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 untyped 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 Skakkebæk 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.

Sebastian C. Skalberg

Contents

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 refered 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

$$[\![p]\!](s, d) = [\![[\![mix]\!](p, s)]\!](d) \tag{2.1.1}$$

for all programs p and inputs s and d. The equation says that if we divide the input of propgram p in a *static* part s and *dynamic* part d, we can computably construct a new program $p_s = [\![mix]\!](p, s)$. 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. $[\![p_s]\!] = [\![p]\!](s, \cdot)$.

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

$$\begin{aligned} [\![mix]\!](int, p) &= p' && (2.1.2)\\ [\![mix]\!](mix, int) &= comp && (2.1.3)\\ [\![mix]\!](mix, mix) &= cogen && (2.1.4) \end{aligned}$$

Actually implementing these projections is difficult. It was not until 1984 that Neil Jones, Peter Sestoft, and Harald Søndergaard succeded 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 propogate 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 occurences 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 $\mathrm{T}_{p_s}(d) \leq \mathrm{T}_{p'_s}(d)$, where $p_s = [\![mix]\!](p,s)$ and $p'_s = [\![mix']\!](p,s)$, and where $\mathrm{T}_p(d)$ 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 intepretational 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*

$$\mathrm{T}_{sint_p}(d) \leq \mathrm{T}_p(d),$$

where $sint_p = [\![mix]\!](sint, p)$.

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 where 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 evalutor 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 succeded 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. $\texttt{a} = \texttt{a}$ and $\texttt{(a.b)} = \texttt{(a.b)}$, but $\texttt{(a.b)} \neq \texttt{(b.a)}$. 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 $d_1, \ldots, d_n$ of S-expressions is coded as the S-expression $(d_1 \,.\, (d_2 \,.\, (\ldots(d_n \,.\, \texttt{nil})\ldots)$ which is by notational convention written as

$$(d_1 \;\; d_2 \;\; \cdots \;\; d_n).$$

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*.

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

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:

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

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 $\mathbb{A} \backslash \mathbb{S}_2$, are called *variables*, and the

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

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:

[1] Strictly 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 $vars : \mathbb{D}_{\mathbb{A}} \to \mathcal{P}(\mathbb{V})$ is given by*

$$\begin{array}{rcl} vars\ v & = & \{v\} \cap \mathbb{V} \\ vars\ (d_1 \,.\, d_2) & = & vars\ d_1 \cup vars\ d_2 \end{array}$$

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

$$vars\ (d_1\ \ d_2\ \ \cdots\ \ d_n) = vars\ d_1 \cup vars\ d_2 \cup \cdots \cup vars\ d_n$$

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*

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

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':*

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

$$
\begin{array}{rcl}
\mathbb{L}_2 & ::= & \mathbb{L}_1 \\
& | & \texttt{(lift } \mathbb{L}_1\texttt{)} \\
& | & \texttt{(}\underline{\texttt{quote}}\ \mathbb{D}\texttt{)} \\
& | & \texttt{(}\underline{\texttt{lam}}\ \mathbb{V}\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{@}}\ \mathbb{L}_2\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{fix}}\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{if}}\ \mathbb{L}_2\ \mathbb{L}_2\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{cons}}\ \mathbb{L}_2\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{car}}\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{cdr}}\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{eq?}}\ \mathbb{L}_2\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{atom?}}\ \mathbb{L}_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{error}}\ \mathbb{L}_2\texttt{)}
\end{array}
$$

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 $v \notin vars\ M$ in the second line. In effect, this means that

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

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 $\texttt{((}x\,\texttt{.}\,d_x\texttt{)}\ \texttt{(}y\,\texttt{.}\,d_y\texttt{))}$. The lookup function is then defined as:

Definition 3.3.1 *The partial map* $lookup : \mathbb{D}_{\mathbb{A}} \times \mathbb{V} \to \mathbb{D}_{\mathbb{A}}$ *is given by*

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

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 seperately 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 orginal semantics: Consider the expression

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

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 $\texttt{(fix (lam } v\ M\texttt{))}$.

The obvious solution is to make the variable explicit, eg. $\texttt{(fix } f\ M\texttt{)}$. 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 $\texttt{(fix } M\texttt{)}$ 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 lanugauge, 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

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

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.

$$\frac{\mathit{lookup}(\rho, v) = d \quad d \in \mathbb{A} \vee (d = (d_1 \, \texttt{.} \, d_2) \wedge d_1 \neq \texttt{delay})}{\rho \vdash v \longrightarrow d} \tag{3.3.1}$$

$$\frac{\mathit{lookup}(\rho, v) = \texttt{(delay}\ M\ \rho'\texttt{)} \quad \rho' \vdash M \longrightarrow d}{\rho \vdash v \longrightarrow d} \tag{3.3.2}$$

$$\frac{}{\rho \vdash \texttt{(quote}\ d\texttt{)} \longrightarrow d} \tag{3.3.3}$$

$$\frac{}{\rho \vdash \texttt{(lam}\ v\ M\texttt{)} \longrightarrow \texttt{(clos}\ v\ M\ \rho\texttt{)}} \tag{3.3.4}$$

$$\frac{\rho \vdash M \longrightarrow \texttt{(clos}\ v\ M'\ \rho'\texttt{)} \quad \rho \vdash N \longrightarrow d_N \quad \texttt{((}v\,\texttt{.}\,d_N\texttt{)}\,\texttt{.}\,\rho'\texttt{)} \vdash M' \longrightarrow d}{\rho \vdash \texttt{(@}\ M\ N\texttt{)} \longrightarrow d} \tag{3.3.5}$$

$$\frac{\rho \vdash M \longrightarrow \texttt{(clos}\ v\ M'\ \rho'\texttt{)} \quad \texttt{((}v\texttt{.(delay (fix}\ M\texttt{)}\ \rho\texttt{)).}\rho'\texttt{)} \vdash M' \longrightarrow d}{\rho \vdash \texttt{(fix}\ M\texttt{)} \longrightarrow d} \tag{3.3.6}$$

$$\frac{\rho \vdash M \longrightarrow d_M \quad d_M \neq \texttt{\#t} \quad \rho \vdash P \longrightarrow d_P}{\rho \vdash \texttt{(if}\ M\ N\ P\texttt{)} \longrightarrow d_P} \tag{3.3.7}$$

$$\frac{\rho \vdash M \longrightarrow \texttt{\#t} \quad \rho \vdash N \longrightarrow d_N}{\rho \vdash \texttt{(if}\ M\ N\ P\texttt{)} \longrightarrow d_N} \tag{3.3.8}$$

$$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N}{\rho \vdash \texttt{(cons}\ M\ N\texttt{)} \longrightarrow \texttt{(}d_M\,\texttt{.}\,d_N\texttt{)}} \tag{3.3.9}$$

$$\frac{\rho \vdash M \longrightarrow \texttt{(}d_1\,\texttt{.}\,d_2\texttt{)}}{\rho \vdash \texttt{(car}\ M\texttt{)} \longrightarrow d_1} \tag{3.3.10}$$

$$\frac{\rho \vdash M \longrightarrow \texttt{(}d_1\,\texttt{.}\,d_2\texttt{)}}{\rho \vdash \texttt{(cdr}\ M\texttt{)} \longrightarrow d_2} \tag{3.3.11}$$

$$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N \quad d_M \neq d_N}{\rho \vdash \texttt{(eq?}\ M\ N\texttt{)} \longrightarrow \texttt{nil}} \tag{3.3.12}$$

$$\frac{\rho \vdash M \longrightarrow d_M \quad \rho \vdash N \longrightarrow d_N \quad d_M = d_N}{\rho \vdash \texttt{(eq?}\ M\ N\texttt{)} \longrightarrow \texttt{\#t}} \tag{3.3.13}$$

$$\frac{\rho \vdash M \longrightarrow d_M \quad d_M \in \mathbb{A}}{\rho \vdash \texttt{(atom?}\ M\texttt{)} \longrightarrow \texttt{\#t}} \tag{3.3.14}$$

$$\frac{\rho \vdash M \longrightarrow \texttt{(}d_1\,\texttt{.}\,d_2\texttt{)}}{\rho \vdash \texttt{(atom?}\ M\texttt{)} \longrightarrow \texttt{nil}} \tag{3.3.15}$$

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

$$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\texttt{lift}\ \ M) \longrightarrow (\texttt{quote}\ \ d_M)} \tag{3.3.16}$$

$$\frac{}{\rho \vdash (\underline{\texttt{quote}}\ \ d) \longrightarrow (\texttt{quote}\ \ d)} \tag{3.3.17}$$

$$\frac{v_{\text{new}} \in \mathbb{V}\backslash(\mathit{vars}\ \rho \cup \mathit{vars}\ M) \qquad ((v\,\texttt{.}\,v_{\text{new}})\,\texttt{.}\,\rho) \vdash M \longrightarrow d_M}{\rho \vdash (\underline{\texttt{lam}}\ \ v\ \ M) \longrightarrow (\texttt{lam}\ v_{\text{new}}\ \ d_M)} \tag{3.3.18}$$

$$\frac{\rho \vdash M \longrightarrow d_M \qquad \rho \vdash N \longrightarrow d_N}{\rho \vdash (\underline{\texttt{@}}\ \ M\ \ N) \longrightarrow (\texttt{@}\ d_M\ \ d_N)} \tag{3.3.19}$$

$$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\underline{\texttt{fix}}\ \ M) \longrightarrow (\texttt{fix}\ \ d_M)} \tag{3.3.20}$$

$$\frac{\rho \vdash M \longrightarrow d_M \qquad \rho \vdash N \longrightarrow d_N \qquad \rho \vdash P \longrightarrow d_P}{\rho \vdash (\underline{\texttt{if}}\ \ M\ \ N\ \ P) \longrightarrow (\texttt{if}\ \ d_M\ \ d_N\ \ d_P)} \tag{3.3.21}$$

$$\frac{\rho \vdash M \longrightarrow d_M \qquad\qquad \rho \vdash N \longrightarrow d_N}{\rho \vdash (\underline{\texttt{cons}}\ \ M\ \ N) \longrightarrow (\texttt{cons}\ \ d_M\ \ d_N)} \tag{3.3.22}$$

$$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\underline{\texttt{car}}\ \ M) \longrightarrow (\texttt{car}\ \ d_M)} \tag{3.3.23}$$

$$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\underline{\texttt{cdr}}\ \ M) \longrightarrow (\texttt{cdr}\ \ d_M)} \tag{3.3.24}$$

$$\frac{\rho \vdash M \longrightarrow d_M \qquad\qquad \rho \vdash N \longrightarrow d_N}{\rho \vdash (\underline{\texttt{eq?}}\ \ M\ \ N) \longrightarrow (\texttt{eq?}\ \ d_M\ \ d_N)} \tag{3.3.25}$$

$$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\underline{\texttt{atom?}}\ \ M) \longrightarrow (\texttt{atom?}\ \ d_M)} \tag{3.3.26}$$

$$\frac{\rho \vdash M \longrightarrow d_M}{\rho \vdash (\underline{\texttt{error}}\ \ M) \longrightarrow (\texttt{error}\ \ d_M)} \tag{3.3.27}$$

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

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

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

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

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 effecient 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 $sint_{\mathrm{ann}}$ 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*

$$\texttt{nil} \vdash \texttt{(@ (@ } sint_{\mathrm{ann}} \texttt{ (quote } M\texttt{)) (lam x x))} \longrightarrow d_M.$$

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

$$\texttt{nil} \vdash \texttt{(@ (@ } sint_{\mathrm{ann}} \texttt{ (quote } M\texttt{)) (lam x x))} \longrightarrow d_M \Longrightarrow M =_\alpha d_M.$$

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 totallity, see (Jones [13]). Regarding uniqueness, by inspection of the annotated self-interpreter $sint_{\mathrm{ann}}$ 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 $sint$ 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

```
(lam expr (lam env expr))
```

also terminates with a term α-equivalent to M when applied to `(quote` M`)` and `(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

$$\rho \vdash M \longrightarrow d \quad \Longleftrightarrow \quad \rho' \vdash M' \longrightarrow d'$$

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

$$sint_{\mathrm{ann}} = \texttt{(fix (lam eval (lam expr (lam env } body_{\mathrm{ann}}\texttt{)))).}$$

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

$$\begin{array}{rcl}
\texttt{(}cadr\ \ M\texttt{)} & = & \texttt{(car (cdr } M\texttt{))} \\
\texttt{(}caddr\ \ M\texttt{)} & = & \texttt{(car (cdr (cdr } M\texttt{)))} \\
\texttt{(}cadddr\ \ M\texttt{)} & = & \texttt{(car (cdr (cdr (cdr } M\texttt{))))} \\
env_\lambda & = & \texttt{(lam var (if (eq? var (}cadr\texttt{ expr)) value (@ env var)))}
\end{array}$$

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

$$\texttt{nil} \vdash \texttt{(@ (@ } \mathit{sint}_{\mathrm{ann}} \texttt{ (quote } M\texttt{)) (lam x x))} \longrightarrow d_M \tag{4.2.1}$$

on the form $\rho \vdash \mathit{body}_{\mathrm{ann}} \longrightarrow d$. 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

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

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 $\mathit{body}_{\mathrm{ann}}$, evaluates the $\mathbb{L}_1$-expression M using no variable renamings (hence the "$[]$").

As a second insight, we examine the derivation of

$$\langle \texttt{(lam } v \ \ M\texttt{)}, []\rangle \vdash \mathit{body}_{\mathrm{ann}} \longrightarrow d \tag{4.2.2}$$

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

$$\begin{array}{c}\texttt{((env.(clos var (if (eq? var (}\mathit{cadr}\texttt{ expr)) value (@ env var))} \\ \texttt{((value.}v'\texttt{).}\langle\texttt{(lam } v \ \ M\texttt{)}, []\rangle\texttt{)))) (expr.}M\texttt{) (eval.(delay } \mathit{sint}_{\mathrm{ann}} \texttt{ nil)))} \\ \vdash \mathit{body}_{\mathrm{ann}} \longrightarrow d\end{array} \tag{4.2.3}$$

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

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

We will denote the environment `((env.(clos` $\cdots$`))` $\cdots$`(delay` $\mathit{sint}_{\mathrm{ann}}$ `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

$$\langle M, [v_1 := v_1'|M_1] \cdots [v_n := v_n'|M_n]\rangle \vdash \mathit{body}_{\mathrm{ann}} \longrightarrow d$$

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

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

but not

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

It remains to say a bit about the M_is 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 $\mathbf{K}$-combinator, `(lam x (lam y x))`[1]:

$$\begin{array}{rl} & \langle\texttt{(lam x (lam y x))}, []\rangle \vdash \mathit{body}_{\mathrm{ann}} \longrightarrow \texttt{(lam } x_{\mathrm{new}} \texttt{ (lam } y_{\mathrm{new}} \ x_{\mathrm{new}}\texttt{))} \\ \Longleftrightarrow & \langle\texttt{(lam y x)}, [x := x_{\mathrm{new}}|\texttt{(lam y x)}]\rangle \vdash \mathit{body}_{\mathrm{ann}} \longrightarrow \texttt{(lam } y_{\mathrm{new}} \ x_{\mathrm{new}}\texttt{)} \\ \Longleftrightarrow & \langle\texttt{x}, [y := y_{\mathrm{new}}|\texttt{x}][x := x_{\mathrm{new}}|\texttt{(lam y x)}]\rangle \vdash \mathit{body}_{\mathrm{ann}} \longrightarrow x_{\mathrm{new}}\end{array}$$

[1]We 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

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

ii. If

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

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

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

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

$$\begin{array}{l}\texttt{((env.(clos var}\\ \quad\texttt{(if (eq? var (}\mathit{cadr}\texttt{ expr)) value (@ env var))}\\ \quad\texttt{((value.}v'\texttt{).}\langle\texttt{(lam } v \; M_v\texttt{)}, vl\rangle\texttt{)))}\\ \;\texttt{(expr.}M\texttt{)}\\ \;\texttt{(eval.(delay } \mathit{sint}_{\text{ann}} \texttt{ nil)))}\end{array}$$

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*

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

if and only if

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

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

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

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

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

and for all $i > 0$:

$$\begin{array}{rcl} v_i & \in & \mathit{vars}\ M_{i-1} \\ v_i' & \notin & \{v_1, \ldots, v_i, v_1', \ldots, v_{i-1}'\} \end{array}$$

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*

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

Proof. By derivation. □

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

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

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*

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

if and only if

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

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

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

By a further derivation, we can deduce

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

if and only if

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

But now we are finished, since

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

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 $\langle\texttt{(quote }d\texttt{)}, vl\rangle \in \mathbb{E}$ we have*

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

Proof. By derivation. □

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

$$\langle\texttt{(lam }v\ \ M\texttt{)}, vl\rangle \vdash \mathit{body}_{\text{ann}} \longrightarrow d$$

if and only if there exists a variable v' and an S-expression d_M such that *i.* $d = \texttt{(lam}\ v'\ d_M\texttt{)}$,

ii. $\langle M, [v := v'|M]vl\rangle \in \mathbb{E}$, *and*

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

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. $v' \notin \langle\texttt{(lam}\ v\ \ M\texttt{)}, vl\rangle$. From the generalization of rule 4.2.4 on page 15, ie.

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

it follows immediately. □

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

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

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

i. $d' = \texttt{(@}\ d_M\ \ d_N\texttt{)}$,

ii. $\langle M, vl\rangle \vdash body_{\text{ann}} \longrightarrow d_M$, *and*

iii. $\langle N, vl\rangle \vdash body_{\text{ann}} \longrightarrow d_N$.

Proof. By derivation. □

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

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

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

i. $d' = \texttt{(fix}\ \ d_M\texttt{)}$ *and*

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

Proof. By derivation. □

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

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

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

i. $d' = \texttt{(if}\ \ d_M\ \ d_N\ \ d_P\texttt{)}$,

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

iii. $\langle N, vl\rangle \vdash body_{\text{ann}} \longrightarrow d_N$, *and*

iv. $\langle P, vl\rangle \vdash body_{\text{ann}} \longrightarrow d_P$.

Proof. By derivation. □

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

$$\langle\texttt{(cons}\ \ M\ \ N\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d'$$

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

i. $d' = \texttt{(cons}\ \ d_M\ \ d_N\texttt{)}$,

ii. $\langle M, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_M$, and

iii. $\langle N, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_N$.

Proof. By derivation. □

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

$$\langle\texttt{(car}\ \ M\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d'$$

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

i. $d' = \texttt{(car}\ \ d_M\texttt{)}$ *and*

ii. $\langle M, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_M$.

Proof. By derivation. □

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

$$\langle\texttt{(cdr}\ \ M\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d'$$

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

i. $d' = \texttt{(cdr}\ \ d_M\texttt{)}$ *and*

ii. $\langle M, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_M$.

Proof. By derivation. □

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

$$\langle\texttt{(eq?}\ \ M\ \ N\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d'$$

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

i. $d' = \texttt{(eq?}\ \ d_M\ \ d_N\texttt{)}$,

ii. $\langle M, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_M$, and

iii. $\langle N, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_N$.

Proof. By derivation. □

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

$$\langle\texttt{(atom?}\ \ M\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d'$$

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

i. $d' =$ (atom? d_M) *and*

ii. $\langle M, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_M$.

Proof. By derivation. □

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

$$\langle \texttt{(error}\ \ M\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d'$$

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

i. $d' =$ (error d_M) *and*

ii. $\langle M, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_M$.

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 se shortly, is fairly easy to prove, the Uniqueness theorem demands a bit more by way of technical lemmata.

4.4.1 Termination

That $sint_{\mathrm{ann}}$ 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*

$$\langle M, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_M.$$

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* $M \equiv$ (quote d). Follows directly from the Quote Evaluation lemma.

iii. *Case* $M \equiv$ (lam v P). Choose a variable v' with $v' \notin vars\ \langle$(lam v P)$, vl\rangle$. (This can be done since the number of variables in $\langle$(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

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

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

$$\langle \texttt{(lam}\ \ v\ \ P\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow \texttt{(lam}\ \ v'\ \ d_P\texttt{)}$$

as wanted.

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

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

$$\langle P, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_P$$

and

$$\langle Q, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow d_Q.$$

By the Application Evaluation lemma, we have

$$\langle \texttt{(@}\ P\ \ Q\texttt{)}, vl\rangle \vdash body_{\mathrm{ann}} \longrightarrow \texttt{(@}\ d_P\ \ d_Q\texttt{)}$$

as wanted.

This concludes the proof. □

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

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* $vars\ M' \subseteq vars\ M$ *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. $\langle \texttt{(lam}\ v\ \ M_v\texttt{)}, vl'\rangle \in \mathbb{E}$,

ii. $vars\ M \subseteq vars\ M_v$, and

iii. $v' \notin vars\ \langle \texttt{(lam}\ v\ \ M_v\texttt{)}, vl'\rangle$.

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

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 $\langle \texttt{(lam}\ v\ \ M_v\texttt{)}, [y := y'|M_y]vl_2'\rangle \in \mathbb{E}$, and hence $\langle \texttt{(lam}\ y\ \ M_y\texttt{)}, vl_2'\rangle \in \mathbb{E}$ and $y' \notin vars\ \langle \texttt{(lam}\ y\ \ M_y\texttt{)}, vl_2'\rangle$. Finally, $vars\ M \subseteq vars\ M_v \subseteq vars\ M_y$ 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) $\langle \texttt{(lam}\ y\ \ M_y\texttt{)}, vl_1' vl_2\rangle \in \mathbb{E}$,

(b) $vars\ M \subseteq vars\ M_y$, and

(c) $y' \notin vars\ \langle \texttt{(lam}\ y\ \ M_y\texttt{)}, vl_1' vl_2\rangle$.

From the assumption and the definition of lambda-mix environments, we get $vars\ M \subseteq vars\ M_y$, which takes care of case iib. Further, we get $\langle(\texttt{lam}\ y\ \ M_y)\texttt{,}\ vl_1'[v := v'|M_v]vl_2\rangle \in \mathbb{E}$. Using the induction hypothesis, this gives us case iia. Finally, we have

$$y' \notin \langle(\texttt{lam}\ y\ \ M_y)\texttt{,}\ vl_1'[v := v'|M_v]vl_2\rangle$$

which together with the observation

$$vars\ \langle(\texttt{lam}\ y\ \ M_y)\texttt{,}\ vl_1'vl_2\rangle \subseteq vars\ \langle(\texttt{lam}\ y\ \ M_y)\texttt{,}\ vl_1'[v := v'|M_v]vl_2\rangle$$

gives us case iic.

This completes the proof. □

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 $\langle M, [v := v'|M_v]vl\rangle \vdash body_{\text{ann}} \longrightarrow d$ implies $\langle M[v := v'], vl\rangle \vdash body_{\text{ann}} \longrightarrow d$. 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 $vars\ M[v := v'] \subseteq vars\ M_y$. When $v' \in vars\ M[v := v']$, this cannot hold since v' was specifically chosen such that $v' \notin vars\ \langle(\texttt{lam}\ v\ \ M_v)\texttt{,}\ vl\rangle$, and so in particular $v' \notin vars\ M_y$.

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 $vars\ (M \uplus v) = vars\ M \cup \{v\}$. 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. $(\texttt{@}\ M\ \ v)$.

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) $\langle(\texttt{lam}\ y\ \ M_y \uplus v')\texttt{,}\ []\rangle \in \mathbb{E}$,

   (b) $vars\ M[v := v'] \subseteq vars\ M_y \uplus v'$, and

   (c) $y' \notin \langle(\texttt{lam}\ y\ \ M_y \uplus v')\texttt{,}\ []\rangle \in \mathbb{E}$.

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

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

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

   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. □

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* $\langle M, [v := v'|M_v]vl\rangle \in \mathbb{E}$,

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

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

$$\langle y, vl\rangle \vdash body_{\text{ann}} \longrightarrow d' \quad \Longrightarrow \quad \langle y, vl \uplus v'\rangle \vdash body_{\text{ann}} \longrightarrow d'$$

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 $\langle y, [z := z'|M_z \uplus v']vl'\rangle \vdash body_{\text{ann}} \longrightarrow d'$, as wanted.

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

This completes the proof. □

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* $v \notin \{v_1, \ldots, v_n\}$, *we have*

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

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. □

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*

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

then

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

and

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

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*

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

then

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

and

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

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*

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

then

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

and

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

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*

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

then

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

and

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

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

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

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

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

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 $v' \notin vars\ \langle(\texttt{lam}\ v\ \ M_v), vl\rangle$, 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* $M \equiv (\texttt{quote}\ \ d)$. The Quote Evaluation lemma gives us $d_M = (\texttt{quote}\ \ d)$. Another application of the same lemma gives us

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

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

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

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

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

(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

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

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

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

Using the Application Evaluation lemma, we get $d_M = \texttt{(@}\ d_P\ \ d_Q\texttt{)}$ 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

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

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

This concludes the proof of the Renaming lemma. □

Finally, we are able to prove the following lemma:

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

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

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

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

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* $M \equiv$ (quote d). By the Quote Evaluation lemma, we have $d_M =$ (quote d), which is trivially α-equivalent to (quote d).

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

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

Using the Renaming lemma, we get

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

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

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

as wanted.

iv. *Cases* $M \equiv$ (@ P Q), $M \equiv$ (fix P), $M \equiv$ (if P Q R), $M \equiv$ (cons P Q), $M \equiv$ (car P), $M \equiv$ (cdr P), $M \equiv$ (eq? P Q). $M \equiv$ (atom? P), *and* $M \equiv$ (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

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

and

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

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

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

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

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

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

$$\mathtt{clos} \quad \notin \quad \{\mathtt{delay}, \mathtt{nil}, \mathtt{\#t}, \mathtt{lam}, \ldots\}$$

$$\begin{array}{rcl} \texttt{delay} & \notin & \{\texttt{clos}, \texttt{nil}, \texttt{\#t}, \texttt{lam}, \ldots\} \\ \texttt{nil} & \notin & \{\texttt{clos}, \texttt{delay}, \texttt{\#t}, \texttt{lam}, \ldots\} \\ & \vdots & \end{array}$$

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

$$\begin{array}{rcl} \texttt{nil} & = & 0 \\ \texttt{clos} & = & 1 \\ & \vdots & \end{array}$$

etc. to ensure that the atoms found in $\mathbb{S}_2$ and $\mathbb{V}_{\mathrm{sint}}$ 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 $\mathbb{V}_{\mathrm{sint}}$ (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 "$\texttt{[}d_1\texttt{|}d_2\texttt{]}$" 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`-a-`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]

[1] We 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 techincally 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 : l1expr ==> EX dM : l1expr.
              (nil |- [App|[[App|[sintann|[[Quote|[M|nil]]|nil]]]|
                      [[Lam|[x|[x|nil]]]|nil]]] ---> dM)
```

Goal 5.2.2 (Theorem 3.5.2, Uniqueness)

```
M : l1expr ==> (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 |- [RIsAtom|[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 us 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 `<`M`,` vl_1`#`vl_2`>`.

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 evey 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 succeded 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; ··· ; Pn |] ==> P
```

abbreviates

```
P1 ==> (P2 ==> (··· ==> (Pn ==> P) ···))
```

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, ..., 'an] => a
```

abbreviates

```
'a1 => ('a2 => ... ('an => a) ···)
```

`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_isone "x == 1"
```

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

`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 --> Q`	`P` implies `Q`
`A = B`	`A` is equal to `B`
`A ~= B`	`A` is not equal to `B`
`a : B`	`a` is a member of `B`
`a ~: B`	`a` is not a member of `B`
`A <= B`	`A` is a subset of B
`? a. P` or `EX a. P`	there exists `a` such that `P`
`! a. P` or `ALL a. P`	for all `a`, `P`
`? a : B. P` or `EX a : B. P`	there exists `a` from `B` such that `P`
`! a : B. P` or `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 occurences 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

```
(*********************************************************************
*
* LMixEnv.thy
*
*********************************************************************)

LMixEnv = Sint + L2Eval +

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

consts
        lmixenv   :: sexpr set
        lmixrlist :: rlist set
```

```
        varsin    :: rlist => (atom set)
        lenv      :: rlist => sexpr
        lwrap     :: [sexpr,rlist] => sexpr ("<_,/ _>")
        lconcat   :: [rlist,rlist] => rlist ("_ #/  _")

primrec
        varsin_none "varsin [] = {}"
        varsin_nemp "varsin [v’;v;M,vlist] = {v’,v} Un (varsin vlist)"

primrec
        lconcat_none "lconcat [] vlist = vlist"
        lconcat_nemp "lconcat [v’;v;M,vlist’] vlist =
                [v’;v;M,lconcat vlist’ vlist]"

defs
        lwrap_def "lwrap M vlist == [[env|(lenv vlist)]|[[expr|M]|
                        [[eval|[delay|[sintann|[nil|nil]]]]|nil]]]"

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]]]]"

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"

end
```

```
(**********************************************************************
*
* LMixEnv.ML
*
**********************************************************************)

Addsimps lmixrlist.intrs;
AddIs lmixrlist.intrs;

Addsimps lmixenv.intrs;
AddIs lmixenv.intrs;

val lmrelims = map (lmixrlist.mk_cases rlist.simps)
[
        "[] : lmixrlist",
        "[v’;v;M,vlist] : lmixrlist"
];
```

```
AddSEs lmrelims;

AddSEs lmixenv.elims;

val [prem1,prem2,prem3] =
Goal "[|P ==> S; [|P ; S|] ==> R ; P|] ==> R";
br prem2 1;
br prem1 2;
br prem3 1;
br prem3 1;
qed "gen_lemma1";

val [prem1,prem2] =
Goal "[|P; P ==> R|] ==> R";
br prem2 1;
br prem1 1;
qed "gen_lemma2";

val [prem1,prem2,prem3] =
Goal "[|P --> S; [|P ; S|] ==> R ; P|] ==> R";
br prem2 1;
br (prem1 RS mp) 2;
br prem3 1;
br prem3 1;
qed "gen_lemma3";

(***
*
* LMixEnv is injective!
*
***)

Addsimps [lwrap_def];

Goal "!vlist vlist'. (lenv vlist = lenv vlist' --> vlist = vlist')";
br allI 1;
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
br allI 1;
by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
force 1;
force 1;
br allI 1;
br impI 1;
by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
force 1;
by (Asm_full_simp_tac 1);
qed "lenv_lemma1";

Goal "(<M,vlist> = <M',vlist'>) = ((M=M') & (vlist = vlist'))";
auto();
br (lenv_lemma1 RS gen_lemma2) 1;
auto();
qed "lmixenv_inj";

Addsimps [lmixenv_inj];

(********)
```

```
(********)

Goal "!q. lenv vlist ~= [delay|q]";
by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
auto();
qed "lenvnotdelay";

Addsimps [lenvnotdelay];

(********)
(********)

Goal "(<Atom v,vlist> |- env ---> d) = (d = lenv vlist)";
auto();
qed "lenv_lemma2";

(********)
(********)

Goal "([[var|Atom v]|[[value|Atom vn’]|<[Lam|[Atom vn|[M|nil]]],vlist>]] |- \
\ env ---> d) = (d = lenv vlist)";
auto();
qed "lenv_lemma3";

Addsimps [lenv_lemma2,lenv_lemma3];

(********)
(********)

Addsimps [vars_atom,vars_pair];

Goal "vars (lenv (vlist # vlist’)) = \
\ (vars (lenv vlist)) Un (vars (lenv vlist’))";
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
auto();
by (res_inst_tac [("rlist","vlist’")] rlist.induct 1);
auto();
qed "vars_lemma3";

Addsimps [vars_lemma3];

Goal "vars <M,vlist> <= vars <M,vlist # vlist’>";
auto();
qed "vars_lemma4";

Addsimps [vars_lemma4];

Delsimps [vars_atom,vars_pair];
Delsimps [lwrap_def];

Goal "[|A <= B; a ~: B|] ==> a ~: A";
auto();
qed "sub_lemma1";

Goal "(vlist # []) = vlist";
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
auto();
qed "lconcat_lemma2";
```

```

Goal "(vlist # [v';v;M,vlist']) ~= []";
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
auto();
qed "lconcat_lemma3";

Addsimps [lconcat_lemma2,lconcat_lemma3];

(**
Goal "(( vlist # vlist') : lmixrlist) --> \
\ (vlist : lmixrlist & vlist' : lmixrlist)";
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
auto();

by (eres_inst_tac [("x","z'")] allE 1);
by (eres_inst_tac [("x","z")] allE 1);
by (eres_inst_tac [("x","Mz")] allE 1);

by (res_inst_tac [("y","vlist'")] rlist.exhaust 2);

qed "lconcat_lemma1";
**)
```

C.2 Preliminary Definitions

```
(**********************************************************************
*
* ROOT.ML
*
**********************************************************************)

use_thy "LemRen";
use "LemAlpha";
```

```
(**********************************************************************
*
* Defs.thy
*
**********************************************************************)

Defs = SExpr + ExtNat +

syntax
        "nil"           :: sexpr ("nil")

        "Quote"         :: sexpr ("Quote")
        "Lam"           :: sexpr ("Lam")
        "App"           :: sexpr ("App")
        "Fix"           :: sexpr ("Fix")
        "Cond"          :: sexpr ("Cond")
        "Cons"          :: sexpr ("Cons")
        "Car"           :: sexpr ("Car")
        "Cdr"           :: sexpr ("Cdr")
        "IsEq"          :: sexpr ("IsEq")
        "IsAtom"        :: sexpr ("IsAtom")
        "Error"         :: sexpr ("Error")

        "Lift"          :: sexpr ("Lift")
```

```
        "RQuote"        :: sexpr ("RQuote")
        "RLam"          :: sexpr ("RLam")
        "RApp"          :: sexpr ("RApp")
        "RFix"          :: sexpr ("RFix")
        "RCond"         :: sexpr ("RCond")
        "RCons"         :: sexpr ("RCons")
        "RCar"          :: sexpr ("RCar")
        "RCdr"          :: sexpr ("RCdr")
        "RIsEq"         :: sexpr ("RIsEq")
        "RIsAtom"       :: sexpr ("RIsAtom")
        "RError"        :: sexpr ("RError")

        "clos"          :: sexpr ("clos")
        "delay"         :: sexpr ("delay")
        "#t"            :: sexpr ("#t")

        "eval"          :: sexpr ("eval")
        "expr"          :: sexpr ("expr")
        "env"           :: sexpr ("env")
        "value"         :: sexpr ("value")
        "var"           :: sexpr ("var")
        "x"             :: sexpr ("x")

translations
        "nil"           == "(Atom  0)"

        "Quote"         == "(Atom  1)"
        "Lam"           == "(Atom  2)"
        "App"           == "(Atom  3)"
        "Fix"           == "(Atom  4)"
        "Cond"          == "(Atom  5)"
        "Cons"          == "(Atom  6)"
        "Car"           == "(Atom  7)"
        "Cdr"           == "(Atom  8)"
        "IsEq"          == "(Atom  9)"
        "IsAtom"        == "(Atom 10)"
        "Error"         == "(Atom 11)"

        "Lift"          == "(Atom 12)"
        "RQuote"        == "(Atom 13)"
        "RLam"          == "(Atom 14)"
        "RApp"          == "(Atom 15)"
        "RFix"          == "(Atom 16)"
        "RCond"         == "(Atom 17)"
        "RCons"         == "(Atom 18)"
        "RCar"          == "(Atom 19)"
        "RCdr"          == "(Atom 20)"
        "RIsEq"         == "(Atom 21)"
        "RIsAtom"       == "(Atom 22)"
        "RError"        == "(Atom 23)"

        "delay"         == "(Atom 24)"
        "clos"          == "(Atom 25)"
        "#t"            == "(Atom 26)"

        "eval"          == "(Atom 35)"
        "expr"          == "(Atom 36)"
        "env"           == "(Atom 37)"
```

```
        "value"         == "(Atom 38)"
        "var"           == "(Atom 39)"
        "x"             == "(Atom 40)"
end
```

```
(**********************************************************************
*
* ExtNat.thy
*
**********************************************************************)

ExtNat = Nat +

syntax

         "3" :: nat ( "3")
         "4" :: nat ( "4")
         "5" :: nat ( "5")
         "6" :: nat ( "6")
         "7" :: nat ( "7")
         "8" :: nat ( "8")
         "9" :: nat ( "9")
        "10" :: nat ("10")
        "11" :: nat ("11")
        "12" :: nat ("12")
        "13" :: nat ("13")
        "14" :: nat ("14")
        "15" :: nat ("15")
        "16" :: nat ("16")
        "17" :: nat ("17")
        "18" :: nat ("18")
        "19" :: nat ("19")
        "20" :: nat ("20")
        "21" :: nat ("21")
        "22" :: nat ("22")
        "23" :: nat ("23")
        "24" :: nat ("24")
        "25" :: nat ("25")
        "26" :: nat ("26")
        "27" :: nat ("27")
        "28" :: nat ("28")
        "29" :: nat ("29")
        "30" :: nat ("30")
        "31" :: nat ("31")
        "32" :: nat ("32")
        "33" :: nat ("33")
        "34" :: nat ("34")
        "35" :: nat ("35")
        "36" :: nat ("36")
        "37" :: nat ("37")
        "38" :: nat ("38")
        "39" :: nat ("39")
        "40" :: nat ("40")
        "41" :: nat ("41")
        "42" :: nat ("42")
        "43" :: nat ("43")

translations

```

```
         "3" == "Suc  2"
         "4" == "Suc  3"
         "5" == "Suc  4"
         "6" == "Suc  5"
         "7" == "Suc  6"
         "8" == "Suc  7"
         "9" == "Suc  8"
        "10" == "Suc  9"
        "11" == "Suc 10"
        "12" == "Suc 11"
        "13" == "Suc 12"
        "14" == "Suc 13"
        "15" == "Suc 14"
        "16" == "Suc 15"
        "17" == "Suc 16"
        "18" == "Suc 17"
        "19" == "Suc 18"
        "20" == "Suc 19"
        "21" == "Suc 20"
        "22" == "Suc 21"
        "23" == "Suc 22"
        "24" == "Suc 23"
        "25" == "Suc 24"
        "26" == "Suc 25"
        "27" == "Suc 26"
        "28" == "Suc 27"
        "29" == "Suc 28"
        "30" == "Suc 29"
        "31" == "Suc 30"
        "32" == "Suc 31"
        "33" == "Suc 32"
        "34" == "Suc 33"
        "35" == "Suc 34"
        "36" == "Suc 35"
        "37" == "Suc 36"
        "38" == "Suc 37"
        "39" == "Suc 38"
        "40" == "Suc 39"
        "41" == "Suc 40"
        "42" == "Suc 41"
        "43" == "Suc 42"

end
```

C.3 SExpr

```
(**********************************************************************
*
* SExpr.thy
*
**********************************************************************)

SExpr = Datatype + ExtNat +

types atom = nat

consts
        var :: atom set
```

```

translations
        "a : var" == "34 < a"

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

consts
        vars :: sexpr => atom set

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

end
```

```
(**********************************************************************
*
* SExpr.ML
*
**********************************************************************)

Delsimps [vars_atom,vars_pair];
```

C.4 Alpha

```
(**********************************************************************
*
* Alpha.thy
*
**********************************************************************)

Alpha = Rename +

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''"

end
```

```
(*********************************************************************
*
* Alpha.ML
*
*********************************************************************)
Addsimps alpha.intrs;
AddIs alpha.intrs;

Delsimps [alpha.alpha_sym,alpha.alpha_trans];
Delrules [alpha.alpha_sym,alpha.alpha_trans];

val alpelims = map (alpha.mk_cases sexpr.simps)
[
        "M -a- M",
        "[Lam|[Atom v|[M|nil]]] -a- [Lam|[Atom v'|[M'|nil]]]",
        "[App|[M|[N|nil]]] -a- [App|[M'|[N'|nil]]]",
        "[Fix|[M|nil]] -a- [Fix|[M'|nil]]",
        "[Cond|[M|[N|[P|nil]]]] -a- [Cond|[M'|[N'|[P'|nil]]]]",
        "[Cons|[M|[N|nil]]] -a- [Cons|[M'|[N'|nil]]]",
        "[Car|[M|nil]] -a- [Car|[M'|nil]]",
        "[Cdr|[M|nil]] -a- [Cdr|[M'|nil]]",
        "[IsEq|[M|[N|nil]]] -a- [IsEq|[M'|[N'|nil]]]",
        "[IsAtom|[M|nil]] -a- [IsAtom|[M'|nil]]",
        "[Error|[M|nil]] -a- [Error|[M'|nil]]"
];

AddSEs alpelims;
```

C.5 L1Expr

```
L1Expr = Defs +

consts
        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"

end
```

```
(***********************************************************************
*
* L1Expr.ML
*
***********************************************************************)

Addsimps l1expr.intrs;
AddIs l1expr.intrs;

val l1elims = map (l1expr.mk_cases sexpr.simps)
[
        "Atom a : l1expr",
        "[Quote|[d|nil]] : l1expr",
        "[Lam|[Atom v|[N|nil]]] : l1expr",
        "[App|[M|[N|nil]]] : l1expr",
        "[Fix|[M|nil]] : l1expr",
        "[Cond|[M|[N|[P|nil]]]] : l1expr",
        "[Cons|[M|[N|nil]]] : l1expr",
        "[Car|[M|nil]] : l1expr",
        "[Cdr|[M|nil]] : l1expr",
        "[IsEq|[M|[N|nil]]] : l1expr",
        "[IsAtom|[M|nil]] : l1expr",
        "[Error|[M|nil]] : l1expr"
];

AddSEs l1elims;
```

```
Goal "M:l1expr --> (!q. M ~= [delay|q])";
br impI 1;
by (etac l1expr.induct 1);
auto();
qed "l1notdelay";

Goal "[delay|q] ~: l1expr";
br notI 1;
by (eresolve_tac l1expr.elims 1);
auto();
qed "delaynotl1";

Addsimps [l1notdelay,delaynotl1];
```

C.6 L2Eval

```
(***********************************************************************
*
* L2Eval.thy
*
***********************************************************************)

L2Eval = L1Expr + WF_Rel +

consts
        lookup :: "(sexpr*sexpr) => sexpr"
        l2eval :: "(sexpr * sexpr * sexpr)set"
        "@l2eval" :: [sexpr,sexpr,sexpr] => bool
                ("_ |- _ / ---> _" [0,0,50] 50)

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))"

translations "en |- exp ---> res" == "(exp,en,res) : l2eval"

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"

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 |- [RIsAtom|[M|nil]] ---> [IsAtom|[dM|nil]]"

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

end
```

```
(*********************************************************************
*
* L2Eval.ML
*
*********************************************************************)

Addsimps lookup.rules;
Addsimps l2eval.intrs;
AddIs l2eval.intrs;

val l2elims = map (l2eval.mk_cases (sexpr.simps @ l1expr.defs))
[
        "rho |- Atom v ---> d",
        "rho |- [Quote|[M|nil]] ---> d",
        "rho |- [Lam|[Atom v|[M|nil]]] ---> d",
        "rho |- [App|[M|[N|nil]]] ---> d",
        "rho |- [Fix|[M|nil]] ---> d",
        "rho |- [Cond|[M|[N|[P|nil]]]] ---> d",
        "rho |- [Cons|[M|[N|nil]]] ---> d",
        "rho |- [Car|[M|nil]] ---> d",
        "rho |- [Cdr|[M|nil]] ---> d",
        "rho |- [IsEq|[M|[N|nil]]] ---> d",
        "rho |- [IsAtom|[M|nil]] ---> d",
        "rho |- [Error|[M|nil]] ---> d",
        "rho |- [Lift|[M|nil]] ---> d",
        "rho |- [RQuote|[M|nil]] ---> d",
        "rho |- [RLam|[Atom v|[M|nil]]] ---> d",
        "rho |- [RApp|[M|[N|nil]]] ---> d",
        "rho |- [RFix|[M|nil]] ---> d",
        "rho |- [RCond|[M|[N|[P|nil]]]] ---> d",
        "rho |- [RCons|[M|[N|nil]]] ---> d",
        "rho |- [RCar|[M|nil]] ---> d",
        "rho |- [RCdr|[M|nil]] ---> d",
        "rho |- [RIsEq|[M|[N|nil]]] ---> d",
        "rho |- [RIsAtom|[M|nil]] ---> d",
        "rho |- [RError|[M|nil]] ---> d"
];

AddSEs l2elims;

val deriv_forw = fn i =>
        REPEAT (FIRST [Force_tac i,swap_res_tac l2eval.intrs i]);

```

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

C.7 LMix

```
(*********************************************************************
*
* LMix.thy
*
*********************************************************************)

LMix = L2Eval + Alpha + LMixEnv + Sint + NatInf +
end
```

```
(*********************************************************************
*
* LMix.ML
*
*********************************************************************)

use "LWrap";
use "Eval";
use "Dispatch";
use "Deriv";
```

C.8 LemRen

```
(*********************************************************************
*
* LemRen.thy
*
*********************************************************************)

LemRen = LMix +

consts
        addvar :: [atom,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]"

end
```

```
(*********************************************************************
*
* LemRen.ML
*
*********************************************************************)

use "EnvLem";
use "LemEnv";
use "Corr";
use "LemTerm";

```

```
Goal "M : l1expr & v' : var --> rename (v,v',M) : l1expr";
br impI 1;
by (res_inst_tac [("xa","M")] l1expr.induct 1);
auto();
qed "renl1";

Goal "M : l1expr & v' : var --> \
\ vars (rename (v,v',M)) <= vars M Un {v'}";
br impI 1;
by (res_inst_tac [("xa","M")] l1expr.induct 1);
auto();
qed "ren_lemma1";

Goal "vlist : lmixrlist --> (<M,[v';v;Mv,vlist]> : lmixenv --> \
\ (<rename (v,v',M),addvar v' vlist> : lmixenv))";
auto();
by (eresolve_tac lmixenv.elims 1);
force 1;
by (subgoal_tac "[v';v;Mv,vlist] : lmixrlist" 1);
force 2;
by (subgoal_tac "addvar v' vlist : lmixrlist" 1);
by (res_inst_tac [("y2","v"),("My2","Mv")]
        ((add_lemma1 RS mp) RS mp) 2);
force 2;
force 2;
auto();
by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
auto();
by (swap_res_tac lmixenv.intrs 1);
br (renl1 RS mp) 1;
force 1;
by (swap_res_tac lmixenv.intrs 1);
br (renl1 RS mp) 1;
force 1;
force 1;
by (subgoal_tac "vars (rename (v,v',M)) <= vars M Un {v'}" 1);
br (ren_lemma1 RS mp) 2;
force 2;
auto();
qed "ren_lemma2";

Goal "vlist : lmixrlist --> varsin (addvar y vlist) = varsin vlist";
br impI 1;
by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
force 1;
force 1;
force 1;
qed "ren_lemma3";

Addsimps [ren_lemma3,lemma_511];

use "LemWeak";

Goal "M : l1expr --> (!vlist. <M,[v';v;Mv,vlist]> : lmixenv --> \
\  (vlist : lmixrlist & <rename (v,v',M),addvar v' vlist> : lmixenv) \
\ --> (!d. (<M,[v';v;Mv,vlist]> |- bodyann ---> d) --> \
\  (<rename (v,v',M),addvar v' vlist> |- bodyann ---> d)))";
br impI 1;
```

```
by (res_inst_tac [("xa","M")] l1expr.induct 1);
force 1;
force 2;

(** App case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (subgoal_tac "<N, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (res_inst_tac [("x","dN")] exI 3);
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
by (subgoal_tac "<rename (v,v',N),addvar v' vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** Fix case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[Fix|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** Cond case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<B, [v';v;Mv,vlist]> : lmixenv" 3);
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (subgoal_tac "<N, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")]
```

```
        (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M’1","[Cond|[B|[Ma|[N|nil]]]]")]
        (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M’1","[Cond|[B|[Ma|[N|nil]]]]")]
        (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (res_inst_tac [("x","dN")] exI 3);
by (res_inst_tac [("x","dP")] exI 3);
by (subgoal_tac "<rename (v,v’,B),addvar v’ vlist> : lmixenv" 3);
by (subgoal_tac "<rename (v,v’,Ma),addvar v’ vlist> : lmixenv" 3);
by (subgoal_tac "<rename (v,v’,N),addvar v’ vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** Cons case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<Ma, [v’;v;Mv,vlist]> : lmixenv" 3);
by (subgoal_tac "<N, [v’;v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M’1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M’1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (res_inst_tac [("x","dN")] exI 3);
by (subgoal_tac "<rename (v,v’,Ma),addvar v’ vlist> : lmixenv" 3);
by (subgoal_tac "<rename (v,v’,N),addvar v’ vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** Car case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
```

```
br impI 3;
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[Car|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** Cdr case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cdr|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** IsEq case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (subgoal_tac "<N, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (res_inst_tac [("x","dN")] exI 3);
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
by (subgoal_tac "<rename (v,v',N),addvar v' vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** IsAtom case **)
```

```
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsAtom|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** Error case **)
br allI 3;
br impI 3;
br impI 3;
br allI 3;
br impI 3;
by (subgoal_tac "<Ma, [v';v;Mv,vlist]> : lmixenv" 3);
by (res_inst_tac [("M'1","[Error|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (Asm_full_simp_tac 3);
by (REPEAT (etac exE 3));
by (res_inst_tac [("x","dM")] exI 3);
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 3);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 4);
force 4;
force 4;
force 3;

(** Lam case **)
br allI 2;
br impI 2;
br impI 2;
br allI 2;
br impI 2;
by (subgoal_tac "<Ma,[v';v;Mv,vlist]> : lmixenv" 2);
by (res_inst_tac [("M'1","[Lam|[Atom a|[Ma|nil]]]")]
        (env_lemma6 RS mp) 3);
force 3;
by (subgoal_tac "<rename (v,v',Ma),addvar v' vlist> : lmixenv" 2);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 3);
force 3;
force 3;

by (Asm_full_simp_tac 2);
br conjI 2;
by (eres_inst_tac [("x","vlist")] allE 2);

br impI 2;
by (Asm_full_simp_tac 2);
by (REPEAT (etac exE 2));
by (Asm_full_simp_tac 2);
```

```
br conjI 2;
by (REPEAT (etac conjE 2));
by (REPEAT (etac conjE 3));
br corr_513 3;
force 3;
force 3;
force 3;
by (swap_res_tac lmixenv.intrs 2);
force 2;
by (swap_res_tac lmixrlist.intrs 2);
by (res_inst_tac [("y2","a"),("My2","Mv")]
        ((add_lemma1 RS mp) RS mp) 2);
force 2;
by (eresolve_tac lmixenv.elims 2);
force 2;
force 2;
force 2;
force 2;
by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,[v';a;Mv,vlist]]>")]
        lmixenv.elim 2);
force 2;
force 2;
by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,[v';a;Mv,vlist]]>")]
        lmixenv.elim 2);
force 2;
by (res_inst_tac [("y","vlist")] rlist.exhaust 3);
force 3;
force 4;
by (Asm_full_simp_tac 3);
by (eresolve_tac lmixenv.elims 3);
force 3;
by (dres_inst_tac [("s","<[Lam|[Atom a|[Ma|nil]]], \
\ [v';a;Mv,[atom1;atom2;sexpr,rlist]]>")] sym 3);
by (Asm_full_simp_tac 3);
force 3;
by (dres_inst_tac [("s","<Ma,[v'a;a;Ma,[v';a;Mv,vlist]]>")] sym 2);
by (Asm_full_simp_tac 2);
by (eres_inst_tac [("a","[v'a;a;Ma,[v';a;Mv,vlist]]")]
        lmixrlist.elim 2);
by (Asm_full_simp_tac 2);
by (dres_inst_tac [("s","[v'a;a;Ma,[v';a;Mv,vlist]]")] sym 2);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 2;
force 2;
force 2;
force 2;
br impI 2;
by (Asm_full_simp_tac 2);
by (REPEAT (etac exE 2));
by (Asm_full_simp_tac 2);
by (subgoal_tac "<rename (v,v',Ma), \
\ [v'a;a;rename (v,v',Ma),addvar v' vlist]> : lmixenv" 2);
br conjI 2;
```

```
force 2;
by (defer_tac 2);
by (REPEAT (etac conjE 2));
by (REPEAT (etac conjE 3));

by (swap_res_tac lmixenv.intrs 2);
br (renl1 RS mp) 2;
by (eresolve_tac lmixenv.elims 2);
force 2;
force 2;
force 3;
by (swap_res_tac lmixrlist.intrs 2);
by (subgoal_tac "[v';v;Mv,vlist] : lmixrlist" 2);
by (res_inst_tac [("y2","v"),("My2","Mv")]
        ((add_lemma1 RS mp) RS mp) 2);
force 2;
force 2;
by (eresolve_tac lmixenv.elims 2);
force 2;
force 2;
br (renl1 RS mp) 2;
by (eresolve_tac lmixenv.elims 2);
force 2;
force 2;
force 2;
by (res_inst_tac [("a","<Ma,[v'a;a;Ma,[v';v;Mv,vlist]]>")]
        lmixenv.elim 2);
force 2;
force 2;
force 2;
by (res_inst_tac [("a","<Ma,[v'a;a;Ma,[v';v;Mv,vlist]]>")]
        lmixenv.elim 2);
force 2;
force 2;
by (dres_inst_tac [("s","<Ma,[v'a;a;Ma,[v';v;Mv,vlist]]>")] sym 2);
by (Asm_full_simp_tac 2);
by (eres_inst_tac [("a","[v'a;a;Ma,[v';v;Mv,vlist]]")]
        lmixrlist.elim 2);
force 2;
by (dres_inst_tac [("s","[v'a;a;Ma,[v';v;Mv,vlist]]")] sym 2);
by (Asm_full_simp_tac 2);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
br conjI 2;
br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 2;
force 2;
force 2;
force 2;
br ((ren_lemma1 RS mp) RS sub_lemma1) 2;
force 2;
force 2;

by (eres_inst_tac [("x","vlist")] allE 2);
by (res_inst_tac [("y","vlist")] rlist.exhaust 2);
force 2;
by (Asm_full_simp_tac 2);
by (eresolve_tac lmixenv.elims 2);
force 2;
by (dres_inst_tac [("s","<[Lam|[Atom a|[Ma|nil]]], \
```

```
\ [v';v;Mv,[atom1;atom2;sexpr,rlist]]>")] sym 2);
by (Asm_full_simp_tac 2);
br conjI 2;
force 3;
br impI 2;
br conjI 2;
by (eres_inst_tac
        [("a","<Ma,[v'a;a;Ma,[v';v;Mv,[atom1;atom2;sexpr,rlist]]]>")]
        lmixenv.elim 2);
force 2;
force 2;
by (subgoal_tac "vars Ma <= vars sexpr" 2);
force 3;
by (subgoal_tac "vars (rename (v,v',Ma)) <= vars Ma Un {v'}" 2);
br (ren_lemma1 RS mp) 3;
force 3;
by (subgoal_tac "vars Ma Un {v'} <= vars sexpr Un {v'}" 2);
force 3;
by (subgoal_tac "vars sexpr Un {v'} <= insert v' (vars sexpr)" 2);
force 3;
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
force 2;

by (subgoal_tac "<Ma,[] # \
\ [v';v;Mv,[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]> : lmixenv" 2);
br (env_lemma3 RS mp) 3;
force 3;
by (subgoal_tac "<Ma,[v';v;Mv,[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]> \
\ |- bodyann ---> dM" 2);
br corr_512 3;
force 3;
force 3;
force 3;
by (eres_inst_tac [("x","[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]")]
        allE 2);
by (Asm_full_simp_tac 2);

by (subgoal_tac "<rename(v,v',Ma), \
\ addvar v' [v'a;a;[App|[Mv|[Atom v|nil]]],vlist]> : lmixenv" 2);
by (res_inst_tac [("Mv2","Mv")] ((ren_lemma2 RS mp) RS mp) 3);
by (eres_inst_tac
        [("a","<Ma,[v';v;Mv,[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]>")]
```

```
        lmixenv.elim 3);
force 3;
force 3;
force 3;
by (Asm_full_simp_tac 2);
by (eres_inst_tac
        [("a","<Ma,[v';v;Mv,[v'a;a;[App|[Mv|[Atom v|nil]]],vlist]]>")]
        lmixenv.elim 2);
force 2;
by (dres_inst_tac [("t","<Maa,[v'aa;va;M',vlista]>")] sym 2);
by (Asm_full_simp_tac 2);
by (SELECT_GOAL Auto_tac 2);
by (eres_inst_tac [("x","dM")] allE 2);
by (mp_tac 2);

by (res_inst_tac [("M2","rename (v,v',Ma)"),("Mv'2","rename (v,v',Ma)"),
        ("Mv2","[App|[[App|[Mv|[Atom v|nil]]]|[Atom v'|nil]]]"),
        ("vlist'2","addvar v' vlist"), ("v'2","v'a"),("v2","a"),
        ("x","[]")] ((lem_weak RS mp) RS allE) 2);
br (renl1 RS mp) 2;
force 2;
by (Asm_full_simp_tac 2);
by (subgoal_tac "vars (rename (v,v',Ma)) <= vars \
\ [App|[[App|[Mv|[Atom v|nil]]]|[Atom v'|nil]]]" 2);
force 2;
by (Asm_full_simp_tac 2);
by (eres_inst_tac [("a","<rename (v,v',Ma), \
\ [v'a;a;[App|[[App|[Mv|[Atom v|nil]]]|[Atom v'|nil]]], \
\ addvar v' vlist]>")] lmixenv.elim 2);
force 2;
by (SELECT_GOAL Auto_tac 2);

(** Var case **)
br allI 1;
br impI 1;
br impI 1;
be conjE 1;
br allI 1;
br impI 1;
by (case_tac "a = v" 1);

by (Asm_full_simp_tac 1);
by (subgoal_tac "v' ~: varsin (addvar v' vlist)" 1);
force 1;
by (Asm_full_simp_tac 1);
by (eresolve_tac lmixenv.elims 1);
force 1;
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (SELECT_GOAL Auto_tac 1);
by (subgoal_tac "v' ~: varsin vlista" 1);
force 1;
br ((vars_lemma1 RS mp) RS sub_lemma1) 1;
force 1;
force 1;

by (SELECT_GOAL Auto_tac 1);
by (SELECT_GOAL Auto_tac 1);
```

```
by (res_inst_tac [("y","a"),("v'","v'"),("v","v"),("M","Mv"),
        ("vlist","vlist"),("d","d")] derivelimvar1 1);
force 1;
br add_lemma2b 1;
force 1;
force 1;
qed "lemma_515";
```

C.9 NatInf

```
(*********************************************************************
*
* NatInf.thy
*
*********************************************************************)

NatInf = Nat + Finite + SExpr +

consts
        "new_var" :: "[nat,nat set] => bool"

defs
        new_var_def "new_var n nset == (n : var) & (!m:nset. m < n)"

end
```

```
(*********************************************************************
*
* NatInf.ML
*
*********************************************************************)

Addsimps [new_var_def,max_def];

Goal "finite nset ==> (? n. new_var n nset)";
auto();
br Finites.induct 1;
force 1;
force 1;
be exE 1;
by (res_inst_tac [("x","Suc (max a n)")] exI 1);
auto();
qed "newexists";

Goal "finite nset ==> (? v. (v : var) & (v ~: nset))";
by (subgoal_tac "(? n. new_var n nset)" 1);
br newexists 2;
force 2;
be exE 1;
by (res_inst_tac [("x","n")] exI 1);
force 1;
qed "freshvar";
```

C.10 Rename

```
(*********************************************************************
```

```
*
* Rename.thy
*
*********************************************************************)

Rename = L1Expr + WF_Rel +

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]]"

end
```

```
(*********************************************************************
*
* Rename.ML
*
*********************************************************************)

Addsimps rename.rules;
```

C.11 Sint

```
(*********************************************************************
*
* Sint.thy
*
```

```
***********************************************************************)

Sint = L1Expr +

consts
        bodyann :: sexpr
        sintann :: sexpr

syntax
        "Cadr"    :: sexpr => sexpr ("Cadr")
        "Caddr"   :: sexpr => sexpr ("Caddr")
        "Cadddr"  :: sexpr => sexpr ("Cadddr")
        "bodyann'" :: sexpr ("bodyann'")
        "sintann'" :: sexpr ("sintann'")
        "envlam"  :: sexpr ("envlam")

translations

"envlam" == "[Lam|[var|
        [[Cond|[[IsEq|[var|[[Cadr|[expr|nil]]|nil]]]|
        [value|[[App|[env|[var|nil]]]|nil]]]]|nil]]]"

"[Cadr|[M|nil]]" == "[Car|[[Cdr|[M|nil]]|nil]]"

"[Caddr|[M|nil]]" == "[Cadr|[[Cdr|[M|nil]]|nil]]"

"[Cadddr|[M|nil]]" == "[Caddr|[[Cdr|[M|nil]]|nil]]"

"bodyann'" ==
  "[Cond|[[IsAtom|[expr|nil]]|
        [[App|[env|[expr|nil]]]|

   [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Quote|nil]]|nil]]]|
        [[Lift|[[Cadr|[expr|nil]]|nil]]|

   [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Lam|nil]]|nil]]]|
        [[RLam|[value|
                [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[envlam|nil]]]|nil]]]|

   [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[App|nil]]|nil]]]|
        [[RApp|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|
                  [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]]|

   [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Fix|nil]]|nil]]]|
        [[RFix|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]|

   [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Cond|nil]]|nil]]]|
        [[RCond|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|
                [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]|
                [[App|[[App|[eval|[[Cadddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]]]|

   [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Cons|nil]]|nil]]]|
        [[RCons|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|
                [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]]|

   [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Car|nil]]|nil]]]|
        [[RCar|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]|
```

```
    [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Cdr|nil]]|nil]]]|
         [[RCdr|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]|

    [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[IsEq|nil]]|nil]]]|
         [[RIsEq|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|
                 [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]]|

    [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[IsAtom|nil]]|nil]]]|
         [[RIsAtom|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]|

    [[Cond|[[IsEq|[[Car|[expr|nil]]|[[Quote|[Error|nil]]|nil]]]|
         [[RError|[[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]|

    [[RError|[expr|nil]]|nil]]]]|nil]]]]|nil]]]]|nil]]]]|nil]]]]|nil]]]]|nil]]]]|
         nil]]]]|nil]]]]|nil]]]]|nil]]]]|nil]]]]"

"sintann'" ==
        "[Fix|[[Lam|[eval|[[Lam|[expr|[[Lam|[env|[bodyann|nil]]]|nil]]]|nil]]]|nil]]"

defs
        bodyann_def "bodyann == bodyann'"
        sintann_def "sintann == sintann'"

end
```

C.12 Miscellaneous Proofs

```
(*********************************************************************
*
* Corr.ML
*
*********************************************************************)

Goal "[|y ~= z; <M,[y';y;My,[z';z;Mz,vlist']]> : lmixenv; \
\ <M,[y';y;My,[z';z;Mz,vlist']]> |- bodyann ---> d |] \
\ ==> (<M,[z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]> \
\ |- bodyann ---> d)";
by (res_inst_tac [("x","[]"),("y2","y"),("z2","z"),("M2","M"),
        ("y'2","y'"),("My2","My"),("z'2","z'"),("Mz2","Mz"),
        ("vlist'2","vlist'")] ((lemma_512 RS mp) RS allE) 1);
by (eresolve_tac lmixenv.elims 1);
auto();
qed "corr_512";

Goal "[|y = z;<M,[y';y;My,[z';z;Mz,vlist']]> : lmixenv; \
\ <M,[y';y;My,[z';z;Mz,vlist']]> |- bodyann ---> d |] \
\ ==> (<M,[y';y;My,addvar z' vlist']> |- bodyann ---> d)";
by (res_inst_tac [("x","[]"),("y2","y"),("z2","z"),("M2","M"),
        ("y'2","y'"),("My2","My"),("z'2","z'"),("Mz2","Mz"),
        ("vlist'2","vlist'")] ((lemma_513 RS mp) RS allE) 1);
by (eresolve_tac lmixenv.elims 1);
auto();
qed "corr_513";
```

```
(*********************************************************************
*
* Deriv.ML
*
```

```
***********************************************************************)

Addsimps [vars_atom,vars_pair];

(***********)
(* Lemmata *)
(***********)

Goal "(<Atom v,vlist> : lmixenv) --> \
\ (!d. (<Atom v,vlist> |- env ---> d) = \
\ ([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] \
\ |- env ---> d))";
auto();
qed "envenv";

Goal "(<Atom v,vlist> : lmixenv) --> \
\ (!d. (<Atom v,vlist> |- expr ---> d) = \
\ ([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] \
\ |- var ---> d))";
auto();
by (deriv_back 2);
by (deriv_forw 1);
qed "varexpr";

Goal "((!dM. (rho |- M ---> dM) = (rho' |- M' --->dM)) \
\ & (!dN. (rho |- N ---> dN) = (rho' |- N' ---> dN))) --> \
\ (!d.(rho |- [App|[M|[N|nil]]] ---> d) = \
\    (rho' |- [App|[M'|[N'|nil]]] ---> d))";
auto();
by (eresolve_tac l2elims 1);
by (Asm_full_simp_tac 1);
by (swap_res_tac l2eval.intrs 1);
auto();
by (subgoal_tac "!dM.(rho'|-M' ---> dM)=(rho|-M --->dM)" 1);
by (subgoal_tac "!dN.(rho'|-N' ---> dN)=(rho|-N --->dN)" 1);
force 2;
force 2;
by (thin_tac "?Q" 1);
by (thin_tac "?Q" 1);
by (rotate_tac ~2 1);
by (eresolve_tac l2elims 1);
by (Asm_full_simp_tac 1);
by (swap_res_tac l2eval.intrs 1);
auto();
qed "appdet";

(********)
(********)

Goal "(<Atom v,vlist> : lmixenv) --> \
\ (!d. (<Atom v,vlist> |- [App|[env|[expr|nil]]] ---> d) = \
\ ([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] \
\ |- [App|[env|[var|nil]]] ---> d))";
br impI 1;
br allI 1;
by (res_inst_tac [("v1","v"),("vn'1","vn'"),("vn1","vn"),("M1","M"),
        ("vlist1","vlist")] (envenv RS gen_lemma3) 1);
by (res_inst_tac [("v1","v"),("vn'1","vn'"),("vn1","vn"),("M1","M"),
```

```
        ("vlist1","vlist")] (varexpr RS gen_lemma3) 1);
by (res_inst_tac [("rho1","<Atom v,vlist>"),("rho'1",
"[[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]]"),
        ("M1","env"), ("M'1","env"), ("N1","expr"), ("N'1","var")]
        (appdet RS gen_lemma3) 1);
auto();
qed "lemma1";

(**********************)
(* Derivation Lemmata *)
(**********************)

Goal "(<Atom w,[]> : lmixenv) --> \
\ (<Atom w,[]> |- bodyann ---> d') = (d' = Atom w)";
auto();
by (REPEAT ((Force_tac 2) ORELSE (CHANGED (deriv_forw 2))));
by (deriv_back 1);
qed "evaluation_varemp";

(********)
(********)

Goal "(<Atom v,[v';v;M,vlist]> : lmixenv) --> \
\ (<Atom v,[v';v;M,vlist]> |- bodyann ---> d') = (d' = (Atom v'))";
auto();
by (REPEAT ((Force_tac 2) ORELSE (CHANGED (deriv_forw 2))));
back();
back();
back();
back();
back();
back();
by (deriv_back 1);
qed "evaluation_varnonemp";

(********)
(********)

Goal "(<Atom v,[vn';vn;M,vlist]> : lmixenv & v ~= vn) --> \
\ (<Atom v,[vn';vn;M,vlist]> |- bodyann ---> d) = \
\ (<Atom v,vlist> : lmixenv & (<Atom v,vlist> |- bodyann ---> d))";
br impI 1;
by (subgoal_tac "<Atom v,vlist> : lmixenv" 1);
auto();
by (eresolve_tac lmixenv.elims 3);
force 3;
by (SELECT_GOAL Auto_tac 3);
by (res_inst_tac [("y","vlist")] rlist.exhaust 3);
force 3;
force 3;
by (REPEAT (swap_res_tac l2eval.intrs 2));
by (REPEAT (Force_tac 2));
by (REPEAT (swap_res_tac l2eval.intrs 2));
by (REPEAT (Force_tac 2));
by (REPEAT (swap_res_tac l2eval.intrs 2));
by (REPEAT (Force_tac 2));
by (res_inst_tac [("v1","v"),("vlist1","vlist")]
        (lemma1 RS gen_lemma3) 2);
```

```
by (rotate_tac ~1 2);
auto();
by (REPEAT (eresolve_tac l2elims 1));
auto();
by (eresolve_tac l2elims 1);
by (deriv_back 2);
by (res_inst_tac [("v1","v"),("vlist1","vlist")]
        (lemma1 RS gen_lemma3) 1);
force 2;
by (subgoal_tac
"!d.([[var|Atom v]|[[value|Atom vn']|<[Lam|[Atom vn|[M|nil]]],vlist>]] \
\ |- [App|[env|[var|nil]]] ---> d) = (<Atom v,vlist> \
\ |- [App|[env|[expr|nil]]] ---> d)" 1);
force 2;
by (rotate_tac ~1 1);
by (Asm_full_simp_tac 1);
qed "evaluation_varnonemp2";

(********)
(********)

Goal "(<[Quote|[d|nil]],vlist> : lmixenv) --> \
\ (<[Quote|[d|nil]],vlist> |- bodyann ---> d') = \
\ (d' = [Quote|[d|nil]])";
auto();
by (REPEAT ((Force_tac 2) ORELSE (CHANGED (deriv_forw 2))));
by (deriv_back 1);
qed "evaluation_quote";

(********)
(********)

Goal "(<[Lam|[Atom v|[M|nil]]],vlist> : lmixenv) --> \
\ ((<[Lam|[Atom v|[M|nil]]],vlist> |- bodyann ---> d') = \
\ (? dM v'. (<M,[v';v;M,vlist]> : lmixenv \
\   & (d' = [Lam|[Atom v'|[dM|nil]]]) & \
\ (<M,[v';v;M,vlist]> |- bodyann ---> dM))))";
auto();
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
by (res_inst_tac [("x","v'")] exI 1);
by (fold_tac l1expr.defs);
auto();
by (eresolve_tac lmixenv.elims 1);
force 1;
by (Asm_full_simp_tac 1);
by (REPEAT (etac conjE 1));
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (eresolve_tac l1elims 1);
auto();
by (deriv_back 1);
auto();
by (simp_tac (HOL_basic_ss addsimps [lwrap_def]) 1);
by (Simp_tac 1);

by (swap_res_tac l2eval.intrs 1);
by (eres_inst_tac [("a","<M,[v';v;M,vlist]>")] lmixenv.elim 1);
```

```
force 1;
force 1;
by (eres_inst_tac [("a","<M,[v';v;M,vlist]>")] lmixenv.elim 1);
force 1;

by (eresolve_tac lmrelims 1);
by (dtac sym 1);
by (asm_full_simp_tac (simpset() delsimps [vars_atom,vars_pair]) 1);
by (res_inst_tac [("a1","v'"),("M2","[Lam|[Atom v|[M|nil]]]"),
        ("vlist2","vlist")]
        ((vars_lemma2 RS sub_lemma1) RS gen_lemma1) 1);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
force 1;

by (deriv_forw 1);
back();
by (simp_tac (HOL_basic_ss addsimps [lenv_nemp RS sym]) 1);
by (fold_tac [lwrap_def]);
ba 1;
qed "evaluation_lam";

(********)
(********)

Goalw [lwrap_def] "(<M,vlist> : lmixenv & <M',vlist> : lmixenv) --> \
\ ([[env|lookup (<M',vlist>,env)]|[ \
\ [expr|M]|[[eval|[delay|[sintann|[nil|nil]]]]|nil]]] = <M,vlist>)";
auto();
qed "envlem42";

Goalw [lwrap_def] "! q. (lookup (<M,vlist>,env) ~= [delay|q])";
auto();
qed "envlem43";

Addsimps [envlem42,envlem43];

(********)
(********)

Goal "(<[App|[M|[N|nil]]],vlist> : lmixenv) --> \
\ ((<[App|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
\  (? dM dN. (d' = [App|[dM|[dN|nil]]]) & \
\   <M,vlist> : lmixenv & <N,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM) & (<N,vlist> |- bodyann ---> dN)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
by (res_inst_tac [("x","dN")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 3);
force 3;
by (dtac sym 3);
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
by (eresolve_tac l1elims 3);
auto();
```

```
by (eresolve_tac lmixenv.elims 3);
force 3;
by (dtac sym 3);
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
by (eresolve_tac l1elims 3);
auto();
by (thin_tac "?x |- ?y ---> dN" 1);
by (thin_tac "?x |- ?y ---> dM" 2);
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 3))
        THEN (REPEAT1 (Force_tac 3))));
by (deriv_back 1);
qed "evaluation_app";

(********)
(********)

Goal "(<[Cons|[M|[N|nil]]],vlist> : lmixenv) --> \
\ ((<[Cons|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
\  (? dM dN. (d' = [Cons|[dM|[dN|nil]]]) & \
\   <M,vlist> : lmixenv & <N,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM) & (<N,vlist> |- bodyann ---> dN)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
by (res_inst_tac [("x","dN")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 3);
force 3;
by (dtac sym 3);
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
by (eresolve_tac l1elims 3);
auto();
by (eresolve_tac lmixenv.elims 3);
force 3;
by (dtac sym 3);
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
by (eresolve_tac l1elims 3);
auto();
by (thin_tac "?x |- ?y ---> dN" 1);
by (thin_tac "?x |- ?y ---> dM" 2);
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 3))
        THEN (REPEAT1 (Force_tac 3))));
by (deriv_back 1);
qed "evaluation_cons";

(********)
(********)

Goal "(<[IsEq|[M|[N|nil]]],vlist> : lmixenv) --> \
\ ((<[IsEq|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
\  (? dM dN. (d' = [IsEq|[dM|[dN|nil]]]) & \
\   <M,vlist> : lmixenv & <N,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM) & (<N,vlist> |- bodyann ---> dN)))";
auto();
```

```
by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
by (res_inst_tac [("x","dN")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 3);
force 3;
by (dtac sym 3);
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
by (eresolve_tac l1elims 3);
auto();
by (eresolve_tac lmixenv.elims 3);
force 3;
by (dtac sym 3);
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
by (eresolve_tac l1elims 3);
auto();
by (thin_tac "?x |- ?y ---> dN" 1);
by (thin_tac "?x |- ?y ---> dM" 2);
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 3))
        THEN (REPEAT1 (Force_tac 3))));
by (deriv_back 1);
qed "evaluation_iseq";

(********)
(********)

Goal "(<[Fix|[M|nil]],vlist> : lmixenv) --> \
\ ((<[Fix|[M|nil]],vlist> |- bodyann ---> d') = \
\  (? dM. (d' = [Fix|[dM|nil]]) & \
\   <M,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (REPEAT (etac conjE 2));
by (eresolve_tac l1elims 2);
auto();
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
        THEN (REPEAT1 (Force_tac 2))));
by (deriv_back 1);
qed "evaluation_fix";

(********)
(********)

Goal "(<[Car|[M|nil]],vlist> : lmixenv) --> \
\ ((<[Car|[M|nil]],vlist> |- bodyann ---> d') = \
\  (? dM. (d' = [Car|[dM|nil]]) & \
\   <M,vlist> : lmixenv & \
```

```
\  (<M,vlist> |- bodyann ---> dM)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (REPEAT (etac conjE 2));
by (eresolve_tac l1elims 2);
auto();
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
        THEN (REPEAT1 (Force_tac 2))));
by (deriv_back 1);
qed "evaluation_car";

(********)
(********)

Goal "(<[Cdr|[M|nil]],vlist> : lmixenv) --> \
\ ((<[Cdr|[M|nil]],vlist> |- bodyann ---> d') = \
\  (? dM. (d' = [Cdr|[dM|nil]]) & \
\   <M,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (REPEAT (etac conjE 2));
by (eresolve_tac l1elims 2);
auto();
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
        THEN (REPEAT1 (Force_tac 2))));
by (deriv_back 1);
qed "evaluation_cdr";

(********)
(********)

Goal "(<[IsAtom|[M|nil]],vlist> : lmixenv) --> \
\ ((<[IsAtom|[M|nil]],vlist> |- bodyann ---> d') = \
\  (? dM. (d' = [IsAtom|[dM|nil]]) & \
\   <M,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 2);
force 2;
```

```
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (REPEAT (etac conjE 2));
by (eresolve_tac l1elims 2);
auto();
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
        THEN (REPEAT1 (Force_tac 2))));
by (deriv_back 1);
qed "evaluation_isatom";

(********)
(********)

Goal "(<[Error|[M|nil]],vlist> : lmixenv) --> \
\ ((<[Error|[M|nil]],vlist> |- bodyann ---> d') = \
\  (? dM. (d' = [Error|[dM|nil]]) & \
\   <M,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (REPEAT (etac conjE 2));
by (eresolve_tac l1elims 2);
auto();
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 2))
        THEN (REPEAT1 (Force_tac 2))));
by (deriv_back 1);
qed "evaluation_error";

(********)
(********)

Goal "(<[Cond|[M|[N|[P|nil]]]],vlist> : lmixenv) --> \
\ ((<[Cond|[M|[N|[P|nil]]]],vlist> |- bodyann ---> d') = \
\  (? dM dN dP. (d' = [Cond|[dM|[dN|[dP|nil]]]]) & \
\   <M,vlist> : lmixenv & <N,vlist> : lmixenv & <P,vlist> : lmixenv & \
\  (<M,vlist> |- bodyann ---> dM) & (<N,vlist> |- bodyann ---> dN) & \
\  (<P,vlist> |- bodyann ---> dP)))";
auto();
by (subgoal_tac "<M,vlist> : lmixenv & <N,vlist> : lmixenv & \
\ <P,vlist> : lmixenv" 1);
by (eresolve_tac l2elims 1);
by (res_inst_tac [("x","dM")] exI 1);
by (res_inst_tac [("x","dN")] exI 1);
by (res_inst_tac [("x","dP")] exI 1);
auto();
by (eresolve_tac lmixenv.elims 4);
force 4;
by (dtac sym 4);
by (Asm_full_simp_tac 4);
by (REPEAT (etac conjE 4));
by (eresolve_tac l1elims 4);
```

```
auto();
by (eresolve_tac lmixenv.elims 4);
force 4;
by (dtac sym 4);
by (Asm_full_simp_tac 4);
by (REPEAT (etac conjE 4));
by (eresolve_tac l1elims 4);
auto();
by (eresolve_tac lmixenv.elims 4);
force 4;
by (dtac sym 4);
by (Asm_full_simp_tac 4);
by (REPEAT (etac conjE 4));
by (eresolve_tac l1elims 4);
auto();
by (thin_tac "?x |- ?y ---> dN" 1);
by (thin_tac "?x |- ?y ---> dP" 1);
by (thin_tac "?x |- ?y ---> dM" 2);
by (thin_tac "?x |- ?y ---> dP" 2);
by (thin_tac "?x |- ?y ---> dM" 3);
by (thin_tac "?x |- ?y ---> dN" 3);
by (REPEAT ((REPEAT (swap_res_tac l2eval.intrs 4))
        THEN (REPEAT1 (Force_tac 4))));
by (deriv_back 1);
qed "evaluation_if";

val evaluation_lemmata = [evaluation_varemp,evaluation_varnonemp,evaluation_varnonemp2,
        evaluation_quote,evaluation_lam,evaluation_app,evaluation_fix,evaluation_if,evaluation_co
        evaluation_car,evaluation_cdr,evaluation_iseq,evaluation_isatom,evaluation_error];

Addsimps evaluation_lemmata;
Delsimps dispatch_lemmata;
```

```
(***********************************************************************
*
* Determ.ML
*
***********************************************************************)

AddSEs lmixenv.elims;

Goal "(<[Fix|[M|nil]],vlist> : lmixenv & \
\ <[Fix|[M|nil]],vlist'> : lmixenv) --> \
\ ((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) --> \
\  ((<[Fix|[M|nil]],vlist> |- bodyann ---> d) --> \
\   (<[Fix|[M|nil]],vlist'> |- bodyann ---> d)))";
auto();
qed "determ_fix";

(*********)
(*********)

Goal "(<[Car|[M|nil]],vlist> : lmixenv & \
\ <[Car|[M|nil]],vlist'> : lmixenv) --> \
\ ((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) --> \
\  ((<[Car|[M|nil]],vlist> |- bodyann ---> d) --> \
\   (<[Car|[M|nil]],vlist'> |- bodyann ---> d)))";
```

```
auto();
qed "determ_car";

(*********)
(*********)

Goal "(<[Cdr|[M|nil]],vlist> : lmixenv & \
\ <[Cdr|[M|nil]],vlist'> : lmixenv) --> \
\ ((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) --> \
\  ((<[Cdr|[M|nil]],vlist> |- bodyann ---> d) --> \
\   (<[Cdr|[M|nil]],vlist'> |- bodyann ---> d)))";
auto();
qed "determ_cdr";

(*********)
(*********)

Goal "(<[IsAtom|[M|nil]],vlist> : lmixenv & \
\ <[IsAtom|[M|nil]],vlist'> : lmixenv) --> \
\ ((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) --> \
\  ((<[IsAtom|[M|nil]],vlist>  |- bodyann ---> d) --> \
\   (<[IsAtom|[M|nil]],vlist'> |- bodyann ---> d)))";
auto();
qed "determ_isatom";

(*********)
(*********)

Goal "(<[Error|[M|nil]],vlist> : lmixenv & \
\ <[Error|[M|nil]],vlist'> : lmixenv) --> \
\ ((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) --> \
\  ((<[Error|[M|nil]],vlist>  |- bodyann ---> d) --> \
\   (<[Error|[M|nil]],vlist'> |- bodyann ---> d)))";
auto();
qed "determ_error";

Delrules lmixenv.elims;

(*********)
(*********)

Goal "(<[App|[M|[N|nil]]],vlist> : lmixenv & \
\ <[App|[M|[N|nil]]],vlist'> : lmixenv) --> \
\ (((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) & \
\   (!dN. (<N,vlist> |- bodyann ---> dN) --> \
\ (<N,vlist'> |- bodyann ---> dN))) --> \
\  ((<[App|[M|[N|nil]]],vlist>  |- bodyann ---> d) --> \
\   (<[App|[M|[N|nil]]],vlist'> |- bodyann ---> d)))";
br impI 1;
be conjE 1;
by (subgoal_tac "M : l1expr & N : l1expr" 1);
by (eresolve_tac lmixenv.elims 2);
auto();
by (res_inst_tac [("M'1","[App|[M|[N|nil]]]")] (env_lemma6 RS mp) 1);
```

```
by (res_inst_tac [("M'1","[App|[M|[N|nil]]]")] (env_lemma6 RS mp) 2);
auto();
qed "determ_app";

(*********)
(*********)

Goal "(<[Cons|[M|[N|nil]]],vlist> : lmixenv & \
\ <[Cons|[M|[N|nil]]],vlist'> : lmixenv) --> \
\ (((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) & \
\   (!dN. (<N,vlist> |- bodyann ---> dN) --> \
\ (<N,vlist'> |- bodyann ---> dN))) --> \
\  ((<[Cons|[M|[N|nil]]],vlist>  |- bodyann ---> d) --> \
\   (<[Cons|[M|[N|nil]]],vlist'> |- bodyann ---> d)))";
br impI 1;
be conjE 1;
by (subgoal_tac "M : l1expr & N : l1expr" 1);
by (eresolve_tac lmixenv.elims 2);
auto();
by (res_inst_tac [("M'1","[Cons|[M|[N|nil]]]")] (env_lemma6 RS mp) 1);
by (res_inst_tac [("M'1","[Cons|[M|[N|nil]]]")] (env_lemma6 RS mp) 2);
auto();
qed "determ_cons";

(*********)
(*********)

Goal "(<[IsEq|[M|[N|nil]]],vlist> : lmixenv & \
\ <[IsEq|[M|[N|nil]]],vlist'> : lmixenv) --> \
\ (((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) & \
\   (!dN. (<N,vlist> |- bodyann ---> dN) --> \
\ (<N,vlist'> |- bodyann ---> dN))) --> \
\  ((<[IsEq|[M|[N|nil]]],vlist>  |- bodyann ---> d) --> \
\   (<[IsEq|[M|[N|nil]]],vlist'> |- bodyann ---> d)))";
br impI 1;
be conjE 1;
by (subgoal_tac "M : l1expr & N : l1expr" 1);
by (eresolve_tac lmixenv.elims 2);
auto();
by (res_inst_tac [("M'1","[IsEq|[M|[N|nil]]]")] (env_lemma6 RS mp) 1);
by (res_inst_tac [("M'1","[IsEq|[M|[N|nil]]]")] (env_lemma6 RS mp) 2);
auto();
qed "determ_iseq";

Goal "(<[Cond|[M|[N|[P|nil]]]],vlist> : lmixenv & \
\      <[Cond|[M|[N|[P|nil]]]],vlist'> : lmixenv) --> \
\ (((!dM. (<M,vlist> |- bodyann ---> dM) --> \
\ (<M,vlist'> |- bodyann ---> dM)) & \
\   (!dN. (<N,vlist> |- bodyann ---> dN) --> \
\ (<N,vlist'> |- bodyann ---> dN)) & \
\   (!dP. (<P,vlist> |- bodyann ---> dP) --> \
\ (<P,vlist'> |- bodyann ---> dP))) --> \
\  ((<[Cond|[M|[N|[P|nil]]]],vlist>  |- bodyann ---> d) --> \
\   (<[Cond|[M|[N|[P|nil]]]],vlist'> |- bodyann ---> d)))";
br impI 1;
be conjE 1;
```

```
by (subgoal_tac "M : l1expr & N : l1expr & P : l1expr" 1);
by (eresolve_tac lmixenv.elims 2);
auto();
by (res_inst_tac [("M'1","[Cond|[M|[N|[P|nil]]]]")]
        (env_lemma6 RS mp) 1);
by (res_inst_tac [("M'1","[Cond|[M|[N|[P|nil]]]]")]
        (env_lemma6 RS mp) 2);
by (res_inst_tac [("M'1","[Cond|[M|[N|[P|nil]]]]")]
        (env_lemma6 RS mp) 3);
auto();
qed "determ_if";

Goal "(<[Quote|[d|nil]],vlist> : lmixenv & \
\ <[Quote|[d|nil]],vlist'> : lmixenv) --> \
\  ((<[Quote|[d|nil]],vlist> |- bodyann ---> d') --> \
\   (<[Quote|[d|nil]],vlist'> |- bodyann ---> d'))";
auto();
qed "determ_quote";

Goal "(<[Lam|[Atom v|[M|nil]]],vlist> : lmixenv & \
\       <[Lam|[Atom v|[M|nil]]],vlist'> : lmixenv & \
\        vars <[Lam|[Atom v|[M|nil]]],vlist'> \
\     <= vars <[Lam|[Atom v|[M|nil]]],vlist>) --> \
\     ((!vlist''. (<M,vlist'' # vlist> : lmixenv) --> \
\    (!dM. (<M,vlist'' # vlist>  |- bodyann ---> dM) --> \
\         (<M,vlist'' # vlist'> |- bodyann ---> dM))) --> \
\  ((<[Lam|[Atom v|[M|nil]]],vlist>  |- bodyann ---> d) --> \
\   (<[Lam|[Atom v|[M|nil]]],vlist'> |- bodyann ---> d)))";
auto();
by (eres_inst_tac [("x","[v';v;M,[]]")] allE 1);
by (Asm_full_simp_tac 1);
by (eres_inst_tac [("x","[v';v;M,[]]")] allE 2);
by (Asm_full_simp_tac 2);

by (swap_res_tac lmixenv.intrs 1);
by (eresolve_tac lmixenv.elims 1);
force 1;
force 1;
force 2;

by (eres_inst_tac [("a","<M,[v';v;M,vlist]>")] lmixenv.elim 1);
force 1;
by (dtac sym 1);
auto();
by (swap_res_tac lmixrlist.intrs 1);
force 2;
force 2;
force 2;

AddSEs lmixenv.elims;
auto();
Delrules lmixenv.elims;
force 1;
force 1;
qed "determ_lam";

Goal "(<Atom v,[v';v;M,vlist]> : lmixenv & \
\ <Atom v,[v';v;M,vlist']> : lmixenv) --> \
```

```
\  ((<Atom v,[v';v;M,vlist]> |- bodyann ---> d') --> \
\    (<Atom v,[v';v;M,vlist']> |- bodyann ---> d'))";
auto();
qed "determ_varnonemp";

Goal "(<Atom y,[v';v;M,vlist]> : lmixenv & y ~= v & \
\  (<Atom y,vlist> |- bodyann ---> d')) --> \
\  (<Atom y,[v';v;M,vlist]> |- bodyann ---> d')";
auto();
br (lmered RS mp) 1;
auto();
qed "determ_varnonemp2";

val [prem1,prem2] =
Goal "[|y ~= v & <Atom y,[v';v;M,vlist]> : lmixenv & \
\  (<Atom y,[v';v;M,vlist]> |- bodyann ---> d); \
\ [|<Atom y,vlist> : lmixenv; <Atom y,vlist> |- bodyann ---> d|] \
\ ==> P|] ==> P";
br prem2 1;
by (subgoal_tac "y ~= v & <Atom y,[v';v;M,vlist]> : lmixenv & \
\ (<Atom y,[v';v;M,vlist]> |- bodyann ---> d)" 1);
br prem1 2;
br (lmered RS mp) 1;
force 1;
by (subgoal_tac "y ~= v & <Atom y,[v';v;M,vlist]> : lmixenv & \
\ (<Atom y,[v';v;M,vlist]> |- bodyann ---> d)" 1);
br prem1 2;
force 1;
qed "derivelimvar1";
```

```
(*********************************************************************
*
* Dispatch.ML
*
*********************************************************************)

Goal "(<[C|q],vlist> : lmixenv) --> \
\ (<[C|q],vlist> |- [IsEq|[[Car|[expr|nil]]|[[Quote|[C|nil]]|nil]]] \
\ ---> d) = (d = #t)";
auto();
by (deriv_forw 1);
qed "iseqtrue";

Goal "(C ~= C' & <[C|q],vlist> : lmixenv) --> \
\ (<[C|q],vlist> |- [IsEq|[[Car|[expr|nil]]|[[Quote|[C'|nil]]|nil]]] \
\ ---> d) = (d = nil)";
auto();
by (deriv_forw 1);
qed "iseqfalse";

Addsimps [iseqtrue,iseqfalse];

Goal "(<Atom w,vlist> : lmixenv) --> \
\ (<Atom w,vlist> |- [IsAtom|[expr|nil]] ---> d) = (d = #t)";
auto();
qed "isatomtrue";

Goal "(<[C|q],vlist> : lmixenv) --> \
\ (<[C|q],vlist> |- [IsAtom|[expr|nil]] ---> d) = (d = nil)";
```

```
auto();
qed "isatomfalse";

Addsimps [isatomtrue,isatomfalse];

Goal "<Atom w,vlist> : lmixenv --> \
\ lookup (<Atom w,vlist>,expr) = Atom w";
auto();
qed "lookup_atom";
Addsimps [lookup_atom];

Delrules l2elims;
Delrules l2eval.intrs;
Delsimps l2eval.intrs;
Delrules lmixenv.elims;

(********************)
(* Dispatch Lemmata *)
(********************)
Goalw [bodyann_def] "(<Atom w,vlist> : lmixenv) --> \
\ (<Atom w,vlist> |- bodyann ---> d') = \
\ (<Atom w,vlist> |- [App|[env|[expr|nil]]] ---> d')";
auto();
by (SELECT_GOAL (auto_tac (claset() addSIs l2eval.intrs, simpset())) 2);
by (deriv_back 1);
qed "dispatch_var";

Goalw [bodyann_def] "(<[Quote|[d|nil]],vlist> : lmixenv) --> \
\ (<[Quote|[d|nil]],vlist> |- bodyann ---> d') = \
\ (<[Quote|[d|nil]],vlist> |- [Lift|[[Cadr|[expr|nil]]|nil]] ---> d')";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_quote";

Goalw [bodyann_def] "(<[Lam|[Atom v|[M|nil]]],vlist> : lmixenv) --> \
\ (<[Lam|[Atom v|[M|nil]]],vlist> |- bodyann ---> d') = \
\ (<[Lam|[Atom v|[M|nil]]],vlist> |- [RLam| \
\ [value|[[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[envlam|nil]]]| \
\ nil]]] ---> d')";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_lam";

Goalw [bodyann_def] "(<[App|[M|[N|nil]]],vlist> : lmixenv) --> \
\ (<[App|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
\ (<[App|[M|[N|nil]]],vlist> |- [RApp| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]| \
\ [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]] \
\ ---> d')";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_app";
```

```
Goalw [bodyann_def] "(<[Fix|[M|nil]],vlist> : lmixenv) --> \
\ (<[Fix|[M|nil]],vlist> |- bodyann ---> d’) = \
\ (<[Fix|[M|nil]],vlist> |- [RFix| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]] \
\ ---> d’)";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_fix";

Goalw [bodyann_def] "(<[Cond|[M|[N|[P|nil]]]],vlist> : lmixenv) --> \
\ (<[Cond|[M|[N|[P|nil]]]],vlist> |- bodyann ---> d’) = \
\ (<[Cond|[M|[N|[P|nil]]]],vlist> |- [RCond| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]| \
\ [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]| \
\ [[App|[[App|[eval|[[Cadddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]]] \
\ ---> d’)";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_if";

Goalw [bodyann_def] "(<[Cons|[M|[N|nil]]],vlist> : lmixenv) --> \
\ (<[Cons|[M|[N|nil]]],vlist> |- bodyann ---> d’) = \
\ (<[Cons|[M|[N|nil]]],vlist> |- [RCons| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]| \
\ [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]] \
\ ---> d’)";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_cons";

Goalw [bodyann_def] "(<[Car|[M|nil]],vlist> : lmixenv) --> \
\ (<[Car|[M|nil]],vlist> |- bodyann ---> d’) = \
\ (<[Car|[M|nil]],vlist> |- [RCar| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]] \
\ ---> d’)";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_car";

Goalw [bodyann_def] "(<[Cdr|[M|nil]],vlist> : lmixenv) --> \
\ (<[Cdr|[M|nil]],vlist> |- bodyann ---> d’) = \
\ (<[Cdr|[M|nil]],vlist> |- [RCdr| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]] \
\ ---> d’)";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_cdr";
```

```

Goalw [bodyann_def] "(<[IsEq|[M|[N|nil]]],vlist> : lmixenv) --> \
\ (<[IsEq|[M|[N|nil]]],vlist> |- bodyann ---> d') = \
\ (<[IsEq|[M|[N|nil]]],vlist> |- [RIsEq| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]| \
\ [[App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[env|nil]]]|nil]]] \
\ ---> d')";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_iseq";

Goalw [bodyann_def] "(<[IsAtom|[M|nil]],vlist> : lmixenv) --> \
\ (<[IsAtom|[M|nil]],vlist> |- bodyann ---> d') = \
\ (<[IsAtom|[M|nil]],vlist> |- [RIsAtom| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]] \
\ ---> d')";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_isatom";

Goalw [bodyann_def] "(<[Error|[M|nil]],vlist> : lmixenv) --> \
\ (<[Error|[M|nil]],vlist> |- bodyann ---> d') = \
\ (<[Error|[M|nil]],vlist> |- [RError| \
\ [[App|[[App|[eval|[[Cadr|[expr|nil]]|nil]]]|[env|nil]]]|nil]] \
\ ---> d')";
auto();
by (deriv_forw 2);
back();
by (deriv_back 1);
qed "dispatch_error";

val dispatch_lemmata = [dispatch_var,dispatch_quote,dispatch_lam,
        dispatch_app,dispatch_fix,dispatch_if,dispatch_cons,dispatch_car,
        dispatch_cdr,dispatch_iseq,dispatch_isatom,dispatch_error];

Addsimps dispatch_lemmata;
```

```
(**********************************************************************
*
* EnvLem.ML
*
**********************************************************************)

Goal "vlist : lmixrlist --> y' : var --> \
\ (vars (lenv (addvar y' vlist)) <= \
\ vars (lenv vlist) Un {y'})";
br impI 1;
by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
force 1;
force 1;
br impI 1;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
force 1;
qed "lenv_lemma_add";

```

```
Goal "vlist : lmixrlist --> [y';y;My,vlist] : lmixrlist --> \
\ (addvar y' vlist : lmixrlist)";
br impI 1;
by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
force 1;
force 1;
br impI 1;
by (eres_inst_tac [("a","[y';y;My,[v';v;M,vlista]]")] lmixrlist.elim 1);
force 1;
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (subgoal_tac "[y';y;My,vlista] : lmixrlist" 1);
by (swap_res_tac lmixrlist.intrs 2);
force 2;
force 2;
force 2;
force 2;
force 3;
by (asm_full_simp_tac (simpset() addsimps
        [lwrap_def,bodyann_def,sintann_def]) 2);
by (mp_tac 1);
by (swap_res_tac lmixrlist.intrs 1);
force 1;
force 1;
force 1;
force 1;
by (asm_full_simp_tac (simpset() addsimps
        [lwrap_def,bodyann_def,sintann_def]) 1);
by (REPEAT (etac conjE 1));
br conjI 1;
force 1;
br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 1;
force 1;
force 1;
force 1;
by (res_inst_tac [("y","vlista")] rlist.exhaust 1);
force 1;
by (Asm_full_simp_tac 1);
by (REPEAT (etac conjE 1));
force 1;
qed "add_lemma1";

Goal "<M,[v';v;Mv,vl]> : lmixenv --> <M,vl> : lmixenv";
br impI 1;
by (eresolve_tac lmixenv.elims 1);
by (res_inst_tac [("y","vl")] rlist.exhaust 2);
auto();
force 1;
qed "lmered";

Goal "((vlist # vlist') : lmixrlist) --> \
\ (vlist : lmixrlist & vlist' : lmixrlist)";
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
auto();
by (subgoal_tac
        "atom1 ~: vars <[Lam|[Atom atom2|[sexpr|nil]]],rlist>" 1);
br (vars_lemma4 RS sub_lemma1) 2;
force 2;
```

```
by (subgoal_tac "!y' y My vlisty. rlist = [y';y;My,vlisty] --> \
\ vars [Lam|[Atom atom2|[sexpr|nil]]] <= vars My" 1);
force 1;
auto();
qed "lconcat_lemma1";

Goal "(<Atom v,vlist> : lmixenv & v ~: varsin vlist) --> \
\ ((<Atom v,vlist> |- bodyann ---> d) = (d = (Atom v)))";
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
force 1;
br impI 1;
be conjE 1;
by (subgoal_tac "<Atom v,rlist> : lmixenv" 1);
by (res_inst_tac [("v'1","atom1"),("v1","atom2"),("Mv1","sexpr")]
        (lmered RS mp) 2);
force 1;
auto();
qed "lemma_511";

Goal "(<M,vlist> : lmixenv & y ~: vars <M,vlist>) --> \
\ (y ~: vars M Un varsin vlist)";
br impI 1;
be conjE 1;
by (eresolve_tac lmixenv.elims 1);
by (Asm_full_simp_tac 1);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (subgoal_tac "y ~: varsin vlist" 1);
by (res_inst_tac [("vlist2","vlist"),("a","y"),("M2","M")]
        ((vars_lemma1 RS mp) RS sub_lemma1) 2);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
auto();
qed "env_lemma1";

Addsimps [env_lemma1];

Goal "(vlist'' : lmixrlist) --> \
\ ((vlist'' # [y';y;My,[z';z;Mz,vlist']]) : lmixrlist) --> \
\ ((vlist'' # [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]) \
\ : lmixrlist)";
br impI 1;
by (res_inst_tac [("xa","vlist''")] lmixrlist.induct 1);
force 1;
by (SELECT_GOAL Auto_tac 1);
by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
by (Asm_full_simp_tac 1);
by (subgoal_tac "[y';y;[App|[Mz|[Atom z|nil]]],[]] : lmixrlist" 1);
by (subgoal_tac "y' ~: \
\ vars <[Lam|[Atom y|[[App|[Mz|[Atom z|nil]]]|nil]]],[]>" 2);
force 2;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
by (subgoal_tac "vars [Lam|[Atom z|[Mz|nil]]] \
\ <= vars [App|[Mz|[Atom z|nil]]]" 1);
force 2;
by (subgoal_tac "z' ~: vars <[Lam|[Atom z|[Mz|nil]]],[y';y; \
\ [App|[Mz|[Atom z|nil]]],[]]>" 1);
by (swap_res_tac lmixrlist.intrs 1);
```

```
by (REPEAT (Force_tac 1));
by ((asm_full_simp_tac (simpset() addsimps
        [lwrap_def,sintann_def,bodyann_def]) 1) THEN (Force_tac 1));
by (Asm_full_simp_tac 1);
by (subgoal_tac
"[y';y;[App|[Mz|[Atom z|nil]]],[atom1;atom2;sexpr,rlist]] : lmixrlist" 1);
by (subgoal_tac "y' ~: \
\ vars <[Lam|[Atom y|[[App|[Mz|[Atom z|nil]]]|nil]]],[atom1; \
\ atom2;sexpr,rlist]>" 2);
by (subgoal_tac "vars [Lam|[Atom y|[[App|[Mz|[Atom z|nil]]]|nil]]] \
\ <= vars sexpr" 2);
force 2;
force 2;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
by (swap_res_tac lmixrlist.intrs 1);
force 1;
force 1;
force 1;
force 1;
by (subgoal_tac "vars [Lam|[Atom z|[Mz|nil]]] \
\ <= vars [App|[Mz|[Atom z|nil]]]" 1);
force 2;
by (subgoal_tac "z' ~: vars <[Lam|[Atom z|[Mz|nil]]], \
\ [y';y;[App|[Mz|[Atom z|nil]]],[atom1;atom2;sexpr,rlist]]>" 1);
by ((asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2)
        THEN (Force_tac 2));
by (swap_res_tac lmixrlist.intrs 1);
force 1;
auto();
by (swap_res_tac lmixrlist.intrs 1);
force 1;
force 1;
force 1;
force 1;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
by (Asm_full_simp_tac 1);
force 1;
by (Asm_full_simp_tac 1);
qed "env_lemma2";

Goal "(<M,vlist'' # [y';y;My,[z';z;Mz,vlist']]> : lmixenv) --> \
\ (<M,vlist'' # [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]> \
\ : lmixenv)";
auto();
by (eresolve_tac lmixenv.elims 1);
force 1;
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (subgoal_tac "(vlist'' # \
\ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]) : lmixrlist" 1);
by (subgoal_tac "vlist'' : lmixrlist" 2);
by (res_inst_tac [("My2","My")] ((env_lemma2 RS mp) RS mp) 2);
force 2;
force 2;
br ((lconcat_lemma1 RS mp) RS conjE) 2;
force 2;
force 2;
```

```
by (res_inst_tac [("y","vlist''")] rlist.exhaust 1);
by (Asm_full_simp_tac 1);
by (swap_res_tac lmixenv.intrs 1);
force 1;
force 1;
by (eresolve_tac lmrelims 1);
force 1;
auto();
qed "env_lemma3";

AddSEs lmixenv.elims;

Goal "(vlist'' : lmixrlist) --> \
\ ((vlist'' # [y';y;My,[z';z;Mz,vlist']]) : lmixrlist) --> \
\ ((vlist'' # [y';y;My,addvar z' vlist']) : lmixrlist)";
br impI 1;
by (res_inst_tac [("xa","vlist''")] lmixrlist.induct 1);
force 1;

by (SELECT_GOAL Auto_tac 1);
by (swap_res_tac lmixrlist.intrs 1);
by (subgoal_tac "[z';z;Mz,vlist'] : lmixrlist" 1);
force 2;
by (res_inst_tac [("y2","z"),("My2","Mz")]
        ((add_lemma1 RS mp) RS mp) 1);
force 1;
force 1;
force 1;
force 1;
force 1;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 1;
force 1;
force 1;
force 1;
by (res_inst_tac [("y","vlist'")] rlist.exhaust 1);
force 1;
force 1;

auto();
by (res_inst_tac [("vlist2","vlist"),
        ("vlist'2","[y';y;My,[z';z;Mz,vlist']]")]
        ((lconcat_lemma1 RS mp) RS conjE) 1);
force 1;

by (swap_res_tac lmixrlist.intrs 1);
force 1;
force 1;
force 1;
force 1;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 1;
force 1;
force 1;
force 1;

by (res_inst_tac [("y","vlist")] rlist.exhaust 1);
auto();
```

```
qed "env_lemma4";

Delrules lmixenv.elims;

Goal "(<M,vlist'' # [y';y;My,[z';z;Mz,vlist']]> : lmixenv) --> \
\ (<M,vlist'' # [y';y;My,addvar z' vlist']> : lmixenv)";
auto();
by (eresolve_tac lmixenv.elims 1);
force 1;
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (subgoal_tac "(vlist'' # [y';y;My,addvar z' vlist']) : lmixrlist" 1);
by (subgoal_tac "vlist'' : lmixrlist" 2);
by (res_inst_tac [("z'2","z'"),("z2","z"),("Mz2","Mz")]
        ((env_lemma4 RS mp) RS mp) 2);
force 2;
force 2;
br ((lconcat_lemma1 RS mp) RS conjE) 2;
force 2;
force 2;
by (res_inst_tac [("y","vlist''")] rlist.exhaust 1);
auto();
qed "env_lemma5";

AddSEs lmixenv.elims;

Goal "(M : l1expr & (vars M <= vars M') & (<M',vlist> : lmixenv)) \
\ --> <M,vlist> : lmixenv";
auto();
by (swap_res_tac lmixenv.intrs 1);
auto();
qed "env_lemma6";

Delrules lmixenv.elims;

use "Determ";

Goal "vlist : lmixrlist --> <Atom v, [z';z;Mz,vlist]> : lmixenv --> \
\ (!d. (<Atom v,vlist> |- bodyann ---> d) --> \
\ (<Atom v, addvar z' vlist> |- bodyann --->d))";
br impI 1;
by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
force 1;
force 1;
br impI 1;
by (eresolve_tac lmixenv.elims 1);
force 1;
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (eres_inst_tac [("a","[z';z;Mz,[v';va;M,vlista]]")]
        lmixrlist.elim 1);
force 1;
by (dtac sym 1);
by (Asm_full_simp_tac 1);
by (subgoal_tac "[z';z;Mz,vlista] : lmixrlist" 1);
by (swap_res_tac lmixrlist.intrs 2);
force 2;
force 2;
```

```
force 2;
force 2;
force 3;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);

by (subgoal_tac "addvar z' vlista : lmixrlist" 1);
by (res_inst_tac [("y2","z"),("My2","Mz")]
        ((add_lemma1 RS mp) RS mp) 2);
force 2;
force 2;

by (subgoal_tac "<Atom v,[z';z;Mz,vlista]> : lmixenv" 1);
by (swap_res_tac lmixenv.elims 2);
force 2;
force 2;
force 2;
by (mp_tac 1);

br allI 1;
br impI 1;
by (subgoal_tac "<Atom v, \
\ [v';va;[App|[M|[Atom z'|nil]]],addvar z' vlista]> : lmixenv" 1);
by (swap_res_tac lmixenv.intrs 2);
force 2;
by (swap_res_tac lmixrlist.intrs 2);
force 2;
force 2;
force 2;
force 2;
by (res_inst_tac [("y","vlista")] rlist.exhaust 3);
force 3;
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
force 3;
by (eres_inst_tac [("a","<Atom v,[z';z;Mz,vlista]>")] lmixenv.elim 3);
force 3;
by (Asm_full_simp_tac 3);
by (REPEAT (etac conjE 3));
force 3;
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
br conjI 2;
force 2;
br (((lenv_lemma_add RS mp) RS mp) RS sub_lemma1) 2;
force 2;
force 2;
force 2;

by (case_tac "v = va" 1);
auto();

by (res_inst_tac [("y","v"),("v","va"),("v'","v'"),("M","M"),
        ("vlist","vlista"),("d","d")] derivelimvar1 3);
br conjI 3;
force 3;
br conjI 3;
force 4;
force 4;
by (swap_res_tac lmixenv.intrs 3);
```

```
force 3;
force 3;
force 3;

by (res_inst_tac [("y","vlista")] rlist.exhaust 2);
by (Asm_simp_tac 2);
by (Asm_simp_tac 2);
by (swap_res_tac lmixenv.intrs 2);
force 2;
force 2;

by (Asm_full_simp_tac 2);
by (subgoal_tac "v : vars sexpr" 2);
force 2;
force 2;

by (subgoal_tac "<Atom v,[v';v;M,vlista]> : lmixenv" 1);
by (swap_res_tac lmixenv.intrs 2);
force 2;
force 2;
force 2;
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (rotate_tac 1 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
by (thin_tac "?xx" 1);
force 1;
qed "add_lemma2";

AddSEs lmixenv.elims;
Goal "[|<Atom v, [z';z;Mz,vlist]> : lmixenv; <Atom v,vlist> \
\ |- bodyann --->d|]  ==> <Atom v, addvar z' vlist> |- bodyann ---> d";
by (res_inst_tac [("vlist3","vlist"),("v3","v"),("z'3","z'"),("z3","z"),
        ("Mz3","Mz"),("x","d")] (((add_lemma2 RS mp) RS mp) RS allE) 1);
auto();
qed "add_lemma2b";
Delrules lmixenv.elims;

Goal "(lconcat (lconcat vl1 vl2) vl3) = \
\ (lconcat vl1 (lconcat vl2 vl3))";
by (res_inst_tac [("rlist","vl1")] rlist.induct 1);
auto();
qed "lcdist";
```

```
Addsimps [lcdist];

Goal "(y = z & M : l1expr) --> (!vlist. \
\ (<M,vlist # [y';y;My,[z';z;Mz,vlist']]> : lmixenv) --> \
\ (!d. (<M,vlist # [y';y;My,[z';z;Mz,vlist']]> |- bodyann ---> d) \
\ --> (<M,vlist # [y';y;My,addvar z' vlist']> |- bodyann ---> d)))";
br impI 1;
by (res_inst_tac [("xa","M")] l1expr.induct 1);

(** App case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
        (((determ_app RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Fix case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
        (((determ_fix RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Cond case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
        (((determ_if RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Cons case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
        (((determ_cons RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;
```

```
(** Car case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
        (((determ_car RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Cdr case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
   (((determ_cdr RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** IsEq case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
   (((determ_iseq RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** IsAtom case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
   (((determ_isatom RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Error case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
        (((determ_error RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 6);
force 5;
force 5;
force 5;
```

```
force 5;

(** Triv case **)
force 1;

(** Quote case **)
by (SELECT_GOAL Auto_tac 2);
by (res_inst_tac [("vlist2","vlist # [y';z;My,[z';z;Mz,vlist']]")]
   ((determ_quote RS mp) RS mp) 2);
br conjI 2;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 3);
force 2;
force 2;
force 2;

(** Lam case **)
by (SELECT_GOAL Auto_tac 2);
by (res_inst_tac [("vlist3","vlist # [y';z;My,[z';z;Mz,vlist']]")]
        (((determ_lam RS mp) RS mp) RS mp) 2);
force 4;

br conjI 2;
force 2;
br conjI 2;
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 2);
force 2;

by (eres_inst_tac
        [("a","<Ma,[v';a;Ma,vlist # [y';z;My,[z';z;Mz,vlist']]]>")]
        lmixenv.elim 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (subgoal_tac "(vlist # [y';z;My,[z';z;Mz,vlist']]) : lmixrlist" 2);
force 3;
by (res_inst_tac [("vlist2","vlist"),
        ("vlist'2","[y';z;My,[z';z;Mz,vlist']]")]
        ((lconcat_lemma1 RS mp) RS conjE) 2);
force 2;
by (subgoal_tac "y' : var" 2);
force 3;
by (subgoal_tac "z : var" 2);
force 3;
by (subgoal_tac "z' : var" 2);
force 3;

by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
by (SELECT_GOAL Auto_tac 2);
by (rename_tac "Ma a vlist dM v' q" 2);
by (subgoal_tac "q : vars (lenv vlist') Un {z'}" 2);
force 2;
br (((lenv_lemma_add RS mp) RS mp) RS subsetD) 2;
force 2;
force 2;
force 2;
```

```
br allI 2;
br impI 2;
br allI 2;
br impI 2;
by (eres_inst_tac [("x","(vlist'' # vlist)")] allE 2);
by (Asm_full_simp_tac 2);

(** Var case **)

br allI 1;
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);

(** induct start **)
by (SELECT_GOAL Auto_tac 1);
by (case_tac "a = z" 1);
by (SELECT_GOAL Auto_tac 1);
by (res_inst_tac [("vlist2","[z';z;Mz,vlist']")]
        ((determ_varnonemp RS mp) RS mp) 1);
br conjI 1;
force 1;
force 2;

by (subgoal_tac "<Atom z,[] # [y';z;My,addvar z' vlist']> : lmixenv" 1);
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 2);
force 1;
force 1;

br (derivelimvar1) 1;
force 1;
br (derivelimvar1) 1;
force 1;

br (determ_varnonemp2 RS mp) 1;
br conjI 1;
by (subgoal_tac "<Atom a,[] # [y';z;My,addvar z' vlist']> : lmixenv" 1);
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 2);
force 1;
force 1;
br conjI 1;
force 1;

br add_lemma2b 1;
force 1;
force 1;

(** induct step **)
br impI 1;
br allI 1;
br impI 1;
by (case_tac "a = atom2" 1);
by (Asm_full_simp_tac 1);

by (subgoal_tac "<Atom atom2,[atom1;atom2;sexpr,rlist] \
\ # [y';z;My,addvar z' vlist']> : lmixenv" 1);
by (res_inst_tac [("z'1","z'"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 2);
```

```
force 1;
force 1;

br (derivelimvar1) 1;
force 1;
by (Asm_full_simp_tac 1);
by (eres_inst_tac [("x","d")] allE 1);
by (Asm_full_simp_tac 1);

br (determ_varnonemp2 RS mp) 1;
br conjI 1;
force 2;
by (subgoal_tac "<Atom a,[atom1;atom2;sexpr,rlist] \
\ # [y’;z;My,addvar z’ vlist’]> : lmixenv" 1);
by (res_inst_tac [("z’1","z’"),("z1","z"),("Mz1","Mz")]
        (env_lemma5 RS mp) 2);
force 1;
force 1;
qed "lemma_513";
```

```
(**********************************************************************
*
* Eval.ML
*
**********************************************************************)

Goalw [sintann_def] "(nil |- sintann ---> d) = \
\ (d = [clos|[expr|[[Lam|[env|[bodyann|nil]]]|[ \
\ [[eval|[delay|[sintann|[nil|nil]]]]|nil]|nil]]]])";
auto();
qed "nilsint";

Addsimps [nilsint];

Goal "(nil |- [App|[[App|[sintann|[[Quote|[M|nil]]|nil]]]|[ \
\ [Lam|[x|[x|nil]]]|nil]]] ---> d) = \
\ (<M,[]> |- bodyann ---> d)";
auto();
by (rewtac lwrap_def);
force 1;
force 1;
qed "sintstart";

Goal "(<M,vlist> : lmixenv) --> \
\ (<M,vlist> |- expr ---> d) = (d = M)";
auto();
qed "eval_expr";

Goal "(<M,vlist> |- eval ---> d) = \
\ (d = [clos|[expr|[[Lam|[env|[bodyann|nil]]]|[[[eval|[delay| \
\ [sintann|[nil|nil]]]]|nil]|nil]]]])";
auto();
qed "eval_eval";

Goal "(<M,[]> |- env ---> d) = (d = [clos|[x|[x|[nil|nil]]]])";
auto();
qed "eval_envnone";

Goal "(<M,[v’;v;M’,vlist]> : lmixenv) --> \
```

```
\ (<M,[v';v;M',vlist]> |- env ---> d) = (d = \
\ [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]]]])";
auto();
qed "eval_envnonemp";

Addsimps [eval_expr,eval_eval,eval_envnone,eval_envnonemp];
```

```
(*******************************************************************
*
* LEnv.ML
*
*******************************************************************)

Addsimps [lconcat_none,lconcat_nemp];
Addsimps [varsin_none,varsin_nemp];

Goal "(Atom w : l1exprs & [v';v;M',vlist] : lmixenv & \
\ w ~: varsin [v';v;M',vlist]) --> (w ~= v & w ~: varsin vlist)";
auto();
qed "atomin_lemma1";

Delsimps [varsin_none,varsin_nemp];

(*********)
(*********)

Goal "(Atom v : l1exprs & vlist : lmixenv) --> \
\ (v ~: varsin vlist) --> \
\ ((<Atom v,vlist> |- bodyann ---> d) = (d = (Atom v)))";
br impI 1;
br lmixenv.induct 1;
force 1;
br impI 1;
br (derivvaremp RS mp) 1;
force 1;
br impI 1;
by (subgoal_tac "[v';va;M',vlista] : lmixenv & \
\ v ~= va & v ~: varsin vlista" 1);
br conjI 2;
by (res_inst_tac [("w1","v"),("v1","va"),("v'1","v'"),("M'1","M'"),
        ("vlist1","vlista")] (atomin_lemma1 RS mp) 3);
by (resolve_tac lmixenv.intrs 2);
by (REPEAT (Force_tac 2));
br conjI 2;
force 2;
br conjI 2;
force 3;
by (resolve_tac lmixenv.intrs 2);
by (REPEAT (Force_tac 2));
auto();
qed "lemma_511";

(*********)
(*********)

Addsimps [varsin_none,varsin_nemp,vars_atom,vars_pair];
```

```
Goal "(vlist : lmixenv) --> (varsin vlist <= vars (lenv vlist))";
br impI 1;
br lmixenv.induct 1;
by ((simp_tac (simpset() addsimps [lwrap_def]) 3) THEN Auto_tac);
qed "whee67";

Addsimps [whee67];
Delsimps [varsin_none,varsin_nemp];
Delsimps [lconcat_none,lconcat_nemp];
```

```
(*********************************************************************
*
* LWrap.ML
*
*********************************************************************)

(***
*
* Results regarding vars.
*
***)

Addsimps [lwrap_def];

Addsimps [vars_atom,vars_pair];

Goal "(vlist : lmixrlist) --> varsin vlist <= vars <M,vlist>";
br impI 1;
by (res_inst_tac [("xa","vlist")] lmixrlist.induct 1);
auto();
qed "vars_lemma1";

Addsimps [bodyann_def,sintann_def];
Goal "vars [App|[[App|[eval|[[Caddr|[expr|nil]]|nil]]]|[envlam|nil]]] <= \
\ vars <M,vlist>";
auto();
qed "vars_lemma2";
Delsimps [bodyann_def,sintann_def];

Addsimps [vars_lemma1,vars_lemma2];

Delsimps [vars_atom,vars_pair];

(***
*
* Results regarding lookup.
*
***)

Goal "(<M,vlist> : lmixenv | (!q. M ~= [delay|q])) --> \
\ (lookup (<M,vlist>,expr) = M)";
auto();
qed "lookup_expr";

Goal "(<M,vlist> : lmixenv | (!q. M ~= [delay|q])) --> \
\ (lookup (<M,vlist>,expr) = d) = (d = M)";
auto();
qed "lookup_expr2";

```

```
Goal "(lookup (<M,vlist>,eval) = [delay|[sintann|[nil|nil]]])";
auto();
qed "lookup_eval";

Goal "(lookup (<M,vlist>,eval) = d) = (d = [delay|[sintann|[nil|nil]]])";
auto();
qed "lookup_eval2";

Goal "(lookup (<M,[]>,env) = [clos|[x|[x|[nil|nil]]]])";
auto();
qed "lookup_none";

Goal "(lookup (<M,[]>,env) = d) = (d = [clos|[x|[x|[nil|nil]]]])";
auto();
qed "lookup_none2";

Goal "lookup (<M,[v';v;M',vlist]>,env) = \
\ [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]]]]";
auto();
qed "lookup_nonemp";

Goal "(lookup (<M,[v';v;M',vlist]>,env) = d) = (d = \
\ [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]]]])";
auto();
qed "lookup_nonemp2";

Addsimps [lookup_expr,lookup_expr2,lookup_eval,lookup_eval2];
Addsimps [lookup_none,lookup_none2,lookup_nonemp,lookup_nonemp2];

Delsimps [lwrap_def];
```

```
(*********************************************************************
*
* LemAlpha.ML
*
*********************************************************************)

Goal "M : l1expr ==> size (rename (a,b,M)) = size M";
br l1expr.induct 1;
auto();
qed "rensz";

Goal "M : l1expr ==> rename (a,a,M) = M";
br l1expr.induct 1;
auto();
qed "renid";

Addsimps [rensz,renid];

Goal "! M : l1expr. size M < i --> \
\ (!dM. (<M,[]> |- bodyann ---> dM) --> (M -a- dM))";
by (induct_tac "i" 1);
force 1;
by (simp_tac (simpset() addsimps [less_Suc_eq]) 1);
auto();
```

```
by (subgoal_tac "<M,[]> : lmixenv" 1);
force 2;
by (rotate_tac ~1 1);
by (eresolve_tac l1expr.elims 1);

force 1;
force 1;
by (REPEAT (Force_tac 2));

(** Lam case **)
auto();

by (res_inst_tac [("M2","Ma"),("x","[]"),("v’2","v’"),("v2","a"),
        ("Mv2","Ma")] ((lemma_515 RS mp) RS allE) 1);
force 1;
by (Asm_full_simp_tac 1);
by (subgoal_tac "<rename (a,v’,Ma),[]> : lmixenv" 1);
by (swap_res_tac lmixenv.intrs 2);
br (renl1 RS mp) 2;
by (eres_inst_tac [("a","<Ma,[v’;a;Ma,[]]>")] lmixenv.elim 2);
force 2;
force 2;

by (Asm_full_simp_tac 1);
by (subgoal_tac "rename (a,v’,Ma) : l1expr" 1);
by (eres_inst_tac [("a","<rename (a,v’,Ma),[]>")] lmixenv.elim 2);
force 2;
force 2;
by (eres_inst_tac [("x","rename (a,v’,Ma)")] ballE 1);
force 2;
by (eres_inst_tac [("x","dMa")] allE 1);

by (Asm_full_simp_tac 1);
by (eres_inst_tac [("x","dMa")] allE 1);
by (Asm_full_simp_tac 1);
by (case_tac "a=v’" 1);
force 1;
br alpha.alpha_sym 1;
by (res_inst_tac [("M’","[Lam|[Atom v’|[rename (a,v’,Ma)|nil]]]")]
        alpha.alpha_trans 1);
br alpha.alpha_lam1 1;
by (eres_inst_tac [("a","<Ma,[v’;a;Ma,[]]>")] lmixenv.elim 1);
force 1;
force 1;
br alpha.alpha_sym 1;
force 1;
br alpha.alpha_sym 1;
br alpha.alpha_lam2 1;
by (eres_inst_tac [("a","<Ma,[v’;a;Ma,[]]>")] lmixenv.elim 1);
force 1;
force 1;
by (eres_inst_tac [("a","<Ma,[v’;a;Ma,[]]>")] lmixenv.elim 1);
force 1;
force 2;
auto();
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 1);
qed "lemma_516";
```

```
Goal "M : l1expr --> (? dM : l1expr. (<M,[]> |- bodyann ---> dM))";
br impI 1;
by (res_inst_tac [("M2","M"),("x","[]")] ((lemma_514 RS mp) RS allE) 1);
auto();
qed "lemma_514b";

Goal "M : l1expr --> (<M,[]> |- bodyann ---> dM) --> (M -a- dM)";
br impI 1;
by (res_inst_tac [("x","M"),("i1","size [Fix|[M|nil]]")]
        (lemma_516 RS ballE) 1);
auto();
qed "lemma_516b";

Addsimps [sintstart];

Goal "M : l1expr ==> EX dM : l1expr. (nil |- \
\ [App|[[App|[sintann|[[Quote|[M|nil]]|nil]]]|[ \
\ [Lam|[x|[x|nil]]]|nil]]] ---> dM)";
auto();
br (lemma_514b RS mp) 1;
force 1;
qed "thm_term";

Goal "M : l1expr ==> \
\ ( nil |- [App|[[App|[sintann|[[Quote|[M|nil]]|nil]]]|[[Lam|[x|[x|nil]]]|nil]]] \
\ ---> dM ) --> (M -a- dM)";
auto();
br ((lemma_516b RS mp) RS mp) 1;
auto();
qed "thm_uniq";
```

```
(*********************************************************************
*
* LemEnv.ML
*
*********************************************************************)

Goal "(y ~= z & M : l1expr) --> (!vlist. \
\ (<M,vlist # [y';y;My,[z';z;Mz,vlist']]> : lmixenv) --> \
\ (!d. (<M,vlist # [y';y;My,[z';z;Mz,vlist']]> |- bodyann ---> d) \
\ --> (<M,vlist # [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]> \
\ |- bodyann ---> d)))";
br impI 1;
by (res_inst_tac [("xa","M")] l1expr.induct 1);

(** App case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
        (((determ_app RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Fix case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
```

```
        (((determ_fix RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Cond case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
        (((determ_if RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Cons case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
        (((determ_cons RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Car case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
        (((determ_car RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Cdr case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
   (((determ_cdr RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** IsEq case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
   (((determ_iseq RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
```

```
force 5;
force 5;
force 5;
force 5;

(** IsAtom case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
   (((determ_isatom RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Error case **)
by (SELECT_GOAL Auto_tac 5);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
   (((determ_error RS mp) RS mp) RS mp) 5);
br conjI 5;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 6);
force 5;
force 5;
force 5;
force 5;

(** Triv case **)
force 1;

(** Quote case **)
by (SELECT_GOAL Auto_tac 2);
by (res_inst_tac [("vlist2","vlist # [y';y;My,[z';z;Mz,vlist']]")]
   ((determ_quote RS mp) RS mp) 2);
br conjI 2;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 3);
force 2;
force 2;
force 2;

(** Lam case **)
by (SELECT_GOAL Auto_tac 2);
by (res_inst_tac [("vlist3","vlist # [y';y;My,[z';z;Mz,vlist']]")]
   (((determ_lam RS mp) RS mp) RS mp) 2);
force 4;

br conjI 2;
force 2;
br conjI 2;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
force 2;
by ((asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2)
        THEN (Force_tac 2));

br allI 2;
br impI 2;
br allI 2;
br impI 2;
```

```
by (eres_inst_tac [("x","(vlist’’ # vlist)")] allE 2);
by (Asm_full_simp_tac 2);

(** Var case **)
br allI 1;
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);

(** induct start **)
by (SELECT_GOAL Auto_tac 1);
by (case_tac "a = y" 1);
by (SELECT_GOAL Auto_tac 1);
br (determ_varnonemp2 RS mp) 1;
br conjI 1;
by (subgoal_tac "<Atom y,[] # \
\ [z’;z;Mz,[y’;y;[App|[Mz|[Atom z|nil]]],vlist’]]> : lmixenv" 1);
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
force 1;
force 1;
br conjI 1;
force 1;
by (subgoal_tac "<Atom y,[] # \
\ [y’;y;[App|[Mz|[Atom z|nil]]],vlist’]> : lmixenv" 1);
force 1;
by (subgoal_tac "<Atom y,[] # \
\ [z’;z;Mz,[y’;y;[App|[Mz|[Atom z|nil]]],vlist’]]> : lmixenv" 1);
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
by (Asm_full_simp_tac 1);
br (lmered RS mp) 1;
force 1;
force 1;
by (case_tac "a = z" 1);
by (SELECT_GOAL Auto_tac 1);
by (res_inst_tac [("vlist2","vlist’")]
        ((determ_varnonemp RS mp) RS mp) 1);
br conjI 1;
br (lmered RS mp) 1;
force 1;
by (subgoal_tac "<Atom z,[] # \
\ [z’;z;Mz,[y’;y;[App|[Mz|[Atom z|nil]]],vlist’]]> : lmixenv" 1);
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
force 1;
force 1;
br (derivelimvar1) 1;
force 1;
force 1;
br (derivelimvar1) 1;
force 1;
by (res_inst_tac [("v","z")] derivelimvar1 1);
force 1;
br (determ_varnonemp2 RS mp) 1;
br conjI 1;
by (subgoal_tac "<Atom a,[] # \
\ [z’;z;Mz,[y’;y;[App|[Mz|[Atom z|nil]]],vlist’]]> : lmixenv" 1);
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
force 1;
force 1;
br conjI 1;
force 1;
```

```
br (determ_varnonemp2 RS mp) 1;
br conjI 1;
by (subgoal_tac "<Atom a,[] # \
\ [y';y;[App|[Mz|[Atom z|nil]]],vlist']> : lmixenv" 1);
force 1;
by (subgoal_tac "<Atom a,[] # \
\ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]> : lmixenv" 1);
by (Asm_full_simp_tac 1);
br (lmered RS mp) 1;
force 1;
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 1);
force 1;
force 1;

(** induct step **)
br impI 1;
br allI 1;
br impI 1;
by (case_tac "a = atom2" 1);
by (Asm_full_simp_tac 1);

by (subgoal_tac "<Atom atom2,[atom1;atom2;sexpr,rlist] # \
\ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]> : lmixenv" 1);
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
force 1;
force 1;

br (derivelimvar1) 1;
force 1;
by (Asm_full_simp_tac 1);
by (eres_inst_tac [("x","d")] allE 1);
by (Asm_full_simp_tac 1);

br (determ_varnonemp2 RS mp) 1;
br conjI 1;
force 2;
by (subgoal_tac "<Atom a,[atom1;atom2;sexpr,rlist] # \
\ [z';z;Mz,[y';y;[App|[Mz|[Atom z|nil]]],vlist']]> : lmixenv" 1);
by (res_inst_tac [("My1","My")] (env_lemma3 RS mp) 2);
force 1;
force 1;
qed "lemma_512";
```

```
(*********************************************************************
*
* LemTerm.ML
*
*********************************************************************)

Goal "vars sexpr : Finites";
br sexpr.induct 1;
auto();
qed "varsfin";

Addsimps [varsfin];

Goal "M : l1expr --> (!vlist. (vlist : lmixrlist --> \
\ (<M,vlist> : lmixenv --> \
\ (? dM : l1expr. (<M,vlist> |- bodyann ---> dM)))))";
```

```
br impI 1;
by (res_inst_tac [("xa","M")] l1expr.induct 1);
force 1;
force 2;

(** App case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (subgoal_tac "<N,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Fix case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[Fix|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Cond case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<B,vlist> : lmixenv" 3);
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (subgoal_tac "<N,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (eres_inst_tac [("x","vlist")] allE 3);
by (eres_inst_tac [("x","vlist")] allE 3);
by (eres_inst_tac [("x","vlist")] allE 3);
force 3;

(** Cons case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (subgoal_tac "<N,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Car case **)
br allI 3;
```

```
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[Car|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Cdr case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cdr|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** IsEq case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (subgoal_tac "<N,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** IsAtom case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsAtom|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Error case **)
br allI 3;
br impI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist> : lmixenv" 3);
by (res_inst_tac [("M'1","[Error|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Var case **)
br allI 1;
br impI 1;
br lmixrlist.induct 1;
force 1;
force 1;
br impI 1;
by (case_tac "a = v" 1);
force 1;
by (subgoal_tac "<Atom a,vlista> : lmixenv" 1);
br (lmered RS mp) 2;
force 1;
```

```
force 1;

(** Lam case **)
auto();
by (subgoal_tac "? v'. <Ma,[v';a;Ma,vlist]> : lmixenv" 1);
be exE 1;
by (subgoal_tac "[v';a;Ma,vlist] : lmixrlist" 1);
by (eres_inst_tac [("a","<Ma,[v';a;Ma,vlist]>")] lmixenv.elim 2);
force 2;
force 2;

by (eres_inst_tac [("x","[v';a;Ma,vlist]")] allE 1);
auto();

(** Choose case **)
by (res_inst_tac [("x","@ v'. v' : var & \
\ v' ~: vars <[Lam|[Atom a|[Ma|nil]]],vlist>")] exI 1);
by (thin_tac "?xx" 1);
by (rotate_tac ~3 1);
by (thin_tac "?xx" 1);
by (subgoal_tac "? v'. v' : var & \
\ v' ~: vars <[Lam|[Atom a|[M|nil]]],vlist>" 1);
by (swap_res_tac lmixenv.intrs 1);
force 1;
force 2;
by (swap_res_tac lmixrlist.intrs 1);
force 1;
force 1;
force 1;
by (eresolve_tac lmixenv.elims 3);
force 3;
force 3;
br selectI2EX 1;
force 1;
force 1;
br selectI2EX 1;
force 1;
force 1;
br freshvar 1;
force 1;
qed "lemma_514";
```

```
(*********************************************************************
*
* LemWeak.ML
*
*********************************************************************)

Goal "M : l1expr --> \
\ (!vlist. <M,vlist # [v';v;Mv,vlist']> : lmixenv & \
\          <M,vlist # [v';v;Mv',vlist']> : lmixenv & \
\          vars Mv' <= vars Mv --> \
\          (!dM. (<M,vlist # [v';v;Mv,vlist']> |- bodyann ---> dM) --> \
\                (<M,vlist # [v';v;Mv',vlist']> |- bodyann ---> dM)))";
br impI 1;
by (res_inst_tac [("xa","M")] l1expr.induct 1);

force 1;
force 2;
```

```

(** App case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[App|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Fix case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Fix|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Fix|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Cond case **)
br allI 3;
br impI 3;
by (subgoal_tac "<B,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<B,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cond|[B|[Ma|[N|nil]]]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Cons case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
```

```
by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cons|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Car case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Car|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Car|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Cdr case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cdr|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Cdr|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** IsEq case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<N,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<N,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsEq|[Ma|[N|nil]]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** IsAtom case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsAtom|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[IsAtom|[Ma|nil]]")] (env_lemma6 RS mp) 4);
```

```
force 4;
force 3;

(** Error case **)
br allI 3;
br impI 3;
by (subgoal_tac "<Ma,vlist # [v';v;Mv,vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Error|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
by (subgoal_tac "<Ma,vlist # [v';v;Mv',vlist']> : lmixenv" 3);
by (res_inst_tac [("M'1","[Error|[Ma|nil]]")] (env_lemma6 RS mp) 4);
force 4;
force 3;

(** Lam case **)
by (SELECT_GOAL Auto_tac 2);

by (swap_res_tac lmixenv.intrs 2);
force 2;
by (eres_inst_tac [("a","<[Lam|[Atom a|[Ma|nil]]],vlist # \
\ [v';v;Mv',vlist']>")] lmixenv.elim 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,vlist # [v';v;Mv,vlist']]>")]
        lmixenv.elim 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);

by (swap_res_tac lmixrlist.intrs 2);
force 2;
force 2;
force 2;
force 2;
force 3;
force 3;
by (subgoal_tac "[v';v;Mv',vlist'] : lmixrlist" 2);
br ((lconcat_lemma1 RS mp) RS conjE) 3;
force 3;
force 3;
by (thin_tac "?xx" 2);
by (rotate_tac 1 2);
by (thin_tac "?xx" 2);
by (rotate_tac 1 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (rotate_tac 1 2);
by (thin_tac "?xx" 2);
by (SELECT_GOAL Auto_tac 2);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
force 2;
```

```
by (subgoal_tac "<Ma,[v'a;a;Ma,vlist # [v';v;Mv',vlist']]> : lmixenv" 2);
by (defer_tac 2);

by (swap_res_tac lmixenv.intrs 2);
force 2;
by (eres_inst_tac [("a","<[Lam|[Atom a|[Ma|nil]]],vlist # \
\ [v';v;Mv',vlist']>")] lmixenv.elim 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);
by (eres_inst_tac [("a","<Ma,[v'a;a;Ma,vlist # [v';v;Mv,vlist']]>")]
   lmixenv.elim 2);
force 2;
by (dtac sym 2);
by (Asm_full_simp_tac 2);

by (swap_res_tac lmixrlist.intrs 2);
force 2;
force 2;
force 2;
force 2;
force 3;
force 3;
by (subgoal_tac "[v';v;Mv',vlist'] : lmixrlist" 2);
br ((lconcat_lemma1 RS mp) RS conjE) 3;
force 3;
force 3;
by (thin_tac "?xx" 2);
by (rotate_tac 1 2);
by (thin_tac "?xx" 2);
by (rotate_tac 1 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (thin_tac "?xx" 2);
by (rotate_tac 1 2);
by (thin_tac "?xx" 2);
by (SELECT_GOAL Auto_tac 2);
by (asm_full_simp_tac (simpset() addsimps [lwrap_def]) 2);
force 2;

by (eres_inst_tac [("x","[v'a;a;Ma,vlist]")] allE 2);
force 2;

(** Var case **)
br allI 1;
by (res_inst_tac [("rlist","vlist")] rlist.induct 1);
by (Asm_full_simp_tac 1);
by (SELECT_GOAL Auto_tac 1);
by (case_tac "a = v" 1);
force 1;
br derivelimvar1 1;
force 1;
force 1;
```

```
br impI 1;
br allI 1;
br impI 1;
by (REPEAT (etac conjE 1));
by (case_tac "a = atom2" 1);
by (Asm_full_simp_tac 1);
by (res_inst_tac [("v2","atom2"),("v'2","atom1")]
        ((evaluation_varnonemp RS mp) RS iffD1) 1);
force 1;
force 1;
by (Asm_full_simp_tac 1);
br conjI 1;
br (lmered RS mp) 1;
force 1;
by (subgoal_tac "<Atom a,rlist # [v';v;Mv,vlist']> : lmixenv" 1);
br (lmered RS mp) 2;
force 2;
by (subgoal_tac "<Atom a,rlist # [v';v;Mv',vlist']> : lmixenv" 1);
br (lmered RS mp) 2;
force 2;
auto();
by (eres_inst_tac [("x","dM")] allE 1);
br derivelimvar1 1;
force 1;
force 1;
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. *Similix 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. Æ. Self-applicable partial evaluation for pure lambda calculus. In *Proceedings of PEPM '92* (1992), Yale University Press, pp. 116–121.

[17] MOGENSEN, T. Æ. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of PEPM '95* (1995), ACM Press, pp. 39–44.

[18] MOGENSEN, T. Æ. Inherited limits. In *Partial Evaluation: Practice and Theory* (Copenhagen, July 1998), J. Hatcliff, T. Æ. 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.