Lambda-term evalution
a calculational approach

Besjis, A.J.

Award date:
2009

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

master's thesis

Lambda-term evaluation

a calculational approach

by
Arjan Besjis

Supervisor:
dr. ir. R.R. Hoogerwoord

Eindhoven, August 2009

Acknowledgements

I am grateful to everyone that helped me, in any way, with my Master's project.

In particular, I would like to thank Rob Hoogerwoord and Jaap van der Woude for their help and their considerable display of patience and understanding, especially during the more difficult times.

As always, I am grateful to my parents for their everlasting support.

Eindhoven, August 2009

Arjan Besjis

In memory of my brother and dear friend

MARCO BESJIS

* 23 August 1973 † 16 April 2007

Contents

Chapter 1

Introduction

A lot of research effort has been put into the construction of efficient implementations for functional programming languages. One branch of implementations uses compilation techniques to transform the functional program into code for a Von Neumann type of machine, which we will call the *target machine*. As there are many different versions of Von Neumann type of machines, it is useful to abstract from the details of particular machines as much as possible. Therefore, the compilation is usually split into (at least two) compilation steps. The first step transforms the functional program into code for an *abstract machine*, the second step transforms that "abstract machine code" to code for the target machine. In this way, the details of the target machine are only relevant in the second compilation step.

Examples of abstract machines that are designed for this purpose are the ABC machine [Plasmeijer and Eekelen, 1993] and the STG machine [Jones and Salkild, 1989]. They are used in the design of compilers for the languages "Clean" and "Haskell", respectively. In [de la Encina and Peña, 2009], the STG-machine is proven to be equivalent to an extended version of Launchbury's semantics [Launchbury, 1993]. As far as we know, there is no correctness proof for the ABC-machine.

An abstract machine can be considered as an evaluator for terms from the λ-calculus, so-called λ-terms. The λ-calculus [Barendregt, 1984] is a well-known theory of computation that is widely considered as the basis of functional programming. A well-known evaluator for λ-terms is the "normal order" evaluator. It calculates the "normal form" of any λ-term, provided such a normal form exists. However, a straightforward implementation of a normal order evaluator is very inefficient.

In 1990, Rob Hoogerwoord used calculational techniques from transformational/ functional programming to *transform* the normal order evaluator into an equivalent but more efficient one that resembles an abstract machine. That work has never been fully completed. In this thesis, we will attempt to complete that work. We present the following:

Problem statement

> Taking normal order evaluation as a starting point, derive an abstract machine, in a fully calculational way, and such that its correctness is beyond doubt. Here "correctness" means: equivalent to normal order evaluation.

End of problem statement.

A function represents an abstract machine if it fits the following description. The function's definition is tail recursive and the function has two parameters: one representing the "state" of the machine, the other representing the machine "code", that is, a sequence of "instructions". Application of the function to a state, x say, and an instruction sequence, ins say, yields x if ins is empty, and if ins is not empty, it yields a recursive application of the function to a new state, y say, and the tail of sequence ins. The tail of a sequence is what remains after removing the first element of the sequence. The new state y represents the result of "executing" the first instruction of ins in state x.

In what follows, we first present a few notions and notations, followed by a formal specification of a normal order evaluator. In the subsequent chapters, we transform this evaluator to increasingly more efficient versions while retaining correctness. The last transformation yields a tail-recursive evaluator which can be considered as a description of an abstract machine.

Chapter 2

Notions and notations

We adopt the style of calculational programming as presented in [Hoogerwoord, 2007]. This chapter is a (brief) summary of some notions and notations from that work which are relevant for this thesis.

Regarding the use of the predicate calculus [Dijkstra and Scholten, 1990] we note the following; to indicate the scope of a quantification we will use angular brackets, i.e. "$\langle\ \rangle$", instead of (the perhaps more conventional) parentheses, i.e. "$(\ )$".

We will sometimes use a function that is not yet defined or make an assumption, in a hint of a derivation. Such assumptions (or use of functions) will be prefixed with a $\star$ to emphasise that something "new" is assumed or introduced.

2.1 Basic functional programming

The set Ω contains all possible values of functional expressions. In this chapter dummies x, y, z have type Ω.

2.1.1 Function application

We use the infix-operator $\cdot$ ("dot"), to denote function application; operator $\cdot$ is left-binding, i.e.:

$$x \cdot y \cdot z \quad \text{is read as} \quad (x \cdot y) \cdot z \ .$$

2.1.2 Function type

A type is a subset of Ω. For types X and Y, we denote the type of functions from X to Y as $X \to Y$, hence (for all f in Ω):

$$f \in X \to Y \quad \equiv \quad \langle \forall x : x \in X : \ f \cdot x \in Y \ \rangle \ .$$

The "$\to$" in a function type is right-binding, i.e.:

$$X \to Y \to Z \quad \text{is read as} \quad X \to (Y \to Z) \ .$$

2.1.3 Specification and declaration

A *specification* is a formalisation of a problem; a *declaration* is a solution to that problem. The latter is a definition, of a restricted shape, of a value in Ω. Informally, a declaration is a definition with such a fine grain of detail that it can be considered as "code" for a functional evaluator.

2.1.4 Where-clauses

A so-called *where-clause* is a way to bind a declaration to an expression. For example, consider the following declaration:

$$square \cdot n \;=\; product \cdot n \cdot n \;\; \textbf{whr} \;\; product \cdot k \cdot l \;=\; k \times l \;\; \textbf{end} \;\; .$$

The where-clause binds the declaration of function *product* to expression $product \cdot n \cdot n$.

2.1.5 Guarded selections

Expression:

$$B \;\rightarrow\; E \;,$$

is called a *guarded expression*. If (the value of) B is TRUE then the value of $B \rightarrow E$ is the value of E, otherwise the value of $B \rightarrow E$ is undetermined. The combination of several guarded expressions into one single expression is called a *guarded selection*. For example, the following declaration uses a guarded selection to represent case distinction on natural n:

$$\begin{array}{lll} f \cdot n \;=\; \textbf{if} & n = 0 \;\rightarrow\; 3 \\ \quad [\!] & n = 1 \;\rightarrow\; 9 \\ \quad [\!] & n \geq 2 \;\rightarrow\; f \cdot x \;+\; f \cdot (x+1) \;\; \textbf{whr} \;\; x \;=\; n-2 \;\; \textbf{end} \\ \quad \textbf{fi} & \end{array}$$

We usually abbreviate such declarations to:

$$\begin{aligned} f \cdot 0 \;&=\; 3 \\ f \cdot 1 \;&=\; 9 \\ f \cdot (n+2) \;&=\; f \cdot n \;+\; f \cdot (n+1) \end{aligned}$$

2.1.6 Operator sections

For any binary operator $\oplus$ and constants a, b (of the right type) we consider a function, f say, that maps its argument to an application of $\oplus$, e.g.:

$$f \cdot x \;=\; a \oplus x \;.$$

We use a short-hand for such functions, specifically, we write $(a \oplus)$ for f; there are similar short-hands for other applications of $\oplus$, namely, for all x and y we have:

$$\begin{aligned} (a \oplus) \cdot x \;&=\; a \oplus x \;, \\ (\oplus b) \cdot y \;&=\; y \oplus b \;, \\ (\oplus) \cdot y \cdot x \;&=\; y \oplus x \;. \end{aligned}$$

2.1.7 Tuples

For any natural n, the combination of n values into a singe item is called an *n-tuple*. A tuple formed from expressions E_i, $0 \leq i < n$, is denoted as follows:

$$\langle E_0, \cdots, E_n \rangle \;.$$

An n-tuple can be regarded as a mapping from $[0, n)$ to the set of E_i, $0 \leq i < n$. Hence, for all natural i satisfying $0 \leq i < n$, we have:

$$\langle E_0, \cdots, E_n \rangle \cdot i \;\;=\;\; E_i \;.$$

2.1.8 Function composition

We use the infix-operator $\circ$ to denote *function composition*, hence, for all x and functions f, g of type $\Omega \to \Omega$ we have:

$$(f \circ g) \cdot x \;\;=\;\; f \cdot (g \cdot x) \;.$$

2.2 Finite lists

For some type X and natural n, the datatype of lists of length n containing elements of X is denoted by $\mathcal{L}_n(X)$. The datatype $\mathcal{L}_*(X)$ is the union of $\mathcal{L}_n(X)$ over all n, i.e. it is the type of all finite lists of X's. When we are not interested in the type of the elements of a list, we abbreviate $\mathcal{L}_n(X)$ to $\mathcal{L}_n$. In this section, dummy i has type $\mathbb{N}$, dummies x, y have type X and dummies xs, ys have type $\mathcal{L}_*(X)$.

Operator $\#$ ("size") maps a list to the length of that list, hence, for all natural n, operator $\#$ has type $\mathcal{L}_n \to \mathbb{N}$. For all xs we have:

$$xs \in \mathcal{L}_n \;\;\Rightarrow\;\; \#xs \;=\; n \;.$$

The empty list is denoted by "$[\,]$" and its the only element in $\mathcal{L}_0$, i.e.:

$$\mathcal{L}_0 \;=\; \{\,[\,]\,\}\,.$$

For all natural n, operator $\triangleright$ ("cons") has type $X \times \mathcal{L}_n(X) \to \mathcal{L}_{(n+1)}(X)$. Operator $\triangleright$ is right-binding and has the following properties, for all x, i and xs:

$$\begin{array}{rcll} x \triangleright xs & \neq & [\,] & \\ (x \triangleright xs)\cdot 0 & = & x & \\ (x \triangleright xs)\cdot(i+1) & = & xs\cdot i & \text{, if } i < \#xs \end{array}$$

We will use the following abbreviations when we write out a list and its elements explicitly:

$$\begin{array}{rcl} [x] & = & x \triangleright [\,]\,, \\ [x,y] & = & x \triangleright y \triangleright [\,]\,, \\ & \vdots & \end{array}$$

For all natural n, operator $\triangleleft$ ("snoc") has type $\mathcal{L}_n(X) \times X \to \mathcal{L}_{(n+1)}(X)$. Operator $\triangleleft$ is left-binding and has the following properties, for all y, i and ys we have:

$$\begin{array}{rcll} ys \triangleleft y & \neq & [\,] & \\ (ys \triangleleft y)\cdot n & = & y & \text{, if } n = \#ys \\ (ys \triangleleft y)\cdot i & = & ys\cdot i & \text{, if } i < \#ys \\ [\,] \triangleleft y & = & [y] & \end{array}$$

For all naturals m and n, operator $+\!\!+$ ("concatenation") has type $\mathcal{L}_m \times \mathcal{L}_n \to \mathcal{L}_{(m+n)}$. For all x, y, i, xs and ys, operator $+\!\!+$ has the following properties:

$$\begin{array}{rcll} (xs +\!\!+ ys)\cdot i & = & xs\cdot i & \text{, if } i < \#xs \\ (xs +\!\!+ ys)\cdot(m+i) & = & ys\cdot i & \text{, if } m = \#xs \;\wedge\; i < \#ys \\ (x \triangleright xs) +\!\!+ ys & = & x \triangleright (xs +\!\!+ ys) & \\ xs +\!\!+ (ys \triangleleft y) & = & (xs +\!\!+ ys) \triangleleft y & \\ [x] +\!\!+ xs & = & x \triangleright xs & \\ ys +\!\!+ [y] & = & ys \triangleleft y & \end{array}$$

We denote the set of the elements in list xs by $[\![xs]\!]$. We have the following properties:

$$\begin{array}{rcl} [\![\,[\,]\,]\!] & = & \emptyset \\ [\![x \triangleright xs]\!] & = & \{x\} \cup [\![xs]\!] \\ [\![ys \triangleleft y]\!] & = & [\![ys]\!] \cup \{y\} \\ [\![xs +\!\!+ ys]\!] & = & [\![xs]\!] \cup [\![ys]\!] \end{array}$$

Operator $\lceil$ ("take") maps a list, xs say, and a natural, i say, to xs's prefix of length i. We have, for all x, i and xs:

$$\begin{array}{rcll} xs \lceil 0 & = & [\,] & \\ (x \triangleright xs) \lceil (i+1) & = & x \triangleright xs \lceil i & , \text{ if } i \leq \#xs \end{array}$$

The counterpart of $\lceil$ is operator $\lfloor$ ("drop"). Application $xs \lfloor i$ yields the part of xs that remains after removal of its prefix of length i. We have, for all x, i and xs:

$$\begin{array}{rcll} xs \lfloor 0 & = & xs & \\ (x \triangleright xs) \lfloor (i+1) & = & xs \lfloor i & , \text{ if } i \leq \#xs \\ xs \lceil i \mathbin{+\!\!+} xs \lfloor i & = & xs & , \text{ if } i \leq \#xs \end{array}$$

Consider a function, f say, and a list, xs say. The application of f to every element of xs is denoted by $f \bullet xs$; operator $\bullet$ is called "map". For types X and Z, and for all natural n, operator $\bullet$ has type $(X \to Z) \times \mathcal{L}_n(X) \to \mathcal{L}_n(Z)$. For all x, y, i, xs, ys and f ($f \in X \to Z$) we have:

$$\begin{array}{rcll} f \bullet [\,] & = & [\,] & \\ (f \bullet xs) \cdot i & = & f \cdot (xs \cdot i) & , \text{ if } i < \#xs \\ f \bullet (x \triangleright xs) & = & f \cdot x \triangleright f \bullet xs & \\ f \bullet (ys \triangleleft y) & = & f \bullet ys \triangleleft f \cdot y & \\ f \bullet (xs \mathbin{+\!\!+} ys) & = & f \bullet xs \mathbin{+\!\!+} f \bullet ys & \end{array}$$

2.3 Folded operators

For some types B and Y, binary operator $\oplus$ has type $B \times Y \to B$, dummy b has type B and dummies x, y, z have type Y. We can form expressions, of type B, using $\oplus$ as follows:

$$b\,, \quad b \oplus x\,, \quad (b \oplus x) \oplus y\,, \quad ((b \oplus x) \oplus y) \oplus z\,, \quad \text{etc.}$$

These expressions have a similar pattern, they all have one element of B and zero or more elements of Y. Thus, we can represent expressions with $\oplus$ using one B, a list of Y's and a new binary operator, $\otimes$ say, with type $B \times \mathcal{L}_*(Y) \to B$ and definition (for all ys, $ys \in \mathcal{L}_*(Y)$):

$$\begin{array}{rcl} b \otimes [\,] & = & b\,, \\ b \otimes (y \triangleright ys) & = & (b \oplus y) \otimes ys\,. \end{array}$$

The above expressions with $\oplus$ may be represented using $\otimes$ as follows:

$$b \otimes [\,]\,, \quad b \otimes [x]\,, \quad b \otimes [x, y]\,, \quad b \otimes [x, y, z]\,, \quad \text{etc.}$$

We call $\otimes$ the *left-folded* version of $\oplus$. For binary operators of type $Y \times B \to B$, *right-folded* operators with type $\mathcal{L}_*(Y) \times B \to B$ can be defined in a similar fashion. We will mostly use left-folded operators.

We have the following properties of $\otimes$, for all b, y, xs and ys ($xs, ys \in \mathcal{L}_*(Y)$):

$$b \otimes (ys \triangleleft y) = (b \otimes ys) \oplus y \,, \tag{2.1}$$

$$b \otimes (xs +\!\!+ ys) = (b \otimes xs) \otimes ys \,. \tag{2.2}$$

Chapter 3

The specification

We use the untyped lambda calculus [Barendregt, 1984] to specify our evaluator. This chapter presents that specification. For proofs of claims made about the lambda calculus we refer to [Barendregt, 1984].

3.1 Syntax

We define $\mathcal{V}$ to be the infinite set of variables and Λ the set of λ-terms. In this thesis, dummies u, v have type $\mathcal{V}$ and dummies E, F, G have type Λ. The set of λ-terms is defined as the smallest set satisfying:

$$\begin{array}{rll} v \in \Lambda & \text{, for all } v \in \mathcal{V} & \text{(variable)} \\ v \prec E \in \Lambda & \text{, for all } v \in \mathcal{V} \ \wedge \ E \in \Lambda & \text{(abstraction)} \\ E \odot F \in \Lambda & \text{, for all } E, F \in \Lambda & \text{(application)} \end{array}$$

The functions var, $abst$ and $appl$ have type $\Lambda \to Bool$. They will be used to specify conditions on the structure of λ-terms. Their definitions are:

$$var \cdot E \ \equiv \ E \in \mathcal{V} \,, \tag{3.1}$$
$$abst \cdot E \ \equiv \ \langle \exists v, F :: \ E = v \prec F \ \rangle \,, \tag{3.2}$$
$$appl \cdot E \ \equiv \ \langle \exists F, G :: \ E = F \odot G \ \rangle \,. \tag{3.3}$$

Operator $\#$ ("size of") on λ-terms is defined such that $\#E$ represents the size of E. Operator $\#$ has type $\Lambda \to \mathbb{N}$ and definition:

$$\#v \ = \ 1 \tag{3.4}$$
$$\#(v \prec E) \ = \ 1 + \#E \tag{3.5}$$
$$\#(E \odot F) \ = \ 1 + \#E + \#F \tag{3.6}$$

3.2 Alpha equality

The set of *free variables* of λ-term E is defined as $fv \cdot E$, where function fv has type $\Lambda \to \mathrm{Set}(\mathcal{V})$ and definition:

$$fv \cdot v = \{v\} \tag{3.7}$$
$$fv \cdot (v \prec E) = fv \cdot E \setminus \{v\} \tag{3.8}$$
$$fv \cdot (E \odot F) = fv \cdot E \cup fv \cdot F \tag{3.9}$$

If variable v satisfies $v \in fv \cdot E$ then we call v a *free variable* of E. If variable v occurs in λ-term E such that $v \notin fv \cdot E$ then we call v a *bound variable* of E.

Due to the nature of reduction on λ-terms it is useful to define an equality relation on Λ that identifies two λ-terms if they are equal modulo bound variable renaming. This relation is called *alpha equality* and we denote it by $=_\alpha$. Alpha equality is a congruence relation, it partitions set Λ into a set of equivalence classes. We denote that set by $\Lambda \,/=_\alpha$. Every element of $\Lambda \,/=_\alpha$ is a set of λ-terms, we consider all λ-terms in such a set as *equal*. In other words, we write for all E, F:

$$E = F \quad \equiv \quad E =_\alpha F \,.$$

3.3 Beta reduction

Reduction of a λ-term is the process of repeatedly rewriting subexpressions in that λ-term. Such subexpressions are usually called *reducible subexpressions*, or *redexes*. For *beta reduction*, a redex is defined as a subexpression of the shape:

$$(v \prec E) \odot F \,,$$

and such redexes are rewritten to:

"E with F substituted for every free occurrence of v in E" ,

which we will denote by:

$$E \leftarrow \langle v, F \rangle \,,$$

where binary operator $\leftarrow$ has type $\Lambda \times \langle \mathcal{V}, \Lambda \rangle \to \Lambda$ and definition:

$$v \leftarrow \langle u, G \rangle = v \qquad \text{, if } u \neq v \tag{3.10}$$
$$u \leftarrow \langle u, G \rangle = G \tag{3.11}$$
$$(v \prec E) \leftarrow \langle u, G \rangle = v \prec (E \leftarrow \langle u, G \rangle) \qquad \text{, if } v \neq u \wedge v \notin fv \cdot G \tag{3.12}$$
$$(E \odot F) \leftarrow \langle u, G \rangle = (E \leftarrow \langle u, G \rangle) \odot (F \leftarrow \langle u, G \rangle) \tag{3.13}$$

The condition to (3.12) can always be satisfied by renaming bound variable v in:

$$v \prec E \,,$$

because we consider λ-terms equal modulo bound-variable renaming.

3.3.1 Reduction

Relation $\rightarrow_\beta$ ("one step β-reduction") is defined on Λ as the strongest relation satisfying, for all v, E, F and G:

$$(v \prec E) \odot F \quad \rightarrow_\beta \quad E \leftarrow \langle v, F \rangle \,, \tag{3.14}$$

$$E \rightarrow_\beta F \quad \Rightarrow \quad (v \prec E) \rightarrow_\beta (v \prec F) \,, \tag{3.15}$$

$$E \rightarrow_\beta F \quad \Rightarrow \quad (E \odot G) \rightarrow_\beta (F \odot G) \,, \tag{3.16}$$

$$E \rightarrow_\beta F \quad \Rightarrow \quad (G \odot E) \rightarrow_\beta (G \odot F) \,. \tag{3.17}$$

Relation $\twoheadrightarrow_\beta$ ("zero or more steps β-reduction") is the reflexive transitive closure of $\rightarrow_\beta$, i.e. $\twoheadrightarrow_\beta$ is the strongest relation satisfying, for all E, F and G:

$$E \rightarrow_\beta F \quad \Rightarrow \quad E \twoheadrightarrow_\beta F \,, \tag{3.18}$$

$$E \twoheadrightarrow_\beta E \,, \tag{3.19}$$

$$E \twoheadrightarrow_\beta F \;\wedge\; F \twoheadrightarrow_\beta G \quad \Rightarrow \quad E \twoheadrightarrow_\beta G \,. \tag{3.20}$$

Relation $=_\beta$ ("β-equality") is the reflexive symmetric transitive closure of $\rightarrow_\beta$, i.e. $=_\beta$ is the strongest relation satisfying, for all E, F and G:

$$E \rightarrow_\beta F \quad \Rightarrow \quad E =_\beta F \,, \tag{3.21}$$

$$E =_\beta E \,, \tag{3.22}$$

$$E =_\beta F \quad \Rightarrow \quad F =_\beta E \,, \tag{3.23}$$

$$E =_\beta F \;\wedge\; F =_\beta G \quad \Rightarrow \quad E =_\beta G \,. \tag{3.24}$$

A *β-reduction* (or just *reduction*) is a finite or infinite sequence

$$E \rightarrow_\beta F \rightarrow_\beta G \rightarrow_\beta \cdots$$

3.3.2 Normal form

A λ-term without redexes is in *normal form*. Function nf is a predicate on Λ that formalises this. It has type $\Lambda \rightarrow Bool$ and definition:

$$nf \cdot E \;\equiv\; \neg \langle \exists F :: E \rightarrow_\beta F \rangle \,. \tag{3.25}$$

Rewriting a redex in a λ-term can add redexes to that λ-term; as a result, a reduction may be infinite. For example, the following (well-known) λ-term yields an infinite reduction:

$$(v \prec (v \odot v)) \;\odot\; (v \prec (v \odot v)) \,.$$

A λ-term *has a normal form* if there exists a finite reduction for that λ-term. I.e., λ-term E has a normal form if there exists a λ-term *in* normal form that is β-equal to E. We define function $hasnf$ of type $\Lambda \rightarrow Bool$ as follows:

$$hasnf \cdot E \;\equiv\; \langle \exists F : nf \cdot F : E =_\beta F \rangle \,. \tag{3.26}$$

A λ-term has *at most one* normal form, i.e.:

$$nf \cdot E \ \wedge \ nf \cdot F \ \wedge \ E =_\beta F \quad \Rightarrow \quad E \ = \ F \ . \tag{3.27}$$

This is a (well-known) consequence of the *Church-Rosser* theorem, which states:

$$E =_\beta F \quad \Rightarrow \quad \langle \exists G :: \ E \twoheadrightarrow_\beta G \ \wedge \ F \twoheadrightarrow_\beta G \ \rangle \ . \tag{3.28}$$

Another consequence of this theorem is:

$$nf \cdot F \quad \Rightarrow \quad (E \twoheadrightarrow_\beta F \ \equiv \ E =_\beta F) \ . \tag{3.29}$$

3.3.3 Reduction strategy

A *reduction strategy* prescribes which redex is reduced if a λ-term contains more than one. Such a strategy can yield an infinite reduction when applied to a λ-term, while a different strategy, applied to the same λ-term, may produce a finite reduction. A reduction strategy is *normalizing* when the reduction is finite for every λ-term that has a normal form. For beta reduction, the *normal order* reduction strategy is normalizing. It is defined as:

> reduce the leftmost outermost redex first.

3.4 The specification

We are now ready to present the specification of our evaluator. Partial function *red0* ("reduce") maps a λ-term that has a normal form to that normal form, i.e. we have the:

Specification of function *red0*

> Partial function *red0* has type $\Lambda \rightarrow \Lambda$ and specification:
>
> $$\langle \forall E : hasnf \cdot E : \ E \twoheadrightarrow_\beta red0 \cdot E \ \ \wedge \ \ nf \cdot (red0 \cdot E) \ \rangle \ .$$

End of specification of function *red0*.

Note that by (3.29) this is equivalent to:

$$\langle \forall E : hasnf \cdot E : \ E =_\beta red0 \cdot E \ \ \wedge \ \ nf \cdot (red0 \cdot E) \ \rangle \ .$$

Chapter 4

A declaration

We derive two declarations for function *red0* . The first is small and simple but inefficient. The second declaration lacks some of the inefficiencies of the first and indicates that a generalization is possible. It will act as a stepping stone towards the next chapter.

4.1 First attempt

We observe, from the specification of *red0* , that λ-term $red0 \cdot E$ is in normal form. If λ-term E is also in normal form then:

$$E \twoheadrightarrow_\beta red0 \cdot E \ ,$$

is equivalent to:

$$E = red0 \cdot E \ ,$$

because a λ-term has at most one normal form. Hence, we have:

$$nf \cdot E \quad \Rightarrow \quad red0 \cdot E = E \ .$$

We consider the other case, $\neg nf \cdot E$:

$$\begin{array}{ll} & E \twoheadrightarrow_\beta red0 \cdot E \\ \equiv & \quad \{ \text{ definition of } nf \ (3.25), \text{ using } \neg nf \cdot E \ \} \\ & E \twoheadrightarrow_\beta red0 \cdot E \ \wedge \ \langle \exists G :: \ E \rightarrow_\beta G \ \rangle \\ \equiv & \quad \{ \ \wedge \text{ over } \vee \ \} \\ & \langle \exists G :: \ E \twoheadrightarrow_\beta red0 \cdot E \ \ \wedge \ \ E \rightarrow_\beta G \ \rangle \\ \Leftarrow & \quad \{ \text{ transitivity of } \twoheadrightarrow_\beta \ \} \end{array}$$

$\langle \exists G :: \; E \rightarrow_\beta G \;\; \wedge \;\; G \twoheadrightarrow_\beta red0 \cdot E \; \rangle$

$\Leftarrow$ { $\star$ instantiation: $G := rd1 \cdot E$, where $rd1$ is a function of type $\Lambda \rightarrow \Lambda$ }

$E \rightarrow_\beta rd1 \cdot E \;\; \wedge \;\; rd1 \cdot E \twoheadrightarrow_\beta red0 \cdot E$

$\equiv$ { we choose the left conjunct as specification for $rd1$, see below }

$rd1 \cdot E \twoheadrightarrow_\beta red0 \cdot E$

$\Leftarrow$ { transitivity of $\twoheadrightarrow_\beta$, moving towards an appeal to an induction hypothesis }

$rd1 \cdot E \twoheadrightarrow_\beta red0 \cdot (rd1 \cdot E) \;\; \wedge \;\; red0 \cdot (rd1 \cdot E) \twoheadrightarrow_\beta red0 \cdot E$

$\equiv$ { specification of $red0$, by induction hypothesis, see below, using lemma 4.2, see also below }

$red0 \cdot (rd1 \cdot E) \twoheadrightarrow_\beta red0 \cdot E$

$\equiv$ { both sides are in normal form }

$red0 \cdot (rd1 \cdot E) \; = \; red0 \cdot E$.

Combining both cases, for all E such that $hasnf \cdot E$, we have the:

Declaration for function *red0*

$$\begin{array}{lll} red0 \cdot E \; = & \textbf{if} \;\; nf \cdot E \;\; \rightarrow \;\; E & \\ & [] \;\; \neg nf \cdot E \;\; \rightarrow \;\; red0 \cdot (rd1 \cdot E) & \\ & \textbf{fi} & \end{array} \tag{4.1}$$

End of declaration for function *red0*.

We used mathematical induction to derive (4.1) from the specification of function $red0$. This is only valid if $red0$ implements a normalizing reduction (i.e. a finite reduction), which it does, if function $rd1$ implements the normal order reduction strategy. Therefore, we have the following:

Specification of function *rd1*

Partial function $rd1$ has type $\Lambda \rightarrow \Lambda$ and specification:

$\langle \forall E : \neg nf \cdot E : \; E \rightarrow_\beta rd1 \cdot E \; \rangle$,

"$rd1$ implements the normal order reduction strategy" .

End of specification of function *rd1*.

We also used:

Lemma 4.2 For all E:

$$\neg nf \cdot E \quad \Rightarrow \quad (hasnf \cdot E \;\equiv\; hasnf \cdot (rd1 \cdot E)) \; .$$

End of lemma 4.2.

This follows from the definition of *hasnf*.

4.2 Efficiency

It is useful to consider the following declarations for functions *nf* and *rd1*, because they provide us with a better understanding of the (in)efficiency of (the declaration for) function *red0*. They satisfy the definition of *nf* and the specification of *rd1*, by induction on the structure of their arguments.

Declaration for function *nf*

$$nf \cdot v \;=\; \text{TRUE} \tag{4.3}$$
$$nf \cdot (v \prec E) \;=\; nf \cdot E \tag{4.4}$$
$$nf \cdot ((v \prec E) \odot F) \;=\; \text{FALSE} \tag{4.5}$$
$$nf \cdot (E \odot F) \;=\; nf \cdot E \;\wedge\; nf \cdot F \qquad \text{, if } \neg abst \cdot E \tag{4.6}$$

End of declaration for function *nf*.

Declaration for function *rd1*

$$rd1 \cdot (v \prec E) \;=\; v \prec rd1 \cdot E \tag{4.7}$$
$$rd1 \cdot ((v \prec E) \odot F) \;=\; E \leftarrow \langle v, F \rangle \tag{4.8}$$
$$rd1 \cdot (E \odot F) \;=\; E \odot (rd1 \cdot F) \qquad \text{, if } \neg abst \cdot E \wedge nf \cdot E \tag{4.9}$$
$$rd1 \cdot (E \odot F) \;=\; (rd1 \cdot E) \odot F \qquad \text{, if } \neg abst \cdot E \wedge \neg nf \cdot E \tag{4.10}$$

End of declaration for function *rd1*.

Consider the declaration for *rd1*, every instance of $rd1 \cdot E$ traverses (the tree that is represented by) λ-term E to identify a redex to rewrite. Application $red0 \cdot E$ can boil down to:

$$rd1 \cdot (rd1 \cdot (rd1 \cdot (\ldots rd1 \cdot (rd1 \cdot E) \ldots))) \; .$$

The evaluation of $rd1 \cdot (rd1 \cdot E)$ yields a traversal of E to identify "the first" redex to rewrite, followed by another traversal of the *entire* λ-term (represented by) $rd1 \cdot E$, to identify "the next" redex to rewrite. This is inefficient because E and $rd1 \cdot E$ are equal modulo the rewriting of (only) one redex.

Regarding the broader view, the main source of inefficiency is the substitution operation (not surprisingly as substitution is the basis of β-reduction). Substitutions are expensive and not always necessary. Also, substitution is likely to duplicate subterms which are evaluated separately.

This is by no means an exhaustive enumeration of the (in)efficiencies of (the declaration for) function *red0* . In what follows, we will address most of these issues.

4.3 Induction on the reduction of a λ-term

In the rest of this thesis, we will be more explicit about the form of induction that we use. Variant (partial) function *vfr0* , which maps a λ-term (that has a normal from) to the length of its normal order reduction, has the following:

Definition of function *vfr0*

> Partial function *vfr0* has type $\Lambda \to \mathbb{N}$ and application $vfr0 \cdot E$ has precondition $hasnf \cdot E$. Function *vfr0* has definition:
>
> $$\begin{array}{lll} vfr0 \cdot E \;=\; \textbf{if} & nf \cdot E & \to \;\; 0 \\ \;[\!] & \neg nf \cdot E & \to \;\; 1 + vfr0 \cdot (rd1 \cdot E) \\ \textbf{fi} & & \end{array}$$

End of definition of function *vfr0* .

We use function *vfr0* to define "induction on the reduction of a λ-term", for all predicates P on Λ, as follows:

$$\begin{array}{ll} & \langle \forall E : hasnf \cdot E : \; P \cdot E \; \rangle \\ \Leftarrow & \quad \{ \text{ induction on the reduction of } E \; \} \\ & \langle \forall E : hasnf \cdot E : \; \langle \forall F : hasnf \cdot F \; \wedge \; vfr0 \cdot F < vfr0 \cdot E : \; P \cdot F \; \rangle \;\; \Rightarrow \;\; P \cdot E \; \rangle \; . \end{array}$$

4.4 Second attempt

We try to derive a more efficient declaration for function *red0* , by case distinction on its argument. The goal of this derivation is a declaration that does not depend on functions *nf* and *rd1* anymore, thereby avoiding superfluous tree traversals.

We use the previous declaration of *red0* in the derivation of the new declaration. To distinguish the two, we introduce a new (partial) function, *red1* say, with the following:

Specification of function *red1*

Partial function *red1* has type $\Lambda \to \Lambda$ and specification:

$$\langle \forall E : \mathit{hasnf} \cdot E : \ \mathit{red1} \cdot E \ = \ \mathit{red0} \cdot E \ \rangle \ .$$

End of specification of function *red1*.

The derivation of a recursive declaration for a function, by case distinction on the structure of (one of) its argument(s), requires (in general) structural induction on that argument. In this case, structural induction is not sufficient, as the β-reduction of a λ-term does not necessarily result in a (structurally) smaller λ-term. Hence, we *also* use induction on the reduction of the argument. To formalise this form of induction, we use order relation $\trianglelefteq$ and a variant (partial) function, called *vfr1* , with the following:

Definition of function *vfr1*

Partial function *vfr1* has type $\Lambda \to \mathcal{L}_2(\mathbb{N})$ and definition, for all E such that $\mathit{hasnf} \cdot E$:

$$\mathit{vfr1} \cdot E \ = \ [\mathit{vfr0} \cdot E] +\!\!+ [\#E] \ .$$

End of definition of function *vfr1*.

We use a list instead of a pair as the type of $\mathit{vfr1} \cdot E$, because (in the following chapters) the need will arise to extend (the list represented by) $\mathit{vfr1} \cdot E$ and it is easier to extend a list than a pair.

Definition of relation $\trianglelefteq$

For all positive natural n, relation $\trianglelefteq$ is the lexicographic order on $\mathcal{L}_n(\mathbb{N})$.

End of definition of relation $\trianglelefteq$.

* * *

We will use the following two lemmas in the derivation:

Lemma 4.11 For all v and E such that $\mathit{hasnf} \cdot (v \prec E)$:

$$\mathit{red0} \cdot (v \prec E) \ = \ v \prec (\mathit{red0} \cdot E) \ .$$

End of lemma 4.11.

Lemma 4.12 For all E and F:

$$\begin{array}{ll} & \mathit{hasnf} \cdot (E \odot F) \ \wedge \ \neg \mathit{abst} \cdot E \ \wedge \ \mathit{nf} \cdot E \\ \Rightarrow & \\ & \mathit{red0} \cdot (E \odot F) \ = \ E \odot (\mathit{red0} \cdot F) \ . \end{array}$$

End of lemma 4.12.

Lemma 4.11 can be proved by induction on the reduction of E. Lemma 4.12 can be proved by induction on the reduction of F.

* * *

The following five cases comprise the derivation of a declaration for function *red1* .

Case $E := v$:

$$\begin{array}{ll} & \mathit{red1} \cdot v \\ = & \quad \{ \text{ specification of } \mathit{red1} \ \} \\ & \mathit{red0} \cdot v \\ = & \quad \{ \text{ declaration of } \mathit{red0} \ (4.1), \text{ using } \mathit{nf} \cdot v \text{, by } (4.3) \ \} \\ & v \ . \end{array}$$

Case $E := (v \prec E)$, assuming $\mathit{hasnf} \cdot (v \prec E)$:

$$\begin{array}{ll} & \mathit{red1} \cdot (v \prec E) \\ = & \quad \{ \text{ specification of } \mathit{red1} \ \} \\ & \mathit{red0} \cdot (v \prec E) \\ = & \quad \{ \text{ lemma 4.11, using } \mathit{hasnf} \cdot (v \prec E) \ \} \\ & v \prec (\mathit{red0} \cdot E) \\ = & \quad \{ \text{ specification of } \mathit{red1}, \text{ by induction hypothesis, see below } \} \\ & v \prec (\mathit{red1} \cdot E) \ . \end{array}$$

The use of the induction hypothesis requires:

$$\mathit{vfr1} \cdot E \ \trianglelefteq \ \mathit{vfr1} \cdot (v \prec E) \quad \text{and:} \quad \mathit{hasnf} \cdot (v \prec E) \ \Rightarrow \ \mathit{hasnf} \cdot E \ .$$

The latter follows from the definition of *hasnf* , the former is satisfied by:

$$\mathit{vfr0} \cdot (v \prec E) \ = \ \mathit{vfr0} \cdot E \quad \text{and:} \quad \#E \ < \ \#(v \prec E) \ .$$

The former can be proved by induction on the reduction of E, the latter follows from the definition of $\#$ (3.5).

Case $E := ((v \prec E) \odot F)$, assuming $\mathit{hasnf} \cdot ((v \prec E) \odot F)$:

$$
\begin{array}{ll}
 & \mathit{red1} \cdot ((v \prec E) \odot F) \\
= & \quad \{ \text{ specification of } \mathit{red1} \ \} \\
 & \mathit{red0} \cdot ((v \prec E) \odot F) \\
= & \quad \{ \text{ declaration of } \mathit{red0} \ (4.1), \text{ using } \neg \mathit{nf} \cdot ((v \prec E) \odot F), \text{ by } (4.5) \ \} \\
 & \mathit{red0} \cdot (\ \mathit{rd1} \cdot ((v \prec E) \odot F)\ ) \\
= & \quad \{ \text{ declaration of } \mathit{rd1} \ (4.8) \ \} \\
 & \mathit{red0} \cdot (E \leftarrow \langle v, F \rangle) \\
= & \quad \{ \text{ specification of } \mathit{red1}, \text{ by induction hypothesis, see below } \} \\
 & \mathit{red1} \cdot (E \leftarrow \langle v, F \rangle) \ .
\end{array}
$$

The use of the induction hypothesis requires:

$$\mathit{vfr1} \cdot (E \leftarrow \langle v, F \rangle) \ \trianglelefteq \ \mathit{vfr1} \cdot ((v \prec E) \odot F) \ ,$$

and:

$$\mathit{hasnf} \cdot ((v \prec E) \odot F) \ \Rightarrow \ \mathit{hasnf} \cdot (E \leftarrow \langle v, F \rangle) \ .$$

Using the definition of $\mathit{rd1}$ (4.8), this is satisfied by the definitions of $\mathit{vfr1}$, $\trianglelefteq$ and $\mathit{vfr0}$, and lemma 4.2.

Case $E := (E \odot F)$, assuming $\mathit{hasnf} \cdot (E \odot F)$, $\neg \mathit{abst} \cdot E$ and $\mathit{nf} \cdot E$:

$$
\begin{array}{ll}
 & \mathit{red1} \cdot (E \odot F) \\
= & \quad \{ \text{ specification of } \mathit{red1} \ \} \\
 & \mathit{red0} \cdot (E \odot F) \\
= & \quad \{ \text{ lemma 4.12, using } \mathit{hasnf} \cdot (E \odot F), \ \neg \mathit{abst} \cdot E \text{ and } \ \mathit{nf} \cdot E \ \} \\
 & E \odot (\mathit{red0} \cdot F) \\
= & \quad \{ \text{ specification of } \mathit{red1}, \text{ by induction hypothesis, see below } \}
\end{array}
$$

$$E \odot (\mathit{red1} \cdot F) \ .$$

The use of the induction hypothesis requires:

$$\mathit{vfr1} \cdot F \ \trianglelefteq \ \mathit{vfr1} \cdot (E \odot F) \quad \text{and:} \quad \mathit{hasnf} \cdot (E \odot F) \ \Rightarrow \ \mathit{hasnf} \cdot F \ .$$

The latter follows from the definition of *hasnf*, the former is satisfied by:

$$\mathit{vfr0} \cdot (E \odot F) \ = \ \mathit{vfr0} \cdot F \quad \text{and:} \quad \#F \ < \ \#(E \odot F) \ .$$

The former can be proved by induction on the reduction of F, the latter follows from the definition of # (3.6).

Case $E := (E \odot F)$, assuming $\mathit{hasnf} \cdot (E \odot F)$, $\neg \mathit{abst} \cdot E$ and $\neg \mathit{nf} \cdot E$:

$$\begin{array}{ll} & \mathit{red1} \cdot (E \odot F) \\ = & \quad \{ \text{ specification of } \mathit{red1} \ \} \\ & \mathit{red0} \cdot (E \odot F) \\ = & \quad \{ \text{ declaration of } \mathit{red0} \ (4.1), \text{ we have } \neg \mathit{nf} \cdot (E \odot F) \text{ by } (4.6), \text{ using } \\ & \quad \neg \mathit{abst} \cdot E \ \wedge \ \neg \mathit{nf} \cdot E \ \} \\ & \mathit{red0} \cdot (\mathit{rd1} \cdot (E \odot F)) \\ = & \quad \{ \text{ declaration of } \mathit{rd1} \ (4.10), \text{ using } \neg \mathit{abst} \cdot E \ \wedge \ \neg \mathit{nf} \cdot E \ \} \\ & \mathit{red0} \cdot ((\mathit{rd1} \cdot E) \odot F) \\ = & \quad \{ \text{ specification of } \mathit{red1}, \text{ by induction hypothesis, see below } \} \\ & \mathit{red1} \cdot ((\mathit{rd1} \cdot E) \odot F) \ . \end{array}$$

The use of the induction hypothesis requires:

$$\mathit{vfr1} \cdot ((\mathit{rd1} \cdot E) \odot F) \ \trianglelefteq \ \mathit{vfr1} \cdot (E \odot F) \ ,$$

and:

$$\mathit{hasnf} \cdot (E \odot F) \ \Rightarrow \ \mathit{hasnf} \cdot ((\mathit{rd1} \cdot E) \odot F) \ .$$

Using the definition of *rd1* (4.10), this is satisfied by the definitions of *vfr1*, $\trianglelefteq$ and *vfr0*, and lemma 4.2.

4.5 Summary

We have derived the following declaration for function *red1*:

$$red1 \cdot v = v \tag{4.13}$$
$$red1 \cdot (v \prec E) = v \prec (red1 \cdot E) \tag{4.14}$$
$$red1 \cdot ((v \prec E) \odot F) = red1 \cdot (E \leftarrow \langle v, F \rangle) \tag{4.15}$$
$$red1 \cdot (E \odot F) = E \odot (red1 \cdot F) \quad \text{, if } \neg abst \cdot E \wedge nf \cdot E \tag{4.16}$$
$$red1 \cdot (E \odot F) = red1 \cdot ((rd1 \cdot E) \odot F) \quad \text{, if } \neg abst \cdot E \wedge \neg nf \cdot E \tag{4.17}$$

and the following properties of function *vfr1*:

$$vfr1 \cdot E \trianglelefteq vfr1 \cdot (v \prec E) \tag{4.18}$$
$$vfr1 \cdot (E \leftarrow \langle v, F \rangle) \trianglelefteq vfr1 \cdot ((v \prec E) \odot F) \tag{4.19}$$
$$vfr1 \cdot F \trianglelefteq vfr1 \cdot (E \odot F) \quad \text{, if } \neg abst \cdot E \wedge nf \cdot E \tag{4.20}$$
$$vfr1 \cdot ((rd1 \cdot E) \odot F) \trianglelefteq vfr1 \cdot (E \odot F) \quad \text{, if } \neg abst \cdot E \wedge \neg nf \cdot E \tag{4.21}$$

and the following properties of function *hasnf*:

$$hasnf \cdot E \Leftarrow hasnf \cdot (v \prec E) \tag{4.22}$$
$$hasnf \cdot (E \leftarrow \langle v, F \rangle) \Leftarrow hasnf \cdot ((v \prec E) \odot F) \tag{4.23}$$
$$hasnf \cdot F \Leftarrow hasnf \cdot (E \odot F) \quad \text{, if } \neg abst \cdot E \wedge nf \cdot E \tag{4.24}$$
$$hasnf \cdot ((rd1 \cdot E) \odot F) \Leftarrow hasnf \cdot (E \odot F) \quad \text{, if } \neg abst \cdot E \wedge \neg nf \cdot E \tag{4.25}$$

4.6 Remarks

There is one case in the declaration of *red1*, namely (4.17), that still depends on function *rd1*. However, this case indicates that a generalization is possible.

Chapter 5

Application folding

We define operator $\circledcirc$ ("folded application") as the left-folded version of $\odot$, hence, operator $\circledcirc$ has type $\Lambda \times \mathcal{L}_*(\Lambda) \to \Lambda$. From here on, dummies as and bs have type $\mathcal{L}_*(\Lambda)$.

Consider the following two equalities (we omitted the preconditions for the sake of brevity):

$$\begin{aligned} \mathit{red1} \cdot (E \odot F) &= \mathit{red1} \cdot ((\mathit{rd1} \cdot E) \odot F) \,, \\ \mathit{red1} \cdot (E) &= \mathit{red1} \cdot (\mathit{rd1} \cdot E) \,. \end{aligned}$$

The first equality is the only case from the declaration of *red1* that uses function *rd1*. The second equality follows from the specification of *red1* and the declaration of *red0*. Both equalities are instances of the more general expression:

$$\mathit{red1} \cdot (E \circledcirc as) = \mathit{red1} \cdot ((\mathit{rd1} \cdot E) \circledcirc as) \,.$$

Hence, we can specify a new function (a generalization of *red1*, using $\circledcirc$) which admits a declaration that does not contain function *rd1* anymore. In this chapter we specify that function and derive a declaration for it. First, we consider some properties of folded application.

5.1 Properties of folded application

Lambda-term E is not in normal form if and only if it contains redexes. Lambda-term $E \circledcirc as$ has at least as many redexes as E, hence, if E is not in normal form then $E \circledcirc as$ is not in normal form either. I.e. we have

Lemma 5.1 For all E and as:

$$\neg \mathit{nf} \cdot E \Rightarrow \neg \mathit{nf} \cdot (E \circledcirc as) \,.$$

End of lemma 5.1.

For λ-terms E and F such that E is not an abstraction we have:

$$nf \cdot (E \odot F) \;=\; nf \cdot E \;\wedge\; nf \cdot F \;,$$

as part of the definition of nf (see (4.6)). Applying this (repeatedly) to e.g.: $nf \cdot (E \circledcirc [F, G])$, results in: $nf \cdot E \;\wedge nf \cdot F \wedge nf \cdot G$. This is an instance of

Lemma 5.2 For all as and E such that $\neg abst \cdot E$:

$$nf \cdot (E \circledcirc as) \;\equiv\; nf \cdot E \;\wedge\; \langle \forall i : 0 \leq i < \#as :\; nf \cdot (as \cdot i) \;\rangle \;.$$

End of lemma 5.2.

This lemma can be proved by induction on $\#as$.

* * *

In a similar fashion, for all E and as such that $\neg abst \cdot E$ and $\neg nf \cdot E$, the equality:

$$rd1 \cdot (E \odot F) \;=\; (rd1 \cdot E) \odot F \;,$$

is part of the definition of $rd1$ (see (4.10)). This can be used to prove the following lemma by induction on $\#as$.

Lemma 5.3 For all E and F such that $\neg abst \cdot E$ and $\neg nf \cdot E$:

$$rd1 \cdot (E \circledcirc as) \;=\; (rd1 \cdot E) \circledcirc as \;,$$

End of lemma 5.3.

5.2 Addition of lists

Binary operator $\boxplus$ ("plus") defines addition on lists of naturals, as follows:

Definition of operator $\boxplus$

For all positive natural n, operator $\boxplus$ has type $\mathcal{L}_n(\mathbb{N}) \times \mathcal{L}_n(\mathbb{N}) \;\rightarrow\; \mathcal{L}_n(\mathbb{N})$ and definition, for all k, l $(k, l \in \mathbb{N})$ and ks, ls $(ks, ls \in \mathcal{L}_n(\mathbb{N}))$:

$$\begin{aligned} [k] \;\boxplus\; [l] \;&=\; [k + l] \;, \\ (k \triangleright ks) \;\boxplus\; (l \triangleright ls) \;&=\; (k + l) \triangleright (ks \;\boxplus\; ls) \;. \end{aligned}$$

End of definition of operator $\boxplus$.

Operator $\boxplus$ is associative and symmetric, and it has the following property for all ks, ls and ms (all non-empty lists of naturals):

$$ks \;\trianglelefteq\; ls \;\;\equiv\;\; (ks \;\boxplus\; ms) \;\trianglelefteq\; (ls \;\boxplus\; ms) \;.$$

This follows from its definition.

Also, operator $\boxplus$ has as its identity element a list of type $\mathcal{L}_n(\,\{0\}\,)$.

5.3 Derivation

We use folded operator $\circledcirc$ to represent *one* λ-term by *one or more* λ-terms. This is a generalization because we replace a constant (*one* λ-term) by a variable (*one or more* λ-terms). A generalization step yields additional manipulative freedom, in this case the freedom to choose between, for example, the following expressions:

$$((E \odot F) \odot G) \circledcirc [\,]\,, \qquad (E \odot F) \circledcirc [G]\,, \qquad E \circledcirc [F, G]\,.$$

We exploit that freedom to derive a declaration for function *red1* that does not contain function *rd1* anymore. As before, we use the previous declaration of *red1* in the derivation of the new declaration. To distinguish the two, we introduce a new (partial) function, *red2* say, with the following:

Specification of function *red2*

Partial function *red2* has type $\mathcal{L}_*(\Lambda) \to \Lambda \to \Lambda$ and specification:

$$\langle\, \forall E, as : \mathit{hasnf} \cdot (E \circledcirc as) :\ \mathit{red2} \cdot as \cdot E \ =\ \mathit{red1} \cdot (E \circledcirc as)\ \rangle\,.$$

End of specification of function *red2*.

As function *red2* is a generalization of function *red1*, we can express *red1* in terms of *red2*, as follows:

$$\mathit{red1} \ =\ \mathit{red2} \cdot [\,]\,.$$

Thus, we verify that our original problem (an efficient implementation for *red1*) is an instance (or special case) of our new problem (an efficient implementation for *red2*).

As before, a variant function defines the form of induction that we use in the derivation. We use induction on the reduction (and size) of $E \circledcirc as$ and on the size of E, as follows:

Definition of function *vfr2*

Partial function *vfr2* has type $\mathcal{L}_*(\Lambda) \to \Lambda \to \mathcal{L}_3(\mathbb{N})$ and definition, for all E and as such that $\mathit{hasnf} \cdot (E \circledcirc as)$:

$$\mathit{vfr2} \cdot as \cdot E \ =\ \mathit{vfr1} \cdot (E \circledcirc as) \mathbin{\triangleleft} \#E\,.$$

End of definition of function *vfr2*.

We have the following property of lexicographic ordering, which we will apply tacitly.

For all as, bs, E and F, such that $hasnf \cdot (E \circledcirc as)$ and $hasnf \cdot (F \circledcirc bs)$, we can prove:

$$vfr2 \cdot as \cdot E \ \trianglelefteq \ vfr2 \cdot bs \cdot F \ ,$$

by proving either:

$$vfr1 \cdot (E \circledcirc as) \ \trianglelefteq \ vfr1 \cdot (F \circledcirc bs) \ ,$$

or:

$$(E \circledcirc as = F \circledcirc bs \ \wedge \ \#E < \#F) \ .$$

The following four cases comprise the derivation of a declaration for function *red2*.

Case $E := v$, assuming $hasnf \cdot (v \circledcirc as)$:

$$\begin{array}{ll} & red2 \cdot as \cdot v \\ = & \quad \{ \text{ specification of } red2 \ \} \\ & red1 \cdot (v \circledcirc as) \\ = & \quad \{ \ \bigstar \text{ lemma 5.4, see below, using } hasnf \cdot (v \circledcirc as) \ \} \\ & v \ \circledcirc \ (red1 \bullet as) \\ = & \quad \{ \text{ specification of } red2, \text{ by induction hypothesis, see below } \} \\ & v \ \circledcirc \ (red2 \cdot [\,] \bullet as) \ . \end{array}$$

We used the following:

Lemma 5.4 For all v and as such that $hasnf \cdot (v \circledcirc as)$:

$$red1 \cdot (v \circledcirc as) = v \circledcirc (red1 \bullet as) \ .$$

End of lemma 5.4.

This can be proved by induction on the reduction of $v \circledcirc as$, see appendix A.

The appeal to the induction hypothesis is valid if we can prove the conjunction of the following, for all v and as:

$$\begin{array}{l} \langle \boxplus i : 0 \leq i < \#as : \ vfr1 \cdot (as \cdot i) \ \rangle \ \trianglelefteq \ vfr1 \cdot (v \circledcirc as) \ , \\ \langle \forall i : 0 \leq i < \#as : \ hasnf \cdot (as \cdot i) \ \rangle \ \Leftarrow \ hasnf \cdot (v \circledcirc as) \ . \end{array}$$

The proofs of both conjuncts are similar to the proof of lemma 5.4.

Note that the correctness of the first conjunct depends on $\#v = 1$ (consider $as = [\,]$). Also note that the first conjunct is stronger than the condition that is required for a valid appeal to the induction hypothesis, after all, the following (weaker) condition is sufficient:

$$\langle \forall i : 0 \le i < \#as : \ \mathit{vfr1} \cdot (as \cdot i) \ \trianglelefteq \ \mathit{vfr1} \cdot (v \circledcirc as) \ \rangle \ .$$

However, the stronger condition will be more useful in what follows, specifically, in chapter 8.

Case $as, E := [\,], (v \prec E)$, assuming $\mathit{hasnf} \cdot (v \prec E)$:

$$\begin{array}{ll} & \mathit{red2} \cdot [\,] \cdot (v \prec E) \\ = & \quad \{ \text{ specification of } \mathit{red2} \ \} \\ & \mathit{red1} \cdot ((v \prec E) \ \circledcirc \ [\,]) \\ = & \quad \{ \text{ operator folding } \} \\ & \mathit{red1} \cdot (v \prec E) \\ = & \quad \{ \text{ declaration of } \mathit{red1} \ (4.14), \text{ using } \mathit{hasnf} \cdot (v \prec E) \ \} \\ & v \ \prec \ (\mathit{red1} \cdot E) \\ = & \quad \{ \text{ operator folding } \} \\ & v \ \prec \ (\mathit{red1} \cdot (E \circledcirc [\,])) \\ = & \quad \{ \text{ specification of } \mathit{red2}, \text{ by induction hypothesis, using:} \\ & \qquad \mathit{vfr1} \cdot E \ \trianglelefteq \ \mathit{vfr1} \cdot (v \prec E) \ , \text{ by } (4.18) \ , \\ & \quad \text{and:} \\ & \qquad \mathit{hasnf} \cdot E \ \Leftarrow \ \mathit{hasnf} \cdot (v \prec E) \ , \text{ by } (4.22) \ . \\ & \quad \} \\ & v \ \prec \ (\mathit{red2} \cdot [\,] \cdot E) \ . \end{array}$$

Case $as, E := (F \triangleright as), (v \prec E)$, assuming $\mathit{hasnf} \cdot ((v \prec E) \circledcirc (F \triangleright as))$:

$$
\begin{array}{ll}
 & \mathit{red2} \cdot (F \triangleright as) \cdot (v \prec E) \\
= & \quad \{ \text{ specification of } \mathit{red2} \ \} \\
 & \mathit{red1} \cdot (\ (v \prec E) \circledcirc (F \triangleright as)\ ) \\
= & \quad \{ \text{ operator folding } \} \\
 & \mathit{red1} \cdot (\ ((v \prec E) \odot F) \circledcirc as\ ) \\
= & \quad \{ \text{ we have:} \\
 & \qquad \neg \mathit{nf} \cdot (\ ((v \prec E) \odot F) \circledcirc as\ ) \ , \\
 & \quad \text{by (4.5) and lemma 5.1; using the specification of } \mathit{red1} \text{ and the} \\
 & \quad \text{declaration of } \mathit{red0} \ \} \\
 & \mathit{red1} \cdot (\ \mathit{rd1} \cdot (((v \prec E) \odot F) \circledcirc as)\ ) \\
= & \quad \{ \text{ lemma 5.3 } \} \\
 & \mathit{red1} \cdot (\ (\mathit{rd1} \cdot ((v \prec E) \odot F)) \ \circledcirc \ as\ ) \\
= & \quad \{ \text{ declaration of } \mathit{rd1} \ \ (4.8) \ \} \\
 & \mathit{red1} \cdot (\ (E \leftarrow \langle v, F \rangle) \ \circledcirc \ as\ ) \\
= & \quad \{ \text{ specification of } \mathit{red2}, \text{ by induction hypothesis, see below } \} \\
 & \mathit{red2} \cdot as \cdot (E \leftarrow \langle v, F \rangle) \ .
\end{array}
$$

We need to prove the conjunction of the following, for the appeal to the induction hypothesis, for all v , E , F and as :

$$
\begin{array}{l}
\mathit{vfr1} \cdot ((E \leftarrow \langle v, F \rangle) \circledcirc as) \ \trianglelefteq \ \mathit{vfr1} \cdot ((v \prec E) \circledcirc (F \triangleright as)) \ , \\
\mathit{hasnf} \cdot ((E \leftarrow \langle v, F \rangle) \circledcirc as) \ \Leftarrow \ \mathit{hasnf} \cdot ((v \prec E) \circledcirc (F \triangleright as)) \ ,
\end{array}
$$

both can be proved in a similar way as the proof of:

$$
\mathit{red1} \cdot ((E \leftarrow \langle v, F \rangle) \circledcirc as) \ = \ \mathit{red1} \cdot ((v \prec E) \circledcirc (F \triangleright as)),
$$

which is part of the above derivation.

Case $E := (E \odot F)$, assuming $\mathit{hasnf} \cdot ((E \odot F) \circledcirc as)$:

$$
\begin{array}{ll}
 & \mathit{red2} \cdot as \cdot (E \odot F) \\
= & \quad \{ \text{ specification of } \mathit{red2} \ \} \\
 & \mathit{red1} \cdot ((E \odot F) \circledcirc as) \\
= & \quad \{ \text{ operator folding } \} \\
 & \mathit{red1} \cdot (E \circledcirc (F \triangleright as)) \\
= & \quad \{ \text{ specification of } \mathit{red2}, \text{ by induction hypothesis, using:} \\
 & \qquad (E \odot F) \circledcirc as \ = \ E \circledcirc (F \triangleright as) \ \text{ and: } \ \#E < \#(E \odot F) \ . \\
 & \quad \} \\
 & \mathit{red2} \cdot (F \triangleright as) \cdot E \ .
\end{array}
$$

5.4 Summary

We derived the following declaration for function $\mathit{red2}$:

$$\mathit{red2} \cdot as \cdot v \ = \ v \circledcirc (\mathit{red2} \cdot [\,] \bullet as) \tag{5.5}$$
$$\mathit{red2} \cdot [\,] \cdot (v \prec E) \ = \ v \prec (\mathit{red2} \cdot [\,] \cdot E) \tag{5.6}$$
$$\mathit{red2} \cdot (F \triangleright as) \cdot (v \prec E) \ = \ \mathit{red2} \cdot as \cdot (E \leftarrow \langle v, F \rangle) \tag{5.7}$$
$$\mathit{red2} \cdot as \cdot (E \odot F) \ = \ \mathit{red2} \cdot (F \triangleright as) \cdot E \tag{5.8}$$

and the following properties of function $\mathit{vfr2}$:

$$\langle \boxplus i : 0 \le i < \#as : \ \mathit{vfr2} \cdot [\,] \cdot (as \cdot i) \ \rangle \trianglelefteq \mathit{vfr2} \cdot as \cdot v \tag{5.9}$$
$$\mathit{vfr2} \cdot [\,] \cdot E \trianglelefteq \mathit{vfr2} \cdot [\,] \cdot (v \prec E) \tag{5.10}$$
$$\mathit{vfr2} \cdot as \cdot (E \leftarrow \langle v, F \rangle) \trianglelefteq \mathit{vfr2} \cdot (F \triangleright as) \cdot (v \prec E) \tag{5.11}$$
$$\mathit{vfr2} \cdot (F \triangleright as) \cdot E \trianglelefteq \mathit{vfr2} \cdot as \cdot (E \odot F) \tag{5.12}$$

and the following properties of function hasnf:

$$\langle \forall i : 0 \le i < \#as : \ \mathit{hasnf} \cdot (as \cdot i) \ \rangle \Leftarrow \mathit{hasnf} \cdot (v \circledcirc as) \tag{5.13}$$
$$\mathit{hasnf} \cdot E \Leftarrow \mathit{hasnf} \cdot (v \prec E) \tag{5.14}$$
$$\mathit{hasnf} \cdot ((E \leftarrow \langle v, F \rangle) \circledcirc as) \Leftarrow \mathit{hasnf} \cdot ((v \prec E) \circledcirc (F \triangleright as)) \tag{5.15}$$
$$\mathit{hasnf} \cdot (E \circledcirc (F \triangleright as)) \Leftarrow \mathit{hasnf} \cdot ((E \odot F) \circledcirc as) \tag{5.16}$$

5.5 Similarities

The declaration of *red2* has a similar shape as the properties of *vfr2* and *hasnf*. Even the cases (5.9) and (5.13) can be transformed into expressions with similar shapes as that of (5.5). This is not surprising, because functions *red0* and *vfr0* and predicate *hasnf* also have a similar shape, namely (for all E such that $hasnf \cdot E$):

$$nf \cdot E \;\Rightarrow\; \begin{cases} red0 \cdot E &= E \\ hasnf \cdot E &\equiv \text{TRUE} \\ vfr0 \cdot E &= 0 \end{cases}$$

$$\neg nf \cdot E \;\Rightarrow\; \begin{cases} red0 \cdot E &= red0 \cdot (rd1 \cdot E) \\ hasnf \cdot E &\equiv hasnf \cdot (rd1 \cdot E) \\ vfr0 \cdot E &= 1 + vfr0 \cdot (rd1 \cdot E) \end{cases}$$

As of yet, I have not been able to determine how (even *if*) this can be exploited. However, I have found that this pattern persists in the coming transformations. To avoid repetition and improve readability we define the following:

Has-normal-form rule

From this point on, all relevant λ-terms E satisfy $hasnf \cdot E$.

End of has-normal-form rule.

5.6 Remarks

Consider the following application:

$$red2 \cdot as \cdot E \ .$$

The list $(E \triangleright as)$ is known in the literature (on implementing functional languages), amongst others, as the "evaluation stack", "spine stack" or just "stack" (see e.g. [Sestoft, 1997; Jones and Lester, 1992; Krivine, 2007]).

In our derivation, the use of list as as a stack "emerged" from the manipulation of the symbols involved. Knowledge of a stack (or its properties) was not needed (beforehand or otherwise) to design or derive the current declaration. The main design decision was a "simple" generalization.

In the design of the specification for function *red2*, we have chosen to make λ-term E ("the top of the stack") a separate parameter, mainly to allow for a more clear and concise presentation.

Chapter 6

Substitution folding

Evaluation of an expression like $E \leftarrow \langle v, F \rangle$, boils down to identifying in λ-term E every free occurrence of variable v and replacing it with λ-term F. That is a time-consuming operation which (in some cases) is not even necessary, just consider a λ-term E that has no free occurrences of v. We will use a folded version of operator $\leftarrow$ in an attempt to *postpone* substitutions as much as possible and to *avoid* needless substitutions altogether.

Operator $\Subset$ ("folded substitution") is the left-folded version of $\leftarrow$, hence, operator $\Subset$ has type $\Lambda \times \mathcal{L}_*(\langle \mathcal{V}, \Lambda \rangle) \to \Lambda$. From here on, dummies ss and ts have type $\mathcal{L}_*(\langle \mathcal{V}, \Lambda \rangle)$.

6.1 On lists of pairs

For some types X and Y, we consider lists of pairs X, Y, i.e. lists of type $\mathcal{L}_*(\langle X, Y \rangle)$. In this section, dummy xys has type $\mathcal{L}_*(\langle X, Y \rangle)$, dummy x has type X and dummy y has type Y. We introduce a few functions, defined on lists of pairs, that are useful in what follows.

Definition of function *dom*

> Function *dom* ("domain") has type $\mathcal{L}_*(\langle X, Y \rangle) \to \mathrm{Set}(X)$ and definition, for all xys:
>
> $$dom \cdot xys \;=\; [\![(\cdot 0) \bullet xys]\!] \,.$$

End of definition of function *dom*.

Definition of function *ran*

Function *ran* ("range") has type $\mathcal{L}_*(\langle X, Y \rangle) \to \mathrm{Set}(Y)$ and definition, for all xys:

$$ran \cdot xys \;=\; [\![(\cdot 1) \bullet xys]\!] \;.$$

End of definition of function *ran*.

Definition of operator $\Uparrow$

Binary operator $\Uparrow$ ("lookup") has type $X \times \mathcal{L}_*(\langle X, Y \rangle) \to Y$. For all x, xys, application $x \Uparrow xys$ has precondition:

$$x \in dom \cdot xys \;\;\wedge\;\; \#xys \;=\; \#(\, dom \cdot xys) \;.$$

The second conjunct ensures that an x occurs at most once as a left element of a pair in xys. Application $x \Uparrow xys$ results in the right element of the pair in xys with x as its left element, i.e., operator $\Uparrow$ has specification (for all x, y and xys):

$$\langle x, y \rangle \in [\![xys]\!] \;\;\Rightarrow\;\; x \Uparrow xys \;=\; y \;.$$

End of definition of operator $\Uparrow$.

* * *

We define function *fv* on "postponed substitution" lists as follows:

Definition of function *fv*

Function *fv* of type $\mathcal{L}_*(\langle \mathcal{V}, \Lambda \rangle) \to \mathrm{Set}(\mathcal{V})$ has definition (for all ss):

$$fv \cdot ss \;=\; \langle \bigcup F : F \in ran \cdot ss : \; fv \cdot F \;\rangle \;.$$

End of definition of function *fv*.

6.2 Properties of (folded) substitution

Lambda-term E with F substituted for every free occurrence of v in E, will yield E if v is not a free variable of E. I.e.:

Lemma 6.1 For all v, E and F:

$$v \notin \mathit{fv} \cdot E \quad \Rightarrow \quad E \leftarrow \langle v, F \rangle \;=\; E \;.$$

End of lemma 6.1.

This property is "inherited" by operator $\ominus$ as follows:

Lemma 6.2 For all E and ss:

$$\mathit{fv} \cdot E \cap \mathit{dom} \cdot \mathit{ss} \;=\; \emptyset \quad \Rightarrow \quad E \ominus \mathit{ss} \;=\; E \;.$$

End of lemma 6.2.

Lemma 6.1 can be proved using structural induction on E. Lemma 6.2 can be proved by induction on $\#\mathit{ss}$.

* * *

Operator $\ominus$ also "inherits" other properties from the definition of $\leftarrow$, see the following two lemmas (both can be proved by induction on $\#\mathit{ss}$):

Lemma 6.3 For all v, E and ss:

$$\begin{array}{l} \quad v \notin (\mathit{dom} \cdot \mathit{ss} \;\cup\; \mathit{fv} \cdot \mathit{ss}) \\ \Rightarrow \\ \quad (v \prec E) \ominus \mathit{ss} \;=\; v \prec (E \ominus \mathit{ss}) \;. \end{array}$$

End of lemma 6.3.

Lemma 6.4 For all E, F and ss:

$$(E \odot F) \ominus \mathit{ss} \;=\; (E \ominus \mathit{ss}) \odot (F \ominus \mathit{ss}) \;.$$

End of lemma 6.4.

Note that the assumption in lemma 6.3 can always be satisfied by renaming bound variable v in λ-term $v \prec E$.

6.3 Properties of *red2*

In order to find useful properties of *red2* with respect to operator $\ominus$, we investigate expressions of the shape:

$$\mathit{red2} \cdot \mathit{as} \cdot (E \ominus \mathit{ss}) \;.$$

This will provide a rationale for the next transformation. Note that the above expression is a generalization of $red2 \cdot as \cdot E$, because $E = E \ominus [\,]$.

We present the properties (all but one) without proof, as most of them follow directly from the declaration of *red2* and properties of $\ominus$. We define the following predicates (predicate P will be defined later):

$$
\begin{aligned}
&C_0 \cdot v \cdot ss \equiv v \notin dom \cdot ss \,, \\
&C_1 \cdot v \cdot ss \equiv v \in dom \cdot ss \wedge P \cdot ss \,, \\
&C_2 \cdot v \cdot ss \equiv v \notin (dom \cdot ss \cup fv \cdot ss) \,,
\end{aligned}
$$

and for all relevant v, E, F, as and ss we present the following properties of *red2*:

$$
\begin{aligned}
red2 \cdot as \cdot (v \ominus ss) &= v \circledcirc (red2 \cdot [\,] \bullet as) && \text{, if } C_0 \cdot v \cdot ss && (6.5)\\
red2 \cdot as \cdot (v \ominus ss) &= red2 \cdot as \cdot (v \Uparrow ss) && \text{, if } C_1 \cdot v \cdot ss && (6.6)\\
red2 \cdot [\,] \cdot ((v \prec E) \ominus ss) &= v \prec (red2 \cdot [\,] \cdot (E \ominus ss)) && \text{, if } C_2 \cdot v \cdot ss && (6.7)\\
red2 \cdot (F \triangleright as) \cdot ((v \prec E) \ominus ss) &= red2 \cdot as \cdot (E \ominus (ss \triangleleft \langle v, F \rangle)) && \text{, if } C_2 \cdot v \cdot ss && (6.8)\\
red2 \cdot as \cdot ((E \odot F) \ominus ss) &= red2 \cdot ((F \ominus ss) \triangleright as) \cdot (E \ominus ss) && && (6.9)
\end{aligned}
$$

Note on (6.8) The proof of this case is noteworthy, because it shows why the pair $\langle v, F \rangle$ is added to the back of list ss (when reading (6.8) from left to right) instead of the front:

$$
\begin{aligned}
& red2 \cdot (F \triangleright as) \cdot ((v \prec E) \ominus ss) \\
= \quad & \{ \text{ lemma 6.3, using assumption } C_2 \cdot v \cdot ss \ \} \\
& red2 \cdot (F \triangleright as) \cdot (v \prec (E \ominus ss)) \\
= \quad & \{ \text{ declaration of } red2, \text{ with } E := E \ominus ss \ \} \\
& red2 \cdot as \cdot ((E \ominus ss) \leftarrow \langle v, F \rangle) \\
= \quad & \{ \text{ property of operator folding (2.1) } \} \\
& red2 \cdot as \cdot (E \ominus (ss \triangleleft \langle v, F \rangle)) \,,
\end{aligned}
$$

hence, pair $\langle v, F \rangle$ is added to the back of list ss because that is convenient in this proof.

End of note on (6.8).

6.4 Lookup

We investigate for what P (if any) the following lemma holds, for all ss:

Lemma 6.10 For all v and ss:

$$v \in dom \cdot ss \;\wedge\; P \cdot ss \quad \Rightarrow \quad v \ominus ss \;=\; v \Uparrow ss \;.$$

End of lemma 6.10.

We try to prove this using induction on $\#ss$. If $ss = [\,]$ then the antecedent is FALSE and the lemma holds for any P. For the nonempty case we prove:

$$\begin{array}{ll} & v \ominus (ss \triangleleft \langle v, G \rangle) \\ = & \quad \{ \text{ property of operator folding } \} \\ & (v \ominus ss) \leftarrow \langle v, G \rangle \\ = & \quad \{ \star \text{ assume } v \notin dom \cdot ss \text{, use lemma 6.2 } \} \\ & v \leftarrow \langle v, G \rangle \\ = & \quad \{ \text{ definition of } \leftarrow \} \\ & G \\ = & \quad \{ \text{ definition of } \Uparrow \text{, using } v \notin dom \cdot ss \} \\ & v \Uparrow (ss \triangleleft \langle v, G \rangle) \;. \end{array}$$

We consider the other case, $v \neq u$:

$$\begin{array}{ll} & v \ominus (ss \triangleleft \langle u, G \rangle) \\ = & \quad \{ \text{ property of operator folding } \} \\ & (v \ominus ss) \leftarrow \langle u, G \rangle \\ = & \quad \{ \star \text{ assume } u \notin fv \cdot ss \text{, use lemma 6.1 } \} \\ & v \ominus ss \\ = & \quad \{ \text{ induction hypothesis, } \star \text{ assume } P \cdot ss \} \\ & v \Uparrow ss \\ = & \quad \{ \text{ definition of } \Uparrow \text{, using } v \neq u \} \\ & v \Uparrow (ss \triangleleft \langle u, G \rangle) \;. \end{array}$$

We conclude that lemma 6.10 is correct if predicate P satisfies, for all u, G and ss:

$$P \cdot (ss \triangleleft \langle u, G \rangle) \;\Rightarrow\; P \cdot ss \;\wedge\; u \notin (dom \cdot ss \cup fv \cdot ss) \;.$$

The weakest such P is defined as (for all u, G and ss):

$$\begin{array}{rcl} P \cdot [\,] & \equiv & \text{TRUE} \;, \\ P \cdot (ss \triangleleft \langle u, G \rangle) & \equiv & P \cdot ss \;\wedge\; u \notin (dom \cdot ss \cup fv \cdot ss) \;. \end{array}$$

6.5 Remarks

Consider the properties of *red2* we found (6.5 - 6.9). From (6.9) we conclude that every element of list as is of the shape $G \ominus ts$. Knowing that, we conclude from (6.8) that every right element from the pairs in list ss is also of the shape $G \ominus ts$. In all, to postpone substitutions as much as possible, we conclude that *every* (relevant) λ-term needs to be represented by a λ-term and a "postponed substitutions" list (using folded operator $\ominus$), including the terms in those lists. Hence, some sort of recursive data structure is needed. We introduce such a structure in the next chapter.

Regarding lemma 6.3, we stated that assumption:

$$v \notin (dom \cdot ss \cup fv \cdot ss) ,$$

can always be satisfied by renaming bound variable v in λ-term $v \prec E$. Bound-variable renaming is a well-known solution to avoid name-clashes [Barendregt, 1984], and so is representing variables by numbers, e.g. using "de Bruijn indices". As of yet, I have not been able to derive a declaration for reduction that implements a solution to name-clashes in a satisfactory manner.

Regarding lemma 6.10, we used operator $\triangleleft$ in the proof of the nonempty case. A similar proof, using operator $\triangleright$, yields a similar (weakest) definition for P. In what follows, the current definition for P is preferable.

Consider the declaration of *red2*. List ss is used as a representation of a mapping from $\mathcal{V}$ to Λ. In (6.8) the mapping is extended (when reading from left to right) and in (6.6) it is used to map a variable to a λ-term. In the other cases list ss is not changed. Hence, the order of the elements in ss is irrelevant to the declaration for *red2*, we may just as well use a set instead of a list.

We have not investigated the effects of such a change in datatype to the relevant proofs. In what follows, we still use a list of pairs instead of a set of pairs to represent a mapping, because we expect the relevant proofs to be clearer that way.

Chapter 7

Closures and environments

In our previous attempt to postpone substitutions, we found that some sort of recursive data structure is needed to represent every (relevant) λ-term. In the functional programming community such a structure is known as a *closure*. A closure consists of a λ-term and an *environment* which is a list of pairs, each pair containing a variable and a closure. I.e., the mutually dependent types $\mathcal{C}$ ("closures") and $\mathcal{E}$ ("environments") are defined as follows:

$$\mathcal{C} = \langle \Lambda, \mathcal{E} \rangle \ ,$$
$$\mathcal{E} = \mathcal{L}_*(\langle \mathcal{V}, \mathcal{C} \rangle) \ .$$

From here on, dummy c has type $\mathcal{C}$, dummies as and bs have type $\mathcal{L}_*(\mathcal{C})$ and dummies ss and ts have type $\mathcal{E}$.

7.1 Preliminaries

Definition of function *vfe*

Variant function vfe has type $\mathcal{E} \to \mathbb{N}$ and definition, for all ss (operator $\uparrow$ denotes the maximum on naturals):

$$\begin{array}{lllll} vfe \cdot ss = & \textbf{if} & ss = [\,] & \to & 0 \\ & [\!] & ss \neq [\,] & \to & 1 + \langle \uparrow F, ts : \langle F, ts \rangle \in ran \cdot ss : \ vfe \cdot ts \ \rangle \\ & \textbf{fi} & & & \end{array}$$

End of definition of function *vfe*.

Definition of operator $\diamond$

For some types X and Y, operator $\diamond$ ("applied to 2nd element of") has type:

$$(Y \to Y) \times \langle X, Y \rangle \ \to \ \langle X, Y \rangle \ ,$$

and definition (for all f, $f \in Y \to Y$, x, $x \in X$ and y, $y \in Y$):

$$f \diamond \langle x, y \rangle \ = \ \langle x, f \cdot y \rangle \ .$$

End of definition of operator $\diamond$.

Definition of function ct

Function ct ("closure to term") maps a closure to the λ-term it represents. Function ct has type $\mathcal{C} \to \Lambda$ and definition:

$$ct \cdot \langle E, ss \rangle \ = \ E \ominus ((ct \diamond) \bullet ss) \ .$$

End of definition of function ct.

Definition of function fv

We define function fv ("free variables") on environments and on closures. Functions fv of type $\mathcal{C} \to \mathrm{Set}(\mathcal{V})$ and fv of type $\mathcal{E} \to \mathrm{Set}(\mathcal{V})$ have the following definition:

$$\begin{aligned} fv \cdot c \ &= \ fv \cdot (\, ct \cdot c) \ , \\ fv \cdot ss \ &= \ \langle \bigcup d : d \in ran \cdot ss : \ fv \cdot d \ \rangle \ . \end{aligned}$$

End of definition of function fv.

Due to this definition (or overloading) of function fv on closures and environments, we now have:

Lemma 7.1 For all ss:

$$fv \cdot ((ct \diamond) \bullet ss) \ = \ fv \cdot ss \ .$$

End of lemma 7.1.

* * *

Recall the definition of predicate P (for all u, G and rs, $rs \in \mathcal{L}_*(\langle \mathcal{V}, \Lambda \rangle)$):

$$\begin{array}{rcl} P \cdot [\,] & \equiv & \text{TRUE}\,, \\ P \cdot (rs \triangleleft \langle u, G \rangle) & \equiv & P \cdot rs \ \wedge \ u \notin (\mathit{dom} \cdot rs \ \cup \ \mathit{fv} \cdot rs)\,. \end{array}$$

By this definition and lemma 7.1 we have (for all ss):

$$P \cdot ((ct \diamond) \bullet ss) \quad = \quad P \cdot ss\,.$$

We use P in the following:

Lemma 7.2 For all v and ss satisfying $v \in \mathit{dom} \cdot ss$ and $P \cdot ss$:

$$ct \cdot \langle v, ss \rangle \ = \ ct \cdot (v \Uparrow ss)\,.$$

End of lemma 7.2.

Proof:

$$\begin{array}{ll} & ct \cdot \langle v, ss \rangle \\ = & \quad \{ \text{ definition of } ct \ \} \\ & v \ominus ((ct \diamond) \bullet ss) \\ = & \quad \{ \text{ lemma 6.10, using } v \in \mathit{dom} \cdot ss \text{ and } P \cdot ((ct \diamond) \bullet ss) \ \} \\ & v \Uparrow ((ct \diamond) \bullet ss) \\ = & \quad \{ \text{ property of } \Uparrow, \text{ see below } \} \\ & ct \cdot (v \Uparrow ss) \end{array}$$

□

We used the following property of $\Uparrow$, for all v and ss:

$$v \in \mathit{dom} \cdot ss \ \Rightarrow \ v \Uparrow ((ct \diamond) \bullet ss) \ = \ ct \cdot (v \Uparrow ss)\,.$$

This can be proved by induction on $\# ss$.

7.2 Derivation

We introduce a new (partial) function called *red3* with the following:

Specification of function *red3*

Partial function *red3* has type:

$$\mathcal{L}_*(\mathcal{C}) \to \mathcal{C} \to \Lambda \ ,$$

and for all relevant[1] as and c, function *red3* has specification:

$$red3 \cdot as \cdot c \ = \ red2 \cdot (\, ct \bullet as) \cdot (\, ct \cdot c) \ .$$

End of specification of function *red3*.

Function *red3* is a generalization of function *red2* (for $as \in \mathcal{L}_*(\Lambda)$ and bs such that $\langle \forall i : 0 \leq i < \#as : \ bs \cdot i \ = \ \langle as \cdot i, [\,] \rangle \ \rangle$):

$$red2 \cdot as \cdot E \ = \ red3 \cdot bs \cdot \langle E, [\,] \rangle \ .$$

We also have:

$$red3 \cdot [\,] \ = \ (\, red2 \cdot [\,] \circ ct) \ .$$

This follows from the specification of *red3* .

As usual, a variant function defines the form of induction that we use in the derivation. Variant (partial) function *vfr3* has the following:

Definition of function *vfr3*

Partial function *vfr3* has type:

$$\mathcal{L}_*(\mathcal{C}) \to \mathcal{C} \to \mathcal{L}_4(\mathbb{N}) \ ,$$

and for all relevant as, E and ss, function *vfr3* has definition:

$$vfr3 \cdot as \cdot \langle E, ss \rangle \ = \ vfr2 \cdot (\, ct \bullet as) \cdot (\, ct \cdot \langle E, ss \rangle) \ \triangleleft \ vfe \cdot ss \ .$$

End of definition of function *vfr3*.

Using the properties of *red2* (6.5 - 6.9) and of *vfr2* (5.9 - 5.12), the derivation of a declaration for function *red3* is straightforward. We will present the declaration without proof, with the exception of the following case:

[1] We ignore the precondition regarding predicate *hasnf* , due to the has-normal-form rule.

Assuming $v \in dom \cdot ss$, we derive:

$$
\begin{array}{ll}
 & red3 \cdot as \cdot \langle v, ss \rangle \\
= & \quad \{ \text{ specification of } red3 \ \} \\
 & red2 \cdot (ct \bullet as) \cdot (ct \cdot \langle v, ss \rangle) \\
= & \quad \{ \text{ lemma 7.2, using } v \in dom \cdot ss \text{ and } \star \text{ assuming } P \cdot ss \ \} \\
 & red2 \cdot (ct \bullet as) \cdot (ct \cdot (v \Uparrow ss)) \\
= & \quad \{ \text{ specification of } red3, \text{ by induction hypothesis, see below } \} \\
 & red3 \cdot as \cdot (v \Uparrow ss) \ .
\end{array}
$$

For the appeal to the induction hypothesis, we need to prove the following:

$$
\begin{array}{ll}
 & vfr3 \cdot as \cdot (v \Uparrow ss) \ \trianglelefteq \ vfr3 \cdot as \cdot \langle v, ss \rangle \\
\Leftarrow & \quad \{ \text{ property of lexicographic ordering } \} \\
 & vfr2 \cdot (ct \bullet as) \cdot (ct \cdot (v \Uparrow ss)) \ \trianglelefteq \ vfr2 \cdot (ct \bullet as) \cdot (ct \cdot \langle v, ss \rangle) \\
 & \vee \\
 & (as = as \ \wedge \ ct \cdot (v \Uparrow ss) = ct \cdot \langle v, ss \rangle \ \wedge \ vfe \cdot ((v \Uparrow ss) \cdot 1) < vfe \cdot ss) \\
\equiv & \quad \{ \text{ equals for equals using lemma 7.2; this falsifies the first disjunct and simplifies the second disjunct } \} \\
 & vfe \cdot ((v \Uparrow ss) \cdot 1) \ < \ vfe \cdot ss \ .
\end{array}
$$

We prove this inequality as follows:

$$
\begin{array}{ll}
 & vfe \cdot ss \\
= & \quad \{ \text{ definition of } vfe, \text{ using } ss \neq [\,] \text{ because } v \in dom \cdot ss \ \} \\
 & 1 + \langle \uparrow F, ts : \langle F, ts \rangle \in ran \cdot ss : \ vfe \cdot ts \ \rangle \\
= & \quad \{ \text{ split off } ts \ = \ ((v \Uparrow ss) \cdot 1), \text{ using} \\
 & \quad \ v \in dom \cdot ss \ \ \Rightarrow \ \ (v \Uparrow ss) \in ran \cdot ss \ \} \\
 & 1 + (\ \langle \uparrow F, ts : \langle F, ts \rangle \in ran \cdot ss : \ vfe \cdot ts \ \rangle \uparrow \ vfe \cdot ((v \Uparrow ss) \cdot 1) \ ) \\
\geq & \quad \{ \text{ property of } \uparrow \ \} \\
 & 1 + vfe \cdot ((v \Uparrow ss) \cdot 1) \\
> & \quad \{ \text{ using } 0 < 1 \ \} \\
 & vfe \cdot ((v \Uparrow ss) \cdot 1) \\
\boxtimes &
\end{array}
$$

Declaration for function *red3* For all relevant v, E, F, c, as and ss:

$$
\begin{aligned}
C_0 \cdot v \cdot ss &\equiv v \notin dom \cdot ss \,, \\
C_1 \cdot v \cdot ss &\equiv v \in dom \cdot ss \;\wedge\; P \cdot ss \,, \\
C_2 \cdot v \cdot ss &\equiv v \notin (\, dom \cdot ss \cup fv \cdot ss) \,,
\end{aligned}
$$

$$
\begin{aligned}
red3 \cdot as \cdot \langle v, ss \rangle &= v \circledcirc (red3 \cdot [\,] \bullet as) && \text{, if } C_0 \cdot v \cdot ss && (7.3)\\
red3 \cdot as \cdot \langle v, ss \rangle &= red3 \cdot as \cdot (v \Uparrow ss) && \text{, if } C_1 \cdot v \cdot ss && (7.4)\\
red3 \cdot [\,] \cdot \langle (v \prec E), ss \rangle &= v \prec (red3 \cdot [\,] \cdot \langle E, ss \rangle) && \text{, if } C_2 \cdot v \cdot ss && (7.5)\\
red3 \cdot (c \triangleright as) \cdot \langle (v \prec E), ss \rangle &= red3 \cdot as \cdot \langle E, (ss \triangleleft \langle v, c \rangle) \rangle && \text{, if } C_2 \cdot v \cdot ss && (7.6)\\
red3 \cdot as \cdot \langle (E \odot F), ss \rangle &= red3 \cdot (\langle F, ss \rangle \triangleright as) \cdot \langle E, ss \rangle && && (7.7)
\end{aligned}
$$

End of declaration for function *red3*.

Properties of *vfr3* For all relevant v, E, F, c, as and ss:

$$
\begin{aligned}
\langle \boxplus i : 0 \le i < \#as : \; vfr3 \cdot [\,] \cdot (as \cdot i) \,\rangle &\trianglelefteq vfr3 \cdot as \cdot \langle v, ss \rangle && \text{,if } C_0 \cdot v \cdot ss && (7.8)\\
vfr3 \cdot as \cdot (v \Uparrow ss) &\trianglelefteq vfr3 \cdot as \cdot \langle v, ss \rangle && \text{,if } C_1 \cdot v \cdot ss && (7.9)\\
vfr3 \cdot [\,] \cdot \langle E, ss \rangle &\trianglelefteq vfr3 \cdot [\,] \cdot \langle (v \prec E), ss \rangle && \text{,if } C_2 \cdot v \cdot ss && (7.10)\\
vfr3 \cdot as \cdot \langle E, (ss \triangleleft \langle v, c \rangle) \rangle &\trianglelefteq vfr3 \cdot (c \triangleright as) \cdot \langle (v \prec E), ss \rangle && \text{,if } C_2 \cdot v \cdot ss && (7.11)\\
vfr3 \cdot (\langle F, ss \rangle \triangleright as) \cdot \langle E, ss \rangle &\trianglelefteq vfr3 \cdot as \cdot \langle (E \odot F), ss \rangle && && (7.12)
\end{aligned}
$$

End of properties of *vfr3*.

7.3 Preconditions

We examine whether we can satisfy the second conjunct in condition C_1. We provide application $red3 \cdot as \cdot \langle E, ss \rangle$ with precondition $Q \cdot ss$, where predicate Q is defined as (for all u, F, ts and ss):

$$
\begin{aligned}
Q \cdot [\,] &\equiv \text{TRUE} \,, \\
Q \cdot (ss \triangleleft \langle u, \langle F, ts \rangle \rangle) &\equiv Q \cdot ss \;\wedge\; Q \cdot ts \;\wedge\; u \notin (\, dom \cdot ss \;\cup\; fv \cdot ss) \,.
\end{aligned}
$$

Consequently, we have to prove that $Q \cdot ss$ is satisfied by every application of *red3* that we used. As function *red3* is an implementation (and a generalization) of function *red0*:

$$
red0 \cdot E = red3 \cdot [\,] \cdot \langle E, [\,] \rangle \,,
$$

our first concern is the validity of $Q \cdot [\,]$, which is valid by definition of Q. Considering the declaration for function *red3*, we also need to prove the following:

$$\text{for (7.3):} \qquad Q \cdot ss \;\Rightarrow\; \langle \forall F, ts : \langle F, ts \rangle \in [\![as]\!] :\; Q \cdot ts \;\rangle \tag{7.13}$$

$$\text{for (7.4):} \qquad Q \cdot ss \;\Rightarrow\; Q \cdot ((v \Uparrow ss) \cdot 1) \tag{7.14}$$

$$\text{for (7.5):} \qquad Q \cdot ss \;\Rightarrow\; Q \cdot ss \tag{7.15}$$

$$\text{for (7.6):} \qquad Q \cdot ss \;\Rightarrow\; Q \cdot (ss \triangleleft \langle v, c \rangle) \tag{7.16}$$

$$\text{for (7.7):} \qquad Q \cdot ss \;\Rightarrow\; Q \cdot ss \tag{7.17}$$

We can only prove (7.13) by introducing its consequent as another precondition to application $red3 \cdot as \cdot \langle E, ss \rangle$. We define predicate R as follows (for all as):

$$R \cdot as \;\equiv\; \langle \forall F, ts : \langle F, ts \rangle \in [\![as]\!] :\; Q \cdot ts \;\rangle \,.$$

Application $red3 \cdot as \cdot \langle E, ss \rangle$ now has precondition $Q \cdot ss \wedge R \cdot as$. Luckily, the additional proof obligation generated by the introduction of precondition $R \cdot as$ is small, as can be seen from the following:

Little proof

> We prove that $R \cdot as$ holds for every application of *red3* that we used. By empty range of $R \cdot [\,]$, it is satisfied for application:
>
> $$red3 \cdot [\,] \cdot \langle E, [\,] \rangle \,.$$
>
> Considering the declaration for function *red3* again, we note that list as is only extended in (7.7) (when reading from left to right) and the validity of:
>
> $$R \cdot (\langle F, ss \rangle \triangleright as) \,,$$
>
> follows from precondition $Q \cdot ss \wedge R \cdot as$.

End of little proof.

We continue with the proofs regarding precondition $Q \cdot ss$. The consequent of (7.14) follows from the antecedent by weakening. Cases (7.15) and (7.17) are satisfied by the reflexivity of $\Rightarrow$. That leaves (7.16), the consequent follows from precondition $Q \cdot ss \wedge R \cdot (c \triangleright as)$ and condition $C_2 \cdot v \cdot ss$.

7.4 Remarks

The declaration for function *red3* implements a form of reduction (also called evaluation) which is known in the functional programming community as *demand-driven* or *delayed* evaluation.

Chapter 8

Tail recursive reduction

In this chapter we derive a tail recursive declaration for function *red3* . Such a declaration can be transformed to a sequential program using "standard" techniques. It is also a step towards a declaration that resembles an abstract machine, because the parameters of the tail recursive declaration represent the "state" of the machine. This derivation is similar to an example from [Hoogerwoord, 2007, section "Multiple Recursion: an example"].

8.1 Linearly recursive reduction

The first step towards a tail recursive declaration is a linearly recursive declaration. The only case from the declaration of *red3* that is (possibly) not linearly recursive is (we omitted the preconditions for the sake of brevity):

$$red3 \cdot as \cdot \langle v, ss \rangle \;=\; v \;\circledcirc\; (red3 \cdot [\,] \bullet as) \;.$$

The mapping of $red3 \cdot [\,]$ over list as can yield multiple recursive applications. In general, mapping a function over a list is a generalization of applying the (same) function to a single element. We use that fact to specify a new (partial) function. First, we recall the definitions of predicates Q and R. For all u, F, ts and ss:

$$\begin{array}{rcl} Q \cdot [\,] & \equiv & \text{TRUE} \;, \\ Q \cdot (ss \triangleleft \langle u, \langle F, ts \rangle \rangle) & \equiv & Q \cdot ss \;\wedge\; Q \cdot ts \;\wedge\; u \notin (dom \cdot ss \;\cup\; fv \cdot ss) \;. \end{array}$$

And, for all as:

$$R \cdot as \;\equiv\; \langle \forall F, ts : \langle F, ts \rangle \in [\![as]\!] : \; Q \cdot ts \;\rangle \;.$$

Now (partial) function *red4* has the following:

Specification of function *red4*

For all natural n, partial function *red4* has type:

$$\mathcal{L}_*(\mathcal{C}) \to \mathcal{L}_n(\mathcal{C}) \to \mathcal{L}_n(\Lambda) \ .$$

Application $red4 \cdot as \cdot cs$ $(cs \in \mathcal{L}_n(\mathcal{C}))$ has precondition:

$$R \cdot as \ \wedge \ R \cdot cs \ \wedge \ (cs = [\,] \ \Rightarrow \ as = [\,]) \ .$$

For all relevant[1] as, c and cs, function *red4* has specification:

$$\begin{aligned} red4 \cdot [\,] \cdot [\,] \ &= \ [\,] \ , \\ red4 \cdot as \cdot (c \triangleright cs) \ &= \ (red3 \cdot as \cdot c) \ \triangleright \ (red3 \cdot [\,] \bullet cs) \ . \end{aligned}$$

End of specification of function *red4*.

Function *red4* is basically a generalization of function *red3*:

$$red3 \cdot as \cdot c \ = \ (red4 \cdot as \cdot [c]) \cdot 0 \ .$$

We also have:

$$red4 \cdot [\,] \ = \ (red3 \cdot [\,] \bullet) \ . \tag{8.1}$$

This follows from the specification of *red4*.

As usual, a variant function defines the form of induction that we use in the derivation. Variant (partial) function *vfr4* has the following:

Definition of function *vfr4*

Partial function *vfr4* has type:

$$\mathcal{L}_*(\mathcal{C}) \to \mathcal{L}_*(\mathcal{C}) \to \mathcal{L}_4(\mathbb{N}) \ ,$$

and for all relevant as, cs $(cs \in \mathcal{L}_*(\mathcal{C}))$ and c, function *vfr4* has definition:

$$\begin{aligned} vfr4 \cdot as \cdot [\,] \ &= \ [0,0,0,0] \ , \\ vfr4 \cdot as \cdot (c \triangleright cs) \ &= \ vfr3 \cdot as \cdot c \ \boxplus \ \langle \boxplus i : 0 \leq i < \#cs : \ vfr3 \cdot [\,] \cdot (cs \cdot i) \ \rangle \ . \end{aligned}$$

End of definition of function *vfr4*.

From here on, dummy cs has type $cs \in \mathcal{L}_*(\mathcal{C})$.

[1] We ignore the precondition regarding predicate *hasnf*, due to the has-normal-form rule.

8.1.1 Derivation

We derive two noteworthy cases of the declaration for *red4*. The other cases are presented without proof because their derivation is straightforward. The required properties of function *vfr4* follow directly from properties of *vfr3*.

Case $c := \langle v, ss \rangle$, assuming $v \notin dom \cdot ss$, $R \cdot as$ and $R \cdot (\langle v, ss \rangle \triangleright cs)$:

$$
\begin{array}{ll}
 & red4 \cdot as \cdot (\langle v, ss \rangle \triangleright cs) \\
= & \quad \{ \text{ specification of } red4 \ \} \\
 & (red3 \cdot as \cdot \langle v, ss \rangle) \ \triangleright \ (red3 \cdot [\,] \bullet cs) \\
= & \quad \{ \text{ declaration for } red3 \ (7.3), \text{ using } v \notin dom \cdot ss, \ R \cdot as \text{ and } Q \cdot ss. \text{ The} \\
 & \quad \text{latter follows from } R \cdot (\langle v, ss \rangle \triangleright cs) \ \} \\
 & (v \ \circledcirc \ (red3 \cdot [\,] \bullet as)) \ \triangleright \ (red3 \cdot [\,] \bullet cs) \\
= & \quad \{ \ \bigstar \text{ definition of } \triangleright, \text{ see section 8.1.2 below } \} \\
 & \langle v, \#as \rangle \ \blacktriangleright \ ((red3 \cdot [\,] \bullet as) +\!\!+ (red3 \cdot [\,] \bullet cs)) \\
= & \quad \{ \text{ property of } \bullet \ \} \\
 & \langle v, \#as \rangle \ \blacktriangleright \ (red3 \cdot [\,] \bullet (as +\!\!+ cs)) \\
= & \quad \{ \text{ property of } red4 \ (8.1), \text{ by induction hypothesis, see below } \} \\
 & \langle v, \#as \rangle \ \blacktriangleright \ (red4 \cdot [\,] \cdot (as +\!\!+ cs)) \ .
\end{array}
$$

For the appeal to the induction hypothesis we need to prove the conjunction of the following, for all v, ss, as and cs:

$$
\begin{array}{c}
R \cdot as \ \wedge \ R \cdot (\langle v, ss \rangle \triangleright cs) \ \Rightarrow \ R \cdot [\,] \ \wedge \ R \cdot (as +\!\!+ cs) \ , \\
((\langle v, ss \rangle \triangleright cs) = [\,] \ \Rightarrow \ as = [\,]) \ \Rightarrow \ ((as +\!\!+ cs) = [\,] \ \Rightarrow \ [\,] = [\,]) \ , \\
vfr4 \cdot [\,] \cdot (as +\!\!+ cs) \ \trianglelefteq \ vfr4 \cdot as \cdot (\langle v, ss \rangle \triangleright cs) \ .
\end{array}
$$

The first condition follows directly from the definition of predicate R. The second condition is TRUE because its consequent is TRUE. The third condition we prove as follows:

$$
\begin{array}{ll}
 & \mathit{vfr4} \cdot \mathit{as} \cdot (\langle v, \mathit{ss} \rangle \triangleright \mathit{cs}) \\
= & \quad \{ \text{ definition of } \mathit{vfr4} \ \} \\
 & \mathit{vfr3} \cdot \mathit{as} \cdot \langle v, \mathit{ss} \rangle \ \boxplus \ \langle \boxplus\!\!\!\!\!+ \, i : 0 \leq i < \#\mathit{cs} : \ \mathit{vfr3} \cdot [\,] \cdot (\mathit{cs} \cdot i) \ \rangle \\
\sqsupseteq & \quad \{ \text{ property of } \mathit{vfr3} \ \ (7.8) \ \} \\
 & \langle \boxplus\!\!\!\!\!+ \, i : 0 \leq i < \#\mathit{as} : \ \mathit{vfr3} \cdot [\,] \cdot (\mathit{as} \cdot i) \ \rangle \\
 & \boxplus \\
 & \langle \boxplus\!\!\!\!\!+ \, i : 0 \leq i < \#\mathit{cs} : \ \mathit{vfr3} \cdot [\,] \cdot (\mathit{cs} \cdot i) \ \rangle \\
= & \quad \{ \text{ range merging } \} \\
 & \langle \boxplus\!\!\!\!\!+ \, i : 0 \leq i < \#(\mathit{as} +\!\!+ \mathit{cs}) : \ \mathit{vfr3} \cdot [\,] \cdot ((\mathit{as} +\!\!+ \mathit{cs}) \cdot i) \ \rangle \\
= & \quad \{ \text{ definition of } \mathit{vfr4} \ \} \\
 & \mathit{vfr4} \cdot [\,] \cdot (\mathit{as} +\!\!+ \mathit{cs}) \\
\sqsubseteq &
\end{array}
$$

Case $\mathit{as}, c := [\,], \langle (v \prec E), \mathit{ss} \rangle$, assuming $v \notin (\mathit{dom} \cdot \mathit{ss} \cup \mathit{fv} \cdot \mathit{ss})$ and $R \cdot (\langle v \prec E, \mathit{ss} \rangle \triangleright \mathit{cs})$:

$$
\begin{array}{ll}
 & \mathit{red4} \cdot [\,] \cdot (\langle (v \prec E), \mathit{ss} \rangle \triangleright \mathit{cs}) \\
= & \quad \{ \text{ specification of } \mathit{red4} \ \} \\
 & (\mathit{red3} \cdot [\,] \cdot \langle (v \prec E), \mathit{ss} \rangle) \ \triangleright \ (\mathit{red3} \cdot [\,] \bullet \mathit{cs}) \\
= & \quad \{ \text{ declaration for } \mathit{red3} \ \ (7.5), \text{ using } v \notin (\mathit{dom} \cdot \mathit{ss} \cup \mathit{fv} \cdot \mathit{ss}) \text{ and } Q \cdot \mathit{ss} \ \} \\
 & (v \prec (\mathit{red3} \cdot [\,] \cdot \langle E, \mathit{ss} \rangle)) \ \triangleright \ (\mathit{red3} \cdot [\,] \bullet \mathit{cs}) \\
= & \quad \{ \star \text{ definition of } \triangleright, \text{ see section 8.1.2 below } \} \\
 & v \ \blacktriangleright \ ((\mathit{red3} \cdot [\,] \cdot \langle E, \mathit{ss} \rangle) \triangleright (\mathit{red3} \cdot [\,] \bullet \mathit{cs})) \\
= & \quad \{ \text{ property of } \bullet \ \} \\
 & v \ \blacktriangleright \ (\mathit{red3} \cdot [\,] \bullet (\langle E, \mathit{ss} \rangle \triangleright \mathit{cs})) \\
= & \quad \{ \text{ specification of } \mathit{red4}, \text{ by induction hypothesis, using } R \cdot (\langle E, \mathit{ss} \rangle \triangleright \mathit{cs}) \\
 & \quad \text{and a property of } \mathit{vfr3} \ \ (7.10) \ \} \\
 & v \ \blacktriangleright \ (\mathit{red4} \cdot [\,] \cdot (\langle E, \mathit{ss} \rangle \triangleright \mathit{cs})) \ .
\end{array}
$$

8.1.2 Operator $\triangleright$

We used operator $\triangleright$ ("out") to transform an expression into a shape that allows for an appeal to the induction hypothesis. We conclude from the derivation that operator $\triangleright$ must have the following:

Definition of operator $\triangleright$

Binary operator $\triangleright$ has type (operator $\uplus$ denotes a "disjoint union"):

$$(\langle \mathcal{V}, \mathbb{N} \rangle \uplus \mathcal{V}) \times \mathcal{L}_*(\Lambda) \;\to\; \mathcal{L}_*(\Lambda)\,.$$

From here on dummy es has type $\mathcal{L}_*(\Lambda)$. Application $\langle v, n \rangle \triangleright es$ has precondition $n \leq \#es$; application $v \triangleright es$ has precondition $es \neq [\,]$. For all v, n $(n \in \mathbb{N})$, es and E, operator $\triangleright$ has definition:

$$\begin{aligned} \langle v, n \rangle \triangleright es &= (v \circledcirc es\lceil n) \triangleright es\lfloor n\,, \\ v \triangleright (E \triangleright es) &= (v \prec E) \triangleright es\,. \end{aligned}$$

Note that the overloading of operator $\triangleright$ is harmless, due to the disjoint union in its type definition.

End of definition of operator $\triangleright$.

8.1.3 Summary

Declaration for function *red4* For all relevant v, E, F, c, as, cs and ss:

$$\begin{aligned} C_0 \cdot v \cdot ss &\equiv v \notin dom \cdot ss\,, \\ C_1 \cdot v \cdot ss &\equiv v \in dom \cdot ss\,, \\ C_2 \cdot v \cdot ss &\equiv v \notin (dom \cdot ss \cup fv \cdot ss)\,, \end{aligned}$$

$$red4 \cdot [\,] \cdot [\,] = [\,] \tag{8.2}$$

$$red4 \cdot as \cdot (\langle v, ss \rangle \triangleright cs) = \langle v, \#as \rangle \triangleright (red4 \cdot [\,] \cdot (as +\!\!+ cs)) \quad \text{,if } C_0 \cdot v \cdot ss \tag{8.3}$$

$$red4 \cdot as \cdot (\langle v, ss \rangle \triangleright cs) = red4 \cdot as \cdot ((v \Uparrow ss) \triangleright cs) \quad \text{,if } C_1 \cdot v \cdot ss \tag{8.4}$$

$$red4 \cdot [\,] \cdot (\langle (v \prec E), ss \rangle \triangleright cs) = v \triangleright (red4 \cdot [\,] \cdot (\langle E, ss \rangle \triangleright cs)) \quad \text{,if } C_2 \cdot v \cdot ss \tag{8.5}$$

$$red4 \cdot (c \triangleright as) \cdot (\langle (v \prec E), ss \rangle \triangleright cs) = red4 \cdot as \cdot (\langle E, (ss \triangleleft \langle v, c \rangle) \rangle \triangleright cs) \quad \text{,if } C_2 \cdot v \cdot ss \tag{8.6}$$

$$red4 \cdot as \cdot (\langle (E \odot F), ss \rangle \triangleright cs) = red4 \cdot (\langle F, ss \rangle \triangleright as) \cdot (\langle E, ss \rangle \triangleright cs) \tag{8.7}$$

End of declaration for function *red4*.

8.2 Tail recursive reduction

We transform the linearly recursive declaration for function *red4* to a tail recursive declaration using a folded operator. Operator $\circledcirc$ ("folded out") is the right-folded version of operator $\triangleright$, hence, operator $\circledcirc$ has type $\mathcal{L}_*(\langle \mathcal{V}, \mathbb{N} \rangle \uplus \mathcal{V}) \times \mathcal{L}_*(\Lambda) \to \mathcal{L}_*(\Lambda)$. From here on, dummy os has type $\mathcal{L}_*(\langle \mathcal{V}, \mathbb{N} \rangle \uplus \mathcal{V})$.

We introduce a new (partial) function, called *red5*, with the following:

Specification of function *red5*

For all natural n, partial function *red5* has type:

$$\mathcal{L}_*(\langle \mathcal{V}, \mathbb{N} \rangle \uplus \mathcal{V}) \to \mathcal{L}_*(\mathcal{C}) \to \mathcal{L}_n(\mathcal{C}) \to \mathcal{L}_n(\Lambda) .$$

Application $red5 \cdot os \cdot as \cdot cs$ has precondition:

$$R \cdot as \wedge R \cdot cs \wedge (cs = [\,] \Rightarrow as = [\,]) .$$

For all relevant os, as and cs, function *red5* has specification:

$$red5 \cdot os \cdot as \cdot cs = os \circledcirc red4 \cdot as \cdot cs .$$

End of specification of function *red5*.

Function *red5* is a generalization of function *red4*:

$$red4 \cdot as \cdot cs = red5 \cdot [\,] \cdot as \cdot cs .$$

A straightforward derivation, using variant function *red4*, yields the following:

Declaration for function *red5* For all relevant v, E, F, c, os, as, cs and ss:

$$red5 \cdot os \cdot [\,] \cdot [\,] = os \circledcirc [\,] \tag{8.8}$$

$$red5 \cdot os \cdot as \cdot (\langle v, ss \rangle \triangleright cs) = red5 \cdot (os \triangleleft \langle v, \#as \rangle) \cdot [\,] \cdot (as +\!\!+ cs) , \quad \text{if } v \notin dom \cdot ss . \tag{8.9}$$

$$red5 \cdot os \cdot as \cdot (\langle v, ss \rangle \triangleright cs) = red5 \cdot os \cdot as \cdot ((v \Uparrow ss) \triangleright cs) , \quad \text{if } v \in dom \cdot ss . \tag{8.10}$$

$$red5 \cdot os \cdot [\,] \cdot (\langle (v \prec E), ss \rangle \triangleright cs) = red5 \cdot (os \triangleleft v) \cdot [\,] \cdot (\langle E, ss \rangle \triangleright cs) , \quad \text{if } v \notin (dom \cdot ss \cup fv \cdot ss) . \tag{8.11}$$

$$red5 \cdot os \cdot (c \triangleright as) \cdot (\langle (v \prec E), ss \rangle \triangleright cs) = red5 \cdot os \cdot as \cdot (\langle E, (ss \triangleleft \langle v, c \rangle) \rangle \triangleright cs) , \quad \text{if } v \notin (dom \cdot ss \cup fv \cdot ss) . \tag{8.12}$$

$$red5 \cdot os \cdot as \cdot (\langle (E \odot F), ss \rangle \triangleright cs) = red5 \cdot os \cdot (\langle F, ss \rangle \triangleright as) \cdot (\langle E, ss \rangle \triangleright cs) \tag{8.13}$$

And for all relevant v, n, E, os and es:

$$[\,] \circledcirc es = es \tag{8.14}$$

$$(os \triangleleft \langle v, n \rangle) \circledcirc es = os \circledcirc ((v \circledcirc es \lceil n) \triangleright es \lfloor n) \tag{8.15}$$

$$(os \triangleleft v) \circledcirc (E \triangleright es) = os \circledcirc ((v \prec E) \triangleright es) \tag{8.16}$$

End of declaration for function *red5*.

8.3 Towards an abstract machine

Consider the following application:

$$red5 \cdot os \cdot as \cdot (\langle E, ss \rangle \triangleright cs) \; .$$

If we rewrite the declaration for *red5* such that λ-term E becomes a separate parameter to the function then the resulting declaration will be better recognizable as a description for an abstract machine. We rewrite the declaration for *red5* to a declaration of a binary operator, $\oplus$ say, using the following equality, for all relevant E, os, as, cs and ss:

$$E \oplus \langle os, as, cs, ss \rangle \; = \; red5 \cdot os \cdot as \cdot (\langle E, ss \rangle \triangleright cs) \; .$$

A straightforward rewriting yields the following declaration for operator $\oplus$:

$$\begin{aligned}
v \oplus \langle os, [\,], [\,], ss \rangle &= os \circledcirc [v] \; , && (8.17)\\
&\qquad \text{if } v \notin dom \cdot ss \; . \\
v \oplus \langle os, as, cs, ss \rangle &= E \oplus \langle (os \triangleleft \langle v, \#as \rangle), [\,], bs, ts \rangle \; , && (8.18)\\
&\qquad \textbf{whr } \langle E, ts \rangle \triangleright bs \; = \; as +\!\!+ cs \; \textbf{end} \; , \\
&\qquad \text{if } v \notin dom \cdot ss \; \wedge \; (as \neq [\,] \; \vee \; cs \neq [\,]) \; . \\
v \oplus \langle os, as, cs, ss \rangle &= E \oplus \langle os, as, cs, ts \rangle \; , && (8.19)\\
&\qquad \textbf{whr } \langle E, ts \rangle \; = \; v \Uparrow ss \; \textbf{end} \; , \\
&\qquad \text{if } v \in dom \cdot ss \; . \\
(v \prec E) \oplus \langle os, [\,], cs, ss \rangle &= E \oplus \langle (os \triangleleft v), [\,], cs, ss \rangle \; , && (8.20)\\
&\qquad \text{if } v \notin (\, dom \cdot ss \cup fv \cdot ss \,) \; . \\
(v \prec E) \oplus \langle os, (c \triangleright as), cs, ss \rangle &= E \oplus \langle os, as, cs, (ss \triangleleft \langle v, c \rangle) \rangle \; , && (8.21)\\
&\qquad \text{if } v \notin (\, dom \cdot ss \cup fv \cdot ss \,) \; . \\
(E \odot F) \oplus \langle os, as, cs, ss \rangle &= E \oplus \langle os, (\langle F, ss \rangle \triangleright as), cs, ss \rangle && (8.22)
\end{aligned}$$

The λ-term on the left side of $\oplus$ can be considered as the machine "code" and the tuple on the right side of $\oplus$ can be considered as the "state" of the machine. The machine has three "instructions": variable, abstraction and application. If a more accurate description of an abstract machine is required then additional transformations are needed to, e.g., represent λ-terms by instruction sequences.

8.4 Remarks

Consider the following application:

$$red5 \cdot os \cdot as \cdot cs \; .$$

Parameter cs is a list of closures; every closure in cs will be reduced separately, as can be inferred from (among others) the declaration of *red5*. List cs is similar to what is

known in the literature (on implementing functional languages), amongst others, as the "dump" (see e.g. [Jones and Lester, 1992]) or the "control stack" (see e.g. [Plasmeijer and Eekelen, 1993]).

Note that the order in which the elements of list *cs* are reduced, follows directly from our choice of $\triangleright$ over $\triangleleft$ in the specification of function *red4* . We expect that the other case, that is, using $\triangleleft$ instead of $\triangleright$, will not yield a more efficient declaration. However, this claim was not investigated.

Chapter 9

Results and conclusions

We have derived a declaration for the normal order reduction of λ-terms from the specification of function *red0* and we have gradually transformed that declaration, in a fully calculational way, to a more efficient one that can be considered as a description for an abstract machine. The correctness of every transformation is beyond doubt, firstly, because we have proved that declarations for *red0* ... *red5* imply their respective specifications, which in turn imply the specification of *red0* . Secondly, due to the consistent use of variant functions, we have proved that the declarations for *red0* ... *red5* are well-defined.

The calculational style that we used turned out to be very suitable for this type of problem. It enabled us to improve the evaluator in small, verifiable, steps and it forced us to clearly state the design decisions that were made along the way, in the form of specifications or a strengthening in a derivation. On the other hand, the final declaration (*red5*) is still very abstract; it remains to be seen whether or not a calculational style is appropriate for the derivation of a full-fledged (efficient) functional evaluator.

9.1 Fruitless attempts

In the course of the research that resulted in this thesis, I have made attempts to further improve the efficiency of the evaluator and to make the relevant proofs more concise. Some of these attempts failed, however, two of which are noteworthy. Firstly, I tried to deal with name-clashes by representing variables by numbers using "de Bruijn indices". This transformation turned out to be quite complex and I was not able to derive a satisfactory solution. Secondly, I have proved, *in a general way*, that a folded operator inherits properties from the operator it folds. However, the application of those "inheritance properties" turned out to be more laborious than a straightforward induction proof.

9.2 Related work

There is a lot of literature on abstract machines. The machines that are designed to bridge the gap between the high level of a programming language and the low level of a real machine, like the ABC machine or the STG machine, usually have a high level of detail. Those that are designed for theoretical study, like the SECD machine [Landin, 1964] or Krivine's machine [Krivine, 2007], usually have a low level of detail. We mention two machines in particular. Although they have similar data structures as used in this thesis (stacks, closures, environments), they use a different reduction strategy.

The first example of a λ-calculus machine is the SECD machine, designed by P.J. Landing. The letters SECD stand for the elements of the state of the machine, that is: Stack, Environment, Control and Dump. It reduces λ-terms using strict evaluation, also known as eager or call-by-value evaluation, in which function arguments are evaluated before the function's body.

Krivine's machine is an abstract machine for weak head reduction, which means that the λ-terms are reduced in normal order, but the reduction stops if there is no reducible expression at the head of the λ-term. The resulting λ-term is in "weak head normal form". This form of reduction is used often for abstract machines, because it resembles the desired properties of functional program evaluation more closely. After all, if an evaluation of a functional program yields a function then this is usually considered an error.

When an abstract machine is proven correct, it is usually proven to be equivalent to Launchbury's semantics [Launchbury, 1993] or Abramsky's semantics [Abramsky, 1990], in an *a posteriori* manner. I have not been able to find another calculational proof of an abstract machine for λ-term or function evaluation.

9.3 Future work

The declarations derived in this thesis leave a lot of room for improvement. Consider the following (non-exhaustive) enumeration of "deficiencies" regarding the declaration for *red5* :

- $\diamond$ precondition $v \notin (\mathit{dom} \cdot ss \cup \mathit{fv} \cdot ss)$ to (8.11) and (8.12) is not yet satisfied, i.e., name-clashes are not yet dealt with
- $\diamond$ duplication of subterms can occur, e.g., consider the following one-step reduction:

  $$(u \prec (v \odot u \odot u)) \odot E \quad \rightarrow_\beta \quad v \odot E \odot E \; .$$

  The λ-term on the right side of $\rightarrow_\beta$ has two occurrences of E and both will be reduced separately. In the literature (on implementing functional languages) we find that this problem is often solved by exploiting the opportunities that arise from a different representation of λ-terms, namely, *graphs* instead of *trees*.

- ◇ although the declaration for *red5* can be considered as a description of an abstract machine, there is room to make the description more accurate. Anyone attempting to transform this declaration to an abstract machine might benefit from reading [Ager et al., 2003] and the more calculational [Hoogerwoord, 1994]
- ◇ It is inefficient to use λ-terms to represent so-called "basic values" (like integers) and operations on those values (like addition). In the literature (on implementing functional languages) we find that abstract machines with a high level of detail have instructions to deal with basic values and operations on those values directly.
- ◇ if a clear and concise theorem on "operator folding inheritance" can be defined then the proofs regarding folded operators can be simplified

* * *

If one is interested in an implementation of an abstract machine with such a fine grain of detail that the result can be considered as code for a Von Neumann type of machine, then the representation of the relevant datatypes (closures, environments, etc.) becomes important. This usually involves "pointers" and algorithms for memory management, including "garbage-collection" algorithms. The derivation of an implementation with such a fine grain of detail promises to be a substantial task.

Appendix A

Proof concerning chapter 5

Recall:

Lemma 5.4 For all v and as such that $hasnf \cdot (v \circledcirc as)$:

$$red1 \cdot (v \circledcirc as) = v \circledcirc (red1 \bullet as) \ .$$

End of lemma 5.4.

We prove this lemma by induction on the reduction of $v \circledcirc as$, assuming $hasnf \cdot (v \circledcirc as)$.

Case $nf \cdot (v \circledcirc as)$:

$$\begin{array}{ll} & red1 \cdot (v \circledcirc as) \\ = & \quad \{ \text{ specification of } red1 \text{ and the declaration of } red0, \text{ using } nf \cdot (v \circledcirc as) \ \} \\ & v \circledcirc as \\ = & \quad \{ \text{ specification of } red1 \text{ and the declaration of } red0, \text{ using } nf \cdot (v \circledcirc as) \\ & \quad \text{and lemma 5.2 } \} \\ & v \circledcirc (red1 \bullet as) \end{array}$$

□

Case $\neg nf \cdot (v \circledcirc as)$: at least one λ-term in list as is not in normal form. Therefore, without loss of generality, we have:

- $as \ = \ bs +\!\!+ [E] +\!\!+ cs$, and
- $\langle \forall i : 0 \leq i < \#bs : \ nf \cdot (bs \cdot i) \ \rangle$, and
- $\neg nf \cdot E$,

and we prove:

$$
\begin{array}{ll}
 & \mathit{red1} \cdot (v \circledcirc (bs +\!\!+ [E] +\!\!+ cs)) \\
= & \quad \{ \text{ specification of } \mathit{red1} \text{ and the declaration of } \mathit{red0}, \text{ using } \mathit{hasnf} \cdot (v \circledcirc as) \\
 & \quad \text{ and } \neg \mathit{nf} \cdot (v \circledcirc as) \ \} \\
 & \mathit{red1} \cdot (\mathit{rd1} \cdot (v \circledcirc (bs +\!\!+ [E] +\!\!+ cs))) \\
= & \quad \{ \text{ lemma (A.1), see below } \} \\
 & \mathit{red1} \cdot (v \circledcirc (bs +\!\!+ [\mathit{rd1} \cdot E] +\!\!+ cs)) \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & v \circledcirc (\mathit{red1} \bullet (bs +\!\!+ [\mathit{rd1} \cdot E] +\!\!+ cs)) \\
= & \quad \{ \bullet \text{ over } +\!\!+ ; \text{ definition of } \bullet \ \} \\
 & v \circledcirc ((\mathit{red1} \bullet bs) +\!\!+ [\mathit{red1} \cdot (\mathit{rd1} \cdot E)] +\!\!+ (\mathit{red1} \bullet cs)) \\
= & \quad \{ \text{ specification of } \mathit{red1} \text{ and the declaration of } \mathit{red0}, \text{ using } \mathit{hasnf} \cdot (v \circledcirc as) \\
 & \quad \text{ and property (5.13) } \} \\
 & v \circledcirc ((\mathit{red1} \bullet bs) +\!\!+ [\mathit{red1} \cdot E] +\!\!+ (\mathit{red1} \bullet cs)) \\
= & \quad \{ \text{ definition of } \bullet ; \bullet \text{ over } +\!\!+ \ \} \\
 & v \circledcirc (\mathit{red1} \bullet (bs +\!\!+ E +\!\!+ cs)) \\
= & \quad \{ \text{ we have } (bs +\!\!+ [E] +\!\!+ cs) = as \ \} \\
 & v \circledcirc (\mathit{red1} \bullet as)
\end{array}
$$

□

We used the following lemma, for all v, as, E and bs:

$$
\begin{array}{ll}
 & \langle \forall i : 0 \le i < \#as : \ \mathit{nf} \cdot (as \cdot i) \ \rangle \ \wedge \ \neg \mathit{nf} \cdot E \\
\Rightarrow & \\
 & \mathit{rd1} \cdot (v \circledcirc (as +\!\!+ [E] +\!\!+ bs)) \ = \ v \circledcirc (as +\!\!+ [\mathit{rd1} \cdot E] +\!\!+ bs) \ ,
\end{array}
\tag{A.1}
$$

which we prove (as usual) by assuming the antecedent and proving the consequent:

$$
\begin{array}{ll}
 & rd1 \cdot (v \circledcirc (as +\!\!+ [E] +\!\!+ bs)) \\
= & \quad \{ \text{ property of operator folding (2.2), twice } \} \\
 & rd1 \cdot (((v \circledcirc as) \circledcirc [E]) \circledcirc bs) \\
= & \quad \{ \text{ operator folding } \} \\
 & rd1 \cdot (((v \circledcirc as) \odot E) \circledcirc bs) \\
= & \quad \{ \text{ lemma 5.3, using } \neg nf \cdot E \ \} \\
 & rd1 \cdot ((v \circledcirc as) \odot E) \ \circledcirc \ bs \\
= & \quad \{ \text{ declaration of } rd1 \ (4.7), \text{ using } \langle \forall i : 0 \leq i < \#as : \ nf \cdot (as \cdot i) \ \rangle \text{ and lemma 5.2 } \} \\
 & ((v \circledcirc as) \odot (rd1 \cdot E)) \ \circledcirc bs \\
= & \quad \{ \text{ operator folding } \} \\
 & ((v \circledcirc as) \circledcirc [rd1 \cdot E]) \ \circledcirc bs \\
= & \quad \{ \text{ property of operator folding (2.2), twice } \} \\
 & v \circledcirc (as +\!\!+ [rd1 \cdot E] +\!\!+ bs)
\end{array}
$$

□

Bibliography

Abramsky, S. (1990). The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley.

Ager, M. S., Biernacki, D., Danvy, O., and Midtgaard, J. (2003). A functional correspondence between evaluators and abstract machines. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19. ACM.

Barendregt, H. P. (1984). *The Lambda Calculus. Its Syntax and Semantics*. North-Holland.

de la Encina, A. and Peña, R. (2009). From natural semantics to c: A formal derivation of two stg machines. *Journal of Functional Programming*, 19(01):47–94.

Dijkstra, E. W. and Scholten, C. S. (1990). *Predicate calculus and program semantics*. Springer-Verlag New York, Inc.

Hoogerwoord, R. R. (1994). rh210: The von neumann machine as a functional program.

Hoogerwoord, R. R. (2007). Programming by calculation: techniques and applications draft.

Jones, S. L. P. and Lester, D. R. (1992). *Implementing Functional Languages: a tutorial*. Prentice Hall International (UK) Ltd.

Jones, S. L. P. and Salkild, J. (1989). The spineless tagless g-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201. ACM.

Krivine, J.-L. (2007). A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207.

Landin, P. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320.

Launchbury, J. (1993). A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154. ACM.

Plasmeijer, R. and Eekelen, M. V. (1993). *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc.

Sestoft, P. (1997). Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264.