Morten Welinder

Partial Evaluation and Correctness

Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK–2100 Copenhagen Ø
Denmark

Contents

Chapter 1

Introduction

There are no such things as applied sciences,
only applications of science.
— LOUIS PASTEUR, 1822–1895

Program transformation techniques like partial evaluation and fold-unfold systems have been studied extensively for a number of years. Surprisingly, the question of *what* partial evaluation really is has been somewhat neglected; usually only the external view is considered, i.e., a partial evaluator is a program that realises Kleene's S_n^m-theorem [Kle52]:

$$\texttt{mix} \text{ is a partial evaluator} \iff \forall \texttt{p}, \texttt{s}, \texttt{d} : [\![\texttt{p}]\!](\texttt{s},\texttt{d}) = [\![\,[\![\texttt{mix}]\!](\texttt{p},\texttt{s})\,]\!](\texttt{d})$$

This description is for many purposes not satisfying because most programming languages allow a trivial implementation, i.e., one that given `p` and `s` just modifies `p` to take just one argument and to assign `s` to a variable of the same name as the eliminated parameter.

Furthermore the correctness of these methods has not been subject to much attention, or — when it has — it has been for partial evaluators lacking important aspects of what we believe partial evaluation is, like memorisation.

Claims of correctness ought to be proven but proofs vary significantly in degree of formalisation, from mere hand-waving to fully expansive decomposition into uses of axioms. The choice of degree of formalisation is a balance between work-efficiency and ease on the one hand and thoroughness and safety on the other: a fully expansive proof can easily run into millions of basic inferences, which obviously is a major undertaking, while an informal proof can leave the reader still in doubt.

Constructing a proof consisting of millions of inferences is beyond human capabilities, and even if it were within the possible such a proof would not fulfil its purpose: the chance of mistakes increases with the size of the job and the size of the proof would have an intimidating effect on readers wanting to check the theorem for themselves. But checking a proof expressed in terms of basic inferences is a trivial undertaking except for the size of the job. Therefore it is a straightforward idea to have a computer program check the proof; then it is no longer a problem that a proof might be large and the need to trust the proof has been reduced to the need to trust the program checking the proof — hopefully easier to do.

Having a program check a proof is good but still leaves the gigantic problem of creating the fully expanded proofs in the first place. Proving or disproving a theorem is an undecidable problem so we cannot hope to create the proofs automatically in general, and while constructing a proof for a theorem known to hold (known by the human using the computer, that is) is a decidable problem it is not necessarily practical. But this does not mean that we cannot write a program that proves routine problems.

In this dissertation, when the term *proof of a theorem* is used it implies that that every node in the fully expanded inference tree has been constructed for the theorem and that the proof has been verified by a program.

1.1 Thesis

This dissertation is a study into correctness of partial evaluation, an attempt at isolating an intensive description of the nature of partial evaluation, and a study into the feasibility of working with completely formalised and mechanically verified proofs of the correctness of partial evaluation. More formally, the following three points form the thesis behind the present work:

1. *Partial evaluation can be described as the use of a relatively small set of program transformations.*
2. *The correctness of partial evaluation is susceptible to proofs.*
3. *The use of mechanical verification in programming language research is an asset, not a hindrance.*

1.2 Reader's Prerequisites

The reader of this thesis is expected to have some understanding of (classical) logic, for example corresponding to [Men79], and some knowledge of partial

evaluation, for example corresponding to [JGS93]. Being an expert in the field of partial evaluation or program transformations is not necessary: This dissertation being a first attempt at connecting formal proofs and partial evaluation we will concentrate on the basics.

Having some experience with automated theorem proving and theorem verification is beneficial. Theorem provers, however, vary and in Section 1.5 we therefore give a short introduction to the theorem prover actually used.

Moreover, the reader should understand the idea of decomposing a proof into its basic particles (i.e., uses of axioms) and should probably be able to think of this as useful, at least in principle.

1.3 Organisation of this Dissertation

This dissertation is organised in the following way:

In the present chapter, following this description, a brief introduction (Section 1.4) to partial evaluation will be given. It is followed by a — likewise brief — introduction to the HOL theorem prover in Section 1.5.

The thesis which was introduced in Section 1.1 will be elaborated in Chapter 2 which covers the goals of this dissertation and the work behind it.

We will be concerned mostly with partial evaluation of a small language which we develop and describe in Chapter 3. Also in this chapter is the formalisation of the language and its semantics in the HOL theorem prover.

Chapter 4 contains a rather large number of definitions, lemmas and theorems needed to support the chapters following it. The chapter is placed in this position because it logically belongs there. In a first reading the reader might want to skip it and refer to it later when referenced.

In Chapter 5 we introduce an interpreter for the language we work with. This interpreter is written in the language it itself interprets and we can therefore compare the interpreter with the language semantics to see if it implements the language faithfully. We discuss what this means for an interpreter that uses encoding of syntax and values and we prove that the interpreter does implement the semantics faithfully by formally proving three theorems stating its correctness.

In Chapter 6 we explore program transformations that preserve the meaning of the transformed programs. These transformations range from `let`-unfolding and alpha conversion to folding of function calls.

Then in Chapter 7 we show that these program transformations form a system which is strong enough to perform the same program transformations normally attributed to partial evaluation.

We discuss the possibilities of further work in Chapter 8, both in the direction of program transformations and in the direction of machine verified theorem

proving. We go on to discuss related work in Chapter 9 and draw our conclusions in Chapter 10.

Appendix A contains additional information on the difference between traditional mathematical notation and the (linear, ASCII-based) notation used by HOL. Appendix B contains an example showing how the program transformations can be used to partially evaluate Ackermann's function with respect to a known static first argument. Appendix C contains the self-interpreter discussed in Chapter 5 but with all syntactic sugar removed.

1.4 Partial Evaluation

As briefly mentioned above, a partial evaluator is a program that turns a (one-stage) program into an equivalent two-stage program:

$$\texttt{mix} \text{ is a partial evaluator} \iff \forall \texttt{p}, \texttt{s}, \texttt{d} : [\![\texttt{p}]\!](\texttt{s}, \texttt{d}) = [\![[\![\texttt{mix}]\!](\texttt{p}, \texttt{s})]\!](\texttt{d}). \quad (1.1)$$

Running `mix` on the program `p` and one part of its data — the so-called *static* data, `s` — produces a new program, $\texttt{p}_\texttt{s}$, called the *residual program*. The program $\texttt{p}_\texttt{s}$ when run on the remaining part of the input — the so-called *dynamic* data, `d` — produces the same result as running `p` directly on both `s` and `d`. We say that `p` has been specialised with respect to `s`.

In Equation (1.1) we restrict ourselves to programs having exactly two parameters. This is convenient for notational reasons but too restrictive in practice, where `s` and `d` should be relaxed to be arbitrary disjoint parts of the input. For example, in the following program, which calculates the scalar product of two three-dimensional vectors,

```
f (x1,y1,z1,x2,y2,z2) = x1*x2 + y1*y2 + z1*z2;
```

we might want to specialise `f` with respect to $(\texttt{x1}, \texttt{y1}, \texttt{z1}) = (2, 4, 6)$ and expect to get the program

```
f_246 (x2,y2,z2) = 2*x2 + 4*y2 + 6*z2;
```

An important goal for a partial evaluator is to produce residual programs that are more efficient than their source programs. Consider the following program computing Ackermann's function[1]

[1]This function (which in the form shown really should be attributed to Rósza Péter [Pét51] and not to Wilhelm Ackermann) grows so rapidly that is has no practical use. Nevertheless it is traditionally used as an example for partial evaluation with the claim that the residual program "runs faster" than the original. "Runs faster" should be taken to mean "requires fewer evaluation steps" and not as a claim involving a clock.

```
ack (m,n) =
  if m=0 then
    n+1
  else if n=0 then
    ack (m-1,1)
  else
    ack (m-1,ack (m,n-1));
```

This program can be specialised with respect to `m` $= 2$ resulting in (for example) the program

```
ack_2 (n) = if n=0 then 3 else ack_1 (ack_2 (n-1));
ack_1 (n) = if n=0 then 2 else ack_0 (ack_1 (n-1));
ack_0 (n) = n+1;
```

with `ack_2` as the main function. (At this point it is not important how the shown residual program might be constructed, but the main principle is to reduce the application `ack(2,n)`.) Evaluating `ack(2,n)` for some `n` performs a superset of the steps that evaluating `ack_2(n)` does, so clearly the residual program is faster than the original.

1.4.1 The Trivial Partial Evaluator

Most languages allow us to construct a trivial partial evaluator which does nothing but wrap a binding around the program in such a way that the static parts of the input are bound.[2] Using the Ackermann example from above a trivial partial evaluator might produce

```
ack_2 (n) = ack (2,n);
ack (m,n) = ... (* as in source *)
```

when applied to the program and the constant 2 for `m`.

This is analogous to Kleene's S_n^m-theorem (Theorem XXIII in [Kle52]) which states, in effect, that a partial evaluator exists within the system of primitive recursive functions. The trivial partial evaluator for some programming language proves the corresponding theorem for the (usually larger, see for example [Jon97]) system of programs in that language — provided, of course that the partial evaluator can be proven correct. We shall give such a proof in Section 7.1.

Obviously trivial partial evaluators are not very interesting, but their existence shows that the external view of partial evaluation — Equation (1.1) — is not sufficient.

[2] In fact one has to impose quite uncommon restrictions on a language to prevent this from being possible. One way to do it is to require that all functions have exactly two integer parameters: with such a language no program can produce a one-argument function and a partial evaluator as defined by Equation 1.1 is thus not possible.

1.4.2 Partial Evaluation of Interpreters

A special case of partial evaluation is when the program that is being partially evaluated is an interpreter. Assume that `int` is an interpreter for the language L itself written in the language M and that `mix` is a partial evaluator for the language M. Then, for an arbitrary program `p` written in L and its input `d` we have:

$$[\![\,\mathtt{p}\,]\!]_L(\mathtt{d}) = [\![\,\mathtt{int}\,]\!]_M(\mathtt{p},\mathtt{d}) = [\![\,[\![\,\mathtt{mix}\,]\!](\mathtt{int},\mathtt{p})\,]\!]_M(\mathtt{d}). \tag{1.2}$$

(The implementation language of `mix` is irrelevant for the purpose of this equation.) The equation in particular means that the residual program, $[\![\,\mathtt{mix}\,]\!](\mathtt{int},\mathtt{p})$, is an M-program with the same operational behaviour as the L-program `p`. In other words, `p` has been compiled from L to M. This insight dates back 25 years, see [Fut71].

Keeping the existence of trivial partial evaluators in mind this method will not in itself produce efficient compiled programs. However, with a good partial evaluator we might actually achieve a significant improvement over interpretation. This hope has been repeatedly confirmed by experiments, see for example [GJ91a, JGS93, And94, BW93, WCRS91].

A *self-interpreter*, `sint`, is an interpreter for a language L written itself in L. For such an interpreter we see from the above that `p'` $\equiv [\![\,\mathtt{mix}\,]\!](\mathtt{sint},\mathtt{p})$ is `p` compiled from L into L. In other words, this means that `p` and `p'` have the same functionality. But one may still be more efficient than the other.

To most, if not all, programming languages it is possible to add some reasonable measure $\mathit{time}(\mathtt{p},\mathtt{d})$ describing how long evaluation of `p` with input `d` takes. For a language with a small-step semantics — such as, for example, most Turing machine presentations [Jon97, Section 7.6] — the number of steps is a reasonable measure. For an inference-rule based semantics the number of nodes in the evaluation derivation tree is a good measure. And for language defined by a recursive "eval" function the number of calls to this function is a reasonable measure. For non-terminating evaluations we will assign ∞ as the time measure. We shall explore more concrete *timed semantics* — albeit for a different purpose — in Section 3.4 , but for now we shall just assume that we have one.

For any input data we can now compare the time spent on evaluation by `p` and `p'`. We might hope that the partial evaluator was good enough to remove all the overhead of interpretation. Based on this hope and following [JGS93] we define

Definition 1 (Optimality.) *A partial evaluator* `mix` *is said to be* optimal *with respect to* `sint` *if*

$$\forall \mathtt{p},\mathtt{d} : \mathit{time}([\![\,\mathtt{mix}\,]\!](\mathtt{sint},\mathtt{p}),\mathtt{d}) \leq \mathit{time}(\mathtt{p},\mathtt{d}).$$

From a theoretical point of view this is a bad definition: "optimal" should mean "good," but it does not. The following partial evaluator derived from the self-interpreter `sint` and the trivial partial evaluator `triv_pe` is optimal:

```
pe (p,d) = if p=sint then d else triv_pe (p,d);
```

But obviously this is just as bad a partial evaluator as the trivial partial evaluator itself.

From a practical point of view, however, Definition 1 is a good definition. If we do obtain optimality for a partial evaluation without "cheating" then the partial evaluator must have eliminated all the overhead in interpretation.

1.4.3 Termination

Equation (1.1) is evasive on the subject of termination of programs, or equivalently on the domains of the partial functions $[\![\cdot]\!]$. We would like the situation to be such that $[\![\texttt{mix}]\!]$ is a total function and such that the sides of the equality in (1.1) are either both defined or both undefined.

Unfortunately, this goal has proven difficult to obtain at the same time as getting good specialisation: the more aggressively a partial evaluator tries to reduce, the larger the danger of non-termination becomes. For this reason practical evaluators today — for example Similix [BD90, BD91, Bon91, Bon92, Bon93], C-mix [And92, And93, And94], and SML-Mix [BW93, BW94, BW95] — do not guarantee that $[\![\texttt{mix}]\!]$ is total. Since practice shows that this is more a theoretical problem than a practical one, the problem has had a relatively low priority with researchers, see [AH96].

In this dissertation no actual non-trivial partial evaluator is exhibited, only the transformation components that such a program might use. This means that the only termination problem that will occur is that of showing that the original program $\texttt{p}$ and the residual program $\texttt{p}_\texttt{s}$ have identical termination properties. This will be rigorously proven.

1.5 The HOL Theorem Prover

All proofs in this thesis have been formalised in and proven with the use of the HOL ("Higher Order Logic") theorem prover [GM93], more precisely with the HOL 90.7 revision [Sli94] of the system. Henceforth HOL will be used to denote this particular version although the differences in versions are primarily syntactic. The choice of HOL over other theorem provers is somewhat arbitrary and largely guided by availability and existing knowledge. There is no reason to believe that other theorem provers, for example Nuprl [CAB+86], Isabelle [Pau90],

or Coq [DFH+93], could not have been used instead. A system like Elf, on the other hand, would not provide the same proof security.

This section briefly introduces HOL, what it means to prove things in HOL, and the logic that HOL proofs are based on. This is not a substitute for [GM93], but rather an "innocent bystander's introduction." This section will not get into *how* to prove with HOL.

Some details of HOL will be elaborated on in Section 3.6.

1.5.1 HOL Objects

HOL is written in Standard ML [MTH90] (with certain convenience features, to be discussed shortly, added) and uses this language as meta-language for the logic. HOL inherits Standard ML's read-eval loop and is therefore interactive.

The following table shows how various mathematical entities are represented in the HOL/Standard ML world:

Math World	*HOL World*
Term	Value of SML-type `term`.
Theorem	Value of SML-type `thm`.
Type	Value of SML-type `hol_type`.
Formula	Boolean term, i.e., a math-world term with math-world type `bool`.
Proof	Evaluation leading to a value of SML-type `thm`.
Axiom	As theorem.
Primitive inference rule	Function (or constant) with SML-type $\tau_1 \to \cdots \to \tau_n \to$ `thm` for some τ_i's.
Derived rule	As primitive inference rule.

Note that both the mathematical world and the implementation world have a concept of types. These are, however, very different objects and the table therefore uses "SML-type" to denote a type in the implementation world. The mathematical world's type is, in fact, a value in the implementation world.

The Standard ML types `term` and `thm` are defined inside Standard ML modules constrained by signatures that let the types be visible outside the module while not letting the constructors of the types be visible outside the module. This technique has the consequence that it is not possible for the HOL user to introduce arbitrary objects of these types. Terms can be constructed only when they are well-formed and well-typed and theorems can only be obtained by proofs, i.e., ultimately by applications of axioms[3]. Thus the Standard ML type system provides the proof

[3] Actually there is a way to "cheat" and convert arbitrary sequents into objects of type `thm`. Cheating can be useful for quick what-if reasoning, but obviously threatens proof security. Therefore the system can be set up as to always leave a trace of such cheating.

security of HOL.

In order to ease input and output of terms, types, and theorems, HOL makes use of quotation and anti-quotation extensions of otherwise standard ML. Consider for example the following application

```
--‘x /\ y‘--
```

(which is an application of the function `--` to a quotation and itself). This application parses and type checks the quotation and evaluates to a value of Standard ML type `term` representing a conjunction.

Furthermore, as certain mathematical symbols (like $\exists$, $\forall$, and $\Rightarrow$) are not easy to input and output on most terminals HOL uses a transcription with regular characters. Appendix A gives more information in this regard and also — more importantly — gives the correspondence between notation used in this thesis and the notation used by HOL.

Types

HOL is based on typed lambda-calculus with constants. Types in the calculus can be described as those generated by the grammar:

$$\sigma ::= \alpha \mid (\sigma_1, \ldots, \sigma_n)\, op \tag{1.3}$$

where α ranges over type variables and *op* ranges over a set of type operators, each with its own arity, possibly zero. The set of type operators in particular contains $\rightarrow$ of arity 2 and *bool* of arity 0. Some type operators have special support from the parser and the pretty-printer, allowing a more traditional notation like $bool \rightarrow bool$.

HOL requires that all types be inhabited, i.e., that for any type there is a value of that type. For example, the type *bool* has the two elements T and F. This requirement is enforced when defining new types, where a proof of inhabitation is a prerequisite.

Types do not play an important rôle in this thesis, except for their presence in the background securing consistency. Since types are so relatively unimportant to the understanding of this thesis we shall as a notational convenience *leave out types from terms.* A consequence of this abbreviation is that a reader wishing to enter terms written here into HOL must occasionally re-create the polymorphic type annotations. (All other type information HOL will infer itself.) We stress that the actual proof scripts do not include any such sloppiness.

Terms

There are four kinds of terms that HOL knows about. They form a subset of the objects generated by the following grammar:

$$t ::= x \mid c \mid t_1\, t_2 \mid \lambda x.t \tag{1.4}$$

where x ranges over variables and c ranges over constants. The terms are the well-typed ones and there are no surprises in the type rules. Non-well-typed objects generated by the grammar in 1.4 are not terms and will not be considered further.

Since HOL's logic is a higher-order logic it is possible to define, for example, the universal and the existential quantifier within the logic. In fact, $\forall x.t$ and $\exists x.t$ are just (parser and pretty-printer supported) shorthands for $(\$\forall)(\lambda x.t)$ and $(\$\exists)(\lambda x.t)$ respectively, where '$\forall' and '$\exists' are regular higher-order constants (both of type $(\tau \to \texttt{bool}) \to \texttt{bool}$, i.e., taking a predicate and giving a truth value) defined by

$$\begin{aligned} \$\forall &\stackrel{\text{def}}{=} \lambda P.(P = (\lambda x.T)) \\ \$\exists &\stackrel{\text{def}}{=} \lambda P.P(\varepsilon P) \end{aligned}$$

using the Hilbert choice operator, ε, which given a predicate choses an element in the domain for which the predicate holds, or an unspecified element of the domain if the predicate is everywhere false.

For example, the familiar term "$\forall b : (b = T) \vee (b = F)$" is a shorthand for

$$(\$\forall)(\lambda b.(b = T) \vee (b = F))$$

which when expanded by $\forall's definition and beta-reduced becomes

$$(\lambda b.(b = T) \vee (b = F)) = (\lambda x.T).$$

The HOL system's parser and pretty-printer allow several other kinds of special syntax aimed at supporting traditional mathematical notation. This includes, for instance, infix function applications, pairs, and let-expressions. These notational conversions are not expected to cause problems for the reader and therefore will not be introduced formally here.

Sequents, Theorems, and Lemmas

A sequent (or "goal") is the pair of a set of boolean terms, $\{t_1, \ldots, t_n\}$, called the assumptions and a boolean term, t, called the conclusion. Sequents are written $\{t_1, \ldots, t_n\} \vdash^? t$.

Theorems are proven sequents and will be written $\{t_1,\ldots,t_n\} \vdash t$. For proven or un-proven sequents the assumption set will be omitted if it is empty; most of the theorems presented in this thesis will indeed have empty assumption sets. "Lemma" is an alternative term used for theorem. "Lemmas" are generally considered less important than "theorems." Certain theorems are called "definitional theorems"; such a theorem holds because a new constant (typically having function type) has just been defined to have that property. We shall see later that before constants can be defined we must prove their existence in some way. Thus we cannot, for example, introduce a constant m with the self-contradictory property $((m = T) \wedge (m = F))$.

As explained previously, HOL enforces proof security by giving theorems a distinct abstract type. Therefore theorems and un-proven sequents have incompatible types.

1.5.2 The HOL Logic

HOL's logical deduction system can be described as a higher-order and classical logical system. For instance the theorem

$$\vdash \forall t.t \vee \neg t$$

known as *the law of the excluded middle* holds and the following derived rule

$$\frac{\Gamma, \neg t \vdash F}{\Gamma \vdash t}$$

known as the *classical contradiction* rule are valid.

A description at any significant detail level of the logic is outside the scope of this thesis. The reader is advised to consult [GM93] for more information.

1.5.3 Order of Definitions

HOL requires — quite sensibly — that constants (including functions) be defined before they are used. While this makes consistence checks simpler and thus easier to trust, it can also make a presentation convoluted and hard to follow. We will occasionally take the liberty of presenting utility functions after the functions that use them.

1.6 Proof Script

In order to give people without experience with theorem provers some feeling about the size of the proof script used for the proofs in this dissertation we supply these figures:

- The total size of the files making up the proof script is about 20000 lines or 600 KB.
- The running time for the script is about 4 CPU-hours on an otherwise unused Sun4c Sparcstation with 64 MB memory. An additional hour must be added for checking the transformations in Appendix B and more than an hour must be added for proving the typing of the self-interpreter (Theorem 5.12).
- Running the proof script proves just below two million intermediate theorems.

Chapter 2

Goals

Sed quis custodiet ipsos Custodes?
— JUVENAL, approximately 100 A.D.

The goals of this work could be described simply as "investigating the thesis found in Section 1.1" but deserve to be spelled out in greater detail. The goals are

- to develop and explore an *internal* view of partial evaluation in a formal way.
- to demonstrate that program transformations, including the elusive folding transformation, are susceptible to proofs, even formal proofs.
- to demonstrate that the state-of-the-art in mechanical theorem proving and verification is at a level where substantial programming language semantics can be handled.
- to demonstrate that mixing the fields of programming language semantics and mechanical verification is not simply extra work on top of the semantical work but that the theorem proving can be done directly in the formalisation with the benefits of assured correctness at any point in the process.

In the following we discuss these points in greater detail.

2.1 Internal View of Partial Evaluation

Research that only a small group of researchers know about is research that is not used at its full potential. Therefore, from time to time we give lectures to outside

audiences. When giving introductory lectures on partial evaluation to audiences from other fields of computer science it is common to be given questions along the line

> *What does partial evaluation give us that we could not have gotten from a mix of traditional optimisations like constant folding and function unfolding (or loop unrolling)?*

The traditional answer to such a question will point out at least three important differences:

- Traditional optimisers do not try to unfold recursive calls. In fact, the optimiser part of traditional C compilers like the GNU C Compiler [S$^+$85] primarily unfolds non-recursive calls to functions that the user has explicitly marked for unfolding.

- The program transformations and the analyses behind them are of a global nature whereas conventional optimisations almost always are of a local nature.

- If partial evaluation must be broken up into basic optimisations, then the list should at least include folding and introduction of new functions.

Note that the idea of decomposing the concept of partial evaluation into more basic program transformations is present already in the question posed by people from other fields. Nevertheless this subject has not been investigated thoroughly.

One goal of this thesis is to determine which basic transformations are needed in order to make up partial evaluation as we usually understand it.

As Chapter 3 will show, we are going to use a toy language and not a full-scale language like Scheme or Standard ML for our work. For example, our language is first-order and does not contain user-defined types. This is pragmatics at work: we would have liked to prove correctness of partial evaluation on a language like Standard ML, see [BW93], but we do not believe this is feasible at present because the burden of proof increases sharply with the language's size.

It is a separate goal of this thesis to work towards the construction of a provably correct and optimal partial evaluator [JGS93, Chapter 6] for a typed language and we conjecture that the methods presented here together with an added layer of retyping are strong enough to perform optimal partial evaluation[1].

[1] "Perform" is here used to mean that the methods presented will guarantee the correctness of a number of basic steps. There are no tools to select which transformations to apply, i.e., the controlling parts of a would-be partial evaluator are not presented.

2.2 Mechanically Verified Proofs

The use of a proof verifier for all proofs in this thesis should be seen in part as a reaction to the current state of the art of proofs in programming language theory as it appears in print. There are a number of subjects that are almost always silently ignored or presented incompletely. For instance (in no particular order):

"Fresh variables": often, in a program transformation, there is a need for a variable with a name which is in some sense not yet used. The usual way to solve this need is to have an oracle function producing these variable names, but it is not clear exactly which properties this oracle should have and how it should influence proofs.

In order to handle this problem in the way it is usually described (namely as global uniqueness) it would often be necessary to add extra parameters to the semantics-defining functions. This is clearly not desirable, so this thesis will use a local uniqueness condition which will turn out to be sufficient. (See also the recent article [NN96] by Nazareth and Nipkow which also pays attention to this problems.)

Scoping: when programs contain variables it is usually necessary, or at least desirable, to check that variables are used only within the scope of their definitions. This is usually done implicitly (but rigorously) as part of a typing discipline. The lack of precision is usually in the use of the scoping information.

It appears that there is no easy way out of this problem: Whenever a theorem depends on scoping conditions it seems that scoping conditions must be made part of the induction hypotheses in suitable ways.

Fixed points of inference rules: One way to define a language's semantics is to give a set of inference rules, see [Win93], and define an evaluation predicate as the least (or occasionally greatest) fixed point satisfying these rules. Such least fixed points do not always exist, nor are they necessarily unique, so that ought to be proven. Such proofs are rarely given. Often a reference to Knaster [Kna27] or Tarski [Tar55] is given but the preconditions for using Knaster/Tarski's fixed point theorem are usually not checked.

In this thesis all proofs have been machine verified. More precisely, the proofs in this thesis are transcriptions, cf. Appendix A, of verified proofs. Should there be any problems with proofs in this thesis it is thus most likely that they are of a typographical nature. If they are not, then the scope of the problems will not be limited to this thesis.

* * *

Getting back to the opening quote of this chapter: if — for whatever reason — doubt should be cast on the correctness of the HOL theorem prover computer system, still not all is lost. Wong in [Won95] describes a system designed to capture and independently verify all basic inferences that are part of a proof. Thus it would, at least in principle, be possible to get a second opinion on the validity of the proofs; but this check has not been done yet nor is it planned. [Won95] is based in part on von Wright's HOL-in-HOL package [vW94], which taken to its extreme might lead to formally checking HOL.

2.3 Formalisation as Part of the Process

The rôle of formalisation and formal proofs in connection with programming language semantics has so far been to formalise and verify theories and results that were known in advance. Thus work with the theorem prover will mainly consist of redoing proofs. (See Chapter 9 for examples.)

We submit that this is inheritly the wrong time for formal methods to enter the scene and that they should more properly enter early so that proofs can be done — not *re*-done — with the theorem prover and such that there is no redundant work.

Therefore, the proofs that have been done in connection with this work were done directly using the theorem prover and they are not formalisations of any previously formulated existing theorems, except that there are related theorems in related work.

Chapter 3

Language

You taught me language; and my profit on't
Is, I know how to curse; the red plague rid you,
For learning me your language!
— WILLIAM SHAKESPEARE, 1564–1616

This chapter analyses the consequences that various language features have on partial evaluation, in particular on self-application of a partial evaluator. We then turn to the language that will actually be used throughout the remaining chapters. We describe its abstract and its concrete syntax, define its semantics in both a standard and a "timed" way, and then introduce some correctness issues for programs written in our language.

3.1 Consequences of Language Features

3.1.1 Higher-Order versus First-Order

The decision of whether to include higher-order constructs or not has a number of consequences. Assume that we have a first-order language and want to add a lambda construct. Then

- it gets significantly more complicated to estimate control and data flow with any precision, because it for each higher-order application becomes necessary to known which lambdas could possibly be applied at that point. Knowing control and data flow is important for, e.g., precise binding-time analysis, see [JGS93, Chapters 10 and 15].

- the reduction transformation of $(\lambda x.e_1)e_2$ will have to be considered. With explicit names ("x" above) this is just as complicated with respect to renaming and variable capture as we shall later see that unfolding of `let`-bindings is. Without explicit names — for example using some de Bruijn [dB72] inspired notation — the transformation will instead be complicated by the renumbering necessary.

- value identity, to which expression equivalence is linked, becomes much more involved. In the first-order case, assuming ground terms, it is sufficient to use structural identity but this would not work in the higher-order setting because then, for example, the values $\lambda x.2$ and $\lambda x.1+1$ would be different which is decidedly not what we want. Instead some notion of bi-similarity, see [Pit95], would be needed in order to compare expressions and values.

The first two items are mostly a complication for anyone wanting to write a partial evaluator. Although this is not a direct goal of this thesis, it should still be considered since any practical use of this work would have to go in that direction. The last item, however, would cause a severe and unwelcome complication to all proof work, as seen in [San96]. We believe that the mixing of partial evaluation and formally verified proofs should be done in the simpler first-order setting before inviting these complications in.

3.1.2 Presence of Types

For the purposes of this discussion we will distinguish the following classes of language typedness.

Statically typed languages, like Standard ML [MTH90]. Valid programs from these languages do not commit type errors. It is not possible, nor is it useful, to inspect a value's type.

Dynamically typed languages with inspection, like Scheme [CR91]. Each operation checks the type of its arguments, but no programs are ruled out in advance. It is possible for the program to inspect the type of values.

Dynamically typed languages without inspection. Same as above, but without the primitives to inspect a value's type, in effect giving the language many properties of the statically typed language. For example, it is generally not possible to write a pretty-printer for values since we are unable to determine whether a value (say) is an integer, a string, or something else.

Untyped languages, like the untyped λ-calculus. "Type error" is not a recognised concept. (But reduction cannot continue for an expression like the application $(1\ 2)$.)

Untyped languages have so far turned out to be simpler to handle in the context of partial evaluation than typed languages. The reason for this is the need for explicit universal encoding when programming interpreter-like programs in a typed language. So far the majority of work in partial evaluation has concentrated on languages from the bottom two groups above.

It would probably thus be simplest to chose a Scheme-like language or even a completely untyped language. Doing that, however, would conflict with the goal of working toward optimal typed partial evaluation. On the other hand choosing a statically typed language would require us to formalise a typing system and formally prove the relevant safety theorems. This, while an important task, is a major undertaking in itself and not directly relevant to this thesis. Moreover, it turns out that the correctness of our program transformations do not rely on well-typedness but mostly on correct use of scoping. Therefore, we will chose a dynamically typed language without inspection and write programs (like the interpreter needed in the following chapter) as if the language were statically typed, i.e., with full universal encoding of values. This way our result will apply also to the smaller set of statically well-typed programs.

We are now left with choosing the types with which the language operates. The main guiding line here is simplicity both in terms of the types and in terms of the size of a potential self-interpreter. We chose integers as the sole base type and add pairs and a two-constructor anonymous data type on top, i.e., values are described by the domain *Val* satisfying

$$Val = Num + (Val \times Val) + (Val + Val).$$

Another option considered was having a unit base type with only one element; integers could then be simulated using unary encoding, but this would be at the expense of inflating programs.

3.1.3 Parameters of Functions

Some programming languages dictate that all functions have exactly one formal parameter while others allow functions to have any number of parameters for functions. (But usually each function takes the same number of parameters every time it is called.) Standard ML, for example, dictates one and only one parameter which means that the expression

```
f (1,2)
```

is evaluated by constructing a pair and calling `f` with that pair as its only argument, whereas the Scheme expression

```
(f 1 2)
```

is a call to `f` with two arguments. Note that the Standard ML expression `(f 1 2)` is valid but consists of two function calls with one parameter, not one call with two.

This might appear as a technical distinction with little practical importance. But, for the purpose of optimal partial evaluation there is a significant difference between the two. Partial evaluators work by specialisation, meaning that an argument can be made to disappear because all applications use the same constant for the argument in question. Thus if specialisation was the only thing that the partial evaluator did and we used a language with multiple parameters, then the residual program could not contain a call with more parameters than the maximum observed in the original program.

If we use this observation on the application of the partial evaluator to a self-interpreter and its program input

$$[\![\texttt{mix}]\!](\texttt{sint}, \texttt{p}) = \texttt{p'}$$

we see that `p'` will inherit `sint`'s limit on the number parameters. Therefore, if `p` contains an occurrence of a call with a larger number of parameters than seen in `sint` then `p'` will use some data structure — typically a list structure inherited from the interpreter — to pass the parameters. This means that we cannot expect the partial evaluator to be optimal. See also [Mog96].

There are solutions to this problem, for example by using a technique called *arity raising* [Rom88, Rom90] or the related subject of handling *partially static data structures*, see [Mog88] and [Rom88]. However, we believe that introducing this kind of complication at the present time is not wise because it will inflate the problem without given more insight, and we will therefore use the Standard ML model of calling functions.

3.2 Partial Evaluation Language

For the rest of this thesis we shall concentrate on the following rather simple language chosen, on the one hand as not to require sophisticated methods in the partial evaluation process, and on the other hand in a way that does not cause a self-interpreter to become overly inflated with encoding.

3.2.1 Abstract Syntax

The of the language PEL[1] is found in Figure 1. The fragment $\langle p \rangle$ means an optional occurrence of p.

[1] Partial Evaluation Language, not to be confused with the language of the same name described in [Lau89].

p	::=	$f x$=e; $\langle p \rangle$	Program
e	::=	i	Integer constant
	|	`(`e_1 o e_2`)`	Integer operation
	|	`(`e_1`,`e_2`)`	Pair construction
	|	`fst` e	Pair destruction
	|	`snd` e	Pair destruction
	|	`inl` e	Sum construction
	|	`inr` e	Sum construction
	|	`case` e `of inl.`x_ℓ`->`e_ℓ `| inr.`x_r`->`e_r `end`	Sum destruction
	|	x	Variable reference
	|	`error`	Error indication
	|	f e	Function application
	|	`let` x `=` e_1 `in` e_2 `end`	Let-binding
o	::=	`+` | `-` | `*` | `=`	Integer operators
f	$\in$	*Func*	Function names
x	$\in$	*Var*	Variable names
i	$\in$	$\{0,1,2,\ldots\}$	The integers

Figure 1: PEL syntax

Note that the syntax is chosen such that the grammar is not ambiguous: infix operations are parenthesised and both `case`- and `let`-expressions are explicitly terminated. (Although this can be painful for practical programming it is useful because it eliminates a level of imprecision.)

The concrete syntax will be relaxed slightly in examples by using `L` for `inl` and `R` for `inr` as well as sometimes swapping the cases in a `case`-expression and omitting parentheses from integer operations where the intention is clear.

The particular set of integer operators is not important. In fact, subtraction, which is computationally redundant, was added quite late in the formalisation, resulting in surprisingly few proofs that needed minor modifications.

3.3 Semantics

The semantics of the partial evaluation language is straightforward. A program defines a collection of mutually recursive functions, the first of which defines the meaning of the program. The language is strict.

The semantics to be presented will assign a meaning to any syntactically correct program. In other words, there are no requirements that, e.g., function calls

only go to defined functions, that functions be defined only once, that variables be used only in scope, and that the program be well-typed in some sense. However, we shall later rule out some of this "nastiness" as it will influence the correctness of program transformations. Consider the expressions

`fst ((2+2),x)` and `4`.

We would like these two expressions to be interchangeable anywhere in any program, but given (in this case) that the semantics is strict this is not so. If `x` is not in scope at the point where the change takes place then the left-hand expression will never evaluate to anything while the right-hand expression will always evaluate to the value `4`.

The reason for spelling this out in detail is two-fold: (1) it has been somewhat neglected in previous work, and (2) a theorem prover like HOL will not accept hand-waving arguments as to the correctness of program transformations. It should in fact be noted that a lot of effort has been put into proving seemingly obvious theorems, especially about scoping.

3.3.1 Semantic Objects

The semantics of the language utilises two kinds of semantic objects in addition to the syntactic objects defined above. These are values, denoted by v, and environments denoted by E.

The value domain is defined by the equation

$$Val = Num + (Val \times Val) + (Val + Val),$$

i.e., values are either numbers, pairs of values, or an element tagged with one of two tags, `inl` and `inr`. Just as for expression we will usually abbreviate these as `L` and `R`; the double use of these names does not appear in the formalisation and should cause no confusion.

The set of numbers used is chosen to be the set of natural numbers starting with zero. This allows for slightly simpler formalisation into HOL as it happens to coïncide with HOL's own idea of natural numbers.

The semantics of programs is built on top of the semantics of expressions which in turn operates with environments. Instead of adopting the traditional description where environments are partial finitary functions from variables into values we shall use an abstract version. The benefit of this is that there will be no need for arguing (i.e., proving) any correspondence between partial functions used in the formal semantics and lists more commonly used in implementations. The abstract description requires an abstract type, *Env*, and three operations on it. (1) building a one-point environment from a variable and a value, denoted

$\{x \mapsto v\}$. (2) updating an environment, denoted $E + \{x \mapsto v\}$. (3) looking up a variable in an environment, denoted `lookup_env` $E\,x$ (a partial function). These operations should satisfy

$$(\texttt{lookup\_env}\,\{x \mapsto v\}\,x' = v') \iff (x = x') \wedge (v = v') \tag{3.1}$$

and

$$\begin{aligned}(\texttt{lookup\_env}\,(E + \{x \mapsto v\})\;x' = v') \iff \\ ((x = x') \to (v = v') \mid (\texttt{lookup\_env}\,E\,x' = v')).\end{aligned} \tag{3.2}$$

Note the "+" in $E + \{x \mapsto v\}$ should be regarded as purely syntactic and has very little to do with regular notions of addition. For example it is not commutative.

3.3.2 Semantic Rules

The semantics of a program is given in Figure 2 which defines a predicate (or a "ternary relation") $P \vdash v_{\text{in}} \mapsto v_{\text{out}}$ which is true if program P given input v_{in} produces output v_{out}. It is based on a corresponding predicate, $E \vdash_P e \Rightarrow v$, for expressions. The predicate will be defined shortly.

$$\frac{\{x \mapsto v_{\text{in}}\} \vdash_P e \Rightarrow v_{\text{out}} \qquad (f\;\;x\texttt{=}e\texttt{;}) = \text{first } P}{P \vdash v_{\text{in}} \mapsto v_{\text{out}}} \tag{3.3}$$

Figure 2: PEL program semantics.

The semantic function "first" used in Figure 2 is defined to extract the first PEL-function definition in the program.

$$\texttt{eval\_oper}\;\;o\;i_1\;i_2 = \begin{cases} i_1 + i_2, & \text{if } o = \texttt{+} \\ i_1 \dot{-} i_2, & \text{if } o = \texttt{-} \\ \texttt{L 0}, & \text{if } (o = \texttt{=}) \wedge (i_1 \neq i_2) \\ \texttt{R 0}, & \text{if } (o = \texttt{=}) \wedge (i_1 = i_2) \end{cases} \tag{3.4}$$

Figure 3: PEL operator semantics.

The evaluation of operators is handled by the function `eval_oper` found in Figure 3. Since the number field was chosen not to include the negative numbers the semantics of subtraction is based on the truncating subtraction operator denoted $\dot{-}$ and sometimes called "monus." It can be defined in terms of the usual subtraction operator for example by

$$i_1 \dot{-} i_2 = \begin{cases} i_1 - i_2, & \text{if } i_1 \geq i_2 \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

Note that Figure 3 defines truth as `R 0` and falsity as `L 0`. This encoding is one of many that allows `if`-expressions to be encoded as `case`-expressions.

The interesting part of the semantics of the language is the semantics given to expressions. Figure 4 shows inductive rules defining the predicate (four-way relation) $E \vdash_P e \Rightarrow v$ which is true if expression e evaluates to v in the context of program P (for function calls) and environment E (for free variables).

The "inductive" above means that the relation defined is the smallest relation which satisfies the rules of Figure 4. This method of defining relations is covered in, e.g., [Win93]. The formalisation will prove that this is well-defined and that the predicate has properties allowing us to think of it as a partial function from expressions, programs, and environments to values.

The definition of `eval_expr` uses the partial semantic function "`lookup_func`" which given a program and a function name returns the definition of the first function in the program having that name.

3.3.3 Errors and Non-Termination

It is a consequence of the way that the semantics was defined that there will be no way to distinguish between an error situation — either explicit evaluation of `error` or implicitly such as attempting to take the first component of an integer — and a non-terminating evaluation. For given P and v_{in}, these errors and non-termination will both be indicated by non-existence of a v_{out} satisfying the evaluation predicate $P \vdash v_{\text{in}} \mapsto v_{\text{out}}$.

It should be noted that the collapsing of errors and non-termination is not something inherently present with the methods used. In fact it would be quite easy, if rather work-intensive in the formalisation, to sort things out, for instance by tagging values as either exceptional or regular as for the definition of Standard ML documented in [MTH90]. As readers familiar with this book, more precisely its Section 6.7, will know the authors do not actually state most of the rules dealing with exceptional values. Instead they present an "exception convention" that for most of the approximately 60 rules shown implicitly adds one or more exception rules. This should not be ascribed to laziness on behalf of the authors: a spelled-out version would be huge and readability would suffer significantly. A conservative estimate of the number of rules for Standard ML if the various notational conventions were not used is 200 rules for the dynamic semantics of the core alone. See also Section 9.4.

Our language is a lot smaller than Standard ML and Standard ML takes care to distinguish different kinds of errors which we might not need to; but even so, adding exceptions would significantly increase the number of inference rules. Moreover, formalisation with HOL does not allow informality even at the level of

$$\frac{}{E \vdash_P i \Rightarrow i} \tag{3.6}$$

$$\frac{E \vdash_P e_1 \Rightarrow i_1 \qquad E \vdash_P e_2 \Rightarrow i_2 \qquad \texttt{eval\_oper}\ o\ i_1\ i_2 = v}{E \vdash_P (e_1\ o\ e_2) \Rightarrow v} \tag{3.7}$$

$$\frac{E \vdash_P e_1 \Rightarrow v_1 \qquad E \vdash_P e_2 \Rightarrow v_2}{E \vdash_P (e_1, e_2) \Rightarrow (v_1, v_2)} \tag{3.8}$$

$$\frac{E \vdash_P e \Rightarrow (v_1, v_2)}{E \vdash_P \texttt{fst}\ e \Rightarrow v_1} \tag{3.9}$$

$$\frac{E \vdash_P e \Rightarrow (v_1, v_2)}{E \vdash_P \texttt{snd}\ e \Rightarrow v_2} \tag{3.10}$$

$$\frac{E \vdash_P e \Rightarrow v}{E \vdash_P \texttt{inl}\ e \Rightarrow \texttt{L}\ v} \tag{3.11}$$

$$\frac{E \vdash_P e \Rightarrow v}{E \vdash_P \texttt{inr}\ e \Rightarrow \texttt{R}\ v} \tag{3.12}$$

$$\frac{E \vdash_P e \Rightarrow \texttt{L}\ v_\ell \qquad E + \{x_\ell \mapsto v_\ell\} \vdash_P e_\ell \Rightarrow v}{E \vdash_P \texttt{case}\ e\ \texttt{of inl.}x_\ell\texttt{->}e_\ell\ \texttt{|}\ \texttt{inr.}x_r\texttt{->}e_r\ \texttt{end} \Rightarrow v} \tag{3.13}$$

$$\frac{E \vdash_P e \Rightarrow \texttt{R}\ v_r \qquad E + \{x_r \mapsto v_r\} \vdash_P e_r \Rightarrow v}{E \vdash_P \texttt{case}\ e\ \texttt{of inl.}x_\ell\texttt{->}e_\ell\ \texttt{|}\ \texttt{inr.}x_r\texttt{->}e_r\ \texttt{end} \Rightarrow v} \tag{3.14}$$

$$\frac{\texttt{lookup\_env}\ E\ x = v}{E \vdash_P x \Rightarrow v} \tag{3.15}$$

(there is no rule for `error`) (3.16)

$$\frac{E \vdash_P e \Rightarrow v \qquad \texttt{lookup\_func}\ P\ f = (f\ x\texttt{=}e';) \qquad \{x \mapsto v\} \vdash_P e' \Rightarrow v'}{E \vdash_P f\ e \Rightarrow v'} \tag{3.17}$$

$$\frac{E \vdash_P e_1 \Rightarrow v_1 \qquad E + \{x \mapsto v_1\} \vdash_P e_2 \Rightarrow v_2}{E \vdash_P \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2\ \texttt{end} \Rightarrow v_2} \tag{3.18}$$

Figure 4: PEL expression semantics.

the well-described exception convention. Finally, we do not appear to need the distinction in order to make transformations work.

3.4 Timed Semantics

In the previous section we defined the semantics of our language as predicates describing the input-output behaviour of programs and expressions. In this section we shall define a refinement of the core of the semantics, i.e., the expression evaluation predicate $E \vdash_P e \Rightarrow v$.

The evaluation predicate $E \vdash_P e \Rightarrow v$ and the syntactic types give us several induction proof methods suitable for proving an evaluation assertion:

Structural case analysis: Any expression has one of the twelve forms shown in Figure 1. We can use this observation to split the assertion into twelve cases.

Induction on expression: structural induction based on the expression type. As the language has explicit recursion, this method is usually not strong enough when both expressions and evaluation are involved. This is because the function body in the recursive call is not automatically "smaller" than the call expression itself.

Rule induction: (also known as induction on the structure of derivations) using the least fixed-point property used to define the evaluation predicate.

For assertions which have the right form for attempting either proof method, rule induction is stronger than structural induction which in turn is stronger than structural case analysis. Therefore it is no doubt the case that several of the proofs conducted in the formalisation part of the present work use rule induction where structural induction would have sufficed or even use structural induction instead of structural case analysis. The "penalty" for doing this when using HOL is virtually non-existent.

Strong as they may be, the techniques above do not always suffice. For this reason we introduce the notion of a *timed semantics*, which formally is a sequence of predicates $E \vdash_{P,n} e \Rightarrow v$ where n ranges over non-negative numbers. A timed semantics is required to satisfy the conditions

$$(E \vdash_P e \Rightarrow v) \qquad \text{if and only if} \qquad \exists n : \left(E \vdash_{P,n} e \Rightarrow v\right) \tag{3.19}$$

and

$$\left(E \vdash_{P,n} e \Rightarrow v\right) \qquad \text{implies} \qquad \left(E \vdash_{P,n+1} e \Rightarrow v\right). \tag{3.20}$$

which state that $E \vdash_{P,n} e \Rightarrow v$ is an n-indexed sequence of increasingly better approximations to $E \vdash_P e \Rightarrow v$ and with it as a limit. Furthermore the approximation is monotonic in n.

Not all timed semantics are interesting. For instance, any sequence which coïncides with $E \vdash_P e \Rightarrow v$ at some point (and therefore as a consequence of the conditions *from* that point) does not bring us anything new.

The way to think of the n parameter to the timed semantics predicate is as a restriction on the size (loosely speaking) of the evaluation. In this light Equation 3.19 becomes "evaluations which can be done at all are exactly those which can be done with some finite restriction on the size of the evaluation" and Equation 3.20 becomes "with a larger limit on evaluation size we can evaluate at least the same."

We shall now turn to two interesting timed semantics of different nature. They will both concentrate on function calls, which as explained previously is where proofs are difficult. The first one, based on nested call depth, operates with "reusable resources" in which, e.g., the available resources for the right-hand component of a pairing operation depend only on the available resources for the pairing operation as a whole and not on how resource-demanding the left-hand component is. One can think of this kind of restriction as a chain holding a dog; the dog can run for as long as it wants but never more than a certain distance from the chain's fixed point.

The second is based on total number of function calls; thus the resource limit for the right-hand component of a pairing operation depends on what the left-hand component did not use.[2] This kind of restriction is like fuel added to a car in the desert: you use it and it is gone forever.[3]

3.4.1 Nested Call Depth

Figure 5 shows the most important timed semantics, the *nested call timed expression semantics*. In this semantics, which is so important that it will simply use the general notation introduced above, the n parameter is a limit on the number of nested function calls allowed during evaluation.

Strictly speaking this description would require us to use $n+1$ instead of n in Equation 3.32's left-most hypothesis, i.e., use $E \vdash_{P,n+1} e \Rightarrow v$ and not $E \vdash_{P,n} e \Rightarrow v$. We do use n in this place because

- It makes a kind of call unfolding, namely replacing a function call `(`f e`)` by a call to an identity function `I (let` x_f`=`e `in` e_f `end)`, neutral with respect to this variant of call depth.

[2] This description is not meant to imply any particular evaluation order as far as the semantics is concerned.

[3] A description using "time" as we normally understand it would also be possible, but would not provide the option of arguing the eventual choice of call depth as the right limit based on being environmentally responsible.

$$\frac{}{E \vdash_{P,n} i \Rightarrow i} \tag{3.21}$$

$$\frac{E \vdash_{P,n} e_1 \Rightarrow i_1 \qquad E \vdash_{P,n} e_2 \Rightarrow i_2 \qquad \texttt{eval\_oper}\ o\ i_1\ i_2 = v}{E \vdash_{P,n} (e_1\ o\ e_2) \Rightarrow v} \tag{3.22}$$

$$\frac{E \vdash_{P,n} e_1 \Rightarrow v_1 \qquad E \vdash_{P,n} e_2 \Rightarrow v_2}{E \vdash_{P,n} (e_1, e_2) \Rightarrow (v_1, v_2)} \tag{3.23}$$

$$\frac{E \vdash_{P,n} e \Rightarrow (v_1, v_2)}{E \vdash_{P,n} \texttt{fst}\ e \Rightarrow v_1} \tag{3.24}$$

$$\frac{E \vdash_{P,n} e \Rightarrow (v_1, v_2)}{E \vdash_{P,n} \texttt{snd}\ e \Rightarrow v_2} \tag{3.25}$$

$$\frac{E \vdash_{P,n} e \Rightarrow v}{E \vdash_{P,n} \texttt{inl}\ e \Rightarrow \texttt{L}\ v} \tag{3.26}$$

$$\frac{E \vdash_{P,n} e \Rightarrow v}{E \vdash_{P,n} \texttt{inr}\ e \Rightarrow \texttt{R}\ v} \tag{3.27}$$

$$\frac{E \vdash_{P,n} e \Rightarrow \texttt{L}\ v_\ell \qquad E + \{x_\ell \mapsto v_\ell\} \vdash_{P,n} e_\ell \Rightarrow v}{E \vdash_{P,n} \texttt{case}\ e\ \texttt{of}\ \texttt{inl}.x_\ell\texttt{->}e_\ell\ \texttt{|}\ \texttt{inr}.x_r\texttt{->}e_r\ \texttt{end} \Rightarrow v} \tag{3.28}$$

$$\frac{E \vdash_{P,n} e \Rightarrow \texttt{R}\ v_r \qquad E + \{x_r \mapsto v_r\} \vdash_{P,n} e_r \Rightarrow v}{E \vdash_{P,n} \texttt{case}\ e\ \texttt{of}\ \texttt{inl}.x_\ell\texttt{->}e_\ell\ \texttt{|}\ \texttt{inr}.x_r\texttt{->}e_r\ \texttt{end} \Rightarrow v} \tag{3.29}$$

$$\frac{\texttt{lookup\_env}\ E\ x = v}{E \vdash_{P,n} x \Rightarrow v} \tag{3.30}$$

(there is no rule for `error`) (3.31)

$$\frac{E \vdash_{P,n} e \Rightarrow v \quad \texttt{lookup\_func}\ P\ f = (f\ x\texttt{=}e';) \quad \{x \mapsto v\} \vdash_{P,n} e' \Rightarrow v'}{E \vdash_{P,n+1} f\ e \Rightarrow v'} \tag{3.32}$$

$$\frac{E \vdash_{P,n} e_1 \Rightarrow v_1 \qquad E + \{x \mapsto v_1\} \vdash_{P,n} e_2 \Rightarrow v_2}{E \vdash_{P,n} \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2\ \texttt{end} \Rightarrow v_2} \tag{3.33}$$

Figure 5: Nested Call Timed Expression Semantics.

- It does not appear to complicate proofs which would not need the decrease in n, for example the proof of one of the lemmas used to prove an interpreter correct later, Lemma 5.5.
- Having multiple timed semantics in the formalisation is not attractive because many uninteresting support theorems would need to be proven.

3.4.2 Total Number of Function Calls

The second important timed semantics is shown in Figure 6. In this semantics the n parameter is a limit on the total number of calls allowed during evaluation.

$$\frac{}{E \Vdash_{P,n} i \Rightarrow i} \tag{3.34}$$

$$\frac{E \Vdash_{P,n_1} e_1 \Rightarrow i_1 \qquad E \Vdash_{P,n_2} e_2 \Rightarrow i_2 \qquad \texttt{eval\_oper}\ o\ i_1\ i_2 = v}{E \Vdash_{P,n_1+n_2} (e_1\ o\ e_2) \Rightarrow v} \tag{3.35}$$

$$\frac{E \Vdash_{P,n_1} e_1 \Rightarrow v_1 \qquad E \Vdash_{P,n_2} e_2 \Rightarrow v_2}{E \Vdash_{P,n_1+n_2} (e_1, e_2) \Rightarrow (v_1, v_2)} \tag{3.36}$$

$$\frac{E \Vdash_{P,n} e \Rightarrow (v_1, v_2)}{E \Vdash_{P,n} \texttt{fst}\ e \Rightarrow v_1} \tag{3.37}$$

$$\frac{E \Vdash_{P,n} e \Rightarrow (v_1, v_2)}{E \Vdash_{P,n} \texttt{snd}\ e \Rightarrow v_2} \tag{3.38}$$

$$\frac{E \Vdash_{P,n} e \Rightarrow v}{E \Vdash_{P,n} \texttt{inl}\ e \Rightarrow \texttt{L}\ v} \tag{3.39}$$

$$\frac{E \Vdash_{P,n} e \Rightarrow v}{E \Vdash_{P,n} \texttt{inr}\ e \Rightarrow \texttt{R}\ v} \tag{3.40}$$

$$\frac{E \Vdash_{P,n} e \Rightarrow \texttt{L}\ v_\ell \qquad E + \{x_\ell \mapsto v_\ell\} \Vdash_{P,n_\ell} e_\ell \Rightarrow v}{E \Vdash_{P,n+n_\ell} \texttt{case}\ e\ \texttt{of inl.}x_\ell\texttt{->}e_\ell\ \texttt{|}\ \texttt{inr.}x_r\texttt{->}e_r\ \texttt{end} \Rightarrow v} \tag{3.41}$$

$$\frac{E \Vdash_{P,n} e \Rightarrow \texttt{R}\ v_r \qquad E + \{x_r \mapsto v_r\} \Vdash_{P,n_r} e_r \Rightarrow v}{E \Vdash_{P,n+n_r} \texttt{case}\ e\ \texttt{of inl.}x_\ell\texttt{->}e_\ell\ \texttt{|}\ \texttt{inr.}x_r\texttt{->}e_r\ \texttt{end} \Rightarrow v} \tag{3.42}$$

$$\frac{\texttt{lookup\_env}\ E\ x = v}{E \Vdash_{P,n} x \Rightarrow v} \tag{3.43}$$

(there is no rule for `error`) (3.44)

$$\frac{E \Vdash_{P,n} e \Rightarrow v \quad \texttt{lookup\_func}\ P\ f = (f\ \ x\texttt{=}e'\texttt{;}) \quad \{x \mapsto v\} \Vdash_{P,n'} e' \Rightarrow v'}{E \Vdash_{P,n+n'+1} f\ e \Rightarrow v'} \tag{3.45}$$

$$\frac{E \Vdash_{P,n_1} e_1 \Rightarrow v_1 \qquad E + \{x \mapsto v_1\} \Vdash_{P,n_2} e_2 \Rightarrow v_2}{E \Vdash_{P,n_1+n_2} \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2\ \texttt{end} \Rightarrow v_2} \tag{3.46}$$

Figure 6: #Calls Timed Expression Semantics.

As it can be seen from, e.g., Equation 3.36 the resource for the conclusion is split into two parts for the two subterms that must be evaluated. This is in contrast

to the call depth timed semantics where the entire resource was available to the subterms.

Disregarding otherwise important things — such as the language used — this timed semantics corresponds to $e \Downarrow_{\leq n}$ as used by Sands in [San96]. We will return to this in Sections 8.1.1 and 9.3.

3.5 Static Correctness

Section 3.2.1 defined the abstract syntax of the language we are considering by giving a context-free grammar. One purpose of the grammar is to separate the universe into things we are interested in, "programs," and things we are not interested in, "non-programs." It does not completely succeed in this regard, however, as the set of programs ends up containing more than we are really interested in. Consider for example the program

```
f x = fst (g y + h x);
g z = 2;
```

This is a program by the definition in Section 3.2.1. Several things are wrong with this program:

- Function `f`'s body refers to variable `y`, which is not "in scope."
- Function `f` contains a call to function `h`, which is not defined anywhere.
- Since the result of an addition is always an integer, the argument to `fst` will be an integer. `fst` does not make sense on integers.

The first two of these are scope problems. Even if we had been smarter when writing the grammar there would have been no way to syntactically prohibit this kind of behaviour while still using a context-free grammar, see [ASU86, Section 4.2].

For similar reasons we cannot expect to prevent type errors by means of a grammar. Even worse, we cannot prevent type errors in a decidable way at all without at the same time preventing many perfectly fine programs also. This is the halting problem in disguise, see [Jon97].

Since some of the program transformations we are going to describe later will depend on some scoping-condition we now define a correctness criterion on programs. As all our program transformations preserve this criterion it follows that we only need to prove correctness of the initial program in order to ensure correctness in every step of a series of program transformations.

Definition 2 (Static Correctness) *A program P is statically correct if it has the following three properties:*

- *All defined functions have different names.*
- *If a function f is called within the body of one of P's functions then P also defines f.*
- *Every variable within the body of one of P's functions is used only in scope, i.e., a variable either refers to the function's parameter or it occurs within the right subexpression of a* `let-` *or* `case`*-construct binding the variable in question.*

3.6 Formalisation into HOL

In this section we discuss the formalisation of the syntax and semantics of PEL into the HOL theorem prover.

3.6.1 Formalisation of Types and Syntax

This section discusses the formalisation of the syntax of expressions, values, operators, et cetera into HOL together with the definitions of the related types. Some of this will discuss aspects of HOL closer to the core than users regularly need to go. The basic principles will be discussed using the value type which is the smallest type that shows all the relevant aspects.

Definition of, for example, the value type involves two acts as far as HOL is concerned:

- Definition (in the logic) of such a type.
- Definition of "tags" or "constructors" to separate and identify the summands.

As far as the HOL kernel is concerned these are two different tasks.

Definition of a new type in HOL consists of four things: (1) choosing a new type name, (2) specifying a subset of an existing type by a characteristic predicate, (3) proving that the subset is non-empty, and (4) introducing an axiom stating that the new type is isomorphic to the given subset. This is described in more detail in [GM93, Sections 16 and 18].

Values

At first the definition of "tags" or "constructors" in HOL might seem impossible as the HOL logic is based on typed λ-calculus with constants, i.e., the objects are constants, variables, abstractions, and applications. In particular there are no

constructors. But as we shall see, constructors can be emulated by constants with the right properties. For the value type, for example, we want four constants which we will call constructors:

$$\texttt{V\_Int} : \texttt{num} \to \texttt{value} \qquad \texttt{V\_Pair} : \texttt{value} \to \texttt{value} \to \texttt{value}$$

$$\texttt{V\_Inl} : \texttt{value} \to \texttt{value} \qquad \texttt{V\_Inr} : \texttt{value} \to \texttt{value}$$

The use of currying over pairing in the type for `V_Pair` is not essential, but reflects the way HOL's standard tools will do it. We require the following properties:

Theorem 3.1 (Value Constructors are Injective.)

$$\vdash \forall i_1, i_2 : (\texttt{V\_Int}\ i_1 = \texttt{V\_Int}\ i_2) \Rightarrow (i_1 = i_2)$$
$$\vdash \forall v_{11}, v_{12}, v_{21}, v_{22} : (\texttt{V\_Pair}\ v_{11} v_{12} = \texttt{V\_Pair}\ v_{21} v_{22}) \Rightarrow (v_{11} = v_{21}) \wedge (v_{12} = v_{22})$$
$$\vdash \forall v_1, v_2 : (\texttt{V\_Inl}\ v_1 = \texttt{V\_Inl}\ v_2) \Rightarrow (v_1 = v_2)$$
$$\vdash \forall v_1, v_2 : (\texttt{V\_Inr}\ v_1 = \texttt{V\_Inr}\ v_2) \Rightarrow (v_1 = v_2)$$

(If the value type had had a constructor without any argument there would not have been a corresponding injectivity theorem for that constructor.)

Theorem 3.2 (Value Constructors Have Disjoint Images.)

$$\vdash \forall v_1, v_2, i : \texttt{V\_Int}\ i \neq \texttt{V\_Pair}\ v_1 v_2$$
$$\vdash \forall v, i : \texttt{V\_Int}\ i \neq \texttt{V\_Inl}\ v$$
$$\vdash \forall v, i : \texttt{V\_Int}\ i \neq \texttt{V\_Inr}\ v$$
$$\vdash \forall v_1, v_2, i : \texttt{V\_Pair}\ v_1 v_2 \neq \texttt{V\_Int}\ i$$
$$\vdash \forall v_1, v_2, v : \texttt{V\_Pair}\ v_1 v_2 \neq \texttt{V\_Inl}\ v$$
$$\vdash \forall v_1, v_2, v : \texttt{V\_Pair}\ v_1 v_2 \neq \texttt{V\_Inr}\ v$$
$$\vdash \forall v, i : \texttt{V\_Inl}\ v \neq \texttt{V\_Int}\ i$$
$$\vdash \forall v, v_1, v_2 : \texttt{V\_Inl}\ v \neq \texttt{V\_Pair}\ v_1 v_2$$
$$\vdash \forall v, v' : \texttt{V\_Inl}\ v \neq \texttt{V\_Inr}\ v'$$
$$\vdash \forall v, i : \texttt{V\_Inr}\ v \neq \texttt{V\_Int}\ i$$
$$\vdash \forall v, v_1, v_2 : \texttt{V\_Inr}\ v \neq \texttt{V\_Pair}\ v_1 v_2$$
$$\vdash \forall v, v' : \texttt{V\_Inr}\ v \neq \texttt{V\_Inl}\ v'$$

(Half of these 12 theorems follow from the other half by the symmetry of equality.)

The following theorem states that the construtors' images exhaust the value type, i.e., that the value type contains nothing but values.

Theorem 3.3 (Value Case Analysis Theorem.)

$$\vdash \forall v : \left(\begin{array}{l} (\exists i : v = \texttt{V\_Int}\ i) \vee \\ (\exists v_1, v_2 : v = \texttt{V\_Pair}\ v_1 v_2) \vee \\ (\exists v' : v = \texttt{V\_Inl}\ v') \vee \\ (\exists v' : v = \texttt{V\_Inr}\ v') \end{array} \right)$$

The following theorem allows us to prove properties of values by structural induction.

Theorem 3.4 (Value Structural Induction Theorem.)

$$\vdash \forall P : \left(\begin{array}{l} (\forall i : P(\texttt{V\_Int}\ i)) \wedge \\ (\forall v_1, v_2 : Pv_1 \wedge Pv_2 \Rightarrow P(\texttt{V\_Pair}\ v_1 v_2)) \wedge \\ (\forall v : Pv \Rightarrow P(\texttt{V\_Inl}\ v)) \wedge \\ (\forall v : Pv \Rightarrow P(\texttt{V\_Inr}\ v)) \end{array} \right) \Rightarrow (\forall v : Pv)$$

The following theorem gives us a way to define primitive recursive functions on the value type. '$\exists!$' is the unique-existence qualifier.

Theorem 3.5 (Value Initiality Theorem.)

$$\vdash \forall f_0, f_1, f_2, f_3 : \exists! f : \left(\begin{array}{l} (\forall i : f(\texttt{V\_Int}\ i) = f_0 i) \wedge \\ (\forall v_1, v_2 : f(\texttt{V\_Pair}\ v_1 v_2) = f_1(fv_1)(fv_2)v_1 v_2) \wedge \\ (\forall v : f(\texttt{V\_Inl}\ v) = f_2(fv)v) \wedge \\ (\forall v : f(\texttt{V\_Inr}\ v) = f_3(fv)v) \end{array} \right)$$

Incidentally, Theorem 3.5 implies Theorems 3.1 through 3.4. The question of how to show the implication is left as an interesting exercise for the reader.

From the above list of theorems it would appear as if the introduction of a simple structural type into HOL was major undertaking. In practice this is not the case because the process can be (and has been) automated, i.e., HOL provides a convenient function[4] which given a suitable grammar performs the type definition and proves the relevant theorems about the constructors. In fact, the actual code introducing the type, proving existence of and defining the constructors, and proving 3.5 is just

```
val value =
  define_type{name="value",
              type_spec = `value = V_Int of num
                                 | V_Pair of value => value
                                 | V_Inl of value
                                 | V_Inr of value`,
              fixities = [Prefix, Prefix, Prefix, Prefix]};
```

(where the "fixities" argument controls HOL's pretty-printer and parser for future uses of the constructors). Further functions are available for proving the remaining theorems mentioned above.

It is important to realise that `define_type` as used in the code above is *not primitive to the HOL logic.* It should be considered no more than a shorthand

[4]Recall that HOL is implemented in Standard ML and that the language is available to the HOL user as a Meta-Language.

dealing with messy details. In particular the proof security of the HOL system is not affected by programming or logic errors in `define_type` should there be any. Moreover the user could introduce a replacement if the need should arise.

Functions and Variables

Function and variable names need to come from infinite sets, *Func* and *Var*, but otherwise there are no restriction on them. For simplicity we simply chose to use the natural numbers for them. The obvious thing to do would be to use a type abbreviation, for example "type var = int;" in Standard ML, but unfortunately HOL does not allow type abbreviations.[5] Instead the ML-concept of anti-quotation can be used. When "`^var`" or "`^func`" appears below (and as types are generally not shown when they are not essential these fragments will not appear often) they both stand for "`num`" but with the added clue that they represent variable (respectively function) names.

Operators

The operator type is simpler than the value type in that it is not a recursive type. Nevertheless it is defined using the same HOL-provided tools:

```
val oper =
  define_type{name="oper",
              type_spec = `oper = Add | Mul | Sub | Equal`,
              fixities = [Prefix, Prefix, Prefix, Prefix]};
```

A series of theorems similar to the series that were deduced for the value type can be deduced for the operators type, except that injectivity does not apply because the constructors have no arguments. For example, we have the case analysis theorem and the induction theorem:

Theorem 3.6 (Operator Case Analysis Theorem.)

$$\vdash \forall o : ((o = \texttt{Add}) \vee (o = \texttt{Mul}) \vee (o = \texttt{Sub}) \vee (o = \texttt{Equal}))$$

Theorem 3.7 (Operator Structural Induction Theorem.)

$$\vdash \forall P : P(\texttt{Add}) \wedge P(\texttt{Mul}) \wedge P(\texttt{Sub}) \wedge P(\texttt{Equal}) \Rightarrow (\forall o : Po).$$

The latter is no more useful than the case analysis theorem because the type is not recursive. Theorem 3.6 and Theorem 3.7 imply each other.

[5] Recall that the term "HOL" is used for the HOL 90.7 version of the system. The HOL 88 versions do have such a mechanism for defining type abbreviations.

Expressions

The expression type is built on top of another non-primitive type, namely the operator type, but apart from that it is not more complicated than the value type. The expression type is defined by

```
val expr =
  define_type
    {name="expr",
     type_spec = ‘expr = E_Int of num
                       | E_Op of oper => expr => expr
                       | E_Pair of expr => expr
                       | E_Fst of expr
                       | E_Snd of expr
                       | E_Inl of expr
                       | E_Inr of expr
                       | E_Case of expr => ^var => expr
                                        => ^var => expr
                       | E_Var of ^var
                       | E_Error
                       | E_Call of ^func => expr
                       | E_Let of ^var => expr => expr‘,
     fixities = [Prefix, Prefix, Prefix, Prefix,
                 Prefix, Prefix, Prefix, Prefix,
                 Prefix, Prefix, Prefix, Prefix]};
```

There are no surprises with the related theorems. For instance, the structural induction theorem states:

Theorem 3.8 (Expression Structural Induction Theorem.)

$$\vdash \forall P : \left(\begin{array}{l} (\forall i : P(\texttt{E\_Int}\ i)) \wedge \\ (\forall e_1, e_2 : Pe_1 \wedge Pe_2 \Rightarrow \forall o : P(\texttt{E\_Op}\ o\, e_1\, e_2)) \wedge \\ (\forall e_1, e_2 : Pe_1 \wedge Pe_2 \Rightarrow P(\texttt{E\_Pair}\ e_1\, e_2)) \wedge \\ (\forall e : Pe \Rightarrow P(\texttt{E\_Fst}\ e)) \wedge \\ (\forall e : Pe \Rightarrow P(\texttt{E\_Snd}\ e)) \wedge \\ (\forall e : Pe \Rightarrow P(\texttt{E\_Inl}\ e)) \wedge \\ (\forall e : Pe \Rightarrow P(\texttt{E\_Inr}\ e)) \wedge \\ (\forall e, e_\ell, e_r : Pe \wedge Pe_\ell \wedge Pe_r \Rightarrow \\ \qquad \forall x_\ell, x_r : P(\texttt{E\_Case}\ e\, x_\ell\, e_\ell\, x_r\, e_r)) \wedge \\ (\forall v : P(\texttt{E\_Var}\ v)) \wedge \\ P(\texttt{E\_Error}) \wedge \\ (\forall e : Pe \Rightarrow \forall f : P(\texttt{E\_Call}\ f\, e)) \wedge \\ (\forall e_1, e_2 : Pe_1 \wedge Pe_2 \Rightarrow \forall x : P(\texttt{E\_Let}\ x\, e_1\, e_2)) \end{array} \right) \Rightarrow (\forall e : Pe)$$

Since there are 12 different constructors in the expression type there are no less than 132 ($= 12(12-1)$) theorems stating that the constructors have different images. This does not pose a problem for the logic but it might be a performance problem. This will be discussed further in Section 8.2.1.

Programs

Programs are modelled by lists of tuples (f,x,e) each representing a function definition. As with functions and variables we really need a type abbreviation but can make do with using anti-quotation instead.

A side effect of this modelling is that the empty list becomes a valid value of type `^program` corresponding to the empty "program." This quirk could have been avoided by defining a new list type without the empty list but the difference turns out to be unimportant. It is much easier just to exclude the empty program from the set of statically correct programs.

Being a list type the program type inherits a number of theorems from the list type. For instance, the induction theorem holds:

Theorem 3.9 (List Induction.)

$$\vdash \forall P : (P([\,]) \wedge (\forall T : PT \Rightarrow (\forall H : P(\texttt{CONS}\, H\, T)))) \Rightarrow (\forall L : PL)$$

Note however that the quantification over H and T is not restricted to statically correct programs. Therefore this theorem cannot immediately be used to argue about properties of statically correct programs unless those properties are shared by all programs. This observation reflects the fact that due to mutual recursion we cannot build a large statically correct program by adding functions to a small one.

The Option Type

In order to be able to deal with partial functions in the formalisation it is useful to introduce the following parameterised type:

```
val option =
  define_type{name="option",
              type_spec = `option = OK of 'a | FAIL`,
              fixities = [Prefix, Prefix]};
```

The use of this type shall be to make partiality explicit, i.e., if $f(x) = y$ is a partial function then the function we will formalise is the total function given by

$$\widetilde{f}(x) = \begin{cases} \texttt{FAIL}, & \text{if } f(x) = \bot \\ \texttt{OK}\ f(x), & \text{otherwise.} \end{cases}$$

This method of encoding partiality will be used for the formalisation of the semantic functions `lookup_env` and `lookup_func` and will provide us with the option of discussing situations where looking up variables or functions fail.

Environments

In the tradition of simple interpreters, environments are represented as lists of pairs where the first component of each pair is a variable name and the second component is the corresponding value.

If we now define the function `lookup_env`, `make_env`, and `update_env` in the following way then these operation will have the properties 3.1 and 3.2 with the lifting described in the previous section.

Definitional Theorem 3.10

$$\begin{aligned}&(\forall x : \texttt{lookup\_env}\,[\,]\,x = \texttt{FAIL})\\&(\forall w, E, x : \texttt{lookup\_env}\,(\texttt{CONS}\,w E)\,x =\\&\qquad((x = \texttt{FST}\,w) \rightarrow \texttt{OK}\,(\texttt{SND}\,w) \mid \texttt{lookup\_env}\,E\,x))\end{aligned}$$

Definitional Theorem 3.11

$$\forall x, v : \texttt{make\_env}\,x\,v = [(x, v)]$$

Definitional Theorem 3.12

$$\forall x, v, E : \texttt{update\_env}\,x\,v\,E = \texttt{CONS}\,(x, v)\,E$$

The function `lookup_func` is very similar to `lookup_env`.

Definitional Theorem 3.13

$$\begin{aligned}&(\forall f : \texttt{lookup\_func}\,[\,]\,f = \texttt{FAIL})\\&(\forall b, P, f : \texttt{lookup\_func}\,(\texttt{CONS}\,b P)\,f =\\&\qquad((f = \texttt{FST}\,b) \rightarrow \texttt{OK}\,(\texttt{SND}\,b) \mid \texttt{lookup\_func}\,P\,f))\end{aligned}$$

3.6.2 Formalisation of Semantics

The semantic predicates, for example $E \vdash_P e \Rightarrow v$, are functions or, more precisely, constants with a function type. HOL provides two ways of introducing such constants into the logic:

- specifying a closed term for the function.
- proving the existence of a function with the required properties.

Since HOL terms as defined herein are well-typed per definition the first method is only suited for definition of non-recursive functions. The second method involves a proof in order to guard the soundness of the logic, see [GM93, Section 16.5.2]. In practical terms it disallows the definition of, say, $fn = 1 + (fn)$, which would immediately lead to inconsistencies.

Suppose, for example, that we want to define a recursive function `value2expr` with type `value` $\rightarrow$ `expr` converting a value to its corresponding constant expression, i.e., we want a function satisfying

Definitional Theorem 3.14

$$\vdash \left(\begin{array}{l} (\forall i : \texttt{value2expr}\,(\texttt{V\_Int}\ i) = \texttt{E\_Int}\ i) \wedge \\ (\forall v_1, v_2 : \texttt{value2expr}\,(\texttt{V\_Pair}\ v_1\ v_2) = \\ \quad\quad \texttt{E\_Pair}\,(\texttt{value2expr}\ v_1)\,(\texttt{value2expr}\ v_2)) \wedge \\ (\forall v : \texttt{value2expr}\,(\texttt{V\_Inl}\ v) = \texttt{E\_Inl}\,(\texttt{value2expr}\ v)) \wedge \\ (\forall v : \texttt{value2expr}\,(\texttt{V\_Inr}\ v) = \texttt{E\_Inr}\,(\texttt{value2expr}\ v)) \end{array} \right)$$

This function is recursive so we will have to prove its existence. To prove this, we use Theorem 3.5 and specialise it to the functions

$$\begin{array}{lcllcl} f_0 & : & \lambda i.\,\texttt{E\_Int}\ i & f_1 & : & \lambda r_1, r_2, v_1, v_2.\,\texttt{E\_Pair}\ r_1\ r_2 \\ f_2 & : & \lambda r, v.\,\texttt{E\_Inl}\ r & f_3 & : & \lambda r, v.\,\texttt{E\_Inr}\ r \end{array}$$

After beta-reduction we obtain the theorem

$$\vdash \exists! f : \left(\begin{array}{l} (\forall i : f(\texttt{V\_Int}\ i) = \texttt{E\_Int}\ i) \wedge \\ (\forall v_1, v_2 : f(\texttt{V\_Pair}\ v_1\ v_2) = \texttt{E\_Pair}\,(f v_1)\,(f v_2)) \wedge \\ (\forall v : f(\texttt{V\_Inl}\ v) = \texttt{E\_Inl}\,(f v)) \wedge \\ (\forall v : f(\texttt{V\_Inr}\ v) = \texttt{E\_Inr}\,(f v)) \end{array} \right)$$

Except for the fact that this theorem states unique existence and not just existence (and as unique existence implies existence that complication is small) this theorem states that a function with the recursive behaviour claimed for `value2expr` in 3.14 really does exist. It can therefore be used to define `value2expr` yielding the (axiomatic) Theorem 3.14.

HOL provides convenient tools to define primitive recursive functions. The function `value2expr` which plays a minor rôle later is defined by

```
val value2expr_DEF =
  new_recursive_definition
    {def = --‘(value2expr (V_Int i) = E_Int i) /\
              (value2expr (V_Pair v1 v2) =
                 E_Pair (value2expr v1) (value2expr v2)) /\
              (value2expr (V_Inl v) = E_Inl (value2expr v)) /\
              (value2expr (V_Inr v) = E_Inr (value2expr v))‘--,
     fixity = Prefix,
     name = "value2expr_DEF",
     rec_axiom = value};
```

where a double dash (`--`) is HOL's term parser, which also type checks the term.

Expression Semantics

The semantics of the predicate $E \vdash_P e \Rightarrow v$ was defined as the least fixed point of the rules in Figure 4. Proving the existence of such a function is covered in [Win93]

and with more focus on HOL in [Mel91], [AP91], and [Har95]. These tools are given a suitable transcription of the definition, i.e., as shown in Figure 4, and perform the definition of the constant and prove the corresponding behavioural theorems. Note again that these tools are not primitive to the HOL logic and therefore do not influence proof security. The description of the use of these tools is outside the scope of this thesis; here we shall just present the theorems arising from using them in the context of the evaluation predicate.

First of all the rules of Figure 4 are recovered as theorems for the newly defined `eval_expr` predicate:

Theorem 3.15 (Evaluation Rules.)

$$
\vdash \left\{ \begin{array}{l}
\forall i, P, E : \texttt{eval\_expr}\,(\texttt{E\_Int}\ i)\,P\,E\,(\texttt{V\_Int}\ i) \\
\quad \forall e_1, e_2, P, E, o, v : \\
\qquad \exists i_1, i_2 : \left(\begin{array}{l} \texttt{eval\_expr}\ e_1\,P\,E\,(\texttt{V\_Int}\ i_1) \wedge \\ \texttt{eval\_expr}\ e_2\,P\,E\,(\texttt{V\_Int}\ i_2) \wedge \\ (\texttt{eval\_oper}\ o\ i_1\ i_2 = v) \end{array} \right) \Rightarrow \\
\qquad \texttt{eval\_expr}\,(\texttt{E\_Op}\ o\ e_1\ e_2)\,P\,E\,v \\
\qquad\qquad \vdots \\
\qquad\qquad \textit{(six more cases omitted)} \\
\qquad\qquad \vdots \\
\quad \forall e_b, x_\ell, e_\ell, x_r, e_r, P, E, v : \\
\qquad \exists v_r : \left(\begin{array}{l} \texttt{eval\_expr}\ e_b\,P\,E\,(\texttt{V\_Inr}\ v_r) \wedge \\ \texttt{eval\_expr}\ e_r\,P\,(\texttt{update\_env}\ x_r\ v_r\ E)\,v \end{array} \right) \Rightarrow \\
\qquad \texttt{eval\_expr}\,(\texttt{E\_Case}\ e_b\ x_\ell\ e_\ell\ x_r\ e_r)\,P\,E\,v \\
\quad \forall P, E, x, v : \left(\begin{array}{l} (\texttt{lookup\_env}\ E\ x = \texttt{OK}\ v) \Rightarrow \\ \texttt{eval\_expr}\,(\texttt{E\_Var}\ x)\,P\,E\,v \end{array} \right) \\
\quad \forall x, e_1, e_2, P, E, v : \\
\qquad \exists v' : \left(\begin{array}{l} \texttt{eval\_expr}\ e_1\,P\,E\,v' \wedge \\ \texttt{eval\_expr}\ e_2\,P\,(\texttt{update\_env}\ x\ v'\ E)\,v \end{array} \right) \Rightarrow \\
\qquad \texttt{eval\_expr}\,(\texttt{E\_Let}\ x\ e_1\ e_2)\,P\,E\,v \\
\quad \forall e, P, E, v, f : \\
\qquad \exists v', e', x' : \left(\begin{array}{l} \texttt{eval\_expr}\ e\,P\,E\,v' \wedge \\ \texttt{eval\_expr}\ e'\,P\,(\texttt{make\_env}\ x'\ v')\,v \wedge \\ (\texttt{OK}\,(x', e') = \texttt{lookup\_func}\ P\ f) \end{array} \right) \Rightarrow \\
\qquad \texttt{eval\_expr}\,(\texttt{E\_Call}\ f\ e)\,P\,E\,v
\end{array} \right\}
$$

(For space reasons some of the cases are not shown above. There are no surprises in the ones left out this way.) The evaluation rules above allow us to deduce that the evaluation predicate holds when certain preconditions are satisfied, i.e., they

represent a kind of forward inference. The following theorem, known wither as the *structural cases theorem* or as the *cases theorem*, is used for reasoning the opposite way, namely to deduce why the predicate is true for certain parameters:

Theorem 3.16 (Evaluation Cases.)

$$
\begin{array}{l}
\forall e, P, E, v : \texttt{eval\_expr}\ e\ P\ E\ v \Leftrightarrow \\
\vdash \left(\begin{array}{l}
(\exists i : (e = \texttt{E\_Int}\ i) \wedge (v = \texttt{V\_Int}\ i)) \vee \\
\exists e_1, e_2, o, i_1, i_2 : \left(\begin{array}{l} (e = \texttt{E\_Op}\ o\ e_1\ e_2) \wedge \\ \texttt{eval\_expr}\ e_1\ P\ E\ (\texttt{V\_Int}\ i_1) \wedge \\ \texttt{eval\_expr}\ e_2\ P\ E\ (\texttt{V\_Int}\ i_2) \wedge \\ (\texttt{eval\_oper}\ o\ i_1\ i_2 = v) \end{array} \right) \vee \\
\exists e_1, e_2, v_1, v_2 : \left(\begin{array}{l} (e = \texttt{E\_Pair}\ e_1\ e_2) \wedge (v = \texttt{V\_Pair}\ v_1\ v_2) \wedge \\ \texttt{eval\_expr}\ e_1\ P\ E\ v_1 \wedge \\ \texttt{eval\_expr}\ e_2\ P\ E\ v_2 \end{array} \right) \vee \\
\exists e', v_2 : (e = \texttt{E\_Fst}\ e') \wedge \texttt{eval\_expr}\ e'\ P\ E\ (\texttt{V\_Pair}\ v\ v_2) \vee \\
\exists e', v_1 : (e = \texttt{E\_Snd}\ e') \wedge \texttt{eval\_expr}\ e'\ P\ E\ (\texttt{V\_Pair}\ v_1\ v) \vee \\
\exists e', v' : (e = \texttt{E\_Inl}\ e') \wedge (v = \texttt{V\_Inl}\ v') \wedge \texttt{eval\_expr}\ e'\ P\ E\ v' \vee \\
\exists e', v' : (e = \texttt{E\_Inr}\ e') \wedge (v = \texttt{V\_Inr}\ v') \wedge \texttt{eval\_expr}\ e'\ P\ E\ v' \vee \\
\exists e_b, x_\ell, e_\ell, x_r, e_r, v_\ell : \left(\begin{array}{l} (e = \texttt{E\_Case}\ e_b\ x_\ell\ e_\ell\ x_r\ e_r) \wedge \\ \texttt{eval\_expr}\ e_b\ P\ E\ (\texttt{V\_Inl}\ v_\ell) \wedge \\ \texttt{eval\_expr}\ e_\ell\ P\ (\texttt{update\_env}\ x_\ell\ v_\ell\ E)\ v \end{array} \right) \vee \\
\exists e_b, x_\ell, e_\ell, x_r, e_r, v_r : \left(\begin{array}{l} (e = \texttt{E\_Case}\ e_b\ x_\ell\ e_\ell\ x_r\ e_r) \wedge \\ \texttt{eval\_expr}\ e_b\ P\ E\ (\texttt{V\_Inr}\ v_r) \wedge \\ \texttt{eval\_expr}\ e_r\ P\ (\texttt{update\_env}\ x_r\ v_r\ E)\ v \end{array} \right) \vee \\
\exists x : (e = \texttt{E\_Var}\ x) \wedge (\texttt{lookup\_env}\ E\ x = \texttt{OK}\ v) \vee \\
\exists e_1, e_2, x, v' : \left(\begin{array}{l} (e = \texttt{E\_Let}\ x\ e_1\ e_2) \wedge \\ \texttt{eval\_expr}\ e_1\ P\ E\ v' \wedge \\ \texttt{eval\_expr}\ e_2\ P\ (\texttt{update\_env}\ x\ v'\ E)\ v \end{array} \right) \vee \\
\exists e'', f, v', e', x' : \left(\begin{array}{l} (e = \texttt{E\_Call}\ f\ e'') \wedge \\ \texttt{eval\_expr}\ e''\ P\ E\ v' \wedge \\ \texttt{eval\_expr}\ e'\ P\ (\texttt{make\_env}\ x'\ v')\ v \wedge \\ (\texttt{OK}\ (x', e') = \texttt{lookup\_func}\ P\ f) \end{array} \right)
\end{array} \right)
\end{array}
$$

In words, the evaluation predicate is true exactly when that can be deduced from one of the rules.

The evaluation predicate was defined at the least fixed point satisfying the evaluation rules. This "least" property leads to the following induction principle, which in HOL is not an axiom but a theorem.

Theorem 3.17 (Expression Rule Induction.)

$$
\vdash \forall Q : \left(\begin{array}{l}
(\forall i, P, E : Q\,(\texttt{E\_Int}\ i)\,P\,E\,(\texttt{V\_Int}\ i)) \wedge \\
\left(\begin{array}{l} \forall e_1, e_2, P, E, o, v : \\ \quad \exists i_1, i_2 : \left(\begin{array}{l} Q\,e_1\,P\,E\,(\texttt{V\_Int}\ i_1) \wedge \\ Q\,e_2\,P\,E\,(\texttt{V\_Int}\ i_2) \wedge \\ (\texttt{eval\_oper}\ o\ i_1\ i_2 = v) \end{array} \right) \Rightarrow \\ \quad Q\,(\texttt{E\_Op}\ o\ e_1\ e_2)\,P\,E\,v \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall e_1, e_2, P, E, v_1, v_2 : \\ \quad \left(\begin{array}{l} Q\,e_1\,P\,E\,v_1 \wedge \\ Q\,e_2\,P\,E\,v_2 \end{array} \right) \Rightarrow \\ \quad Q\,(\texttt{E\_Pair}\ e_1\ e_2)\,P\,E\,(\texttt{V\_Pair}\ v_1\ v_2) \end{array} \right) \wedge \\
\vdots \\
\textit{(four more cases omitted)} \\
\vdots \\
\left(\begin{array}{l} \forall e_b, x_\ell, e_\ell, x_r, e_r, P, E, v : \\ \quad \exists v_\ell : \left(\begin{array}{l} Q\,e_b\,P\,E\,(\texttt{V\_Inl}\ v_\ell) \wedge \\ Q\,e_\ell\,P\,(\texttt{update\_env}\ x_\ell\ v_\ell\ E)\,v \end{array} \right) \Rightarrow \\ \quad Q\,(\texttt{E\_Case}\ e_b\ x_\ell\ e_\ell\ x_r\ e_r)\,P\,E\,v \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall e_b, x_\ell, e_\ell, x_r, e_r, P, E, v : \\ \quad \exists v_r : \left(\begin{array}{l} Q\,e_b\,P\,E\,(\texttt{V\_Inr}\ v_r) \wedge \\ Q\,e_r\,P\,(\texttt{update\_env}\ x_r\ v_r\ E)\,v \end{array} \right) \Rightarrow \\ \quad Q\,(\texttt{E\_Case}\ e_b\ x_\ell\ e_\ell\ x_r\ e_r)\,P\,E\,v \end{array} \right) \wedge \\
(\forall P, E, x, v : (\texttt{lookup\_env}\ E\ x = \texttt{OK}\ v) \Rightarrow Q(\texttt{E\_Var}\ x)\,P\,E\,v) \wedge \\
\left(\begin{array}{l} \forall e_1, e_2, P, E, x, v : \\ \quad \exists v' : \left(\begin{array}{l} Q\,e_1\,P\,E\,v' \wedge \\ Q\,e_2\,P\,(\texttt{update\_env}\ x\ v'\ E)\,v \end{array} \right) \Rightarrow \\ \quad Q\,(\texttt{E\_Let}\ x\ e_1\ e_2)\,P\,E\,v \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall e, P, E, v, f : \\ \quad \exists v', e', x' : \left(\begin{array}{l} Q\,e\,P\,E\,v' \wedge \\ Q\,e'\,P\,E\,(\texttt{make\_env}\ x'\ v')\,v \wedge \\ (\texttt{OK}\,(x', e') = \texttt{lookup\_func}\ P\ f) \end{array} \right) \Rightarrow \\ \quad Q\,(\texttt{E\_Call}\ f\ e)\,P\,E\,v \end{array} \right) \\
\Rightarrow \\
(\forall e, P, E, v : \texttt{eval\_expr}\ e\,P\,E\,v \Rightarrow Q\,e\,P\,E\,v)
\end{array} \right)
$$

If the Q predicate satisfies the three base cases and the nine induction steps then it is true whenever the evaluation predicate is, and possibly more often than that. This induction principle was termed *rule induction* by Winskel [Win93]. It may

not be immediately apparent, but this corresponds to "proof by induction on the structure of derivations."

A logically equivalent, but often more useful in practice, version is the following variant

Theorem 3.18 (Strong Expression Rule Induction.)

$$
\vdash \forall Q : \left(\begin{array}{l}
(\forall i, P, E : Q\,(\texttt{E\_Int}\ i)\ P\,E\,(\texttt{V\_Int}\ i)) \wedge \\
\left(\begin{array}{l} \forall e_1, e_2, P, E, o, v : \\ \qquad \exists i_1, i_2 : \left(\begin{array}{l} \texttt{eval\_expr}\ e_1\ P\,E\,(\texttt{V\_Int}\ i_1) \wedge \\ Q\,e_1\ P\,E\,(\texttt{V\_Int}\ i_1) \wedge \\ \texttt{eval\_expr}\ e_2\ P\,E\,(\texttt{V\_Int}\ i_2) \wedge \\ Q\,e_2\ P\,E\,(\texttt{V\_Int}\ i_2) \wedge \\ (\texttt{eval\_oper}\ o\ i_1\ i_2 = v) \end{array} \right) \Rightarrow \\ \qquad Q\,(\texttt{E\_Op}\ o\ e_1\ e_2)\ P\,E\,v \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall e_1, e_2, P, E, v_1, v_2 : \\ \qquad \left(\begin{array}{l} \texttt{eval\_expr}\ e_1\ P\,E\,v_1 \wedge \\ Q\,e_1\ P\,E\,v_1 \wedge \\ \texttt{eval\_expr}\ e_2\ P\,E\,v_2 \wedge \\ Q\,e_2\ P\,E\,v_2 \end{array} \right) \Rightarrow \\ \qquad Q\,(\texttt{E\_Pair}\ e_1\ e_2)\ P\,E\,(\texttt{V\_Pair}\ v_1\ v_2) \end{array} \right) \wedge \\
\vdots \\
\textit{(seven more cases omitted)} \\
\vdots \\
\left(\begin{array}{l} \forall e_1, e_2, P, E, x, v : \\ \qquad \exists v' : \left(\begin{array}{l} \texttt{eval\_expr}\ e_1\ P\,E\,v' \wedge \\ Q\,e_1\ P\,E\,v' \wedge \\ \texttt{eval\_expr}\ e_2\ P\,(\texttt{update\_env}\ x\,v'\,E)\,v \wedge \\ Q\,e_2\ P\,(\texttt{update\_env}\ x\,v'\,E)\,v \end{array} \right) \Rightarrow \\ \qquad Q\,(\texttt{E\_Let}\ x\,e_1\,e_2)\ P\,E\,v \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall e, P, E, v, f : \\ \qquad \exists v', e', x' : \left(\begin{array}{l} \texttt{eval\_expr}\ e\ P\,E\,v' \wedge \\ Q\,e\ P\,E\,v' \wedge \\ \texttt{eval\_expr}\ e'\ P\,(\texttt{make\_env}\ x'\,v')\,v \wedge \\ Q\,e'\ P\,E\,(\texttt{make\_env}\ x'\,v')\,v \wedge \\ (\texttt{OK}\,(x', e') = \texttt{lookup\_func}\ P\,f) \end{array} \right) \Rightarrow \\ \qquad Q\,(\texttt{E\_Call}\ f\,e)\ P\,E\,v \end{array} \right) \\
\Rightarrow \\
(\forall e, P, E, v : \texttt{eval\_expr}\ e\ P\,E\,v \Rightarrow Q\,e\,P\,E\,v)
\end{array} \right)
$$

which makes more assumptions available in the induction cases, namely assumptions stating that `eval_expr` holds for subterms. It is proved by the use of Theorem 3.17. The proof technique will be called *strong rule induction.*

The timed semantics and in particular the nested-call timed semantics have equivalents of Theorems 3.15, 3.16, 3.17, and 3.18. Here we shall just show the rule induction theorem:

Theorem 3.19 (Timed Expression Rule Induction.)

$$
\vdash \forall Q : \left(\begin{array}{l}
(\forall N, i, P, E : Q\,N\,(\texttt{E\_Int}\ i)\,P\,E\,(\texttt{V\_Int}\ i)) \wedge \\
\left(\begin{array}{l} \forall N, e_1, e_2, P, E, o, v : \\ \quad \exists i_1, i_2 : \left(\begin{array}{l} Q\,N\,e_1\,P\,E\,(\texttt{V\_Int}\ i_1) \wedge \\ Q\,N\,e_2\,P\,E\,(\texttt{V\_Int}\ i_2) \wedge \\ (\texttt{eval\_oper}\ o\,i_1\,i_2 = v) \end{array} \right) \Rightarrow \\ \quad Q\,N\,(\texttt{E\_Op}\ o\,e_1\,e_2)\,P\,E\,v \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall N, e_1, e_2, P, E, v_1, v_2 : \\ \quad \left(\begin{array}{l} Q\,N\,e_1\,P\,E\,v_1 \wedge \\ Q\,N\,e_2\,P\,E\,v_2 \end{array} \right) \Rightarrow \\ \quad Q\,N\,(\texttt{E\_Pair}\ e_1\,e_2)\,P\,E\,(\texttt{V\_Pair}\ v_1\,v_2) \end{array} \right) \wedge \\
\vdots \\
\textit{(seven more cases omitted)} \\
\vdots \\
\left(\begin{array}{l} \forall N, e_1, e_2, P, E, x, v : \\ \quad \exists v' : \left(\begin{array}{l} Q\,N\,e_1\,P\,E\,v' \wedge \\ Q\,N\,e_2\,P\,(\texttt{update\_env}\ x\,v'\,E)\,v \end{array} \right) \Rightarrow \\ \quad Q\,N\,(\texttt{E\_Let}\ x\,e_1\,e_2)\,P\,E\,v \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall N, e, P, E, v, f : \\ \quad \exists v', e', x' : \left(\begin{array}{l} Q\,(\texttt{PRE}\ N)\,e\,P\,E\,v' \wedge \\ Q\,(\texttt{PRE}\ N)\,e'\,P\,E\,(\texttt{make\_env}\ x'\,v')\,v \wedge \\ (N \neq 0) \wedge \\ (\texttt{OK}\,(x', e') = \texttt{lookup\_func}\ P\,f) \end{array} \right) \Rightarrow \\ \quad Q\,N\,(\texttt{E\_Call}\ f\,e)\,P\,E\,v \end{array} \right) \\
\Rightarrow \\
(\forall N, e, P, E, v : \texttt{eval\_expr\_n}\ N\,e\,P\,E\,v \Rightarrow Q\,N\,e\,P\,E\,v)
\end{array} \right)
$$

Program Semantics

The formalisation of the program semantics is straightforward given the expression semantics. One slightly complicating point is that Figure 2 uses a pattern

matching equality to name the components — name, name of formal parameter, and body expression — of the first function in the program.

Definitional Theorem 3.20 (Semantics of Programs.)

$$\vdash \forall P, v_{in}\, v_{out} : \left(\begin{array}{l} \texttt{eval\_prg}\, P\, v_{in}\, v_{out} \Leftrightarrow \\ \left(\begin{array}{l} (P \neq [\,]) \wedge \\ \texttt{eval\_expr}\,(\texttt{SND}\,(\texttt{SND}\,(\texttt{HD}\,P)))\,P \\ \qquad (\texttt{make\_env}\,(\texttt{FST}\,(\texttt{SND}\,(\texttt{HD}\,P)))\,v_{in})\,v_{out} \end{array} \right) \end{array} \right)$$

3.6.3 Static Correctness

The formalisation of the definition of static correctness, i.e., that variables and functions are used only when defined, is straightforward. Since the formalisation of the program type made the empty program valid we also have to exclude this program from being statically correct.

Definitional Theorem 3.21 (Static Correctness)

$$\vdash \forall P : \texttt{stat\_ok}\, P \Leftrightarrow \left(\begin{array}{l} (P \neq [\,]) \wedge \\ \texttt{no\_rebinds}\, P \wedge \\ \texttt{EVERY}\,(\texttt{check\_names}\,(\texttt{functions}\,P))\,P \end{array} \right)$$

This definition uses some utility functions to be defined below. The function `functions` calculates the list of function names in a program.

Definitional Theorem 3.22 (Bound Functions)

$$\vdash \left(\begin{array}{l} (\texttt{functions}\,[\,] = [\,]) \wedge \\ (\forall x, P : \texttt{functions}\,(\texttt{CONS}\,x\,P) = \texttt{CONS}\,(\texttt{FST}\,x)\,(\texttt{functions}\,P)) \end{array} \right)$$

The function `no_rebinds` checks that no function name is bound twice in a program.

Definitional Theorem 3.23

$$\vdash \left(\begin{array}{l} (\texttt{no\_rebinds}\,[\,] = T) \wedge \\ \forall x, P : \texttt{no\_rebinds}\,(\texttt{CONS}\,x\,P) = \texttt{no\_rebinds}\,P \wedge \\ \qquad \neg(\texttt{MEM}\,(\texttt{FST}\,x)\,(\texttt{functions}\,P)) \end{array} \right)$$

The function `check_names` takes a set F and a triple (f, x, e) as arguments and checks that all functions called in e belong to F and that all variables in e are used correctly. (Note, that in this and the following definitions we use F for a variable of list type, not for a constant of type *bool*. HOL allows this.)

Definitional Theorem 3.24

$$\vdash \forall F, f, x, e : \texttt{check\_names}\, F\, (f, x, e) = \texttt{check\_vars}\, [x]\, e \wedge \texttt{check\_funcs}\, F\, e$$

The function `check_vars` checks that the list V contains all free variables of an expression, i.e., that variables not in that set are used in scope only.

Definitional Theorem 3.25

$$\vdash \left\{ \begin{array}{l} \forall V, i : \texttt{check\_vars}\, V\, (\texttt{E\_Int}\, i) = T \\ \forall V, o, e_1, e_2 : \texttt{check\_vars}\, V\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \\ \quad \texttt{check\_vars}\, V\, e_1 \wedge \texttt{check\_vars}\, V\, e_2 \\ \qquad \vdots \\ \forall V, e, x_\ell, e_\ell, x_r, e_r : \texttt{check\_vars}\, V\, (\texttt{E\_Case}\, e\, x_\ell\, e_\ell\, x_r\, e_r) = \\ \quad \texttt{check\_vars}\, V\, e \wedge \\ \quad \texttt{check\_vars}\, (\texttt{CONS}\, x_\ell\, V)\, e_\ell \wedge \texttt{check\_vars}\, (\texttt{CONS}\, x_r\, V)\, e_r \\ \forall V, x : \texttt{check\_vars}\, V\, (\texttt{E\_Var}\, x) = \texttt{MEM}\, x\, V \\ \qquad \vdots \\ \forall V, x, e_1, e_2 : \texttt{check\_vars}\, V\, (\texttt{E\_Let}\, x\, e_1\, e_2) = \\ \quad \texttt{check\_vars}\, V\, e_1 \wedge \texttt{check\_vars}\, (\texttt{CONS}\, x\, V)\, e_2 \end{array} \right\}$$

The function `check_funcs` checks that all functions called in an expression belongs to a certain set.

Definitional Theorem 3.26

$$\vdash \left\{ \begin{array}{l} \forall F, i : \texttt{check\_funcs}\, F\, (\texttt{E\_Int}\, i) = T \\ \forall F, o, e_1, e_2 : \texttt{check\_funcs}\, F\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \\ \quad \texttt{check\_funcs}\, F\, e_1 \wedge \texttt{check\_funcs}\, F\, e_2 \\ \qquad \vdots \\ \forall F, x : \texttt{check\_funcs}\, F\, (\texttt{E\_Var}\, x) = T \\ \qquad \vdots \\ \forall F, f, e : \texttt{check\_funcs}\, F\, (\texttt{E\_Call}\, f\, e) = \\ \quad \texttt{check\_funcs}\, F\, e \wedge (\texttt{MEM}\, f\, F) \\ \forall F, x, e_1, e_2 : \texttt{check\_funcs}\, F\, (\texttt{E\_Let}\, x\, e_1\, e_2) = \\ \quad \texttt{check\_funcs}\, F\, e_1 \wedge \texttt{check\_vars}\, F\, e_2 \end{array} \right\}$$

3.7 Types: An Aside

Even though our language is not strongly typed as we have presented it, it is still possible to add types on top of it. We have not pursued this approach very far but in the following we present the basic foundation needed.

We will use the following grammar to define types, and we will not consider polymorphism.

$$\tau ::= \texttt{int} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid t \mid \mu\, t.\tau \tag{3.47}$$

where t ranges over some set of type variables, *TVar*. We need the explicit recursion because circular objects are not allowed in HOL. (For example, HOL does not allow the first component of a pair to be the pair itself.) Our intended meaning with this is to have the atomic type `int`, a pair type, a sum type, and recursive types. We have formalised this type in HOL using constructors `T_Int`, `T_Pair`, `T_Sum`, `T_Var`, and `T_Rec`.

For a list of integers, for example, we will use the type

$$\mu t.\, \texttt{int} + (\texttt{int} \times t) \tag{3.48}$$

(A richer type language might have used some kind of unit type instead of the left-hand `int`. The right-hand `int` carries the data.) This type describes values like `L 0`, `R (1,L 0)`, and `R (1,R (2,R (3,L 0)))`, i.e., the empty list, the list containing just 1, and the list containing 1, 2, and 3.

$$\frac{}{TE \vdash i : \texttt{int}} \tag{3.49}$$

$$\frac{TE \vdash v_1 : \tau_1 \qquad TE \vdash v_2 : \tau_2}{TE \vdash (v_1, v_2) : \tau_1 \times \tau_2} \tag{3.50}$$

$$\frac{TE \vdash v : \tau_1}{TE \vdash \texttt{L}\ v : \tau_1 + \tau_2} \tag{3.51}$$

$$\frac{TE \vdash v : \tau_2}{TE \vdash \texttt{R}\ v : \tau_1 + \tau_2} \tag{3.52}$$

$$\frac{\texttt{lookup\_env}\ TE\, t = \tau \qquad TE \vdash v : \tau}{TE \vdash v : t} \tag{3.53}$$

$$\frac{TE + \{t \mapsto \mu\, t.\tau\} \vdash v : \tau}{TE \vdash v : \mu\, t.\tau} \tag{3.54}$$

Figure 7: Value typing.

Since our values do not contain cycles, i.e., traversal of a value is a finite process, it is straightforward to define what it means for a given value to have a given type. This is done with the inference rules[6] in Figure 7, and the relation $TE \vdash v : \tau$ is defined to be the unique least fixed-point of those rules. A judgement of the form $TE \vdash v : \tau$ means that the value v has type τ given that the free type variables

[6] We overload the definition of the environment operations to handle type environments also.

of τ interpreted as specified by the type environment *TE*. We have formalised this relation in HOL as the predicate `value_has_type`. As a result of the formalisation we obtain rule, cases, and induction theorems similar to those obtained for the evaluation predicates.

One might naïvely think that something similar would work for expressions but that is not so. For the `case` construct, for example, we would need a sum type for the controlling component. But this sum type might be hidden in a μ-type so we need an equality concept for type that would allow us to conclude, for example,

$$\mu\, t.\, \mathtt{int} + (\mathtt{int} \times t) \equiv \mathtt{int} + (\mathtt{int} \times (\mu\, t.\, \mathtt{int} + (\mathtt{int} \times t))) \tag{3.55}$$

i.e., that we can unfold a recursive type when we need to.

$$\frac{}{(TE_1, \mathtt{int}) \equiv (TE_2, \mathtt{int})} \; - \tag{3.56}$$

$$\frac{(TE_1, \tau_{11}) \equiv (TE_2, \tau_{21}) \qquad (TE_1, \tau_{12}) \equiv (TE_2, \tau_{22})}{(TE_1, \tau_{11} \times \tau_{12}) \equiv (TE_2, \tau_{21} \times \tau_{22})} \; - \tag{3.57}$$

$$\frac{(TE_1, \tau_{11}) \equiv (TE_2, \tau_{21}) \qquad (TE_1, \tau_{12}) \equiv (TE_2, \tau_{22})}{(TE_1, \tau_{11} + \tau_{12}) \equiv (TE_2, \tau_{21} + \tau_{22})} \; - \tag{3.58}$$

$$\frac{(TE_1, \tau_1) \equiv (TE_2, \tau_2) \qquad \mathtt{lookup\_env}\ TE_1\, t = \tau_1}{(TE_1, t) \equiv (TE_2, \tau_2)} \; - \tag{3.59}$$

$$\frac{(TE_1, \tau_1) \equiv (TE_2, \tau_2) \qquad \mathtt{lookup\_env}\ TE_2\, t = \tau_2}{(TE_1, \tau_1) \equiv (TE_2, t)} \; - \tag{3.60}$$

$$\frac{(TE_1 + \{t \mapsto \mu\, t.\tau_1\}, \tau_1) \equiv (TE_2, \tau_2)}{(TE_1, \mu\, t.\tau_1) \equiv (TE_2, \tau_2)} \; - \tag{3.61}$$

$$\frac{(TE_1, \tau_1) \equiv (TE_2 + \{t \mapsto \mu\, t.\tau_2\}, \tau_2)}{(TE_1, \tau_1) \equiv (TE_2, \mu\, t.\tau_2)} \; - \tag{3.62}$$

Figure 8: Type equivalence.

In Figure 8 we define such an equivalence for types and type environments by giving inference rules for it. Note however, that we need to make a co-inductive definition [Win93] in order to make the relation large enough. loosely speaking, with a co-inductive definition we compare the infinitely unfolded types. If we had used an inductive definition it would not even become reflexive: the type $\mu\, t.(\mathtt{int} + t)$ would not be equivalent to itself, for example.

HOL does not come with tools for defining co-inductive definitions but it is not too difficult to construct such a tool from the fixed-point theory. Such tools

work by constructing a functional from a textual representation of the rules. This functional is then proven to be monotonic. The general greatest fixed-point theorem is then derived and used to prove the existence of a predicate, `type_eq` in our case, that has the greatest fixed-point property. This existence theorem is used to define the constant in the HOL logic.

We have constructed such a general co-inductive definition tool, and although it is a bit weak with respect to automation of the monotonicity of the rules[7] it is clear that Harrison's [Har95] monotonicity prover could be used almost without changes.

When defining a co-inductive relation we obtain the rule and cases theorem as for an inductive relation. We will not show these here. Furthermore, we get the greatest fixed-point theorem. For `type_eq` which is the HOL-formalisation of type equivalence we get the following theorem.

Theorem 3.27 (Type Equivalence Co-Induction Theorem.)

$$
\vdash \forall P : \left(\begin{array}{l}
\forall TE_1, t_1, TE_2, t_2 : P\ TE_1\ t_1\ TE_2\ t_2 \Rightarrow \\
\quad \left(\begin{array}{l}
(t_1 = \texttt{T\_Int}) \wedge (t_2 = \texttt{T\_Int}) \vee \\
\exists t_{11}, t_{12}, t_{21}, t_{22} : \left(\begin{array}{l} (t_1 = \texttt{T\_Pair}\ t_{11}\ t_{12}) \wedge \\ (t_2 = \texttt{T\_Pair}\ t_{21}\ t_{22}) \wedge \\ P\ TE_1\ t_{11}\ TE_2\ t_{21} \wedge \\ P\ TE_1\ t_{12}\ TE_2\ t_{22} \end{array} \right) \vee \\
\exists t_{11}, t_{12}, t_{21}, t_{22} : \left(\begin{array}{l} (t_1 = \texttt{T\_Sum}\ t_{11}\ t_{12}) \wedge \\ (t_2 = \texttt{T\_Sum}\ t_{21}\ t_{22}) \wedge \\ P\ TE_1\ t_{11}\ TE_2\ t_{21} \wedge \\ P\ TE_1\ t_{12}\ TE_2\ t_{22} \end{array} \right) \vee \\
\exists x, t_1' : \left(\begin{array}{l} (t_1 = \texttt{T\_Var}\ x) \wedge \\ P\ TE_1\ t_1'\ TE_2\ t_2 \wedge \\ (\texttt{lookup\_env}\ TE_1\ x = \texttt{OK}\ t_1') \end{array} \right) \vee \\
\exists x, t_2' : \left(\begin{array}{l} (t_2 = \texttt{T\_Var}\ x) \wedge \\ P\ TE_1\ t_1\ TE_2\ t_2' \wedge \\ (\texttt{lookup\_env}\ TE_2\ x = \texttt{OK}\ t_2') \end{array} \right) \vee \\
\exists x, t_1' : \left(\begin{array}{l} (t_1 = \texttt{T\_Rec}\ x\ t_1') \wedge \\ P\ (\texttt{CONS}\ (x, t_1)\ TE_1)\ t_1'\ TE_2\ t_2 \end{array} \right) \vee \\
\exists x, t_2' : \left(\begin{array}{l} (t_2 = \texttt{T\_Rec}\ x\ t_2') \wedge \\ P\ TE_1\ t_1\ (\texttt{CONS}\ (x, t_2)\ TE_2)\ t_2' \end{array} \right)
\end{array} \right) \\
\Rightarrow \\
\forall TE_1, t_1, TE_2, t_2 : P\ TE_1\ t_1\ TE_2\ t_2 \Rightarrow \texttt{type\_eq}\ TE_1\ t_1\ TE_2\ t_2
\end{array} \right)
$$

[7] Full automation is not possible since it is not computable. Nevertheless, some approximations are better than others. Harrison in [Har95] describes a monotonicity prover which handles a very large syntactic class of rules.

From the shape of this theorem, we see that it can be used to deduce that a certain class of (TE_1, t_1, TE_2, t_2) are type equivalent, i.e., that a certain relation is a subset of the one we have defined. This is fundamentally different from an inductive rule induction theorem which can be used for proving that all members of our newly defined relation have certain properties.

The result type of a binary integer operation depends on the operator being used. We therefore define the operator environment, *OE*, to be used when we define expression typing.

$$OE = [(\texttt{+}, \texttt{int}), (\texttt{-}, \texttt{int}), (\texttt{*}, \texttt{int}), (\texttt{=}, \texttt{int} + \texttt{int})] \tag{3.63}$$

We are now ready to define the typing relation for expressions. This is done by the inductive inference rules in Figure 9.

A judgement of the form $VE, FE \vdash e : \tau$ means that expression e has type τ under the assumption that variables are typed as specified in the variable type environment *VE* and functions have the argument-type/result-type specified in *FE* environment.[8] *FE* is never modified by the rules, just passed around.

There are few surprises in the type rules, perhaps excepting Rule 3.76 which is the rule that allow us to unfold μ's in a type. The expression type relation is formalised by the HOL function `expr_has_type`.

For example, if e is known to have type $\mu\, t.\, \texttt{int} + (\texttt{int} \times t)$, then Rule 3.76 together with the equivalence from 3.55 would allow us to deduce that e also has type $\texttt{int} + (\texttt{int} \times (\mu\, t.\, \texttt{int} + (\texttt{int} \times t)))$, i.e., the sum type that corresponds to a one-level unfolding of the μt binding. If e is the control expression in a `case`-expression, `case` e `of...`, we can now continue with Rule 3.71 which *requires* a sum type for the controlling expression.

At top level, we are not interested in types with free type variables. We therefore define the concept of closed types. To get a simple definition we define closedness with respect to a set of type variables. This is the HOL formalisation.

Definitional Theorem 3.28

$$\vdash \left\{ \begin{array}{l} \forall V : \texttt{closed\_type}\, V\, \texttt{T\_Int} = T \\ \forall V, t_1, t_2 : \texttt{closed\_type}\, V\, (\texttt{T\_Pair}\, t_1\, t_2) = \\ \qquad \texttt{closed\_type}\, V\, t_1 \wedge \texttt{closed\_type}\, V\, t_2 \\ \forall V, t_1, t_2 : \texttt{closed\_type}\, V\, (\texttt{T\_Sum}\, t_1\, t_2) = \\ \qquad \texttt{closed\_type}\, V\, t_1 \wedge \texttt{closed\_type}\, V\, t_2 \\ \forall V, v : \texttt{closed\_type}\, V\, (\texttt{T\_Var}\, v) = \texttt{MEM}\, v\, V \\ \forall V, v, t : \texttt{closed\_type}\, V\, (\texttt{T\_Rec}\, v\, t) = \texttt{closed\_type}\, (\texttt{CONS}\, v\, V)\, t \end{array} \right\}$$

A type t is closed if $(\texttt{closed\_type}\, []\, t)$ holds.

[8] We again overload the environment operations.

$$\frac{}{VE,FE \vdash i : \texttt{int}} \tag{3.64}$$

$$\frac{VE,FE \vdash e_1 : \texttt{int} \quad VE,FE \vdash e_2 : \texttt{int} \quad \texttt{lookup\_env}\ OE\, o = \tau}{VE,FE \vdash \texttt{(}e_1\ o\ e_2\texttt{)} : \tau} \tag{3.65}$$

$$\frac{VE,FE \vdash e_1 : \tau_1 \qquad VE,FE \vdash e_2 : \tau_2}{VE,FE \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} : \tau_1 \times \tau_2} \tag{3.66}$$

$$\frac{VE,FE \vdash e : \tau_1 \times \tau_2}{VE,FE \vdash \texttt{fst}\ e : \tau_1} \tag{3.67}$$

$$\frac{VE,FE \vdash e : \tau_1 \times \tau_2}{VE,FE \vdash \texttt{snd}\ e : \tau_2} \tag{3.68}$$

$$\frac{VE,FE \vdash e : \tau_1}{VE,FE \vdash \texttt{inl}\ e : \tau_1 + \tau_2} \tag{3.69}$$

$$\frac{VE,FE \vdash e : \tau_2}{VE,FE \vdash \texttt{inr}\ e : \tau_1 + \tau_2} \tag{3.70}$$

$$\frac{\begin{array}{c} VE,FE \vdash e_b : \tau_\ell + \tau_r \\ VE + \{x_\ell \mapsto \tau_\ell\},FE \vdash e_\ell : \tau \qquad VE + \{x_r \mapsto \tau_r\},FE \vdash e_r : \tau \end{array}}{VE,FE \vdash \texttt{case}\ e\ \texttt{of inl.}x_\ell\texttt{->}e_\ell\ \texttt{|}\ \texttt{inr.}x_r\texttt{->}e_r\ \texttt{end} : \tau} \tag{3.71}$$

$$\frac{\texttt{lookup\_env}\ VE\, x = \tau}{VE,FE \vdash x : \tau} \tag{3.72}$$

$$\frac{}{VE,FE \vdash \texttt{error} : \tau} \tag{3.73}$$

$$\frac{VE,FE \vdash e_1 : \tau_1 \qquad VE + \{x \mapsto \tau_1\},FE \vdash e_2 : \tau_2}{VE,FE \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau_2} \tag{3.74}$$

$$\frac{\texttt{lookup\_env}\ FE\, f = (\tau_a,\tau_r) \qquad VE,FE \vdash e : \tau_a}{VE,FE \vdash f\ \ e : \tau_r} \tag{3.75}$$

$$\frac{([\,],\tau_1) \equiv ([\,],\tau_2) \qquad VE,FE \vdash e : \tau_1}{VE,FE \vdash e : \tau_2} \tag{3.76}$$

Figure 9: Expression typing.

We can now define what it means for a program to have a given type. We require that all functions in the program be given a closed type for its argument and a closed type for its body.

Definitional Theorem 3.29

$$\vdash \forall P, FE : \left(\begin{array}{l} \texttt{prg\_has\_type}\, P\, FE \Leftrightarrow \\ \left(\begin{array}{l} \texttt{stat\_ok}\, P \wedge \\ (\texttt{MAP FST}\ P = \texttt{MAP FST}\ FE) \wedge \\ \texttt{EVERY} \\ \qquad \lambda(f, t_a, t_r).\ \left(\begin{array}{l} \texttt{closed\_type}\,[]\, t_a \wedge \\ \texttt{closed\_type}\,[]\, t_r \wedge \\ \exists x, e : (\texttt{lookup\_func}\, P\, f = \texttt{OK}\,(x, e)) \Rightarrow \\ \qquad \texttt{expr\_has\_type}\, e\, FE\, [(x, t_a)]\, t_r \end{array} \right) \\ \qquad FE \end{array} \right) \end{array} \right)$$

The formalisation of programs and function type environment together with the `MAP`-conjunct imply that P and FE bind the same functions and in the same order.

* * *

At this point it would make sense to start proving theorems on the relationship between typing and evaluation, for example that evaluation of an expression with a given type produces a result with that type if it the context, i.e., the program and environment, that it is evaluated in is suitably typed. We have not pursued this, in part because our program transformations in Chapter 6 are not dependent on typing and in part because of lack of resources.

Chapter 4

Supporting Lemmas and Theorems

It has been said that man is a rational animal.
All my life I have been searching for evidence
which could support this.
— BERTRAND RUSSELL, 1872–1970

This chapter contains support structure, i.e., function definitions, lemmas, and theorems used in the upcoming chapters for proving correctness of an interpreter (Chapter 5) and a range of program transformations in Chapter 6.

In a first reading of the dissertation the reader may want to skip this chapter and to return to it when a lemma or function is referenced later. As lemmas may reference other lemmas, this makes for a *depth first* reading of the proofs, reflecting by-and-large how they were created in the first place. Another possible approach is to read the important Section 4.5 on replacement and Section 4.12 on improvement now and use the rest for reference.

In the following the lemmas and theorems are collected in groups such as "Substitution" for theorems having to do with substitution and the definitions that go with it. Some lemmas fall within one group only and are easy to place but others link several groups and are therefore placed where convenient. As a consequence of this grouping the order in which the lemmas and theorems occur here does not bear any resemblance to the order in which they were originally proven. (Using the original order is ill-suited for a linear medium.)

The timed semantics and the regular semantics are closely related. Therefore, many of the theorems which we shall present for `eval_expr_n` have obvious `eval_expr`-counterparts where the Ns are simply removed. In such cases — see for example Lemma 4.39 — we shall neither state nor prove the non-N theorems in this presentation. When, on the other hand, the `eval_expr`-theorem differs by

more than just the *N*s we do state it in full, see Lemmas 4.80 and 4.81.

4.1 Overview of Definitions

The following table shows a list of HOL functions defined in this dissertation together with a brief description of their purpose.

Function	*Definition*	*Meaning*
`alpha`	6.5	Alpha equivalence
`check_funcs`	3.26	Check that only defined functions are used
`check_names`	3.24	Check that names are used in scope only
`check_vars`	3.25	Check that variable names are used in scope only
`cst_fold`	6.1	Constant fold expression
`eval_expr`	3.15–3.17	Expression semantics predicate
`eval_expr_n`	3.19	Timed expression semantics predicate
`eval_prg`	3.20	Program semantics
`expr2int`	4.50	Convert integer expression to integer value
`func_called`	4.83	Test if a function is called in an expression
`functions`	3.22	Function names in program
`genfunc`	4.88	Generate fresh function name
`genvar`	4.29	Generate fresh variable name
`harmless`	4.36	Syntactic check for an expression being error- and loop-free
`id_func`	4.79	Test for syntactic identity function
`improvement`	4.56	Compare efficiency of programs
`improvementR`	4.73	Variant of `improvement`
`local_improvement`	4.54	Local improvement of expressions
`lookup_env`	3.10	Look up variable in environment
`lookup_func`	3.13	Look up function in program
`make_env`	3.11	Create one-point environment
`MAX`	4.1	Maximum of two numbers
`MEM`	4.2	List membership test
`no_rebinds`	3.23	Tests that no function occurs twice in a program
`rawsubst`	4.33	Substitution without regard to scope
`rename_bound`	4.32	Rename all bound variables in expression
`rename_free`	4.31	Rename free variable in expression
`repl_expr`	4.18	Test replacement in expressions
`repl_prg`	4.19	Test replacement in programs
`semeq`	4.53	Semantically equivalence
`SETMINUS`	4.4	List difference
`SETREMOVE`	4.3	List difference of single element
`stat_ok`	3.21	Test static correctness

`strict`	4.9	Identity function for forcing evaluation
`subst`	4.34	Substitution
`triv_pe`	7.1	The trivial partial evaluator
`update_env`	3.12	Add binding to environment
`used_strictly`	4.38	Syntactic check for strict usage of variable
`value2expr`	4.48	Convert value to constant expression
`var_bound`	4.26	Test whether variable is bound in expression
`var_free`	4.25	Test whether variable is used free in expression
`var_used`	4.27	Test whether variable used at all in expression
`vars_in_env`	4.35	Test whether a set of variables are bound

4.2 Basic List and Numeric Functions

In this section we define some simple list and numeric functions used, for example, for manipulating environments and variable names. A number of algebraïc theorems go with these definitions but we shall leave them out for space reasons.

The function `MAX` calculates the maximum of two numbers.

Definitional Theorem 4.1

$$\vdash \forall x, y : \texttt{MAX}\ x\, y = ((x < y) \rightarrow y \mid x)$$

The function `MEM` tests whether a value occurs in a list. We use this mostly for lists which we treat as sets so we could have used one of HOL's set libraries, but none seemed adequate for our purposes.

Definitional Theorem 4.2

$$\vdash (\forall x : \texttt{MEM}\ x\, [\,] = F) \wedge (\forall x, H, T : \texttt{MEM}\ x\, (\texttt{CONS}\ H\, T) = (x = H) \vee \texttt{MEM}\ x\, T)$$

The function `SETREMOVE` removes all occurrences of a value in a list.

Definitional Theorem 4.3

$$\vdash \left\{ \begin{array}{l} \forall x : \texttt{SETREMOVE}\, [\,]\, x = [\,] \\ \forall x, H, T : \texttt{SETREMOVE}\, (\texttt{CONS}\ H\, T)\, x = \\ \qquad ((x = H) \rightarrow \texttt{SETREMOVE}\ T\, x \mid \texttt{CONS}\ H\, (\texttt{SETREMOVE}\ T\, x)) \end{array} \right\}$$

The function `SETMINUS` iterates `SETREMOVE` over a list.

Definitional Theorem 4.4

$$\vdash \left\{ \begin{array}{l} \forall X : \texttt{SETMINUS}\ X\, [\,] = X \\ \forall X, H, T : \texttt{SETMINUS}\ X\, (\texttt{CONS}\ H\, T) = \texttt{SETMINUS}\, (\texttt{SETREMOVE}\ X\, H)\, T \end{array} \right\}$$

4.3 Reduction

4.3.1 Evaluation As a Partial Function

The following theorem states that two evaluations of the same expression relative to the same program and environment will give the same result. This theorem allows us to think of the evaluation predicate as a partial function.

Theorem 4.5 (Evaluation is deterministic.)

$$\vdash \forall e, P, E, v_1 : \left(\begin{array}{l} \texttt{eval\_expr}\ e\,P\,E\,v_1 \Rightarrow \\ \forall v_2 : \texttt{eval\_expr}\ e\,P\,E\,v_2 \Rightarrow (v_1 = v_2) \end{array} \right)$$

Proof: By rule induction, reduction, and implication resolution. □

The timed evaluation predicate has the same property as the regular predicate, even disregarding the resource limit:

Theorem 4.6 (Timed evaluation is deterministic.)

$$\vdash \forall N, e, P, E, v_1 : \left(\begin{array}{l} \texttt{eval\_expr\_n}\ N\,e\,P\,E\,v_1 \Rightarrow \\ \forall v_2, N' : \texttt{eval\_expr\_n}\ N'\,e\,P\,E\,v_2 \Rightarrow (v_1 = v_2) \end{array} \right)$$

Proof: By rule induction, reduction, and implication resolution. □

A variant of the determinism theorem is the following, which is better suited for HOL's automation because it is an equality.

Theorem 4.7 (Timed evaluation is deterministic.)

$$\vdash \forall N, e, P, E, v_1 : \left(\begin{array}{l} \texttt{eval\_expr\_n}\ N\,e\,P\,E\,v_1 \Rightarrow \\ \forall v_2 : \texttt{eval\_expr\_n}\ N\,e\,P\,E\,v_2 \Leftrightarrow (v_1 = v_2) \end{array} \right)$$

Proof: By Lemma 4.6. □

In other words when we observe one evaluation, then we can replace any other such evaluation by an equality on values.

4.3.2 Evaluation Reduction

Suppose we have an expression $\texttt{eval\_expr}\,e\,P\,E\,v$ where all but the last of the parameters are fully instantiated. In principle we can use Theorem 3.16 to perform the evaluation of *e,* but in practice this does not work at all.

HOL's automation consists primarily of rewriting, i.e., replacing an instance of the left-hand side of an equational theorem with the corresponding instance of the right-hand side. Doing this with Theorem 3.16 would always loop since the right-hand side contains numerous subterms that themselves match the left-hand side. The first step to solving this is to note that evaluation is syntax-directed (when collapsing the two cases for `case`) and to prove a theorem like the following.

Theorem 4.8

$$\vdash \left\{ \begin{array}{l}
\forall i, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Int}\ i)\, P\, E\, v \Leftrightarrow (v = \texttt{V_Int}\ i) \\
\forall o, e_1, e_2, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Op}\ o\, e_1\, e_2)\, P\, E\, v \Leftrightarrow \\
\qquad \exists i_1, i_2 : \left(\begin{array}{l} \texttt{eval_expr_n}\, N\, e_1\, P\, E\, (\texttt{V_Int}\ i_1) \wedge \\ \texttt{eval_expr_n}\, N\, e_2\, P\, E\, (\texttt{V_Int}\ i_2) \wedge \\ (\texttt{eval_oper}\ o\, i_1\, i_2 = v) \end{array} \right) \\
\forall e_1, e_2, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Pair}\ e_1\, e_2)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v_1, v_2 : \left(\begin{array}{l} (v = \texttt{V_Pair}\ v_1\, v_2) \wedge \\ \texttt{eval_expr_n}\, N\, e_1\, P\, E\, v_1 \wedge \\ \texttt{eval_expr_n}\, N\, e_2\, P\, E\, v_2 \end{array} \right) \\
\forall e, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Fst}\ e)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v_2 : \texttt{eval_expr_n}\, N\, e\, P\, E\, (\texttt{V_Pair}\ v\, v_2) \\
\forall e, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Snd}\ e)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v_1 : \texttt{eval_expr_n}\, N\, e\, P\, E\, (\texttt{V_Pair}\ v_1\, v) \\
\qquad\qquad \vdots \\
\qquad \textit{(three more cases omitted)} \\
\qquad\qquad \vdots \\
\forall N, P, E, v : \neg(\texttt{eval_expr_n}\, N\, \texttt{E_Error}\, P\, E\, v) \\
\forall e, x_\ell, e_\ell, x_r, e_r, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Case}\ e\, x_\ell\, e_\ell\, x_e\, e_r)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v' : \left(\begin{array}{l} \texttt{eval_expr_n}\, N\, e\, P\, E\, (\texttt{V_Inl}\ v') \wedge \\ \texttt{eval_expr_n}\, N\, e_\ell\, P\, (\texttt{update_env}\ x_\ell\, v'\, E)\, v \end{array} \right) \vee \\
\qquad \exists v' : \left(\begin{array}{l} \texttt{eval_expr_n}\, N\, e\, P\, E\, (\texttt{V_Inr}\ v') \wedge \\ \texttt{eval_expr_n}\, N\, e_r\, P\, (\texttt{update_env}\ x_r\, v'\, E)\, v \end{array} \right) \\
\forall x, e_1, e_2, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Let}\ x\, e_1\, e_2)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v_1 : \left(\begin{array}{l} \texttt{eval_expr_n}\, N\, e_1\, P\, E\, v_1 \wedge \\ \texttt{eval_expr_n}\, N\, e_2\, P\, (\texttt{update_env}\ x\, v_1\, E)\, v \end{array} \right) \\
\forall f, e, N, P, E, v : \texttt{eval_expr_n}\, N\, (\texttt{E_Call}\ f\, e)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v', x, e' : \left(\begin{array}{l} \texttt{eval_expr_n}\, (\texttt{PRE}\, N)\, e\, P\, E\, v' \wedge \\ \texttt{eval_expr_n}\, (\texttt{PRE}\, N)\, e'\, P\, (\texttt{make_env}\ x\, v')\, v \\ (\texttt{OK}\, (x, e') = \texttt{lookup_func}\ P\, f) \wedge \\ (N \neq 0) \end{array} \right)
\end{array} \right\}$$

Proof: By the `eval_expr_n` counterpart of Theorem 3.16. □

None of Theorem 4.8's right-hand sides match any of the left-hand sides[1] so continued rewriting with respect to Theorem 4.8 will now in principle expand an expression to its meaning. But it still does not work in practice.

The problem this time has to do with limited resources. When the `case`-equation, for example, is used then the corresponding right-hand side has three instances of `eval_expr_n` instead of the one they replaced. Further rewriting of each of these might produce a number of branches which is exponential in the number of passes of rewriting. Provided that the controlling expression terminates, it will sooner or later produce a value which allows us to eliminate one of the disjuncts.

For the proofs in Chapter 5 this "sooner or later" is much too late: the body of the `eval` function contains eleven nested `case` expressions plus several `let`-bindings. The estimated memory consumption of brute-force expansion by Theorem 4.8 is easily in the GB-range. Since rewriting time is proportional to term size this might be considered another problem.

To solve these practical problems with proofs we define the following function which is an identity function with an extra, but unused, parameter.

Definitional Theorem 4.9

$$\vdash \left\{ \begin{array}{l} \forall x, i : \texttt{strict}\,(\texttt{V\_Int}\ i)\,x = x \\ \forall x, v_1, v_2 : \texttt{strict}\,(\texttt{V\_Pair}\ v_1\ v_2)\,x = x \\ \forall x, v : \texttt{strict}\,(\texttt{V\_Inl}\ v)\,x = x \\ \forall x, v : \texttt{strict}\,(\texttt{V\_Inr}\ v)\,x = x \end{array} \right\}$$

If we do rewriting with respect to Theorem 4.9 in a term where $(\texttt{strict}\ v x)$ occurs then rewriting will take place only if v's outermost constructor is known. Otherwise $(\texttt{strict}\ v x)$ does not match any of Theorem 4.9's left-hand sides. We are going to use this to create a practically useful version of Theorem 4.8 but first we will prove that we can always get rid of `strict` at will.

Lemma 4.10 (Strict Elimination Lemma.)

$$\vdash \forall v, x : \texttt{strict}\ v x = x$$

Proof: By structural case analysis on v. □

Experience shows that the problematic constructs are `let` and `case` and not binary operations and pairing, even though they also contain several subexpressions.

[1] Note the wording. An *instance* of a right-hand side might very well match a left-hand side but this is not a problem because the expression parameter will have gotten smaller.

Since applications of `strict` have a tendency of getting in the way for automation — that is after all the purpose of inventing them in the first place — we will use our experience and only put blocks on evaluation of `let` and `case`. For `let` we require that the bound expression is evaluated in advance before we go on, and for `case` we require that the controlling expression be evaluated.

Theorem 4.11 (Evaluation Reduction Theorem.)

$$
\vdash \left\{ \begin{array}{l}
\forall i,N,P,E,v : \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Int}\ i)\, P\, E\, v \Leftrightarrow (v = \texttt{V\_Int}\ i) \\
\forall o,e_1,e_2,N,P,E,v : \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Op}\ o\, e_1\, e_2)\, P\, E\, v \Leftrightarrow \\
\qquad \exists i_1,i_2 : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, (\texttt{V\_Int}\ i_1) \wedge \\ \texttt{eval\_expr\_n}\, N\, e_2\, P\, E\, (\texttt{V\_Int}\ i_2) \wedge \\ (\texttt{eval\_oper}\ o\, i_1\, i_2 = v) \end{array} \right) \\
\forall e_1,e_2,N,P,E,v : \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Pair}\ e_1\, e_2)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v_1,v_2 : \left(\begin{array}{l} (v = \texttt{V\_Pair}\ v_1\, v_2) \wedge \\ \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v_1 \wedge \\ \texttt{eval\_expr\_n}\, N\, e_2\, P\, E\, v_2 \end{array} \right) \\
\forall e,N,P,E,v : \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Fst}\ e)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v_2 : \texttt{eval\_expr\_n}\, N\, e\, P\, E\, (\texttt{V\_Pair}\ v\, v_2) \\
\qquad\qquad \vdots \\
\qquad\qquad \textit{(four more cases omitted)} \\
\qquad\qquad \vdots \\
\forall N,P,E,v : \neg(\texttt{eval\_expr\_n}\, N\, \texttt{E\_Error}\, P\, E\, v) \\
\forall e,x_\ell,e_\ell,x_r,e_r,N,P,E,v : \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Case}\ e\, x_\ell\, e_\ell\, x_e\, e_r)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v' : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P\, E\, (\texttt{V\_Inl}\ v') \wedge \\ \texttt{eval\_expr\_n}\, N\, (\texttt{strict}\ v'\, e_\ell)\, P\, (\texttt{update\_env}\ x_\ell\, v'\, E)\, v \end{array} \right) \vee \\
\qquad \exists v' : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P\, E\, (\texttt{V\_Inr}\ v') \wedge \\ \texttt{eval\_expr\_n}\, N\, (\texttt{strict}\ v'\, e_r)\, P\, (\texttt{update\_env}\ x_r\, v'\, E)\, v \end{array} \right) \\
\forall x,e_1,e_2,N,P,E,v : \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Let}\ x\, e_1\, e_2)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v_1 : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v_1 \wedge \\ \texttt{eval\_expr\_n}\, N\, (\texttt{strict}\ v_1\, e_2)\, P\, (\texttt{update\_env}\ x\, v_1\, E)\, v \end{array} \right) \\
\forall f,e,N,P,E,v : \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Call}\ f\, e)\, P\, E\, v \Leftrightarrow \\
\qquad \exists v',x,e' : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, (\texttt{PRE}\, N)\, e\, P\, E\, v' \wedge \\ \texttt{eval\_expr\_n}\, (\texttt{PRE}\, N)\, e'\, P\, (\texttt{make\_env}\ x\, v')\, v \\ (\texttt{OK}\, (x,e') = \texttt{lookup\_func}\, P\, f) \wedge \\ (N \neq 0) \end{array} \right)
\end{array} \right\}
$$

Proof: By Lemma 4.10 and Theorem 4.8. □

4.4 Timed Evaluation Approximates Evaluation

The following lemmas show that timed evaluation approximates regular evaluation in the monotonic way required by equations 3.20 and 3.19.

Lemma 4.12

$$\vdash \forall N, e, P, E, v : \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v \Rightarrow \texttt{eval\_expr}\ e\ P\ E\ v$$

Proof: By rule induction. □

Lemma 4.13

$$\vdash \forall N, e, P, E, v : \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v \Rightarrow \texttt{eval\_expr\_n}\ (\texttt{SUC}\ N)\ e\ P\ E\ v$$

Proof: By rule induction. □

Lemma 4.14

$$\vdash \forall N, N', e, P, E, v : \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v \Rightarrow \texttt{eval\_expr\_n}\ (N + N')\ e\ P\ E\ v$$

Proof: By induction on N and Lemma 4.13. □

Lemma 4.15

$$\vdash \forall e, P, E, v : \texttt{eval\_expr}\ e\ P\ E\ v \Rightarrow \exists N : \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v$$

Proof: By rule induction. This leaves cases for expression constructs with two or more subexpressions. For these cases we use the sum of the N's provided by the induction hypotheses, and follow up by using Lemma 4.14. □

Lemma 4.16

$$\vdash \forall e, P, E, v : \texttt{eval\_expr}\ e\ P\ E\ v \Leftrightarrow \exists N : \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v$$

Proof: By Lemmas 4.12 and 4.15. □

Lemma 4.17

$$\vdash \forall N, e, P, E, v : \texttt{eval\_expr\_n}\ (\texttt{PRE}\ N)\ e\ P\ E\ v \Rightarrow \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v$$

Proof: By splitting into cases $N = 0$ and $N = \texttt{SUC}\ M$. The first case follows from $\texttt{PRE}\ 0 = 0$ and the second case follows from Lemma 4.13. □

4.5 Replacement

Many of the program transformations we shall discuss have the informal form "replace X by Y." However, we do not necessarily want to replace *every* occurrence of X, so instead of presenting replacement as a function we shall present it as a predicate that checks whether two given programs or expressions are identical except that some occurrences of X have been replaced by Y. This allows us to discuss situations like where we locally replace `(2+2)` by `4` without considering that `(2+2)` might occur elsewhere in the program.

Moreover, we will want to restrict the context in which replacement can take place in such a way that we can guarantee that certain variables are bound in that context. We want this in order to be able to replace `fst (2,x)` by `2` safely, i.e., only where `x` is bound. We therefore define $\texttt{repl\_expr}\, V\, e_1\, e_2\, e_a\, e_b$ to be true exactly when e_2 can be obtained from e_1 by replacing some occurrences of e_a by e_b but only at places where all variables in the list V have been bound:

Definitional Theorem 4.18 (Replacement for Expressions.)

$$\vdash \left\{ \begin{array}{l} \forall V, i, e', e_a, e_b : \texttt{repl\_expr}\, V\, (\texttt{E\_Int}\, i)\, e'\, e_a\, e_b = \\ \quad (\texttt{E\_Int}\, i = e_a) \wedge (e' = e_b) \wedge (V = [\,]) \vee (\texttt{E\_Int}\, i = e') \\ \forall V, o, e_1, e_2, e', e_a, e_b : \texttt{repl\_expr}\, V\, (\texttt{E\_Op}\, o\, e_1\, e_2)\, e'\, e_a\, e_b = \\ \quad (\texttt{E\_Op}\, o\, e_1\, e_2 = e_a) \wedge (e' = e_b) \wedge (V = [\,]) \vee \\ \quad \exists e_1', e_2' : (e' = \texttt{E\_Op}\, o\, e_1'\, e_2') \wedge \\ \qquad \texttt{repl\_expr}\, V\, e_1\, e_1'\, e_a\, e_b \wedge \texttt{repl\_expr}\, V\, e_2\, e_2'\, e_a\, e_b \\ \qquad\qquad \vdots \\ \forall V, x, e_1, e_2, e', e_a, e_b : \texttt{repl\_expr}\, V\, (\texttt{E\_Let}\, x\, e_1\, e_2)\, e'\, e_a\, e_b = \\ \quad (\texttt{E\_Let}\, x\, e_1\, e_2 = e_a) \wedge (e' = e_b) \wedge (V = [\,]) \vee \\ \quad \exists e_1', e_2' : (e' = \texttt{E\_Let}\, x\, e_1'\, e_2') \wedge \\ \qquad \texttt{repl\_expr}\, V\, e_1\, e_1'\, e_a\, e_b \wedge \\ \qquad \texttt{repl\_expr}\, (\texttt{SETREMOVE}\, V\, x)\, e_2\, e_2'\, e_a\, e_b \end{array} \right\}$$

This looks quite complicated, partly because it is written in a primitively recursive way and partly because of the details with variables. The key to understanding is to notice that the left disjunct always means that replacement took place at the given level, while the right disjunct means that replacement might have taken place in a subexpression. Alternatively the definition could have been done using an inference system, proving the above theorem as a lemma instead.

The disjunctions of the right-hand side of the equalities have a way of producing a large number of sub-cases in proofs that have replacement as a pre-condition. Many of the proofs have that as we shall see.

We need the same infrastructure for programs. Since the parameter of a function should be counted as binding a slight amount of care is needed:

Definitional Theorem 4.19 (Replacement for Programs.)

$$\vdash \left\{ \begin{array}{l} \forall V, P', e_a, e_b : \texttt{repl\_prg}\, V\, [\,]\, P'\, e_a\, e_b = (P' = [\,]) \\ \forall V, P, b, P', e_a, e_b : \texttt{repl\_prg}\, V\, (\texttt{CONS}\, b\, P)\, P'\, e_a\, e_b = \\ \quad \exists f, x, e, e', P'' : ((f, x, e) = b) \wedge \\ \quad\quad ((\texttt{CONS}\, (f, x, e')\, P'') = P') \wedge \\ \quad\quad \texttt{repl\_expr}\, (\texttt{SETREMOVE}\, V\, x)\, e\, e'\, e_a\, e_b \wedge \\ \quad\quad \texttt{repl\_prg}\, V\, P\, P''\, e_a\, e_b \end{array} \right\}$$

Not surprisingly, replacement is reflexive and symmetric or reversible in a certain sense. The symmetry is especially useful as it allows us to reduce applications of `repl_expr`/`repl_prg` which have their second expression/program argument instantiated but not the first: First we rewrite once with respect to the symmetry equations, then we rewrite with respect to the definition above, and finally we might rewrite once again with respect to the symmetry equation.

Theorem 4.20 (Replacement for Expressions is Reflexive.)

$$\vdash \forall V, e, e_a, e_b : \texttt{repl\_expr}\, V\, e\, e\, e_a\, e_b$$

Proof: By structural induction on e and a lemma stating that `repl_expr` is closed under removal of elements from the variable set. □

Theorem 4.21 (Replacement for Expressions is Symmetric.)

$$\vdash \forall e_1, e_2, e_a, e_b, V : \texttt{repl\_expr}\, V\, e_1\, e_2\, e_a\, e_b \Leftrightarrow \texttt{repl\_expr}\, V\, e_2\, e_1\, e_b\, e_a$$

Proof: By reduction to bi-implication and structural induction. □

Theorem 4.22 (Replacement for Programs is Reflexive.)

$$\vdash \forall V, P, e_a, e_b : \texttt{repl\_prg}\, V\, P\, P\, e_a\, e_b$$

Proof: By list induction and Theorem 4.20. □

Theorem 4.23 (Replacement for Programs is Symmetric.)

$$\vdash \forall P_1, P_2, e_a, e_b, V : \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Leftrightarrow \texttt{repl\_prg}\, V\, P_2\, P_1\, e_b\, e_a$$

Proof: By reduction to bi-implication, list induction, and Theorem 4.21. □

The following important lemma characterises what happens to functions defined in a program when the program undergoes replacement:

Lemma 4.24

$$\vdash \forall P_1, P_2, f, V, x, e : \left(\begin{array}{l} (\texttt{OK}\,(x,e) = \texttt{lookup\_func}\; P_1\; f) \Rightarrow \\ \texttt{repl\_prg}\; V\; P_1\; P_2\; e_a\; e_b \Rightarrow \\ \exists e' : (\texttt{OK}\,(x,e') = \texttt{lookup\_func}\; P_2\; f) \Rightarrow \\ \qquad \texttt{repl\_expr}\,(\texttt{SETREMOVE}\; V\; x)\; e\; e'\; e_a\; e_b \end{array} \right)$$

Proof: By list induction. □

In words, if two programs are identical except for replacing one expression with another, then any function body that we find in one program has a twin in the other except for the same replacement.

4.6 Variables

This section defines predicates to test whether a given variable name (number) is used in an expression and in what ways. Then a function, `genvar`, producing locally unique variable names is introduced.

Definitional Theorem 4.25 (Variable Free in Expression.)

$$\vdash \left\{ \begin{array}{l} \forall x, i : \texttt{var\_free}\; x\,(\texttt{E\_Int}\; i) = F \\ \forall x, o, e_1, e_2 : \texttt{var\_free}\; x\,(\texttt{E\_Op}\; o\, e_1\, e_2) = \\ \qquad \texttt{var\_free}\; x\, e_1 \vee \texttt{var\_free}\; x\, e_2 \\ \qquad\qquad \vdots \\ \forall x, x' : \texttt{var\_free}\; x\,(\texttt{E\_Var}\; x') = (x' = x) \\ \qquad\qquad \vdots \\ \forall x, x', e_1, e_2 : \texttt{var\_free}\; x\,(\texttt{E\_Let}\; x'\, e_1\, e_2) = \\ \qquad \texttt{var\_free}\; x\, e_1 \vee (x' \neq x) \wedge \texttt{var\_free}\; x\, e_2 \end{array} \right\}$$

Definitional Theorem 4.26 (Variable Bound in Expression.)

$$\vdash \left\{ \begin{array}{l} \forall x, i : \texttt{var\_bound}\; x\,(\texttt{E\_Int}\; i) = F \\ \forall x, o, e_1, e_2 : \texttt{var\_bound}\; x\,(\texttt{E\_Op}\; o\, e_1\, e_2) = \\ \qquad \texttt{var\_bound}\; x\, e_1 \vee \texttt{var\_bound}\; x\, e_2 \\ \qquad\qquad \vdots \\ \forall x, x' : \texttt{var\_bound}\; x\,(\texttt{E\_Var}\; x') = F \\ \qquad\qquad \vdots \\ \forall x, x', e_1, e_2 : \texttt{var\_bound}\; x\,(\texttt{E\_Let}\; x'\, e_1\, e_2) = \\ \qquad (x' = x) \vee \texttt{var\_bound}\; x\, e_1 \vee \texttt{var\_bound}\; x\, e_2 \end{array} \right\}$$

Definitional Theorem 4.27 (Variable Used in Expression.)

$$\vdash \left\{ \begin{array}{l} \forall x, i : \texttt{var\_used}\, x\, (\texttt{E\_Int}\, i) = F \\ \forall x, o, e_1, e_2 : \texttt{var\_used}\, x\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \\ \qquad \texttt{var\_used}\, x\, e_1 \vee \texttt{var\_used}\, x\, e_2 \\ \qquad\qquad \vdots \\ \forall x, x' : \texttt{var\_used}\, x\, (\texttt{E\_Var}\, x') = (x' = x) \\ \qquad\qquad \vdots \\ \forall x, x', e_1, e_2 : \texttt{var\_used}\, x\, (\texttt{E\_Let}\, x'\, e_1\, e_2) = \\ \qquad (x' = x) \vee \texttt{var\_used}\, x\, e_1 \vee \texttt{var\_used}\, x\, e_2 \end{array} \right\}$$

The following theorem could have been used as an alternative definition:

Lemma 4.28

$$\vdash \forall e, x : \texttt{var\_used}\, x\, e \Leftrightarrow \texttt{var\_free}\, x\, e \vee \texttt{var\_bound}\, x\, e$$

Proof: By structural induction. □

There is a slight amount of textual redundancy by defining all three predicates by primitive recursion but it pays to have them on that form when they are to be used. (To be precise: they often occur in induction hypotheses. Using the primitive recursive form for rewriting makes it trivial to satisfy the relevant part of the induction steps' hypotheses. Using Lemma 4.28, on the other hand, will require more than just left-to-right rewriting.)
Now for the more interesting generation of fresh variables. We are unable to define a global oracle to create fresh variables without dragging along a list of variable names already used. This is so because HOL is a logical system and not a programming language that allows us to have a hidden counter somewhere. Instead we define this local oracle which given an expression creates a variable name that is not used in the expression:

Definitional Theorem 4.29

$$\vdash \left\{ \begin{array}{l} \forall i : \texttt{genvar}\, (\texttt{E\_Int}\, i) = 0 \\ \forall o, e_1, e_2 : \texttt{genvar}\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \texttt{MAX}\, (\texttt{genvar}\, e_1)\, (\texttt{genvar}\, e_2) \\ \qquad\qquad \vdots \\ \forall x : \texttt{genvar}\, (\texttt{E\_Var}\, x) = \texttt{SUC}\, x \\ \qquad\qquad \vdots \\ \forall x, e_1, e_2 : \texttt{genvar}\, (\texttt{E\_Let}\, x\, e_1\, e_2) = \\ \qquad \texttt{MAX}\, (\texttt{SUC}\, x)\, (\texttt{MAX}\, (\texttt{genvar}\, e_1)\, (\texttt{genvar}\, e_2)) \end{array} \right\}$$

Since the specific variable names should not matter the definition of `genvar` will only be used to prove the following theorem and in examples where some actual (non-symbolic) value is called for.

Theorem 4.30 (genvar Property.)

$$\vdash \forall e : \neg(\texttt{var\_used}\,(\texttt{genvar}\,e)\,e)$$

Proof: By first proving the slightly stronger lemma

$$\vdash \forall e, x : (x \geq \texttt{genvar}\,e) \Rightarrow \neg(\texttt{var\_used}\,x\,e)$$

by structural induction and then specialising x to `genvar` e. □

4.7 Substitution

One of the program transformations that we shall be concerned with is unfolding of `let`-expressions. For example we might want to unfold the `let` in

```
f x = let y = (3 + x) in (y * 4) end;
```

yielding

```
f x = ((3 + x) * 4);
```

What happens here is that the expression `(3 + x)` gets substituted for every occurrence of the bound variable `y` in the main expression `y * 4`.

It might at first look like a simple thing to define this substitution formally but it turns out to be far from simple. First of all the above description of what we want substitution to do is at best incomplete. We really ought to have taken care of the situation where the bound expression might mean something else in the places where it is put. For example, we should not try to unfold the outer-most `let` in the following program, by the above principles:

```
f x = let
        y = (3 + x)
      in
        let x = 1 in (y + x)
      end;
```

This is obviously because the free variable `x` in the expression `(3 + x)` would be captured by the inner-most `let`-expression:

```
f x = let x = 1 in ((3 + x) + x) end; (* Wrong *)
```

In order to prevent variable capture we must rename all bound variables that might cause capture. But it is simpler just to rename *all* bound variables to something safe, e.g., fresh variable names generated suitably by the `genvar` function introduced previously.

Unfortunately, solving the variable-capturing problem creates another problem: it is no longer trivial to define substitution formally. More precisely, the usual way of defining substitution is no longer primitive recursive: the branch for substituting e' for x' into a `let`-expression might look like

$$\begin{array}{l}\texttt{subst}\ e'\,x'\,(\texttt{E\_Let}\ x\,e_1\,e_2) = \\ \quad \text{let}\ x'' = \texttt{genvar}\ e\ \text{in} \\ \quad \texttt{E\_Let}\ x''\,(\texttt{subst}\ e'\,x'\,e_1)\,(\texttt{subst}\ e'\,x'\,(\texttt{rename\_free}\ x\,x''\,e_2))\end{array} \tag{4.1}$$

where `genvar` is as described above and `rename_free` is a function that renames all occurrences of a free variable in an expression.

Since this way of defining substitution is not primitive recursive (because the second recursive call does not recur directly on e_2 but on e_2 after renaming of free variables) the initiality theorem for expressions is not applicable. This means that we would have to prove the existence of `subst` somehow else, very likely using the property that `rename_free` does not change the size of the term in which renaming takes place. Instead of going that way we shall continue using primitive recursion and split substitution and renaming into two separate phases. For this we shall need functions to rename free and bound variables.

The function `rename_free` as used in the above discussion takes three parameters: x_1, x_2, and e. It returns e with all free occurrences of x_1 renamed to x_2, even if the renaming causes variable capture. (The circumstances under which the function will be used will prevent this from happening.)

Definitional Theorem 4.31

$$\vdash \left\{ \begin{array}{l} \forall x_1, x_2, i : \texttt{rename\_free}\ x_1\,x_2\,(\texttt{E\_Int}\ i) = (\texttt{E\_Int}\ i) \\ \forall x_1, x_2, o, e_1, e_2 : \texttt{rename\_free}\ x_1\,x_2\,(\texttt{E\_Op}\ o\,e_1\,e_2) = \\ \quad \texttt{E\_Op}\ o\,(\texttt{rename\_free}\ x_1\,x_2\,e_1)\,(\texttt{rename\_free}\ x_1\,x_2\,e_2) \\ \qquad \vdots \\ \forall x_1, x_2, x : \texttt{rename\_free}\ x_1\,x_2\,(\texttt{E\_Var}\ x) = \texttt{E\_Var}\,((x = x_1) \to x_2 \mid x) \\ \qquad \vdots \\ \forall x_1, x_2, x, e_1, e_2 : \texttt{rename\_free}\ x_1\,x_2\,(\texttt{E\_Let}\ x\,e_1\,e_2) = \\ \quad \texttt{E\_Let}\ x\,(\texttt{rename\_free}\ x_1\,x_2\,e_1) \\ \qquad ((x = x_1) \to e_2 \mid (\texttt{rename\_free}\ x_1\,x_2\,e_2)) \end{array} \right\}$$

The function `rename_bound` takes two arguments: e' and e. It returns e with all bound variables renamed in such a way that the new bound variables do not occur at all in e'.

Definitional Theorem 4.32

$$\vdash \left\{ \begin{array}{l} \forall e', i : \texttt{rename\_bound}\, e'\, (\texttt{E\_Int}\, i) = (\texttt{E\_Int}\, i) \\ \forall e', o, e_1, e_2 : \texttt{rename\_bound}\, e'\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \\ \qquad \texttt{E\_Op}\, o\, (\texttt{rename\_bound}\, e'\, e_1)\, (\texttt{rename\_bound}\, e'\, e_2) \\ \qquad\qquad \vdots \\ \forall e', x : \texttt{rename\_bound}\, e'\, (\texttt{E\_Var}\, x) = (\texttt{E\_Var}\, x) \\ \qquad\qquad \vdots \\ \forall e', x, e_1, e_2 : \texttt{rename\_bound}\, e'\, (\texttt{E\_Let}\, x\, e_1\, e_2) = \\ \qquad \mathit{let}\ e_2' = (\texttt{rename\_bound}\, e'\, e_2)\ \mathit{in} \\ \qquad \mathit{let}\ x' = (\texttt{genvar}\, (\texttt{E\_Pair}\, e'\, e_2'))\ \mathit{in} \\ \qquad \texttt{E\_Let}\, x'\, (\texttt{rename\_bound}\, e'\, e_1)\, (\texttt{rename\_free}\, x\, x'\, e_2') \end{array} \right\}$$

Notice that `rename_bound` is used in such a way that variable capture cannot occur: The new variable name does not occur in the object expression. This completes the renaming part of substitution.

We now define a primitive version of substitution that does not consider capture or even variable bindings. In other words it substitutes for every occurrence of some variable:

Definitional Theorem 4.33

$$\vdash \left\{ \begin{array}{l} \forall e', x, i : \texttt{rawsubst}\, e'\, x\, (\texttt{E\_Int}\, i) = (\texttt{E\_Int}\, i) \\ \forall e', x, o, e_1, e_2 : \texttt{rawsubst}\, e'\, x\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \\ \qquad \texttt{E\_Op}\, o\, (\texttt{rawsubst}\, e'\, x\, e_1)\, (\texttt{rawsubst}\, e'\, x\, e_2) \\ \qquad\qquad \vdots \\ \forall e', x', x : \texttt{rawsubst}\, e'\, x'\, (\texttt{E\_Var}\, x) = ((x = x') \rightarrow e' \mid \texttt{E\_Var}\, x) \\ \qquad\qquad \vdots \\ \forall e', x', x, e_1, e_2 : \texttt{rawsubst}\, e'\, x'\, (\texttt{E\_Let}\, x\, e_1\, e_2) = \\ \qquad \texttt{E\_Let}\, x\, (\texttt{rawsubst}\, e'\, x'\, e_1)\, (\texttt{rawsubst}\, e'\, x'\, e_2) \end{array} \right\}$$

The only reasonable use of function `rawsubst` is in the following definition of regular substitution. Hence the name.

Definitional Theorem 4.34

$$\vdash \forall e', x', e : \texttt{subst}\, e'\, x'\, e = \texttt{rawsubst}\, e'\, x'\, (\texttt{rename\_bound}\, (\texttt{E\_Pair}\, (\texttt{E\_Var}\, x')\, e')\, e)$$

Finally it is occasionally necessary to restrict a theorem involving environments to those environments which bind at least a certain set of variables. The following predicate is useful for that.

Definitional Theorem 4.35 (Variables Bound by Environment.)

$$\vdash \left\{ \begin{array}{l} \forall E : \texttt{vars\_in\_env}\,[\,]\,E = T \\ \forall x, X, E : \texttt{vars\_in\_env}\,(\texttt{CONS}\,x\,X)\,E = \\ \qquad \exists v.\,(\texttt{OK}\,v = \texttt{lookup\_env}\,E\,x) \wedge \texttt{vars\_in\_env}\,X\,E \end{array} \right\}$$

4.8 Harmlessness

Unfolding of `let`-bindings is one of the program transformations we shall study. Since our language is strict and since we are very concerned with termination properties of our object programs we must take care to allow only safe unfoldings. Consider the program

```
main x = let y=loop 0 in 42 end;
loop x = loop x;
```

If we unfold the `let` in this program we would end up with

```
main x = 42;
loop x = loop x;
```

which is a radically different program: The first program never terminates, the second one always does. The problem here is that the variable `y` is not used in the body of the `let`.

Unfolding of `let`-bindings might also lead to duplication of code as in the following example

```
main x = let y=(big x) in (y,y) end;
big x = ...;
```

where the unfolding program contains the call `(big x)` twice. However, we shall ignore this potential complication completely as it does not influence correctness — our language is free of side-effects. We are only concerned with the question "are we *allowed* to unfold a given `let`?" and not with the question "do we *want* to unfold it?"

Usually in programming language matters discussion about `let` unfolding stops here. We cannot do this, as the following program fragment demonstrates:

```
f x = let y=x in 2 end;
g x = let y=z in 2 end;
```

It is important for us to be able to unfold the first `let`-binding because that eliminates it completely. It is even more important that we do not unfold the second `let`-binding because doing so changes evaluation properties from never producing a result to always doing so. The only difference between `f` and `g` is that `g` uses a variable, `z`, that is not in scope where it is used.

The traditional solution to this problem is hand-waving: "We will only consider well-⟨*whatever*⟩ programs." The solution as such is correct but it only makes sense at top level. Proofs involving expressions have no way of using that information. We solve the problem by defining a syntactic property, *harmlessness*, telling when evaluation of an expression is guaranteed to succeed. The property is relative to a set of variables guaranteed to be bound where the expression occurs. Being a syntactic property it can, of course, only be an approximation of the semantic property that an expression always evaluates without erring.

Definitional Theorem 4.36 (Harmless.)

$$\vdash \left\{ \begin{array}{l} \forall V, i : \texttt{harmless}\, V\, (\texttt{E\_Int}\, i) = T \\ \forall V, o, e_1, e_2 : \texttt{harmless}\, V\, (\texttt{E\_Op}\, o\, e_1\, e_2) = F \\ \forall V, e_1, e_2 : \texttt{harmless}\, V\, (\texttt{E\_Pair}\, e_1\, e_2) = \\ \quad \texttt{harmless}\, V\, e_1 \wedge \texttt{harmless}\, V\, e_2 \\ \forall V, e : \texttt{harmless}\, V\, (\texttt{E\_Fst}\, e) = F \\ \forall V, e : \texttt{harmless}\, V\, (\texttt{E\_Snd}\, e) = F \\ \forall V, e : \texttt{harmless}\, V\, (\texttt{E\_Inl}\, e) = \texttt{harmless}\, V\, e \\ \forall V, e : \texttt{harmless}\, V\, (\texttt{E\_Inr}\, e) = \texttt{harmless}\, V\, e \\ \forall V, e_b, x_\ell, e_\ell, x_r, e_r : \texttt{harmless}\, V\, (\texttt{E\_Case}\, e_b\, x_\ell\, e_\ell\, x_r\, e_r) = F \\ \forall V, x : \texttt{harmless}\, V\, (\texttt{E\_Var}\, x) = \texttt{MEM}\, x\, V \\ \forall V : \texttt{harmless}\, V\, \texttt{E\_Error} = F \\ \forall V, f, e : \texttt{harmless}\, V\, (\texttt{E\_Call}\, f\, e) = F \\ \forall V, x, e_1, e_2 : \texttt{harmless}\, V\, (\texttt{E\_Let}\, x\, e_1\, e_2) = \\ \quad \texttt{harmless}\, V\, e_1 \wedge \texttt{harmless}\, (\texttt{CONS}\, x\, V)\, e_2 \end{array} \right\}$$

This definition of harmlessness is a very conservative approximation. In general any constructing expression is deemed harmless as its subexpressions, while any destructing expression is considered potentially harmful since a run-time type check might fail.

The V parameter is a set of variables guaranteed to be bound. Having this parameter is important as we would otherwise have to classify *every* variable as potential harmful. In particular, both `x` and `z` in the motivating example above would have to be classified as harmful.

The purpose of the harmlessness concept is that harmless expressions should never fail to terminate and never err. This lemma proves that this purpose is fulfilled.

Lemma 4.37

$$\vdash \forall e,V,P,E : \left(\begin{array}{l} \texttt{vars\_in\_env}\, V\, E \Rightarrow \\ \texttt{harmless}\, V\, e \Rightarrow \\ \exists v : \forall N : \texttt{eval\_expr\_n}\, N\, e\, P\, E\, v \end{array} \right)$$

Proof: By structural induction on e. □

4.9 Strict Usage

It is well known that it is not necessary to require that the bound expression in a `let`-binding terminates in order to ensure that the unfolding preserves termination. An alternative is to make sure that substitution will place the bound expression in such a position or in such positions of the body expression that it is guaranteed to be evaluated. For example, it is safe to unfold the `let` in the following program even though `(g x)` might not always terminate or might sometimes produce run-time errors. It is the fact that `y` is used strictly in the expression `(y,y)` that makes the unfolding safe.

```
f x = let y=(g x) in (y,y) end;
g z = ...
```

Detecting strict usage is not computable, so we shall settle for a syntactic approximation of detecting when a variable is used strictly. Accordingly, we define `used_strictly` this way.

Definitional Theorem 4.38 (Strict Usage.)

$$\vdash \left\{ \begin{array}{l} \forall x, i : \texttt{used\_strictly}\, x\, (\texttt{E\_Int}\, i) = F \\ \forall x, o, e_1, e_2 : \texttt{used\_strictly}\, x\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \\ \quad \texttt{used\_strictly}\, x\, e_1 \vee \texttt{used\_strictly}\, x\, e_2 \\ \forall x, e_1, e_2 : \texttt{used\_strictly}\, x\, (\texttt{E\_Pair}\, e_1\, e_2) = \\ \quad \texttt{used\_strictly}\, x\, e_1 \vee \texttt{used\_strictly}\, x\, e_2 \\ \forall x, e : \texttt{used\_strictly}\, x\, (\texttt{E\_Fst}\, e) = \texttt{used\_strictly}\, x\, e \\ \forall x, e : \texttt{used\_strictly}\, x\, (\texttt{E\_Snd}\, e) = \texttt{used\_strictly}\, x\, e \\ \forall x, e : \texttt{used\_strictly}\, x\, (\texttt{E\_Inl}\, e) = \texttt{used\_strictly}\, x\, e \\ \forall x, e : \texttt{used\_strictly}\, x\, (\texttt{E\_Inr}\, e) = \texttt{used\_strictly}\, x\, e \\ \forall x, e_b, x_\ell, e_\ell, x_r, e_r : \texttt{used\_strictly}\, x\, (\texttt{E\_Case}\, e_b\, x_\ell\, e_\ell\, x_r\, e_r) = \\ \quad \texttt{used\_strictly}\, x\, e_b \vee \\ \quad (x \neq x_\ell) \wedge \texttt{used\_strictly}\, x\, e_\ell \wedge (x \neq x_r) \wedge \texttt{used\_strictly}\, x\, e_r \\ \forall x, x' : \texttt{used\_strictly}\, x\, (\texttt{E\_Var}\, x') = (x = x') \\ \forall x : \texttt{used\_strictly}\, x\, \texttt{E\_Error} = F \\ \forall x, f, e : \texttt{used\_strictly}\, x\, (\texttt{E\_Call}\, f\, e) = \texttt{used\_strictly}\, x\, e \\ \forall x, x', e_1, e_2 : \texttt{used\_strictly}\, x\, (\texttt{E\_Let}\, x'\, e_1\, e_2) = \\ \quad \texttt{used\_strictly}\, x\, e_1 \vee (x \neq x') \wedge \texttt{used\_strictly}\, x\, e_2 \end{array} \right\}$$

Three points are worth noting: (1) the `case`-construct will evaluate exactly one of its branches. Since we cannot determine which one we must therefore require both to use x strictly or else have the condition use x strictly before we say that the `case` uses x strictly. (2) bound variables are never used strictly so we test variable names when binding occurs. (3) Many, but not all, of the theorems involving strict usage would also work if we defined `E_Error` to use all variables strictly.

4.10 Evaluation Properties

The following lemma states that if an evaluation succeeds in an environment E then it will also succeed and with the same result in any other environment E' which has all bindings of E (but possibly more).

Lemma 4.39

$$\vdash \forall N, e, P, E, v, E' : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P\, E\, v \Rightarrow \\ \quad \forall x : \left(\begin{array}{l} (\texttt{lookup\_env}\, E\, x \neq \texttt{FAIL}) \Rightarrow \\ (\texttt{lookup\_env}\, E\, x = \texttt{lookup\_env}\, E'\, x) \end{array} \right) \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, e\, P\, E'\, v \end{array} \right)$$

Proof: By rule induction. □

The following technical lemma states that if the two front bindings in an environment bind different variables then the bindings can be swapped. This is quite hard to prove directly, i.e., without generalising it, because the induction cases for `let` and `case` add to the environment in front. The lemma is, however, easy to prove using the above lemma.

Lemma 4.40

$$\vdash \forall x_1, x_2 : \left(\begin{array}{l} (x_1 \neq x_2) \Rightarrow \\ \forall N, e, P, E, v, v_1, v_2 : \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, (\texttt{CONS}\, (x_1, v_1)\, (\texttt{CONS}\, (x_2, v_2)\, E))\, v \Leftrightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, (\texttt{CONS}\, (x_2, v_2)\, (\texttt{CONS}\, (x_1, v_1)\, E))\, v \end{array} \right)$$

Proof: By applying Lemma 4.39 twice. □

If the two front bindings do bind the same variable then the following lemma can be used to remove the second binding. Again, this follows easily from the general Lemma 4.39 but is hard to prove without generalising the theorem.

Lemma 4.41

$$\vdash \forall N, e, P, E, v, x, v_1, v_2 : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, E\, P\, (\texttt{CONS}\, (x, v_1)\, (\texttt{CONS}\, (x, v_2)\, E))\, v \\ \Leftrightarrow \\ \texttt{eval\_expr\_n}\, N\, E\, P\, (\texttt{CONS}\, (x, v_1)\, E)\, v \end{array} \right)$$

Proof: By applying Lemma 4.39 twice. □

The following lemma states that variables that are not free anywhere in an expression can be removed from the environment without affecting the evaluation of the expression.

Lemma 4.42

$$\vdash \forall e, x : \left(\begin{array}{l} \neg (\texttt{var\_free}\, x\, e) \Rightarrow \\ \forall N, P, E, v, v' : \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \Leftrightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, E\, v \end{array} \right)$$

Proof: By structural induction on e. Reductions with definitions and Lemma 4.41 leaves cases for `let` and `case` with bindings added to the front of the environment. These cases are handled by Lemma 4.40. □

A special case of a variable not being free in an expression is when the variable does not occur at all in the expression.

Lemma 4.43

$$\vdash \forall e, x : \left(\begin{array}{l} \neg(\texttt{var\_used}\, x\, e) \Rightarrow \\ \forall N, P, E, v, v' : \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \Leftrightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, E\, v \end{array} \right)$$

Proof: By Lemmas 4.28 and 4.42. □

One particular case where a variable will not occur in the expression is the case where it has been constructed for exactly that purpose by the `genvar` function.

Lemma 4.44

$$\vdash \forall v, e, P, E, v', N : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P\, E\, v' \Leftrightarrow \\ \texttt{eval\_expr\_n}\, N\, e\, P\, (\texttt{CONS}\, (\texttt{genvar}\, e, v)\, E)\, v' \end{array} \right)$$

Proof: By Lemma 4.43 and Theorem 4.30. □

We now turn to the effect of variable renaming on evaluation. Our first lemma states that the effect of using `rename_free` on an expression is identical to renaming in the environment. Note that we require that the renamed-to variable does not occur in the original expression; this is because `rename_free` does not take variable capture into account.

Lemma 4.45 .

$$\vdash \forall e, x' : \left(\begin{array}{l} \neg(\texttt{var\_used}\, x'\, e) \Rightarrow \\ \forall N, P, E, v, x, v' : \\ \quad \texttt{eval\_expr\_n}\, N\, (\texttt{rename\_free}\, x\, x'\, e)\, P\, (\texttt{CONS}\, (x', v')\, E)\, v \Leftrightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \end{array} \right)$$

Proof: By structural induction on the expression. The non-trivial cases are the cases for `let` and `case` which add to the environment in front. These cases are easily handled by Lemmas 4.41, 4.40, and 4.43. □

Changing the names of free variables does influence evaluation, as we just saw. But changing the names of bound variables does not:

Lemma 4.46

$$\vdash \forall e, N, e', P, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, (\texttt{rename\_bound}\, e'\, e)\, P\, E\, v \Leftrightarrow \\ \texttt{eval\_expr\_n}\, N\, e\, P\, E\, v \end{array} \right)$$

Proof: By structural induction on e. Cases for `let` and `case` modify the environment and are handled by Lemma 4.45. □

Combining the previous two lemmas we get the following lemma, which turns out to be useful in connection with substitution and alpha conversion.

Lemma 4.47

$$\vdash \forall e, x' : \left(\begin{array}{l} \neg(\texttt{var\_used}\, x'\, e) \Rightarrow \\ \forall N, P, E, v, x, v' : \\ \quad \texttt{eval\_expr\_n}\, N \\ \quad\quad (\texttt{rename\_free}\, x x' (\texttt{rename\_bound} (\texttt{E\_Var}\, x')\, e) \\ \quad\quad P\, (\texttt{CONS}\, (x', v')\, E)\, v \Leftrightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \end{array} \right)$$

Proof: By Lemmas 4.45 and 4.46. □

Values and expressions are two different entities even if they have members that are written the same way like `4` and `L 0`. The following function and lemma shows that the set of values can be embedded within the set of expressions. This is useful for the constant folding transformation which will need to place its result in the target program in the form of an expression.

Definitional Theorem 4.48

$$\vdash \left\{ \begin{array}{l} \forall i : \texttt{value2expr}\, (\texttt{V\_Int}\, i) = \texttt{E\_Int}\, i \\ \forall v_1, v_2 : \texttt{value2expr}\, (\texttt{V\_Pair}\, v_1\, v_2) = \\ \quad \texttt{E\_Pair}\, (\texttt{value2expr}\, v_1)\, (\texttt{value2expr}\, v_2) \\ \forall v : \texttt{value2expr}\, (\texttt{V\_Inl}\, v) = \texttt{E\_Inl}\, (\texttt{value2expr}\, v) \\ \forall v : \texttt{value2expr}\, (\texttt{V\_Inr}\, v) = \texttt{E\_Inr}\, (\texttt{value2expr}\, v) \end{array} \right\}$$

Lemma 4.49

$$\vdash \forall v, N, P, E, v' : \texttt{eval\_expr\_n}\, N\, (\texttt{value2expr}\, v)\, P\, E\, v' \Leftrightarrow (v = v')$$

Proof: By structural induction on the value. □

We shall only need the corresponding projection in the case of integers so we therefore just define:

Definitional Theorem 4.50

$$\vdash \forall i : \texttt{expr2int}\, (\texttt{E\_Int}\, i) = i$$

The injection of values into expressions has a number of trivial nice properties such as the following two which we shall need later.

Lemma 4.51 $\vdash \forall v, V : \texttt{check\_vars}\, V\, (\texttt{value2expr}\, v)$

Proof: By structural induction on v. □

Lemma 4.52 $\vdash \forall v, F : \texttt{check\_funcs}\, F\, (\texttt{value2expr}\, v)$

Proof: By structural induction on v. □

4.11 Semantic Equivalence

We now define the notion of two expressions being equivalent evaluation-wise. Intuitively this should mean that they when evaluated in identical environments always produce indentical results. That, however, is to strong a requirement.

Consider the expressions `(1+2)` and `fst (3,x)`. We would like these two expressions to be equivalent, but they only share evaluation properties if we can guarantee that `x` is bound.

Consider now the expressions `(f 1)` and `(g 2)`. These two expressions might be equivalent if the program in which we evaluate them contains the definition `g x = f (x - 1);`. In another program they might not be equivalent.

These concerns guide us to the following definition of two expressions being *semantically equivalent* relative to a program and to a set of variable.

Definitional Theorem 4.53 (Sematical Equivalence.)

$$\vdash \forall e_1, e_2, P, V : \left(\begin{array}{l} \texttt{semeq}\, V\, P\, e_1\, e_2 \Leftrightarrow \\ \quad \forall E, v : \texttt{vars\_in\_env}\, V\, E \Rightarrow \\ \quad\quad (\texttt{eval\_expr}\, e_1\, P\, E\, v \Leftrightarrow \texttt{eval\_expr}\, e_2\, P\, E\, v) \end{array} \right)$$

This definition plays an important rîle in the Section 4.12.

The definition of `semeq` is ignorant of timing issues; two expressions are equivalent if they behave identically to `eval_expr` even if they behave quite differently to `eval_expr_n`. A corresponding definition for the timed case, called two-way local improvement, will be discussed shortly.

4.12 Improvement

This section introduces the concept of improvement of expressions, the notion that two expressions might be equivalent but that one requires less resources than the other. Improvement will be generalised to programs, where it will be used to mean that P_2 is an improvement over P_1 if all P_2's function bodies are improvements over the corresponding bodies in P_1. (As it will be discussed further in Section 9.3, many of the ideas here originate with [San96].)

Improvement as discussed in this context is not a question of efficiency. Moreover, as the improvement discussed here will be based on the required function-call nesting to complete an evaluation, there is no direct relation to any stop-watch type of improvement in efficiency. The real reason that we shall be concerned with improvement is that the concept plays an important rôle in guaranteeing the correctness of a folding transformation.

The property that one expression improves another is called *local improvement.* Just like sematical equivalence we need to make our definition of local improvement relative to a set of variable and a program: expressions can contain function calls, so the definition should be relative to some program, and the expressions can contain free variables so there should also be a restriction on the environments for which improvement is considered. That leads us to the following definition of e_1 being improved by e_2:

Definitional Theorem 4.54 (Local Improvement.)

$$\vdash \forall e_1, e_2, P, V : \left(\begin{array}{l} \texttt{local\_improvement}\ V\ P\ e_1\ e_2 \Leftrightarrow \\ \quad \forall E, v, N : \texttt{vars\_in\_env}\ V\ E \Rightarrow \\ \quad\quad (\texttt{eval\_expr}\ e_2\ P\ E\ v \Rightarrow \texttt{eval\_expr}\ e_1\ P\ E\ v) \wedge \\ \quad\quad (\texttt{eval\_expr\_n}\ N\ e_1\ P\ E\ v \Rightarrow \texttt{eval\_expr\_n}\ N\ e_2\ P\ E\ v) \end{array} \right)$$

The left conjunct tells us that any evaluation of e_2 can also be done with e_1. The right conjunct tells us that not only can any evaluation of e_1 can also be done with e_2 — doing it with e_2 will be faster (in the usual special meaning of "faster"). The two clauses taken together means in particular that e_1 and e_2 have the same semantics.

Local improvement is a stonger condition than equivalence as the following lemma shows:

Lemma 4.55 (Local Improvement Implies Equivalence.)

$$\vdash \forall e_1, e_2, V, P : \texttt{local\_improvement}\ V\ P\ e_1\ e_2 \Rightarrow \texttt{semeq}\ V\ P\ e_1\ e_2$$

Proof: By rewriting with the defintions of `local_improvement` and `semeq`, then using Lemmas 4.12 and 4.15. □

Assume that we have a series of programs, $P_1, \ldots, P_n$, obtained by doing a series of program transformations, and that the function name f occurs in all the programs and with the same semantics. If we would like to do folding of some expression e in P_n with a function definition, f_k = e; in P_1, we should have some means of comparing the cost of evaluating e to the cost of evaluating a call to f *with respect to* P_n.

For this purpose we introduce the concept of improving a program defined as improving (in the above reflexive way) *all* its functions while possibly adding more functions. We will not consider removal and renaming of functions because removal and renaming would make the relation non-transitive. Furthermore we insist that the programs' main functions have the same name since the main funcions define the meaning of the programs.

Definitional Theorem 4.56 (Improvement.)

$$\vdash \forall P_1, P_2 : \left(\begin{array}{l} \texttt{improvement}\, P_1\, P_2 \Leftrightarrow \\ \left(\begin{array}{l} (\texttt{FST}\,(\texttt{HD}\, P_1) = \texttt{FST}\,(\texttt{HD}\, P_2)) \wedge \\ \forall f, x_1, e_1 : (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x_1, e_1)) \Rightarrow \\ \quad \exists x_2, e_2 : \forall v, v', N : \\ \quad\quad (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x_2, e_2)) \wedge \\ \quad\quad \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_1\, P_1\, [(x_1, v)]\, v' \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, e_2\, P_2\, [(x_2, v)]\, v' \end{array} \right) \wedge \\ \quad\quad \left(\begin{array}{l} \texttt{eval\_expr}\, e_2\, P_2\, [(x_2, v)]\, v' \Rightarrow \\ \texttt{eval\_expr}\, e_1\, P_1\, [(x_1, v)]\, v' \end{array} \right) \end{array} \right) \end{array} \right)$$

Recall that the representation of programs is such that $\texttt{FST}\,(\texttt{HD}\, P)$ is P's name. This explains the first equality. The rest of the definition says that functions defined in P_1 are also defined in P_2 and that P_2's definition is an improvement over P_1's in top-level environments.

Improvement has the important property that it implies equivalence of programs. (Due to the way we have defined programs we must also require that the programs be non-empty.)

Lemma 4.57 (Improvement Meaning Lemma.)

$$\vdash \forall P_1, P_2, v_{in}, v_{out} : \left(\begin{array}{l} (P_1 \neq [\,]) \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \Rightarrow \\ (\texttt{eval\_prg}\, P_1\, v_{in}\, v_{out} \Leftrightarrow \texttt{eval\_prg}\, P_2\, v_{in}\, v_{out}) \end{array} \right)$$

Proof: By the definitions of `improvement` and `eval_prg` together with Lemmas 4.12 and 4.15. □

Both local and global improvement are reflexive and transitive. This can be useful in combining program transformations.

Lemma 4.58 (Local Improvement is Reflexive.)

$$\vdash \forall e, V, P : \texttt{local\_improvement}\, V\, P\, e\, e$$

Proof: Trivial: rewrite with definition of `local_improvement` and use reflexivity of equality. □

Lemma 4.59 (Improvement is Reflexive.)

$$\vdash \forall P : \texttt{improvement}\, P\, P$$

Proof: By rewriting with the definition of `improvement`. □

Lemma 4.60 (Local Improvement is Transitive.)

$$\vdash \forall e_1, e_2, e_3, V, P : \left(\begin{array}{l} \texttt{local\_improvement}\, V\, P\, e_1\, e_2 \Rightarrow \\ \texttt{local\_improvement}\, V\, P\, e_2\, e_3 \Rightarrow \\ \texttt{local\_improvement}\, V\, P\, e_1\, e_3 \end{array} \right)$$

Proof: By rewriting with the definition of `local_improvement` and repeated use of modus ponens. □

Note that we use just one variable set V because this is the situation where we will use it. The theorem is believed to hold even with different V_1, V_2, and V_3 as long as the latter is a superset of the two former.

Lemma 4.61 (Improvement is Transitive.)

$$\vdash \forall P_1, P_2, P_3 : \left(\begin{array}{l} \texttt{improvement}\, P_1\, P_2 \Rightarrow \\ \texttt{improvement}\, P_2\, P_3 \Rightarrow \\ \texttt{improvement}\, P_1\, P_3 \end{array} \right)$$

Proof: By rewriting with the definition of `improvement` and repeated use of modus ponens. □

The following important lemma provides us with a simple characterization of what it means for two expressions to be local improvements of each other.

Lemma 4.62 (Two-Way Local Improvement Lemma.)

$$\vdash \forall P, e_1, e_2, V : \left(\begin{array}{l} \left(\begin{array}{l} \texttt{local\_improvement}\, V\, P\, e_1\, e_2 \wedge \\ \texttt{local\_improvement}\, V\, P\, e_2\, e_1 \end{array} \right) \\ \Leftrightarrow \\ \forall E, v, N : \texttt{vars\_in\_env}\, V\, E \Rightarrow \\ \qquad \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v \Leftrightarrow \\ \texttt{eval\_expr\_n}\, N\, e_2\, P\, E\, v \end{array} \right) \end{array} \right)$$

Proof: By rewriting with the definition of `local_improvement` and Lemmas 4.12 and 4.15. □

Note, that the lemma shows that two-way local improvement is to `eval_expr_n` as semantical equivalence is to `eval_expr`.

4.12.1 Improvement Globalisation

In the following we shall work through a number of lemmas leading up to two of the major results, namely that local improvement implies global improvement and that two-way local improvement implies two-way global improvement.

All these lemmas consider two expressions, e_a and e_b, where the latter is a local improvement over the first. The lemmas state what happens with respect to evaluation when we replace e_a by e_b in an expression or a program. Under sufficiently strong side conditions we shall see that little happens. Some of the theorem depend only on the weaker semantical equivalence condition and will be stated using that; the corresponding theorem with local improvement follows immediately from Lemma 4.55.

Our first lemma deals with replacing e_a by e_b in an expression under the assumption that the resultant expression evaluates. We claim that the original expression then also evaluates and with the same result and that this only depends of equivalence, not on improvement:

Lemma 4.63

$$\vdash \forall e_2, P, E, v : \left(\begin{array}{l} \texttt{eval\_expr}\, e_2\, P\, E\, v \Rightarrow \\ \forall e_1, e_a, e_b, V_1, V_2 : \\ \qquad \texttt{semeq}\, V_1\, P\, e_a\, e_b \Rightarrow \\ \qquad \texttt{repl\_expr}\, V_2\, e_1\, e_2\, e_a\, e_b \Rightarrow \\ \qquad \texttt{vars\_in\_env}\,(\texttt{SETMINUS}\, V_1\, V_2)\, E \Rightarrow \\ \qquad \texttt{eval\_expr}\, e_1\, P\, E\, v \end{array} \right)$$

Proof: By strong rule induction (as defined by Equation 3.18, which see). □

(There is a lot of fiddling with variables required for this and the following lemmas but no fundamental difficulties are involved here.)

Proving the opposite implication, i.e., that if the original expression evaluates then so does the result of replacing, is harder since that really does depend on the improvement. We will prove it using the following technical lemma which looks the ways it does in order to allow a nested induction.

Lemma 4.64

$$\vdash \forall M, N, e_1, P, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v \Rightarrow \\ \forall e_2, e_a, e_b, V_1, V_2 : \\ \quad \texttt{local\_improvement}\, V_1\, P\, e_a\, e_b \Rightarrow \\ \quad (N < M) \Rightarrow \\ \quad \texttt{repl\_expr}\, V_2\, e_1\, e_2\, e_a\, e_b \Rightarrow \\ \quad \texttt{vars\_in\_env}\,(\texttt{SETMINUS}\, V_1\, V_2)\, E \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e_2\, P\, E\, v \end{array} \right)$$

Proof: Numerical induction on M produces two cases. The base case, 0, is trivial since $(N < 0)$ is always false for natural numbers. This leaves us with the induction step, $\texttt{SUC}\, M'$.

We now do strong rule induction. The rule induction hypotheses are strong enough to handle all but one of the resulting subcases. The left-over case, the recursive function call, follows immediately from the numerical induction hypothesis. □

As mentioned we are not really interested in Lemma 4.64 but instead in the following special case.

Lemma 4.65

$$\vdash \forall N, e_1, P, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v \Rightarrow \\ \forall e_2, e_a, e_b, V_1, V_2 : \\ \quad \texttt{local\_improvement}\, V_1\, P\, e_a\, e_b \Rightarrow \\ \quad \texttt{repl\_expr}\, V_2\, e_1\, e_2\, e_a\, e_b \Rightarrow \\ \quad \texttt{vars\_in\_env}\,(\texttt{SETMINUS}\, V_1\, V_2)\, E \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e_2\, P\, E\, v \end{array} \right)$$

Proof: Specialise Lemma 4.64 to $\texttt{SUC}\, N$ and N. □

If e_a and e_b are both improvements over each other, then Lemma 4.63 does not quite fit our needs as information about evaluation depth is lost with the use of `eval_expr` instead of `eval_expr_n`. Therefore we need the following lemma.

Lemma 4.66

$$\vdash \forall N, e_2, P, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_2\, P\, E\, v \Rightarrow \\ \forall e_1, e_a, e_b, V_1, V_2 : \\ \quad \texttt{local\_improvement}\, V_1\, P\, e_b\, e_a \Rightarrow \\ \quad \texttt{repl\_expr}\, V_2\, e_1\, e_2\, e_a\, e_b \Rightarrow \\ \quad \texttt{vars\_in\_env}\,(\texttt{SETMINUS}\, V_1\, V_2)\, E \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v \end{array} \right)$$

Proof: By strong rule induction. □

Note that the order of e_a and e_b in the improvement condition is reversed. Note also that we left out Lemma 4.63's precondition $\texttt{semeq}\, V_1\, P\, e_a\, e_b$ as it is redundant.

We now turn to see what happens if we fix the expression and do the replacement in the program instead.

Lemma 4.67

$$\vdash \forall e, P_2, E, v : \left(\begin{array}{l} \texttt{eval\_expr}\, e\, P_2\, E\, v \Rightarrow \\ \forall P_1, e_a, e_b, V : \\ \quad \texttt{semeq}\, V\, P\, e_a\, e_b \Rightarrow \\ \quad \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Rightarrow \\ \quad \texttt{eval\_expr}\, e\, P_1\, E\, v \end{array} \right)$$

Proof: Strong rule induction, followed by the usual reductions, leaves one unsolved case, namely the $\texttt{E\_Call}\ f\ e$ case. The rule induction hypothesis cannot be used for the evaluation of f's body since two different versions are extracted from P_1 and P_2. But by using Lemma 4.24 we see that the body extracted from P_2 is an improvement over the one extracted from P_1. This reduces to a special case of Lemma 4.63. □

The opposite direction, i.e., assuming that e evaluates in the program before replacement produces the same problem with the rule induction hypothesis being too weak by itself as we saw for replacement in expressions. The solution is again a technical lemma.

Lemma 4.68

$$\vdash \forall M, N, e, P_1, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P_1\, E\, v \Rightarrow \\ (N < M) \Rightarrow \\ \forall P_2, e_a, e_b, V : \\ \quad \texttt{local\_improvement}\, V\, P\, e_a\, e_b \Rightarrow \\ \quad \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P_2\, E\, v \end{array} \right)$$

Proof: Numerical induction on M produces two cases. The base case, 0, is trivial since again $(N \not< 0)$. This leaves us with the induction step, $\texttt{SUC}\, M'$.

We now do strong rule induction. Just like the previous lemma the rule induction hypothesis is strong enough to handle all cases but the $\texttt{E\_Call}\ f\ e$ case which the rule induction hypothesis cannot handle. This case, however, is caught by the numerical induction hypothesis together with Lemmas 4.65 and 4.24. □

Just like it was the case for the expression replacement case we are not really interested in Lemma 4.68 but instead in the following special case.

Lemma 4.69

$$\vdash \forall N, e, P_1, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P_1\, E\, v \Rightarrow \\ \forall P_2, e_a, e_b, V : \\ \quad \texttt{local\_improvement}\, V\, P\, e_a\, e_b \Rightarrow \\ \quad \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P_2\, E\, v \end{array} \right)$$

Proof: Specialise Lemma 4.68 to $\texttt{SUC}\, N$ and N. □

Lemma 4.67 also has a two-way version. Note that the order of e_a and e_b in the improvement condition is reversed.

Lemma 4.70

$$\vdash \forall N, e, P_2, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P_2\, E\, v \Rightarrow \\ \forall P_1, e_a, e_b, V : \\ \quad \texttt{local\_improvement}\, V\, P\, e_b\, e_a \Rightarrow \\ \quad \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P_1\, E\, v \end{array} \right)$$

Proof: Strong rule induction, followed by the usual reductions, leaves one unsolved case, namely the $\texttt{E\_Call}\ f\ e$ case. The rule induction hypothesis cannot be used for the evaluation of f's body since two different versions are extracted from P_1 and P_2. But by using Lemma 4.24 we see that the body extracted from P_2 is an improvement over the one extracted from P_1. This reduces to a special case of Lemma 4.66. □

We now combine the previous lemmas into two very important theorems. These tell us that improvement of a subexpression causes improvement of the program in which replacement is done.

Theorem 4.71 (Improvement Globalisation Theorem.)

$$\vdash \forall P_1, P_2, e_a, e_b, V : \left(\begin{array}{l} \texttt{local\_improvement}\, V\, P_1\, e_a\, e_b \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \end{array} \right)$$

Proof: Only the two inner conjuncts in the definition of the improvement predicate pose any problems. The `eval_expr_n` conjunct follows from Lemmas 4.65 and 4.69. The `eval_expr` conjunct follows from Lemmas 4.63 and 4.67. □

Theorem 4.72 (Two-Way Improvement Globalisation Theorem.)

$$\vdash \forall P_1, P_2, e_a, e_b, V : \left(\begin{array}{l} \texttt{local\_improvement}\, V\, P_1\, e_a\, e_b \Rightarrow \\ \texttt{local\_improvement}\, V\, P_1\, e_b\, e_a \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Rightarrow \\ (\texttt{improvement}\, P_1\, P_2 \wedge \texttt{improvement}\, P_2\, P_1) \end{array} \right)$$

Proof: The first conjunct follows directly from Theorem 4.71. The second conjunct follows from Lemmas 4.65, 4.66, 4.69, and 4.70. □

4.12.2 Improvement with Fewer Functions

The improvement predicate for programs implies that if P_2 is an improvement over P_1 then P_2 defines at least the same functions as P_1. For most of the program transformations we shall consider this is the case, but not for all. We need a similar condition for the situation when functions are eliminated and we therefore define the concept of reverse improvement. This predicate is not supposed to be used alone and therefore does not contain, for example, a condition on what is the first function.

Definitional Theorem 4.73 (Reverse Improvement.)

$$\vdash \forall P_1, P_2 : \left(\begin{array}{l} \texttt{improvementR}\, P_1\, P_2 \Leftrightarrow \\ \forall f, x_1, e_1, x_2, e_2 : \\ \qquad \left(\begin{array}{l} (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x_1, e_1)) \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x_2, e_2)) \Rightarrow \\ \forall v, v', n : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_2\, P_2\, [(x_2, v)]\, v' \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, e_1\, P_1\, [(x_1, v)]\, v' \end{array} \right) \end{array} \right) \end{array} \right)$$

When `improvementR` is to be used it will be in connection with `improvement`. Under this condition `improvementR` is transitive.

Lemma 4.74 (Reverse Improvement is Transitive.)

$$\vdash \forall P_1, P_2, P_3 : \left(\begin{array}{l} \texttt{improvement}\ P_1\ P_2 \Rightarrow \\ \texttt{improvement}\ P_2\ P_3 \Rightarrow \\ \texttt{improvementR}\ P_1\ P_2 \Rightarrow \\ \texttt{improvementR}\ P_2\ P_3 \Rightarrow \\ \texttt{improvementR}\ P_1\ P_3 \end{array} \right)$$

Proof: By rewriting with the definitions of `improvement` and `improvementR` and repeated use of modus ponens. □

Two-way improvement implies reverse improvement. This is useful because most of our program transformation produce two-way improvement. Putting the pieces together requires improvement and reverse improvement.

Lemma 4.75

$$\vdash \forall P_1, P_2 : \left(\begin{array}{l} \texttt{improvement}\ P_1\ P_2 \Rightarrow \\ \texttt{improvement}\ P_2\ P_1 \Rightarrow \\ \texttt{improvementR}\ P_1\ P_2 \end{array} \right)$$

Proof: By rewriting with the definitions of `improvement` and `improvementR` and repeated use of modus ponens. □

As mentioned above, reverse improvement is to be used in connection with improvement. When two programs are in an `improvement`/`improvementR` relationship expressions are evaluated in the same way as we shall see in Lemma 4.78. First a few lemmas to help proving that, though.

It is a consequence of improvement that the second program contains definition for all functions defined in the first program. This makes it fairly easy to prove that evaluations in the first program work the same way in the second program.

Lemma 4.76

$$\vdash \forall P_1, e, N, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\ N\ e\ P_1\ E\ v \Rightarrow \\ \forall P_2 : \texttt{improvement}\ P_1\ P_2 \Rightarrow \texttt{eval\_expr\_n}\ N\ e\ P_2\ E\ v \end{array} \right)$$

Proof: By strong rule induction. □

For the other direction we must first make sure that the expression we are considering does not use functions that are not defined in the first program.

Lemma 4.77

$$\vdash \forall P_2, e, N, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P_2\, E\, v \Rightarrow \\ \forall P_1 : \left(\begin{array}{l} \texttt{check\_funcs}\,(\texttt{functions}\, P_1)\, e \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \Rightarrow \\ \texttt{improvementR}\, P_1\, P_2 \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, e\, P_1\, E\, v \end{array} \right) \end{array} \right)$$

Proof: By strong rule induction. □

Lemma 4.78

$$\vdash \forall P_1, P_2, e, N, E, v : \left(\begin{array}{l} \texttt{check\_funcs}\,(\texttt{functions}\, P_1)\, e \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \Rightarrow \\ \texttt{improvementR}\, P_1\, P_2 \Rightarrow \\ (\texttt{eval\_expr\_n}\, N\, e\, P_1\, E\, v \Leftrightarrow \texttt{eval\_expr\_n}\, N\, e\, P_2\, E\, v) \end{array} \right)$$

Proof: "⇒" by Lemma 4.76; "⇐" by Lemma 4.77. □

4.13 Identity Functions

Identity functions play an important rôle in the programs transformations in Chapter 6. We therefore present some utility functions and lemmas.

The utility function, `id_func`, checks whether a given function name in a given program refers to a syntactic identity function, i.e., a function whose body consists only of a reference to the formal parameter.

Definitional Theorem 4.79

$$\vdash \forall P, f : \texttt{id\_func}\, P\, f \Leftrightarrow \exists x : \texttt{lookup\_func}\, P\, f = \texttt{OK}\,(x, \texttt{E\_Var}\, x)$$

The purpose of identity functions is to serve as marks for a level of call depth. This is stated in the following lemma.

Lemma 4.80 (Identity Functions Consume One Time-Step.)

$$\vdash \forall N, e, P, E, v, I : \left(\begin{array}{l} \texttt{id\_func}\, P\, I \Rightarrow \\ \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, (\texttt{E\_Call}\, I\, e)\, P\, E\, v \Leftrightarrow \\ (N \neq 0) \wedge \texttt{eval\_expr\_n}(\texttt{PRE}\, N)\, e\, P\, E\, v \end{array} \right) \end{array} \right)$$

Proof: Trivial. □

For completeness we also prove that identity functions do not influence untimed evaluation.

Lemma 4.81 (Identity Functions do not Influence Evaluation.)

$$\vdash \forall e, P, E, v, I : \left(\begin{array}{l} \texttt{id\_func}\, P\, I \Rightarrow \\ (\texttt{eval\_expr}\,(\texttt{E\_Call}\, I\, e)\, P\, E\, v \Leftrightarrow \texttt{eval\_expr}\, e\, P\, E\, v) \end{array} \right)$$

Proof: Utterly trivial. □

Generally, identity functions are still identity functions after some replacement has taken place in a program.

Lemma 4.82 (Identity Function Replacement Lemma.)

$$\vdash \forall P_1, P_2, I, V, e_a, e_b : \left(\begin{array}{l} \texttt{id\_func}\, P_1\, I \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, e_a\, e_b \Rightarrow \\ (\forall x : e_a \neq \texttt{E\_Var}\, x) \Rightarrow \\ \texttt{id\_func}\, P_2\, I \end{array} \right)$$

Proof: By rewriting with the definition of `id_func` and using Lemma 4.24. □

4.14 Functions

In order to describe the preconditions needed to ensure that replacement preserves static correctness, we first need the following predicate that tests whether a function is called somewhere in an expression.

Definitional Theorem 4.83 (Test if Function is Used.)

$$\vdash \left\{ \begin{array}{l} \forall f, i : \texttt{func\_called}\, f\, (\texttt{E\_Int}\, i) = F \\ \forall f, o, e_1, e_2 : \texttt{func\_called}\, f\, (\texttt{E\_Op}\, o\, e_1\, e_2) = \\ \qquad \texttt{func\_called}\, f\, e_1 \vee \texttt{func\_called}\, f\, e_2 \\ \qquad\qquad \vdots \\ \forall f, f', e, : \texttt{func\_called}\, f\, (\texttt{E\_Call}\, f'\, e) = \\ \qquad (f' = f) \vee \texttt{func\_called}\, f\, e \\ \forall f, x, e_1, e_2 : \texttt{func\_called}\, f\, (\texttt{E\_Let}\, x\, e_1\, e_2) = \\ \qquad \texttt{func\_called}\, f\, e_1 \vee \texttt{func\_called}\, f\, e_2 \end{array} \right\}$$

We shall need a few lemmas concerning functions in the concatenation of two programs. Firstly, the definition of `functions` satisfies the following lemma about the set of function names in the concatenation of two programs.

Lemma 4.84

$$\vdash \forall P_1, P_2 : \texttt{functions}\,(\texttt{APPEND}\,P_1\,P_2) = \texttt{APPEND}\,(\texttt{functions}\,P_1)\,(\texttt{functions}\,P_2)$$

Proof: By list induction on P_1. □

Secondly, the check for rebindings of functions in a program is indifferent to the order of bindings.

Lemma 4.85

$$\vdash \forall P_1, P_2 : \texttt{no\_rebinds}\,(\texttt{APPEND}\,P_1\,P_2) \Leftrightarrow \texttt{no\_rebinds}\,(\texttt{APPEND}\,P_2\,P_1)$$

Proof: By list induction on P_1. □

Thirdly, in a program binding more functions it is easier to satisfy the name usage requirements as the following two simple lemmas show.

Lemma 4.86

$$\vdash \forall e, F_1, F_2 : \left(\begin{array}{l} (\forall f : \texttt{MEM}\ f\,F_1 \Rightarrow \texttt{MEM}\ f\,F_2) \Rightarrow \\ \texttt{check\_funcs}\,F_1\,e \Rightarrow \\ \texttt{check\_funcs}\,F_2\,e \end{array} \right)$$

Proof: By structural induction on e. □

Lemma 4.87

$$\vdash \forall b, F_1, F_2 : \left(\begin{array}{l} (\forall f : \texttt{MEM}\ f\,F_1 \Rightarrow \texttt{MEM}\ f\,F_2) \Rightarrow \\ \texttt{check\_names}\,F_1\,b \Rightarrow \\ \texttt{check\_names}\,F_2\,b \end{array} \right)$$

Proof: By Lemma 4.86. □

Just as for variables we have a function that generates new function names.

Definitional Theorem 4.88

$$\vdash \left\{ \begin{array}{l} (\texttt{genfunc}\,[\,] = 0) \\ \forall b, P : \texttt{genfunc}\,(\texttt{CONS}\,b\,P) = \texttt{MAX}\,(\texttt{SUC}\,(\texttt{FST}\,b))\,(\texttt{genfunc}\,P) \end{array} \right\}$$

The following lemma states that `genfunc` works as intended.

Lemma 4.89 (genfunc property.)

$$\vdash \forall P : \neg(\texttt{MEM}\,(\texttt{genfunc}\,P)\,(\texttt{functions}\,P))$$

Proof: By specialising the following lemma to $(\texttt{genfunc}\,P)$ for f. □

Lemma 4.90

$$\vdash \forall P, f : (f \geq \texttt{genfunc}\,P) \Rightarrow \neg(\texttt{MEM}\,f\,(\texttt{functions}\,P))$$

Proof: By list induction on P. □

4.15 Static Correctness

This section describes a theorem stating under what circumstances replacement in a statically correct program produces another statically correct program. Furthermore we show two lemmas that tell us that functions looked up in a statically correct program use at most one binding in the environment, namely that of their formal parameter.

The theorem states that static correctness is preserved as long as the target expression is "nicer" than the source expressions, i.e., that it has no extra free variables and any extra functions that it calls are known to be defined.

Theorem 4.91

$$\vdash \forall P_1, P_2, V, e_a, e_b : \left(\begin{array}{l} \texttt{stat\_ok}\,P_1 \Rightarrow \\ \texttt{repl\_prg}\,V\,P_1\,P_2\,e_a\,e_b \Rightarrow \\ (\forall x : \texttt{var\_free}\,x\,e_b \Rightarrow \texttt{var\_free}\,x\,e_a) \Rightarrow \\ \forall f : \left(\begin{array}{l} \texttt{func\_called}\,f\,e_b \Rightarrow \\ \texttt{func\_called}\,f\,e_a \vee \texttt{MEM}\,f\,(\texttt{functions}\,P_1) \end{array} \right) \Rightarrow \\ \texttt{stat\_ok}\,P_2 \end{array} \right)$$

Proof: The proof for this theorem is long, quite technical, but does not require special insight. It is therefore omitted. □

When we look up a function in a statically correct program that function is well-behaved, i.e., it uses only the allowed variables and functions.

Lemma 4.92

$$\vdash \forall P, f, x, e : \left(\begin{array}{l} \texttt{stat\_ok}\, P \Rightarrow \\ (\texttt{lookup\_func}\, P\, f = \texttt{OK}\,(x,e)) \Rightarrow \\ \texttt{check\_names}\,(\texttt{functions}\, P)\,(f,x,e) \end{array} \right)$$

Proof: Simple list induction does not work here, as the set of statically correct programs is not closed under the `TL` function. Instead we rewrite with respect to the definition of `stat_ok` and generalise over $(\texttt{functions}\, P)$. Then list induction on P solves the problem. □

We now use the previous lemma to prove what happens when a function looked up in a statically correct program is evaluated. We find, not surprisingly, that the only binding that matters in the environment is one binding the formal parameter.

Lemma 4.93

$$\vdash \forall P, f, x, e : \left(\begin{array}{l} \texttt{stat\_ok}\, P \Rightarrow \\ (\texttt{lookup\_func}\, P\, f = \texttt{OK}\,(x,e)) \Rightarrow \\ \forall E, N, v, v' : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P\,(\texttt{CONS}\,(x,v')\, E)\, v \Leftrightarrow \\ \texttt{eval\_expr\_n}\, N\, e\, P\,[(x,v')]\, v \end{array} \right) \end{array} \right)$$

Proof: By Lemmas 4.92 and 4.39. □

Chapter 5

Self-Interpretation

Never express yourself more clearly
than you are able to think.
— NIELS BOHR, 1885–1962

5.1 The Importance of Self-Interpretation

This section exhibits a *self-interpreter*, i.e., an interpreter that interprets the language it itself is written in. The reason for considering such an interpreter were discussed in Section 1.4.2.

The notion of correctness for an interpreter that works with encoded data is considered and the concrete self-interpreter is proven correct according to this notion. The notion of correctness also considers non-termination and run-time errors.

5.2 The Concrete Self-Interpreter

The concrete self-interpreter for the PEL language introduced in Chapter 3 to be presented in this section is coded in the same style as a typical interpreter would be coded in a statically typed language. As the language it interprets (and thus the language that it is written in) is dynamically typed, the coding style might seem peculiar, but we have our reasons:

- Although the language is dynamically typed it does not contain a means of inspecting a value of unknown type. Thus at least the syntax of the program being interpreted has to be coded in a way that allows this.

- Part of the goal of studying the self-interpreter is to say something about partial evaluation of typed languages. The self-interpreter is coded in a way that would make it well-typed in a language equipped with a suitable type discipline.

The following sections discuss the coding issues and the program text of the interpreter.

5.2.1 Encoding of Syntax

The encoding of a program's syntax as values can be done in many ways. But because the program argument to an interpreter is the one we want to be static (i.e., completely known), there appears to be no benefit for partial evaluation or for ease of correctness proofs in using one encoding over another, subject to the constraint that the decoding obviously should be computable. (It should not be hard to deduce computability of the decoding from existence of an interpreter using that encoding.) Different encodings may have different efficiency characteristics, but that is not an issue here.

In the following we shall see how the self-interpreter to be presented in Section 5.2.5 encodes the syntax of the programs it interprets.

In the formalisation we shall arrange the situation such that all function names and all variable names are just integers, which conveniently will be represented as such. This leaves the encoding of program structure, expressions, and operators.

Encoding of Programs

Programs are essentially lists of function definitions and therefore coded as such. More precisely, for the purpose of this definition extending programs to include the empty program:

$$\begin{aligned} \mathcal{C}_{\mathrm{prg}}(\varepsilon) &= \texttt{L 0} \\ \mathcal{C}_{\mathrm{prg}}(f x\texttt{=}e\texttt{;}p) &= \texttt{R ((}f\texttt{,(}x\texttt{,}\mathcal{C}_{\mathrm{exp}}(e)\texttt{))),}\mathcal{C}_{\mathrm{prg}}(p)\texttt{)} \end{aligned}$$

The function $\mathcal{C}_{\mathrm{prg}}$ — defined on PEL expression and producing PEL values — is realised by the HOL-function `code_prg`. Note, that `L` and `R` as used here are constructors of PEL's value type and should not be confused with the contructors of HOL's built-in sum type (which is not explicitly used in this thesis).

Encoding of Expressions

There are twelve different syntactic expression constructs, so the encoding of expressions is essentially going to be a simulation of twelve constructors in a language with only two. The chosen way could be called the left-factorisation. In this definition let $\mathtt{R}^n$ be a short-hand for a sequence of n `R`'s.

$$
\begin{aligned}
\mathcal{C}_{\mathrm{exp}}(i) &= \mathtt{L}\ i \\
\mathcal{C}_{\mathrm{exp}}(e_1\ o\ e_2) &= \mathtt{R\ L\ (}\mathcal{C}_{\mathrm{oper}}(o)\mathtt{,(}\mathcal{C}_{\mathrm{exp}}(e_1)\mathtt{,}\mathcal{C}_{\mathrm{exp}}(e_2)\mathtt{))} \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{(}e_1\mathtt{,}e_2\mathtt{)}) &= \mathtt{R}^2\ \mathtt{L\ (}\mathcal{C}_{\mathrm{exp}}(e_1)\mathtt{,}\mathcal{C}_{\mathrm{exp}}(e_2)\mathtt{)} \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{fst}\ e) &= \mathtt{R}^3\ \mathtt{L\ (}\mathcal{C}_{\mathrm{exp}}(e)\mathtt{)} \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{snd}\ e) &= \mathtt{R}^4\ \mathtt{L\ (}\mathcal{C}_{\mathrm{exp}}(e)\mathtt{)} \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{inl}\ e) &= \mathtt{R}^5\ \mathtt{L\ (}\mathcal{C}_{\mathrm{exp}}(e)\mathtt{)} \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{inr}\ e) &= \mathtt{R}^6\ \mathtt{L\ (}\mathcal{C}_{\mathrm{exp}}(e)\mathtt{)} \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{case}\ e\ \mathtt{of}\ \ldots) &= \mathtt{R}^7\ \mathtt{L\ (}\mathcal{C}_{\mathrm{exp}}(e)\mathtt{,(}x_\ell\mathtt{,(}\mathcal{C}_{\mathrm{exp}}(e_\ell)\mathtt{,} \\
&\qquad \mathtt{(}x_r\mathtt{,(}\mathcal{C}_{\mathrm{exp}}(e_r)\mathtt{)))))} \\
\mathcal{C}_{\mathrm{exp}}(x) &= \mathtt{R}^8\ \mathtt{L}\ x \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{error}) &= \mathtt{R}^9\ \mathtt{L\ 0} \\
\mathcal{C}_{\mathrm{exp}}(f\ e) &= \mathtt{R}^{10}\ \mathtt{L\ (}f\mathtt{,}\mathcal{C}_{\mathrm{exp}}(e)\mathtt{)} \\
\mathcal{C}_{\mathrm{exp}}(\mathtt{let}\ x\mathtt{=}e_1\ \mathtt{in}\ e_2\ \mathtt{end}) &= \mathtt{R}^{11}\ \mathtt{(}x\mathtt{,(}\mathcal{C}_{\mathrm{exp}}(e_1)\mathtt{,}\mathcal{C}_{\mathrm{exp}}(e_2)\mathtt{))}
\end{aligned}
$$

The function $\mathcal{C}_{\mathrm{exp}}$ is realised by the HOL-function `code_exp`.

Note that it is easy for a program to decode values constructed in this way by using eleven nested `case`-expressions with all the nesting in the `R`-branch.

Encoding of Operators

Operators are encoded in the same style as expressions, i.e., simulating four constructors using only two:

$$
\begin{aligned}
\mathcal{C}_{\mathrm{oper}}(\mathtt{=}) &= \mathtt{L\ 0} \\
\mathcal{C}_{\mathrm{oper}}(\mathtt{+}) &= \mathtt{R\ L\ 0} \\
\mathcal{C}_{\mathrm{oper}}(\mathtt{-}) &= \mathtt{R\ R\ L\ 0} \\
\mathcal{C}_{\mathrm{oper}}(\mathtt{*}) &= \mathtt{R\ R\ R\ 0}
\end{aligned}
$$

The function $\mathcal{C}_{\mathrm{oper}}$ is realised by the HOL-function `code_oper`.

5.2.2 Encoding of Values

The choice of encoding for values is slightly, but only slightly, more sensitive than encoding of syntax due to the fact that during partial evaluation we expect values

to be unknown. There are four kinds of values: integers, pairs of values, left-injected values, and right-injected values. It seems wise to use an encoding such that the images of (1) the integers, (2) the pairs, and (3) the left- or right-injected values are (easily) distinguishable. This means that the encoding shows the type of the encoded value. Left-factorisation achieves this (when the kinds of values are listed in the order above):

$$\begin{aligned}
\mathcal{C}_{\text{val}}(i) &= \texttt{L}\ i \\
\mathcal{C}_{\text{val}}((v_1, v_2)) &= \texttt{R L}\ (\mathcal{C}_{\text{val}}(v_1), \mathcal{C}_{\text{val}}(v_2)) \\
\mathcal{C}_{\text{val}}(\texttt{L}\ v) &= \texttt{R R L}\ (\mathcal{C}_{\text{val}}(v)) \\
\mathcal{C}_{\text{val}}(\texttt{R}\ v) &= \texttt{R R R}\ (\mathcal{C}_{\text{val}}(v))
\end{aligned}$$

The function $\mathcal{C}_{\text{val}}$ is realised by the HOL-function `code_value`.

5.2.3 Encoding of Environments

The self-interpreter keeps track of values of bound variables by using an environment in the same way the inference rules do. An environment is a list and we will use a simple encoding that encodes $[]$ as `L 0` and `CONS` as `R`. The variable names are coded as numbers (they already are) and the values are encoded as above.

$$\begin{aligned}
\mathcal{C}_{\text{env}}([]) &= \texttt{L 0} \\
\mathcal{C}_{\text{env}}(\texttt{CONS}\,(x, v)\,E) &= \texttt{R}\ ((x, \mathcal{C}_{\text{val}} v), \mathcal{C}_{\text{env}} E)
\end{aligned}$$

The function $\mathcal{C}_{\text{env}}$ is realised by the HOL-function `code_env`.

5.2.4 Example

As an example of how the encoding works, consider the tiny program

```
f x = x * x;
```

Let `f` be encoded as `1` and `x` be encoded as `2`. Then the program will be coded as the value

```
R ((1,(2,R L (R R R 0,(R R R R R R R R L 2,
                       R R R R R R R R L 2)))),
   L 0)
```

The presence of a linear blow-up in size is normal for encodings; the factor is somewhat large here because the target value system is quite limited.

5.2.5 Self-Interpreter Program Text

Figures 10 through 13 show the actual program text of the self-interpreter. To ease the presentation, a small amount of ad-hoc syntactic sugar has been added:

General tupling is used with the meaning that `(e1,e2,...,en)` is a shorthand for `(e1,(e2,(...,en)...))`, i.e., right-associative pairing.

Multiple function parameters as in `f (x1,...,xn) = e` are used as shorthands for `f x = e'` where `x` is a fresh variable and `e'` is `e` wrapped in a series of `let`-bindings picking out components of `x`. Note that the corresponding sugaring at the other variable-binding places (i.e., `let` and `case`) is not done.

Standard ML-isms like `fun`, `_`, and `=>` have been used.

The above is an informal description only; the authoritative word on the desugared version of the self-interpreter can be found in Appendix C which shows the actual HOL-term used for the proofs. Note that the functions are shown in a different order, except for the main function which defines the meaning.

Some notes on the coding are in order. The interpreter uses environments to keep track of bound variables and the values they are bound to. These environments are represented as lists of pairs of variable names and values, where the empty list is `L 0` and the "cons" operator is just `R`.

Error handling in the interpreter is very simple: in case of type errors, e.g., trying to take the first component of an integer, the interpreter will try to do the same and thus fail as the object program would. In case of scope violations and in case of an object program `error` being reached an explicit `error` construct will be evaluated to signal the condition.

Furthermore, some functions need a little description:

Function pelint is the main function. It extracts the first function of the object program, builds the initial environment for it, and starts the expression evaluator.

Functions int2val, pair2val, and sum2val are the tagging operators injecting into the universal type. Note that these do not form an implementation of $\mathcal{C}_{\text{val}}$ as they only encode at top level.

Functions val2int, val2pair, and val2sum are the corresponding projections. If the projections are used on ill-formed values an error is signalled.

Function eval_op implements the binary integer operations. Note that it takes two integers and returns a (universal) value. This is because the operators do not return the same types of values.

```
fun pelint (prg,val) =
  let
    main = hd prg
  in
    eval (snd snd main, update_env (L 0,fst snd main,val),
          prg)
  end;

fun eval (expr,env,prg) =
  case expr of
    L.i => int2val i
  | R.e1 =>
    case e1 of
      L.Oe1e2 =>
        eval_op (fst Oe1e2,
                 val2int (eval (fst snd Oe1e2,env,prg)),
                 val2int (eval (snd snd Oe1e2,env,prg)))
    | R.e2 =>
      case e2 of
        L.ee => pair2val (eval (fst ee,env,prg),
                          eval (snd ee,env,prg))
      | R.e3 =>
        case e3 of
          L.e => fst (val2pair (eval (e,env,prg)))
        | R.e4 =>
          case e4 of
            L.e => snd (val2pair (eval (e,env,prg)))
          | R.e5 =>
            case e5 of
              L.e => sum2val L (eval (e,env,prg))
            | R.e6 =>
              case e6 of
                L.e => sum2val R (eval (e,env,prg))
              | R.e7 =>
                case e7 of
                  L.exexe =>
                    case val2sum (eval (fst exexe,env,prg)) of
                      L.v => eval (fst snd snd exexe,
                                   update_env (env,
                                               fst snd exexe,
                                               v),
                                   prg)
                    | R.v => eval (snd snd snd snd exexe,
                                   update_env (env,
                                               fst snd
                                                 snd snd exexe,
                                               v),
                                   prg)
                    end
                | R.e8 =>
(continued)
```

Figure 10: Self-interpreter, part 1.

(continued)

```
                    case e8 of
                      L.v => lookup_env (v,env)
                    | R.e9 =>
                      case e9 of
                        L._ => error
                      | R.e10 =>
                        case e10 of
                          L.fe =>
                            let
                              varbody = lookup_func (fst fe,prg)
                            in
                              eval (snd varbody,
                                    update_env
                                      (L 0,
                                       fst varbody,
                                       eval (snd fe,env,prg)),
                                    prg)
                            end
                        | R.vee =>
                            let
                              v1 = eval (fst snd vee,env,prg)
                            in
                              eval (snd snd vee,
                                    update_env (env,fst vee,v1),
                                    prg)
                            end
                        end
                      end
                    end
                  end
                end
              end
            end
          end
        end
      end
    end;

  fun lookup_env (v,env) =
    case env of
      L._ => error
    | R.vve =>
        case (v = fst (fst vve)) of
          L._ => lookup_env (v,snd vve)
        | R._ => snd (fst vve)
        end
    end;

  fun update_env (env,var,val) =
    R ((var,val),env);
```

Figure 11: Self-interpreter, part 2.

```
fun int2val i = L i;
fun pair2val p = R L p;
fun sum2val s = R R s;

fun val2int v =
  case v of
    L.i => i
  | R.x => error
  end;

fun val2pair v =
  case v of
    L.i => error
  | R.v' =>
    case v' of
      L.p => p
    | R.x => error
    end
  end;

fun val2sum v =
  case v of
    L.i => error
  | R.v' =>
    case v' of
      L.p => error
    | R.s => s
    end
  end;

fun lookup_func (f,prg) =
  case prg of
    L._ => error
  | R.fxeP =>
      case (f = fst fst fxeP) of
        L._ => lookup_func (f,snd fxeP)
      | R._ => snd fst fxeP
      end
  end;

fun hd l =
  case l of
    L._ => error
  | R.el => fst el
  end;
```

Figure 12: Self-interpreter, part 3.

Function eval is the expression evaluator — the heart of the self-interpreter. It takes the complete object program as an extra argument in order to be able to look up functions that are called. The function decodes the expression and performs the object program's operations by similar operations in the

```
fun eval_op (op,i1,i2) =
  case op of
    L._ =>
      case (i1 = i2) of
        L._ => sum2val (L (int2val 0))
      | R._ => sum2val (R (int2val 0))
      end
  | R.op' =>
    case op' of
      L._ => int2val (i1 + i2)
    | R.op'' =>
      case op'' of
        L._ => int2val (i1 - i2)
      | R._ => int2val (i1 * i2)
      end
    end
  end;
```

Figure 13: Self-interpreter, part 4

self-interpreter. Operations are wrapped by suitable tagging and untagging stemming from the universal encoding.

5.3 Correctness of the Self-Interpreter

Intuitively, an interpreter is correct if evaluation via the interpreter produces the same result and direct evaluation for any program and for any possible input:

$$\texttt{int} \text{ is correct (ver. 1)} \iff \forall P, i : [\![P]\!](i) = [\![\texttt{int}]\!](P, i). \tag{5.1}$$

There are complications, however.

- Equation 5.1 requires that the application of `int` to P and i is valid. In a language like Scheme, where all values and all programs are S-expressions, this is not a problem but in a language like Standard ML, it is not possible since both $\texttt{int}(P, 4)$ and $\texttt{int}(P, (2, 3))$ would have to be acceptable. This means that `int` should have type $\forall \tau_1, \tau_2.(\texttt{prg} \times \tau_1) \to \tau_2$ effectively barring it from inspecting the input value. Thus, we must encode input values when working with a typed language.

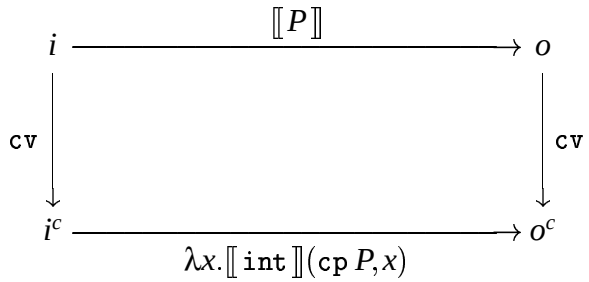
- If we restrict ourselves to constant-valued Ps, i.e., programs that do not inspect their input values, then we see that the interpreter must also be able to produce both 1 and $(2, 3)$ (if these are values for the language). No reasonable typed language will allow one program, `int`, to produce two such values from nothing but a (monomorphic) program. Output values must therefore likewise be encoded in the typed setting.

- The equality should be interpreted such that the two sides are defined at the same time and when the two sides are defined their values are identical.

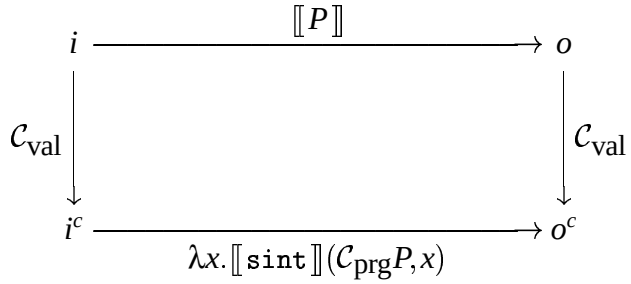
To be able to describe typed languages we must take the necessary encoding into account, i.e., the fact that the interpreter work with encodings of i and not i itself. Assume that the function `cv` describes the encoding of both input and output values and that the function `cp` encodes programs. Then we have the following situation:

$$\texttt{int} \text{ is correct (ver. 2)} \iff \forall P, i : \texttt{cv}\,([\![P]\!](i)) = [\![\,\texttt{int}\,]\!](\texttt{cp}\,(P), \texttt{cv}\,(i)). \qquad (5.2)$$

Assuming, quite reasonably, that the encoding operations are strict and total then the encoding operation do not add to or subtract from the need to compare the sides with respect to definedness also. Equation 5.2 corresponds to requiring the following diagram be commutative:



In the following we will show this correctness theorem for the self-interpreter in Figures 10 through 13 together with the encoding operations $\mathcal{C}_{\text{prg}}$ and $\mathcal{C}_{\text{val}}$, i.e., we will prove that this diagram is commutative:



5.3.1 Organisation of the Correctness Proof

Even though the correctness condition is expressed in terms of the semantics for programs it is no surprise that the key to its proof lies in proving a more general lemma for the expression semantics. But before doing this it is convenient to state and prove properties of several of the self-interpreter's functions.

We therefore start by proving the correctness of the interpreter's `lookup_func` and `lookup_env` functions. These two functions are recursive and hence simple unfolding where they are used is not an option. We then prove the correctness of `eval_oper` which contains a case split (on the operator) that would be awkward if simply unfolded. Having done that we are ready to prove correctness as outlined above.

We conclude by proving the interpreter statically correct. Note that this fact will not be used in the proof of evaluation correctness.

5.3.2 The Interpreter's "`lookup_func`" Function

The function `lookup_func` searches through an encoded program for a given function. It returns the formal parameter and the body expression as a pair. In the HOL-encoding of the self-interpreter, see Appendix C, the function has the number 11.

The following shows that in a sense the self-interpreter's `lookup_func` function works like the semantic function `lookup_func` thereby earning its name:

Lemma 5.1 (`lookup_func`)

$$\vdash \forall P, f, v : \begin{pmatrix} \exists x, e : \begin{pmatrix} (\texttt{OK}\,(x,e) = \texttt{lookup\_func sint}\ 11) \wedge \\ \texttt{eval\_expr}\ e\ \texttt{sint}\,[(x, \texttt{V\_Pair}\,(\texttt{V\_Int}\ f)\,(\texttt{code\_prg}\,P))]\,v \end{pmatrix} \\ \Leftrightarrow \\ \exists x, e : \begin{pmatrix} (\texttt{OK}\,(x,e) = \texttt{lookup\_func}\,P\,f) \wedge \\ (v = \texttt{V\_Pair}\,(\texttt{V\_Int}\,x)\,(\texttt{code\_exp}\,e)) \end{pmatrix} \end{pmatrix}$$

Proof: List induction on the program. The base case, $[\,]$, is trivial. The induction step, $\texttt{CONS}\,(f',x',e')\,T$, needs two cases: if $f = f'$ then the case follows by reduction; if $f \neq f'$ then the case follows by the induction hypothesis. □

5.3.3 The Interpreter's "`lookup_env`" Function

The function `lookup_env` in the self-interpreter searches through an encoded environment for a given variable. It returns the corresponding value. In the HOL-encoding of the self-interpreter, see Appendix C, the function has the number 10.

The following lemma shows that in a sense the self-interpreter's `lookup_env` function works like the semantic function `lookup_env`.

Lemma 5.2 (`lookup_env`)

$$\vdash \forall E, x, v : \left(\begin{array}{l} \exists x', e' : \left(\begin{array}{l} (\texttt{OK}\,(x', e') = \texttt{lookup\_func sint } 10) \wedge \\ \texttt{eval\_expr}\; e'\; \texttt{sint} \\ \qquad\qquad [(x', \texttt{V\_Pair}\,(\texttt{V\_Int}\; x)\,(\texttt{code\_env}\; E))]\, v \end{array} \right) \\ \Leftrightarrow \\ \exists v' : \left(\begin{array}{l} (\texttt{OK}\; v' = \texttt{lookup\_env}\; E\; x) \wedge \\ (v = \texttt{code\_value}\; v') \end{array} \right) \end{array} \right)$$

Proof: List induction on the environment. The base case, $[\,]$, is trivial. The induction step, $\texttt{CONS}\,(x', v')\,E'$, needs two cases: if $x = x'$ then the case follows by reduction; if $x \neq x'$ then the case follows by the induction hypothesis. □

5.3.4 The Interpreter's "`eval_op`" Function

The function `eval_op` in the self-interpreter is a function that given two integers and an encoded operator performs the corresponding binary operation and returns the result as an encoded value. In the HOL-encoding of the self-interpreter, see Appendix C, the function has the number 12.

The following lemma shows that in a sense the self-interpreter's `eval_oper` function works like the semantic function `eval_oper` thereby earning its name:

Lemma 5.3 (`eval_oper`)

$$\vdash \forall o, i_1, i_2, v : \left(\begin{array}{l} \exists x, e : \left(\begin{array}{l} (\texttt{OK}\,(x, e) = \texttt{lookup\_func sint } 12) \wedge \\ \texttt{eval\_expr}\; e\; \texttt{sint}\,[(x, \texttt{V\_Pair}\,(\texttt{code\_oper}\; o) \\ \qquad\qquad (\texttt{V\_Pair}\,(\texttt{V\_Int}\; i_2)\,(\texttt{V\_Int}\; i_2)))]\, v \end{array} \right) \\ \Leftrightarrow \\ (v = \texttt{code\_value}\,(\texttt{eval\_oper}\; o\, i_1\, i_2)) \end{array} \right)$$

Proof: By structural case analysis on the operator. □

5.3.5 The Interpreter's "`eval`" Function

Not surprisingly the `eval` function of the self-interpreter is harder to prove correct than any of the other functions. It turns out that it is necessary to invoke the nested-call timed semantics in order to prove a crucial lemma for the correctness.

The following lemma shows that if the `eval_expr` predicate holds for certain arguments then evaluation of the interpreter's `eval` function could have been used

to deduce this fact. Since the interpreter does evaluation using more steps than direct evaluation this is the relatively easy part. In the HOL-encoding of the self-interpreter, see Appendix C, the function has the number 8.

Lemma 5.4 (First `eval` Lemma.)

$$\vdash \forall e,P,E,v: \left(\begin{array}{l} \texttt{eval\_expr}\ e\,P\,E\,v \Rightarrow \\ \quad \exists x',e': \left(\begin{array}{l} (\texttt{OK}\,(x',e') = \texttt{lookup\_func}\ \texttt{sint}\ 8) \wedge \\ \texttt{eval\_expr}\ e'\ \texttt{sint} \\ \qquad [(x', \texttt{V\_Pair}\,(\texttt{code\_exp}\,e)\,(\texttt{V\_Pair} \\ \qquad\quad (\texttt{code\_env}\,E)\,(\texttt{code\_prg}\,P)))] \\ \qquad (\texttt{code\_value}\,v) \end{array} \right) \end{array} \right)$$

Proof: By strong rule induction. For each case the `code_exp` function is expanded and the expression is reduced leaving only the relevant branch of `eval`'s nested `case`-structure. Each branch is then handled in the following way:

Case `E_Int` i**:** By unfolding of call to `int2val`.

Case `E_Op` $o\,e_1\,e_2$**:** By unfolding of calls to `val2int`. Then by induction hypothesis and Lemma 5.3.

Case `E_Pair` $e_1\,e_2$**:** By unfolding of call to `pair2val`. Then by induction hypothesis.

Cases `E_Fst` e **and** `E_Snd` e**:** By unfolding of call to `val2pair`. Then by induction hypothesis.

Cases `E_Inl` e **and** `E_Inr` e**:** By unfolding of call to `sum2val`. Then by induction hypothesis.

Cases `E_Case` $e\,x_\ell\,e_\ell\,x_r\,e_r$ **[left,right]:** By unfolding of the calls to `val2sum` and to `update_env`. Then by induction hypothesis.

Case `E_Var` x**:** By Lemma 5.2.

Case `E_Let` $x\,e_1\,e_2$**:** By unfolding of call to `update_env`. Then by induction hypothesis.

Case `E_Call` $f\,e$**:** By Lemma 5.1 and unfolding of call to `update_env`. Then by induction hypothesis.

The theorem now follows from Theorem 3.18. □

The implication in the other direction is harder to prove because direct evaluation needs fewer evaluation steps than evaluation via the interpreter. We therefore start

out by proving the following lemma, which uses the nested-call timed semantics instead. After this we will use the approximation property of the nested-call timed semantics, see Section 4.4, to prove the lemma we really need.

Furthermore, this lemma is slightly stronger than simply the opposite implication of Lemma 5.4. Experience shows that it is necessary to prove — in advance or simultaneously — that any result from the interpreter is a properly encoded value (provided that the inputs have the right form). Note that this is essentially a type property.

Lemma 5.5 (Second `eval` Lemma.)

$$\vdash \exists x', e' : \left(\begin{array}{l} (\texttt{OK}\,(x', e') = \texttt{lookup\_func sint}\ 8) \wedge \\ \\ \forall N, e, P, E, v^c : \left(\begin{array}{l} \texttt{eval\_expr\_n}\ N\, e'\ \texttt{sint} \\ \qquad [(x', \texttt{V\_Pair}\,(\texttt{code\_exp}\, e)\,(\texttt{V\_Pair} \\ \qquad\quad (\texttt{code\_env}\, E)\,(\texttt{code\_prg}\, P)))]\, v^c \\ \Rightarrow \\ \exists v : \texttt{eval\_expr}\, e\, P\, E\, v \wedge (v^c = \texttt{code\_value}\, v) \end{array} \right) \end{array} \right)$$

Proof: The existence of x' and e' such that the first conjunct is satisfied is first proven by brute-force expansion of `sint`'s and `lookup_func`'s definitions.

We now prove that the resulting x' and e' satisfy the second conjunct also. This is done by (numeric) induction on N. The base case is trivial as the antecedent is always false (because every evaluation of `eval_expr_n` includes a function call).

The numeric induction case is handled by structural induction on the expression. Every case has the call to `code_exp` expanded and is reduced. Again, this leaves us with only the relevant branch of `eval`'s nested `case`-structure. We unfold all calls to `int2val`, `val2int`, `pair2val`, `val2pair`, `sum2val`, `val2sum`, and `update_env`. This leaves the following cases:

Case `E_Int` i**:** Trivial.

Case `E_Op` $o\, e_1\, e_2$**:** By Lemma 4.17 and the structural induction hypothesis. Then by structural case analysis on the results obtained by evaluating the subexpressions directly. Finally by Lemma 4.12 and Lemma 5.4.

Case `E_Pair` $e_1\, e_2$**:** By Lemma 4.17 and the structural induction hypothesis.

Cases `E_Fst` e **and** `E_Snd` e**:** By Lemma 4.17 and the structural induction hypothesis. Then by structural case analysis on the result obtained by evaluating the subexpression directly.

Cases `E_Inl` e **and** `E_Inr` e**:** By Lemma 4.17 and the structural induction hypothesis.

Case `E_Case` $e\,x_\ell\,e_\ell\,x_r\,e_r$**:** By Lemma 4.17 and the structural induction hypothesis. Then by structural case analysis on the result obtained by evaluating e directly.

Case `E_Var` x**:** By Lemmas 4.12 and 5.2.

Case `E_Error`**:** Trivial.

Case `E_Let` $x\,e_1\,e_2$**:** By Lemma 4.17 and the structural induction hypothesis.

Case `E_Call` $f\,e$**:** The antecedent, the structural induction hypothesis, Lemmas 4.12 and 5.1 handle the evaluation of the argument. The numeric induction hypothesis and Lemma 4.17 handle the evaluation of the body.

The theorem now follows from Theorem 3.8. □

We are now able to prove (essentially) the opposite implication of 5.4:

Lemma 5.6 (Third `eval` Lemma.)

$$\vdash \exists x', e' : \left(\begin{array}{l} (\mathtt{OK}\,(x', e') = \mathtt{lookup\_func}\ \mathtt{sint}\ 8) \wedge \\ \forall e, P, E, v^c : \left(\begin{array}{l} \mathtt{eval\_expr}\ e'\ \mathtt{sint} \\ \quad [(x', \mathtt{V\_Pair}\,(\mathtt{code\_exp}\,e)\,(\mathtt{V\_Pair} \\ \quad\quad (\mathtt{code\_env}\,E)\,(\mathtt{code\_prg}\,P)))]\,v^c \\ \Rightarrow \\ \exists v : \mathtt{eval\_expr}\,e\,P\,E\,v \wedge (v^c = \mathtt{code\_value}\,v) \end{array} \right) \end{array} \right)$$

Proof: By Lemmas 5.5 and 4.15. □

The three `eval` lemmas have now cleared the road for the following theorem stating that `eval` is correct. (One could easily eliminate the existential quantifier on x' and e' but that would require explicitly stating the body of `eval` instead; that term is large so this logically equivalent version is used instead.)

Theorem 5.7 (Correctness of `eval`.)

$$\vdash \forall e, P, E, v : \left(\begin{array}{l} \mathtt{eval\_expr}\,e\,P\,E\,v \Leftrightarrow \\ \exists x', e' : \left(\begin{array}{l} (\mathtt{OK}\,(x', e') = \mathtt{lookup\_func}\ \mathtt{sint}\ 8) \wedge \\ \mathtt{eval\_expr}\ e'\ \mathtt{sint} \\ \quad [(x', \mathtt{V\_Pair}\,(\mathtt{code\_exp}\,e)\,(\mathtt{V\_Pair} \\ \quad\quad (\mathtt{code\_env}\,E)\,(\mathtt{code\_prg}\,P)))] \\ \quad (\mathtt{code\_value}\,v) \end{array} \right) \end{array} \right)$$

Proof: By Lemmas 5.4 and 5.6. □

5.3.6 Main Correctness Results

The lemmas of Section 5.3.5 are for expressions but since the semantics of a program is basically the semantics of its first function body we are now ready to state and prove the correctness of the interpreter itself. Since the formalisation uses a predicate and not a partial function the correctness result is split into three theorems.

Theorem 5.8 (Self-Interpreter Evaluates Correctly.)

$$\vdash \forall P, v_{in}, v_{out} : \left(\begin{array}{l} \texttt{eval\_prg}\, P\, v_{in}\, v_{out} \Leftrightarrow \\ \texttt{eval\_prg}\, \texttt{sint}\, (\texttt{V\_Pair}\, (\texttt{code\_prg}\, P)\, (\texttt{code\_value}\, v_{in})) \\ \qquad (\texttt{code\_value}\, v_{out}) \end{array} \right)$$

Proof: Structural case analysis on the program. Case [] (the empty program introduced by the formalisation) is trivial. Case $\texttt{CONS}\, H\, T$ is proven easily by reduction, by unfolding calls to `hd` and `update_env`, and using Theorem 5.7. □

Theorem 5.9 (Self-interpreter Produces Only Correct Results)

$$\vdash \forall P, v_{in}, v^c_{out} : \left(\begin{array}{l} \texttt{eval\_prg}\, \texttt{sint}\, (\texttt{V\_Pair}\, (\texttt{code\_prg}\, P)\, (\texttt{code\_value}\, v_{in}))\, v^c_{out} \\ \Rightarrow \\ \exists v_{out} : v^c_{out} = \texttt{code\_value}\, v_{out} \end{array} \right)$$

Proof: Structural case analysis on the program. Case [] (the empty program introduced by the formalisation) is trivial. Case $\texttt{CONS}\, H\, T$ is proven easily by reduction, by unfolding calls to `hd` and `update_env`, and using Theorem 5.7. □

Theorem 5.10 (Self-Interpreter Has Correct Termination Properties.)

$$\vdash \forall P, v_{in} : \left(\begin{array}{l} \exists v_{out} : \texttt{eval\_prg}\, P\, v_{in}\, v_{out} \Leftrightarrow \\ \exists v^c_{out} : \texttt{eval\_prg}\, \texttt{sint}\, \left(\texttt{V\_Pair}\, (\texttt{code\_prg}\, P)\, (\texttt{code\_value}\, v_{in}) \right) v^c_{out} \end{array} \right)$$

Proof: By Theorems 5.8 and 5.9. □

5.3.7 Static Correctness

The self-interpreter is not only dynamically correct as proven in the previous sections; it is also statically correct:

Theorem 5.11

$$\vdash \texttt{stat\_ok}\ \texttt{sint}$$

Proof: By brute-force expansion of `stat_ok` and the utility functions it uses. □

Note that the proof of dynamic correctness is completely independent of the proof of static correctness. It would, in fact, be simple to modify the self-interpreter in such a way that the dynamic properties were unchanged while the static properties were ruined.

5.3.8 Typing

The self-interpreter can be given a type according to the definitions in Section 3.7. If we define the following shorthands

$$\begin{aligned}
\mathit{SintVal} &\equiv \mu\, t_0.\, \texttt{int} + ((t_0 \times t_0) + (t_0 + t_0)) && (5.3)\\
\mathit{SintOp} &\equiv \texttt{int} + (\texttt{int} + (\texttt{int} + \texttt{int})) && (5.4)\\
\mathit{SintExp} &\equiv \mu\, t_1.\, \texttt{int} && (5.5)\\
&\quad + ((\mathit{SintOp} \times (t_1 \times t_1))\\
&\quad + ((t_1 \times t_1)\\
&\quad + (t_1\\
&\quad + (t_1\\
&\quad + (t_1\\
&\quad + (t_1\\
&\quad + ((t_1 \times (\langle \mathit{Var}\rangle \times (t_1 \times (\langle \mathit{Var}\rangle \times t_1))))\\
&\quad + (\langle \mathit{Var}\rangle\\
&\quad + (\texttt{int}\\
&\quad + ((\langle \mathit{Func}\rangle \times t_1)\\
&\quad + ((\langle \mathit{Var}\rangle \times (t_1 \times t_1)))))))))))\\
\mathit{SintEnv} &\equiv \mu\, t_2.\, \texttt{int} + ((\langle \mathit{Var}\rangle \times \mathit{SintVal}) \times t_2) && (5.6)\\
\mathit{SintPrg} &\equiv \mu\, t_3.\, \texttt{int} + ((\langle \mathit{Func}\rangle \times (\langle \mathit{Var}\rangle \times \mathit{SintExp})) \times t_3) && (5.7)
\end{aligned}$$

where $\langle \mathit{Var}\rangle$ and $\langle \mathit{Func}\rangle$ are used for `int` (the representaion of those names), then we can assign types to every function in the following way.

Function	*Argument*	*Result*
`pel_int`	$\mathit{SintPrg} \times \mathit{SintVal}$	$\mathit{SintVal}$
`int2val`	`int`	$\mathit{SintVal}$
`val2int`	$\mathit{SintVal}$	`int`
`pair2val`	$\mathit{SintVal} \times \mathit{SintVal}$	$\mathit{SintVal}$
`val2pair`	$\mathit{SintVal}$	$\mathit{SintVal} \times \mathit{SintVal}$
`sum2val`	$\mathit{SintVal} + \mathit{SintVal}$	$\mathit{SintVal}$
`val2sum`	$\mathit{SintVal}$	$\mathit{SintVal} + \mathit{SintVal}$
`eval`	$\mathit{SintExp} \times (\mathit{SintEnv} \times \mathit{SintPrg})$	$\mathit{SintVal}$
`update_env`	$\mathit{SintEnv} \times (\langle \mathit{Var} \rangle \times \mathit{SintVal})$	$\mathit{SintEnv}$
`lookup_env`	$\langle \mathit{Var} \rangle \times \mathit{SintEnv}$	$\mathit{SintVal}$
`lookup_func`	$\langle \mathit{Func} \rangle \times \mathit{SintPrg}$	$\langle \mathit{Var} \rangle \times \mathit{SintExp}$
`eval_op`	$\mathit{SintOp} \times (\texttt{int} \times \texttt{int})$	$\mathit{SintVal}$
`hd`	$\mathit{SintPrg}$	$\langle \mathit{Func} \rangle \times (\langle \mathit{Var} \rangle \times \mathit{SintExp})$

Theorem 5.12 (Self-Interpreter is Well-Typed.)

$$\vdash \texttt{prg\_has\_type sint sint\_type}$$

Proof: By construcing the type derivation tree. □

(This proof is gigantic. The fact that the expression typing is not syntax directed makes automation hard — at least when memory usage must be controlled.)

5.4 Final Note

The interpreter in Figures 10 through 13 is correct as proven in the previous sections. It has, however, never been run and therefore never tested.

Chapter 6

Program Transformations

Are my methods unsound?
— COL. WALTER E. KURTZ, Apocalypse Now, 1979

In this section we shall discuss, present, and prove the correctness of a number of program transformations. Later, in Chapter 7, we shall see how these can be combined to do the work of a partial evaluator. the following table gives an overview of the transformations.

Sec.	*Transformation*	*Two-Way?*	*Repl?*
6.1	Extended constant fold, i.e., elimination of neighbouring producer and consumer	Yes	Yes
6.2	Alpha conversion	Yes	Yes
6.3	Unfolding of `let` bindings	Yes	Yes
6.4	Folding (introduction of) `let` bindings	No [1]	Yes
6.5	Moving identity-function calls	Yes	Yes
6.6	Elimination of calls to identity functions	No	Yes
6.7	Moving function argument from `let` to call	Yes	Yes
6.8	Unfolding a function call	Yes	Yes
6.9	Using old definition of function to perform folding of function call	Yes	Yes
6.10	Removing unused function definitions	No [2]	No
6.11	Adding a function definition	Yes	No

[1] Yes, under certain conditions [2] Reverse operation is two-way.

"*Repl?*" means that the program transformation is described in terms of replacing one subexpression by another. "*Two-Way?*" means that the source and target programs will be in both the `improvement` and `improvementR` relations. This, as we

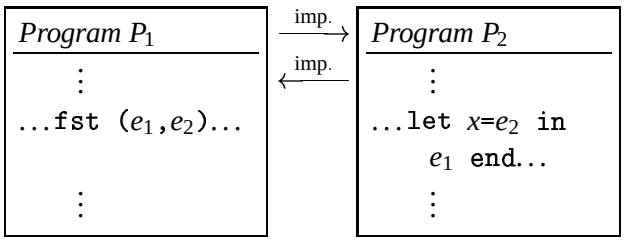
shall see in Sections 6.9 and 7.6, means that the transformations can be chained while still allowing function call folding to be used freely. Program transformations that are not two-way are for use in a clean-up phase at the end.

6.1 Extended Constant Fold

Constant folding is normally the replacement of expressions like `(2+3)` with a constant expression, `5` in situ. It is the closeness of the constants ("the producers") to the addition operator ("the consumer") that makes this operation straightforward. In the work we shall define constant folding to include several other situation where the consumer of a value is located next to the producer or producers of that value:

Source expression		*Target expression*
`(`i_1 o i_2`)`	$\rightarrow$	`value2expr(eval_oper` $o\, i_1\, i_2$`)`
`fst (`e_1`,`e_2`)`	$\rightarrow$	`let` x`=`e_2 `in` e_1 `end`
`snd (`e_1`,`e_2`)`	$\rightarrow$	`let` x`=`e_1 `in` e_2 `end`
`case L` e `of L.`x_ℓ`->`e_ℓ `| ... end`	$\rightarrow$	`let` x_ℓ`=`e `in` e_ℓ `end`
`case R` e `of ... | R.`x_r`->`e_r `end`	$\rightarrow$	`let` x_r`=`e `in` e_r `end`

The different x's here are fresh variables with respect to the expression for which x is in scope. Note that the transformation of `fst (`e_1`,`e_2`)` relies on the fact that evaluation order does not matter.



(example for `fst`*)*

In the above constant folding we carefully avoid changing termination properties by inserting `let`-bindings in the `fst`- and `snd`-cases. A separate program transformation (namely `let`-unfolding) can be used to eliminate that `let` under certain conditions.

The function `cst_fold` realises these transformations and — in order to make the function total — passes anything else:

Definitional Theorem 6.1 (Constant Folding.)

$$
\vdash \left\{ \begin{array}{l}
\forall i : \texttt{cst\_fold}\,(\texttt{E\_Int}\ i) = (\texttt{E\_Int}\ i) \\
\forall o, e_1, e_2 : \texttt{cst\_fold}\,(\texttt{E\_Op}\ o\, e_1\, e_2) = \\
\quad (\exists i_1, i_2 : (e_1 = \texttt{E\_Int}\ i_1) \wedge (e_2 = \texttt{E\_Int}\ i_2)) \\
\quad \rightarrow (\texttt{value2expr}\,(\texttt{eval\_oper}\ o\,(\texttt{expr2int}\ e_1)\,(\texttt{expr2int}\ e_2))) \\
\quad \mid (\texttt{E\_Op}\ o\, e_1\, e_2) \\
\forall e_1, e_2 : \texttt{cst\_fold}\,(\texttt{E\_Pair}\ e_1\, e_2) = (\texttt{E\_Pair}\ e_1\, e_2) \\
\forall e : \texttt{cst\_fold}\,(\texttt{E\_Fst}\ e) = \\
\quad (\exists e_1, e_2 : e = \texttt{E\_Pair}\ e_1\, e_2) \\
\quad \rightarrow \texttt{E\_Let}\,(\texttt{genvar}\,(\texttt{child}_1\, e))\,(\texttt{child}_2\, e)\,(\texttt{child}_1\, e) \\
\quad \mid (\texttt{E\_Fst}\ e) \\
\forall e : \texttt{cst\_fold}\,(\texttt{E\_Snd}\ e) = \\
\quad (\exists e_1, e_2 : e = \texttt{E\_Pair}\ e_1\, e_2) \\
\quad \rightarrow \texttt{E\_Let}\,(\texttt{genvar}\,(\texttt{child}_2\, e))\,(\texttt{child}_1\, e)\,(\texttt{child}_2\, e) \\
\quad \mid (\texttt{E\_Snd}\ e) \\
\forall e : \texttt{cst\_fold}\,(\texttt{E\_Inl}\ e) = (\texttt{E\_Inl}\ e) \\
\forall e : \texttt{cst\_fold}\,(\texttt{E\_Inr}\ e) = (\texttt{E\_Inr}\ e) \\
\forall e, x_\ell, e_\ell, x_r, e_r : \texttt{cst\_fold}\,(\texttt{E\_Case}\ e\, x_\ell\, e_\ell\, x_r\, e_r) = \\
\quad (\exists e'.e = \texttt{E\_Inl}\ e') \rightarrow \texttt{E\_Let}\ x_\ell\,(\texttt{child}_1\, e)\, e_\ell \\
\quad \mid ((\exists e'.e = \texttt{E\_Inr}\ e') \rightarrow \texttt{E\_Let}\ x_r\,(\texttt{child}_1\, e)\, e_r \\
\quad\quad \mid (\texttt{E\_Case}\ e\, x_\ell\, e_\ell\, x_r\, e_r)) \\
\forall x : \texttt{cst\_fold}\,(\texttt{E\_Var}\ x) = (\texttt{E\_Var}\ x) \\
\texttt{cst\_fold}\ \texttt{E\_Error} = \texttt{E\_Error} \\
\forall f, e : \texttt{cst\_fold}\,(\texttt{E\_Call}\ f\, e) = (\texttt{E\_Call}\ f\, e) \\
\forall x, e_1, e_2 : \texttt{cst\_fold}\,(\texttt{E\_Let}\ x\, e_1\, e_2) = (\texttt{E\_Let}\ x\, e_1\, e_2)
\end{array} \right\}
$$

The following theorem states that an expression is equivalent to its constant folding in all contexts:

Theorem 6.2 (Constant Fold Local Improvement.)

$$
\vdash \forall e, P, V : \left(\begin{array}{l} \texttt{local\_improvement}\ V\, P\, e\,(\texttt{cst\_fold}\ e) \wedge \\ \texttt{local\_improvement}\ V\, P\,(\texttt{cst\_fold}\ e)\, e \end{array} \right)
$$

Proof: Structural case analysis on the expression followed by expansion of calls to `cst_fold` and conditional case analysis leaves 17 subgoals, 12 of which are trivially handled by reflexivity of `local_improvement`. The remaining five goals, corresponding to the five cases where the expressions are transformed, are rewritten with respect to Lemma 4.62 and thereafter handled in the following ways.

Case `E_Op`**:** Handled by Lemma 4.49.

Cases `E_Fst` **and** `E_Snd`**:** Handled by Lemma 4.44.

Cases `E_Case` **[left,right]:** Handled by reduction. □

The above local improvement theorem generalises nicely to a global theorem.

Theorem 6.3 (Constant Fold Correctness Theorem.)

$$\vdash \forall e, P_1, P_2, V : \left(\begin{array}{l} \texttt{repl\_prg}\, V\, P_1\, P_2\, e\, (\texttt{cst\_fold}\, e) \Rightarrow \\ (\texttt{improvement}\, P_1\, P_2 \wedge \texttt{improvement}\, P_2\, P_1) \end{array} \right)$$

Proof: By Theorems 6.2 and 4.72. □

Constant folding does not introduce extra free variables in a term nor does it introduce extra function calls. Static correctness is therefore preserved.

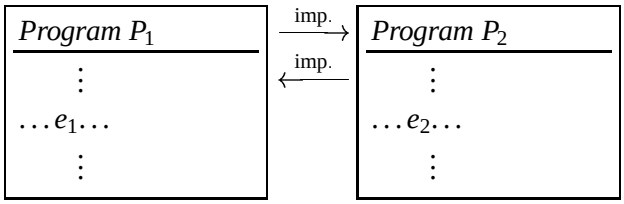
Theorem 6.4 (Constant Fold Preserves Static Correctness.)

$$\vdash \forall e, P_1, P_2, V : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, e\, (\texttt{cst\_fold}\, e) \Rightarrow \\ \texttt{stat\_ok}\, P_2 \end{array} \right)$$

Proof: By Theorem 4.91. □

6.2 Alpha Conversion

Programs contain variable names and from time to time it is convenient to be able to change names. For example, as explained in Section 4.7, the particular instance of substitution used here will rename all bound variables of the term being substituted into names generated by `genvar`. In examples, we do not want to be too specific on what these names are, since the only thing we want to know about `genvar` is Theorem 4.30. Following substitution up with changing the names to something well-chosen makes examples significantly more readable.



(provided `alpha` $e_1\, e_2$ *holds)*

Just as with the definition of substitution it turns out that formalising the definition of alpha convertibility is non-trivial because names are explicit. But the following definition works by defining that two expressions are alpha convertible if they have the same structure and if we can do renaming in the second making it identical to the first. Note that the definition is asymmetrical.[1]

Definitional Theorem 6.5 (Alpha Equivalence.)

$$
\vdash \left\{ \begin{array}{l}
\forall i, e' : \texttt{alpha}\,(\texttt{E\_Int}\ i)\ e' = (e' = \texttt{E\_Int}\ i) \\
\forall o, e_1, e_2, e' : \texttt{alpha}\,(\texttt{E\_Op}\ o\, e_1\, e_2)\ e' = \\
\quad \exists e_1', e_2' : (e' = \texttt{E\_Op}\ o\, e_1'\, e_2') \wedge \texttt{alpha}\ e_1\, e_1' \wedge \texttt{alpha}\ e_2\, e_2' \\
\qquad\qquad \vdots \\
\forall x, e' : \texttt{alpha}\,(\texttt{E\_Var}\ x)\ e' = (e' = \texttt{E\_Var}\ x) \\
\forall e_b, x_\ell, e_\ell, x_r, e_r, e' : \texttt{alpha}\,(\texttt{E\_Case}\ e_b\, x_\ell\, e_\ell\, x_r\, e_r)\ e' = \\
\quad \exists e_b', x_\ell', e_\ell', x_r', e_r' : (e' = \texttt{E\_Case}\ e_b'\, x_\ell'\, e_\ell'\, x_r'\, e_r') \wedge \texttt{alpha}\ e_b\, e_b' \wedge \\
\qquad \texttt{alpha}\ e_\ell\,(\texttt{rename\_free}\ x_\ell'\, x_\ell\,(\texttt{rename\_bound}\,(\texttt{E\_Var}\ x_\ell)\ e_\ell')) \wedge \\
\qquad ((x_\ell = x_\ell') \vee \neg(\texttt{var\_free}\ x_\ell\, e_\ell')) \wedge \\
\qquad \texttt{alpha}\ e_r\,(\texttt{rename\_free}\ x_r'\, x_r\,(\texttt{rename\_bound}\,(\texttt{E\_Var}\ x_r)\ e_r')) \wedge \\
\qquad ((x_r = x_r') \vee \neg(\texttt{var\_free}\ x_r\, e_r')) \wedge \\
\qquad\qquad \vdots \\
\forall x, e_1, e_2, e' : \texttt{alpha}\,(\texttt{E\_Let}\ x\, e_1\, e_2)\ e' = \\
\quad \exists x', e_1', e_2' : (e' = \texttt{E\_Let}\ x'\, e_1'\, e_2') \wedge \texttt{alpha}\ e_1\, e_1' \wedge \\
\qquad \texttt{alpha}\ e_2\,(\texttt{rename\_free}\ x'\, x\,(\texttt{rename\_bound}\,(\texttt{E\_Var}\ x)\ e_2')) \wedge \\
\qquad ((x = x') \vee \neg(\texttt{var\_free}\ x\, e_2'))
\end{array} \right\}
$$

Note that this definition of alpha equivalence is solely a definition for expressions and not for programs. We will thus not be able to use this to change the name of a formal parameter of a function. This quirk turns out to have no significance. (The obvious theorems about such renaming hold and some of them have even been proven. There is, however, no use for them.)

We will now prove that alpha equivalent expressions have identical evaluation properties.

Lemma 6.6 (Alpha Equivalent Terms Evaluate Identically.)

$$
\vdash \forall N, e_a, e_b, P, E, v : \texttt{alpha}\ e_a\, e_b \Rightarrow \left(\begin{array}{l} \texttt{eval\_expr\_n}\ N\, e_a\, P\, E\, v \Leftrightarrow \\ \texttt{eval\_expr\_n}\ N\, e_b\, P\, E\, v \end{array} \right)
$$

[1] We shall call two expressions for which `alpha` $e_1\, e_2$ holds *alpha equivalent* even though we will not prove that the relation is an equivalence relation. We only the need the property that such two terms have identical evaluation properties.

Proof: By structural induction on e_a. The non-trivial cases, `let` and `case` as usual, exactly match Lemma 4.47. □

Given this lemma it is straightforward to prove the corresponding local and global improvement theorems.

Theorem 6.7 (Alpha Equivalence Locally Improves Two-Way.)

$$\vdash \forall e_1, e_2, P, V : \left(\begin{array}{l} \texttt{alpha}\, e_1\, e_2 \Rightarrow \\ \texttt{local\_improvement}\, V\, P\, e_1\, e_2 \wedge \texttt{local\_improvement}\, V\, P\, e_2\, e_1 \end{array} \right)$$

Proof: By Lemmas 4.62 and 6.6. □

Theorem 6.8 (Alpha Equivalence Globally Improves Two-Way.)

$$\vdash \forall e_1, e_2, P_1, P_2, V : \left(\begin{array}{l} \texttt{alpha}\, e_1\, e_2 \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, e_1\, e_2 \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \wedge \texttt{improvement}\, P_2\, P_1 \end{array} \right)$$

Proof: By Theorems 4.72 and 6.7. □

Not surprisingly, alpha equivalent expressions have identical static correctness properties.

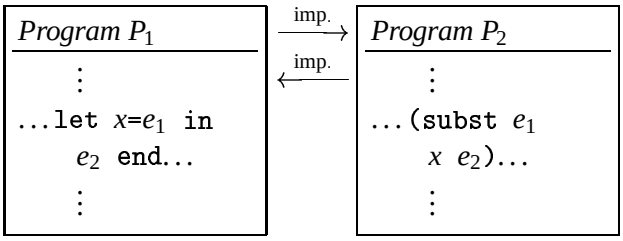
Theorem 6.9 (Alpha Equivalence Preserves Static Correctness.)

$$\vdash \forall P_1, P_2, e_1, e_2, V : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{alpha}\, e_1\, e_2 \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, e_1\, e_2 \Rightarrow \\ \texttt{stat\_ok}\, P_2 \end{array} \right)$$

Proof: By Theorem 4.91. (Proving the preconditions of Theorem 4.91 is not difficult.) □

6.3 Unfolding of Let-Bindings

We now turn to unfolding of `let` constructs. These might, for example, be produced by the constant folding explained previously. We shall eventually prove that under certain conditions $(\texttt{E\_Let}\, x\, e_1\, e_2)$ has the same meaning as $(\texttt{subst}\, e_1\, x\, e_2)$.



(certain conditions on e_1 and e_2 apply)

Recall, however, from Section 4.7 that substitution is defined in terms of renaming of bound variables and raw substitution. We shall therefore first prove lemmas telling us how `rawsubst` influences evaluation. Since raw substitution was not designed to handle variable capture (and since specifying the complicated effect when variable capture does take place is pointless) we set sufficient (but not necessary) preconditions to ensure this.

Lemma 6.10

$$\vdash \forall e_2, e_1, P, E, v, v', x, N, N' : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N'\, e_1\, P\, E\, v' \Rightarrow \\ \neg(\texttt{var\_bound}\, x\, e_2) \Rightarrow \\ (\forall x' : \texttt{var\_bound}\, x'\, e_2 \Rightarrow \neg(\texttt{var\_used}\, x'\, e_1)) \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, (\texttt{rawsubst}\, e_1\, x\, e_2)\, P\, E\, v \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, e_2\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \end{array} \right)$$

Proof: By structural induction on e_2. Non-trivial cases are for `let` and `case` which add to the environment in front of the (x, v') pair. These cases are handled by Lemmas 4.40 and 4.43. □

Note that the value of N' did not matter in the previous lemma because `rawsubst` will always place e_1 at the same or at a deeper level. (This is quite specific to the nested-call timed evaluation, which treats the argument of a function call the same as the body.) For the same reason, when we want to prove the opposite implication we have to require that $N' = 0$ so it does not matter how deep e_2 is placed.

Lemma 6.11

$$\vdash \forall e_2, e_1, P, E, v, v', x, N : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, 0\, e_1\, P\, E\, v' \Rightarrow \\ \neg(\texttt{var\_bound}\, x\, e_2) \Rightarrow \\ (\forall x' : \texttt{var\_bound}\, x'\, e_2 \Rightarrow \neg(\texttt{var\_used}\, x'\, e_1)) \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, e_2\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, (\texttt{rawsubst}\, e_1\, x\, e_2)\, P\, E\, v \end{array} \right)$$

Proof: By structural induction on e_2. Non-trivial cases are for `let` and `case` which add to the environment in front of the (x, v') pair. These cases are handled by Lemmas 4.40 and 4.43. □

We can now combine the previous two lemmas into the following, which tells us the effect of `subst`. Note that this lemma inherits the restrictive zero-depth requirement of Lemma 6.11.

Lemma 6.12

$$\vdash \forall e_1, e_2, E, v, v', x, P, N : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, 0\, e_1\, P\, E\, v' \Rightarrow \\ \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_2\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \\ \Leftrightarrow \\ \texttt{eval\_expr\_n}\, N\, (\texttt{subst}\, e_1\, x\, e_2)\, P\, E\, v \end{array} \right) \end{array} \right)$$

Proof: By rewriting with the definition of `subst`. Then "$\Rightarrow$" by Lemmas 6.11 and 4.46 and "$\Leftarrow$" by Lemmas 6.10 and 4.46. □

Given that we now know how `subst` works and that harmless (in the sense of Theorem 4.36) terms always evaluate it is a simple matter to prove the following two theorems connecting `let`-binding and substitution.

Theorem 6.13 (Unfolding Harmless Lets Locally Improves Two-Way.)

$$\vdash \forall e_1, e_2, V, x, P : \left(\begin{array}{l} \texttt{harmless}\, V\, e_1 \Rightarrow \\ \texttt{local\_improvement}\, V\, P\, (\texttt{E\_Let}\, x\, e_1\, e_2)\, (\texttt{subst}\, e_1\, x\, e_2) \wedge \\ \texttt{local\_improvement}\, V\, P\, (\texttt{subst}\, e_1\, x\, e_2)\, (\texttt{E\_Let}\, x\, e_1\, e_2) \end{array} \right)$$

Proof: By rewriting with Lemma 4.62 and reducing. At this point Lemma 4.37 shows that e_1 evaluates to some value and the theorem follows from 6.12. □

Theorem 6.14 (Unfolding Harmless Lets Globally Improves Two-Way.)

$$\vdash \forall e_1, e_2, x, P_1, P_2, V : \left(\begin{array}{l} \texttt{harmless}\, V\, e_1 \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, (\texttt{E\_Let}\, x\, e_1\, e_2)\, (\texttt{subst}\, e_1\, x\, e_2) \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \wedge \texttt{improvement}\, P_2\, P_1 \end{array} \right)$$

Proof: By Theorems 4.72 and 6.13. □

Finally we note that unfolding of `let` preserves static correctness.

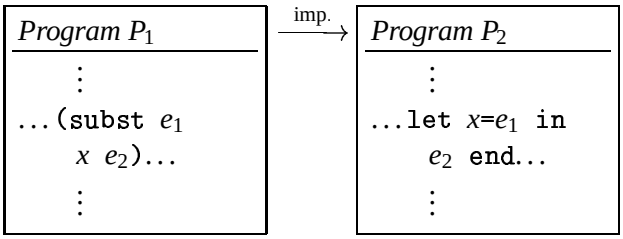
Theorem 6.15 (Unfolding Harmless Lets Preserves Static Correctness.)

$$\vdash \forall P_1, P_2, V, x, e_1, e_2 : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, (\texttt{E\_Let}\, x\, e_1\, e_2)\, (\texttt{subst}\, e_1\, x\, e_2) \Rightarrow \\ \texttt{stat\_ok}\, P_2 \end{array} \right)$$

Proof: By Theorem 4.91. (Proving the preconditions of Theorem 4.91 is not difficult.) □

6.4 Folding of Let-Bindings

Folding of `let`-bindings is the reverse operation of unfolding: instead of replacing free occurrences of a variable by a term we replace occurrences of a term by a free variable and wrap with a `let`-binding.



(certain conditions on e_1 and e_2 apply)

Observe from this diagram that we must look for occurrences of $e_1[e_2/x]$ (or `subst` $e_2\,x\,e_1$ in HOL-parlance). Such a term will have all its bound variables generated by `genvar` so it is likely that any use of this program transformation will be immediately preceded by an alpha conversion transformation.

The `let`-fold program transformation is not always valid. Consider, for example the following program.

```
f x = 2;
loop y = loop y;
```

We should not be able to transform this program into

```
f x = let z=loop 1 in 2 end;
loop y = loop y;
```

by the folding transformation because we would have changed termination properties this way. The solution, as discussed in Section 4.9, is to require that the newly bound variable be used strictly inside the body of the new `let`.

Under this assumption the proof of correctness works approximately as the proof of correctness for unfolding of `let`-bindings. Again, since `subst` is defined in terms of `rawsubst` we need a technical lemma for that.

Lemma 6.16

$$\vdash \forall e_2, e_1, P, E, v, x, N : \left(\begin{array}{l} (\forall x' : \texttt{var\_bound}\, x'\, e_2 \Rightarrow \neg(\texttt{var\_used}\, x'\, e_1)) \Rightarrow \\ \texttt{used\_strictly}\, x\, e_2 \Rightarrow \\ \forall v' : \neg(\texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v') \Rightarrow \\ \neg(\texttt{var\_bound}\, x\, e_2) \Rightarrow \\ \neg(\texttt{eval\_expr\_n}\, N\, (\texttt{rawsubst}\, e_1\, x\, e_2)\, P\, E\, v) \end{array} \right)$$

Proof: By structural induction on e_2. Non-trivial cases are for `let` and `case` which add to the environment in front of the (x, v') pair. These cases are handled by Lemma 4.43. □

Lemma 6.17

$$\vdash \forall e_1, e_2, E, v, v', x, P, N, N' : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N'\, e_1\, P\, E\, v' \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, (\texttt{subst}\, e_1\, x\, e_2)\, P\, E\, v \Rightarrow \\ \texttt{eval\_expr\_n}\, N\, e_2\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \end{array} \right)$$

Proof: By the definition of `subst` and Lemmas 6.10 and 4.46. □

The following two lemmas correspond to Lemma 6.11 and "⇒" from Lemma 6.12 except that they use `eval_expr` instead of `eval_expr_n` and therefore need not limit the evaluation resources available to the bound expression, e_2.

Lemma 6.18

$$\vdash \forall e_2, e_1, P, E, v, v', x : \left(\begin{array}{l} \texttt{eval\_expr}\, e_1\, P\, E\, v' \Rightarrow \\ \neg(\texttt{var\_bound}\, x\, e_2) \Rightarrow \\ (\forall x' : \texttt{var\_bound}\, x'\, e_2 \Rightarrow \neg(\texttt{var\_used}\, x'\, e_1)) \Rightarrow \\ \texttt{eval\_expr}\, e_2\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \Rightarrow \\ \texttt{eval\_expr}\, (\texttt{rawsubst}\, e_1\, x\, e_2)\, P\, E\, v \end{array} \right)$$

Proof: By structural induction on e_2. Non-trivial cases are for `let` and `case` which add to the environment in front of the (x, v') pair. These cases are handled by Lemmas 4.40 and 4.43. □

Lemma 6.19

$$\vdash \forall e_1, e_2, P, E, v, v', x : \left(\begin{array}{l} \texttt{eval\_expr}\, e_1\, P\, E\, v' \Rightarrow \\ \texttt{eval\_expr}\, e_2\, P\, (\texttt{CONS}\, (x, v')\, E)\, v \Rightarrow \\ \texttt{eval\_expr}\, (\texttt{subst}\, e_1\, x\, e_2)\, P\, E\, v \end{array} \right)$$

Proof: By rewriting with the definition of `subst`. Then the lemma follows by Lemmas 6.18 and 4.46. □

We are now ready to prove that the folding of `let`-bindings improve an expression locally. Since the newly bound expression might have occurred deeply inside function arguments in the original term we cannot hope to have improvement the other way.

Theorem 6.20 (Let Folding Improves Locally.)

$$\vdash \forall P,V,x,e_1,e_2 : \left(\begin{array}{l} \texttt{used\_strictly}\, x\, e_2 \Rightarrow \\ \texttt{local\_improvement}\, V\, P\, (\texttt{subst}\, e_1\, x\, e_2)\, (\texttt{E\_Let}\, x\, e_1\, e_2) \end{array} \right)$$

Proof: We rewrite by Theorem 4.54 and then need to prove the two conjuncts in the definition of `local_improvement`. The `eval_expr` conjunct follows by Lemma 6.19. The `eval_expr_n` is more difficult and we spilt into two cases: If e_1 terminates (i.e., $\exists v : \texttt{eval\_expr\_n}\, N\, e_1\, P\, E\, v$ using the variable names of Theorem 4.54) then Lemma 6.17 solves the problem. Otherwise, Lemma 6.16 solves the problem. □

The above theorem generalises nicely to a global improvement theorem.

Theorem 6.21 (Let Folding Improves Globally.)

$$\vdash \forall P_1,P_2,e_1,e_2,x,V : \left(\begin{array}{l} \texttt{used\_strictly}\, x\, e_2 \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, (\texttt{subst}\, e_1\, x\, e_2)\, (\texttt{E\_Let}\, x\, e_1\, e_2) \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \end{array} \right)$$

Proof: By Theorems 4.71 and 6.20. □

Finally, the folding of `let`-bindings preserves static correctness.

Theorem 6.22 (Let Folding Preserves Static Correctness.)

$$\vdash \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{used\_strictly}\, x\, e_2 \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, (\texttt{subst}\, e_1\, x\, e_2)\, (\texttt{E\_Let}\, x\, e_1\, e_2) \Rightarrow \\ \texttt{stat\_ok}\, P_2 \end{array} \right)$$

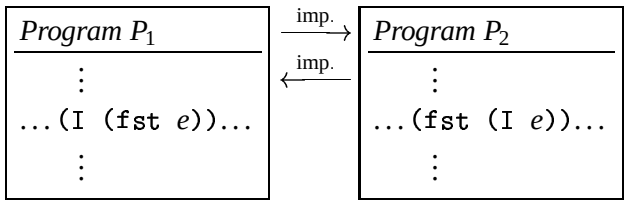
Proof: By Theorem 4.91. (Proving the preconditions of Theorem 4.91 is not difficult.) □

* * *

Combining our knowledge of `let`-unfolding and `let`-folding we see that if we do a `let`-folding that creates a harmless binding then, by Theorems 6.13, 6.14, and 6.22 we have a program transformation that two-way improves and preserves static correctness.

6.5 Identity-Function Moving

The language's semantics is defined in such a way that it allows us to move calls to identity functions around with respect to some of the language's constructs while preserving both static correctness properties and timed evaluation properties.



(example for `fst`*)*

More precisely we have the following theorems.

$$
\begin{array}{rcl}
\texttt{E\_Call}\, I\, (\texttt{E\_Op}\, o\, e_1\, e_2) & \leftrightarrow & \texttt{E\_Op}\, o\, (\texttt{E\_Call}\, I\, e_1)\, (\texttt{E\_Call}\, I\, e_2) \\
\texttt{E\_Call}\, I\, (\texttt{E\_Pair}\, e_1\, e_2) & \leftrightarrow & \texttt{E\_Pair}\, (\texttt{E\_Call}\, I\, e_1)\, (\texttt{E\_Call}\, I\, e_2) \\
\texttt{E\_Call}\, I\, (\texttt{E\_Fst}\, e) & \leftrightarrow & \texttt{E\_Fst}\, (\texttt{E\_Call}\, I\, e) \\
\texttt{E\_Call}\, I\, (\texttt{E\_Snd}\, e) & \leftrightarrow & \texttt{E\_Snd}\, (\texttt{E\_Call}\, I\, e) \\
\texttt{E\_Call}\, I\, (\texttt{E\_Inl}\, e) & \leftrightarrow & \texttt{E\_Inl}\, (\texttt{E\_Call}\, I\, e) \\
\texttt{E\_Call}\, I\, (\texttt{E\_Inr}\, e) & \leftrightarrow & \texttt{E\_Inr}\, (\texttt{E\_Call}\, I\, e) \\
\texttt{E\_Call}\, I\, (\texttt{E\_Let}\, x\, e_1\, e_2) & \leftrightarrow & \texttt{E\_Let}\, x\, (\texttt{E\_Call}\, I\, e_1)\, (\texttt{E\_Call}\, I\, e_2) \\
\texttt{E\_Call}\, I\, (\texttt{E\_Case}\, e\, x_\ell\, e_\ell\, x_r\, e_r) & \leftrightarrow & \left\{ \begin{array}{l} \texttt{E\_Case}\, (\texttt{E\_Call}\, I\, e) \\ \qquad x_\ell\, (\texttt{E\_Call}\, I\, e_\ell)\, x_r\, (\texttt{E\_Call}\, I\, e_r) \end{array} \right. \\
\texttt{E\_Call}\, f\, e & \leftrightarrow & \texttt{E\_Let}\, x\, (\texttt{E\_Call}\, I\, e)\, (\texttt{E\_Call}\, f\, (\texttt{E\_Var}\, x))
\end{array}
$$

Figure 14: Identity-Function Moves Equivalences.

Theorem 6.23 (Identity-Function Moves Locally Two-Way Improve.) *Let X and Y be any of the expression pairs from Figure 14. Then*

$$
\vdash \forall P, V, \ldots : \left(\begin{array}{l} \texttt{id\_func}\, P_1\, I \Rightarrow \\ (\texttt{local\_improvement}\, V\, P\, X\, Y \wedge \texttt{local\_improvement}\, V\, P\, Y\, X) \end{array} \right)
$$

where "…" represents the free variables of X and Y.

Proof: Using Lemma 4.82 makes the theorem symmetric in the two program arguments, reducing the problem to proving one conjunct. The first conjunct follows by expanding `local_improvement`'s definition and using Lemma 4.80. □

Theorem 6.24 (Identity-Function Moves Globally Two-Way Improve.) *Let X and Y be any of the expression pairs from Figure 14. Then*

$$\forall P_1, P_2, V, \ldots : \left(\begin{array}{l} \texttt{id\_func}\ P_1\ I \Rightarrow \\ \texttt{repl\_prg}\ V\ P_1\ P_2\ X\ Y \Rightarrow \\ \texttt{improvement}\ P_1\ P_2 \wedge \texttt{improvement}\ P_2\ P_1 \end{array} \right)$$

where "…" represents the free variables of X and Y.

Proof: By Theorems 6.23 and 4.72. □

Theorem 6.25 (Identity-Function Moves Preserve Static Correctness.) *Let X and Y be any of the expression pairs from Figure 14. Then*

$$\vdash \forall P_1, P_2, V, \ldots : \texttt{repl\_prg}\ V\ P_1\ P_2\ X\ Y \Rightarrow (\texttt{stat\_ok}\ P_1 \Leftrightarrow \texttt{stat\_ok}\ P_2)$$

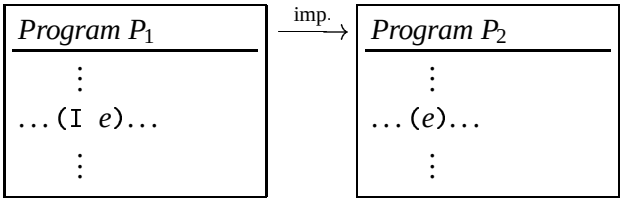
where "…" represents the free variables of X and Y.

Proof: Easy: "⇒" follows from Theorem 4.91; "⇐" follows from Lemma 4.23 and Theorem 4.91. □

Note that the program transformations as described here use the same identity function before and after substitutions. We could just as easily have used two potentially different ones, if we wanted to apply an identity function to arguments of different type.

6.6 Identity-Function Elimination

Any call to an identity function can be eliminated and will improve the program and preserve static correctness. Improvement is generally one-way only.



Theorem 6.26 (Identity-Function Elimination Improves Locally.)

$$\vdash \forall P, V, I, e : \texttt{id\_func}\ P\ I \Rightarrow \texttt{local\_improvement}\ V\ P\ (\texttt{E\_Call}\ I\ e)\ e$$

Proof: By expanding `local_improvement`'s definition and using Lemma 4.80. □

Theorem 6.27 (Identity-Function Elimination Improves Globally.)

$$\vdash \forall P_1, P_2, V, I, e : \left(\begin{array}{l} \texttt{id\_func}\, P_1\, I \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, (\texttt{E\_Call}\, I\, e)\, e \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \end{array} \right)$$

Proof: By Theorems 6.26 and 4.71. □

Theorem 6.28 (Identity-Function Elimination Preserves Static Correctness)

$$\vdash \forall P_1, P_2, V, I, e : \left(\begin{array}{l} \texttt{id\_func}\, P_1\, I \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2\, (\texttt{E\_Call}\, I\, e)\, e \Rightarrow \\ (\texttt{stat\_ok}\, P_1 \Leftrightarrow \texttt{stat\_ok}\, P_2) \end{array} \right)$$

Proof: Lemmas 4.82 and 4.23 produce extra assumptions such that Lemma 4.91 can be used to prove both implications. □

6.7 Argument Manœuvres

In this section we shall describe program transformations designed to put a function call into the right form for folding as covered in Section 6.9. Assume that we have a call

```
...(g (e_s,e_d))...
```

in a program fragment

```
f x = g (e_s,x);
g y = ...
h z = ... (* in here, for example *) ...
```

In other words, `f` is a specialised version of `g` and the call is an instance of `f`'s body. We should therefore eventually be able to transform the call of `g` into a call to `f`. As one condition of using folding in the way we shall later define it, we must transform the call into

```
...let x=e_d in g (e_s,x) end...
```

i.e., we must make the match of the call and the body of the function literal, not just through instantiation.

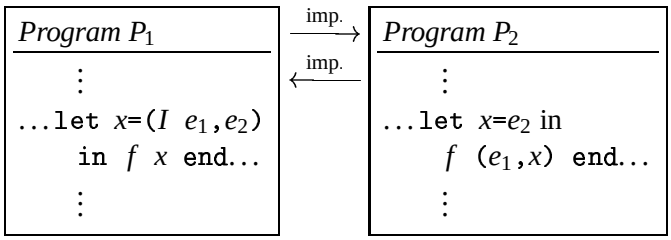
The first steps of making this transformation is covered in Section 6.5, by which we can transform the call into

```
...let x=I (e_s,e_d) in g x end...
```

and then into

```
...let x=(I e_s,I e_d) in g x end...
```

We shall now prove theorems that allow us to transform expressions like the latter into the expression stated above. Unfortunately, the theorems are not quite as general as we would want them to be; the structure of a call with a pair as argument is built into the theorems.



(first argument move)

The theorems would therefore not be very useful for functions with three or more parameters. Since such theorems would be proven in the same easy way we shall ignore this generality problem.

Theorem 6.29 (First Argument Move Locally Two-Way Improves.)

$$\vdash \forall P,V,I,f,x,e_1,e_2 : \left(\begin{array}{l} \texttt{id\_func}\ P\ I \Rightarrow \\ \neg(\texttt{var\_free}\ x\ e_1) \Rightarrow \\ \texttt{local\_improvement}\ V\ P \\ \quad (\texttt{E\_Let}\ x\ (\texttt{E\_Pair}\ (\texttt{E\_Call}\ I\ e_1)\ e_2) \\ \qquad\quad (\texttt{E\_Call}\ f\ (\texttt{E\_Var}\ x))) \\ \quad (\texttt{E\_Let}\ x\ e_2\ (\texttt{E\_Call}\ f\ (\texttt{E\_Pair}\ e_1\ (\texttt{E\_Var}\ x)))) \wedge \\ \texttt{local\_improvement}\ V\ P \\ \quad (\texttt{E\_Let}\ x\ e_2\ (\texttt{E\_Call}\ f\ (\texttt{E\_Pair}\ e_1\ (\texttt{E\_Var}\ x)))) \\ \quad (\texttt{E\_Let}\ x\ (\texttt{E\_Pair}\ (\texttt{E\_Call}\ I\ e_1)\ e_2) \\ \qquad\quad (\texttt{E\_Call}\ f\ (\texttt{E\_Var}\ x))) \end{array} \right)$$

Proof: By reduction with Lemmas 4.62, 4.80, and 4.42. □

Theorem 6.30 (Second Argument Move Locally Two-Way Improves.)

$$\vdash \forall P,V,I,f,x,e_1,e_2 : \left(\begin{array}{l} \texttt{id\_func}\, P\, I \Rightarrow \\ \neg(\texttt{var\_free}\, x\, e_2) \Rightarrow \\ \texttt{local\_improvement}\, V\, P \\ \quad (\texttt{E\_Let}\, x\, (\texttt{E\_Pair}\, e_1\, (\texttt{E\_Call}\, I\, e_2)) \\ \quad\quad (\texttt{E\_Call}\, f\, (\texttt{E\_Var}\, x))) \\ \quad (\texttt{E\_Let}\, x\, e_1\, (\texttt{E\_Call}\, f\, (\texttt{E\_Pair}\, (\texttt{E\_Var}\, x)\, e_2))) \wedge \\ \texttt{local\_improvement}\, V\, P \\ \quad (\texttt{E\_Let}\, x\, e_1\, (\texttt{E\_Call}\, f\, (\texttt{E\_Pair}\, (\texttt{E\_Var}\, x)\, e_2))) \\ \quad (\texttt{E\_Let}\, x\, (\texttt{E\_Pair}\, e_1\, (\texttt{E\_Call}\, I\, e_2)) \\ \quad\quad (\texttt{E\_Call}\, f\, (\texttt{E\_Var}\, x))) \end{array} \right)$$

Proof: By reduction with Lemmas 4.62, 4.80, and 4.42. □

Theorem 6.31 (First Argument Move Globally Two-Way Improves.)

$$\vdash \forall P_1,P_2,V,I,f,x,e_1,e_2 : \left(\begin{array}{l} \texttt{id\_func}\, P\, I \Rightarrow \\ \neg(\texttt{var\_free}\, x\, e_2) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2 \\ \quad (\texttt{E\_Let}\, x\, (\texttt{E\_Pair}\, (\texttt{E\_Call}\, I\, e_1)\, e_2) \\ \quad\quad (\texttt{E\_Call}\, f\, (\texttt{E\_Var}\, x))) \\ \quad (\texttt{E\_Let}\, x\, e_2\, (\texttt{E\_Call}\, f\, (\texttt{E\_Pair}\, e_1\, (\texttt{E\_Var}\, x)))) \\ \Rightarrow \\ (\texttt{improvement}\, P_1\, P_2 \wedge \texttt{improvement}\, P_2\, P_1) \end{array} \right)$$

Proof: By Theorems 4.72 and 6.29. □

Theorem 6.32 (Second Argument Move Globally Two-Way Improves.)

$$\vdash \forall P_1,P_2,V,I,f,x,e_1,e_2 : \left(\begin{array}{l} \texttt{id\_func}\, P\, I \Rightarrow \\ \neg(\texttt{var\_free}\, x\, e_2) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2 \\ \quad (\texttt{E\_Let}\, x\, (\texttt{E\_Pair}\, e_1\, (\texttt{E\_Call}\, I\, e_2)) \\ \quad\quad (\texttt{E\_Call}\, f\, (\texttt{E\_Var}\, x))) \\ \quad (\texttt{E\_Let}\, x\, e_1\, (\texttt{E\_Call}\, f\, (\texttt{E\_Pair}\, (\texttt{E\_Var}\, x)\, e_2))) \\ \Rightarrow \\ (\texttt{improvement}\, P_1\, P_2 \wedge \texttt{improvement}\, P_2\, P_1) \end{array} \right)$$

Proof: By Theorems 4.72 and 6.30. □

Theorem 6.33 (First Argument Move Preserves Static Correctness.)

$$\vdash \forall P_1, P_2, V, I, f, x, e_1, e_2 : \left(\begin{array}{l} \texttt{id\_func}\, P\, I \Rightarrow \\ \neg(\texttt{var\_free}\, x\, e_2) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2 \\ \quad (\texttt{E\_Let}\, x\, (\texttt{E\_Pair}\, (\texttt{E\_Call}\, I\, e_1)\, e_2) \\ \qquad\quad (\texttt{E\_Call}\, f\, (\texttt{E\_Var}\, x))) \\ \quad (\texttt{E\_Let}\, x\, e_2\, (\texttt{E\_Call}\, f\, (\texttt{E\_Pair}\, e_1\, (\texttt{E\_Var}\, x)))) \\ \Rightarrow \\ (\texttt{stat\_ok}\, P_1 \Leftrightarrow \texttt{stat\_ok}\, P_2) \end{array} \right)$$

Proof: "⇒" by Theorem 4.91. "⇐" by Theorems 4.91 and 4.23 plus Lemma 4.82. □

Theorem 6.34 (Second Argument Move Preserves Static Correctness.)

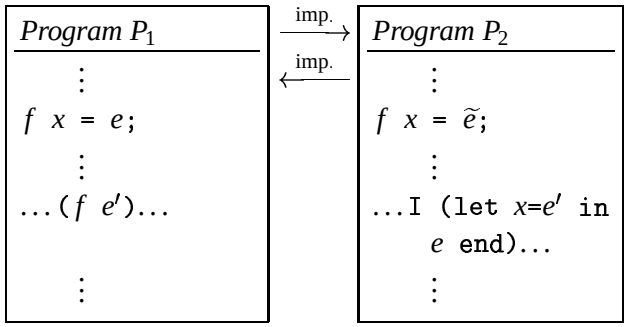
$$\vdash \forall P_1, P_2, V, I, f, x, e_1, e_2 : \left(\begin{array}{l} \texttt{id\_func}\, P\, I \Rightarrow \\ \neg(\texttt{var\_free}\, x\, e_2) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2 \\ \quad (\texttt{E\_Let}\, x\, (\texttt{E\_Pair}\, e_1\, (\texttt{E\_Call}\, I\, e_2)) \\ \qquad\quad (\texttt{E\_Call}\, f\, (\texttt{E\_Var}\, x))) \\ \quad (\texttt{E\_Let}\, x\, e_1\, (\texttt{E\_Call}\, f\, (\texttt{E\_Pair}\, (\texttt{E\_Var}\, x)\, e_2))) \\ \Rightarrow \\ (\texttt{stat\_ok}\, P_1 \Leftrightarrow \texttt{stat\_ok}\, P_2) \end{array} \right)$$

Proof: "⇒" by Theorem 4.91. "⇐" by Theorems 4.91 and 4.23 plus Lemma 4.82. □

6.8 Unfold Function Call

One of the most important program transformations we shall prove correct is the unfolding of function calls. In the form we shall discuss it here it is the replacement of $(\texttt{E\_Call}\ f\ e')$ by $(\texttt{E\_Call}\ I\ (\texttt{E\_Let}\ x\ e'\ e))$ where the program in question contains the definition $f\ \ x{=}e$; and where I is an identity function.

The identity function call as introduced this way does not influence regular evaluation. It does, however, influence timed evaluation and that in such a way that the replacement becomes a two-way improvement.



(The diagram uses $\widetilde{e}$ in P_2 because an instance of the unfolded call might occur inside e.)

Theorem 6.35 (Function Unfold Locally Two-Way Improves.)

$$\vdash \forall P, f, x, e', V, I : \left(\begin{array}{l} \texttt{id\_func}\ P\ I \Rightarrow \\ (\texttt{lookup\_func}\ P\ f = \texttt{OK}\ (x, e)) \Rightarrow \\ \left(\begin{array}{l} \texttt{local\_improvement}\ V\ P \\ \quad (\texttt{E\_Call}\ I\ (\texttt{E\_Let}\ x\ e'\ e)) \\ \quad (\texttt{E\_Call}\ f\ e') \wedge \\ \texttt{local\_improvement}\ V\ P \\ \quad (\texttt{E\_Call}\ f\ e') \\ \quad (\texttt{E\_Call}\ I\ (\texttt{E\_Let}\ x\ e'\ e)) \end{array} \right) \end{array} \right)$$

Proof: Rewriting by Lemma 4.62. Then reduction using Lemmas 4.80 and 4.93. □

The local improvement generalises nicely to the following global improvement theorem.

Theorem 6.36 (Function Unfold Globally Two-Way Improves.)

$$\vdash \forall P_1, P_2, V, I, f, x, e, e' : \left(\begin{array}{l} \texttt{id\_func}\ P_1\ I \Rightarrow \\ (\texttt{lookup\_func}\ P_1\ f = \texttt{OK}\ (x, e)) \Rightarrow \\ \texttt{repl\_prg}\ V\ P_1\ P_2 \\ \quad (\texttt{E\_Call}\ f\ e') \\ \quad (\texttt{E\_Call}\ I\ (\texttt{E\_Let}\ x\ e'\ e)) \Rightarrow \\ (\texttt{improvement}\ P_1\ P_2 \wedge \texttt{improvement}\ P_2\ P_1) \end{array} \right)$$

Proof: By Theorems 4.72 and 6.35. □

Note at this point that Theorem 6.36 can be used to prove the correctness of a limited form of folding also when considering the symmetry of `repl_prg` (i.e.,

Lemma 4.23). If we do that, we must however still lookup f in P_1 which would then be the target program and not the source program. (By Lemma 4.82 we have that I will automatically be an identity function in P_2 also, so that does not cause any problems.)

If we require that the body of the unfolded function come from a statically correct program then the replacement will produce a new statically correct program.

Theorem 6.37 (Function Unfold Preserves Static Correctness.)

$$\vdash \forall P_1, P_2, V, I, f, x, e, e' : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{id\_func}\, P_1\, I \Rightarrow \\ (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x, e)) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2 \\ \quad (\texttt{E\_Call}\ f\, e') \\ \quad (\texttt{E\_Call}\, I\, (\texttt{E\_Let}\, x\, e'\, e)) \Rightarrow \\ \texttt{stat\_ok}\, P_2 \end{array} \right)$$

Proof: By Theorem 4.91 and Lemma 4.92. □

Since we are sometimes able to use function unfolding backwards for folding we should also provide a theorem allowing us to prove static correctness in such situation. This turns out to be simpler than forward preservation.

Theorem 6.38 (Reverse Function Unfold Preserves Static Correctness.)

$$\vdash \forall P_1, P_2, V, I, f, x, e, e' : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{MEM}\ f\, (\texttt{functions}\, P_1) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_1\, P_2 \\ \quad (\texttt{E\_Call}\, I\, (\texttt{E\_Let}\, x\, e'\, e)) \\ \quad (\texttt{E\_Call}\ f\, e') \Rightarrow \\ \texttt{stat\_ok}\, P_2 \end{array} \right)$$

Proof: By Theorem 4.91. □

6.9 Invoking the History of a Function

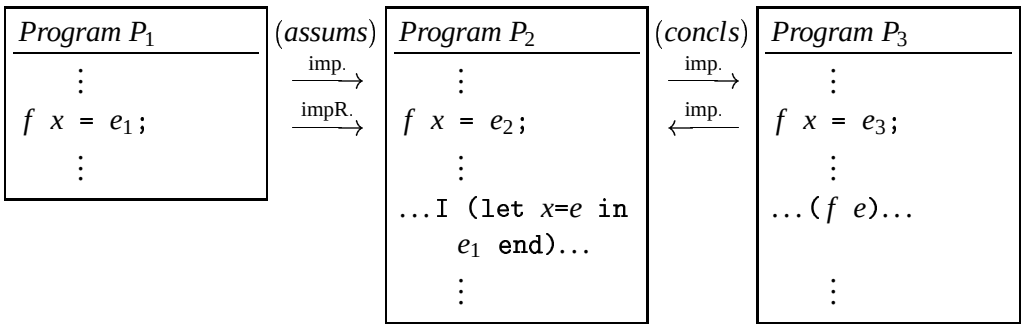
Folding is the opposite of unfolding: instead of replacing a call with the corresponding body we replace an occurrence of the body with a call. Most discussions on folding allow some kind of matching to take place in the search for occurrences but we shall restrict ourselves to literal occurrences wrapped in a `let`-binding and a call to an identity function as produced also by unfolding.[2] For example it is

[2]This is not a real restriction as other program transformations allow us to mutate the source in advance.

correct to replace the `I`-call in the following program by the call `(sqr (2+x))`.

```
main x = I (let y=2+x in y*y end);
sqr y = y*y;
```

To make this program transformation extra useful in our case we need to generalise it somewhat. Instead of looking for occurrences of `sqr`'s body in the current program — let us call that P_2 — we look for occurrences of `sqr`'s body in any program, P_1, which is in an `improvement`/`improvementR` relationship with P_2. This is exactly the relationship in which P_1 and P_2 would be if we had obtained P_2 from P_1 by the program transformations discussed in the present chapter, c.f. Lemma 4.82. In other words we reach back into `sqr`'s history and use the definition it had then. The following diagram outlines our folding transformation.



Note, that in this particular illustration P_2 is our source program and P_3 is our target program. P_1 is a program assumed to have an `improvement`/`improvementR` relationship with P_2. Note furthermore, that the diagram uses e_3 in P_3 because folding might take place inside e_2 itself. (In fact, if it does not then the situation is a special case of function *un*folding.) Note finally, that the diagram uses e_1 in P_1 because `f` in P_1 might have a body expression different from e_2.

The following theorems state that this kind of folding improves globally as well as locally (and in particular that it preserves evaluation and termination properties). First a necessary lemma.

When P_1 and P_2 are in an `improvement`/`improvementR`-relationship then any of P_1's function bodies evaluates just as fast relative to P_1 as the corresponding function body in P_2 does relative to P_2.

Lemma 6.39

$$\vdash \forall P_1, P_2, e_1, e_2, f, x, N, E, v : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{stat\_ok}\, P_2 \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \Rightarrow \\ \texttt{improvementR}\, P_1\, P_2 \Rightarrow \\ (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x, e_1)) \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x, e_2)) \Rightarrow \\ \texttt{vars\_in\_env}\,[x]\, E \Rightarrow \\ \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e_1\, P_1\, E\, v \Leftrightarrow \\ \texttt{eval\_expr\_n}\, N\, e_2\, P_2\, E\, v \end{array} \right) \end{array} \right)$$

Proof: Lemmas 4.93 and 4.39 reduce E to a single binding for x. Then the improvement conditions show the rest. □

Theorem 6.40 (History-Based Two-Way Local Improvement.)

$$\vdash \forall P_1, P_2, f, x, e_1, e_2 : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{stat\_ok}\, P_2 \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \Rightarrow \\ \texttt{improvementR}\, P_1\, P_2 \Rightarrow \\ (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x, e_1)) \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x, e_2)) \Rightarrow \\ \left(\begin{array}{l} \texttt{local\_improvement}\,[x]\, P_2\, e_1\, e_2 \wedge \\ \texttt{local\_improvement}\,[x]\, P_2\, e_2\, e_1 \end{array} \right) \end{array} \right)$$

Proof: First by rewriting with Lemma 4.62. Then Lemmas 4.92, 4.78, and 6.39 proves the theorem. □

Theorem 6.41 (History-Based Folding Globally Improves Two-Way.)

$$\vdash \forall P_1, P_2, P_3, f, x, e_1, e_2, e, I, V : \left(\begin{array}{l} \texttt{stat\_ok}\, P_1 \Rightarrow \\ \texttt{stat\_ok}\, P_2 \Rightarrow \\ \texttt{improvement}\, P_1\, P_2 \Rightarrow \\ \texttt{improvementR}\, P_1\, P_2 \Rightarrow \\ (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x, e_1)) \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x, e_2)) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_2\, P_3 \\ \qquad (\texttt{E\_Call}\, I\, (\texttt{E\_Let}\, x\, e\, e_1)) \\ \qquad (\texttt{E\_Call}\, f\, e) \Rightarrow \\ \texttt{id\_func}\, P_2\, I \Rightarrow \\ (\texttt{improvement}\, P_2\, P_3 \wedge \texttt{improvement}\, P_3\, P_2) \end{array} \right)$$

Proof: By Theorem 4.72 and Lemma 6.40 this is reduced to proving two-way local improvement between $(\texttt{E\_Call}\, I\, (\texttt{E\_Let}\, x\, e\, e_1))$ and $(\texttt{E\_Call}\, f\, e)$ under the given circumstances. This follows by Lemmas 4.93, 4.80, and 4.62. □

Finally, we note that this form of folding preserves static correctness.

Theorem 6.42 (History-Based Folding Preserves Static Correctness.)

$$\vdash \forall P_2, P_3, f, x, e_1, e_2, e, I, V : \left(\begin{array}{l} \texttt{stat\_ok}\, P_2 \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\, (x, e_2)) \Rightarrow \\ \texttt{repl\_prg}\, V\, P_2\, P_3 \\ \qquad (\texttt{E\_Call}\, I\, (\texttt{E\_Let}\, x\, e\, e_1)) \\ \qquad (\texttt{E\_Call}\, f\, e) \Rightarrow \\ \texttt{id\_func}\, P_2\, I \Rightarrow \\ \texttt{stat\_ok}\, P_3 \end{array} \right)$$

Proof: By Theorem 4.91. □

6.10 Removing Unused Functions

We now turn to one of the few of our program transformation that cannot be described by replacement. Unfortunately this means that powerful machinery like Theorem 4.71 cannot be used.

The program transformation we are interested in is one that removes unused functions. This is useful when, due to specialisation, some general functions are no longer needed. It should be obvious that this program transformation does not change the evaluation of a program but we must prove this anyway.

We will present a transformation that removes a set of functions from a program at once because removing just one might leave us with a program that is no longer statically correct. A correct subset extraction is one that leaves a statically correct program and does not change the first function.

Program P_1
$f_1\ x_1\ =\ e_1;$
$f_2\ x_2\ =\ e_2;$
$\vdots$
$f_{n-1}\ x_{n-1}\ =\ e_{n-1};$
$f_n\ x_n\ =\ e_n;$

$\xleftarrow{\text{imp.}}$
$\xleftarrow{\text{impR.}}$

Program P_2
$f_1\ x_1\ =\ e_1;$
$f_{i_1}\ x_{i_1}\ =\ e_{i_1};$
$\vdots$
$f_{i_k}\ x_{i_k}\ =\ e_{i_k};$

To prove that evaluation of programs is not influenced by removal of unused functions we first prove the corresponding theorem for expressions. Having done this we can use it for the main function of a program. Since we are going to use strong rule induction we need to split the lemma into two.

Lemma 6.43

$$\vdash \forall N, e, P_1, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P_1\, E\, v \Rightarrow \\ \forall P_1 : \texttt{stat\_ok}\, P_2 \Rightarrow \\ \quad \forall f : \left(\begin{array}{l} (\texttt{lookup\_func}\, P_1\, f = \texttt{FAIL}) \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{FAIL}) \end{array} \right) \Rightarrow \\ \quad \forall f, x, e : \left(\begin{array}{l} (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x,e)) \Rightarrow \\ \quad\quad (\texttt{lookup\_func}\, P_2\, f = \texttt{FAIL}) \vee \\ \quad\quad (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x,e)) \end{array} \right) \Rightarrow \\ \quad \forall f : \left(\begin{array}{l} \texttt{func\_called}\, f\, e \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f \neq \texttt{FAIL}) \end{array} \right) \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P_2\, E\, v \end{array} \right)$$

Proof: By strong rule induction and reduction. □

Lemma 6.44

$$\vdash \forall N, e, P_2, E, v : \left(\begin{array}{l} \texttt{eval\_expr\_n}\, N\, e\, P_2\, E\, v \Rightarrow \\ \forall P_1 : \texttt{stat\_ok}\, P_2 \Rightarrow \\ \quad \forall f : \left(\begin{array}{l} (\texttt{lookup\_func}\, P_1\, f = \texttt{FAIL}) \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{FAIL}) \end{array} \right) \Rightarrow \\ \quad \forall f, x, e : \left(\begin{array}{l} (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x,e)) \Rightarrow \\ \quad\quad (\texttt{lookup\_func}\, P_2\, f = \texttt{FAIL}) \vee \\ \quad\quad (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x,e)) \end{array} \right) \Rightarrow \\ \quad \forall f : \left(\begin{array}{l} \texttt{func\_called}\, f\, e \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f \neq \texttt{FAIL}) \end{array} \right) \Rightarrow \\ \quad \texttt{eval\_expr\_n}\, N\, e\, P_1\, E\, v \end{array} \right)$$

Proof: By strong rule induction and reduction. □

Theorem 6.45 (Subset of Programs Global Improvement.)

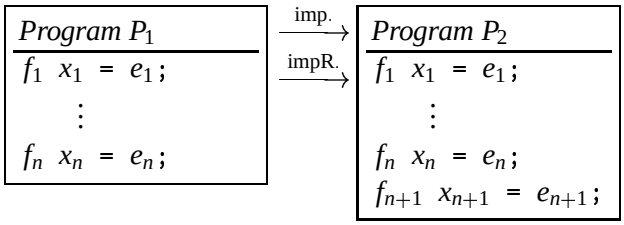
$$\vdash \forall P_1, P_2 : \left(\begin{array}{l} \forall f : \left(\begin{array}{l} (\texttt{lookup\_func}\, P_1\, f = \texttt{FAIL}) \Rightarrow \\ (\texttt{lookup\_func}\, P_2\, f = \texttt{FAIL}) \end{array} \right) \Rightarrow \\ \forall f, x, e : \left(\begin{array}{l} (\texttt{lookup\_func}\, P_1\, f = \texttt{OK}\,(x,e)) \Rightarrow \\ \quad (\texttt{lookup\_func}\, P_2\, f = \texttt{FAIL}) \vee \\ \quad (\texttt{lookup\_func}\, P_2\, f = \texttt{OK}\,(x,e)) \end{array} \right) \Rightarrow \\ \texttt{stat\_ok}\, P_2 \Rightarrow \\ (\texttt{HD}\, P_1 = \texttt{HD}\, P_2) \Rightarrow \\ \texttt{improvement}\, P_2\, P_1 \wedge \texttt{improvementR}\, P_2\, P_1 \end{array} \right)$$

Proof: By Lemmas 6.43 and 6.44. □

There is no local improvement theorem because the program transformation cannot be described in terms of replacement. Also, there is no static correctness preservement theorem because checking the necessary preconditions would be worse than just checking static correctness.

6.11 Adding a New Function

Adding a new function to a program cannot be described by replacement either. Fortunately, it is very simple to prove the improvement theorems for this transformation anyway: it is just a special case of removing unused functions![3] To see this, we just need two simple lemmas.



Lemma 6.46

$$\vdash \forall P, f, x, e, f' : \left(\begin{array}{l} (\texttt{lookup\_func}\,(\texttt{APPEND}\,P\,[(f,x,e)])\,f' = \texttt{FAIL}) \Rightarrow \\ (\texttt{lookup\_func}\,P\,f' = \texttt{FAIL}) \end{array} \right)$$

Proof: By list induction. □

Lemma 6.47

$$\vdash \forall P, f, x, e, f', x', e' : \left(\begin{array}{l} (\texttt{lookup\_func}\,(\texttt{APPEND}\,P\,[(f,x,e)])\,f' = \texttt{OK}\,(x',e')) \Rightarrow \\ \quad (\texttt{lookup\_func}\,P\,f' = \texttt{FAIL}) \vee \\ \quad (\texttt{lookup\_func}\,P\,f' = \texttt{OK}\,(x',e')) \end{array} \right)$$

Proof: By list induction. □

With these two lemmas we can now prove that adding a new function does not influence evaluation.

[3] Given that we work mainly with predicates this might not be such a surprise for people in the Prolog world.

Theorem 6.48 (Adding a Function Two-Way Improves.)

$$\vdash \forall P, f, x, e : \texttt{stat\_ok}\, P \Rightarrow \left(\begin{array}{l} \texttt{improvement}\, P\, (\texttt{APPEND}\, P\, [(f, x, e)]) \wedge \\ \texttt{improvementR}\, P\, (\texttt{APPEND}\, P\, [(f, x, e)]) \end{array} \right)$$

Proof: By Theorem 6.45 and Lemmas 6.46 and 6.47. □

Static correctness is also preserved when adding a new function, provided that the new function is suitably well-behaved: It must not already be a function in the program and it must use its variables and functions in an orderly manner.

Theorem 6.49 (Adding a Function Preserves Static Correctness.)

$$\vdash \forall P, f, x, e : \left(\begin{array}{l} \texttt{stat\_ok}\, P \Rightarrow \\ \neg(\texttt{MEM}\ f\, (\texttt{functions}\, P)) \Rightarrow \\ \texttt{check\_vars}\, [x]\, e \Rightarrow \\ \texttt{check\_funcs}\, (\texttt{functions}\, (\texttt{APPEND}\, P\, [(f, x, e)])\, e \Rightarrow \\ \texttt{stat\_ok}\, (\texttt{APPEND}\, P\, [(f, x, e)]) \end{array} \right)$$

Proof: There are three conjuncts to check in the definition of `stat_ok`: Non-null: because P is non-null. No rebindings: by Lemmas 4.84 and 4.85. Name usage: by Lemmas 4.84 and 4.87. □

6.12 Summary

We have presented a series of program transformations for our language. These transformations, primarily based on replacing one equivalent expression with another, have been proven semantics-preserving. In Chapter 7 we shall demonstrate that our set of transformations is strong enough to describe partial evaluation.

Chapter 7

Partial Evaluation

For every problem, there is one solution which is
simple, neat and wrong.
— H. L. MENCKEN, 1880–1956

In this chapter we present a concrete partial evaluator, namely the *trivial partial evaluator*, and show its correctness. We use this partial evaluator and the program transformations from Chapter 6 to give an internal definition of what partial evaluation is.

We shall show how a traditional partial evaluator's actions can be embedded into our framework, i.e., that our definition of partial evaluation contains the traditional meaning.

We also show that our program transformations can be sequenced meaningfully and use Ackermann's function to demonstrate that our definition of partial evaluation is strong enough to handle the problems that arise when specialising this function.

7.1 The Trivial Partial Evaluator

The *trivial partial evaluator* is defined in the following way.

Definitional Theorem 7.1 (The Trivial Partial Evaluator.)

$$\vdash \forall P, v_s : \left(\begin{array}{l} \texttt{triv\_pe}\, P\, v_s = \\ \texttt{CONS}\,(\texttt{genfunc}\, P, \\ \quad 1, \\ \quad \texttt{E\_Call}\,(\texttt{FST}\,(\texttt{HD}\, P))\,(\texttt{E\_Pair}\,(\texttt{value2expr}\, v_s)\,(\texttt{E\_Var}\, 1))) \\ \quad P \end{array} \right)$$

In other words, for a program whose main function is `f` and which is being specialised to static first argument v_s the trivial partial evaluator simply adds a new main function g `x = f (`$\widetilde{v_s}$`,x);` where $\widetilde{v_s}$ is the constant expression evaluating to v_s and g is some fresh function name.

This is, as we shall prove shortly, a partial evaluator. It is only supposed to work for programs whose main function takes a pair as its argument. The trivial partial evaluator will always instantiate the first component of the programs' argument pair. The generalised definition with multiple static values is notationally complex and we shall ignore it here.

The following theorem states that the trivial partial evaluator produces programs that are at least as statically correct as its input.

Theorem 7.2 (The Trivial Partial Evaluator Preserves Static Correctness.)

$$\vdash \forall P, v : \texttt{stat\_ok}\, P \Rightarrow \texttt{stat\_ok}\,(\texttt{triv\_pe}\, P\, v)$$

Proof: By structural case analysis on P and Lemmas 4.86, 4.87, 4.89, 4.51, and 4.52. □

The following theorem states that the trivial partial evaluator has the correct external behaviour as a partial evaluator.

Theorem 7.3 (The Trivial Partial Evaluator is Correct.)

$$\vdash \forall P, v_s, v_d, v : \texttt{stat\_ok}\, P \Rightarrow \left(\begin{array}{l} \texttt{eval\_prg}\, P\,(\texttt{V\_Pair}\, v_s\, v_d)\, v = \\ \texttt{eval\_prg}\,(\texttt{triv\_pe}\, P\, v_s)\, v_d\, v \end{array} \right)$$

Proof: By rewriting with `eval_prg`'s definition. Then by Theorem 7.2, Lemmas 4.49 and 4.89, and the following lemma. □

Lemma 7.4

$$\vdash \forall P, f, x, e', e, E, v : \left(\begin{array}{l} \texttt{stat\_ok}\,(\texttt{CONS}\,(f, x, e')\, P) \Rightarrow \\ \texttt{stat\_ok}\, P \Rightarrow \\ \neg(\texttt{func\_called}\, f\, e) \Rightarrow \\ (\texttt{eval\_expr}\, e\, P\, E\, v \Leftrightarrow \texttt{eval\_expr}\, e\,(\texttt{CONS}\,(f, x, e')\, P)\, E\, v) \end{array} \right)$$

Proof: By the (following) Lemmas 7.5 and 7.6. □

The following lemma, which is the easy ("⇒") implication of Lemma 7.4 shows that an expression's meaning does not change if we add another function to the program, provided the new function does not ruin static correctness.

Lemma 7.5

$$\vdash \forall e,P,E,v: \left(\begin{array}{l} \texttt{eval\_expr}\, e\, P\, E\, v \Rightarrow \\ \forall f,x,e': \left(\begin{array}{l} \texttt{stat\_ok}\,(\texttt{CONS}\,(f,x,e')\,P) \Rightarrow \\ \texttt{stat\_ok}\, P \Rightarrow \\ \texttt{eval\_expr}\, e\,(\texttt{CONS}\,(f,x,e')\,P)\, E\, v \end{array} \right) \end{array} \right)$$

Proof: By strong rule induction. □

The following lemma, which is the other ("⇐") implication of Lemma 7.4, is slightly more difficult because we have to make sure that the expression we are evaluating does not depend on the function added to the program.

Lemma 7.6

$$\vdash \forall e,P,E,v: \left(\begin{array}{l} \texttt{eval\_expr}\, e\, P\, E\, v \Rightarrow \\ \forall f,x,e',P': \left(\begin{array}{l} \neg(\texttt{func\_called}\, f\, e) \Rightarrow \\ (P = \texttt{CONS}\,(f,x,e')\,P') \Rightarrow \\ \texttt{stat\_ok}\, P \Rightarrow \\ \texttt{stat\_ok}\, P' \Rightarrow \\ \texttt{eval\_expr}\, e\, P'\, E\, v \end{array} \right) \end{array} \right)$$

Proof: By strong rule induction. Then by reduction and Lemma 4.92. □

7.2 An Internal Definition of Partial Evaluation

From Section 7.1 we now have a partial evaluator which is proven correct but does not produce optimised programs. From Chapter 6 we furthermore have program transformations which are proven correct and improving. This means that we are now ready to give our internal definition of what partial evaluation is by combining these two.

Definition 3 *Partial evaluation is optimisation*[1] *of the trivial partial evaluator's output using the program transformations discussed in Chapter 6 subject to the following constraints.*

1. *The first program transformation introduces a syntactic identity function.*

[1] Here "optimisation" is used in the sloppy and incorrect way it usually is in the program transformation community, i.e., as "program transformation with the honest intent of improving execution time for the program in question for most allowed input." We do not claim to reach the/a best possible program.

2. *In this phase, the main phase, partial evaluation shall do the following.*
   (a) *All added new function definitions are specialisations of existing functions, i.e., their bodies are calls to an existing function and their argument is a pairing of constants and components of the formal parameters.*
   (b) *Any function call folding, i.e., use of the inverse function call unfold or the history rules, shall be with respect to either a function whose initial definition was introduced in phase 2 or with respect to the main function added by the trivial partial evaluator.*
   (c) *No functions are removed in this phase.*
3. *The penultimate program transformations shall be a possibly empty series of eliminations of calls to the syntactic identity function introduced above, and the last program transformation shall be a subset transformation eliminating at least the syntactic identity function.*

Constraints 1 and 3 are purely technical constraints that allow us to get things started and finished in an orderly manner. The interesting part is constraint 2.

Constraint 2a characterises partial evaluation in that it only allows introduction of a certain class of functions, namely projections of existing functions. This distinguishes partial evaluation from more general fold-unfold transformation schemes like [BD77], which allow introduction of arbitrary function definitions.

Constraint 2b characterises partial evaluation in that the only folding we need is recognising an already specialised function and using it instead of unfolding the same call again. The methods in [BD77] are more general in this respect also, in that any folding is allowed.

Constraint 2c is technical and makes sure that we always improve programs.

Note, that the above definition does not by itself define an algorithm. It lacks two important aspects of an algorithm:

- It is not deterministic. In the main phase there will always be many possible program transformation that can be applied.
- There is no termination criterion (and termination is not guaranteed). Since most of the program transformations are two-way we can perform an endless series of transformations.

The former means that many more precise specifications will fall under our definition of partial evaluation. The latter means that we have not solved the termination problems of partial evaluation (nor have we tried to), but as we shall see in Section 7.6 we can stop at any time we want to and still have provably correct behaviour of the residual program.

7.3 Traditional Partial Evaluation

A traditional partial evaluator — for example the Scheme0 Mix from [JGS93, Figure 5.6] — has the structure shown in Figure 15. (The figure shows a pre-algorithm, not an algorithm, since termination is not guaranteed.) The outline does not include pre- and post-processing of any kind and is shown slightly simplified to match the two-parameter scheme.

```
pending := {(f0, s0)};
marked := { };
while pending ≠ { } do begin
    (f, s) := ⟨pick an element from pending⟩;
    e := ⟨reduce f's body given static parameter s⟩;
    for each call g(s', d') in e do begin
        ⟨change call to g_s'(d')⟩;
        pending := pending ∪ {(g, s')}
    end;
    ⟨output function "f_s x_f = e;"⟩;
    marked := marked ∪ {(f, s)};
    pending := pending \ marked
end
```

Figure 15: Outline of traditional partial evaluation pre-algorithm.

In *pending* we keep a list of function-value pairs to keep track of which functions we need to specialise with respect to which values. Initially that is the main function, f_0, and the initial static data s_0.

Specialisation of a function f to a static value s is a purely local transformation where f's body is optimised given the static value. This leaves some residual expression e, which usually will contain calls to other functions with other static values. These calls we turn into residual calls — to functions which may or may not have been created yet — and we add the calls to our *pending* list. At this point we have specialised f to s and we output the new definition and move the function-value pair is moved to the *marked* list.

Finally we subtract all *marked* pairs from *pending* so the same function will not be specialised to the same static value twice. (An obvious optimization here is to avoid adding members of *marked* to *pending* in the inner loop and then just remove (f,s) from pending.)

7.4 Comparing Partial Evaluation Meanings

Some connections between the method of Figure 15 and Definition 3 are immediately clear:

- All new functions created by either method are specialisations of existing functions. (Actually, requirement 2a ensures only that new functions are specialisation of functions present at the time of introduction. However, since the composition of two projections is another projection the difference is immaterial.)

- All folding is done with respect to specialised functions. For Definition 3 this is captured by requirement 2b; for the traditional partial evaluation this holds because *all* functions in the residual program are specialisations.[2]

Some differences are also apparent:

- All intermediate programs produced during partial evaluation as defined in Definition 3 are self-sufficient. Intermediate programs produced by the method in Figure 15 will always contain calls to undefined functions except just before the loop terminates. (For each $(f,s) \in$ *pending* there will be at least one call to the not yet defined function f_s.)

  This difference is not important: it is trivial to augment Figure 15 so we can exit the loop at any time we might want to and still end up with a correct program.

- In partial evaluation using Figure 15 there is a clear distinction between the original program and the residual program. With Definition 3 the original program and residual functions are mixed.

  Since, as noted above, projections of projections are projections themselves, this difference is immaterial.

7.5 Embedding Traditional Partial Evaluation

We shall now see how the traditional partial evaluation pre-algorithm can be embedded in the framework of Definition 3.

[2] If we changed requirement 3 slightly to require that the final operation removed *all* functions not introduced as specialisations then partial evaluation by Definition 3 would also have this property. But in doing so, we would have to give up the ability to stop the main phase at any time of our choice.

Instead of working with *pending* and *marked* and instead of separating source program from target program, we shall keep all information in one mutating program. There will be four kinds of functions in this program:

1. The original program's functions, not counting the projection added by the trivial partial evaluator. These functions will remain unchanged until they are finally eliminated.

2. The syntactic identity function. This will, likewise, remain unchanged and eventually removed.

3. Introducted, but not yet reduced, functions. The new main function introduced by the trivial partial evaluator will initially belong to this group. These functions are all projections, $h\ d = g(s,d)$, and each correspond to an element in *pending*.

4. Reduced functions. Such a function corresponds to a element in *marked* and to a function output by Figure 15.

The grouping will not be evident from the intermediate program; it is instead something we must keep track of separately.

Using this representation, Figure 15 becomes the pre-algorithm in Figure 16 augmented by the notes below.

⟨introduce the new main function;⟩
⟨introduce a syntactic identity function;⟩
while ⟨program contains group 3 function⟩ **do begin**
    f := ⟨pick a group 3 function⟩;
    ⟨unfold the call in f⟩;
    ⟨reduce f by repeated constant folding and `let`-unfolding⟩;
    ⟨reduce f by propagating the outer I-call down the syntax tree⟩;
    for each "`let` x`=(I` s'`,`d'`) in` $g\ x$ `end`" in f's body **do begin**
        ⟨change to `let` x`=`d' `in` $g(s',x)$ `end`⟩;
        ⟨introduce $h\ x$ `=` g`(`s'`,`x`);` in group 3 if (g,s') is new⟩;
        ⟨fold f's `let` to $h\ d'$⟩
    end;
    ⟨move f to group 4⟩
end;
⟨Eliminate all I-calls⟩;
⟨Eliminate functions not in group 4⟩

Figure 16: Embedding of traditional partial evaluation pre-algorithm.

In Figure 16 we pick a group 3 function. Such a function is a projection, and as explained above it corresponds to an (f,s) in *pending* $\setminus$ *marked*.

In traditional partial evaluation we look up f's body and reduce it with the knowledge of the static parameter. In Figure 16 this has become an unfolding of a call $g(s,d)$ and then reduction using `let`-unfolding and (extended) constant folding. These two program transformations cover what a typical partial evaluator would do at this point.

Note that the body expression will now contain a call to a syntactic identity function at top level. This call which does not have a counterpart in traditional partial evaluation is now propagated down the syntax tree using the rules of Figure 14 read left-to-right. The rules should be used exhaustively, except that an I-call should never be moved into a constant expression[3]. (To ensure termination, the function call rule should not be used with the syntactic identity function.) Note that the rules of Figure 14 are such that an I-call will be propagated down to every leaf or constant.

At any point where traditional partial evaluation would leave us with $g(s,d)$, the I-call propagation and especially the function call rule of Figure 14 will have left us with `let` x`=(I s,`d'`)` `in` g x `end` where d' is derived from I d. Where Figure 15 searches for a call $g(s,d)$, in Figure 16 this has therefore become a search for such a `let`-binding.

For such a `let`-binding, we use the argument manœuvres of Section 6.7 to move the static value back to the function call. If we previously introduced a function for g specialised to s', we can now fold with that, possibly after an alpha-conversion not shown in Figure 16. If (g,s') is new, we can introduce a new group 3 function and do a fold with that. (In the latter case, the fold degenerates to an inverse function unfolding.)

The postprocessing eliminates all I-calls and functions no longer needed. As discussed, this has not counterpart in Figure 15.

* * *

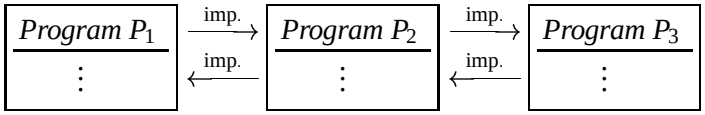
We have not formally proven that the embedding above works. It would be interesting to do so, but we judge the task to be too big. We would, for one thing, have to present a concreate non-trivial partial evaluator or other precise version of Figure 15. The embedding would then essentially be a proof of correctness for the partial evaluator.

[3] A constant expression is an expression in the range of `value2expr`, i.e., any expression consisting solely of integers, pairs, and injections.

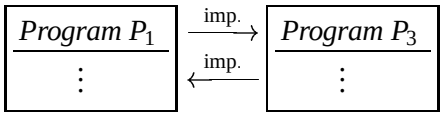
7.6 Sequencing Program Transformations

In this section we show that sequencing program transformations produce new program transformation which inherit the weakest improvement property of the components. In doing so, we show that partial evaluation as defined in Definition 3 is semantics preserving.

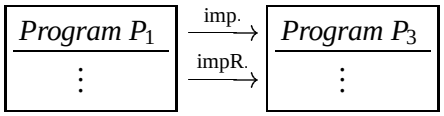
If we have three programs, P_1, P_2, and P_3, where P_2 is generated from P_1 with a two-way improving program transformation and P_3 is generated from P_2 in a similar manner, i.e., we have the situation



then by transitivity of improvement (Lemma 4.61) we also have



Furthermore, if one or both (or none) of the '$\xleftarrow{\text{imp.}}$' is replaced by an '$\xrightarrow{\text{impR.}}$' then we are able to conclude



by Lemmas 4.61, 4.74, and 4.75. Together with Lemma 4.59 then, if we have a series of zero or more two-way improving program transformations

$$P_1 \xrightarrow{T_1} P_2 \xrightarrow{T_2} \cdots P_{n-1} \xrightarrow{T_{n-1}} P_n$$

then we can combine them into one two-way improving program transformation. Two-way improvements as generated this way are needed in order to be able to invoke the history of a function, see Section 6.9.

Assume now that we do partial evaluation as specified in Definition 3. Let P_a be the output from the trivial partial evaluator, let P_b be the program after phase 1, let P_c be the program after phase 2, let P_d be the program just before removal of unused functions in phase 3, and let P_e be the final program.

By the global improvement theorems of Chapter 6 and transitivity of improvement we have `improvement` $P_a\,P_d$ and we have `improvement` $P_e\,P_d$. This situation exactly matches the preconditions in the following theorem which states that the meaning of a program — including termination and error properties — is preserved for three programs in such relationships.

Theorem 7.7 (Correctness of partial evaluation.)

$$\vdash \forall P_a, P_d, P_e : \left(\begin{array}{l} (P_a \neq [\,]) \Rightarrow \\ (P_e \neq [\,]) \Rightarrow \\ \texttt{improvement}\ P_a\, P_d \Rightarrow \\ \texttt{improvement}\ P_e\, P_d \Rightarrow \\ \forall v_{in}, v_{out} : \texttt{eval\_prg}\ P_a\, v_{in}\, v_{out} \Leftrightarrow \texttt{eval\_prg}\ P_e\, v_{in}\, v_{out} \end{array} \right)$$

Proof: By double use of Lemma 4.57. □

(The grandiose name of this theorem does not imply that it is a deep theorem; it is in fact a simple putting together of pieces.)

7.7 Example: Ackermann's Function

The Ackermann function is, as mentioned in the introduction, a very common test case for partial evaluators and partial evaluation techniques. In our language the function looks like this.

```
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
```

If a value for the first component of `mn` is known it is possible to derive a specialised version which does not contain the `case` constructs for this component. If, for example, we use the value `2` for the first component of `mn` we can obtain the following program.

```
fun ack2 n =
  case (n = 0) of
    R._ => ack1 1
  | L._ => ack1 (ack2 (n - 1))
  end;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 1
  | L._ => ack0 (ack1 (n - 1))
  end;
fun ack0 n = (n + 1);
```

The calls `(ack0 1)` and `(ack1 1)` could even be reduced further, but following the off-line partial evaluation tradition with mono-variant binding-time analysis [JGS93] we have not done this.

Studying the residual program we see why Ackermann's function so often is used as a test case. It is a small example which exhibits most of the problems with the technique:

- There is a need for having several specialised versions of the same function.
- There is a need for memorisation, i.e., when we see a call `ack(`e_m`,`e_n`)` during our specialisation it is important that we do not always blindly unfold the call because that would lead to non-termination of the partial evaluation. (In this case this is because contains a recursive call where the static parameter does not decrease.) Instead we must check whether we have already specialised `ack` with respect to the (hopefully) constant expression e_m. If we have, we should create a residual call, `(ack`$_{e_m}$ e_n`)`, instead of unfolding.
- There is a need for constant folding, i.e., the partial evaluation process actually performs part of the computation. For example, during specialisation we see that the zero-tests on `mn`'s first component as well as component extractions with `fst` and `snd` are computations that are constant folded.

Appendix B shows how the program transformations of Chapter 6 can be used to perform partial evaluation of Ackermann's function with respect to the value `2` for the first parameter. There are a total of 70 transformations used to derive (modulo naming of functions) the above residual program. The transformations were selected as per Figure 16, except that the order of a few transformations were swapped.

Note that, as explained in Appendix B, when a transformation $P_1 \rightarrow P_2$ is shown it is often the case that we have a hidden middle program which is alpha equivalent to either P_1 or P_2:

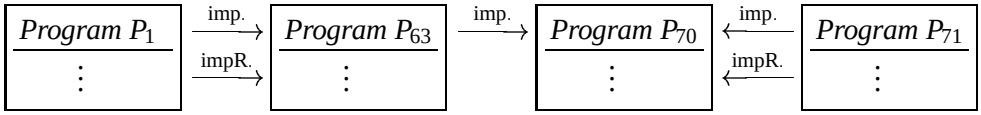
$$P_1 \xrightarrow{\alpha} P_1' \xrightarrow{T} P_2 \qquad \text{or} \qquad P_1 \xrightarrow{T} P_2' \xrightarrow{\alpha} P_2$$

The middle program has been hidden because it contains one or more variable names generated by `genvar`. We do not want to keep such names because (1) the inner workings of `genvar` should be hidden, and (2) having variables get new names all the time makes the example hard to follow.

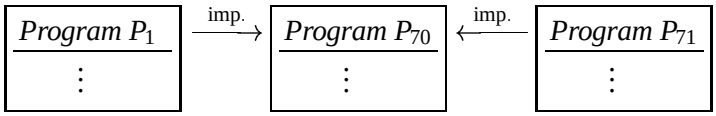
Note that when constant folding occurs — 14 times — we use the opportunity to replace *all* occurrences of the expression being constant folded simultaneously. This reduces the number of transformations needed significantly.

There are four phases in the transformation: From P_1 to P_2 we prepare for the main work by adding a syntactic identity function to the program; from P_2

to P_{63} we perform the specialisation; from P_{63} to P_{70} we clean up by removing calls to the identity function; and finally from P_{70} to P_{71} we eliminate unused function definitions. This gives us the following improvement diagram (collapsing for space reasons the first two phases as shown above),



If we furthermore collapse the first/second and the third phase by using transitivity of improvement (Lemma 4.61) and drop our knowledge of the reverse improvement relation we get the following diagram:



By Theorem 7.7 we have correctness of the residual program.

Chapter 8

Further Work

Prediction is difficult —
especially of the future.
— STORM P., 1882–1949

The present work is concentrated in two areas: programming language theory and the use of mechanically verified proofs. In our discussion of future research directions we will therefore split into work that has to do with programming languages and work that has to do with HOL.

8.1 The Programming Language Theory Side

This section discusses further research that should be conducted in the programming language theory direction of the present thesis.

8.1.1 Higher-Order Languages and timed Semantics

In [San96] Sands shows that the folding operation can be handled in a language with higher-order constructs also. However, the presence of closures as values opens the need for a weaker equality concept, usually called bi-similarity. It is immediately clear that a formalisation of Sands' work would be highly complicated by this need. What is not immediately clear, however, is whether Sands' work could be made to work with a timed semantics of "re-usable resource" kind, see Section 3.4. Since the non-reusable resourced timed semantics put counter-intuitive restrictions on the moving of identity function calls, see Section 9.3, we

strongly believe that further research should be conducted in the direction of finding a re-usable resource timed semantics that will work with Sands' system or a slightly modified version of it.

Seen from the point of folding, a good timed semantics is one that puts as many expressions as possible in the same class while still having a true Improvement Globalisation Theorem. To see this, consider the alternative: if we count evaluation steps in the traditional sense, then very few program transformations are two-way improving. If, on the other hand, we count nothing at all (i.e., use `eval_expr`) then local improvement — which degenerates to evaluation equivalence — is not strong enough to give us global improvement. The right choice is somewhere in-between and research should locate where.

8.1.2 More Program Transformations

The set of program transformations in Chapter 6 has been selected with partial evaluation in mind. There is no reason why it should not be possible to extend the set to include, for example, deforestation's `case`-of-`case` rule, (7) in [Wad90, Figure 4]. This rule says that the fragment

```
case (case e of L.xl1 -> el1 | R.xr1 -> er1 end) of
  L.xl2 -> el2
| R.xr2 -> er2
end
```

and the fragment

```
case e of
  L.xl1 -> case el1 of  L.xl2 -> el2 | R.xr2 -> er2 end
| R.xr1 -> case er1 of  L.xl2 -> el2 | R.xr2 -> er2 end
end
```

are equivalent provided neither `xl1` nor `xr1` are free in either `el2` or `er2`. The effect of this is to move the destructor in the hope that constant folding can be used, `case`, closer to the construction of the value it destructs. There is a similar `case`-of-`let` rule.

It should be straightforward to prove that these two program transformations two-way improve a program, as they do not involve function calls.

8.1.3 Typing

Section 3.7 contains basic work on adding a type system to our language. But it is not carried through: the relationship between typing of expressions and programs and the evaluation of these is not stated, nor have interesting theorems about it been proven. These holes should be patched.

Furthermore we should make sure that our program transformation not only preserve static correctness, as we have proven, but also typing. No properties stating this have been proven.

We expect both of these directions to require a large amount of work.

8.2 The HOL Side

HOL has been used for theorem proving in connection with both hardware and software, although there seems to be an imbalance in favour of hardware theorem proving. The present work and previous works like [MG94] show that HOL is perfectly able to work with programming language theory so the bias is likely due to historical reasons.

During the present work some problems and potential problems with using the HOL theorem prover in software contexts have been identified. These concentrate on automation, i.e., the proof search techniques and tools present in HOL.

8.2.1 Quadratic-Time Behaviour with Abstract Syntax

Programming language theory typically involves definition of some language using "abstract syntax," i.e., by defining a type and a set of constants with properties as constructors, see Section 3.2.1.

There is a serious problem with efficiency programmed into this approach to abstract syntax: the number of theorems[1] stating that the constructors have different images grows as $n(n-1)$ where n is the number of constructors. For the language used herein there were 132 theorems for this purpose.

This large number of theorems hurts automation: it is essential for productivity to have some automatic method to replace equalities between different constructors with falsity. Without automation we would explicitly have to exhibit the theorem stating that constructors C_1 and C_2 are different. Currently we just request rewriting with respect to *all* 132 theorems.

Automatic tools for improving the efficiency of such rewriting do exist: HOL comes with relatively efficient discrimination-net based functions that allow left-to-right rewriting with respect to a set of equalities. In the face of large numbers of theorems these methods become slow, however, as they are of linear complexity when applied to sets of theorems with the same shape. A variation of this method, turning the rewriter's kernel into a generating extension, was explored

[1] Counted as HOL's rewriting logic will see it, i.e., disregarding that several theorems can be combined into one by using conjunctions. The HOL tools actually return all the theorems stating that constructors are different as one gigantic conjunction.

in [Wel95]. This approach results in significantly[2] faster rewriting at the cost of reduced flexibility and higher start-up cost. The proofs described in this thesis have been conducted using rewriting tools produced by this generating extension technique.

Although tools as described above provide some relief we feel that further research into a more fundamental solution to this problem is needed. A possible direction this research might take could be to give up having all n constructors belong to the same type. If, for example, n constructors were divided into two groups and we defined three mutually recursive types

$$\begin{array}{rcl} T & ::= & T_1 \mid T_2 \\ T_1 & ::= & C_1 \ldots \mid \ldots \mid C_{\lfloor n/2 \rfloor} \ldots \\ T_2 & ::= & C_{1+\lfloor n/2 \rfloor} \ldots \mid \ldots \mid C_n \ldots \end{array}$$

then the number of theorems stating syntactic difference would be approximately $2+2(n/2)^2$, i.e., about half the $n(n-1)$ theorems the direct method would produce. If the constructors were divided into smaller, but more groups — for example with two in each — then the number of theorems could be reduced to approximately $2n$. (When defining a type with HOL's standard method [Mel91] the type is actually defined as isomorphic to a type having this structure.) Unfortunately, using such methods would be visible to the user of HOL. Furthermore, the initiality theorem for mutually recursive types are complicated.

8.2.2 Existential Quantifiers with Witness

Working with a semantics described by an inductive set of rules often lead to goals with a subterm of the form

$$\exists x_1 : \exists x_2 : \ldots \exists x_n : \Big(\ldots (x_1 = e) \ldots \Big) \tag{8.1}$$

where the equality appears in some "positive context" that would allow us to drop the outermost existential quantifier and replace x_1 by e in the body. These subterms are annoying in the extreme since: (a) they occur often, (b) they block automatic simplification, (c) they are not quite easy to handle manually as they involve bound variables and might occur deep inside the goal. There are two major sources of such terms in the proof described earlier in this thesis: rewriting by Theorems 4.7 and 4.11 which state that the semantics of the language is deterministic and that syntax directed reduction is possible. Consider, for example, an attempt at proving

[2]The actual complexity depends on the implementation's treatment of Standard ML's case-constructs.

the goal (one half of Equation 3.19)

$$\vdash^? \forall N, e, P, E, v : \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v \Rightarrow \texttt{eval\_expr}\ e\ P\ E\ v \tag{8.2}$$

by rule induction will lead to 12 subgoals. After reduction by Theorem 4.11 and injectivity of constructors one of the subgoals — the one for `E_Inl` — will be

$$\left\{ \begin{array}{l} \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v \\ \texttt{eval\_expr}\ e\ P\ E\ v \end{array} \right\} \vdash^? \exists v' : (v' = v) \wedge \texttt{eval\_expr}\ e\ P\ E\ v' \tag{8.3}$$

In this simple example, the existential quantifier is at top level and we might proceed by exhibiting the witness, v, by hand. In more complicated situations this approach does not work, either because the quantifier is not at top level or because the witness depends on variables not in scope at the quantifier's position, e.g., by x_2 in Equation 8.1.

To help solve goals like this a quite general existential quantifier eliminating conversion had to be constructed. When applied to a term of the form in 8.1 it will eliminate the outermost existential quantifier provided a witness in form of an equality is present as a top-level conjunct of the body. The elimination will take place even if the witness depends on any of $x_2, \ldots, x_n$. (This complication is handled by moving the quantifier inwards.) When applied to Equation 8.3 it will produce the simplified goal

$$\left\{ \begin{array}{l} \texttt{eval\_expr\_n}\ N\ e\ P\ E\ v \\ \texttt{eval\_expr}\ e\ P\ E\ v \end{array} \right\} \vdash^? \texttt{eval\_expr}\ e\ P\ E\ v \tag{8.4}$$

which is easily proven.

There is a similar problem with universally quantified implications where the antecedent names a witness, i.e., terms of the form

$$\forall x_1 : \forall x_2 : \ldots \forall x_n : \Big(\Big(\ldots (x_1 = e) \ldots \Big) \Rightarrow \ldots \Big). \tag{8.5}$$

Again, the equality should occur in a positive position but with respect to the antecedent. A similar reduction should be possible to automate for this class of terms which occur frequently in, for example, induction proofs using `var_bound`.

We feel that HOL needs some work in this direction, i.e., that quantifier-eliminating tools as general as possible should be provided.

8.2.3 Redefinition of Constants

The HOL system does not allow redefinition of constants as this could be used to make the logic inconsistent. (For example: define $c = T$ and prove $\vdash c$; then redefine $c = F$ and use this definition and the previous theorem to obtain $\vdash F$.)

Thus prohibiting redefinition of constants is good for consistency. But at the same time it is also very damaging to the use of HOL in the development phase because any change necessary in a constant's value can make it necessary to replay literally several hours of proof scripts. This happened twice to the author while proving the correctness of the self-interpreter which initially had a few problems with badly placed nested pairing.

It is therefore the author's impression that HOL needs to be extended with a means of redefining constants. Since it poses the same threat to proof security as the `mk_thm` function[3] it would make sense to flag any use of redefinition in the same way. This would tell the user to re-run the proof scripts for optimal proof security, *but at a convenient time instead of insisting on it right away.*

8.2.4 Syntax-Directed Reduction

As discussed in Section 4.3.2 the case analysis theorem one gets from defining a function by an inference scheme — for example Theorem 3.16 — is in an inconvenient shape because it is hard to use with automatic tools.

Since the start-up costs of using HOL (or any other theorem prover, probably) are high just for learning the system it is our impression that tools for deriving syntax-direction reduction theorems like Theorem 4.8 from Theorem 3.16 should be provided.

[3] A "cheat" function which converts arbitrary sequents into objects of theorem type.

Chapter 9

Related Work

Which of us [. . .] is to do the hard and dirty work
for the rest — and for what pay?
Who is to do the pleasant and clean work,
and for what pay?
— JOHN RUSKIN, 1819–1900

9.1 Partial Evaluation

There is now extensive literature on partial evaluation. For a guide to the literature in the field in general, see [JGS93, Chapter 18]. Here we shall only deal with more closely related works.

9.1.1 Partial Evaluation and Correctness

John Hughes in [Hug96] presents a novel way of doing partial evaluation, namely by specifying the partial evaluator as a non-standard type inference. Almost every feature of any other partial evaluator is embedded into Hughes' system, e.g., poly-variance, constructor specialisation [Mog93], and higher-order specialisation. Being based on an inference system it appears that there is some hope that the method can eventually be proven correct. Unfortunately, the inference system is not syntax directed so some kind of fixed-point iteration is needed in order to perform actual partial evaluation with the system. It remains to be seen whether this promising new approach can be efficiently implemented and what effect it will have on partial evaluation.

Specialisation of various lambda calculi has been the subject of much research. In the following we cover some of the λ-calculus specialisers. However, since these specialisers are built with a completely different evaluation paradigm in mind compared to this dissertation — λ-calculus compared to recursive functions — there are many differences. Some of the problems in partial evaluation, memoisation for example, simple do not show up in connection with λ-calculus at a recognisable level.

λ-Mix

The correctness of λ-mix [Gom91, GJ91b], a partial evaluator for the untyped λ-calculus, has been studied by Gomard in [Gom92]. λ-mix works on annotated two-level λ-calculus terms. The partial evaluator performs constant folding and function unfolding but does no memoisation. Even so, λ-mix is self-applicable in the practical sense of the word.

In [Gom92] a correctness proof for λ-mix appears. The correctness proof, essentially that λ-mix satisfies Equation 1.1, is conducted under the condition that $[\![\lambda - \texttt{mix}]\!]$ is defined on not only the expression, *e*, and its static input but also on any subexpression of *e* and *any* static input. The effect of having such strong a precondition is unclear, but one might fear that the precondition simply does not hold in many interesting situation and thus that the correctness proof cannot be applied.

Hatcliff's Specialiser

Hatcliff in [Hat95] works with the simply typed lambda calculus with constants. (The fragment used in [Hat95] is strongly normalising but this is not used in the proofs. In the upcoming [Hat96] this restriction has been removed.) Following λ-mix, both a regular (one-level) and a two-level language is defined and well-annotation rules are given. A partial evaluator is an implementation of the semantics of the two-level language.

Using Elf, an implementation of LF (Logical Framework) and in some respects comparable to a theorem prover like HOL, Hatcliff shows the correctness of the well-annotatedness rules, i.e., that a two-level program which satisfies these rules will not commit type errors during specialisation.

A specialiser is also given, and it is proven sound that any terminating computation (i.e., any specialisation) of a two-level term reflects a computation of the corresponding one-level term with the same result. The proof is also in Elf.

The specialiser in [Hat95] mostly works like λ-mix, i.e., it does constant folding and function unfolding but not memoisation.

Mogensen's Specialisers

Mogensen in [Mog92a], [Mog92b], and [Mog95] discusses partial evaluation of the pure lambda calculus. Mogensen first exhibits a self-reducer, *R*, operating on higher-order syntax. A self-reducer is a program that reduces the representation of a term to the representation of the term's normal form if such exist. (So the self-reducer in Mogensen's terminology does what a self-interpreter in this dissertation does. The term "self-interpreter," following [Bar91] is used for a reverse of the encoding function.) From the self-reducer it is easy to program a partial evaluator

$$P \equiv \lambda m, n.R(Apply\ m\ n)$$

where *Apply* is a function that constructs the higher-order representation of the application of *m* and *n*, themselves higher-order representations of lambda terms. The self-reducer is proven correct in [Mog92a].

Also this specialiser for a lambda calculus works by constant folding and function unfolding. It does not do memoisation.

In [Wan93] Wand takes up Mogensen's specialiser, specifies correctness criteria, and prove that the specialiser is correct according to these criteria.

9.2 Correctness of Interpreters

In [And91] a self-interpreter for a small Lisp-like language is presented and then proven correct. The language treated works with S-expressions and allows one (recursive) function having one parameter. The set of operators include basic list manipulation and a conditional. The language is thus Turing-complete even though it is much simpler than the language used in this dissertation. As expected, using a smaller language makes the self-interpreter relative larger. In this case, for example, one function must emulate evaluation for both programs and expressions.

Andersen proves the self-interpreter correct by giving a denotational semantics for the language, which is the limit of a nested-call timed semantics. Even though the semantics is given in a completely different way and even though the proof is domain theory based, it is not very different from the proof given in Chapter 5. There is a difference in notation, for example that an implication between two fully applied `eval_expr_n` predicates becomes a relation, $\sqsubseteq$, between the result of fully applying the meaning functions. Another major difference is that the language and the interpreter are untyped and that there are no encoding functions involved in the proofs.

The proof is in traditional mathematical style and cuts some corners with respect to selecting the right branch of the *n*-way nested conditional that checks

the outermost constructor of an expression. While this "syntactic dispatch" is no doubt correct, one of the proofs for a lemma stating part of this correctness is just as obviously incorrect. (To be precise: proving Lemma 1 by case analysis will never work. Structural induction, at the minimum, will be needed.) Based on our own work especially with Lemma 5.6 we have some doubt that the outlined proof of termination correctness will in fact work.

The correctness of interpreters for λ-calculi has been proven numerous times, see for example [Bar91], [Kle36], [Mog92a], but these proofs have a completely different structure. We will therefore not discuss these in any detail.

9.3 Correctness of Folding

In [San96] Sands presents a language and a set of program transformations designed to cover techniques like fold-unfold transformations [BD77], partial evaluation, and deforestation [Wad90]. The present dissertation borrows ideas like the use of identity functions (called "ticks" ($\surd$) in [San96]) and improvement as the source of correctness in connection with folding.

In Sands' language a program is a set of mutually recursive equations. Expressions allowed are given by the grammar outline

$$e ::= x \mid f \mid \lambda x.e \mid e_1\, e_2 \mid e_1 @ e_2 \mid \texttt{case}\ldots \mid c(\vec{e}) \mid p(\vec{e})$$

where c ranges over a set of constructors and p ranges over a set of primitive functions. "@" is strict application; the implicit application is lazy. The language is thus higher-order language. Values for the language are weak head normal form terms, i.e., constants and constant expressions are identified. Identity of values is by observational equivalence, see [Mil77] or [Pit95].

Sands introduces a timed semantics for the language with the resource bound being a limit on the total number of non-primitive function calls, i.e., the semantics is closely related to the semantics in Section 3.4.2. Based on this, Sands introduces local improvement (under another name) and improvement which, since Sands only allows addition of new functions, not mutation of old ones, is a property that is not relative to a specific program. Sands proves equivalents of the improvement globalisation theorems (Theorems 4.71 and 4.72) contained in this dissertation. Sands notes that the two-way improvement globalisation theorem does not seem to follow from the one-way improvement globalisation theorem. Based on our experience with proving Theorem 4.72 we agree.

The rôle of identity functions is the same in [San96] as here: a call to an identity function servers as a marker for a position where folding is allowed. But as the timed semantics used is of the non-reusable kind none of the program transformations in Section 6.5 that multiply identity function calls are two-way improve-

ments in Sands' system. Instead, transformations like the following examples using our notation.

$$\begin{array}{rcl}
\texttt{E_Call}\, I\, (\texttt{E_Op}\, o\, e_1\, e_2) & \leftrightarrow & \texttt{E_Op}\, o\, (\texttt{E_Call}\, I\, e_1)\, e_2 \\
\texttt{E_Call}\, I\, (\texttt{E_Op}\, o\, e_1\, e_2) & \leftrightarrow & \texttt{E_Op}\, o\, e_1\, (\texttt{E_Call}\, I\, e_2) \\
\texttt{E_Call}\, I\, (\texttt{E_Pair}\, e_1\, e_2) & \leftrightarrow & \texttt{E_Pair}\, (\texttt{E_Call}\, I\, e_1)\, e_2 \\
\texttt{E_Call}\, I\, (\texttt{E_Pair}\, e_1\, e_2) & \leftrightarrow & \texttt{E_Pair}\, e_1\, (\texttt{E_Call}\, I\, e_2)
\end{array}$$

The interpretation of these rules is that if we need an outer identity function call propagated into a term in order to do folding safely there, then we must select *one* of the terms in which we will be allowed to fold. We find this need for selection counter-intuitive, see Section 8.1.1. To get around this restriction Sands shows how an identity function call located at top level in the definition of a function call be used to "pay" for folding calls to that function.

Finally, [San96] has an excellent coverage of related work in this particular area. We shall not try to duplicate this here.

9.4 Standard ML and HOL

A lot of work has gone into formalising the semantics of Standard ML, as defined in the language definition [MTH90], in HOL. There are many reasons for this, in particular

- HOL and Standard ML are closely related: The ML languages owe their existence to theorem provers such as HOL and HOL is implemented in Standard ML. In particular every HOL researcher knows Standard ML[1].

- The definition of Standard ML is an unusually rigorous language specification (at least from the level of parse trees and up). That is, [MTH90] contains more precise mathematical specifications and much less descriptive-language specifications than for example Scheme's definition [CR91].

- There are a number of known problems with the semantics of Standard ML, see [Kah93] and [Kah95]. A good formalisation would help confirm and complete this list of problems as could be used to make sure that suggested work-arounds do not introduce problems of their own.

[1]Recall that HOL in this thesis stands for HOL version 90.7 which is the version of HOL written in Standard ML. Its predecessor, HOL-88, predates the definition of Standard ML and is written in "classic" ML which is slightly different from Standard ML though mostly in syntactic ways. But still most users will know Standard ML.

- Standard ML is not a toy language like the language used in this thesis. The reason for using a toy language in this thesis and other places is that treating a full language like Standard ML is not currently feasible.

Common to the various approaches to the problem of formalising Standard ML is the need for defining mutually recursive data types, as the *The Core Language* [MTH90] is defined syntactically by a grammar with mutual recursion. While the tools described previously in Chapter 3 do not directly handle this there are no unsurmountable difficulties linked to this. See [Rus92], [Gun93], or [Har95] for more details.

Also common is the need to define a representation for a program's values. Since the language definition does not put any limit on the number of user defined types in a program, some layer of encoding must be used. In fact, Gunter showed in [Gun92] that even if we did restrict ourselves to a fixed set of types there would be problems: HOL's and Standard ML's ideas of types are different in essence and that if a type T was defined in HOL in such a way that it was a solution to

$$T = \mathit{Var}\ \mathit{num} \mid \mathit{App}\ T\ T \mid \mathit{Abs}\ (T \to T)$$

then *Abs* would not be injective and thus not constructor-like.

Syme in [Sym93] describes one attempt at formalising The Core Language. The approach used is very different in nature from the one used to define the language in Chapter 3 in that inferability of a sentence

$$s, \mathit{obj} \vdash \mathit{phrase} \Rightarrow \mathit{res}, s'$$

is defined to be equal to existence of a *valid and finite* inference tree with the sentence in question as root. Validity for inference trees means that they follow the inference rules for The Core Language. Thus, inferability is based directly on existence of inference trees while a least fixed point approach was used in Chapter 3. Syme's definition of inferability is not directly suitable for proving properties about the language but he shows that the forward inference rules follow from his definition.

The forward inference rules and the structural induction theorem from the syntax definition is used to prove two important theorems about the language:

- the Pattern Matching Theorem: If a sentence is inferable for a pattern then the two states s and s' are identical. In other words: pattern matching does not influence the store.

- The Determinacy Theorem: if two inferable sentences are identical to the left of "$\Rightarrow$" then they are completely identical, i.e., the results and final stores are also identical.

> In order to for this to hold, Syme formalises a deterministic choice of new addresses in stores. This is done using Hilbert's choice operator.

Another work aimed at formalising The Core Language in HOL was undertaken by VanInwegen and Gunter in [VG93] at the same time as [Sym93] discussed above. Interestingly, [Sym93] and [VG93] restrict themselves from the same parts of The Core Language: "reals," streams, and the `Interrupt` exception. The reasons for omitting these parts are not given but it is a safe bet that the reason is that Standard ML's definition [MTH90] is very weak on exactly these subjects. It is a — perhaps surprising — fact that the definition of the C language [KR88] is considerably more precise on, for instance, machine-represented "reals": the C Standard defines minimum ranges and resolutions for the relevant types and provides symbolic names for the actual ranges and resolutions.

9.5 Late Arrivals

A few related works surfaced around the time the present thesis was turned over to the committee.

In [NN96], Nazareth and Nipkow prove the correctness of Algorithm W paying special attention to the issue of fresh variables. The authors use global uniqueness since the set of variable names — integers as in the present work — is strongly ordered they only have to pass the "counter" around. Proofs are done with the Isabelle theorem prover.

Graham Collins discusses co-induction with a PCF-like language and mechanisation of co-inductive proofs in [Col96]. A tool that works with rules not unlike those used in Section 3.7 is developed but it is not immediately clear how powerful its monotonicity prover is but it is likely to be stronger than the one used for Section 3.7.

Chapter 10

Summary and Conclusions

The object of oratory alone is not truth but persuasion
— THOMAS BABINGTON MACAULAY, 1800–1859

We will now summarise this dissertation and the contributions in it. We will then re-discuss our thesis and goals. In this dissertation we have

- presented a simple language and formalised it and its semantics in the HOL theorem proving environment. Our formalisation has allowed us to reason about programs and expressions from our language, their evaluation properties, and termination within the HOL system.
- exhibited an interpreter for this language written in the language itself. We have proven, using HOL, that the interpreter is correct with respect to the programming language's semantics and with respect to the coding operations for values and syntax needed in order to program the interpreter in a typed style. The proof takes termination properties into account.
- shown a series of program transformations ranging from purely local transformations like constant folding, over transformations with a larger area of effect such as `let`-unfolding and alpha conversion, to transformations with global consequences such a folding and unfolding of function calls. These program transformations have been proven correct with respect both to succeeding evaluation and to non-terminating or erring evaluation.
- defined and shown the correctness of the trivial partial evaluator.
- given an internal definition of partial evaluation and demonstrated that the above program transformations are powerful enough to be used for partial evaluation with both specialisation and memorisation.

Returning now to our thesis from Section 1.1 we conclude the following for each of its three points.

1. Although the term "partial evaluation" has been used to describe many program transformations and techniques, thus making a statement of what partial evaluation can or cannot be described as hard to reason on, we find that we have given convincing evidence that the transformations described in Chapter 6 can be used to perform partial evaluation.

2. Yes, the correctness of partial evaluation has definitively been shown to be susceptible to proofs. This is clear from Chapters 4 and 6.

3. We believe that mechanical verification is indeed an asset to programming language researchers because it provides an unprecedented assurance of correctness in the results. The price is some extra work:

   - There is a significant start-up cost for learning the tool, in our case HOL. There is no hope of eliminating this cost, but it ought to be possible to lower it by creating better documentation[1] and by having more programming language examples available.
   - In proving the interesting theorems one seems to need an incredible amount of trivial lemmas, for example "`APPEND` is null exactly when the arguments are" and "`MAX` is commutative". Most of the 'obvious theorems' are true but a few are false. The purpose of HOL is to weed out the few false ones by requiring a formal proof of each and every one. Proving these theorems takes a significant amount of time and even though libraries of theorems exist they somehow never seem to contain the theorem one needs.
   - There is a significant need for computer time involved, both CPU-time and interactive time. Moreover, it is occasionally necessary to abandon a logically correct proof method solely because carrying it through would require unacceptable amounts of time or memory. This was discussed also in Section 4.3.2.

   On the other hand, mechanical theorem proving can also save time: in paper proofs a structural induction or rule induction means a serious amount of work and one is tempted to put it aside for later only to discover that it turns out to be false. When using a theorem prover, however, induction proofs

[1] HOL is a freely distributable program and the authors make no or little money directly from it. We realise that the level of academic recognition obtained by writing and maintaining documentation makes it difficult to defend spending time on it. HOL, in other words, needs at least a minimal funding.

as such do not imply much work because one can simultaneously reduce all resulting subgoals. In fact, the rule induction proofs conducted for this work most often had only two non-trivial kinds of cases: recursive function calls and `let`/`case`.

Based on the above observations and the work we have been through we strongly believe that much programming language work would benefit from some kind of formalisation.

In Chapter 2 we put forward the separate goal of working towards a provably correct and optimal partial evaluator for a typed language. We believe that we have forwarded this goal on two counts.

- We provided a self-interpreter using a universal encoding of values and proved that it functions correctly.
- We presented proven program transformations which are strong enough to do what some optimal partial evaluators for untyped languages do.

* * *

One of the most hard-earned lessons learnt during HOL proof sessions for this dissertation is that *false theorems are hard to prove.* If a full day's proof work has not resulted in a HOL-acceptable proof of some proposed theorem then that theorem probably is not true. Mostly this happened with goals involving evaluation and variable renaming and/or substitution which can look very convincing, yet be false. On the other hand, some true theorems are hard to prove too.

Chapter 11

Acknowledgements

I would like to thank the TOPPS group, the programming language group at the University of Copenhagen's Department of Computer Science, for valuable discussions. A well-functioning group is always the best start for fruitful research.

I thank *Neil Jones* for being my advisor through the majority of my studies, both at the Master's and the Ph.D. level. Without Neil Jones' persistent efforts, the partial evaluation would have dwindled away before it ever got started. I thank Neil for the help and advice to get a Ph.D. project going in spite my shortcomings and a false start. This thesis would not have been otherwise.

Without *David Sands'* prior work in proving correctness of the folding transformation, I doubt that could have formalised proofs of all the program transformations that I needed and my thesis would have felt like a chair missing a leg. I am grateful for the ideas and the advise that helped me complete my set.

It was *Tom Melham* who introduced me to HOL and machine verification of proofs in the very beginning of my Ph.D. studies. Back then, it was hoped that partial evaluation could be used to speed-up HOL directly. While this never materialised, the indirect method described in [Wel95] shows promise. I thank Tom Melham for opening up the field for me and for his very thorough and useful comments on an earlier version of this thesis.

I had the good fortune of being able to visit Carnegie Mellon University in the first half of 1995. Thanks go to *Peter Lee* for hosting me and introducing me to semantics group at CMU. Also thanks to *Greg Morrisett* for useful discussions on the connection between types and evaluation of a language, discussions that played a significant rôle in focusing this thesis.

Finally, I wish to thank *Morten H. Sørensen* for having the necessary large-scale overview for giving strategic proof advice. Formalisation of a proof is certainly faster given a sense of direction which Morten can generally provide. Praise also for his patience and tolerance in sharing an otherwise orderly office with me.

Appendix A

Transcription of HOL Terms

Die Mathematiker sind eine Art Franzosen:
redet man zu ihnen, so übersetzen sie es in ihre Sprache,
und dann ist es alsobald ganz etwas Anders.
— JOHANN WOLFGANG VON GOETHE, 1829

HOL is mainly an ASCII based system: all input and output takes place using a sequence of characters. The benefit of this approach is that it is efficient, it is portable, and it has simple and unambiguous syntactic rules. Some of the drawbacks are

- it does not make use of extended character sets that might be available in some settings. For example one might get lucky and find a keyboard with a "∀"-key only to find that HOL will not let one use it.

- it is inheritly one-dimensional, i.e., just a string of characters, while theorems, as other mathematical objects, traditionally are shown with the occasional excursion into two dimensions.

This thesis uses notation that is more from the mathematical world than from the HOL world and this appendix shows the correspondence.

For traditional mathematical entities used in formulae the following table summarises how this thesis uses a character set larger than the one used by HOL. In addition to these symbols, some of the functions defined for the purpose of this thesis also have special notation. For example, the semantics predicate for expressions is written $E \vdash_P e \Rightarrow v$.

Object	*HOL world*	*Mathematical world*
Universal quantifier	`!x. x=x`	$\forall x.x = x$
Existential quantifier	`?x. x=1`	$\exists x.x = 1$
Unique existence	`?!x. x=2`	$\exists! x.x = 2$
Abstraction	`\x. x+1`	$\lambda x.x + 1$
Negation	`~b`	$\neg b$
Conjunction	`s /\ t`	$s \wedge t$
Disjunction	`s \/ t`	$s \vee t$
Implication	`s ==> t`	$s \Rightarrow t$
Bi-implication	`s = t`	$s \Leftrightarrow t$ *or* $s = t$
Conditional	`b => 1 | 0`	$b \rightarrow 1 \mid 0$
Inequality	`~(m = 4)`	$x \neq 4$
Less-or-equal relation	`m <= n`	$m \leq n$
Greater-or-equal relation	`m >= n`	$m \geq n$
Sequent	`|- 2 = 2`	$\vdash 2 = 2$

Formulae in this borderline between computer science and mathematics have an eerie tendency of becoming larger than traditional mathematical notation was prepared for. HOL solves this by linearisation, i.e., a large formula is just a large string of symbols. This works but makes for poor readability, so this thesis instead uses two-dimensional displays like

$$\vdash \left\{ \begin{array}{l} \forall x, i : \texttt{strict}\,(\texttt{V\_Int}\ i)\,x = x \\ \forall x, v_1, v_2 : \texttt{strict}\,(\texttt{V\_Pair}\ v_1\ v_2)\,x = x \\ \forall x, v : \texttt{strict}\,(\texttt{V\_Inl}\ v)\,x = x \\ \forall x, v : \texttt{strict}\,(\texttt{V\_Inr}\ v)\,x = x \end{array} \right\}$$

to stand for a sequent whose conclusion is a universally quantified conjunct of four conjuncts. Note that the conjunction ($\wedge$) is implicit in the above display. Furthermore, we will occasionally omit parentheses that would be necessary in a linear representation of such displays. This is especially true for formulae threatening to exceed the line width.

Appendix B

Ackermann's Function

It's as large as life, and twice as natural!
— LEWIS CARROLL, 1832–1898

This appendix shows in great detail partial evaluation of Ackermann's function with respect to a known first argument equal to 2.

In this particular derivation there are a total of 70 program transformations, i.e., there are 69 intermediate programs. All 71 programs can be summarised as follows:

#	*Phase*	*Raison d'être*
1	Initialise	Source program with static argument inserted
2	"	Identity function introduced
2–63	Main work	Partial evaluation
64–70	Cleanup	Elimination of identity function calls
71	"	Final program, removal of unused functions

All the program that appear in the initialisation and main work phases, i.e., programs 1–63, are equivalent in the sense that the depth of any evaluation is preserved for any expression moved from being evaluated in one program context to being evaluated in another. Once the cleanup phase starts this no longer holds, but the transformed programs still calculate the same functions.

The actual proofs use numbered, not "named" functions and variables. The correspondence for function is 1 ↔ `main`, 2 ↔ `ack`, 3 ↔ `I`, 4 ↔ `ack1`, and 5 ↔ `ack0`. Uniformly over all programs, the correspondence for variables is 1 ↔ `n`, 2 ↔ `mn`, 3 ↔ `_`, 4 ↔ `x`, and 5 ↔ `n'`.

Some program transformations needed to keep the uniform variable naming scheme have been left out, namely alpha conversions in connection with constant

folds, `let`-unfolds, and `let`-folds. This is because aa constant fold can introduce a `genvar`-constructed variable and `let`-operations rename all bound variables using `genvar`. Using an intermediate unshown program which is alpha-equivalent allows us to hide the implementation details of `genvar`.

Program 1: Initial program.

```
fun main n = ack (2,n);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
```

Program 2: Introduce new function.

```
fun main n = ack (2,n);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 3: Unfold call.

```
fun main n =
  I (let
       mn = (2,n)
     in
       case (fst mn = 0) of
         R._ => (snd mn + 1)
       | L._ =>
           case (snd mn = 0) of
             R._ => ack ((fst mn - 1),1)
           | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
           end
       end
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 4: Unfold harmless `let`.

```
fun main n =
  I (case (fst (2,n) = 0) of
       R._ => (snd (2,n) + 1)
     | L._ =>
         case (snd (2,n) = 0) of
           R._ => ack ((fst (2,n) - 1),1)
         | L._ => ack ((fst (2,n) - 1),ack (fst (2,n),(snd (2,n) - 1)))
         end
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 5: Constant fold `fst (2,n)`.

```
fun main n =
  I (case (let _ = n in 2 end = 0) of
       R._ => (snd (2,n) + 1)
     | L._ =>
         case (snd (2,n) = 0) of
           R._ => ack ((let _ = n in 2 end - 1),1)
         | L._ => ack ((let _ = n in 2 end - 1),
                       ack (let _ = n in 2 end,(snd (2,n) - 1)))
         end
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 6: Unfold harmless `let`.

```
fun main n =
  I (case (2 = 0) of
       R._ => (snd (2,n) + 1)
     | L._ =>
         case (snd (2,n) = 0) of
           R._ => ack ((2 - 1),1)
         | L._ => ack ((2 - 1),ack (2,(snd (2,n) - 1)))
         end
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 7: Constant fold `2 - 1`.

```
fun main n =
  I (case (2 = 0) of
       R._ => (snd (2,n) + 1)
     | L._ =>
         case (snd (2,n) = 0) of
           R._ => ack (1,1)
         | L._ => ack (1,ack (2,(snd (2,n) - 1)))
         end
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 8: Constant fold `2 = 0`.

```
fun main n =
  I (case L 0 of
       R._ => (snd (2,n) + 1)
     | L._ =>
         case (snd (2,n) = 0) of
           R._ => ack (1,1)
         | L._ => ack (1,ack (2,(snd (2,n) - 1)))
         end
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 9: Constant fold `case False ...`.

```
fun main n =
  I (let
       _ = 0
     in
       case (snd (2,n) = 0) of
         R._ => ack (1,1)
       | L._ => ack (1,ack (2,(snd (2,n) - 1)))
       end
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 10: Unfold harmless `let`.

```
fun main n =
  I (case (snd (2,n) = 0) of
       R._ => ack (1,1)
     | L._ => ack (1,ack (2,(snd (2,n) - 1)))
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 11: Constant fold `snd (2,n)`.

```
fun main n =
  I (case (let _ = 2 in n end = 0) of
       R._ => ack (1,1)
     | L._ => ack (1,ack (2,(let _ = 2 in n end - 1)))
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 12: Unfold harmless `let`.

```
fun main n =
  I (case (n = 0) of
       R._ => ack (1,1)
     | L._ => ack (1,ack (2,(n - 1)))
     end);
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 13: Identity function/`case` *move.*

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (ack (1,ack (2,(n - 1))))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 14: Argument move.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = I (1,ack (2,(n - 1))) in ack x end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 15: Identity function/pairing move.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = (I 1,I (ack (2,(n - 1)))) in ack x end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 16: Argument move.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = (I 1,I (let n = I (2,(n - 1)) in ack n end))
              in ack x end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 17: Identity function/pairing move.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = (I 1,I (let n = (I 2,I (n - 1)) in ack n end))
              in ack x end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 18: First argument move.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = (I 1,I (let n = I (n - 1) in ack (2,n) end))
              in ack x end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 19: Invoke history of `main`.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = (I 1,main (I (n - 1))) in ack x end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 20: First argument move.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = main (I (n - 1)) in ack (1,x) end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
```

Program 21: Introduce new function.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let x = main (I (n - 1)) in ack (1,x) end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n = ack (1,n);
```

Program 22: Alpha conversion.

```
fun main n =
  case I (n = 0) of
    R._ => I (ack (1,1))
  | L._ => I (let n = main (I (n - 1)) in ack (1,n) end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n = ack (1,n);
```

Program 23: Argument move.

```
fun main n =
  case I (n = 0) of
    R._ => I (let n = I (1,1) in ack n end)
  | L._ => I (let n = main (I (n - 1)) in ack (1,n) end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n = ack (1,n);
```

Program 24: Identity function/pairing move.

```
fun main n =
  case I (n = 0) of
    R._ => I (let n = (I 1,I 1) in ack n end)
  | L._ => I (let n = main (I (n - 1)) in ack (1,n) end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n = ack (1,n);
```

Program 25: First argument move.

```
fun main n =
  case I (n = 0) of
    R._ => I (let n = I 1 in ack (1,n) end)
  | L._ => I (let n = main (I (n - 1)) in ack (1,n) end)
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n = ack (1,n);
```

Program 26: Inverse function unfold.

```
fun main n =
  case I (n = 0) of
    R._ => I (let n = I 1 in ack (1,n) end)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n = ack (1,n);
```

Program 27: Inverse function unfold.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n = ack (1,n);
```

Program 28: Unfold call.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (let
       mn = (1,n)
     in
       case (fst mn = 0) of
         R._ => (snd mn + 1)
       | L._ =>
           case (snd mn = 0) of
             R._ => ack ((fst mn - 1),1)
           | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
           end
       end
     end);
```

Program 29: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case (fst (1,n) = 0) of
       R._ => (snd (1,n) + 1)
     | L._ =>
         case (snd (1,n) = 0) of
           R._ => ack ((fst (1,n) - 1),1)
         | L._ => ack ((fst (1,n) - 1),ack (fst (1,n),(snd (1,n) - 1)))
         end
     end);
```

Program 30: Constant fold `fst (1,n)`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case (let _ = n in 1 end = 0) of
       R._ => (snd (1,n) + 1)
     | L._ =>
         case (snd (1,n) = 0) of
           R._ => ack ((let _ = n in 1 end - 1),1)
         | L._ => ack ((let _ = n in 1 end - 1),
                       ack (let _ = n in 1 end,(snd (1,n) - 1)))
         end
     end);
```

Program 31: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case (1 = 0) of
       R._ => (snd (1,n) + 1)
     | L._ =>
         case (snd (1,n) = 0) of
           R._ => ack ((1 - 1),1)
         | L._ => ack ((1 - 1),ack (1,(snd (1,n) - 1)))
         end
     end);
```

Program 32: Constant fold `1 - 1`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case (1 = 0) of
       R._ => (snd (1,n) + 1)
     | L._ =>
         case (snd (1,n) = 0) of
           R._ => ack (0,1)
         | L._ => ack (0,ack (1,(snd (1,n) - 1)))
         end
     end);
```

Program 33: Constant fold `1 = 0`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case L 0 of
       R._ => (snd (1,n) + 1)
     | L._ =>
         case (snd (1,n) = 0) of
           R._ => ack (0,1)
         | L._ => ack (0,ack (1,(snd (1,n) - 1)))
         end
     end);
```

Program 34: Constant fold `case False ...`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (let
       _ = 0
     in
       case (snd (1,n) = 0) of
         R._ => ack (0,1)
       | L._ => ack (0,ack (1,(snd (1,n) - 1)))
       end
     end);
```

Program 35: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case (snd (1,n) = 0) of
       R._ => ack (0,1)
     | L._ => ack (0,ack (1,(snd (1,n) - 1)))
     end);
```

Program 36: Constant fold `snd (1,n)`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case (let _ = 1 in n end = 0) of
       R._ => ack (0,1)
     | L._ => ack (0,ack (1,(let _ = 1 in n end - 1)))
     end);
```

Program 37: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  I (case (n = 0) of
       R._ => ack (0,1)
     | L._ => ack (0,ack (1,(n - 1)))
     end);
```

Program 38: Identity function/`case` *move.*

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (ack (0,ack (1,(n - 1))))
  end;
```

Program 39: Argument move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = I (0,ack (1,(n - 1))) in ack x end)
  end;
```

Program 40: Identity function/pairing move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = (I 0,I (ack (1,(n - 1)))) in ack x end)
  end;
```

Program 41: Argument move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = (I 0,I (let n = I (1,(n - 1)) in ack n end))
              in ack x end)
  end;
```

Program 42: Identity function/pairing-move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = (I 0,I (let n = (I 1,I (n - 1)) in ack n end))
              in ack x end)
  end;
```

Program 43: First argument move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = (I 0,I (let n = I (n - 1) in ack (1,n) end))
              in ack x end)
  end;
```

Program 44: Invoke history of `ack1`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = (I 0,ack1 (I (n - 1))) in ack x end)
  end;
```

Program 45: First argument move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = ack1 (I (n - 1)) in ack (0,x) end)
  end;
```

Program 46: Introduce new function.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (ack (0,1))
  | L._ => I (let x = ack1 (I (n - 1)) in ack (0,x) end)
  end;
fun ack0 n = ack (0,n);
```

Program 47: Argument move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (let n = I (0,1) in ack n end)
  | L._ => I (let x = ack1 (I (n - 1)) in ack (0,x) end)
  end;
fun ack0 n = ack (0,n);
```

Program 48: Identity function/pairing move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (let n = (I 0,I 1) in ack n end)
  | L._ => I (let x = ack1 (I (n - 1)) in ack (0,x) end)
  end;
fun ack0 n = ack (0,n);
```

Program 49: First argument move.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => I (let n = I 1 in ack (0,n) end)
  | L._ => I (let x = ack1 (I (n - 1)) in ack (0,x) end)
  end;
fun ack0 n = ack (0,n);
```

Program 50: Inverse function unfold.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => I (let x = ack1 (I (n - 1)) in ack (0,x) end)
  end;
fun ack0 n = ack (0,n);
```

Program 51: Fold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => let n' = n in I (let x = ack1 (I (n' - 1)) in ack (0,x) end) end
  end;
fun ack0 n = ack (0,n);
```

Program 52: Alpha conversion.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => let n' = n in I (let n = ack1 (I (n' - 1)) in ack (0,n) end) end
  end;
fun ack0 n = ack (0,n);
```

Program 53: Inverse function unfold.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => let n' = n in ack0 (ack1 (I (n' - 1))) end
  end;
fun ack0 n = ack (0,n);
```

Program 54: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = ack (0,n);
```

Program 55: Unfold call.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n =
  I (let
       mn = (0,n)
     in
       case (fst mn = 0) of
         R._ => (snd mn + 1)
       | L._ =>
           case (snd mn = 0) of
             R._ => ack ((fst mn - 1),1)
           | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
           end
       end
     end);
```

Program 56: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n =
  I (case (fst (0,n) = 0) of
       R._ => (snd (0,n) + 1)
     | L._ =>
         case (snd (0,n) = 0) of
           R._ => ack ((fst (0,n) - 1),1)
         | L._ => ack ((fst (0,n) - 1),ack (fst (0,n),(snd (0,n) - 1)))
         end
     end);
```

Program 57: Constant fold `fst (0,n)`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n =
  I (case (let _ = n in 0 end = 0) of
       R._ => (snd (0,n) + 1)
     | L._ =>
         case (snd (0,n) = 0) of
           R._ => ack ((let _ = n in 0 end - 1),1)
         | L._ => ack ((let _ = n in 0 end - 1),
                       ack (let _ = n in 0 end,(snd (0,n) - 1)))
         end
     end);
```

Program 58: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n =
  I (case (0 = 0) of
       R._ => (snd (0,n) + 1)
     | L._ =>
         case (snd (0,n) = 0) of
           R._ => ack ((0 - 1),1)
         | L._ => ack ((0 - 1),ack (0,(snd (0,n) - 1)))
         end
     end);
```

Program 59: Constant fold `0 = 0`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n =
  I (case R 0 of
       R._ => (snd (0,n) + 1)
     | L._ =>
         case (snd (0,n) = 0) of
           R._ => ack ((0 - 1),1)
         | L._ => ack ((0 - 1),ack (0,(snd (0,n) - 1)))
         end
     end);
```

Program 60: Constant fold `case True ...`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n =
  I (let
       _ = 0
     in
       (snd (0,n) + 1)
     end);
```

Program 61: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = I (snd (0,n) + 1);
```

Program 62: Constant fold `snd (0,n)`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = I (let _ = 0 in n end + 1);
```

Program 63: Unfold harmless `let`.

```
fun main n =
  case I (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = I (n + 1);
```

Program 64: Identity function call elimination.

```
fun main n =
  case (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case I (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = I (n + 1);
```

Program 65: Identity function call elimination.

```
fun main n =
  case (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = I (n + 1);
```

Program 66: Identity function call elimination.

```
fun main n =
  case (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (I (n - 1)))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = (n + 1);
```

Program 67: Identity function call elimination.

```
fun main n =
  case (n = 0) of
    R._ => ack1 (I 1)
  | L._ => ack1 (main (n - 1))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = (n + 1);
```

Program 68: Identity function call elimination.

```
fun main n =
  case (n = 0) of
    R._ => ack1 1
  | L._ => ack1 (main (n - 1))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (I (n - 1)))
  end;
fun ack0 n = (n + 1);
```

Program 69: Identity function call elimination.

```
fun main n =
  case (n = 0) of
    R._ => ack1 1
  | L._ => ack1 (main (n - 1))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 (I 1)
  | L._ => ack0 (ack1 (n - 1))
  end;
fun ack0 n = (n + 1);
```

Program 70: Identity function call elimination.

```
fun main n =
  case (n = 0) of
    R._ => ack1 1
  | L._ => ack1 (main (n - 1))
  end;
fun ack mn =
  case (fst mn = 0) of
    R._ => (snd mn + 1)
  | L._ =>
      case (snd mn = 0) of
        R._ => ack ((fst mn - 1),1)
      | L._ => ack ((fst mn - 1),ack (fst mn,(snd mn - 1)))
      end
  end;
fun I x = x;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 1
  | L._ => ack0 (ack1 (n - 1))
  end;
fun ack0 n = (n + 1);
```

Program 71: Subset.

```
fun main n =
  case (n = 0) of
    R._ => ack1 1
  | L._ => ack1 (main (n - 1))
  end;
fun ack1 n =
  case (n = 0) of
    R._ => ack0 1
  | L._ => ack0 (ack1 (n - 1))
  end;
fun ack0 n = (n + 1);
```

Appendix C

The Self-Interpreter as a HOL Term

Still with me?
I hope you're all taking notes —
there will be a short quiz afterwards.
— TOM LEHRER, The Elements

For purposes of completeness and as a reference this appendix shows the Self-interpreter of Section 5.2.5 as the HOL-term that encodes it. In particular, the term below shows the translation from names to integers of variable and function names. Since some proofs of lemmas reference functions from the self-interpreter by number we show the correspondence here:

1 ↔ `pelint`	2 ↔ `int2val`	3 ↔ `val2int`
4 ↔ `pair2val`	5 ↔ `val2pair`	6 ↔ `sum2val`
7 ↔ `val2sum`	8 ↔ `eval`	9 ↔ `update_env`
10 ↔ `lookup_env`	11 ↔ `lookup_func`	12 ↔ `eval_op`
13 ↔ `hd`		

Note that the term below is how HOL sees the self-interpreter, not how it would be presented to itself if self-application was desired. That would add another layer of encoding producing a term an order of magnitude larger.

```
--‘(CONS (1,1,E_Let 2 (E_Fst (E_Var 1)) (E_Let 3 (E_Snd (E_Var 1))
(E_Let 4 (E_Call 13 (E_Var 2)) (E_Call 8 (E_Pair (E_Snd (E_Snd
(E_Var 4))) (E_Pair (E_Call 9 (E_Pair (E_Inl (E_Int 0)) (E_Pair
(E_Fst (E_Snd (E_Var 4))) (E_Var 3)))) (E_Var 2)))))))) (CONS (2,1,
E_Inl (E_Var 1)) (CONS (3,1,E_Case (E_Var 1) 2 (E_Var 2) 2 E_Error)
(CONS (4,1,E_Inr (E_Inl (E_Var 1))) (CONS (5,1,E_Case (E_Var 1) 2
E_Error 2 (E_Case (E_Var 2) 3 (E_Var 3) 3 E_Error)) (CONS (6,1,
E_Inr (E_Inr (E_Var 1))) (CONS (7,1,E_Case (E_Var 1) 2 E_Error 2
```

```
(E_Case (E_Var 2) 3 E_Error 3 (E_Var 3))) (CONS (8,1,E_Let 2 (E_Fst
(E_Var 1)) (E_Let 3 (E_Fst (E_Snd (E_Var 1))) (E_Let 4 (E_Snd
(E_Snd (E_Var 1))) (E_Case (E_Var 2) 5 (E_Call 2 (E_Var 5)) 5
(E_Case (E_Var 5) 6 (E_Call 12 (E_Pair (E_Fst (E_Var 6)) (E_Pair
(E_Call 3 (E_Call 8 (E_Pair (E_Fst (E_Snd (E_Var 6))) (E_Pair
(E_Var 3) (E_Var 4))))) (E_Call 3 (E_Call 8 (E_Pair (E_Snd (E_Snd
(E_Var 6))) (E_Pair (E_Var 3) (E_Var 4)))))))) 6 (E_Case (E_Var 6)
7 (E_Call 4 (E_Pair (E_Call 8 (E_Pair (E_Fst (E_Var 7)) (E_Pair
(E_Var 3) (E_Var 4)))) (E_Call 8 (E_Pair (E_Snd (E_Var 7)) (E_Pair
(E_Var 3) (E_Var 4)))))) 7 (E_Case (E_Var 7) 8 (E_Fst (E_Call 5
(E_Call 8 (E_Pair (E_Var 8) (E_Pair (E_Var 3) (E_Var 4)))))) 8
(E_Case (E_Var 8) 9 (E_Snd (E_Call 5 (E_Call 8 (E_Pair (E_Var 9)
(E_Pair (E_Var 3) (E_Var 4)))))) 9 (E_Case (E_Var 9) 10 (E_Call 6
(E_Inl (E_Call 8 (E_Pair (E_Var 10) (E_Pair (E_Var 3) (E_Var 4))))
) 10 (E_Case (E_Var 10) 11 (E_Call 6 (E_Inr (E_Call 8 (E_Pair
(E_Var 11) (E_Pair (E_Var 3) (E_Var 4)))))) 11 (E_Case (E_Var 11)
12 (E_Case (E_Call 7 (E_Call 8 (E_Pair (E_Fst (E_Var 12)) (E_Pair
(E_Var 3) (E_Var 4))))) 13 (E_Call 8 (E_Pair (E_Fst (E_Snd (E_Snd
(E_Var 12)))) (E_Pair (E_Call 9 (E_Pair (E_Var 3) (E_Pair (E_Fst
(E_Snd (E_Var 12))) (E_Var 13)))) (E_Var 4)))) 13 (E_Call 8 (E_Pair
(E_Snd (E_Snd (E_Snd (E_Snd (E_Var 12))))) (E_Pair (E_Call 9
(E_Pair (E_Var 3) (E_Pair (E_Fst (E_Snd (E_Snd (E_Snd (E_Var 12))))
) (E_Var 13)))) (E_Var 4))))) 12 (E_Case (E_Var 12) 13 (E_Call 10
(E_Pair (E_Var 13) (E_Var 3))) 13 (E_Case (E_Var 13) 14 E_Error 14
(E_Case (E_Var 14) 15 (E_Let 16 (E_Call 11 (E_Pair (E_Fst (E_Var
15)) (E_Var 4))) (E_Call 8 (E_Pair (E_Snd (E_Var 16)) (E_Pair
(E_Call 9 (E_Pair (E_Inl (E_Int 0)) (E_Pair (E_Fst (E_Var 16))
(E_Call 8 (E_Pair (E_Snd (E_Var 15)) (E_Pair (E_Var 3) (E_Var 4))))
))) (E_Var 4))))) 15 (E_Let 16 (E_Call 8 (E_Pair (E_Fst (E_Snd
(E_Var 15))) (E_Pair (E_Var 3) (E_Var 4)))) (E_Call 8 (E_Pair
(E_Snd (E_Snd (E_Var 15))) (E_Pair (E_Call 9 (E_Pair (E_Var 3)
(E_Pair (E_Fst (E_Var 15)) (E_Var 16)))) (E_Var 4))))))))))))))))
) (CONS (9,1,E_Let 2 (E_Fst (E_Var 1)) (E_Let 3 (E_Fst (E_Snd
(E_Var 1))) (E_Let 4 (E_Snd (E_Snd (E_Var 1))) (E_Inr (E_Pair
(E_Pair (E_Var 3) (E_Var 4)) (E_Var 2)))))) (CONS (10,1,E_Let 2
(E_Fst (E_Var 1)) (E_Let 3 (E_Snd (E_Var 1)) (E_Case (E_Var 3) 4
E_Error 4 (E_Case (E_Op Equal (E_Var 2) (E_Fst (E_Fst (E_Var 4))))
5 (E_Call 10 (E_Pair (E_Var 2) (E_Snd (E_Var 4)))) 5 (E_Snd (E_Fst
(E_Var 4))))))) (CONS (11,1,E_Let 2 (E_Fst (E_Var 1)) (E_Let 3
(E_Snd (E_Var 1)) (E_Case (E_Var 3) 4 E_Error 4 (E_Case (E_Op Equal
(E_Var 2) (E_Fst (E_Fst (E_Var 4)))) 5 (E_Call 11 (E_Pair (E_Var 2)
(E_Snd (E_Var 4)))) 5 (E_Snd (E_Fst (E_Var 4))))))) (CONS (12,1,
E_Let 2 (E_Fst (E_Var 1)) (E_Let 3 (E_Fst (E_Snd (E_Var 1))) (E_Let
4 (E_Snd (E_Snd (E_Var 1))) (E_Case (E_Var 2) 5 (E_Case (E_Op Equal
(E_Var 3) (E_Var 4)) 6 (E_Call 6 (E_Inl (E_Call 2 (E_Int 0)))) 6
(E_Call 6 (E_Inr (E_Call 2 (E_Int 0))))) 5 (E_Case (E_Var 5) 6
(E_Call 2 (E_Op Add (E_Var 3) (E_Var 4))) 6 (E_Case (E_Var 6) 7
(E_Call 2 (E_Op Sub (E_Var 3) (E_Var 4))) 7 (E_Call 2 (E_Op Mul
(E_Var 3) (E_Var 4)))))))) (CONS (13,1,E_Case (E_Var 1) 2 E_Error
2 (E_Fst (E_Var 2))) NIL))))))))))))):^program‘--
```

Bibliography

A man will turn over half a library
to make one book
— DR. SAMUEL JOHNSON, 1775

[AH96] Peter H. Andersen and Carsten K. Holst. Termination analysis for offline partial evaluation of a higher order functional language. In Radhia Cousot and David A. Schmidt, editors, *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 67–82. Springer Verlag, September 1996.

[AJLW91] Myla Archer, Jefferey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors. *Proceedings of the 1991 International Tutorial and Workshop on the HOL Theorem Proving System, Davis, California*. IEEE Computer Society Press, August 1991.

[And91] Lars O. Andersen. Correctness proof for a self-interpreter. Master's-level examination project, University of Copenhagen, Denmark, December 1991.

[And92] Lars O. Andersen. C program specialization. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, May 1992. (Also DIKU Technical Report 92/14.).

[And93] Lars O. Andersen. Binding time analysis and the taming of c pointers. In Schmidt [Sch93], pages 47–58.

[And94] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, May 1994.

[AP91] Flemming Andersen and Kim D. Petersen. Recursive boolean functions in HOL. In Archer et al. [AJLW91], pages 367–377.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ulmann. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar91] Henk Barendregt. Self-interpretation in lambda calculus. *Journal of Functional Programming*, 1(2):229–233, April 1991.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BD90] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. Technical Report 90/4, DIKU, 1990. Revised version in [Bon91].

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[BEJ88] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation, Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark*. North-Holland, October 1988.

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, March 1991.

[Bon92] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California. (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. ACM, 1992.

[Bon93] Anders Bondorf. Similix 5.0 manual. Distributed with the Similix system, 1993.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, 1993. (Revised version of October 22, 1993. Also DIKU technical report 93/22.).

[BW94] Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *6th International Symposium on Programming Language Implementation and Logic Programming, Madrid, Spain*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214. Springer Verlag, September 1994.

[BW95] Lars Birkedal and Morten Welinder. Binding-time analysis for standard ML. *Lisp and Symbolic Computation*, 8:191–208, 1995.

[CAB+86] Robert L. Constable, S. Allen, H. Bromely, W. Cleveland, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[Col96] Graham Collins. A proof tool for reasoning about functional programs. In von Wright et al. [vWGH96], pages 109–124.

[CR91] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *ACM LISP Pointers*, 4(3), November 1991. (The "4" is the authors' way of denoting the fourth revision).

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Technical Report 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[DGT96] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, Selected Papers*, volume 1110 of *Lecture Notes in Computer Science*. Springer Verlag, February 1996.

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[GJ91a] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.

[GJ91b] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GM93] Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[Gom91] Carsten K. Gomard. *Program Analysis Matters*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, November 1991.

[Gom92] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.

[Gun92] Elsa L. Gunter. Why we can't have SML-style datatype declarations in HOL. In Luc J. M. Claesen and Michael J. C. Gordon, editors, *Proceedings of the IFIP TC10/WG 10.2 International Workshop on Higher Order Logic Theorem Proving and Its Applications, Leuven, Belgium*, volume A-20 of *IFIP Transactions A: Computer Science and Technology*, pages 561–568. North-Holland, 1992.

[Gun93] Elsa L. Gunter. A broader class of trees for recursive type definitions for HOL. In Joyce and Seger [JS93], pages 141–154.

[Har95] John Harrison. Inductive definitions: automation and application. In Schubert et al. [SWAF95], pages 200–213.

[Hat95] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Manuel Hermenegildo and S. D. Swierstra, editors, *7th International Symposium on Programming Language Implementation and Logic Programming, Utrecht, The Netherlands*, volume 982 of *Lecture Notes in Computer Science*, pages 279–298. Springer Verlag, September 1995.

[Hat96] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. Technical Report 95/14, University of Copenhagen, Department of Computer Science, 1996. Extended version of [Hat95].

[Hug96] John Hughes. Type specialisation for the λ-calculus. In Danvy et al. [DGT96], pages 183–215.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Program Generation*. Prentice-Hall, 1993.

[Jon97] Neil D. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1 edition, 1997.

[JS93] Jeffrey J. Joyce and Carl-Johan H. Seger, editors. *6th International Workshop on Higher Order Logic Theorem Proving and Its Applications, Vancouver, Canada*, volume 780 of *Lecture Notes in Computer Science*. Springer Verlag, August 1993.

[Kah93] Stefan Kahrs. Mistakes and ambiguities in The Definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, April 1993.

[Kah95] Stefan Kahrs. Mistakes and ambiguities in The Definition of Standard ML — addenda, February 1995.

[Kle36] Stephen C. Kleene. λ-definability and recursiveness. *Duke Math. J.*, 2:340–353, 1936.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. D. van Nostrand Company, Princeton, New Jersey, 1952.

[Kna27] B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134, 1927.

[KR88] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Prentice-Hall, 2 edition, 1988.

[Lau89] John Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, Department of Computing, University of Glasgow, November 1989.

[MC94] Thomas F. Melham and Juanito Camilleri, editors. *7th International Workshop on Higher Order Logic Theorem Proving and Its Applications, Valetta, Malta*, volume 859 of *Lecture Notes in Computer Science*. Springer Verlag, September 1994.

[Mel91] Thomas F. Melham. A package for inductive relation definitions in HOL. In Archer et al. [AJLW91], pages 350–357.

[Men79] Elliott Mendelson. *Introduction to Mathematical Logic*. D. van Nostrand Company, second edition, 1979.

[MG94] Savi Maharaj and Elsa L. Gunter. Studying the ML module system in HOL. In Melham and Camilleri [MC94], pages 346–361.

[Mil77] Robin Milner. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science*, 4(1):1–22, 1977.

[Mog88] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Bjørner et al. [BEJ88], pages 325–347.

[Mog92a] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.

[Mog92b] Torben Æ. Mogensen. Self-applicable partial evaluation of the pure lambda calculus. In ACM, editor, *Proceedings of the 1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California*, pages 116–121. ACM Press, June 1992.

[Mog93] Torben Æ. Mogensen. Constructor specialization. In Schmidt [Sch93], pages 22–33.

[Mog95] Torben Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In William L. Scherlis, editor, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California*, pages 39–44. ACM, ACM Press, June 1995.

[Mog96] Torben Æ. Mogensen. Evolution of partial evaluators: Removing inherited limits. In Danvy et al. [DGT96], pages 303–321.

[MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NN96] Dieter Nazareth and Tobias Nipkow. Formal verification of algorithm W: The monomorphic case. In von Wright et al. [vWGH96], pages 331–346.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, APIC Studies in Data Processing, pages 361–386. Academic Press, 1990.

[Pét51] Rósza Péter. *Rekursive Funktionen*. Académiai Kiadó, Budapest, Hungary, 1951.

[Pit95] Andrew M. Pitts. Operationally-based theories of program equivalence. Published for the Summer School on Semantics and Logics of Computation, University of Cambridge, Isaac Newton Institute for Mathematical Sciences, September 1995.

[Rom88] Sergei A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Bjørner et al. [BEJ88], pages 445–463.

[Rom90] Sergei A. Romanenko. Arity raiser and its use in program specialization. In Neil D. Jones, editor, *3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432 of *Lecture Notes in Computer Science*, pages 341–360. Springer Verlag, May 1990.

[Rus92] Claudio V. Russo. *Automating Mutually Recursive Type Definitions in HOL*. University of Edinburgh, June 1992. Honours Thesis.

[S+85] Richard Stallman et al. The GNU C compiler, 1985. Available from the Free Software Foundation.

[San95] David Sands. Total correctness by local improvement in program transformation. In *22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 176–234, New York, January 1995. ACM Press.

[San96] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996. Extended version of [San95].

[Sch93] David A. Schmidt, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark*. ACM Press, June 1993.

[Sli94] Konrad Slind. HOL version 90.7. Distributed as Standard ML source code from (e.g.) `ftp://ftp.research.att.com/dist/ml/hol90/`, November 1994.

[SWAF95] E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors. *8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*. Springer Verlag, September 1995.

[Sym93] Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [JS93], pages 43–60.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[VG93] Myra VanInwegen and Elsa L. Gunter. HOL-ML. In Joyce and Seger [JS93], pages 61–74.

[vW94] Joakim von Wright. Representing higher-order logic proofs in HOL. In Melham and Camilleri [MC94], pages 456–469.

[vWGH96] Joakim von Wright, J. Grundy, and John Harrison, editors. *9th international Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*. Springer Verlag, August 1996.

[Wad88] Phillip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *2nd European Symposium on Programming, Nancy, France*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer Verlag, March 1988.

[Wad90] Phillip Wadler. Deforestation: Transforming porgrams to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990. (Revised version of [Wad88]).

[Wan93] Michell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993.

[WCRS91] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In John Hughes, editor, *Proceedings of the 5th Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191. Springer Verlag, August 1991.

[Wel95] Morten Welinder. Very efficient conversions. In Schubert et al. [SWAF95], pages 340–351.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, 1993.

[Won95] Wai Wong. Recording and checking HOL proofs. In Schubert et al. [SWAF95], pages 353–368.

Index

"We seek him here, we seek him there,
Those Frenchies seek him everywhere.
Is he in heaven? — Is he in hell?
That demmed, elusive Pimpernel."
— BARONESS ORCZY, The Scarlet Pimpernel, 1865–1947

To find an entry for a mathematical entity such as "$\dot{-}$" look in the alphabetical position for its pronunciation, i.e., look for "monus." Page numbers that are underlined point to an entry's definition. An overview of function definitions can be found on page 58.