107 ((vars_lemma1 RS mp) RS sub_lemma1) 2);
108 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
109 auto();
110 qed "env_lemma1";
111
112 Addsimps [env_lemma1];
113
114 Goal "(vlist'' : lmixrlist) --> \
115 \ ((vlist'' # [y';y;My,[z';z;Mz,vlist'']]) : lmixrlist) --> \
116 \ ((vlist'' # [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist'']]) \
117 \ : lmixrlist)";
118 br impI 1;
119 by (res_inst_tac [("xa","vlist'']") lmixrlist.induct 1);
120 force 1;
121 by (SELECT_GOAL Auto_tac 1);
122 by (res_inst_tac [("y","vlist'']") rlist.exhaust 1);
123 by (Asm_full_simp_tac 1);
124 by (subgoal_tac "[y';y;[App|[Mz|[Atom z|nil]]],[[]] : lmixrlist" 1);
125 by (subgoal_tac "y' ~: \
126 \ vars <[Lam|[Atom y|[App|[Mz|[Atom z|nil]]]|nil]]],[[]]>" 2);
127 force 2;
128 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
129 by (subgoal_tac "vars [Lam|[Atom z|[Mz|nil]]] \
130 \ <= vars [App|[Mz|[Atom z|nil]]]" 1);
131 force 2;
132 by (subgoal_tac "z' ~: vars <[Lam|[Atom z|[Mz|nil]]],[y';y; \
133 \ [App|[Mz|[Atom z|nil]]],[[]]>" 1);
134 by (swap_res_tac lmixrlist.intrs 1);

```

```

135 by (REPEAT (Force_tac 1));
136 by ((asm_full_simp_tac (simpset() addsimps
137   [lwrap_def,sintann_def,bodyann_def]) 1) THEN (Force_tac 1));
138 by (Asm_full_simp_tac 1);
139 by (subgoal_tac
140 "[y';y;[App|[Mz|[Atom z|nil]]],[atom1;atom2;sexpr,rlist]] : lmixrlist" 1);
141 by (subgoal_tac "y' ~: \
142 \ vars <[Lam|[Atom y|[[App|[Mz|[Atom z|nil]]]|nil]]],[atom1; \
143 \ atom2;sexpr,rlist]>" 2);
144 by (subgoal_tac "vars [Lam|[Atom y|[[App|[Mz|[Atom z|nil]]]|nil]] \
145 \ <= vars sexpr" 2);
146 force 2;
147 force 2;
148 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
149 by (swap_res_tac lmixrlist.intrs 1);
150 force 1;
151 force 1;
152 force 1;
153 force 1;
154 by (subgoal_tac "vars [Lam|[Atom z|[Mz|nil]]] \
155 \ <= vars [App|[Mz|[Atom z|nil]]]" 1);
156 force 2;
157 by (subgoal_tac "z' ~: vars <[Lam|[Atom z|[Mz|nil]]], \
158 \ [y';y;[App|[Mz|[Atom z|nil]]],[atom1;atom2;sexpr,rlist]]>" 1);
159 by ((asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2)
160   THEN (Force_tac 2));
161 by (swap_res_tac lmixrlist.intrs 1);
162 force 1;
163 auto();
164 by (swap_res_tac lmixrlist.intrs 1);
165 force 1;
166 force 1;
167 force 1;
168 force 1;
169 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
170 by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
171 by (Asm_full_simp_tac 1);
172 force 1;
173 by (Asm_full_simp_tac 1);
174 qed "env_lemma2";
175
176 Goal "(<M,vlist'' # [y';y;My,[z';z;Mz,vlist']])> : lmixenv) --> \
177 \ (<M,vlist'' # [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']])> \
178 \ : lmixenv)";
179 auto();
180 by (eresolve_tac lmixenv.elims 1);
181 force 1;
182 by (dtac sym 1);
183 by (Asm_full_simp_tac 1);
184 by (subgoal_tac "(vlist'' # \
185 \ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist'])) : lmixrlist" 1);
186 by (subgoal_tac "vlist'' : lmixrlist" 2);
187 by (res_inst_tac [("My2","My")] ((env_lemma2 RS mp) RS mp) 2);
188 force 2;
189 force 2;
190 br ((lconcat_lemma1 RS mp) RS conjE) 2;
191 force 2;
192 force 2;

```

```

193 by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
194 by (Asm_full_simp_tac 1);
195 by (swap_res_tac lmixenv.intrs 1);
196 force 1;
197 force 1;
198 by (eresolve_tac lmrelims 1);
199 force 1;
200 auto();
201 qed "env_lemma3";
202
203 AddSEs lmixenv.elims;
204
205 Goal "(vlist' : lmixrlist) --> \
206 \ ((vlist' # [y';y;My,[z';z;Mz,vlist']]) : lmixrlist) --> \
207 \ ((vlist' # [y';y;My,addvar z' vlist']) : lmixrlist)";
208 br impI 1;
209 by (res_inst_tac [("xa","vlist'")] lmixrlist.induct 1);
210 force 1;
211
212 by (SELECT_GOAL Auto_tac 1);
213 by (swap_res_tac lmixrlist.intrs 1);
214 by (subgoal_tac "[z';z;Mz,vlist'] : lmixrlist" 1);
215 force 2;
216 by (res_inst_tac [("y2","z"),("My2","Mz")]
217      ((add_lemma1 RS mp) RS mp) 1);
218 force 1;
219 force 1;
220 force 1;
221 force 1;
222 force 1;
223 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
224 br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 1;
225 force 1;
226 force 1;
227 force 1;
228 by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
229 force 1;
230 force 1;
231
232 auto();
233 by (res_inst_tac [("vlist2","vlist"),
234      ("vlist'2","[y';y;My,[z';z;Mz,vlist']])]
235      ((lconcat_lemma1 RS mp) RS conjE) 1);
236 force 1;
237
238 by (swap_res_tac lmixrlist.intrs 1);
239 force 1;
240 force 1;
241 force 1;
242 force 1;
243 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
244 br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 1;
245 force 1;
246 force 1;
247 force 1;
248
249 by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
250 auto();

```



```

251 qed "env_lemma4";
252
253 Delrules lmixenv.elims;
254
255 Goal "(<M,vlist'' # [y';y;My,[z';z;Mz,vlist']]> : lmixenv) --> \
256 \ (<M,vlist'' # [y';y;My,addvar z' vlist']> : lmixenv)";
257 auto();
258 by (eresolve_tac lmixenv.elims 1);
259 force 1;
260 by (dtac sym 1);
261 by (Asm_full_simp_tac 1);
262 by (subgoal_tac "(vlist'' # [y';y;My,addvar z' vlist']) : lmixrlist" 1);
263 by (subgoal_tac "vlist'' : lmixrlist" 2);
264 by (res_inst_tac [("z'2","z'"),("z2","z"),("Mz2","Mz")]
265       ((env_lemma4 RS mp) RS mp) 2);
266 force 2;
267 force 2;
268 br ((lconcat_lemma1 RS mp) RS conjE) 2;
269 force 2;
270 force 2;
271 by (res_inst_tac [("y","vlist''])] rlist.exhaust 1);
272 auto();
273 qed "env_lemma5";
274
275 AddSEs lmixenv.elims;
276
277 Goal "(M : llexpr & (vars M <= vars M') & (<M',vlist> : lmixenv)) \
278 \ --> <M,vlist> : lmixenv";
279 auto();
280 by (swap_res_tac lmixenv.intrs 1);
281 auto();
282 qed "env_lemma6";
283
284 Delrules lmixenv.elims;
285
286 use "Determ";
287
288 Goal "vlist : lmixrlist --> <Atom v, [z';z;Mz,vlist]> : lmixenv --> \
289 \ (!d. (<Atom v,vlist> |- bodyann ---> d) ---> \
290 \ (<Atom v, addvar z' vlist> |- bodyann --->d))";
291 br impI 1;
292 by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
293 force 1;
294 force 1;
295 br impI 1;
296 by (eresolve_tac lmixenv.elims 1);
297 force 1;
298 by (dtac sym 1);
299 by (Asm_full_simp_tac 1);
300 by (eres_inst_tac [("a","[z';z;Mz,[v';va;M,vlista]]")])
301       lmixrlist.elim 1);
302 force 1;
303 by (dtac sym 1);
304 by (Asm_full_simp_tac 1);
305 by (subgoal_tac "[z';z;Mz,vlista] : lmixrlist" 1);
306 by (swap_res_tac lmixrlist.intrs 2);
307 force 2;
308 force 2;

```

```

309 force 2;
310 force 2;
311 force 3;
312 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
313
314 by (subgoal_tac "addvar z' vlista : lmixrlist" 1);
315 by (res_inst_tac [("y2","z"),("My2","Mz")]
316     ((add_lemma1 RS mp) RS mp) 2);
317 force 2;
318 force 2;
319
320 by (subgoal_tac "<Atom v,[z';z;Mz,vlista]> : lmixenv" 1);
321 by (swap_res_tac lmixenv.elims 2);
322 force 2;
323 force 2;
324 force 2;
325 by (mp_tac 1);
326
327 br allI 1;
328 br impI 1;
329 by (subgoal_tac "<Atom v, \
330 \ [v';va;[App|[M|[Atom z'|nil]]],addvar z' vlista]> : lmixenv" 1);
331 by (swap_res_tac lmixenv.intrs 2);
332 force 2;
333 by (swap_res_tac lmixrlist.intrs 2);
334 force 2;
335 force 2;
336 force 2;
337 force 2;
338 by (res_inst_tac [("y","vlista")] rlist.exhaust 3);
339 force 3;
340 by (Asm_full_simp_tac 3);
341 by (REPEAT (etac conjE 3));
342 force 3;
343 by (eres_inst_tac [("a","<Atom v,[z';z;Mz,vlista]>")] lmixenv.elim 3);
344 force 3;
345 by (Asm_full_simp_tac 3);
346 by (REPEAT (etac conjE 3));
347 force 3;
348 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
349 br conjI 2;
350 force 2;
351 br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 2;
352 force 2;
353 force 2;
354 force 2;
355
356 by (case_tac "v = va" 1);
357 auto();
358
359 by (res_inst_tac [("y","v"),("v","va"),("v'","v'"),("M","M"),
360     ("vlist","vlista"),("d","d")] derivelimvar1 3);
361 br conjI 3;
362 force 3;
363 br conjI 3;
364 force 4;
365 force 4;
366 by (swap_res_tac lmixenv.intrs 3);

```

```

367 force 3;
368 force 3;
369 force 3;
370
371 by (res_inst_tac [("y","vlista")] rlist.exhaust 2);
372 by (Asm_simp_tac 2);
373 by (Asm_simp_tac 2);
374 by (swap_res_tac lmixenv.intrs 2);
375 force 2;
376 force 2;
377
378 by (Asm_full_simp_tac 2);
379 by (subgoal_tac "v : vars sexpr" 2);
380 force 2;
381 force 2;
382
383 by (subgoal_tac "<Atom v,[v';v;M,vlista]> : lmixenv" 1);
384 by (swap_res_tac lmixenv.intrs 2);
385 force 2;
386 force 2;
387 force 2;
388 by (thin_tac "?xx" 1);
389 by (thin_tac "?xx" 1);
390 by (thin_tac "?xx" 1);
391 by (thin_tac "?xx" 1);
392 by (thin_tac "?xx" 1);
393 by (thin_tac "?xx" 1);
394 by (thin_tac "?xx" 1);
395 by (thin_tac "?xx" 1);
396 by (thin_tac "?xx" 1);
397 by (thin_tac "?xx" 1);
398 by (thin_tac "?xx" 1);
399 by (thin_tac "?xx" 1);
400 by (thin_tac "?xx" 1);
401 by (thin_tac "?xx" 1);
402 by (rotate_tac 1 1);
403 by (thin_tac "?xx" 1);
404 by (thin_tac "?xx" 1);
405 by (thin_tac "?xx" 1);
406 by (thin_tac "?xx" 1);
407 by (thin_tac "?xx" 1);
408 force 1;
409 qed "add_lemma2";
410
411 AddSEs lmixenv.elims;
412 Goal "[|<Atom v, [z';z;Mz,vlist]> : lmixenv; <Atom v,vlist> \
413 \ |- bodyann --->d|] ==> <Atom v, addvar z' vlist> |- bodyann ---> d";
414 by (res_inst_tac [("vlist3","vlist"),("v3","v"),("z'3","z'"),("z3","z"),
415 ("Mz3","Mz"),("x","d")] ((add_lemma2 RS mp) RS mp) RS allE) 1);
416 auto();
417 qed "add_lemma2b";
418 Delrules lmixenv.elims;
419
420 Goal "(lconcat (lconcat v11 v12) v13) = \
421 \ (lconcat v11 (lconcat v12 v13))";
422 by (res_inst_tac [("rlist","v11")] rlist.induct 1);
423 auto();
424 qed "lcdist";

```

```

425
426 Addsimps [lcdist];
427
428 Goal "(y = z & M : l1expr) --> (!vlist. \
429 \ (<M,vlist # [y';y;My,[z';z;Mz,vlist']> : lmixenv) --> \
430 \ (!d. (<M,vlist # [y';y;My,[z';z;Mz,vlist']> |- bodyann ----> d) \
431 \ --> (<M,vlist # [y';y;My,addvar z' vlist']> |- bodyann ----> d)))";
432 br impI 1;
433 by (res_inst_tac [("xa","M")] l1expr.induct 1);
434
435 (** App case **)
436 by (SELECT_GOAL Auto_tac 5);
437 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']")])
438   ((determ_app RS mp) RS mp) 5);
439 br conjI 5;
440 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
441   (env_lemma5 RS mp) 6);
442 force 5;
443 force 5;
444 force 5;
445 force 5;
446
447 (** Fix case **)
448 by (SELECT_GOAL Auto_tac 5);
449 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']")])
450   ((determ_fix RS mp) RS mp) 5);
451 br conjI 5;
452 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
453   (env_lemma5 RS mp) 6);
454 force 5;
455 force 5;
456 force 5;
457 force 5;
458
459 (** Cond case **)
460 by (SELECT_GOAL Auto_tac 5);
461 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']")])
462   ((determ_if RS mp) RS mp) 5);
463 br conjI 5;
464 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
465   (env_lemma5 RS mp) 6);
466 force 5;
467 force 5;
468 force 5;
469 force 5;
470
471 (** Cons case **)
472 by (SELECT_GOAL Auto_tac 5);
473 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']")])
474   ((determ_cons RS mp) RS mp) 5);
475 br conjI 5;
476 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
477   (env_lemma5 RS mp) 6);
478 force 5;
479 force 5;
480 force 5;
481 force 5;
482

```

```

483 (** Car case **)
484 by (SELECT_GOAL Auto_tac 5);
485 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']"])]
486     (((determ_car RS mp) RS mp) RS mp) 5);
487 br conjI 5;
488 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
489     (env_lemma5 RS mp) 6);
490 force 5;
491 force 5;
492 force 5;
493 force 5;
494
495 (** Cdr case **)
496 by (SELECT_GOAL Auto_tac 5);
497 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']"])]
498     (((determ_cdr RS mp) RS mp) RS mp) 5);
499 br conjI 5;
500 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
501     (env_lemma5 RS mp) 6);
502 force 5;
503 force 5;
504 force 5;
505 force 5;
506
507 (** IsEq case **)
508 by (SELECT_GOAL Auto_tac 5);
509 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']"])]
510     (((determ_iseq RS mp) RS mp) RS mp) 5);
511 br conjI 5;
512 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
513     (env_lemma5 RS mp) 6);
514 force 5;
515 force 5;
516 force 5;
517 force 5;
518
519 (** IsAtom case **)
520 by (SELECT_GOAL Auto_tac 5);
521 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']"])]
522     (((determ_isatom RS mp) RS mp) RS mp) 5);
523 br conjI 5;
524 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
525     (env_lemma5 RS mp) 6);
526 force 5;
527 force 5;
528 force 5;
529 force 5;
530
531 (** Error case **)
532 by (SELECT_GOAL Auto_tac 5);
533 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']"])]
534     (((determ_error RS mp) RS mp) RS mp) 5);
535 br conjI 5;
536 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
537     (env_lemma5 RS mp) 6);
538 force 5;
539 force 5;
540 force 5;

```

```

541 force 5;
542
543 (** Triv case **)
544 force 1;
545
546 (** Quote case **)
547 by (SELECT_GOAL Auto_tac 2);
548 by (res_inst_tac [("vlist2","vlist # [y';z;My,[z';z;Mz,vlist']"])]
549     ((determ_quote RS mp) RS mp) 2);
550 br conjI 2;
551 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
552     (env_lemma5 RS mp) 3);
553 force 2;
554 force 2;
555 force 2;
556
557 (** Lam case **)
558 by (SELECT_GOAL Auto_tac 2);
559 by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']"])]
560     (((determ_lam RS mp) RS mp) RS mp) 2);
561 force 4;
562
563 br conjI 2;
564 force 2;
565 br conjI 2;
566 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
567     (env_lemma5 RS mp) 2);
568 force 2;
569
570 by (eres_inst_tac
571     [("a","<Ma,[v';a;Ma,vlist # [y';z;My,[z';z;Mz,vlist']]>")]
572     lmixenv.elim 2);
573 force 2;
574 by (dtac sym 2);
575 by (Asm_full_simp_tac 2);
576 by (subgoal_tac "(vlist # [y';z;My,[z';z;Mz,vlist']]) : lmixrlist" 2);
577 force 3;
578 by (res_inst_tac [("vlist2","vlist"),
579     ("vlist'2","[y';z;My,[z';z;Mz,vlist']")])
580     ((lconcat_lemma1 RS mp) RS conjE) 2);
581 force 2;
582 by (subgoal_tac "y' : var" 2);
583 force 3;
584 by (subgoal_tac "z : var" 2);
585 force 3;
586 by (subgoal_tac "z' : var" 2);
587 force 3;
588
589 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
590 by (SELECT_GOAL Auto_tac 2);
591 by (rename_tac "Ma a vlist dM v' q" 2);
592 by (subgoal_tac "q : vars (lenv vlist') Un {z'}" 2);
593 force 2;
594 br (((lenv_lemma_add RS mp) RS mp) RS subsetD) 2;
595 force 2;
596 force 2;
597 force 2;
598

```

```

599 br allI 2;
600 br impI 2;
601 br allI 2;
602 br impI 2;
603 by (eres_inst_tac [("x","(vlist'' # vlist)")] allE 2);
604 by (Asm_full_simp_tac 2);
605
606 (** Var case **)
607
608 br allI 1;
609 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
610
611 (** induct start **)
612 by (SELECT_GOAL Auto_tac 1);
613 by (case_tac "a = z" 1);
614 by (SELECT_GOAL Auto_tac 1);
615 by (res_inst_tac [("vlist2","[z';z;Mz,vlist']"])
616       ((determ_varnonemp RS mp) RS mp) 1);
617 br conjI 1;
618 force 1;
619 force 2;
620
621 by (subgoal_tac "<Atom z,[] # [y';z;My,addvar z' vlist']> : lmixenv" 1);
622 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
623       (env_lemma5 RS mp) 2);
624 force 1;
625 force 1;
626
627 br (derivelimvar1) 1;
628 force 1;
629 br (derivelimvar1) 1;
630 force 1;
631
632 br (determ_varnonemp2 RS mp) 1;
633 br conjI 1;
634 by (subgoal_tac "<Atom a,[] # [y';z;My,addvar z' vlist']> : lmixenv" 1);
635 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
636       (env_lemma5 RS mp) 2);
637 force 1;
638 force 1;
639 br conjI 1;
640 force 1;
641
642 br add_lemma2b 1;
643 force 1;
644 force 1;
645
646 (** induct step **)
647 br impI 1;
648 br allI 1;
649 br impI 1;
650 by (case_tac "a = atom2" 1);
651 by (Asm_full_simp_tac 1);
652
653 by (subgoal_tac "<Atom atom2,[atom1;atom2;sexpr,rlist] \
654 \ # [y';z;My,addvar z' vlist']> : lmixenv" 1);
655 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
656       (env_lemma5 RS mp) 2);

```

```

657 force 1;
658 force 1;
659
660 br (derivelimvar1) 1;
661 force 1;
662 by (Asm_full_simp_tac 1);
663 by (eres_inst_tac [("x","d")] allE 1);
664 by (Asm_full_simp_tac 1);
665
666 br (determ_varnonemp2 RS mp) 1;
667 br conjI 1;
668 force 2;
669 by (subgoal_tac "<Atom a,[atom1;atom2;sexpr,rlist] \
670 \ # [y';z;My,addvar z' vlist]> : lmixenv" 1);
671 by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
672       (env_lemma5 RS mp) 2);
673 force 1;
674 force 1;
675 qed "lemma_513";

1  (*****
2  *
3  * Eval.ML
4  *
5  *****)
6
7  Goalw [sintann_def] "(nil |- sintann ---> d) = \
8  \ (d = [clos|[expr|[[Lam|[env|[bodyann|nil]]]| \
9  \ [[eval|[delay|[sintann|[nil|nil]]]]|nil|nil]]]])";
10 auto();
11 qed "nilsint";
12
13 Addsimps [nilsint];
14
15 Goal "(nil |- [App|[[App|[sintann|[[Quote|[M|nil]]|nil]]]| \
16 \ [Lam|[x|[x|nil]]]|nil]]] ---> d) = \
17 \ (<M,[ ]> |- bodyann ---> d)";
18 auto();
19 by (rewtac lwrap_def);
20 force 1;
21 force 1;
22 qed "sintstart";
23
24 Goal "<M,vlist> : lmixenv) --> \
25 \ (<M,vlist> |- expr ---> d) = (d = M)";
26 auto();
27 qed "eval_expr";
28
29 Goal "<M,vlist> |- eval ---> d) = \
30 \ (d = [clos|[expr|[[Lam|[env|[bodyann|nil]]]|[[eval|[delay| \
31 \ [sintann|[nil|nil]]]]|nil|nil]]]])";
32 auto();
33 qed "eval_eval";
34
35 Goal "<M,[ ]> |- env ---> d) = (d = [clos|[x|[x|[nil|nil]]]])";
36 auto();
37 qed "eval_envnone";
38
39 Goal "<M,[v';v;M',vlist]> : lmixenv) --> \

```



```

40 \ (<M,[v';v;M',vlist]> |- env ---> d) = (d = \
41 \ [clos|[var|[[Cond|[[IsEq|[var|[[Cadr|[expr|nil]]|nil]]]]| \
42 \ [value|[App|[env|[var|nil]]]|nil]]]]| \
43 \ [[value|Atom v']|<[Lam|[Atom v|[M'|nil]]],vlist>|nil]]]]");
44 auto();
45 qed "eval_envnone";
46
47 Addsimps [eval_expr,eval_eval,eval_envnone,eval_envnone];

1  (*****
2  *
3  * LEnv.ML
4  *
5  *****)
6
7  Addsimps [lconcat_none,lconcat_nemp];
8  Addsimps [varsin_none,varsin_nemp];
9
10 Goal "(Atom w : llexprs & [v';v;M',vlist] : lmixenv & \
11 \ w ~: varsin [v';v;M',vlist]) --> (w ~ = v & w ~: varsin vlist)";
12 auto();
13 qed "atomin_lemma1";
14
15 Delsimps [varsin_none,varsin_nemp];
16
17 (*****)
18 (*****)
19
20 Goal "(Atom v : llexprs & vlist : lmixenv) --> \
21 \ (v ~: varsin vlist) --> \
22 \ ((<Atom v,vlist> |- bodyann ---> d) = (d = (Atom v)))";
23 br impI 1;
24 br lmixenv.induct 1;
25 force 1;
26 br impI 1;
27 br (derivvaremp RS mp) 1;
28 force 1;
29 br impI 1;
30 by (subgoal_tac "[v';va;M',vlista] : lmixenv & \
31 \ v ~ = va & v ~: varsin vlista" 1);
32 br conjI 2;
33 by (res_inst_tac [("w1","v"),("v1","va"),("v'1","v'),("M'1","M'),
34 ("vlist1","vlista")] (atomin_lemma1 RS mp) 3);
35 by (resolve_tac lmixenv.intrs 2);
36 by (REPEAT (Force_tac 2));
37 br conjI 2;
38 force 2;
39 br conjI 2;
40 force 3;
41 by (resolve_tac lmixenv.intrs 2);
42 by (REPEAT (Force_tac 2));
43 auto();
44 qed "lemma_511";
45
46 (*****)
47 (*****)
48
49 Addsimps [varsin_none,varsin_nemp,vars_atom,vars_pair];
50

```

```

51 Goal "(vlist : lmixenv) --> (varsin vlist <= vars (lenv vlist))";
52 br impI 1;
53 br lmixenv.induct 1;
54 by ((simp_tac (simpset() addsimps [lwrap_def]) 3) THEN Auto_tac);
55 qed "whee67";
56
57 Addsimps [whee67];
58 Delsimps [varsin_none, varsin_nemp];
59 Delsimps [lconcat_none, lconcat_nemp];

1  (*****
2  *
3  * LWrap.ML
4  *
5  *****)
6
7  (**
8  *
9  * Results regarding vars.
10 *
11 **)
12
13 Addsimps [lwrap_def];
14
15 Addsimps [vars_atom, vars_pair];
16
17 Goal "(vlist : lmixrlist) --> varsin vlist <= vars <M,vlist>";
18 br impI 1;
19 by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
20 auto();
21 qed "vars_lemma1";
22
23 Addsimps [bodyann_def, sintann_def];
24 Goal "vars [App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[envlam|nil]]] <= \
25 \ vars <M,vlist>";
26 auto();
27 qed "vars_lemma2";
28 Delsimps [bodyann_def, sintann_def];
29
30 Addsimps [vars_lemma1, vars_lemma2];
31
32 Delsimps [vars_atom, vars_pair];
33
34 (**
35 *
36 * Results regarding lookup.
37 *
38 **)
39
40 Goal "(<M,vlist> : lmixenv | (!q. M ~= [delay|q])) --> \
41 \ (lookup (<M,vlist>,expr) = M)";
42 auto();
43 qed "lookup_expr";
44
45 Goal "(<M,vlist> : lmixenv | (!q. M ~= [delay|q])) --> \
46 \ (lookup (<M,vlist>,expr) = d) = (d = M)";
47 auto();
48 qed "lookup_expr2";
49

```

```

50 Goal "(lookup (<M,vlist>,eval) = [delay|[sintann|[nil|nil]]])";
51 auto();
52 qed "lookup_eval";
53
54 Goal "(lookup (<M,vlist>,eval) = d) = (d = [delay|[sintann|[nil|nil]]])";
55 auto();
56 qed "lookup_eval2";
57
58 Goal "(lookup (<M,[],>,env) = [clos|[x|[x|[nil|nil]]]])";
59 auto();
60 qed "lookup_none";
61
62 Goal "(lookup (<M,[],>,env) = d) = (d = [clos|[x|[x|[nil|nil]]]])";
63 auto();
64 qed "lookup_none2";
65
66 Goal "lookup (<M,[v';v;M',vlist]>,env) = \
67 \ [clos|[var|[[Cond|[[IsEq|[var|[[Cadr|[expr|nil]]|nil]]]]| \
68 \ [value|[[App|[env|[var|nil]]]|nil]]]]| \
69 \ [[value|Atom v']|<[Lam|[Atom v|[M'|nil]]],vlist>|nil]]]";
70 auto();
71 qed "lookup_nonemp";
72
73 Goal "(lookup (<M,[v';v;M',vlist]>,env) = d) = (d = \
74 \ [clos|[var|[[Cond|[[IsEq|[var|[[Cadr|[expr|nil]]|nil]]]]| \
75 \ [value|[[App|[env|[var|nil]]]|nil]]]]| \
76 \ [[value|Atom v']|<[Lam|[Atom v|[M'|nil]]],vlist>|nil]]])";
77 auto();
78 qed "lookup_nonemp2";
79
80 Addsimps [lookup_expr,lookup_expr2,lookup_eval,lookup_eval2];
81 Addsimps [lookup_none,lookup_none2,lookup_nonemp,lookup_nonemp2];
82
83 Delsimps [lwrap_def];

1  (*****
2  *
3  * LemAlpha.ML
4  *
5  *****)
6
7  Goal "M : llexpr ==> size (rename (a,b,M)) = size M";
8  br llexpr.induct 1;
9  auto();
10 qed "rensz";
11
12 Goal "M : llexpr ==> rename (a,a,M) = M";
13 br llexpr.induct 1;
14 auto();
15 qed "renid";
16
17 Addsimps [rensz,renid];
18
19 Goal "! M : llexpr. size M < i --> \
20 \ (!dM. (<M,[],> |- bodyann ----> dM) --> (M -a- dM))";
21 by (induct_tac "i" 1);
22 force 1;
23 by (simp_tac (simpset() addsimps [less_Suc_eq]) 1);
24 auto();

```

```

25 by (subgoal_tac "<M, []> : lmixenv" 1);
26 force 2;
27 by (rotate_tac ~1 1);
28 by (eresolve_tac llexpr.elims 1);
29
30 force 1;
31 force 1;
32 by (REPEAT (Force_tac 2));
33
34 (** Lam case **)
35 auto();
36
37 by (res_inst_tac [("M2", "Ma"), ("x", "[ ]"), ("v'2", "v'"), ("v2", "a"),
38   ("Mv2", "Ma")] ((lemma_515 RS mp) RS allE) 1);
39 force 1;
40 by (Asm_full_simp_tac 1);
41 by (subgoal_tac "<rename (a, v', Ma), []> : lmixenv" 1);
42 by (swap_res_tac lmixenv.intrs 2);
43 br (ren1 RS mp) 2;
44 by (eres_inst_tac [("a", "<Ma, [v'; a; Ma, []]>")] lmixenv.elim 2);
45 force 2;
46 force 2;
47
48 by (Asm_full_simp_tac 1);
49 by (subgoal_tac "rename (a, v', Ma) : llexpr" 1);
50 by (eres_inst_tac [("a", "<rename (a, v', Ma), []>")] lmixenv.elim 2);
51 force 2;
52 force 2;
53 by (eres_inst_tac [("x", "rename (a, v', Ma)"] ballE 1);
54 force 2;
55 by (eres_inst_tac [("x", "dMa")] allE 1);
56
57 by (Asm_full_simp_tac 1);
58 by (eres_inst_tac [("x", "dMa")] allE 1);
59 by (Asm_full_simp_tac 1);
60 by (case_tac "a=v'" 1);
61 force 1;
62 br alpha.alpha_sym 1;
63 by (res_inst_tac [("M'", "[Lam | [Atom v' | [rename (a, v', Ma) | nil]]] ")"]
64   alpha.alpha_trans 1);
65 br alpha.alpha_lam1 1;
66 by (eres_inst_tac [("a", "<Ma, [v'; a; Ma, []]>")] lmixenv.elim 1);
67 force 1;
68 force 1;
69 br alpha.alpha_sym 1;
70 force 1;
71 br alpha.alpha_sym 1;
72 br alpha.alpha_lam2 1;
73 by (eres_inst_tac [("a", "<Ma, [v'; a; Ma, []]>")] lmixenv.elim 1);
74 force 1;
75 force 1;
76 by (eres_inst_tac [("a", "<Ma, [v'; a; Ma, []]>")] lmixenv.elim 1);
77 force 1;
78 force 2;
79 auto();
80 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
81 qed "lemma_516";
82

```

```

83 Goal "M : l1expr --> (? dM : l1expr. (<M,[]> |- bodyann ----> dM))";
84 br impI 1;
85 by (res_inst_tac [("M2","M"),("x","[]")] ((lemma_514 RS mp) RS allE) 1);
86 auto();
87 qed "lemma_514b";
88
89 Goal "M : l1expr --> (<M,[]> |- bodyann ----> dM) --> (M -a- dM)";
90 br impI 1;
91 by (res_inst_tac [("x","M"),("i1","size [Fix|[M|nil]]")]
92      (lemma_516 RS ballE) 1);
93 auto();
94 qed "lemma_516b";
95
96 Addsimps [sintstart];
97
98 Goal "M : l1expr ==> EX dM : l1expr. (nil |- \
99 \ [App|[App|[sintann|[Quote|[M|nil]]|nil]]]| \
100 \ [Lam|[x|[x|nil]]]|nil]]] ----> dM)";
101 auto();
102 br (lemma_514b RS mp) 1;
103 force 1;
104 qed "thm_term";
105
106 Goal "M : l1expr ==> \
107 \ ( nil |- [App|[App|[sintann|[Quote|[M|nil]]|nil]]]| [Lam|[x|[x|nil]]]|nil]] \
108 \ ----> dM ) --> (M -a- dM)";
109 auto();
110 br ((lemma_516b RS mp) RS mp) 1;
111 auto();
112 qed "thm_uniq";

1  (*****
2  *
3  * LemEnv.ML
4  *
5  *****)
6
7  Goal "(y ~= z & M : l1expr) --> (!vlist. \
8  \ (<M,vlist # [y';y;My,[z';z;Mz,vlist']]> : lmixenv) --> \
9  \ (!d. (<M,vlist # [y';y;My,[z';z;Mz,vlist']]> |- bodyann ----> d) \
10 \ --> (<M,vlist # [z';z;Mz,[y';y;App|[Mz|[Atom z|nil]]],vlist']]> \
11 \ |- bodyann ----> d))";
12 br impI 1;
13 by (res_inst_tac [("xa","M")] l1expr.induct 1);
14
15 (** App case **)
16 by (SELECT_GOAL Auto_tac 5);
17 by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")
18      ((determ_app RS mp) RS mp) RS mp) 5);
19 br conjI 5;
20 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
21 force 5;
22 force 5;
23 force 5;
24 force 5;
25
26 (** Fix case **)
27 by (SELECT_GOAL Auto_tac 5);
28 by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")

```

```

29         (((determ_fix RS mp) RS mp) RS mp) 5);
30   br conjI 5;
31   by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
32   force 5;
33   force 5;
34   force 5;
35   force 5;
36
37   (** Cond case **)
38   by (SELECT_GOAL Auto_tac 5);
39   by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']"])]
40       (((determ_if RS mp) RS mp) RS mp) 5);
41   br conjI 5;
42   by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
43   force 5;
44   force 5;
45   force 5;
46   force 5;
47
48   (** Cons case **)
49   by (SELECT_GOAL Auto_tac 5);
50   by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']"])]
51       (((determ_cons RS mp) RS mp) RS mp) 5);
52   br conjI 5;
53   by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
54   force 5;
55   force 5;
56   force 5;
57   force 5;
58
59   (** Car case **)
60   by (SELECT_GOAL Auto_tac 5);
61   by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']"])]
62       (((determ_car RS mp) RS mp) RS mp) 5);
63   br conjI 5;
64   by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
65   force 5;
66   force 5;
67   force 5;
68   force 5;
69
70   (** Cdr case **)
71   by (SELECT_GOAL Auto_tac 5);
72   by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']"])]
73       (((determ_cdr RS mp) RS mp) RS mp) 5);
74   br conjI 5;
75   by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
76   force 5;
77   force 5;
78   force 5;
79   force 5;
80
81   (** IsEq case **)
82   by (SELECT_GOAL Auto_tac 5);
83   by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']"])]
84       (((determ_iseq RS mp) RS mp) RS mp) 5);
85   br conjI 5;
86   by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);

```

```

87 force 5;
88 force 5;
89 force 5;
90 force 5;
91
92 (** IsAtom case **)
93 by (SELECT_GOAL Auto_tac 5);
94 by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']")])
95   (((determ_isatom RS mp) RS mp) RS mp) 5);
96 br conjI 5;
97 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
98 force 5;
99 force 5;
100 force 5;
101 force 5;
102
103 (** Error case **)
104 by (SELECT_GOAL Auto_tac 5);
105 by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']")])
106   (((determ_error RS mp) RS mp) RS mp) 5);
107 br conjI 5;
108 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
109 force 5;
110 force 5;
111 force 5;
112 force 5;
113
114 (** Triv case **)
115 force 1;
116
117 (** Quote case **)
118 by (SELECT_GOAL Auto_tac 2);
119 by (res_inst_tac [("vlist2","vlist # [y';y;My,[z';z;Mz,vlist']")])
120   ((determ_quote RS mp) RS mp) 2);
121 br conjI 2;
122 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 3);
123 force 2;
124 force 2;
125 force 2;
126
127 (** Lam case **)
128 by (SELECT_GOAL Auto_tac 2);
129 by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']")])
130   (((determ_lam RS mp) RS mp) RS mp) 2);
131 force 4;
132
133 br conjI 2;
134 force 2;
135 br conjI 2;
136 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
137 force 2;
138 by ((asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2)
139     THEN (Force_tac 2));
140
141 br allI 2;
142 br impI 2;
143 br allI 2;
144 br impI 2;

```

```

145 by (eres_inst_tac [("x","(vlist'' # vlist)"] allE 2);
146 by (Asm_full_simp_tac 2);
147
148 (** Var case **)
149 br allI 1;
150 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
151
152 (** induct start **)
153 by (SELECT_GOAL Auto_tac 1);
154 by (case_tac "a = y" 1);
155 by (SELECT_GOAL Auto_tac 1);
156 br (determ_varnonemp2 RS mp) 1;
157 br conjI 1;
158 by (subgoal_tac "<Atom y,[] # \
159 \ [z';z;Mz,[y';y];[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
160 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
161 force 1;
162 force 1;
163 br conjI 1;
164 force 1;
165 by (subgoal_tac "<Atom y,[] # \
166 \ [y';y];[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
167 force 1;
168 by (subgoal_tac "<Atom y,[] # \
169 \ [z';z;Mz,[y';y];[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
170 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
171 by (Asm_full_simp_tac 1);
172 br (lmered RS mp) 1;
173 force 1;
174 force 1;
175 by (case_tac "a = z" 1);
176 by (SELECT_GOAL Auto_tac 1);
177 by (res_inst_tac [("vlist2","vlist'")]
178       ((determ_varnonemp RS mp) RS mp) 1);
179 br conjI 1;
180 br (lmered RS mp) 1;
181 force 1;
182 by (subgoal_tac "<Atom z,[] # \
183 \ [z';z;Mz,[y';y];[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
184 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
185 force 1;
186 force 1;
187 br (derivelimvar1) 1;
188 force 1;
189 force 1;
190 br (derivelimvar1) 1;
191 force 1;
192 by (res_inst_tac [("v","z")] derivelimvar1 1);
193 force 1;
194 br (determ_varnonemp2 RS mp) 1;
195 br conjI 1;
196 by (subgoal_tac "<Atom a,[] # \
197 \ [z';z;Mz,[y';y];[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
198 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
199 force 1;
200 force 1;
201 br conjI 1;
202 force 1;

```



```

203 br (determ_varnonemp2 RS mp) 1;
204 br conjI 1;
205 by (subgoal_tac "<Atom a, [] # \
206 \ [y';y;[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
207 force 1;
208 by (subgoal_tac "<Atom a, [] # \
209 \ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
210 by (Asm_full_simp_tac 1);
211 br (lmered RS mp) 1;
212 force 1;
213 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 1);
214 force 1;
215 force 1;
216
217 (** induct step **)
218 br impI 1;
219 br allI 1;
220 br impI 1;
221 by (case_tac "a = atom2" 1);
222 by (Asm_full_simp_tac 1);
223
224 by (subgoal_tac "<Atom atom2,[atom1;atom2;sexpr,rlist] # \
225 \ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
226 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
227 force 1;
228 force 1;
229
230 br (derivelimvar1) 1;
231 force 1;
232 by (Asm_full_simp_tac 1);
233 by (eres_inst_tac [("x","d")] allE 1);
234 by (Asm_full_simp_tac 1);
235
236 br (determ_varnonemp2 RS mp) 1;
237 br conjI 1;
238 force 2;
239 by (subgoal_tac "<Atom a,[atom1;atom2;sexpr,rlist] # \
240 \ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
241 by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
242 force 1;
243 force 1;
244 qed "lemma_512";

1  (*****
2  *
3  * LemTerm.ML
4  *
5  *****)
6
7  Goal "vars sexpr : Finites";
8  br sexpr.induct 1;
9  auto();
10 qed "varsfin";
11
12 Addsimps [varsfin];
13
14 Goal "M : l1expr --> (!vlist. (vlist : lmixrlist --> \
15 \ (<M,vlist> : lmixenv --> \
16 \ (? dM : l1expr. (<M,vlist> |- bodyann ---> dM))))";

```

```

17  br impI 1;
18  by (res_inst_tac [("xa","M")] l1expr.induct 1);
19  force 1;
20  force 2;
21
22  (** App case **)
23  br allI 3;
24  br impI 3;
25  br impI 3;
26  by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
27  by (subgoal_tac "<N,vlist> : lmixenv" 3);
28  by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
29  force 4;
30  by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
31  force 4;
32  force 3;
33
34  (** Fix case **)
35  br allI 3;
36  br impI 3;
37  br impI 3;
38  by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
39  by (res_inst_tac [("M'1","[Fix|Ma|nil]")] (env_lemma6 RS mp) 4);
40  force 4;
41  force 3;
42
43  (** Cond case **)
44  br allI 3;
45  br impI 3;
46  br impI 3;
47  by (subgoal_tac "<B,vlist> : lmixenv" 3);
48  by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
49  by (subgoal_tac "<N,vlist> : lmixenv" 3);
50  by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
51  force 4;
52  by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
53  force 4;
54  by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
55  force 4;
56  by (eres_inst_tac [("x","vlist")] allE 3);
57  by (eres_inst_tac [("x","vlist")] allE 3);
58  by (eres_inst_tac [("x","vlist")] allE 3);
59  force 3;
60
61  (** Cons case **)
62  br allI 3;
63  br impI 3;
64  br impI 3;
65  by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
66  by (subgoal_tac "<N,vlist> : lmixenv" 3);
67  by (res_inst_tac [("M'1","[Cons|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
68  force 4;
69  by (res_inst_tac [("M'1","[Cons|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
70  force 4;
71  force 3;
72
73  (** Car case **)
74  br allI 3;

```

```

75  br impI 3;
76  br impI 3;
77  by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
78  by (res_inst_tac [("M'1","[Car|[Ma|nil]]")] (env_lemma6 RS mp) 4);
79  force 4;
80  force 3;
81
82  (** Cdr case **)
83  br allI 3;
84  br impI 3;
85  br impI 3;
86  by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
87  by (res_inst_tac [("M'1","[Cdr|[Ma|nil]]")] (env_lemma6 RS mp) 4);
88  force 4;
89  force 3;
90
91  (** IsEq case **)
92  br allI 3;
93  br impI 3;
94  br impI 3;
95  by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
96  by (subgoal_tac "<N,vlist> : lmixenv" 3);
97  by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]]") (env_lemma6 RS mp) 4);
98  force 4;
99  by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]]") (env_lemma6 RS mp) 4);
100 force 4;
101 force 3;
102
103 (** IsAtom case **)
104 br allI 3;
105 br impI 3;
106 br impI 3;
107 by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
108 by (res_inst_tac [("M'1","[IsAtom|[Ma|nil]]")] (env_lemma6 RS mp) 4);
109 force 4;
110 force 3;
111
112 (** Error case **)
113 br allI 3;
114 br impI 3;
115 br impI 3;
116 by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
117 by (res_inst_tac [("M'1","[Error|[Ma|nil]]")] (env_lemma6 RS mp) 4);
118 force 4;
119 force 3;
120
121 (** Var case **)
122 br allI 1;
123 br impI 1;
124 br lmixrlist.induct 1;
125 force 1;
126 force 1;
127 br impI 1;
128 by (case_tac "a = v" 1);
129 force 1;
130 by (subgoal_tac "<Atom a,vlista> : lmixenv" 1);
131 br (lmered RS mp) 2;
132 force 1;

```

```

133 force 1;
134
135 (** Lam case **)
136 auto();
137 by (subgoal_tac "? v'. <Ma,[v';a;Ma,vlist]> : lmixenv" 1);
138 be exE 1;
139 by (subgoal_tac "[v';a;Ma,vlist] : lmixrlist" 1);
140 by (eres_inst_tac [("a","<Ma,[v';a;Ma,vlist]>")] lmixenv.elim 2);
141 force 2;
142 force 2;
143
144 by (eres_inst_tac [("x","[v';a;Ma,vlist]")] allE 1);
145 auto();
146
147 (** Choose case **)
148 by (res_inst_tac [("x","@ v'. v' : var & \
149 \ v' ~: vars <[Lam|[Atom a|[Ma|nil]]],vlist>")] exI 1);
150 by (thin_tac "?xx" 1);
151 by (rotate_tac ~3 1);
152 by (thin_tac "?xx" 1);
153 by (subgoal_tac "? v'. v' : var & \
154 \ v' ~: vars <[Lam|[Atom a|[M|nil]]],vlist>" 1);
155 by (swap_res_tac lmixenv.intrs 1);
156 force 1;
157 force 2;
158 by (swap_res_tac lmixrlist.intrs 1);
159 force 1;
160 force 1;
161 force 1;
162 by (eresolve_tac lmixenv.elims 3);
163 force 3;
164 force 3;
165 br selectI2EX 1;
166 force 1;
167 force 1;
168 br selectI2EX 1;
169 force 1;
170 force 1;
171 br freshvar 1;
172 force 1;
173 qed "lemma_514";

1  (*****
2  *
3  * LemWeak.ML
4  *
5  *****)
6
7  Goal "M : l1expr --> \
8  \ (!vlist. <M,vlist # [v';v;Mv,vlist']> : lmixenv & \
9  \      <M,vlist # [v';v;Mv',vlist']> : lmixenv & \
10 \      vars Mv' <= vars Mv --> \
11 \      (!dM. (<M,vlist # [v';v;Mv,vlist']> |- bodyann ----> dM) --> \
12 \      (<M,vlist # [v';v;Mv',vlist']> |- bodyann ----> dM)))";
13 br impI 1;
14 by (res_inst_tac [("xa","M")] l1expr.induct 1);
15
16 force 1;
17 force 2;

```

```

18
19 (** App case **)
20 br allI 3;
21 br impI 3;
22 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
23 by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")]) (env_lemma6 RS mp) 4);
24 force 4;
25 by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
26 by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")]) (env_lemma6 RS mp) 4);
27 force 4;
28 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
29 by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")]) (env_lemma6 RS mp) 4);
30 force 4;
31 by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
32 by (res_inst_tac [("M'1","[App|Ma|[N|nil]]")]) (env_lemma6 RS mp) 4);
33 force 4;
34 force 3;
35
36 (** Fix case **)
37 br allI 3;
38 br impI 3;
39 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
40 by (res_inst_tac [("M'1","[Fix|Ma|nil]")]) (env_lemma6 RS mp) 4);
41 force 4;
42 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
43 by (res_inst_tac [("M'1","[Fix|Ma|nil]")]) (env_lemma6 RS mp) 4);
44 force 4;
45 force 3;
46
47 (** Cond case **)
48 br allI 3;
49 br impI 3;
50 by (subgoal_tac "<B,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
51 by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
52 force 4;
53 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
54 by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
55 force 4;
56 by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
57 by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
58 force 4;
59 by (subgoal_tac "<B,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
60 by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
61 force 4;
62 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
63 by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
64 force 4;
65 by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
66 by (res_inst_tac [("M'1","[Cond|B|[Ma|[N|nil]]]")]) (env_lemma6 RS mp) 4);
67 force 4;
68 force 3;
69
70 (** Cons case **)
71 br allI 3;
72 br impI 3;
73 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
74 by (res_inst_tac [("M'1","[Cons|Ma|[N|nil]]")]) (env_lemma6 RS mp) 4);
75 force 4;

```

```

76 by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
77 by (res_inst_tac [("M'1","[Cons|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
78 force 4;
79 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
80 by (res_inst_tac [("M'1","[Cons|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
81 force 4;
82 by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
83 by (res_inst_tac [("M'1","[Cons|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
84 force 4;
85 force 3;
86
87 (** Car case **)
88 br allI 3;
89 br impI 3;
90 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
91 by (res_inst_tac [("M'1","[Car|Ma|nil]")] (env_lemma6 RS mp) 4);
92 force 4;
93 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
94 by (res_inst_tac [("M'1","[Car|Ma|nil]")] (env_lemma6 RS mp) 4);
95 force 4;
96 force 3;
97
98 (** Cdr case **)
99 br allI 3;
100 br impI 3;
101 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
102 by (res_inst_tac [("M'1","[Cdr|Ma|nil]")] (env_lemma6 RS mp) 4);
103 force 4;
104 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
105 by (res_inst_tac [("M'1","[Cdr|Ma|nil]")] (env_lemma6 RS mp) 4);
106 force 4;
107 force 3;
108
109 (** IsEq case **)
110 br allI 3;
111 br impI 3;
112 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
113 by (res_inst_tac [("M'1","[IsEq|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
114 force 4;
115 by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
116 by (res_inst_tac [("M'1","[IsEq|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
117 force 4;
118 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
119 by (res_inst_tac [("M'1","[IsEq|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
120 force 4;
121 by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
122 by (res_inst_tac [("M'1","[IsEq|Ma|[N|nil]]")] (env_lemma6 RS mp) 4);
123 force 4;
124 force 3;
125
126 (** IsAtom case **)
127 br allI 3;
128 br impI 3;
129 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
130 by (res_inst_tac [("M'1","[IsAtom|Ma|nil]")] (env_lemma6 RS mp) 4);
131 force 4;
132 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
133 by (res_inst_tac [("M'1","[IsAtom|Ma|nil]")] (env_lemma6 RS mp) 4);

```

```

134 force 4;
135 force 3;
136
137 (** Error case **)
138 br allI 3;
139 br impI 3;
140 by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
141 by (res_inst_tac [("M'1","[Error|[Ma|nil]]")](env_lemma6 RS mp) 4);
142 force 4;
143 by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
144 by (res_inst_tac [("M'1","[Error|[Ma|nil]]")](env_lemma6 RS mp) 4);
145 force 4;
146 force 3;
147
148 (** Lam case **)
149 by (SELECT_GOAL Auto_tac 2);
150
151 by (swap_res_tac lmixenv.intrs 2);
152 force 2;
153 by (eres_inst_tac [("a","<[Lam|[Atom a|[Ma|nil]]],vlist # \
154 \ [v';v;Mv',vlist']>")](lmixenv.elim 2);
155 force 2;
156 by (dtac sym 2);
157 by (Asm_full_simp_tac 2);
158 by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,vlist # [v';v;Mv,vlist']>")](
159     lmixenv.elim 2);
160 force 2;
161 by (dtac sym 2);
162 by (Asm_full_simp_tac 2);
163
164 by (swap_res_tac lmixrlist.intrs 2);
165 force 2;
166 force 2;
167 force 2;
168 force 2;
169 force 3;
170 force 3;
171 by (subgoal_tac "[v';v;Mv',vlist'] : lmixrlist" 2);
172 br ((lconcat_lemma1 RS mp) RS conjE) 3;
173 force 3;
174 force 3;
175 by (thin_tac "?xx" 2);
176 by (rotate_tac 1 2);
177 by (thin_tac "?xx" 2);
178 by (rotate_tac 1 2);
179 by (thin_tac "?xx" 2);
180 by (thin_tac "?xx" 2);
181 by (thin_tac "?xx" 2);
182 by (thin_tac "?xx" 2);
183 by (thin_tac "?xx" 2);
184 by (thin_tac "?xx" 2);
185 by (thin_tac "?xx" 2);
186 by (rotate_tac 1 2);
187 by (thin_tac "?xx" 2);
188 by (SELECT_GOAL Auto_tac 2);
189 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
190 force 2;
191

```

```

192 by (subgoal_tac "<Ma,[v'a;a;Ma,vlist # [v';v;Mv',vlist']]> : lmixenv" 2);
193 by (defer_tac 2);
194
195 by (swap_res_tac lmixenv.intrs 2);
196 force 2;
197 by (eres_inst_tac [("a","<[Lam|[Atom a|[Ma|nil]]],vlist # \
198 \ [v';v;Mv',vlist']>")] lmixenv.elim 2);
199 force 2;
200 by (dtac sym 2);
201 by (Asm_full_simp_tac 2);
202 by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,vlist # [v';v;Mv',vlist']]>")]
203   lmixenv.elim 2);
204 force 2;
205 by (dtac sym 2);
206 by (Asm_full_simp_tac 2);
207
208 by (swap_res_tac lmixrlist.intrs 2);
209 force 2;
210 force 2;
211 force 2;
212 force 2;
213 force 3;
214 force 3;
215 by (subgoal_tac "[v';v;Mv',vlist'] : lmixrlist" 2);
216 br ((lconcat_lemma1 RS mp) RS conjE) 3;
217 force 3;
218 force 3;
219 by (thin_tac "?xx" 2);
220 by (rotate_tac 1 2);
221 by (thin_tac "?xx" 2);
222 by (rotate_tac 1 2);
223 by (thin_tac "?xx" 2);
224 by (thin_tac "?xx" 2);
225 by (thin_tac "?xx" 2);
226 by (thin_tac "?xx" 2);
227 by (thin_tac "?xx" 2);
228 by (thin_tac "?xx" 2);
229 by (thin_tac "?xx" 2);
230 by (rotate_tac 1 2);
231 by (thin_tac "?xx" 2);
232 by (SELECT_GOAL Auto_tac 2);
233 by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
234 force 2;
235
236 by (eres_inst_tac [("x","[v'a;a;Ma,vlist]")] allE 2);
237 force 2;
238
239 (** Var case **)
240 br allI 1;
241 by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
242 by (Asm_full_simp_tac 1);
243 by (SELECT_GOAL Auto_tac 1);
244 by (case_tac "a = v" 1);
245 force 1;
246 br derivelimvar1 1;
247 force 1;
248 force 1;
249

```



```
250 br impI 1;
251 br allI 1;
252 br impI 1;
253 by (REPEAT (etac conjE 1));
254 by (case_tac "a = atom2" 1);
255 by (Asm_full_simp_tac 1);
256 by (res_inst_tac [("v2","atom2"),("v'2","atom1")]
257      ((evaluation_varnonemp RS mp) RS iffD1) 1);
258 force 1;
259 force 1;
260 by (Asm_full_simp_tac 1);
261 br conjI 1;
262 br (lmered RS mp) 1;
263 force 1;
264 by (subgoal_tac "<Atom a,rlist # [v';v;Mv,vlist']> : lmixenv" 1);
265 br (lmered RS mp) 2;
266 force 2;
267 by (subgoal_tac "<Atom a,rlist # [v';v;Mv',vlist']> : lmixenv" 1);
268 br (lmered RS mp) 2;
269 force 2;
270 auto();
271 by (eres_inst_tac [("x","dM")] allE 1);
272 br derivelimvar1 1;
273 force 1;
274 force 1;
275 qed "lem_weak";
```

Bibliography

- [1] ANDERSEN, L. O. Correctness proof for a self-interpreter. DIKU technical report, DIKU, Department of Computer Science, University of Copenhagen, Dec. 1991.
- [2] BONDORF, A. *Simlix 5.0 Manual*. DIKU, Department of Computer Science, University of Copenhagen, May 1993.
- [3] CONSEL, C. *Report on Schism*, Jan. 1996.
- [4] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages* (1982), ACM Press, pp. 207–212.
- [5] FUTAMURA, Y. Partial evaluation of computing process—an approach to a compiler-compiler. *Systems, Computers, Controls* 2, 5 (1971), 45–50.
- [6] GOMARD, C. K. Higher Order Partial Evaluation—HOPE for the lambda calculus. Master’s thesis, DIKU, Department of Computer Science, University of Copenhagen, Sept. 1989.
- [7] GOMARD, C. K. Partial type inference for untyped functional programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (Nice, France, June 1990). Extended abstract.
- [8] GOMARD, C. K. *Program Analysis Matters*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1991. DIKU Technical Report 91/17.
- [9] GOMARD, C. K. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems* 14, 2 (Apr. 1992), 147–172.
- [10] GOMARD, C. K., AND JONES, N. D. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming* 1, 1 (Jan. 1991), 21–69.
- [11] HATCLIFF, J. Mechanically verifying the correctness of an offline partial evaluator. In *Programming Languages: Implementations, Logics and Programs*, M. Hermenegildo and D. Doaitse Swierstra, Eds., vol. 982 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995, pp. 279–298.
- [12] JONES, N. D. Challenging problems in partial evaluation and mixed computation. In *Partial Evaluation and Mixed Computation* (1988), D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., North-Holland.
- [13] JONES, N. D. What *not* to do when writing an interpreter for specialization. In *Partial Evaluation. Selected Papers*, O. Danvy, R. Glück, and P. Thiemann, Eds., vol. 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996, pp. 216–237.

- [14] JONES, N. D. *Computability and Complexity: From a Programming Perspective*. Foundations of Computing. The MIT Press, 1997.
- [15] JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, 1993.
- [16] MOGENSEN, T. $\mathcal{A}\mathcal{E}$. Self-applicable partial evaluation for pure lambda calculus. In *Proceedings of PEPM '92* (1992), Yale University Press, pp. 116–121.
- [17] MOGENSEN, T. $\mathcal{A}\mathcal{E}$. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of PEPM '95* (1995), ACM Press, pp. 39–44.
- [18] MOGENSEN, T. $\mathcal{A}\mathcal{E}$. Inherited limits. In *Partial Evaluation: Practice and Theory* (Copenhagen, July 1998), J. Hatcliff, T. $\mathcal{A}\mathcal{E}$. Mogensen, and P. Thiemann, Eds., pp. 1–10.
- [19] NARASCHEWSKI, W., AND NIPKOW, T. Type inference verified: Algorithm \mathcal{W} in Isabelle/HOL. *Journal of Automated Reasoning* (1999). To appear.
- [20] NEDERPELT, R. P., GEUVERS, J. H., AND DE VRIJER, R. C., Eds. *Selected Papers on Automath*, vol. 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.
- [21] NIPKOW, T. *Isabelle/HOL: The Tutorial*. Institut für Informatik, Technische Universität München, Oct. 1998. Draft.
- [22] OWRE, S., SHANKAR, N., RUSHBY, J. M., AND STRINGER-CALVERT, D. W. J. *PVS Language Reference*. SRI International, 1998.
- [23] OWRE, S., SHANKAR, N., RUSHBY, J. M., AND STRINGER-CALVERT, D. W. J. *PVS System Guide*. SRI International, 1998.
- [24] PAULSON, L. C. *Introduction to Isabelle*. University of Cambridge, Computer Laboratory, 1998.
- [25] PAULSON, L. C. *The Isabelle Reference Manual*. University of Cambridge, Computer Laboratory, 1998.
- [26] PAULSON, L. C. *Isabelle's Object-Logics*. University of Cambridge, Computer Laboratory, 1998.
- [27] PFENNING, F., AND ELLIOTT, C. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988), pp. 199–208.
- [28] SHANKAR, N., OWRE, S., RUSHBY, J. M., AND STRINGER-CALVERT, D. W. J. *PVS Prover Guide*. SRI International, 1998.
- [29] STRINGER-CALVERT, D. W. J. *Mechanical Verification of Compiler Correctness*. PhD thesis, Department of Computer Science, University of York, Mar. 1998.
- [30] WELINDER, M. *Partial Evaluation and Correctness*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996. DIKU Technical Report 98/13.
- [31] WENZEL, M. *Using Axiomatic Type Classes in Isabelle: a Tutorial*. University of Cambridge, Computer Laboratory, Oct. 1998.